Parallelizing an LSH index with OpenMP

Michael Bartoloni

Abstract

This is a project work for parallel programming techniques. The parallelized structure is the LSH index used for similarity search across a corpus of documents. A sequential version and a parallel version of the index are created, and tested with experiments to highlight the speedup of the version. The evaluation covers the two common operations of the index, insertions and queries.

1. Introduction

Locality Sensitive Hashing is a framework for approximate similarity search. It works by using two-level hash functions to index documents with binary labels, and multiple hash tables for candidate retrieval, instead of brute-forcing similarity comparisons across all the corpus. This project implements the index, and exploits OpenMP to parallelize the execution of insertions and queries. The benchmarks focus on speedup, highlighting the performance comparison between the sequential and parallel versions, changing some hyperparameters and document sizes to see how the metrics behave. The results verify that the schema is highly compute-bound, and speedup is not influenced by the variables, but rather by just how many threads are deployed.

1.1. LSH Index

Given a query document, instead of calculating the similarity for each dataset document, the LSH index leverages a certain number of hash tables, hashing the query to buckets in the tables and retrieving all documents that collided in the same bucket as candidates to perform the similarity computation on, reducing computational load at query time and preserving a good amount of query accuracy.

Why calculating similarity on a corpus is too expensive For two sets A and B (e.g., documents represented as bag of words sets), the **Jaccard similarity** is

$$\text{Jaccard}(A, B) \;=\; \frac{|A \cap B|}{|A \cup B|}.$$

A naive approach to find all similar documents in a dataset of n items requires computing this similarity for a great amount of documents. Even if linear, for large documents and datasets this scales very badly, not to mention that most of the computation is performed on documents that are not similar to the query, while a small fraction should be very similar.

1.1.1 MinHash

Linear hash family. Fix a prime P larger than every element identifier in the universe. For random integers a, b with $0 < a < P$ and $0 \leq b < P$, define the linear hash

$$h_{a,b}(x) \;=\; (a \cdot x + b) \bmod P$$

We will use families of such linear hashes to simulate random permutations of the universe.

MinHash signature (with 1-bit reduction). Let k be the desired signature length (number of independent bits). For each signature position $i \in$

$\{1, \ldots, k\}$ we use two independent linear hash functions:

$$h_i^{(1)}(x) = (a_i x + b_i) \bmod P,$$
$$h_i^{(2)}(x) = (c_i x + d_i) \bmod P,$$

where the constants a_i, b_i, c_i, d_i are chosen uniformly at random in the appropriate ranges and independently for each i.

Given a set S, the procedure to compute the i-th bit of the signature is:

1. Compute $o_x = h_i^{(1)}(x)$ for each $x \in S$.
2. Let $x^* = \arg\min_{x \in S} o_x$.
3. Compute the single bit

$$b_i \;=\; \mathrm{LSB}\big(h_i^{(2)}(x^*)\big) \;=\; \big(h_i^{(2)}(x^*) \bmod 2\big).$$

Repeating for $i = 1, \ldots, k$ yields the k-bit signature

$$B(S) = (b_1, b_2, \ldots, b_k),$$

which we often pack into a compact integer array or bitstring (the *big signature label*).

Indexing: big signature label, bands, and tables

Let $k = r \cdot b$ for integers r (bits per band) and b (number of bands). Partition the packed signature $B(S)$ into b contiguous bands of size r bits each:

$$B(S) \;=\; \big[\, B_1(S) \mid B_2(S) \mid \ldots \mid B_b(S) \,\big],$$

where each $B_j(S)$ is an r-bit sublabel.

Maintain b hash tables (index tables) $\mathcal{T}_1, \ldots, \mathcal{T}_b$. For each band j, the table $\mathcal{T}_j$ maps B_j to a bucket by interpreting it as an integer.

This scheme turns each band into an independent randomized partitioning of the signature space; an item is inserted into exactly one bucket in each table (the bucket determined by its band key for that table). Candidates are rerieved by computing the signature of the query document and finding the colliding documents in all the tables from the buckets in which the query is hashed into. Algorithms 1, 2 summarize the insert and query operations.

Algorithm 1 Insert a document d into the LSH index

Input: document d, index tables $\mathcal{T}_1, \ldots, \mathcal{T}_b$, hash parameters, prime P, band size r, signature length k
$B(d) \leftarrow$ ComputeSignature(d)
Partition $B(d)$ into bands $B_1, \ldots, B_b$ (each r bits)
for $j = 1$ to b **do**
    Insert d into bucket $\mathcal{T}_j[B_j]$
end for

Algorithm 2 Query for neighbors of a document q

Input: query document q, index tables $\mathcal{T}_1, \ldots, \mathcal{T}_b$, same hash parameters, P, r, k
$B(q) \leftarrow$ ComputeSignature(q)
Partition $B(q)$ into bands $B_1^q, \ldots, B_b^q$
$C \leftarrow \varnothing$ ▷ candidate set
for $j = 1$ to b **do**
    $C \leftarrow C \cup \mathcal{T}_j[B_j^q]$
end for
for each candidate $u \in C$ **do**
    $s(u) \leftarrow$ ComputeSimilarity(q, u)
end for
Return top candidates ranked by $s(\cdot)$

2. Data Structures and Implementation Overview

This section describes the main data structures used to implement the LSH-based index, together with key design choices for document representation and MinHash signature computation.

2.1. Document Representations

Two representations are used for documents:

Dense representation.

- `DenseDoc` stores a document as a fixed-size array `words` of length `DOC_SIZE`.
- Each position contains a word identifier.
- This format is convenient for initial loading or positional data, but not efficient for set-based similarity.

Set representation.

- `SetDoc` stores only the distinct word identifiers that appear in the document.
- The array `words` is sorted and contains no duplicates.
- A sentinel value `SET_END` marks the end of the array.
- The field `count` stores the number of unique words.

This set-based representation is used for both MinHash signature computation and Jaccard similarity.

2.2. MinHash Parameters and Signature Computation

The structure `MinHashParams` stores the parameters of the hash family used for MinHash:

- Four arrays `a_params`, `b_params`, `c_params`, `d_params`, of length `num_funcs` define two families of linear hash functions

$$h_i(x) = (a_i x + b_i) \bmod P,$$

$$h'_i(x) = (c_i x + d_i) \bmod P,$$

  where P is a large prime.
- The method `compute_signature` computes the MinHash signature of a document, producing a bit array of length `num_funcs`.

SIMD acceleration. Signature computation is vectorized using AVX2 intrinsics. This significantly speeds up the MinHash computation, which is the most computationally intensive part of index construction and querying, and is also well suited for vectorization. Below a sketch of the procedure:

1. **Init:** set every slot of the `signature` buffer to the Mersenne prime $P = 2^{31} - 1$ (vector stores for the bulk, scalar loop for the tail).
2. **For each word in the document:**
   (a) Broadcast the word to a SIMD register so the same word can be combined with many hash parameters at once.
   (b) Load blocks of 8 hash parameters (a_j, b_j) into SIMD registers.
   (c) Compute vectorized linear hashes $h_j = a_j \cdot \text{word} + b_j$ (one SIMD multiply + add per block).
   (d) Apply a fast Mersenne-prime reduction in SIMD: mask low 31 bits and add high bits, then correct overflow with a conditional subtract (all vector operations).
   (e) Update the per-hash minima with a vectorized unsigned minimum between the new h_v and the current `signature` block; store the result.
   (f) Handle any remaining (num_funcs mod 8) lanes with a scalar loop.
3. **Projection (second hash + LSB):**
   (a) Load blocks of second-hash parameters (c_j, d_j) and the per-hash minima (vector loads).
   (b) Compute $h'_j = c_j \cdot \min_j + d_j$, apply the same Mersenne reduction, then extract the LSB with a vector bitwise-and.
   (c) Store the 0/1 results back into the signature buffer (vector stores). Finish remaining lanes scalar.
4. **Result:** the signature buffer contains one bit per hash (0/1).

SIMD intrinsics used (brief list and purpose)

`_mm256_set1_epi32(x)`
create a vector with all lanes = `x` (used for broadcasting word, masks, prime, constants).

`_mm256_storeu_si256(...)`
write 8 lanes to memory (used for bulk initialization and storing updated signature blocks).

`_mm256_loadu_si256(...)`
load 8 lanes from memory (used for parameter blocks and current signature values).

`_mm256_mullo_epi32(a,b)`
element-wise 32-bit integer multiply (computes $a_j \cdot$ word for 8 lanes).

`_mm256_add_epi32(a,b)`
element-wise 32-bit add (adds the b_j offsets and implements the Mersenne reduction step).

`_mm256_and_si256(a,b)`
bitwise-and (used to mask low 31 bits and to extract the LSB).

`_mm256_srli_epi32(a,imm)`
logical right shift (extract high-bit contribution for the Mersenne reduction).

`_mm256_cmpgt_epi32(a,b)`
compare greater-than (produces a mask for overflow detection).

`_mm256_sub_epi32(a,b)`
element-wise subtraction (used with the overflow mask to subtract P where needed).

`_mm256_min_epu32(a,b)`
unsigned minimum across lanes (updates per-hash minima branchlessly).

2.3. Buckets and Tables

Bucket nodes. Each bucket in an LSH table is implemented as a linked list of `BucketNode`:

- Each node stores a document identifier `doc_id`.
- The pointer `next` links to the next node in the same bucket.

This linked list design allows multiple documents that hash to the same bucket to be stored efficiently and reduces unnecessary memory allocation.

LSH tables. An `LSHTable` contains:

- An array `buckets` of pointers to `BucketNode`, one per bucket.
- The number of buckets `num_buckets`, typically a power of two.

Each table corresponds to one band of the MinHash signature. Documents whose band hashes collide are inserted into the same bucket of that table.

2.4. LSH Index Structure

The overall index is represented by `LSHIndex`:

- An array of pointers to `LSHTable`, one per band.
- A pointer to `MinHashParams`, shared across all tables.

Memory pool for bucket nodes Instead of dynamically allocating each node, all bucket nodes are preallocated in a single large array. This reduces allocation overhead in parallelized environments.

Per-query timestamping This technique consists of pre-allocating one array of length `max_docs` for each thread. At query time, for each query document, an atomic increment of a global counter yields a unique query identifier q_id for each query. This identifier marks each array location (indexed by the document id) every time a new document is found, so that we avoid duplicate candidates. This way each thread can efficiently store all the candidates without duplicates when scanning different tables. Atomic incrementing ensures q_id uniqueness across concurrent queries, so timestamp arrays do not need to be reset between queries, reducing memory management overhead.

2.5. Jaccard Similarity

For final verification, Jaccard similarity between two `SetDoc` instances is computed by a linear merge-like scan with a double pointer method over their sorted word arrays:

- The intersection size is counted by advancing pointers similarly to a merge step.
- The union size is derived from the set sizes and the intersection.

This takes time proportional to the total number of unique words in the two documents.

2.6. Other Multithreading settings and precautions

1. Dividing the work among the threads is straightforward, and a dynamic schedule is the best bet for this setting: insertions and queries work on sets of words, and candidate retrieval can bring out a different number of candidates for different queries, so a static scheduling may penalize certain threads. So both the for loops across the corpus for the insert and query (+ candidate similarity computation) are parallelized with barriers, leaving the implicit barrier after the insertion to query a fully populated index.

2. Inserting documents in the tables buckets is the part requiring additional synchronization: when a thread has calculated the bucket position in a certain table, race conditions may happen. Instead of locking the table, it's sufficient to lock the designated bucket with a dedicated omp lock, preventing insertion from other threads. Also, the risk of collisions for an appropriate band size is minimal, so it's rare that a thread has to wait a long time to insert. As tables become larger in size, there's a tradeoff between performance and memory, but it shouldn't be a problem if the buckets array is not too big.

3. To avoid false sharing, BucketNode structures and the locks are padded (with a custom structure).

3. Data and Experiments Setup

3.1. Generating the dataset

Data is generated artificially. The distribution of words in each document is designed to emulate natural token-frequency patterns: token occurrences follow a heavy-tailed, Zipf-like distribution with a small head of common tokens and a long tail of rare tokens. Common tokens create a shared background across documents while rare tokens provide the discriminative signal needed for similarity detection.

3.2. Scaling variables and other settings

Each experiment features 100000 documents for both insertions and queries. Configurations variate certain variables to evaluate the scaling of the algorithms on different settings. For instance, the variables are the size of the documents, the length of each signature's band (i.e. length of the signature), the number of tables. Each setting variates a single variable while keeping the other fixed, to effectively measure the impact that each one has on the execution time.

Average execution times are recorded for 10 runs of both the sequential version and for the parallel version of the benchmarks, and the speedup trend for each variable is computed for an evaluation on how the algorithms scale also with the number of threads.

The machine running the benchmarks has 6 cores with 12 threads, so multithreading is tested on an increasing even number of threads, from 2 to 20.

3.3. Results

The results of the sequential benchmark [Table 1] show that queries have the heaviest impact on computation time. Since both insert and query have the signature computation, it must be Jaccard that drags down performance.

We can analyze the impact on insertions and queries for each variable:

- **Document size.** Larger documents contain more tokens to process: MinHash computation costs grow roughly linearly with the number of words (even if SIMD greatly amortizes per-word cost), and Jaccard's computation takes longer because the sets grow larger. Hence both insertion and query times rise with document size, the query cost is more sensitive.

- **Number of tables.** Insertions cost grows nearly linearly with the number of tables (the signature grows since we set the band number per table); queries must probe more buckets

Scenario	Doc Size	Band	Tables	Ins. Time (s)	Qry. Time (s)
Size	200	20	5	0.3316	1.7341
Size	500	20	5	0.7060	3.7877
Size	1000	20	5	1.2134	7.9133
Size	3000	20	5	2.9934	29.0594
Tables	500	20	5	0.6687	3.8413
Tables	500	20	10	1.0783	6.6104
Tables	500	20	15	1.8080	9.1137
Band	500	20	5	0.7318	3.8302
Band	500	25	5	0.8676	3.9245
Band	500	30	5	1.0385	4.1852

Table 1. Sequential benchmark results on the various LSH scenarios.

and typically produce a larger raw candidate list, increasing the workload.

- **Band width (bits per band).** Band width trades off bucket collision rate vs. per-band work: wider bands reduce collision probability (fewer false candidates and less Jaccard verification) but each band-hash combines more signature entries (slightly higher per-band hashing cost).

Figures 1, 2, 3 show the speedup for the various scenarios, varying the number of threads. As they show, the speedup rate is sublinear, but scales evenly with the growing number of threads for both operations. Also, little to no performance loss is caused by spawning more than 12 threads, meaning the overhead is almost has little impact on the execution. Smaller sized documents cause less speedup with respect to using larger documents. This is probably due to the fact that for small documents, the insertion time is already very low, so the multithreading overhead is a larger component in the execution time, and spawning more threads provides little (but noticeable) benefits.

Query speedup is higher than the insertion speedup (Tables 2, 3, 4 for reference), this is to be expected, since there's no synchronization and the scheduling overhead is overshadowed by the heavier computing of the task, resulting in almost linear speedup.

4. Conclusions

This project provided some insights on a parallel implementation of an LSH index for set-based document similarity that combines a MinHash-based 1-bit signature, AVX2-accelerated signature construction, and OpenMP parallelism for both insertions and queries.

Empirical results show that SIMD vectorization substantially reduces the cost of signature construction, shifting the dominant cost of queries to candidate verification (exact Jaccard similarity). As a consequence, tuning the band width and the number of tables to reduce false-positive candidates has a larger impact on end-to-end query time than further micro-optimizations of the hashing kernel. Parallelization yields near-linear speedup for query-heavy workloads; insertion throughput improves with threads but shows sublinear scaling when bucket synchronization and memory contention become relevant.

The chosen timestamping scheme avoids global seen-sets and per-bucket scanning while remaining simple and cache-friendly, at the expense of $O(max\_docs)$ storage per-thread memory. The node-pool approach minimizes allocation overhead and improves locality, but requires provisioning capacity and does not yet reclaim nodes dynamically. These trade-offs—memory for speed and simplicity for predictability—are appropriate for large, in-memory experiments and controlled benchmarks.

Threads	Insertion Speedup				Query Speedup			
	200	**500**	**1000**	**3000**	**200**	**500**	**1000**	**3000**
2	1.16	1.17	1.12	1.07	1.69	1.66	1.64	2.00
4	2.00	2.20	2.20	2.14	3.28	3.31	3.31	3.61
6	2.75	2.85	3.25	3.18	4.56	4.97	4.91	5.34
8	3.65	3.30	4.39	4.30	5.93	6.55	6.66	7.67
10	4.02	3.50	5.42	5.36	7.52	7.89	8.38	9.04
12	4.05	3.79	5.97	5.92	8.60	9.07	9.49	10.91
14	4.07	3.87	6.11	5.90	8.52	8.98	9.40	10.41
16	4.03	3.82	6.02	5.90	8.62	9.03	9.35	9.61
18	4.04	3.86	6.05	5.89	8.62	8.84	9.23	11.03
20	4.05	3.79	5.91	5.83	8.47	8.92	9.78	11.10

Table 2. Parallel speedup: document size speedup

Threads	Insertion Speedup			Query Speedup		
	5	**10**	**15**	**5**	**10**	**15**
2	1.09	1.07	1.18	1.61	1.66	1.68
4	2.13	1.94	2.16	3.29	3.27	3.32
6	3.00	2.78	3.03	4.81	4.82	4.96
8	4.21	3.65	4.02	6.42	6.23	6.45
10	4.87	4.50	4.96	7.99	7.80	7.91
12	5.41	5.14	5.61	9.17	8.56	8.69
14	5.48	5.15	5.67	9.01	8.83	9.00
16	5.51	5.08	5.76	9.26	8.78	9.05
18	5.46	5.13	5.55	9.09	8.75	8.90
20	5.34	5.00	5.49	8.99	8.87	8.66

Table 3. Parallel speedup: number of tables speedup

Threads	Insertion Speedup			Query Speedup		
	20	**25**	**30**	**20**	**25**	**30**
2	1.15	1.20	1.17	1.64	1.59	1.66
4	2.24	2.33	2.28	3.29	3.17	3.22
6	3.14	3.32	3.30	4.93	4.74	4.83
8	4.38	4.54	4.51	6.53	6.26	6.35
10	4.95	5.09	5.13	7.95	7.55	7.92
12	5.36	5.92	5.97	8.72	8.77	9.04
14	5.55	5.95	6.07	8.95	8.75	9.05
16	5.52	5.93	6.01	8.73	8.67	9.07
18	5.40	5.92	6.01	8.82	8.68	8.90
20	5.47	5.93	5.91	8.76	8.66	8.83

Table 4. Parallel speedup: band size speedup

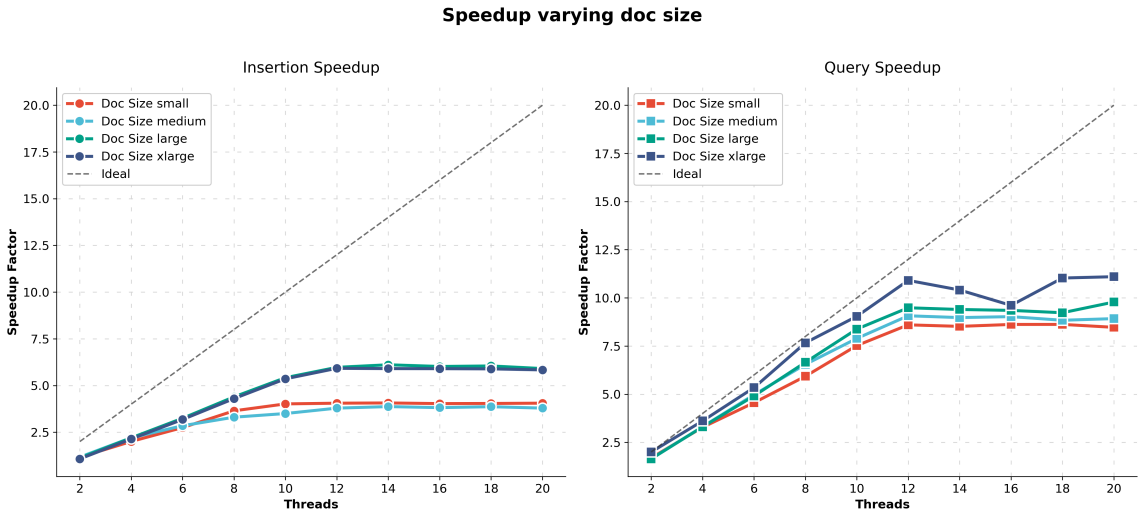

Figure 1. Plot of the execution times with growing number of threads

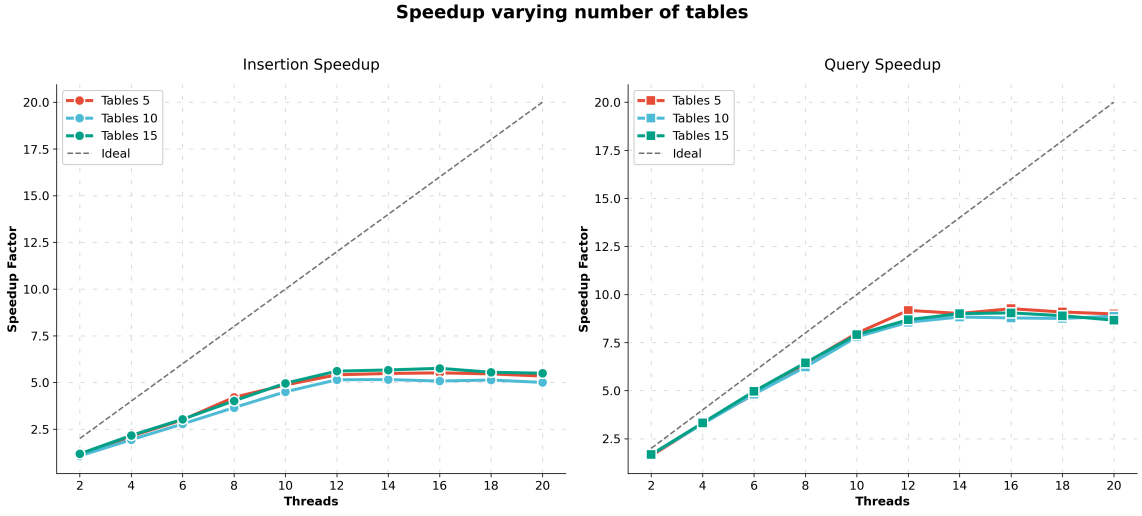

Figure 2. Plot of the execution times with growing number of threads

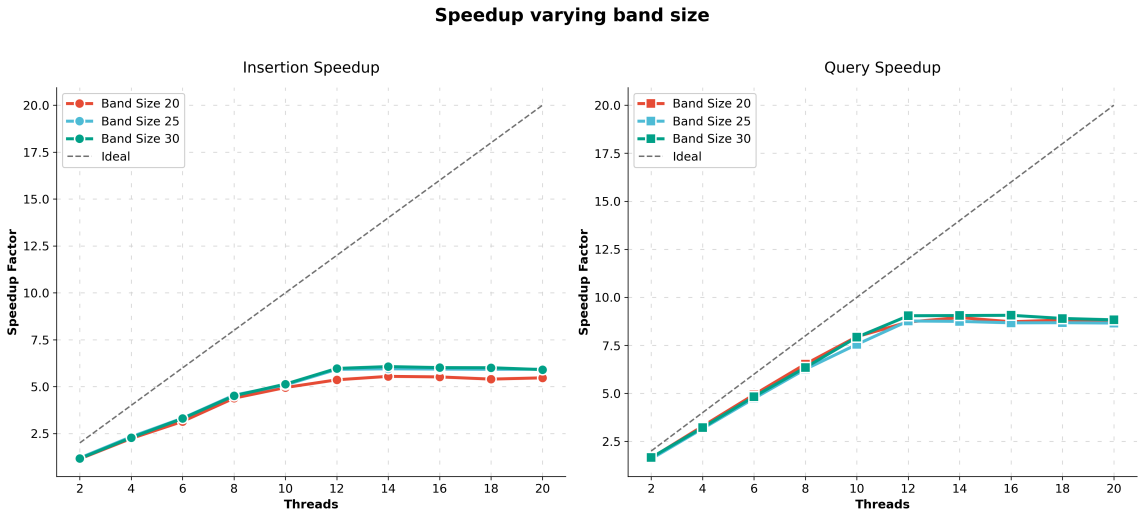

Figure 3. Plot of the execution times with growing number of threads