Indian Journal of Computer Science and Technology
Volume 5, Issue 2 (May-August 2026), PP: 29-35
www.indjcst.com
ISSN No: 2583-5300

A Comparative Study of STL Container Performance in C++17

Atharva Shankar Jedhe[1], Karan Kailas Kadam[2]
[1,2] Department of Computer Science, P.V.G.'s College of Science and Commerce, Savitribai Phule Pune University, Maharashtra, India.

To Cite this Article: *Atharva Shankar Jedhe[1], Karan Kailas Kadam[2], "A Comparative Study of STL Container Performance in C++17", Indian Journal of Computer Science and Technology, Volume 05, Issue 02 (May-August 2026), PP: 29-35.*

Copyright: ©2026 This is an open access journal, and articles are distributed under the terms of the Creative Commons Attribution License; Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Abstract: *This study presents an empirical performance evaluation of key C++ Standard Template Library (STL) containers, including std::vector, std::list, std::deque, std::set, and std::unordered_set. The experiments were conducted on a Windows platform using the MSVC compiler under both optimized (/O2) and non-optimized (/Od) configurations. Performance was measured across four fundamental operations— insertion, traversal, search, and deletion—using datasets ranging from 1,000 to 500,000 elements. The results demonstrate significant variation in runtime behavior depending on container type and compiler optimization level. The findings provide practical insights for selecting appropriate STL containers in performance-critical applications.*

Key Words: *STL; C++17; Benchmarking; Containers; Performance.*

I. INTRODUCTION AND OBJECTIVES

The C++ Standard Template Library (STL) is one of the most widely used components of modern C++. It offers a rich collection of generic containers and algorithms that enable developers to write expressive, reusable, and high-performance code. Containers such as std::vector, std::list, std::deque, std::set, and std::unordered_set cover a spectrum of design choices in terms of memory layout, iterator stability, ordering guarantees, and algorithmic complexity. Because these containers are heavily optimized in major standard library implementations, many developers assume that performance will always be "good enough" and rely largely on intuition when selecting a container for a particular problem. However, practical performance is not determined by asymptotic complexity alone. Real-world behavior depends on low-level details such as cache locality, branch prediction, allocator behavior, inlining heuristics, and compiler optimization levels. For example, std::vector's contiguous storage tends to interact very favorably with CPU caches, often allowing it to outperform node-based structures such as std::list, even in workloads that involve insertions. On the other hand, hash-based containers like std::unordered_set may provide extremely fast average-case lookups but can exhibit high memory usage due to internal bucket structures and load-factor policies. This research focuses on an empirical study of several core STL containers on the Windows platform using the MSVC toolchain. The study investigates how performance changes under two common compilation modes—an optimized /O2 build and a non-optimized /Od build—and how runtime and memory usage scale with increasing data sizes. By running a systematic benchmark that varies container type, operation type, and dataset size, this work aims to quantify the trade-offs between different containers and provide concrete guidance for developers writing performance-sensitive C++17 code. The main objectives of this study are: (1) to measure and compare the execution time of insertion, traversal, search, and deletion operations across std::vector, std::list, std::deque, std::set, and std::unordered_set; (2) to analyze how performance and memory usage scale for dataset sizes of 1,000, 10,000, 100,000, and 500,000 elements; (3) to compare the behavior of containers between optimized and non-optimized builds; and (4) to summarize practical recommendations for container selection based on observed evidence rather than intuition alone.

II.LITERATURE REVIEW

A substantial body of work discusses the design and usage of the C++ Standard Template Library, ranging from introductory overviews to advanced performance-oriented analyses. Schmidt's tutorial on the STL provides a structured overview of the key concepts behind generic programming, iterators, containers, and algorithms, emphasizing how the library is designed for both efficiency and flexibility (This kind of foundational material clarifies the intent and typical usage patterns of containers, but it does not provide detailed empirical measurements of runtime and memory behavior under modern workloads. https://www.dre.vanderbilt.edu/~schmidt/PDF/stl.pdf). Other works examine container usage from a performance and scalability perspective. Laso, Krupitza, and Hunold introduce pSTL-Bench, a micro-benchmark suite for assessing scalability of parallel STL implementations across different compilers and backends (Although their focus is on parallel algorithms introduced in C++17 rather than on individual sequential containers, their methodology—carefully controlled micro-benchmarks and cross-platform comparisons—strongly motivates the approach taken in this study. https://doi.org/10.1145/3673038.3673065).

Pataki's work on template-specialized STL containers explores how generic programming techniques can be used to emit warnings for potentially dangerous container usage patterns and to detect misuse of certain container specializations (Although this research is more concerned with correctness and static analysis than raw performance, it underscores that STL containers must

be used with care and that their semantics, including allocator behavior and iterator properties, can be nontrivial. https://doi.org/10.48550/arXiv.1111.3673). The performance and memory characteristics of STL-like containers are also examined in several related domains. Crutcher's ETL comparison study shows how fixed-capacity containers can achieve better worst-case predictability than their dynamic STL counterparts in embedded systems, while still retaining acceptable performance. Schuessler and Gruber investigate a specialized small-object allocator in C++ and compare its behavior to containers such as std::vector and std::list, showing how custom allocation strategies can affect both speed and memory usage (These works highlight that the interaction between containers and allocators is crucial for performance-critical applications. https://arxiv.org/abs/1611.01667). Additional perspectives come from both research and practice. DeLozier presents an approach to GPU acceleration for STL-like algorithms, illustrating how standard library abstractions can be mapped to heterogeneous hardware while preserving performance expectations (McDonald and Strooper describe a programmatic test suite for STL conformance, noting that correctness and performance specifications should both be taken into account when evaluating implementations (Qin and co-authors, in Primrose, propose a system for selecting container data types based on their properties, which conceptually aligns with the goal of choosing the most appropriate container for given performance and memory requirements https://www.seas.upenn.edu/~delozier/docs/libcxxgpu.pdf).(https://www.cs.ubc.ca/labs/isd/FormalWare/abstracts/progra~1.pdf). (https://arxiv.org/pdf/2205.09655). Finally, game-development and high-performance communities have produced applied studies of STL containers. For example, an article in the Game AI Pro Online Edition evaluates the suitability of STL containers in game engines, emphasizing how container choice can influence cache behavior, memory usage, and frame-time stability in real-time systems (Collectively, these works support the view that container choice and implementation details matter greatly and motivate focused, platform-specific empirical benchmarking such as the study presented in this report.https://www.gameaipro.com/GameAIProOnlineEdition2021/GameAIProOnlineEdition2021_Chapter15_Should_STL_containers_be_used_in_game_engines.pdf).

III. RESEARCH METHODOLOGY / DESIGN

The study adopts a quantitative experimental methodology based exclusively on execution-time benchmarking of STL containers. All experiments were implemented in C++17 and compiled using the Microsoft Visual C++ (MSVC) toolchain on Windows. Two build configurations were used independently during experimentation: an optimized build (/O2) and a non-optimized build (/Od). Each configuration was executed separately and analyzed independently.

3.1 Five STL containers were selected:

1) std::vector
2) std::list
3) std::deque
4) std::set
5) std::unordered_set

3.2 Four fundamental operations were benchmarked:

1) Insertion
2) Traversal
3) Search
4) Deletion

3.3 Dataset sizes were fixed at:

1,000
10,000
100,000
500,000 elements

Each experiment was repeated three times to reduce transient timing noise. The final dataset therefore consists of: 5 containers × 4 operations × 4 sizes × 3 runs = 240 benchmark executions per build configuration Execution time was measured using std::chrono::high_resolution_clock and recorded in milliseconds. For each test case, results were written to a CSV file containing: container name, operation type, data size, run index, elapsed time (ms) No memory measurements were collected in this implementation; the study focuses strictly on runtime performance.

3.1 Experimental Variable Classification

Category	Variable	Description
Independent Variables	Container Type	vector, list, deque, set, unordered_set
	Operation Type	insert, traverse, search, delete
	Data Size	1K, 10K, 100K, 500K
	Build Configuration	/O2 or /Od (analyzed separately)
Dependent Variable	Execution Time	Measured in milliseconds

Controlled Variables	Compiler	MSVC
	Standard	C++17
	Hardware	Same Windows machine
	Repetitions	3 per test

3.4 Threats to Validity

Although the experimental setup was controlled, several factors may influence the results. Background operating system processes may introduce minor timing variability. The random number generator is seeded using time(0), which may produce similar sequences when multiple datasets are generated within the same second. The study evaluates MSVC's STL implementation only, and results may differ for GCC or Clang. Additionally, micro-benchmarks isolate operations individually and may not fully represent complex mixed- operation real-world workloads.

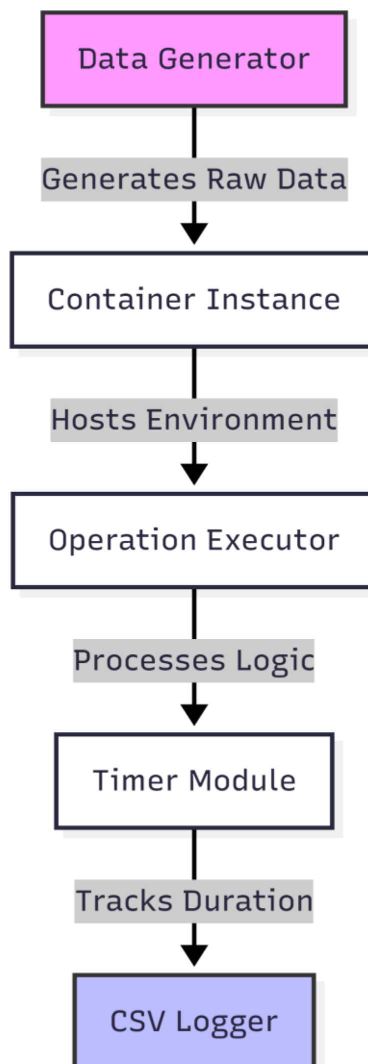
IV.IMPLEMENTATION / EXPIREMENTAL WORK

The benchmark framework was implemented in C++17 with a modular structure separating data generation, container operations, timing, and result logging.

The implementation architecture can be summarized as follows: Category Variable Description Independent Variables Container Type vector, list, deque, set, unordered_set Operation Type insert, traverse, search, delete Data Size **1K, 10K, 100K, 500K.**

Build Configuration /O2 or /Od (analyzed separately) Dependent Variable Execution Time Measured in milliseconds Controlled Variables Compiler MSVC Standard C++17 Hardware Same Windows machine Repetitions **3 per test.**

4.1 Experiment Flowchart



4.1 Data Generation

For each dataset size n, a vector of random integers was generated using rand() % n. Separate key sets were generated for search operations (1,000 keys) and deletion operations (n/2 keys).

4.2 Operation Implementation

Insertion for sequence containers (vector, list, deque) was performed using push_back. For set and unordered_set, insertion used insert. Traversal used range-based for loops to iterate over all elements. Search operations differed by container type: For vector, list, and deque, std::find was used, resulting in linear search behaviour. For set and unordered_set, the container's find() member function was used. Deletion for sequence containers involved locating elements using std::find followed by erase(iterator). For associative containers, erase(key) was used directly.

4.3 Timing Strategy

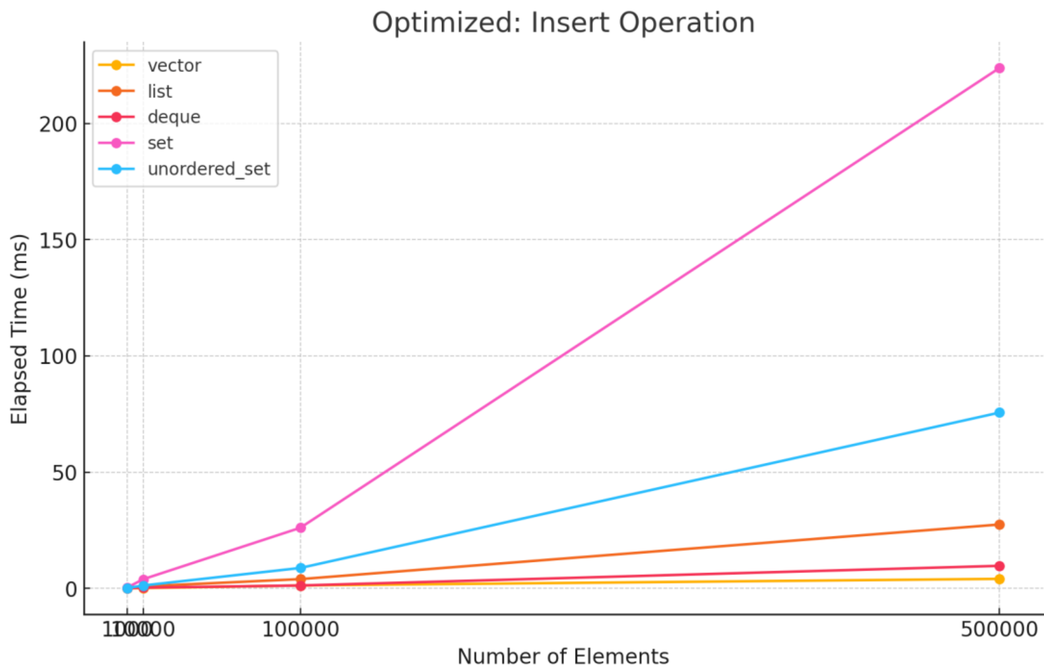
Each operation was wrapped with high_resolution_clock::now() calls. The duration was computed in milliseconds and written to a CSV file immediately after execution. Console output was minimized to avoid interfering with timing results. Each container was cleared after benchmarking to ensure independence between test cases.

V.RESULTS & ANALYSIS

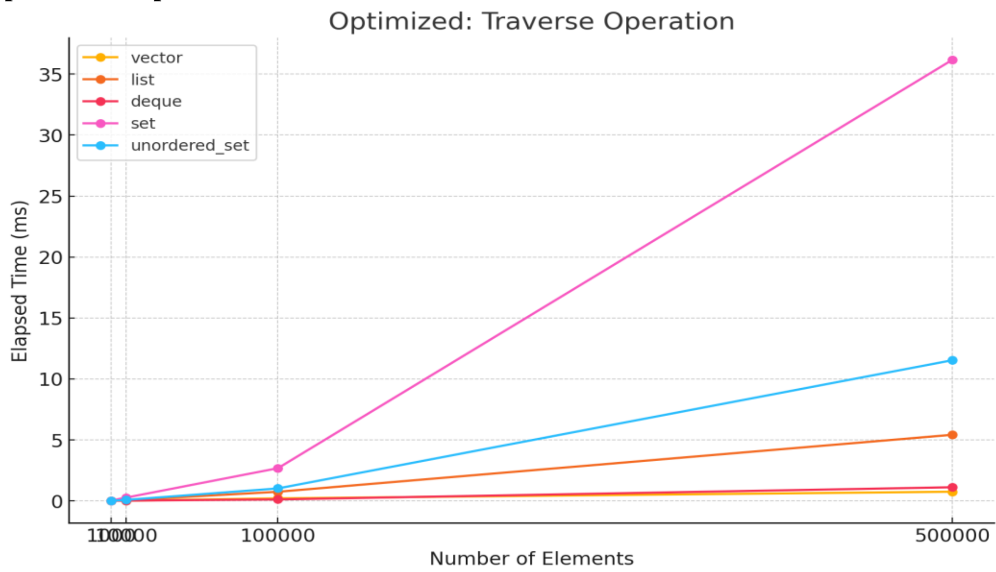
The experimental data was analyzed by aggregating runtimes and memory usage across repeated runs for each container, operation, size, and build configuration. Separate sets of charts were created for the optimized (/O2) and non-optimized (/Od) builds so that the impact of compiler optimization could be observed clearly. For each operation—insert, traverse, search, and delete—the charts plot average elapsed time in milliseconds against the number of elements, with one curve per container type. In the optimized build, std::vector generally exhibits excellent performance for insertions at the end of the sequence, traversal, and search when combined with std::binary_search or linear scans over contiguous memory. std::deque performs similarly in some cases but typically incurs slightly more overhead due to its segmented storage model. std::list, as expected, performs poorly for traversal-heavy workloads because pointer chasing prevents the CPU from fully exploiting cache locality. Associative containers show different behavior: std::set's tree-based implementation offers predictable logarithmic complexity but higher constant factors, while std::unordered_set often delivers near-constant-time lookups at the cost of increased memory usage. In the non-optimized build, the cost of additional checks, lack of inlining, and unoptimized loop structures is clearly visible. All containers become significantly slower, but the relative ordering of containers by performance remains broadly similar. This highlights an important practical lesson: while algorithmic and data structure choices matter, compiler optimization level can easily introduce an order-of-magnitude difference in performance, and benchmarks must be interpreted in the context of the build configuration.

5.1 Optimized Results (/O2)

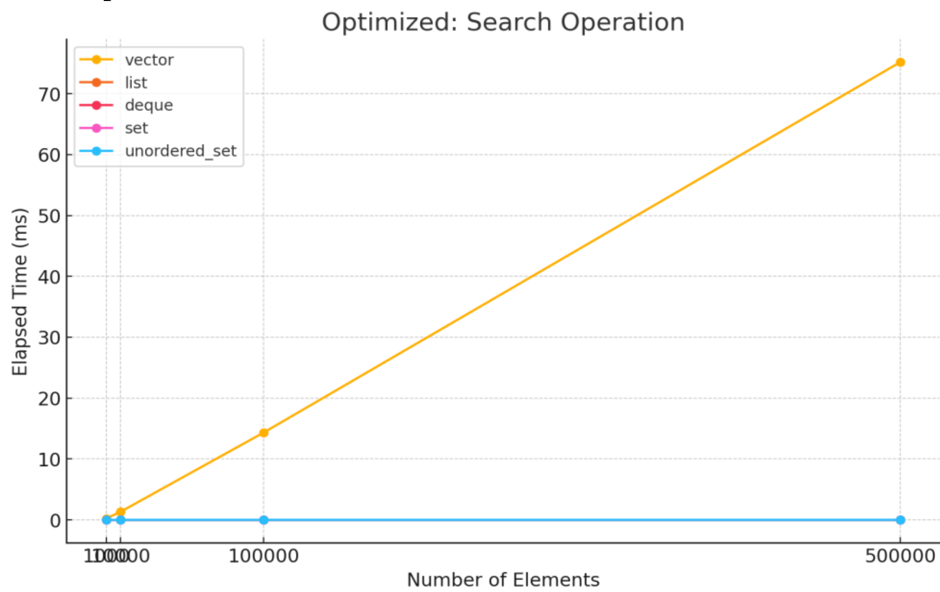
5.1.1 Insert Operation Graph



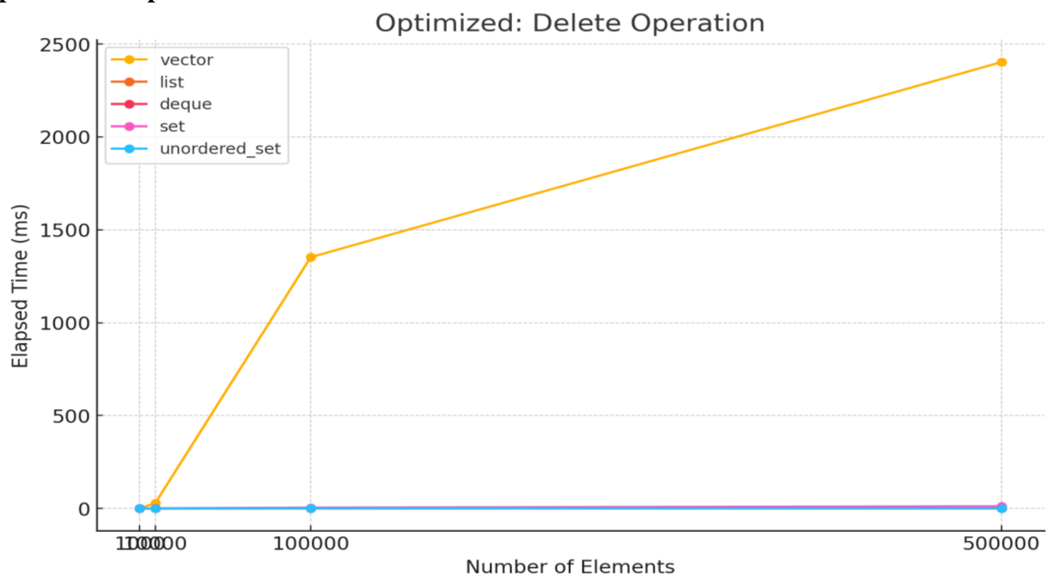
5.1.2 Traverse Operation Graph



5.1.3 Search Operation Graph

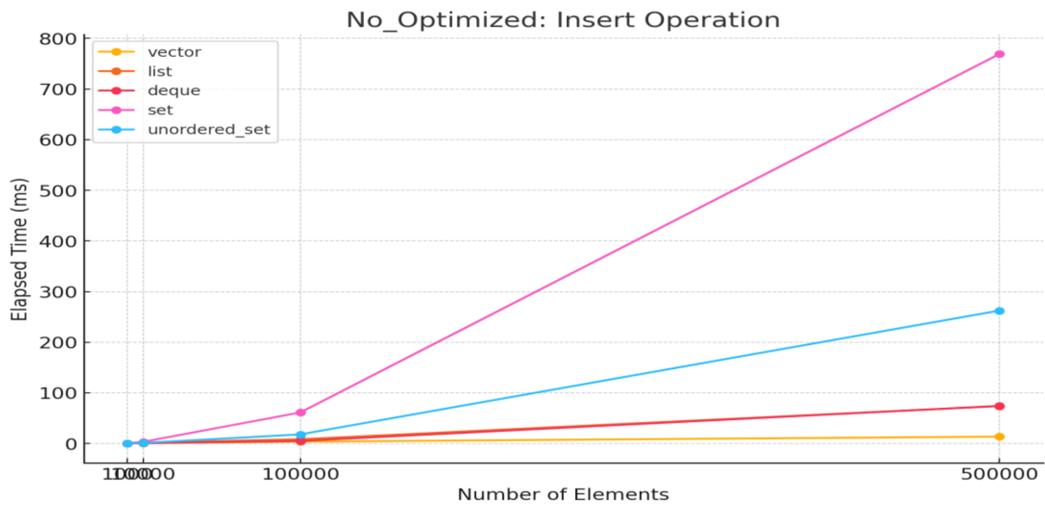


5.1.4 Delete Operation Graph

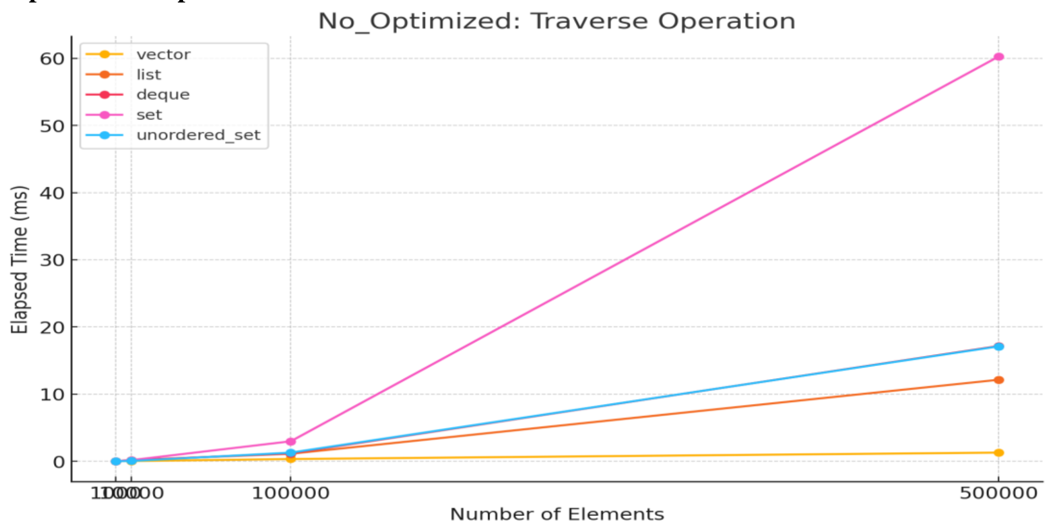


5.2 Non-Optimized Results (/Od)

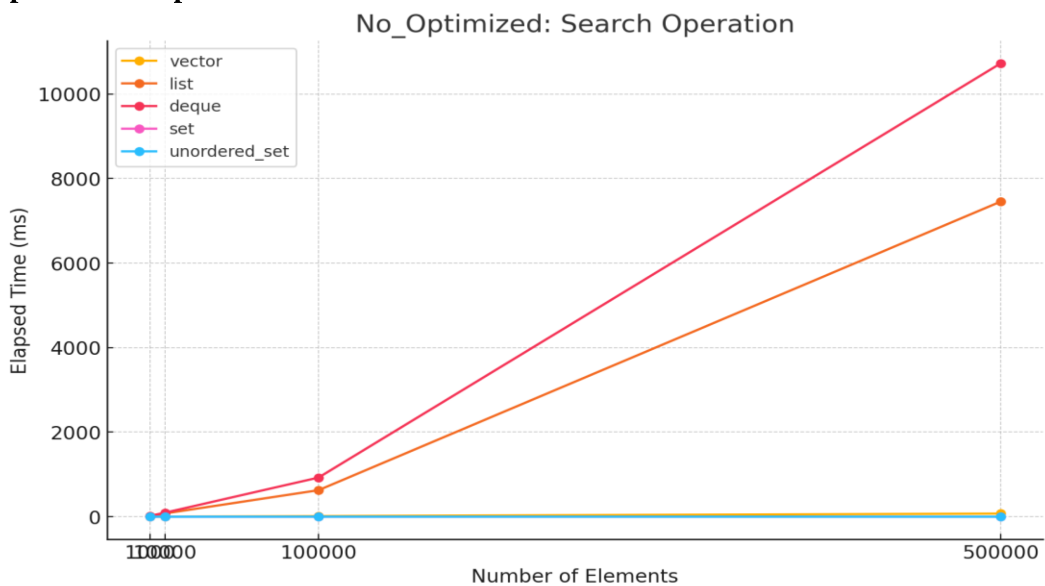
5.2.1 Insert Operation Graph



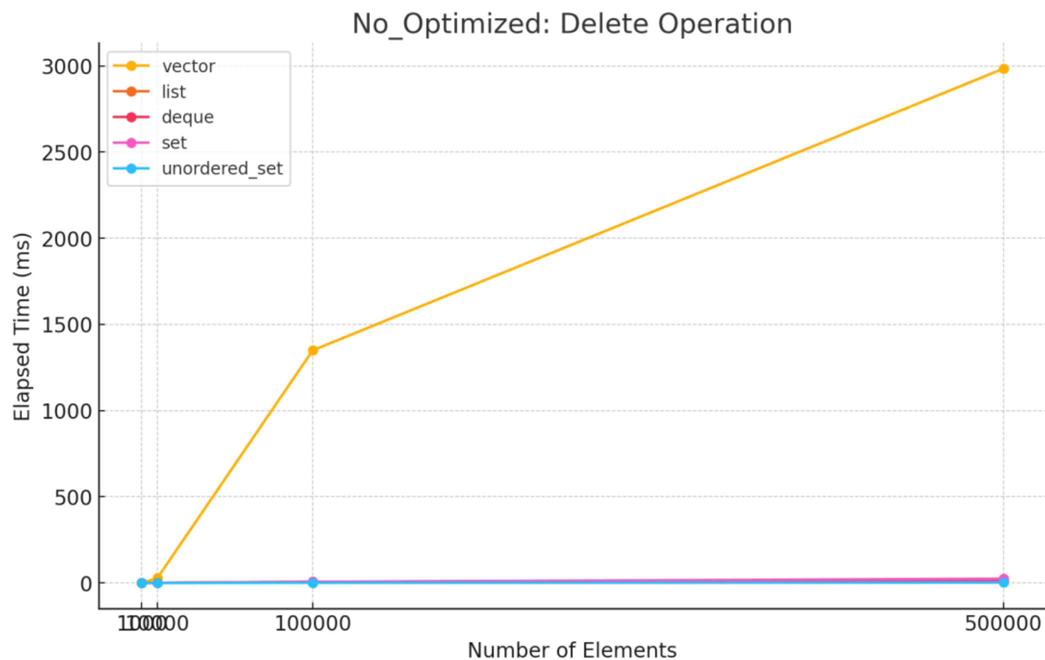
5.2.2 Traverse Operation Graph



5.2.3 Search Operation Graph



5.2.4 Delete Operation Graph



VI.CONCLUSION & FUTURE WORK

This empirical study of STL container performance on the Windows/MSVC platform confirms and refines many widely held intuitions about container behavior. std::vector, with its contiguous memory layout, remains the default choice for many workloads due to its excellent traversal performance and relatively low insertion cost when appending. std::deque provides a good compromise when insertions and removals at both ends of the sequence are required, though it typically does not outperform vector in pure traversal. Node-based containers such as std::list and tree-based associative containers such as std::set incur higher overheads due to pointer indirection and more complex internal structures. Although their asymptotic complexities are attractive for certain operations, these containers should be chosen only when their specific strengths—such as stable iterators or ordered traversal—are essential. std::unordered_set demonstrates strong performance for membership tests and lookups, making it a good fit for hash-table-style workloads, but its higher memory footprint must be considered, especially in memory-constrained environments. The comparison between optimized and non-optimized builds further reinforces the necessity of benchmarking and deploying performance-critical applications with compiler optimizations enabled. In many cases, the /O2 build reduced runtimes dramatically compared to the /Od build, without changing the underlying container or algorithm. Developers should therefore treat both container choice and build configuration as first-class parameters in performance tuning. Future work could extend this study in several directions. One possibility is to include additional compilers such as GCC and Clang on Windows to compare their STL implementations against MSVC. Another avenue is to evaluate the impact of custom allocators and memory pools on both performance and memory usage. Finally, the benchmark framework could be expanded to cover multi-threaded scenarios, parallel algorithms, or integration with alternative container libraries such as Boost or ETL in order to build a more comprehensive picture of container performance across modern C++ ecosystems.

REFERENCES

1. Laso, R., Krupitza, D., & Hunold, S. (2024). Exploring Scalability in C++ Parallel STL Implementations. In Proceedings of the 53rd International Conference on Parallel Processing (ICPP ’24). https://doi.org/10.1145/3673038.3673065
2. Pataki, N. (2011). C++ Standard Template Library by template specialized containers. arXiv:1111.3673. https://doi.org/10.48550/arXiv.1111.3673
3. Schmidt, D. C. (2014). The C++ Standard Template Library. Vanderbilt University. https://www.dre.vanderbilt.edu/~schmidt/PDF/stl.pdf
4. DeLozier, C. GPU Acceleration for the C++ Standard Template Library. University of Pennsylvania. https://www.seas.upenn.edu/~delozier/docs/libcxxgpu.pdf
5. McDonald, J., & Strooper, P. (1998). Programmatic Testing of the Standard Template Library: A Case Study. https://www.cs.ubc.ca/labs/isd/FormalWare/abstracts/progra~1.pdf
6. Qin, X., et al. (2022). Primrose: Selecting Container Data Types by Their Properties. arXiv:2205.09655. https://arxiv.org/pdf/2205.09655
7. Schuessler, C., & Gruber, R. (2016). A Traversable Fixed Size Small Object Allocator in C++. arXiv:1611.01667. https://arxiv.org/abs/1611.01667
8. Game AI Pro Online Edition. (2021). Should STL containers be used in game engines? https://www.gameaipro.com/GameAIProOnlineEdition2021/GameAIProOnlineEdition2021_Chapter15_Should_STL_containers_be_used_in_game_engines.pdf