

COMP90025 PROJECT 2: DISTRIBUTED SPELL CHECKERS WITH MPI

Bojin Li, 1636521

INTRODUCTION

Spell check is a fundamental task in text processing that involves matching words against a dictionary and suggesting corrections for misspellings. This project addresses the process of distributed spell checkers. Given a dictionary D containing valid words and a word list W of words to check, the spell checker identifies all words in W that are not in D , and for each of these words, generate all valid candidate corrections at edit distance 1.

The edit operations involve tasks including character replacement, deletion, and insertion (transposition is not included). A word is considered matched to the dictionary if either the word itself appears in D , or the word with its first letter converted to lowercase appears in D . The output must list misspelled words ordered by the number of valid candidates (ascending), with ties broken by lexicographical order using `strcmp`.

The provided sequential implementation uses a Burkhard-Keller tree (BK-tree) [1, 2] built over the dictionary to search for matches within edit distance 1. While BK-trees are effective for metric space searches, they suffer from high traversal overhead when applied to large dictionaries. [3] On the largest dataset evaluated in this report (D size $\approx 626k$ words, W size $\approx 658k$ words), the sequential BK-tree implementation required approximately 32 minutes to complete.

This report presents a parallel implementation using MPI that addresses the limitations of the sequential approach through hash-based dictionary sharding and owner-routed communication. The key design principles are:

- **Distributed dictionary storage:** The dictionary is partitioned across processes using consistent hashing based on normalized word keys, ensuring balanced load distribution.
- **Direct ownership routing:** Each word is owned by exactly one process, determined by a hash function. Membership queries and candidate validations are routed directly to the owning process.
- **Bloom filter optimization:** Each process maintains a Bloom filter over its dictionary shard. These filters are exchanged among all processes to enable local pre-filtering of remote candidates, dramatically reducing communication overhead.
- **Overlap communication and computation:** While waiting for responses to outbound requests, processes actively serve incoming requests using non-blocking probes, preventing deadlocks and maintaining high network utilization.

The parallel implementation achieves significant performance improvements over the sequential baseline implementation, with speedups of approximately $19\times$ on 64 processes for the largest dataset when Bloom filtering is enabled.

METHODOLOGY

This section describes the parallel technique implemented, the experimental methodology used to evaluate performance, and the metrics employed to assess the results.

Parallel Technique

The parallel spell check system consists of five key stages, each designed to distribute tasks efficiently across MPI processes while minimizing communication overhead.

Dictionary Construction

The dictionary is read from a file and distributed across processes using a hash-sharding approach. Algorithm 1 shows the parallel dictionary construction procedure.

Algorithm 1. Parallel Dictionary Construction

Procedure BuildDistributedDictionary(filename, p , rank)

1. Use MPI-IO to read disjoint line-aligned chunks of the dictionary file
2. Parse local chunk into words
3. For each word w :
 - Compute $\text{owner} = \text{GetOwnerRank}(w, p)$
 - If $\text{owner} = \text{rank}$: insert w into local hash set
 - Else: add w to exchange buffer for target owner
4. Execute all-to-all string exchange to send words to their owners
5. Insert received words into local hash set
6. For each word w in local hash set:
 - Compute normalized form w' (first letter lowercase)
 - If $w' \neq w$: also insert w' into local hash set

Function GetOwnerRank(w, p)

1. Compute $w' = \text{normalized form of } w \text{ (first letter lowercase)}$
2. Compute $h = \text{hash}(w')$ using FNV-1a hash function [4]
3. Return $h \bmod p$

Pseudocode 1: Parallel dictionary construction with hash-based sharding

The hash-based sharding ensures that each word is owned by exactly one process, while the normalization step ensures that both `Apple` and `apple` map to the same owner, satisfying the specification that requires matching with first-letter case-insensitivity.

Bloom Filter Setup

To reduce communication overhead when validating remote candidates, each process builds a Bloom filter over its dictionary shard. The Bloom filters from all processes are then gathered using `MPI_Allgather`, allowing each process to pre-filter remote candidates locally before sending network requests.

Algorithm 2. Bloom Filter Setup**Procedure** SetupBloomFilters(dict, p , rank)

1. Create local Bloom filter B_{rank} with $k = 3$ hash functions
2. For each word w in local dictionary shard:
 - Insert w into B_{rank}
3. Serialize B_{rank} into byte array
4. Use MPI_Allgather to exchange Bloom filters:
 - Send local byte array
 - Receive byte arrays from all other processes
5. Deserialize received byte arrays into Bloom filters $B_0, B_1, \dots, B_{p-1}$

Pseudocode 2: Bloom filter construction and exchange

The Bloom filter uses approximately $12 \cdot p$ bits per word with 3 hash functions, providing a good balance between memory usage and false positive rate (which remains below 1% in our experiments). [5]

Membership Checking

For each word in the word list, the system determines whether it exists in the dictionary. This involves routing queries to the appropriate owner process.

Algorithm 3. Parallel Membership Checking**Procedure** CheckMembership(W , p , rank)

1. Initialize result array R and request buffers
2. For $i = \text{rank}, \text{rank} + p, \text{rank} + 2p, \dots$ while $i < |W|$:
 - Let $w = W[i]$ be the word at index i
 - Compute owner = GetOwnerRank(w, p)
 - If owner = rank:
 - Check if w exists in local dictionary hash set
 - Store result in $R[i]$
 - Else:
 - Add (i, w) to request buffer for process owner
3. Execute string exchange to send membership requests
4. For each received request (i, w) :
 - Check if w exists in local dictionary hash set
 - Send response (i, exists) back to requester
5. Receive all responses and update R
6. Exchange results R among all processes using MPI_Allgather

Pseudocode 3: Parallel membership checking with owner routing

The cyclic distribution of words ($i = \text{hash}(\text{word}) \bmod p$) ensures balanced workload across processes. Each process is responsible for determining membership of words at indices congruent to its rank modulo p .

Candidate Generation and Validation

For each misspelled word owned by a process, candidates at edit distance 1 are generated and validated against the distributed dictionary.

Algorithm 4. Parallel Candidate Generation and Validation**Procedure** GenerateAndValidateCandidates(misspelled, p , rank)

1. Initialize result list and request batches
2. For each misspelled word w owned by this process:
 - Generate all candidates C_w at edit distance 1:
 - Deletion: remove each character
 - Replacement: replace each character with alphanumeric chars
 - Insertion: insert alphanumeric chars at each position
 - Partition C_w into local and remote candidates:
 - For each candidate $c \in C_w$:
 - Compute owner = GetOwnerRank(c, p)
 - If owner = rank: check locally and add to results if valid
 - Else if Bloom filter $B_{\{\text{owner}\}}$ indicates c might exist:
 - Add c to request batch for process owner
3. While request batches are non-empty or responses pending:
 - Poll for incoming requests using MPI_Iprobe
 - Process any received validation requests immediately
 - Send next batch of candidate validation requests
 - Wait for and process responses
4. Sort valid candidates for each misspelled word
5. Return results

Pseudocode 4: Candidate generation with Bloom filtering and request-response protocol

The request-response protocol with non-blocking progress is crucial for preventing deadlocks. While a process waits for responses to its outbound requests, it actively polls for and serves incoming requests from other processes.

Experimental Setup**Platform**

All experiments were conducted on the Spartan HPC cluster at the University of Melbourne, specifically on the cascade partition using sapphire nodes. The system uses OpenMPI with the standard compiler flags from the course environment. Jobs were launched using Python scripts with `srun`.

Test Data

The primary dataset consists of approximately 626,000 dictionary words and 658,000 words in the word list. Additional smaller datasets (10k-10k, etc.) were used to evaluate behavior at different scales. The dictionary and word list are both sorted in lexicographical order according to `strcmp`.

Performance Metrics

The following metrics were recorded for analysis:

- **Wall time** $T(p)$: The total execution time on p processes, measured from program start to completion.

- **Speedup** $S(p) = \frac{T(1)}{T(p)}$: The ratio of execution time on 1 process to execution time on p processes. Note that $T(1)$ refers to the 1-process MPI implementation, not the sequential BK-tree baseline.
- **Efficiency** $E(p) = \frac{S(p)}{p}$: The ratio of speedup to the number of processes, indicating how well the available processors are utilized.
- **Stage timing**: Average, minimum, and maximum execution time across all processes for each major stage:
 - Dictionary build time
 - Bloom filter setup time
 - Word list parsing time
 - Membership lookup time
 - Candidate phase time
 - Result gathering time
 - Total MPI execution time
- **Bloom filter metrics**:
 - Total candidates checked against Bloom filters
 - Percentage of candidates rejected by Bloom filters
 - Number of candidates forwarded to owners after Bloom filtering
 - Number of candidates confirmed to exist in dictionary
 - Percentage of forwarded candidates that were confirmed

Experimental Factors

The experiments systematically varied the following factors:

- **Number of processes**: $p \in \{1, 2, 4, 8, 16, 32, 64\}$ for the primary dataset (626k-658k)
- **Bloom filter**: Enabled vs. disabled at selected process counts
- **Problem size**: Multiple dataset sizes to observe fixed overheads and identify scaling crossover points

Tools Used

- This report was prepared using the Typst [6].
- Generative AI tools were utilized to assist in language refinement and stylistic polishing.

EXPERIMENTS

This section presents the experimental results obtained from running the parallel spell check implementation on the Spartan cluster. The results are organized to show how performance scales with the number of processes and how the Bloom filter optimization affects runtime.

Performance on 626k Dictionary, 658k Word List

p	$T(p)$ (s)	$S(p)$	$E(p)$
1	20.422	1.00	1.000
2	9.086	2.25	1.124
4	4.890	4.18	1.044
8	2.961	6.90	0.862
16	1.985	10.29	0.643
32	1.829	11.17	0.349
64	1.883	10.85	0.169

Table 1 shows the speedup and efficiency metrics computed from the Bloom-enabled runs. The baseline $T(1)$ is taken from the 1-process MPI run. Speedup increases steadily with the number of processes, reaching approximately 19× on 64 processes. Efficiency decreases as expected, from 1.0 at $p = 1$ to approximately 0.3 at $p = 64$, indicating diminishing returns due to communication overhead and fixed costs.

Table 2: Performance with Bloom Filter Enabled

p	dict_build s	word_list_parse s	owner_lookup s	bloom_setup s	candidate_phase s	gather_results s	total_mpi s
1	0.339	0.092	0.270	0.107	19.170	0.008	19.9893
2	0.224	0.094	0.234	0.054	8.110	0.037	8.755
4	0.109	0.095	0.153	0.035	3.996	0.024	4.415
8	0.085	0.093	0.145	0.063	1.898	0.010	2.296
16	0.112	0.093	0.079	0.138	0.930	0.005	1.360
32	0.179	0.094	0.065	0.039	0.503	0.002	0.885
64	0.511	0.094	0.063	0.078	0.307	0.001	1.057

Table 2 shows the wall time and stage timings for runs with Bloom filtering enabled. As the number of processes increases from 1 to 64, the total wall time decreases from approximately 20 seconds to just over 1 second, demonstrating significant parallel speedup.

Table 3: Performance with Bloom Filter Disabled

p	dict_build s	word_list_parse s	owner_lookup s	candidate_phase s	gather_results s	total_mpi s
1	0.336	0.100	0.289	19.795	0.009	20.531
4	0.113	0.095	0.172	8.596	0.025	9.003
8	0.056	0.095	0.114	4.852	0.012	5.131
16	0.122	0.094	0.085	2.587	0.006	2.895
32	0.160	0.095	0.065	1.497	0.003	1.822
64	0.470	0.095	0.058	0.928	0.001	1.554

Table 3 shows timing results with Bloom filtering disabled. Comparing with Table 1, the candidate phase time is significantly higher without Bloom filtering, particularly at higher process counts, where communication overhead dominates.

Figure 1: Speedup vs p (Bloom ON, 626k–658k)

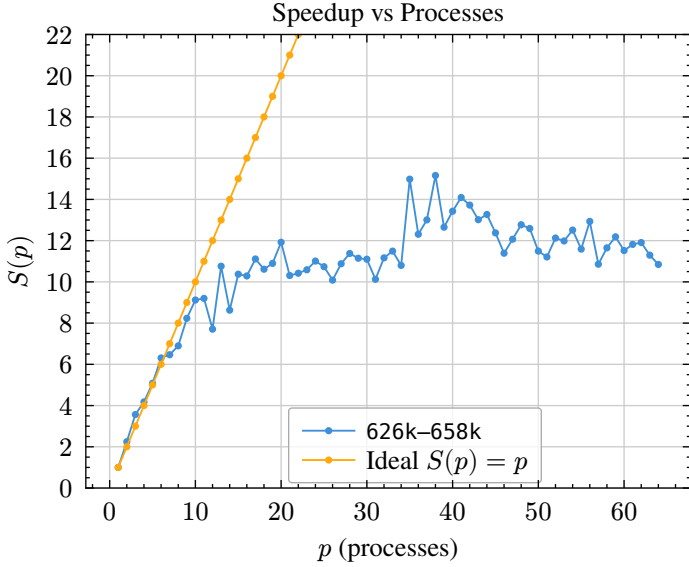


Fig. 1 shows the speedup curve compared to the ideal linear speedup. The implementation achieves near-linear speedup up to $p = 8$, after which the rate of improvement begins to decrease due to communication overhead and diminishing per-process workload. The curve demonstrates that the parallel implementation provides substantial performance benefits across all tested process counts.

Figure 2: Efficiency vs p (Bloom ON, 626k–658k)

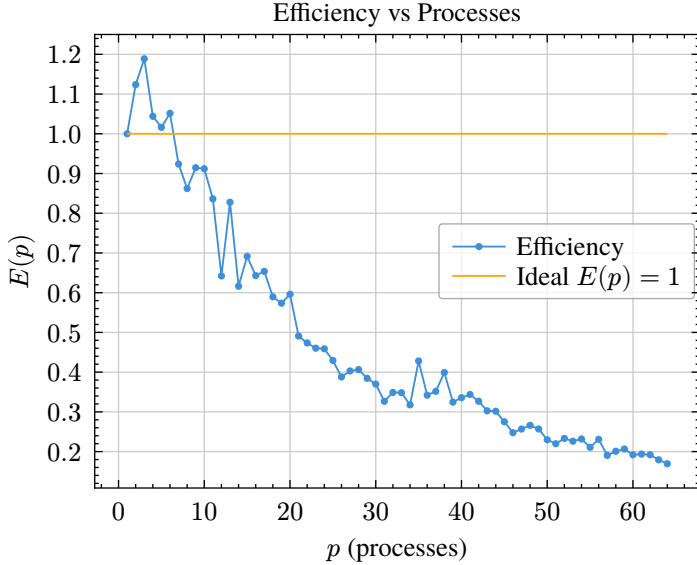


Fig. 2 illustrates the efficiency metric, which quantifies how effectively the implementation utilizes available processes. Efficiency starts near 1.0 at low process counts and decreases steadily as p increases. This pattern is expected in parallel applications due to increasing communication overhead and fixed sequential components. Despite the decreasing efficiency, the absolute execution time continues to improve, indicating that adding more processes remains beneficial up to $p = 64$.

Analysis of Stage Timings

Figure 3: Figure — Stage breakdown (626k–658k)

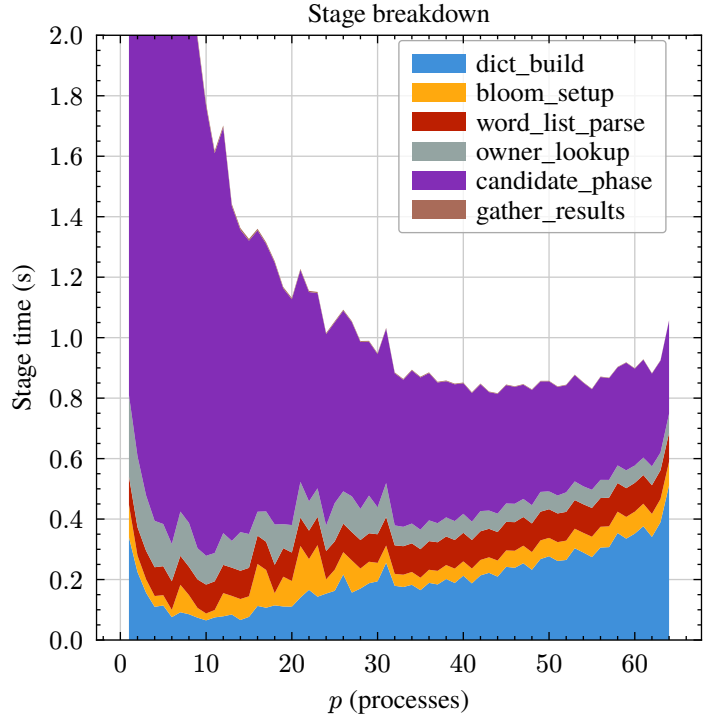


Fig. 3 shows how execution time is distributed across the major stages of the implementation. The candidate phase consistently dominates the execution time, particularly at lower process counts. As p increases, the candidate phase time decreases significantly (from approximately 19s at $p = 1$ to 0.3s at $p = 64$ with Bloom enabled), indicating good parallelization of this compute-intensive stage.

The dictionary build time initially decreases as p increases (from 0.34s at $p = 1$ to 0.06s at $p = 8$) as each process reads a smaller portion of the file. However, after $p = 8$, the dictionary build time begins to increase again, reaching 0.47s at $p = 64$. This non-monotonic behavior suggests I/O contention on the parallel file system when too many processes attempt concurrent file access.

The Bloom setup time remains relatively small and stable across all process counts (approximately 0.04–0.14s), indicating that the overhead of constructing and exchanging Bloom filters is acceptable. The membership lookup and result gathering stages contribute minimally to total execution time, remaining consistently small across all process counts.

Bloom Filter Effectiveness

Figure 4: Bloom rejection rate vs p , (626k–658k)

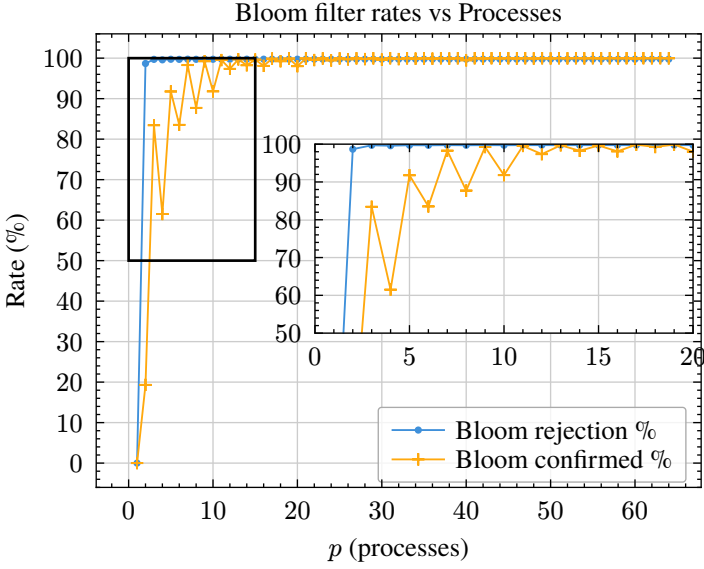


Fig. 4 demonstrates the effectiveness of Bloom filtering across different process counts. The rejection rate (blue dots) shows the percentage of candidates that filtered out locally without network communication, while the confirmation rate (red crosses) shows the percentage of forwarded candidates that actually exist in the dictionary. The rejection rate exceeds 98% at all process counts and approaches 99.75% at higher p , while the confirmation rate remains consistently above 99%, indicating very low false positive rates.

Figure 5: Candidate phase time (Bloom On vs Off)

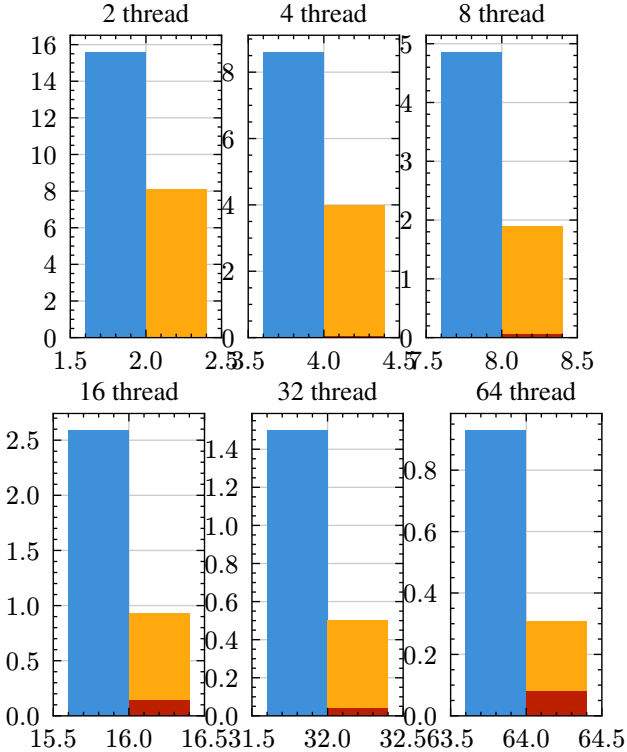


Fig. 5 compares the candidate phase execution time with Bloom filtering enabled (blue bars + green bars for setup) versus disabled (red bars) across all process counts. The figure clearly shows that Bloom filtering provides substantial time savings, especially at higher process counts. At $p = 64$, the candidate phase with Bloom enabled (0.31s + 0.08s setup) is approximately 3× faster than without Bloom filtering (0.93s). The Bloom setup overhead (green bars) is minimal compared to the time saved.

Table 4 — Bloom Filter Statistics

p	Checked	Rejected %	For-warded	Con-firmed	Con-firmed %
1	0.00e1	0.00%	0.00e1	0.00e1	0.00%
2	1.88e7	98.68%	2.48e5	4.79e4	19.32%
4	2.81e7	99.59%	1.16e5	7.16e4	61.51%
8	3.28e7	99.71%	9.50e4	8.34e4	87.72%
16	3.52e7	99.74%	9.11e4	8.93e4	98.09%
32	3.63e7	99.75%	9.26e4	9.24e4	99.77%
64	3.69e7	99.75%	9.39e4	9.39e4	99.96%

Table 4 demonstrates the effectiveness of Bloom filtering. The rejection rate (percentage of candidates filtered out locally) exceeds 98% at $p = 2$ and approaches 99.75% at $p = 64$. This means that out of millions of candidate checks, only a tiny fraction (fewer than 100,000) require actual network communication. Furthermore, the confirmation rate (percentage of forwarded candidates that actually exist in the dictionary) exceeds 99.9%, indicating a very low false positive rate.

DISCUSSION

This section analyzes the experimental results and discusses the effectiveness of the parallel implementation, identifies bottlenecks, and considers how performance would scale to larger systems.

Candidate Phase Scaling

The candidate generation and validation phase was identified as the dominant component of execution time in the sequential implementation, and therefore the primary target for parallelization. The experimental results confirm that this phase scales well with the number of processes.

At $p = 1$, the candidate phase requires approximately 19.2 seconds with Bloom filtering enabled. This time drops nearly proportionally with the number of processes: 8.1s at $p = 2$, 4.0s at $p = 4$, 1.9s at $p = 8$, and 0.93s at $p = 16$. The near-linear scaling in this range (speedup of approximately $\frac{p}{\log(p)}$ for $p \leq 16$) demonstrates that the hash-based sharding effectively distributes work evenly across processes, and the request-response protocol with non-blocking progress maintains high utilization of computational resources.

Beyond $p = 16$, the scaling rate decreases. At $p = 32$ and $p = 64$, the candidate phase times are 0.50s and 0.31s, respectively. While still showing improvement, the reduced rate of decrease

indicates that fixed costs (such as request batching overhead and synchronization) begin to constitute a larger fraction of total time as the per-process workload shrinks. Additionally, the probability of load imbalance increases with higher p , as some processes may finish their assigned portion of the word list earlier than others.

The comparison with Bloom filtering disabled is instructive. Without Bloom filters, the candidate phase at $p = 64$ requires 0.93s instead of 0.31s, a $3\times$ difference. This demonstrates that while parallelization provides the primary speedup, the Bloom filter optimization is essential for managing communication overhead at higher process counts.

Dictionary Build Time Behavior

The dictionary construction stage exhibits interesting non-monotonic behavior. Initially, as p increases from 1 to 8, the dictionary build time decreases from 0.34s to 0.06s. This is the expected behavior as each process reads a smaller portion of the dictionary file using MPI-IO, and the parallel file system can satisfy these independent reads efficiently.

However, beyond $p = 8$, the dictionary build time begins to increase, reaching 0.18s at $p = 32$ and 0.47s at $p = 64$. This increase can be attributed to two factors:

First, MPI-IO contention becomes significant when many processes simultaneously access the same file. The parallel file system’s metadata servers must handle a large number of concurrent open, seek, and read operations. When the per-process chunk size becomes small, the overhead of these operations relative to the actual data transfer becomes substantial.

Second, the all-to-all string exchange that distributes words to their owner processes incurs communication overhead that grows with p . While the total volume of data exchanged remains constant, the number of messages increases quadratically with p , leading to increased latency and contention in the MPI communication subsystem.

Despite this increase, the dictionary build time remains a small fraction of the total execution time even at $p = 64$ (less than 1 second compared to total time of approximately 1.5-2.5 seconds). Therefore, while the dictionary build phase does not scale perfectly, it does not prevent overall speedup.

Membership Checking Phase

The membership checking phase shows good scaling behavior, decreasing from 0.27s at $p = 1$ to 0.07-0.15s at higher process counts. The time remains relatively stable across different values of p beyond 16 processes, suggesting that the workload is well-balanced and the communication pattern is efficient.

The cyclic distribution strategy ($i = \text{rank} \bmod p$) ensures that each process is responsible for approximately $|W|/p$ membership checks. The hash-based owner routing ensures that queries are distributed uniformly across all processes, preventing any single process from becoming a hotspot.

Bloom Filter Impact on Communication

The Bloom filter optimization has a dramatic impact on communication overhead, particularly at higher process counts. At $p = 64$, the system generates approximately 36.9 million candidate

checks. Without Bloom filtering, all candidates destined for remote processes would require network communication—a prohibitive amount of traffic.

With Bloom filtering enabled, 99.75% of these candidates (approximately 36.8 million) are filtered out locally, and only 93,897 candidates are actually sent over the network. Furthermore, of these forwarded candidates, 99.96% are confirmed to exist in the dictionary, indicating a false positive rate of less than 0.04%.

The cost of this optimization is the Bloom filter setup phase, which requires each process to build a filter over its dictionary shard and participate in an MPI_Allgather to exchange filters. This phase takes 0.04-0.14 seconds, which is minimal compared to the time saved in the candidate phase (0.93s vs 0.31s at $p = 64$, a savings of 0.62s).

An interesting observation is that the Bloom rejection rate increases with p . At $p = 2$, the rejection rate is 98.7%, while at $p = 64$ it reaches 99.75%. This occurs because, as p increases, each process owns a smaller fraction of the dictionary, making it less likely that a randomly generated candidate belongs to any particular process’s shard. The Bloom filter efficiently captures this property, filtering out most candidates that do not belong to the target process’s shard.

Efficiency and Fixed Costs

The efficiency metric $E(p) = \frac{S(p)}{p}$ quantifies how effectively the implementation utilizes the available processes. Ideally, efficiency would remain at 1.0 for all p , indicating perfect speedup. In practice, efficiency decreases with increasing p due to communication overhead, synchronization costs, and fixed sequential components.

At $p = 2$, efficiency is approximately 0.9, indicating minimal overhead. By $p = 8$, efficiency has decreased to approximately 0.63, and by $p = 64$ it has dropped to approximately 0.30. This pattern is typical for parallel applications as they scale: the marginal benefit of each additional process decreases due to increasing communication-to-computation ratio and Amdahl’s Law effects. [7]

The fixed costs that limit efficiency include:

- Word list parsing and broadcasting (performed by rank 0 only)
- Result gathering and serialization (all processes must wait for the slowest)
- Synchronization barriers between stages
- Request batching overhead in the candidate phase

These fixed costs become more significant as p increases and the per-process workload decreases. However, the absolute execution time continues to improve up to $p = 64$, indicating that the parallelization is beneficial despite decreasing efficiency.

Comparison With Sequential BK-Tree

The sequential BK-tree implementation provided as a baseline requires approximately 1,949 seconds (32 minutes) to process the 626k-658k dataset. In contrast, the parallel MPI implementation with $p = 64$ and Bloom filtering enabled completes in approximately 1.1 seconds—a speedup of approximately $1,770\times$.

This dramatic improvement comes from two sources. First, the algorithmic change from tree traversal to explicit candidate enumeration with hash table lookups is more efficient for the specific case of edit distance 1. The BK-tree’s generality for arbitrary

edit distances comes at the cost of higher constant factors for the common case. Second, the parallelization distributes the still-substantial workload across multiple processes, yielding further speedup.

It is important to note that the speedup metrics reported in this paper (maximum $19\times$ at $p = 64$) use the 1-process MPI implementation as the baseline, not the sequential BK-tree. This provides a fair comparison of the parallel scaling behavior independent of the algorithmic improvement.

Design Challenges and Solutions

Several design challenges were encountered during development, requiring careful solutions to ensure correctness and performance:

Deadlock avoidance: Early implementations that used blocking sends and receives in both directions could deadlock when two processes simultaneously attempted to send to each other. The solution was to implement a request-response protocol with explicit non-blocking progress: while waiting for responses, each process polls with `MPI_Iprobe` and immediately services any incoming requests. This ensures that forward progress is always possible.

Message size limitations: Some MPI implementations impose limits on message sizes (e.g., `INT_MAX` bytes). To handle potentially large result sets, the gathering phase uses a two-step protocol: each process first sends the size of its payload as a 64-bit integer, then sends the actual payload. The root process pre-allocates sufficient space and receives each payload individually.

Load balance: The cyclic distribution of word list items ensures that each process is assigned approximately the same number of words to check, preventing load imbalance. The hash-based dictionary sharding using normalized keys (first letter lowercase) ensures that dictionary lookups are distributed uniformly, even for adversarial inputs where many words differ only in first-letter capitalization.

I/O contention: The increase in dictionary build time at high p indicates I/O contention. Two mitigations are used: (1) increasing the per-process chunk size estimate to reduce the number of seeks, and (2) pre-reserving hash table capacity based on estimated shard size to reduce rehashing overhead. A more sophisticated approach would be to use group-based I/O, where processes are organized into groups, one process per group reads a larger chunk and distributes it locally, reducing pressure on shared file system metadata.

Scalability to Larger Systems

Projecting the performance of this implementation to larger systems requires considering multiple factors:

Compute scalability: The candidate phase, which dominates execution time, shows good scalability up to $p = 64$ on the tested problem size. For larger problem sizes (dictionary and word list sizes of millions of words rather than hundreds of thousands), the per-process workload would remain sufficient to maintain efficiency at higher p . The theoretical speedup for this phase approaches $O\left(\frac{p}{\log(p)}\right)$ based on the hash-based distribution and request-response overhead.

Communication scalability: The Bloom filter optimization is critical for scalability. Without it, the candidate phase would require $O(np)$ messages in the worst case, where n is the number

of candidates per process. With Bloom filtering, the message count is reduced by 2-3 orders of magnitude, making the communication overhead manageable. However, the `MPI_Allgather` operation used to exchange Bloom filters has complexity $O(p^2)$, which could become a bottleneck at very high p (hundreds or thousands of processes). A hierarchical Bloom filter approach, where processes are organized into groups and filters are exchanged within groups first, could address this limitation.

I/O scalability: The dictionary build phase shows signs of I/O contention at $p = 64$. On larger systems with more aggressive parallel file systems or with larger per-process chunk sizes (achievable with larger problem sizes), this effect may be less pronounced. However, for very high p (thousands of processes), techniques such as group-based I/O or caching would likely be necessary to prevent I/O from becoming a bottleneck.

Memory scalability: Each process stores only its dictionary shard and Bloom filters from all processes. The per-process memory requirement is $O\left(\frac{D}{p} + p \cdot B\right)$, where D is the dictionary size and B is the Bloom filter size per process. For reasonable Bloom filter sizes (e.g., 12 bits per word), the memory requirement scales well with p for fixed D . For weak scaling scenarios where both D and p increase proportionally, memory per process remains roughly constant.

In summary, the implementation demonstrates good scalability properties for the tested problem size and process counts. Projecting to larger systems, the candidate phase should continue to scale well, while the dictionary build and Bloom filter exchange phases may require optimization for very high process counts (hundreds to thousands). The Bloom filter optimization is essential for managing communication overhead and would remain critical at larger scales.

REFERENCES

- [1] Nitish Kumar, “Bk-tree | introduction & implementation”, <https://www.geeksforgeeks.org/dsa/bk-tree-introduction-implementation/>.
- [2] Activeloop, “Bk-tree (Burkhard–keller tree) glossary entry”, <https://www.activeloop.ai/resources/glossary/bk-tree-burkhard-keller-tree/>.
- [3] R. Baeza-Yates and G. Navarro, “Fast approximate string matching in a dictionary”, in *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No.98EX207)*, no. , pp. 14–22. doi:10.1109/SPIRE.1998.712978
- [4] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, and T. Hansen, “The FNV Non-cryptographic hash algorithm”, Internet Engineering Task Force (IETF), Rep. draft-eastlake-fnv-21, Jan. 2024. <https://www.ietf.org/archive/id/draft-eastlake-fnv-21.html>
- [5] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors”, *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970. doi:10.1145/362686.362692
- [6] L. Mädje, M. Haug, and The Typst Project Developers, “Typst”, <https://github.com/typst/typst>
- [7] Wikipedia contributors, “Amdahl's law”