

PCHPC Project Report

MPI-Based Exact Dense Vector Search for RAG Pipelines

Darius Heere

Supervisor: Claas Kochanke

Requested ECTS: 6

Georg-August-Universität Göttingen
Institute of Computer Science

September 27, 2026

Abstract

Dense vector retrieval is a central computational component of Retrieval-Augmented Generation (RAG), where queries are matched against vector representations of candidate documents. Because exact exhaustive retrieval evaluates every indexed vector, its computational cost grows with index size and vector dimensionality. This work investigates whether Message Passing Interface (MPI)-based index sharding can reduce retrieval latency while preserving exact top- k correctness and keeping global result consolidation inexpensive.

The implemented MPI framework partitions the index across ranks, performs exact local search on each shard using NumPy or Facebook AI Similarity Search (FAISS) and reduces the local candidates to an exact global top- k using centralized or tree-based merging. Correctness is validated against a sequential full-index NumPy reference. Using reproducible synthetic dense vectors on the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) cluster, the evaluation covers hot-retrieval scaling, global reduction behavior, session-start performance and weak scaling, with hot retrieval measured separately from loading and index construction.

Correctness validation confirmed that the distributed retrieval preserves the reference top- k result. For the main 500,000-vector workload, median hot-retrieval latency with centralized reduction decreases from approximately 490 ms with one rank to 20.3 ms with 32 ranks, corresponding to a speedup of $24.1\times$. Tree-based reduction shows comparable scaling, while the global top- k reduction process generally remains a small part of the overall retrieval time for the main $k = 10$ workload. Neither reduction strategy is consistently faster across the tested configurations. No substantial multi-node placement penalty is observed in the tested range, while session-start measurements identify vector loading as the main initialization bottleneck. Weak scaling supports larger distributed indexes but shows decreasing efficiency at higher rank counts. Overall, MPI sharding effectively reduces exact retrieval latency, although the exhaustive scan remains the fundamental computational limitation.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☒ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☒ For the development of software source texts
- ☒ For optimizing or restructuring software source texts
- ☒ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.

Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables	iv
List of Figures	v
List of Abbreviations	vi
1 Introduction	1
2 Methodology	2
2.1 Exact dense top- k retrieval	2
2.2 Sequential formulation and parallelization approach	2
2.3 Exact global top- k reduction	3
3 Implementation	4
3.1 Data generation, storage and shard-aware loading	4
3.2 Rank-local search backends	5
3.3 MPI implementations of the global top- k reduction	5
3.4 Correctness validation	6
3.5 Benchmark infrastructure and cluster execution	6
4 Performance analysis and evaluation	7
4.1 Evaluation setup and measurement protocol	7
4.2 Fixed-index hot-retrieval performance	8
4.2.1 Single-node strong scaling	8
4.2.2 Multi-node scaling	9
4.3 Global merge performance and profiling	10
4.3.1 Merge-stress performance	10
4.3.2 Score-P/Vampir analysis and tree refinement	12
4.4 Scalability beyond the core workload	13
4.4.1 Session-start performance	13
4.4.2 Weak scaling	14
5 Challenges and discussion	16
5.1 Interpretation of performance and scalability	16
5.2 Methodological and implementation challenges	16
5.3 Limitations	17
5.4 Future work	17
6 Conclusion	18
References	19
A Additional benchmarks	A1
A.1 Dot product vs. cosine similarity	A1
A.2 FAISS vs. NumPy backend comparison	A2
A.3 MPI ranks vs. FAISS threads	A3
B Individual contribution	A4

List of Tables

1	Overview of the main performance experiments.	8
---	---	---

List of Figures

1	Hot-retrieval strong-scaling results for a fixed 500k-vector index.	9
2	Hot-retrieval multi-node scaling results for a fixed 500k-vector index. . . .	10
3	Mean global merge time under increasing top- k candidate counts.	11
4	Score-P/Vampir analysis of the original tree-based merge implementation. .	12
5	Median phase breakdown of session-start performance for the fixed 500k- vector index.	14
6	Weak-scaling efficiency for exact retrieval with 100k vectors per rank. . . .	15
7	Speedup comparison for dot-product retrieval and cosine similarity imple- mented via normalized inner-product search.	A1
8	Hot-retrieval latency comparison between FAISS <code>IndexFlatIP</code> and NumPy local search.	A2
9	Hot-retrieval latency for different MPI-rank and FAISS-thread allocations under a fixed 16-core budget.	A3

List of Abbreviations

ANN Approximate Nearest Neighbor

CPU Central Processing Unit

FAISS Facebook AI Similarity Search

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HNSW Hierarchical Navigable Small World

MPI Message Passing Interface

RAG Retrieval-Augmented Generation

1 Introduction

Retrieval-augmented generation (RAG) augments a generative model with information retrieved from an external knowledge source, allowing generation to condition on retrieved evidence in addition to the model’s parametric knowledge [Lew+20]. A common retrieval method represents queries and candidate documents as dense vectors in a shared embedding space and ranks candidates by vector similarity [Lew+20]. In this work, similarity is computed by inner product, making retrieval a search over a dense vector index. In the exact dense vector search considered in this work, the query is compared with every indexed vector, so the scoring cost grows with the index size and vector dimensionality.

Dense vector search can be performed using exact or approximate methods: exact search considers the complete index, whereas approximate methods reduce the search space with the risk of omitting vectors that belong to the true top- k result [JDJ19]. To avoid this approximation trade-off and retain an exact reference result, this work focuses on exact search while studying how its computational cost can be parallelized. Because the similarity computations for different indexed vectors are independent, the index can be partitioned across MPI ranks so that each rank searches only a fraction of it.

Against this background, the project addresses the research question: *How effectively can exact dense vector retrieval be parallelized through MPI-based index sharding while preserving exact top- k correctness and limiting communication overhead?* The evaluation addresses the following objectives:

- Quantify retrieval scalability as MPI resources are increased within one node and across multiple nodes.
- Determine when global top- k reduction becomes relevant and compare the centralized and tree-based approaches.
- Evaluate session-start and weak-scaling behavior beyond the initialized fixed-index workload.

This report contributes an MPI-based framework for exact dense vector retrieval¹ together with an empirical evaluation of its performance on the GWDG cluster. The framework combines index sharding across MPI ranks, exact global top- k reduction, sequential full-index correctness validation and benchmark infrastructure for cluster execution. The study is limited to the computational retrieval kernel and therefore does not evaluate semantic retrieval quality or end-to-end RAG performance. The evaluation uses reproducible synthetic dense vectors to isolate the computational behavior of the retrieval workload. The results show that sharding substantially reduces retrieval latency, while global reduction remains a comparatively small cost for the main workload and initialization exposes vector loading as a separate bottleneck.

After this introduction, Chapter 2 formulates the retrieval problem and its MPI-based parallelization. Chapter 3 describes the resulting implementation and validation approach. Chapter 4 then presents the experimental evaluation, followed by the interpretation of the findings, limitations and future work in Chapter 5. Chapter 6 closes the report by relating the results back to the research question and evaluation objectives.

¹MPI-Based Exact Dense Vector Search for RAG Pipelines. URL: <https://github.com/darius-heere-uni/pchpc-project>.

2 Methodology

This chapter defines the methodology for formulating and parallelizing exact dense top- k retrieval. It introduces the retrieval problem and sequential formulation, then motivates its decomposition across MPI ranks and describes the exact global top- k reduction using centralized and tree-based approaches. The central requirement is that parallelization must not alter the retrieval result: for a given query and index, the parallel implementation must preserve the exact global top- k result of the sequential reference. Performance improvements are only valid if this requirement is satisfied.

2.1 Exact dense top- k retrieval

Dense retrieval represents queries and indexed items as vectors in a shared d -dimensional space and ranks the indexed vectors according to their similarity to a query [Kar+20]. Let $q \in \mathbb{R}^d$ denote a query vector and $I = \{x_1, \dots, x_N\}$ an index containing N vectors. In this work, similarity is measured by the inner product $s_i = q^\top x_i$, following the dot-product formulation used for dense passage retrieval [Kar+20].

Exact top- k retrieval consists of two conceptual steps. First, a similarity score is computed for every indexed vector. Second, a k -selection step identifies the k entries with the highest scores, yielding $R_k = \text{topk}(s_1, \dots, s_N)$. This separation between exhaustive scoring and selection is also used in exact similarity-search formulations [JDJ19]. Each result entry consists of a global vector identifier and its corresponding similarity score.

Exact search considers the complete index before selecting the result, whereas approximate methods may restrict the search to a subset of the indexed vectors [JDJ19]. The inner-product formulation also covers cosine similarity for L2-normalized vectors, since cosine similarity is equivalent to inner product for unit vectors [Kar+20]. Exact retrieval therefore provides a reference top- k result without an additional approximation trade-off and forms the computational problem considered in the following sections.

2.2 Sequential formulation and parallelization approach

In the sequential approach, a single process evaluates the query against the complete vector index and subsequently performs the top- k selection. For one query, this requires N independent inner products over d dimensions, corresponding to a scoring cost of $O(Nd)$. As the index grows, the computational workload and memory required for the complete index remain concentrated in a single process.

The scoring phase is well suited to parallelization because the score $s_i = q^\top x_i$ of one indexed vector is independent of the scores of all other vectors. The index can therefore be partitioned into disjoint subsets whose union covers the complete index. This preserves exact exhaustive retrieval: parallelization changes where scores are computed without removing any indexed vector from consideration. With P MPI ranks and an approximately balanced partition, each rank evaluates about N/P vectors, reducing the local scoring work to approximately $O(Nd/P)$ before the partial results are combined.

The parallelization in this work is applied across the vector index for a single query. All ranks therefore cooperate on the same retrieval request rather than processing different

queries independently. Each rank applies the same exact scoring and top- k selection procedure to its local subset, producing a local candidate set. This decomposition targets the latency of one retrieval operation and distinguishes the approach from query-level parallelism, which would primarily increase throughput without reducing the work required for an individual query.

The local candidate sets are not yet sufficient as the final retrieval result because the globally highest-scoring vectors may be distributed across several partitions. A further reduction step is therefore required to combine the local results into one exact global top- k . The structure of this distributed retrieval and the conditions required for the global reduction to remain exact are described in the following section.

2.3 Exact global top- k reduction

The global reduction combines the rank-local candidate sets into the final exact global top- k result. Here, top- k denotes the k candidates with the highest similarity scores. Each rank therefore retains its local top- k after the local search. Keeping k candidates per rank is necessary because all k globally highest-scoring vectors could theoretically reside in a single shard.

Any vector in the global top- k must also be part of the local top- k of its owning shard. Otherwise, at least k vectors in that shard would already have a higher score. The exact global result can therefore be written as

$$R_k^{\text{global}} = \text{topk} \left(\bigsqcup_{p=0}^{P-1} R_{k,p}^{\text{local}} \right). \quad (1)$$

Here, $R_{k,p}^{\text{local}}$ denotes the local top- k candidate set of rank p . Since the shards are disjoint, their candidate sets contain at most Pk distinct vector identifiers. Selecting the top- k from these candidates therefore produces the same result as selecting directly from the complete index.

Each candidate is represented by its similarity score and identifier in the original global index. The dense vectors themselves are no longer required once the local search has completed and are therefore not transferred during the global reduction. Communication consequently depends on the number of retained candidates rather than on the total number or dimensionality of the indexed vectors.

Within this project, the global reduction is organized in two alternative ways. In the centralized approach, all local candidate sets are collected at a root rank, which selects the final top- k from the combined candidates. This concentrates communication and the final selection at one rank. In the tree-based approach, candidate sets are merged pairwise, and only the best k candidates of each intermediate result are forwarded to the next stage until the final result reaches the root rank. For a balanced tree, the number of reduction stages grows logarithmically with P . Both reduction strategies operate on the same local top- k candidate sets produced by the rank-local searches. They differ only in how these candidates are merged to obtain the global top- k result. The implementation and evaluation therefore compare centralized and tree-based reduction as two procedures for merging the same distributed candidate sets.

3 Implementation

This chapter describes the concrete implementation of the exact dense retrieval methodology introduced in Chapter 2. The project implements one distributed retrieval framework in a single codebase, launched across one or more MPI ranks. Its three main stages are shard-aware loading, rank-local search using NumPy or FAISS and global top- k reduction using the centralized or tree-based implementation. The configuration selects the search backend and reduction implementation, and the framework dispatches to the corresponding components at runtime. The same program can therefore execute different combinations within the same retrieval workflow. The chapter also describes the sequential full-index correctness validation and benchmark infrastructure used on the GWDG cluster.

3.1 Data generation, storage and shard-aware loading

The retrieval framework is evaluated in Section 4 using reproducible synthetic datasets consisting of a global vector matrix and a separate query pool. Vectors and queries are generated as dense `float32` arrays with a fixed random seed and stored persistently as `vectors.npy` and `queries.npy`, together with a metadata file containing the dataset parameters. The datasets can be reused across benchmark executions. The `.npy` format² preserves array shape and data type information. The random vectors reproduce the dimensionality and dense arithmetic of the retrieval workload, but do not model semantic retrieval quality.

The query array acts as a pool for repeated one-query retrievals. A different query row is selected for every warmup and measured retrieval, while each invocation processes exactly one query. The main latency configuration with five warmups and 50 measurements therefore stores 55 query vectors. The query pool is loaded by every rank because all ranks cooperate on the same retrieval request and the query data is small compared with the vector index.

The global vector index is stored once and partitioned logically when the MPI program starts. For N vectors and P ranks, the index is divided into balanced contiguous intervals. If N is not divisible by P , the first $N \bmod P$ ranks receive one additional vector. Each shard's start position is retained so that local row indices can be converted to global vector identifiers. This allows the same dataset to be reused for different MPI world sizes without creating rank-specific files. Each rank opens `vectors.npy` using memory-mapped NumPy loading³ and copies only its assigned slice into local memory. The copy allows the local search to operate on a contiguous in-memory array rather than repeatedly accessing the mapped file. Since the vector shards and query data are loaded directly from shared storage, no MPI communication is required for data distribution.

²*NumPy binary format*. URL: <https://numpy.org/doc/stable/reference/generated/numpy.lib.format.html> (visited on Aug. 26, 2026).

³*numpy.load*. URL: <https://numpy.org/doc/stable/reference/generated/numpy.load.html> (visited on Aug. 26, 2026).

3.2 Rank-local search backends

Within the MPI retrieval framework, rank-local search forms a separate stage between shard loading and global result reduction. Each rank uses the configured backend to search its local shard and return similarity scores and row indices for the local top- k candidates. Two exact search backends are supported: NumPy, which directly implements the exhaustive inner-product search from Section 2.1, and FAISS, a specialized vector-search library. This allows the framework to use either a direct implementation of the methodology or a widely used vector-search library for the same exact local-search task.

The NumPy backend performs the local search by computing the dense inner products between the query vectors and the local vector matrix. The k highest-scoring entries are selected using `numpy.argmaxpartition`⁴ and the selected candidates are subsequently sorted in descending score order. Since the score computation considers every vector in the local shard, the NumPy backend implements an exact exhaustive search.

The FAISS backend uses the exact `IndexFlatIP` index from the FAISS vector-similarity search library⁵, which performs exhaustive inner-product search. Local vectors and queries are provided as contiguous `float32` arrays, and the configured thread count is applied to the local FAISS search. After shard loading, the local FAISS index is constructed and populated once for reuse across retrievals. The implementation therefore remains exact and does not use an approximate FAISS index. Both backends return the same local top- k candidate representation, so the global reduction described in Section 3.3 can process their results in the same way.

3.3 MPI implementations of the global top- k reduction

The MPI communication layer implements the two global reduction approaches introduced in Section 2.3 using `mpi4py`⁶. After rank-local search, each rank holds its local top- k scores and corresponding global vector identifiers. These candidate sets are combined using either the centralized reduction implementation (`mpi centralized`) or the tree-based reduction implementation (`mpi tree`). Both implementations compute the exact global top- k defined in Equation 1, but use different communication patterns.

In the centralized implementation, candidate scores and identifiers are gathered separately at rank 0 using blocking `comm.gather()` operations. Rank 0 then combines the received candidates and selects the global top- k , concentrating communication and final selection at the root.

The tree-based implementation distributes the reduction across the ranks using pairwise point-to-point communication. At each stage, receiving ranks merge their current candidates with those received from a partner, retain the best k entries and continue to the next stage until rank 0 holds the exact global top- k result. Since each intermediate result is truncated to k candidates, later messages remain bounded to k scores and identifiers. Candidate arrays are sent with lowercase `comm.send()` and `comm.recv()` operations.

⁴*numpy.argmaxpartition*. URL: <https://numpy.org/doc/stable/reference/generated/numpy.argmaxpartition.html> (visited on Aug. 26, 2026).

⁵*Faiss indexes*. URL: <https://github.com/facebookresearch/faiss/wiki/Faiss-indexes> (visited on Aug. 26, 2026).

⁶*MPI for Python Documentation*. URL: <https://mpi4py.readthedocs.io/en/stable/> (visited on Aug. 26, 2026).

3.4 Correctness validation

For correctness validation, the same repository contains a separate sequential NumPy reference path that operates on the complete vector index. The reference is used only for correctness validation and is not part of the configurable retrieval or benchmark workflow. To validate a result produced by the distributed retrieval framework, it loads the same configuration, complete vector index and query on rank 0 and performs exact NumPy retrieval over the full index. This path does not use shard-aware loading, global MPI reduction or the benchmark logic for warmups, repeated measurements, timing aggregation and result collection. The sequential reference uses the same NumPy search routine as the NumPy rank-local backend, but applies it directly to the complete index rather than to distributed shards. It therefore provides a reference result for validating the distributed sharding and reduction workflow independently of the benchmark execution logic.

For this comparison, the global vector identifiers returned by the sequential reference must match those of the distributed retrieval framework exactly. The corresponding similarity scores are compared using `numpy.allclose`⁷ with absolute and relative tolerances⁸ of 10^{-5} , allowing small floating-point differences while still verifying that the distributed retrieval produces the same top- k result as the sequential full-index reference.

3.5 Benchmark infrastructure and cluster execution

The benchmark infrastructure is integrated into the retrieval framework codebase. On each MPI rank, a benchmark driver reads the JSON configuration and controls the retrieval workflow. It executes the configured shard loading, rank-local search and MPI global reduction. The configuration specifies the dataset, search backend, reduction implementation, top- k value, thread settings, warmups and measurement counts. On the GWDG cluster, benchmark jobs are submitted through Slurm, with the requested task count determining the number of MPI ranks. Within the allocation, the project environment is activated and `mpiexec` launches the program.

Shard loading and local search initialization are performed once before the repeated retrievals. For the FAISS backend, each rank constructs and populates its local `IndexFlatIP` once, while the NumPy backend retains the local vector matrix. Warmup and measured retrievals then reuse this initialized state, separating repeated retrieval from data loading and, for FAISS, index construction. Rank 0 collects the timing information and summarizes the measured runs. The exact synchronization and timing boundaries used for the final experiments are described in Section 4.1.

After each benchmark execution, rank 0 writes structured result files containing aggregate metrics, individual measured runs and the configuration. Unique run directories prevent reruns from overwriting earlier measurements. Slurm allocation metadata is retained for later inspection of individual runs. A separate collection script converts the stored results into flattened CSV for analysis.

⁷`numpy.allclose`. URL: <https://numpy.org/doc/stable/reference/generated/numpy.allclose.html> (visited on Aug. 26, 2026).

⁸Absolute and relative tolerances are arguments of the `numpy.allclose` function. The absolute tolerance bounds errors near zero, while the relative tolerance scales the permitted deviation with the magnitude of the reference value.

4 Performance analysis and evaluation

This chapter evaluates the performance and scalability of the MPI-based exact retrieval implementation. It first defines the experimental environment and measurement protocol, then evaluates fixed-index hot retrieval, global top- k reduction behavior and scalability beyond the initialized core workload.

4.1 Evaluation setup and measurement protocol

The performance experiments were executed on the `scc-cpu` partition of the GWDG Scientific Compute Cluster. Each node is equipped with two Intel Xeon Platinum 8468 processors, providing 96 physical Central Processing Unit (CPU) cores and 192 hardware threads, with approximately 515 GB of memory. Unless stated otherwise, one Slurm CPU was allocated per MPI rank and the local FAISS search used one thread per rank. The generated datasets were stored persistently on the shared CephFS project storage of the GWDG Ceph HDD system under `$PROJECT_DIR` and accessed directly by the participating MPI ranks. During benchmark initialization, each rank loads its assigned slice of the shared vector file into memory and constructs the corresponding local FAISS index.

Benchmark execution distinguishes three phases: rank-local vector loading, rank-local index construction and distributed retrieval. Vector loading and index construction are timed independently on each rank and summarized separately. The aggregation used for the session-start analysis is defined in Section 4.4.1. Distributed retrieval is synchronized with `MPI_Barrier` before timing begins and is measured until the final global top- k result is available on rank 0. Runtime measurements use `MPI.Wtime`, which provides wall-clock timestamps in seconds relative to an arbitrary reference point.

For the main fixed-index workload, the index contains 500,000 `float32` vectors with dimension $d = 2560$, approximately 4.8 GiB of raw vector data. The weak-scaling experiment keeps 100,000 vectors per rank and increases the global index from 100,000 to 1.6 million vectors, reaching approximately 15.3 GiB at the largest configuration. These cardinalities lie within the range of established retrieval benchmarks such as BEIR [Tha+21], while the dimensionality is also used by dense embedding models such as Qwen3-Embedding-4B [Zha+25]. Unless varied by an experiment, retrieval uses exact inner-product search with FAISS `IndexFlatIP` and $k = 10$.

In this report, hot retrieval refers to repeated retrieval from an already initialized in-memory state. This means that the rank-local vector shards are loaded into memory and the local FAISS indexes are constructed there before retrieval timing begins. Each rank initializes this state once, followed by five warmup retrievals and 50 measured retrievals. A different query vector is used for each warmup and measured iteration, while every retrieval invocation processes exactly one query. The initialized state is reused throughout these iterations, so vector loading and index construction are excluded from the reported hot-retrieval latency. Median latency over the 50 measured retrievals is reported.

Table 1 summarizes the main experiment families. Experiments that modify the hot-retrieval protocol or use derived metrics define the corresponding measurement in their respective subsections.

Experiment	Workload	Varied configuration	Reported metric
Single-node strong scaling	500k vectors	1–32 MPI ranks on one node	Median latency and speedup
Multi-node scaling	500k vectors	1–8 nodes, four ranks per node	Median latency and same-rank factor
Merge stress	500k vectors	$k = 10$ –5000, reduction type and placement	Mean global merge time
Session start	500k vectors	1–16 MPI ranks on one node	Median derived session-start time
Weak scaling	100k vectors per rank	1–16 MPI ranks, 100k–1.6M vectors	Median latency and weak-scaling efficiency

Table 1: Overview of the main performance experiments.

All performance measurements, including configurations with a single MPI rank, use the MPI benchmark framework described in Sections 3.2, 3.3 and 3.5. Unless stated otherwise, the main experiments use FAISS for rank-local search and evaluate both the centralized and tree-based global reduction implementations. The sequential full-index NumPy reference path is used exclusively for correctness validation (see Section 3.4). Score-P/Vampir is used qualitatively for selected communication traces in Section 4.3.2, while all quantitative performance comparisons are based on untraced benchmark runs.

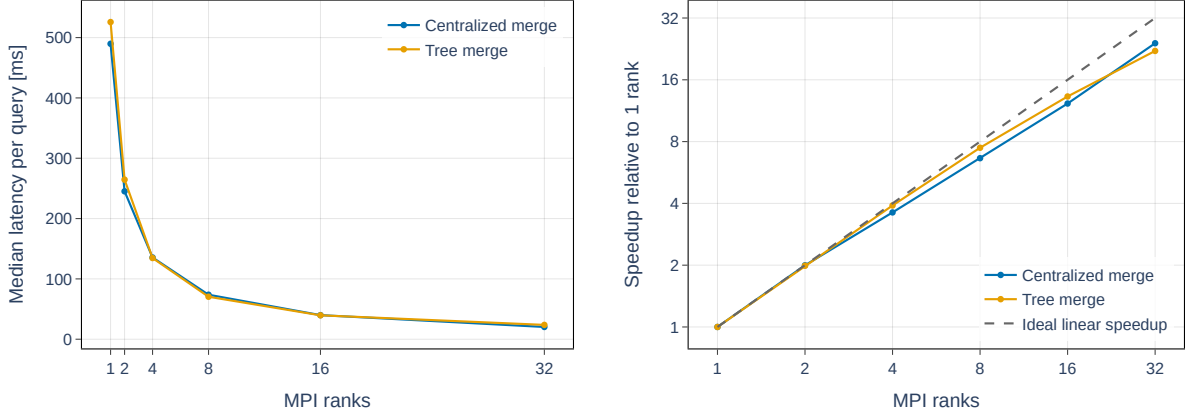
4.2 Fixed-index hot-retrieval performance

This section evaluates the core hot-retrieval workload after vector loading and local index construction have completed. It first examines strong scaling on a single node as the fixed retrieval problem is distributed across more MPI ranks, then compares equivalent rank counts across multiple nodes to assess the effect of node placement.

4.2.1 Single-node strong scaling

The single-node strong-scaling results show that MPI sharding substantially reduces hot-retrieval latency as the fixed workload is distributed from 1 to 32 ranks, while the centralized-merge and tree-merge configurations exhibit similar scaling behavior. The experiment distributes the fixed 500,000-vector workload across increasing numbers of ranks on one compute node. Figure 1 reports the median retrieval latency and corresponding speedup. For each merge configuration, speedup is calculated as $S(P) = T_1/T_P$, where T_1 is the median latency of its one-rank configuration in the same MPI benchmark path and T_P the median latency with P ranks.

For the centralized merge, median latency decreases from approximately 490 ms with one rank to 20.3 ms with 32 ranks, corresponding to a speedup of about $24.1\times$. The tree merge decreases from approximately 526 ms to 23.8 ms over the same range, giving a speedup of about $22.1\times$. Both merge configurations remain close to ideal speedup at lower rank counts, while the gap to linear scaling grows at higher rank counts. The two latency curves remain close throughout the experiment, with no consistent performance advantage for either merge configuration.



(a) Median hot-retrieval latency.

(b) Speedup relative to the one-rank MPI baseline. Log-scaled axes.

The benchmark uses a fixed index of 500k vectors with dimension 2560 and retrieves the top- k results with $k = 10$ for one query at a time. Strong scaling is evaluated from 1 to 32 MPI ranks on a single node with a 32 GB memory allocation, using 1 CPU and 1 FAISS thread per rank. Reported values are medians of 50 measured retrievals after 5 warmup runs. Across the plotted configurations, the coefficient of variation of retrieval latency ranges from 6.0% to 38.1%.

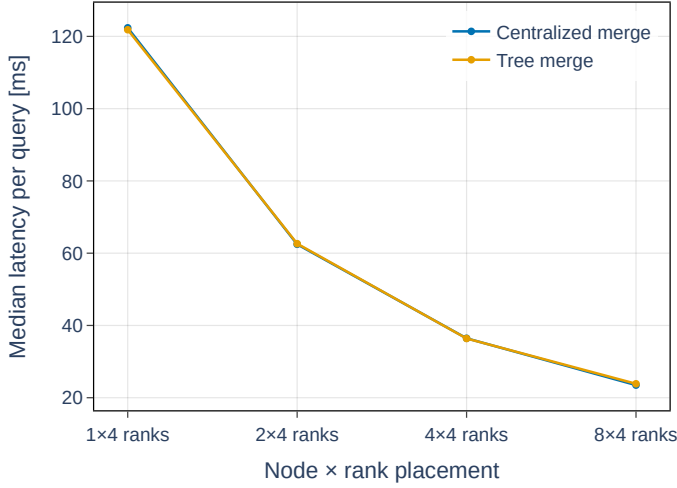
Figure 1: Hot-retrieval strong-scaling results for a fixed 500k-vector index.

The observed scaling follows the index decomposition described in Section 2.2: increasing the number of ranks reduces the number of vectors searched by each rank while the global problem size remains fixed. At $k = 10$, each rank contributes only a small candidate set to the global reduction, so the choice of merge configuration has little influence on the overall retrieval latency. The remaining deviation from ideal speedup at higher rank counts shows that the reduction in local search work is not translated proportionally into end-to-end latency reduction. Since the coefficient of variation ranges from 6.0% to 38.1% across the configurations, the small differences between the two merge configurations should not be interpreted as a systematic performance advantage. The contribution of the global reduction itself is examined separately in Section 4.3.

4.2.2 Multi-node scaling

The latency reduction observed on a single node largely persists when the fixed workload is distributed across multiple nodes, while the same-rank comparison shows no large placement penalty. The benchmark uses the same fixed 500,000-vector workload as Section 4.2.1, with four MPI ranks per node and configurations ranging from 1 node $\times$ 4 ranks to 8 nodes $\times$ 4 ranks. Figure 2 combines the resulting latency scaling with a same-rank comparison, in which each multi-node latency is divided by the latency measured with the same total number of ranks on a single node.

Across the tested placements, median latency decreases from approximately 122 ms at 1 node $\times$ 4 ranks to about 24 ms at 8 nodes $\times$ 4 ranks. The centralized-merge and tree-merge configurations again produce almost identical median latencies across the full range. In the same-rank comparison, the factors lie between 0.85 and 1.16 and are mostly close to one. This indicates that distributing ranks across multiple nodes does not substantially change latency relative to the corresponding single-node configuration.



(a) Multi-node hot-retrieval latency scaling.

Ranks	Placement	Cent. Tree	
4	1 node $\times$ 4 ranks	0.90	0.91
8	2 nodes $\times$ 4 ranks	0.85	0.89
16	4 nodes $\times$ 4 ranks	0.91	0.92
32	8 nodes $\times$ 4 ranks	1.16	1.00

$$\text{Factor} = \frac{\text{multi-node latency}}{\text{same-rank single-node latency}^*}$$

* Same-rank single-node measurements are taken from the strong-scaling benchmark in Section 4.2.1.

(b) Latency factor relative to the same-rank single-node configuration.

The benchmark uses a fixed index of 500k vectors with dimension 2560 and retrieves the top- k results with $k = 10$ for one query at a time. Multi-node scaling is evaluated with 4 MPI ranks per node over 1, 2, 4 and 8 nodes, corresponding to 4, 8, 16 and 32 total ranks, with a 16 GB memory allocation per node, using 1 CPU and 1 FAISS thread per rank. Reported values are medians of 50 measured retrievals after 5 warmup runs. Across the plotted configurations, the coefficient of variation of retrieval latency ranges from 0.3% to 43.1%.

Figure 2: Hot-retrieval multi-node scaling results for a fixed 500k-vector index.

The latency trend therefore extends the single-node scaling result across node boundaries. For the effect of rank placement itself, the same-rank factors provide the relevant comparison by normalizing each multi-node measurement to a single-node run with the same total rank count. Their values remain close to one, indicating only modest placement-related differences in the tested configurations. The median latencies of the centralized and tree merge configurations overlap almost completely across the tested placements. This close agreement in the reported medians occurs despite substantial variability in some configurations, with coefficients of variation ranging from 0.3% to 43.1%. Values below one in the same-rank comparison are therefore treated as measurement variability rather than as evidence that multi-node placement is intrinsically faster.

4.3 Global merge performance and profiling

This section examines the global top- k reduction separately from the overall retrieval latency. It first evaluates how the observed merge cost changes as the number of retained candidates increases and compares the centralized-merge and tree-merge configurations across single- and multi-node placements. It then uses Score-P/Vampir to inspect the communication behavior of the original tree implementation and discusses the explicit-buffer refinement derived from the profiling results.

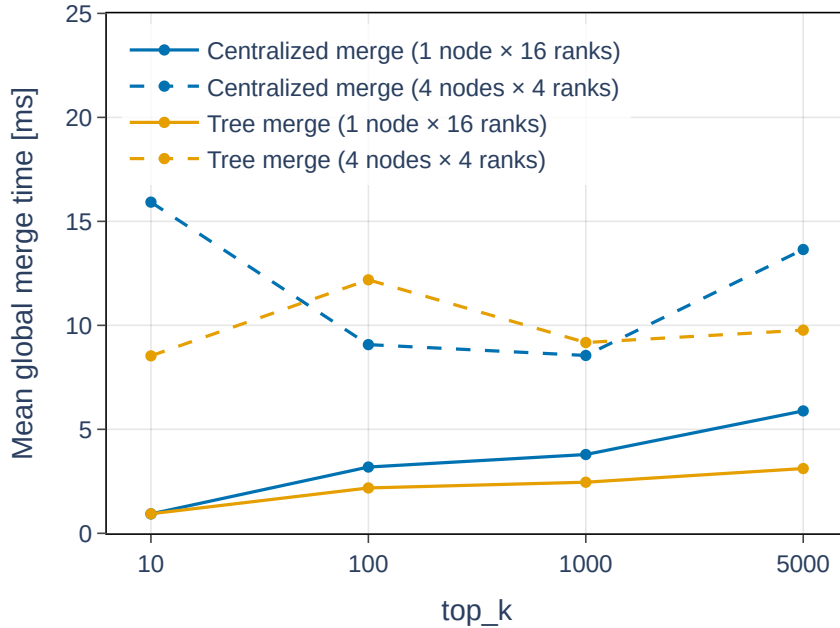
4.3.1 Merge-stress performance

Global merge performance is isolated by increasing the number of retained candidates while keeping the global index size and total rank count fixed. The benchmark uses

500,000 vectors and 16 MPI ranks while increasing k from 10 to 5000. Larger values of k increase the score and identifier candidate sets that must be communicated and merged without changing the underlying exhaustive search problem. Figure 3 compares the centralized-merge and tree-merge configurations for 1 node $\times$ 16 ranks and 4 nodes $\times$ 4 ranks.

The reported metric is the mean global merge time. For centralized reduction, this is a combined metric constructed from two measured parts. The first part is the candidate-gather time. For each measured retrieval, the gather duration is recorded on all ranks and the maximum value is retained. These maximum values are then averaged over the 50 measured retrievals. The second part is the mean time required by rank 0 to merge the received candidates and produce the global top- k result.

For tree-based reduction, the measured value already corresponds to the complete tree-reduction wall time and therefore includes communication, waiting and intermediate merge operations along the reduction path. Means are used because the centralized value is reconstructed from summary statistics: the sum of the means yields the mean of the combined phases, whereas the corresponding medians cannot in general be combined in this way.



The benchmark uses a fixed index of 500k vectors with dimension 2560 and processes one query at a time. Global merge performance is evaluated with 16 MPI ranks using 1 node $\times$ 16 ranks with a 32 GB memory allocation and 4 nodes $\times$ 4 ranks with a 16 GB memory allocation per node, using 1 CPU and 1 FAISS thread per rank while increasing k from 10 to 5000. Reported values are means over 50 measured retrievals after 5 warmup runs. For centralized merging, global merge time includes candidate gathering and the subsequent rank-0 merge. For tree merging, it corresponds to the complete tree reduction.

Figure 3: Mean global merge time under increasing top- k candidate counts.

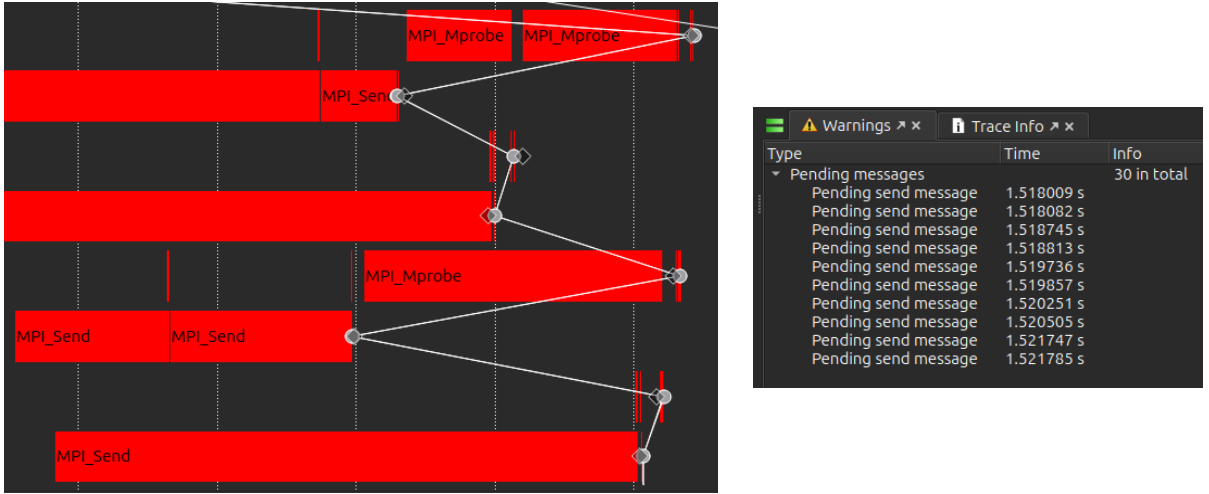
For the single-node placement, both merge configurations require approximately 0.9 ms at $k = 10$. As k increases to 5000, mean global merge time rises to approximately 5.9 ms for the centralized merge and 3.1 ms for the tree merge. The multi-node measurements are

higher overall and show a less regular dependence on k , ranging from approximately 8.5 ms to 15.9 ms across the tested configurations. Neither reduction strategy is consistently faster across all tested values of k and both placements.

At the default $k = 10$, the single-node merge time of approximately 1 ms is small compared with the roughly 40 ms end-to-end hot-retrieval latency observed at 16 ranks in Section 4.2.1. This is consistent with the close centralized-merge and tree-merge latency curves in the main strong-scaling experiment. Increasing k makes the reduction cost more visible, particularly across nodes, but the multi-node measurements show a non-monotonic dependence on the candidate count. The results therefore demonstrate that merge cost depends on both candidate volume and placement, without establishing a generally superior reduction strategy.

4.3.2 Score-P/Vampir analysis and tree refinement

An early version of the merge-stress plot shown in the previous subsection showed substantially higher merge times for the 4 nodes $\times$ 4 ranks tree configuration than for the other tested configurations. This unexpected result motivated tracing the same configuration with Score-P/Vampir. The trace did not identify an MPI communication effect that explained the full discrepancy. A later review of the benchmark instrumentation showed that the centralized and tree timers covered different parts of the global reduction, leading to the corrected metric used in Section 4.3.1. The trace remained useful because it exposed the Python-object communication path of the original tree implementation and motivated the explicit-buffer variant examined below.



(a) Zoomed Vampir trace of the tree-merge communication phase. (b) Vampir warning showing 30 pending send messages.

The Score-P trace captures one retrieval with the original `mpi_tree` implementation using 16 MPI ranks distributed as 4 ranks per node across 4 nodes, with a 16 GB memory allocation per node and 1 CPU per rank. The run uses `top_k=5000`, no warmup and one measured retrieval.

Figure 4: Score-P/Vampir analysis of the original tree-based merge implementation.

The Score-P/Vampir run was configured to trace MPI activity while excluding Python-level execution from instrumentation. FAISS could not be loaded in the dedicated Conda environment used for the Score-P runs because the required Intel oneMKL component `libmkl_intel_lp64.so.2` was missing, so the NumPy search backend was used instead.

This backend substitution does not change the MPI merge implementation or communication structure examined here. The trace is used qualitatively to inspect the MPI events within the tree-reduction region, while quantitative performance comparisons are based on untraced benchmark runs.

Figure 4 shows outgoing communication as `MPI_Send`, while the corresponding receive-side activity appears through `MPI_Mprobe`. Vampir additionally reports 30 pending send messages. This number corresponds to the communication structure of the reduction. With 16 ranks, a complete tree reduction requires 15 logical rank-to-rank exchanges. Since the implementation transfers similarity scores and vector identifiers as separate messages, these exchanges produce 30 candidate messages in total.

The receive-side behavior follows from the communication mechanism used by the original tree implementation. As described in Section 3.3, it uses the lowercase `mpi4py` calls `comm.send()` and `comm.recv()`, which transfer the candidate arrays through `mpi4py`'s Python-object communication path. To address this traceability limitation in the original tree implementation, an additional merge variant, `mpi_tree_explicit`, was introduced. It retains the same reduction topology and exact top- k algorithm but changes how the candidate data is transferred. Scores are stored in contiguous `float32` arrays and identifiers in contiguous `int64` arrays, with explicitly allocated receive buffers and the uppercase `comm.Send()` and `comm.Recv()` operations.

A follow-up trace of the explicit-buffer implementation showed direct communication with matching `MPI_Send` and `MPI_Recv` operations, while the pending-message warnings were not present. The `mpi_tree_explicit` implementation was then evaluated in a separate merge-stress benchmark using the same increasing- k experiment for both the 1 node $\times$ 16 ranks and 4 nodes $\times$ 4 ranks placements. Across both placements, its mean global merge times remained in the same range as the centralized and original tree implementations, with no consistent ordering across the tested values of k . The refinement therefore improved MPI traceability without producing a consistent performance change.

4.4 Scalability beyond the core workload

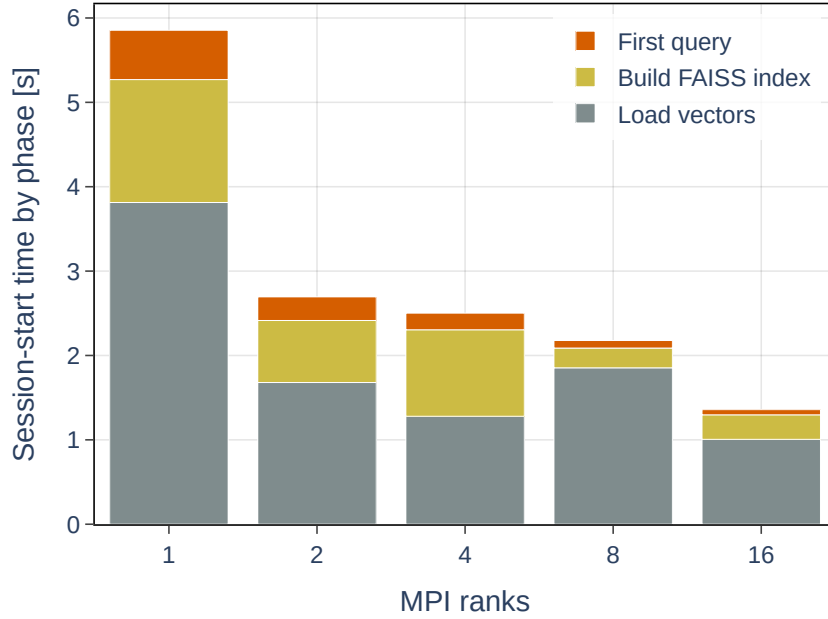
The preceding experiments focus on hot retrieval with initialized local indexes and, for the main scaling results, a fixed global problem size. This section extends the evaluation in two directions: session-start performance includes vector loading and local FAISS index construction before the first query, whereas weak scaling increases the global index together with the number of MPI ranks.

4.4.1 Session-start performance

The session-start results show that the approximate time until the first query result is available decreases substantially as the fixed 500,000-vector index is distributed across more MPI ranks. Unlike the hot-retrieval experiments, this benchmark includes the work required before retrieval can begin and uses the centralized-merge configuration. Each sample is obtained from a fresh Slurm job and consists of vector loading, local FAISS index construction and one measured retrieval without warmup. Five independent jobs are executed for each rank count so that loading and index construction are repeated for every sample rather than reused across measurements.

The session-start time is derived as $T_{\text{session start}} = T_{\text{load,mean}} + T_{\text{build,mean}} + T_{\text{first retrieval}}$ for each job. The loading and index-construction terms use the mean rank-local duration reported for the respective phase, while the retrieval term corresponds to the single measured query. The median is then reported across the five independent Slurm jobs for each rank count.

Derived session-start time decreases from approximately 5.8s with one MPI rank to about 1.3s with 16 ranks. The phase breakdown in Figure 5 shows that vector loading accounts for the largest part of the total time across most rank counts, while local FAISS index construction contributes a smaller fraction. The first retrieval represents only a small part of the session-start time.



The benchmark uses a fixed index of 500k vectors with dimension 2560 and retrieves the top- k results with $k = 10$ for one query at a time. Session-start scaling is evaluated from 1 to 16 MPI ranks on a single node with a 32 GB memory allocation, using 1 CPU and 1 FAISS thread per rank. Each sample is obtained from a fresh Slurm job with no warmup and consists of vector loading, local FAISS index construction and the first retrieval using the centralized-merge configuration. Five independent jobs are executed for each rank count, resulting in 25 independent Slurm jobs in total. Each stacked component reports the median duration of that phase across the five repetitions.

Figure 5: Median phase breakdown of session-start performance for the fixed 500k-vector index.

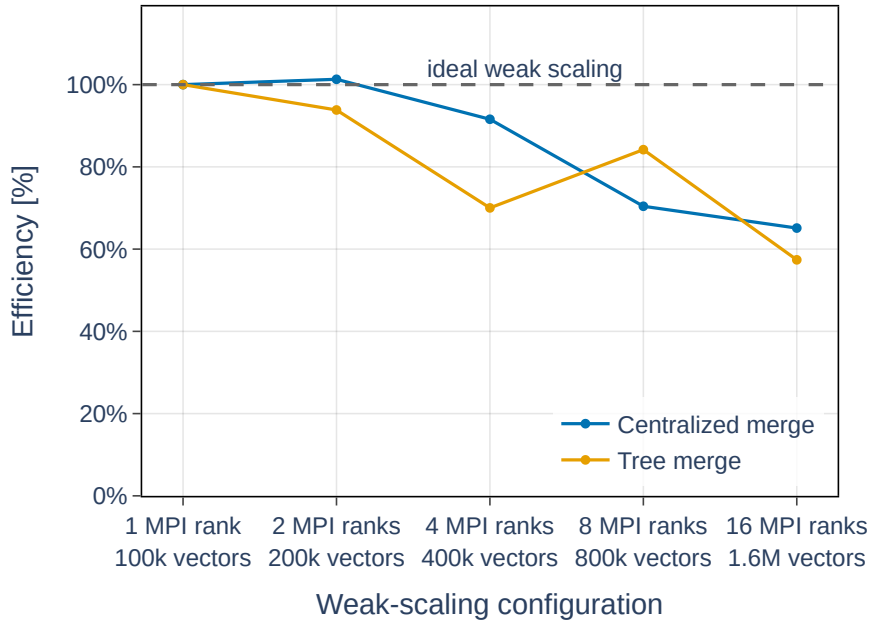
The session-start benchmark therefore exposes a different bottleneck from hot retrieval: initialization, and particularly vector loading, dominates the derived session-start time.

4.4.2 Weak scaling

Weak scaling evaluates retrieval performance as the global index grows while the local search workload remains fixed. The benchmark keeps the local shard size at 100,000 vectors per rank while increasing the number of MPI ranks from 1 to 16. The global index therefore grows from 100,000 to 1.6 million vectors, corresponding to approximately 0.95–15.3 GiB of raw vector data. Weak-scaling efficiency is calculated as $E_{\text{weak}}(P) = T_1/T_P$,

where T_1 is the median retrieval latency of the one-rank configuration for the same merge configuration and T_P the median latency with P ranks. Ideal weak scaling therefore corresponds to an efficiency of 100%, since the number of vectors searched by each rank remains fixed.

At 16 MPI ranks, weak-scaling efficiency is approximately 65% for the centralized merge and 57% for the tree merge. The intermediate measurements are non-monotonic, with the centralized-merge configuration slightly above 100% at two ranks and the tree-merge configuration decreasing at four ranks before increasing again at eight ranks. Across the plotted configurations, the coefficient of variation of retrieval latency ranges from 7.7% to 18.1%. These deviations are therefore not assigned separate algorithmic meaning.



The benchmark uses a fixed local shard size of 100k vectors per rank with dimension 2560 and retrieves the top- k results with $k = 10$ for one query at a time. Weak scaling is evaluated for 1, 2, 4, 8 and 16 MPI ranks on a single node, increasing the global index size from 100k to 1.6M vectors, for both centralized and tree-based merge configurations, using 1 CPU and 1 FAISS thread per rank. The memory allocation is 8 GB for 1 and 2 ranks, 16 GB for 4 ranks and 32 GB for 8 and 16 ranks. Reported values are derived from medians of 50 measured retrievals after 5 warmup runs. Across the plotted configurations, the coefficient of variation of retrieval latency ranges from 7.7% to 18.1%.

Figure 6: Weak-scaling efficiency for exact retrieval with 100k vectors per rank.

The weak-scaling results demonstrate that the MPI retrieval framework is capable of handling substantially larger index sizes, reaching 1.6 million vectors and approximately 15.3 GiB of raw vector data in the tested configuration. This provides evidence that the approach remains scalable as the indexed dataset grows, despite the observed loss in weak-scaling efficiency.

5 Challenges and discussion

This chapter discusses the implications of the performance results presented in Chapter 4 in relation to the research question and evaluation objectives, as well as the retrieval methodology and implementation introduced in Chapters 2 and 3.

5.1 Interpretation of performance and scalability

The results in Chapter 4 show that index sharding is effective for reducing the latency of exact dense retrieval. The strong-scaling results in Section 4.2.1 confirm that reducing the local search work through the decomposition in Section 2.2 translates into substantial latency improvements, although scaling becomes less efficient at higher rank counts. For the default $k = 10$ workload, the global top- k reduction remains small compared with the local search, as reflected by the similar centralized and tree-based retrieval latencies and the low merge times in Section 4.3.1. Increasing k makes the reduction cost more visible, but neither reduction strategy shows a consistent performance advantage.

Beyond single-node strong scaling, the experiments demonstrate that the framework also extends across node boundaries and to larger indexes. The multi-node results in Section 4.2.2 preserve the scaling behavior across up to eight nodes without a substantial placement penalty for the tested configurations. In the weak-scaling experiment in Section 4.4.2, additional ranks increase the global index from 100,000 to 1.6 million vectors while keeping the local workload fixed. The configurations lie within the corpus-size range represented in BEIR [Tha+21]. Although weak-scaling efficiency decreases as the index grows, the experiment shows that exact retrieval can be extended to larger indexes by distributing the index across additional resources.

The session-start measurements in Section 4.4.1 expose a different bottleneck from initialized retrieval. Although distributing the index across more ranks also reduces session-start time, vector loading remains the largest component across most configurations, while index construction and the first retrieval contribute less. For the current storage design, startup is therefore dominated by loading the rank-local slices from the shared vector file.

5.2 Methodological and implementation challenges

Reliable performance comparison required equivalent timing boundaries across the reduction implementations. The centralized and tree-based merge timers initially covered different parts of the global reduction, so their raw values could not support a direct performance comparison. The final metric in Section 4.3.1 therefore covers the complete global reduction for both implementations: candidate gathering and rank-0 merging for the centralized implementation and the full tree reduction for the tree-based implementation. Run-to-run variability in some cluster configurations additionally made it difficult to distinguish small implementation effects from measurement noise, so isolated deviations were not treated as systematic.

As discussed in Section 4.3.2, a further implementation challenge was that `mpi4py` object communication obscured the MPI operations underlying the original tree reduction. The resulting `MPI_Mprobe` activity and pending-message warnings made the communication trace harder to interpret. Replacing this path with explicit typed buffers and `Send/Recv`

produced a clearer correspondence between the implementation and the observed MPI operations. Since the measured merge performance remained in the same range, the main benefit of this refinement was improved traceability rather than lower runtime.

5.3 Limitations

The project isolates the dense retrieval kernel of a RAG pipeline, so the reported timings do not represent end-to-end RAG latency. The use of reproducible random vectors further excludes the statistical and semantic properties of embeddings generated by a specific model and therefore does not evaluate semantic retrieval quality.

The performance results are further limited to exact flat inner-product search and the tested index sizes, rank counts, node placements and vector dimensionality, so the observed scaling and merge behavior should not be generalized beyond these configurations without further evaluation. The measurements were performed on the shared GWDG cluster, where concurrent activity can introduce run-to-run variability. Since the benchmark jobs did not request exclusive node allocation, this source of measurement noise could not be fully controlled. For the session-start experiments, each sample begins in a fresh Slurm job, but the scheduler-provided execution environment and underlying system state may differ between runs. The measurements therefore characterize session-start performance under scheduled cluster conditions rather than from a standardized initial system state. Since all ranks also load their assigned slices from the same shared vector file, the loading results cannot be generalized directly to other storage layouts.

5.4 Future work

One direct extension concerns the storage layout used during initialization. As shown in Section 4.4.1, vector loading is the largest component of session-start time across most tested configurations, while all ranks currently read their assigned ranges from the same shared vector file. A future implementation could instead use physical shard files so that each rank loads its assigned shard directly. This could reduce concurrent file access and potentially lower loading overhead, although physical sharding would reduce the flexibility of repartitioning one global index across different rank counts.

A second direction is to extend the framework beyond exact flat search. Exact retrieval remains exhaustive even when distributed across MPI ranks, whereas Approximate Nearest Neighbor (ANN) methods such as Hierarchical Navigable Small World (HNSW) graphs [MY20] can avoid an exhaustive scan at the cost of approximate results. Since the local search backend is separated from the MPI retrieval logic, a FAISS HNSW backend could replace the exact local search while retaining the existing MPI sharding and candidate-merging structure. The exact implementation would then provide a controlled reference for measuring how much latency is reduced and how much of the exact top- k result is retained. The quality difference can be quantified against the exact top- k result using

$$\text{Recall}@k = \frac{|R_k^{\text{ANN}} \cap R_k^{\text{exact}}|}{k}.$$

The resulting comparison would show how much approximate local search can reduce retrieval latency for larger distributed indexes and what loss in exact top- k recall accompanies that reduction.

6 Conclusion

This project investigated how effectively exact dense vector retrieval can be parallelized through MPI-based index sharding while preserving exact top- k correctness and limiting communication overhead. The implemented framework partitions the vector index across MPI ranks, performs exhaustive local search on each shard and combines the resulting candidates through an exact global top- k reduction. Comparison with a sequential full-index reference confirmed that the distributed implementation produces the same reference top- k result.

The evaluation objectives were addressed by examining scalability, global reduction behavior and performance beyond the initialized fixed-index workload. For the first objective, the strong-scaling experiments showed that distributing the index across additional MPI ranks substantially reduces hot-retrieval latency, with the scaling behavior extending across multiple nodes without a substantial placement penalty in the tested configurations. For the second objective, global top- k reduction remained small relative to local search at the default $k = 10$. Increasing k made communication and merging more visible, but neither centralized nor tree-based reduction was consistently faster across the tested configurations. Score-P/Vampir analysis further clarified the communication behavior of the tree reduction and motivated a more directly traceable MPI implementation. For the third objective, weak scaling showed that the framework can operate on larger distributed indexes, although efficiency decreased at higher rank counts, while the session-start experiments identified vector loading as the main initialization bottleneck.

The project objectives were therefore achieved within the evaluated scope: MPI-based index sharding preserved exact top- k results while substantially reducing retrieval latency, without making global result consolidation the dominant cost for the main workload. The fundamental limitation remains the exhaustive search itself, since distributing the index reduces the work assigned to each rank but does not reduce the total number of vector comparisons. Pre-sharded storage provides a direction for reducing the observed loading overhead, while ANN search provides a direction for reducing the computational cost of retrieval itself.

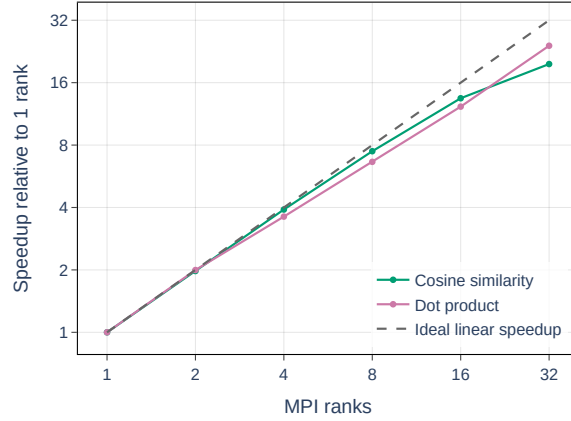
References

- [JDJ19] Jeff Johnson, Matthijs Douze, and Hervé Jégou. “Billion-scale similarity search with GPUs”. In: *IEEE transactions on big data* 7.3 (2019), pp. 535–547.
- [Kar+20] Vladimir Karpukhin, Barlas Oguz, Sewon Min, et al. “Dense passage retrieval for open-domain question answering”. In: *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*. 2020, pp. 6769–6781.
- [Lew+20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.
- [MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge: Cambridge University Press, 2008. ISBN: 978-0-521-86571-5.
- [MY20] Yu. A. Malkov and D. A. Yashunin. “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (2020), pp. 824–836. DOI: 10.1109/TPAMI.2018.2889473.
- [Tha+21] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. “BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models”. In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1*. Ed. by Joaquin Vanschoren and Sai-Kit Yeung. Virtual: Curran Associates, Inc., Dec. 2021. URL: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/65b9eea6e1cc6bb9f0cd2a47751a186f-Abstract-round2.html>.
- [Zha+25] Yanzhao Zhang, Mingxin Li, Dingkun Long, et al. *Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models*. 2025. arXiv: 2506.05176 [cs.CL]. URL: <https://arxiv.org/abs/2506.05176>.

A Additional benchmarks

A.1 Dot product vs. cosine similarity

The main evaluation uses inner-product search on unnormalized synthetic vectors. Since inner product depends on vector direction and magnitude, vectors with larger norms can receive higher scores even when their angular relation is unchanged. Cosine similarity removes this dependence through L2 normalization, making similarity depend only on the angle between vectors [MRS08]. Normalization changes the vector-value distribution and can therefore alter the numerical behavior of floating-point operations. This benchmark tests whether the main strong-scaling behavior persists under this change. Since both database and query vectors are normalized, cosine similarity still uses the same FAISS `IndexFlatIP` search path.



The benchmark compares the standard 500k-vector dot-product workload with a cosine-similarity variant based on an L2-normalized 500k-vector dataset, both with dimension 2560 and $k = 10$. Cosine similarity is implemented as inner-product search on normalized database and query vectors, so the same FAISS `IndexFlatIP` search path is used in both cases. Scaling is evaluated from 1 to 32 MPI ranks on a single node with a 32 GB memory allocation, using 1 CPU and 1 FAISS thread per rank, with centralized merging only. Reported values are medians of 50 measured one-query retrievals after 5 warmup runs.

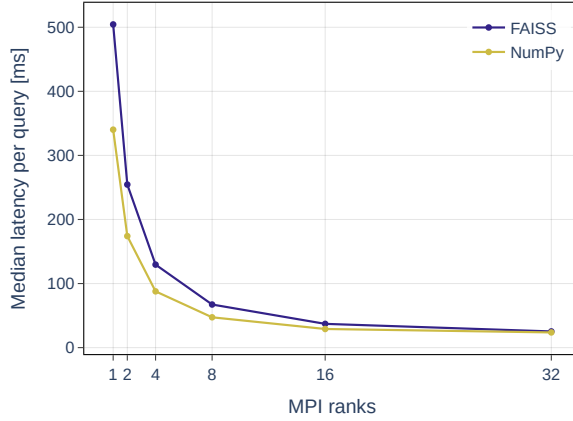
Figure 7: Speedup comparison for dot-product retrieval and cosine similarity implemented via normalized inner-product search.

Figure 7 shows similar strong-scaling behavior for both formulations at low and intermediate rank counts. At 16 ranks, the speedups are approximately 12.3 $\times$ for dot product and 13.5 $\times$ for cosine similarity. At 32 ranks, dot product reaches approximately 24.1 $\times$, while cosine similarity reaches about 19.7 $\times$.

The similar overall trend indicates that L2 normalization does not change the basic scaling behavior of MPI index sharding for this workload. This is consistent with both variants performing the same exhaustive inner-product search after initialization, despite the different vector magnitudes. The difference at 32 ranks is not sufficient to establish a systematic performance advantage for either representation, particularly given the variability observed in the main strong-scaling experiment. The result therefore supports applying the same MPI sharding approach to exact cosine-similarity retrieval when cosine similarity is implemented through normalized vectors.

A.2 FAISS vs. NumPy backend comparison

The retrieval framework supports both FAISS and NumPy as local exact-search backends, as described in Section 3.2. FAISS `IndexFlatIP` performs exhaustive inner-product search using a reusable local index, while the NumPy backend computes the local dot products directly and selects the highest-scoring entries with `argpartition`. This benchmark compares their hot-retrieval performance within the same MPI framework. Since the FAISS index is constructed before measurement, the comparison excludes index-construction time and considers only repeated retrieval. To keep the local CPU resources comparable, FAISS and OpenBLAS are each restricted to one thread per MPI rank.



The benchmark uses a fixed index of 500k vectors with dimension 2560 and retrieves the top- k results with $k = 10$ for one query at a time. The same MPI retrieval path is evaluated with either FAISS `IndexFlatIP` or the NumPy dot-product backend for local exact search. This is therefore a backend comparison within the MPI implementation, not a comparison against the sequential full-index correctness reference. Measurements use centralized merging and 1 to 32 MPI ranks on a single node with a 32 GB memory allocation, using 1 CPU per rank, 1 FAISS thread per rank and 1 OpenBLAS thread per rank. Reported values are medians of 50 measured retrievals after 5 warmup runs. Across the plotted configurations, the coefficient of variation of retrieval latency ranges from 0.3% to 4.1%.

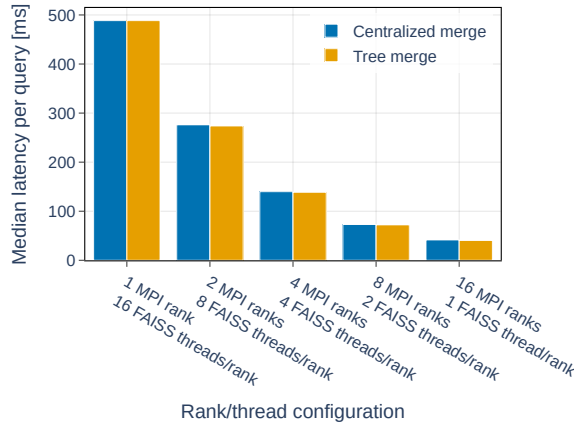
Figure 8: Hot-retrieval latency comparison between FAISS `IndexFlatIP` and NumPy local search.

Figure 8 shows decreasing retrieval latency for both backends as the number of MPI ranks increases. At one rank, the median latency is approximately 504 ms for FAISS and 340 ms for NumPy. The difference decreases as the local shards become smaller, and at 32 ranks the measured latencies are approximately 25.3 ms for FAISS and 23.7 ms for NumPy.

The lower NumPy latency at small rank counts is a notable result because FAISS index construction is excluded from the measured phase and the initialized `IndexFlatIP` index is reused for every retrieval. Determining why the NumPy implementation performs better in these configurations would require more detailed profiling of the local search kernels and is outside the scope of this benchmark. The result should therefore not be generalized into a broader performance ordering between NumPy and FAISS. This backend difference does not affect the quantitative scaling conclusions in the main evaluation because those benchmarks consistently use FAISS as the local search backend. The different absolute latencies therefore show that backend performance can differ substantially even when both implementations produce the same reference top- k result, as established by the correctness validation in Section 3.4.

A.3 MPI ranks vs. FAISS threads

The main evaluation increases the number of MPI ranks while using one FAISS thread per rank. Within each MPI rank, FAISS can additionally exploit CPU parallelism through native OpenMP threads. During these FAISS calls, the Python interface releases the Global Interpreter Lock, so the local search is not restricted by Python’s usual thread-level execution constraints⁹. This provides a second way of using additional CPU cores besides MPI index sharding. With a fixed CPU budget, the available cores can therefore be assigned either to more MPI ranks with smaller local shards or to fewer ranks with more FAISS threads searching larger shards. This benchmark compares these two forms of parallelism under a fixed 16-core allocation to determine which resource distribution provides lower hot-retrieval latency for the evaluated one-query workload.



The benchmark uses a fixed index of 500k vectors with dimension 2560 and retrieves the top- k results with $k = 10$ for one query at a time. A fixed total CPU budget of 16 cores is distributed across the configurations 1 rank $\times$ 16 FAISS threads, 2 ranks $\times$ 8 threads, 4 ranks $\times$ 4 threads, 8 ranks $\times$ 2 threads and 16 ranks $\times$ 1 thread. Both centralized and tree-based retrieval are evaluated on a single node with a 32 GB memory allocation. Reported values are medians of 50 measured one-query retrievals after 5 warmup runs. Across the plotted configurations, the coefficient of variation of retrieval latency ranges from 6.5% to 19.8%.

Figure 9: Hot-retrieval latency for different MPI-rank and FAISS-thread allocations under a fixed 16-core budget.

Figure 9 shows that retrieval latency decreases steadily as the fixed allocation of 16 Slurm CPUs is shifted from fewer ranks with more FAISS threads to more ranks with fewer threads. The highest latency occurs for 1 MPI rank with 16 FAISS threads at approximately 485 ms, while the lowest latency is reached for 16 MPI ranks with 1 FAISS thread per rank at approximately 40 ms. Centralized and tree-based merging remain very close across all tested configurations.

The magnitude of the difference is notable: although the two resource configurations use the same total allocation of Slurm CPUs, distributing the work across independent MPI ranks yields substantially lower latency than increasing FAISS threading within fewer ranks. The result points to differences in how effectively the two forms of parallelism use the available resources for this workload. The underlying cause remains unresolved and would require more detailed profiling of FAISS’s local search and memory behavior.

⁹*Threads and Asynchronous Calls*. URL: <https://github.com/facebookresearch/faiss/wiki/Threads-and-asynchronous-calls> (visited on Aug. 29, 2026).

B Individual contribution

This project was completed individually. All implementation, benchmarking, performance analysis and preparation of this report were carried out by the author.