Project Title	Implementing Dual Pivot Quicksort in C++23 and Comparing its Performance with Standard Sorting Libraries
Name & Student ID	LIN Zhanzhi 22097456D
Programme-Stream	61435-FCS BSc(HONS) COMPUTER SCIENCE
Project Supervisor	CAO Yixin
Date	24th October, 2025

Table of Content

Background and Problem Statement

Sorting remains a cornerstone operation in computer science, forming the foundation for efficient data manipulation, information retrieval, and system optimization. The quicksort algorithm, pioneered by C. A. R. Hoare in 1962, became a fundamental sorting method due to its simplicity and high average performance [4]. Over the decades, continuous refinements have aimed to improve both its asymptotic and empirical efficiency under varying computational architectures.

A major advancement occurred with Yaroslavskiy's Dual-Pivot Quicksort, introduced in 2009 [1]. Unlike the traditional single-pivot approach that splits a dataset into two partitions, the dual-pivot version employs two pivots to divide the data into three regions in a single pass. This method significantly reduces comparisons and enhances cache locality by lowering the number of memory accesses during partitioning. Consequently, it demonstrated superior average-case performance and became the default array-sorting algorithm in Oracle's Java 7 runtime environment, replacing the legacy quicksort variant [5].

Further theoretical studies elaborated on this innovation. Aumüller and Dietzfelbinger developed a rigorous mathematical framework for analyzing dual-pivot quicksort, uncovering an average comparison cost of approximately $1.8n \ln n$, outperforming the classical quicksort's $2n \ln n$ bound [2]. Wild complemented this theoretical foundation by introducing the scanned elements model, which more accurately captures performance characteristics on modern architectures by correlating algorithmic efficiency with cache hierarchy and memory bandwidth utilization [3]. These analyses collectively validate that reduced cache traversal, rather than mere computational optimizations, underpins the superior practical runtime of dual-pivot schemes.

Despite these advancements, modern C++ standard libraries still predominantly employ Introsort, a hybrid algorithm combining quicksort, heapsort, and insertion sort, emphasizing worst-case complexity guarantees and predictable runtime across data patterns [6]. While Introsort provides robust performance, it does not exploit the potential memory-system advantages illuminated by recent dual-pivot studies. Moreover, practical implementations of adaptive sorting algorithms in

C++, such as Pattern-Defeating Quicksort (PDQSort), achieve optimization through heuristic pivot sampling but do not fundamentally replicate the dual-pivot partitioning mechanism.

This research identifies a critical gap: the absence of a systematic C++ implementation and evaluation of dual-pivot quicksort under contemporary hardware and compiler settings. Investigating whether the theoretical and Java-based performance benefits can extend to C++—which emphasizes low-level memory management and template-based generic programming—presents both an engineering challenge and an academic opportunity to enhance the efficiency of core computational routines.

Objectives and Outcome

This research aims to integrate algorithmic theory with modern C++ software engineering practices to assess the practical viability of dual-pivot quicksort in high-performance computing environments. The following objectives are designed to ensure both theoretical depth and empirical rigor.

Implementation Objective

Develop a robust, template-based implementation of Dual Pivot Quicksort in C++23 that supports generic data types and comparator functions, leveraging the newest standard language and library enhancements for improved efficiency and scalability. The implementation will emphasize modular design to allow controlled tuning of parameters such as pivot selection strategies, recursion depth limits, and swap thresholds across diverse partitioning models.

Comparative Analysis Objective

Conduct systematic benchmarking of the developed dual-pivot algorithm against widely used C++ standard sorting methods and contemporary open-source algorithms, including:

- std::sort (Introsort) [7]
- std::stable_sort (Merge Sort) [8]
- std::partial_sort (Heapsort-based) [9]
- PDQSort (Pattern-Defeating Quicksort) [10]

This analysis seeks to evaluate both asymptotic performance and real-world efficiency across different input characteristics. The comparison framework will measure execution time, memory consumption, and branching behavior, following the analytical frameworks proposed by Wild and the cache-behavioral modeling techniques discussed in Nebel and Wild's multi-pivot studies [11] [12] [13].

Analytical Objectives

- Empirical Complexity Evaluation: Quantify runtime behavior under randomized, nearly sorted, reversed, and adversarial input distributions, establishing the empirical time complexity and variance for each case.
- Cache and Memory Analysis: Analyze L1/L2 cache performance and data locality through hardware performance counters, referencing the scanned elements perspective introduced by Wild [3].
- Pivot Sampling Impact: Investigate how various pivot sampling strategies—such as median-of-three, tertiles-of-five, and random dual-pivot selection—affect balance, stability, and partitioning efficiency, leveraging the partitioning optimization framework described by Aumüller and Dietzfelbinger. [2]
- Theoretical Model Derivation: Derive analytical bounds on key comparisons, swap operations, and memory accesses using recurrence-based cost models, extending the mathematical treatments in Yaroslavskiy's and Aumüller's work. [1] [14]

Outcome Objective

Formulate a set of practical recommendations for integrating optimized dual-pivot sorting strategies into the C++ Standard Template Library (STL) or modern performance-oriented libraries. These recommendations will bridge theoretical analysis and implementation outcomes, offering guidance for adaptive hybrid sorting algorithm designs. The results are expected to highlight the algorithm's trade-offs in runtime behavior, cache optimization, and stability, contributing both academically to algorithmic research and practically to C++ systems development.

Project Methodology

The proposed project follows a three-phase methodology involving algorithm design, experimental validation, and analytical modeling, all guided by established computational methods and theoretical models. Each phase ensures that the implementation and evaluation of the Dual Pivot Quicksort (DPQS) algorithm in C++23 is rigorous, measurable, and reproducible.

Implementation Phase

The implementation phase focuses on building a high-performance and flexible Dual Pivot Quicksort using modern C++23 features. This phase involves:

- Language Choice and Rationale
  - C++23 is selected for its expanded compile-time support (constexpr functions) and the std::ranges library, which simplifies algorithmic pipelines and reduces intermediate memory allocations [15]. These features offer measurable performance gains in sorting operations and improve memory locality on cache-sensitive architectures.
- Algorithm Development

  Two implementations will be created and tested:
  - Yaroslavskiy's Asymmetric Partitioning Variant (2009): Uses two carefully chosen pivots to partition data into three subarrays in one scan for balanced recursion [2].
  - Adaptive Dual Pivot Model: Adjusts pivot selection dynamically based on sampled medians, optimizing partition stability and reducing branch mispredictions on skewed data [4].
- Technical Design Goals
  - Template-based implementation supporting any comparable data type (int, double, std::string, and user-defined types).
  - Parameterized thresholds for recursion depth, pivot sampling size, and insertion-sort cutoffs for small subarrays.
  - Full compliance with C++23 iteration and memory models to avoid undefined behavior

or unnecessary copying.

Comparative Analysis Phase

The second phase validates the implemented algorithm through extensive empirical testing against standard and modern C++ sorting algorithms under identical conditions.

- Benchmark Setup
  - Input distributions

    Four dataset configurations will be used to test partitioning adaptability and stability

    1. Uniformly random data
    2. Nearly sorted data
    3. Reverse-sorted data
    4. Repetitive or duplicate-heavy data
  - Input sizes

    Datasets ranging from 10^3 to 10^8 elements will be evaluated to observe both CPU-bound and memory-bound behavior under realistic workloads.
  - Comparison Baselines

    The implemented algorithm will be benchmarked against multiple sorting algorithms for statistical consistency and performance analysis:

    - std::sort (Introsort) [7]
    - std::stable_sort (Merge Sort) [8]
    - std::partial_sort (Heapsort-based) [9]
    - PDQSort (Pattern-Defeating Quicksort) [10]
- Execution Environments

  The tests will be executed on two systems representing distinct design philosophies—Apple's ARM-based architecture with unified memory and Intel's performance-focused hybrid design—for comprehensive cross-platform benchmarking [16] [17].

  1. Apple MacBook Air (M2, 2022)
     - 10 core CPU (8 performance + 2 efficiency cores)

    - 8 core integrated GPU
    - 16 GB unified LPDDR5 memory
    - Operating System: macOS Sequoia 15.0
    - Compiler: Apple Clang with C++23 standard support

    This environment provides insights into energy efficiency, thermal scaling, and ARM-based memory coherence effects during partitioning-heavy sorting workloads.

2. Desktop PC
    - Intel Core i5-12600KF, 2023. 10 cores / 16 threads (6P + 4E architecture, Alder Lake)
    - Motherboard: Colorful B760M AYW WIFI D5
    - RAM: Yingchi DasPro Armor DDR5 32 GB (2×16 GB, 6000 MHz)
    - Operating System: Windows 11 Pro (with WSL2 Ubuntu for Linux-based testing)
    - Compilers: MSVC 19.39 and GCC 14.2 (C++23 mode)

    This setup provides high-frequency multi-core benchmarks representative of modern x86 desktop systems, useful for analyzing cache interactions, thread parallelism, and branch prediction impact.

- Performance Metrics

  Both systems will collect and compare the following benchmark data:

  ■ Execution Time: Wall clock duration using precise timing mechanisms (std::chrono high resolution clocks) [18].

  ■ Operation Count: Number of element comparisons and swap operations.

  ■ Hardware Counters: L1/L2 cache misses, branch mispredictions, and instruction throughput measured via PAPI and Apple Instruments [18], [19].

  ■ Scalability: Speedup and efficiency across varying data volumes and architectures.

  By profiling and cross-analyzing these results, this phase aims to quantify how architectural differences — particularly unified versus discrete memory and big.LITTLE configurations — influence the runtime behavior of Dual Pivot Quicksort in real-world C++23 implementations.

Analytical Phase

The analytical phase aims to connect theoretical performance predictions with experimental data.

- Analytical Model

  Average running time will be evaluated using the recurrence:

  $$T(n) = an \ln n + bn + O(1),$$

  Where a represents the leading coefficient reflecting per-comparison cost. Experimental values of a and b will be estimated using regression across benchmark data and compared to predictions from Aumüller and Dietzfelbinger [4].

- Pivot Strategy Evaluation

  Analyze the performance of:

  - Median of Three Sampling (classical heuristic).
  - Tertiles of Five Sampling (used in Java 7’s dual-pivot version).
  - Randomized Sampling (for robustness against structured input).

  Each strategy will be tested for consistency, variance reduction, and execution stability under large data volumes.

- Complexity and Cache Behavior

  Integrate Wild’s scanned-elements model to quantify the correlation between memory traversal and total sorting time. This model will help isolate cache-related effects beyond mere comparison counts [5].

Literature Review

The literature review provides context and theoretical grounding for the project’s chosen approach.

Foundational Studies:

- Hoare (1962): Established the original quicksort algorithm as a recursive divide-and-conquer framework [1].
- Yaroslavskiy (2009): Introduced a dual pivot design achieving superior performance through asymmetric partitioning and reduced memory scans [2].

- Aumüller & Dietzfelbinger (2015): Provided the mathematical proof of near optimal key comparison cost ($1.8n \ln n$) [4].
- Wild (2016): Linked performance gains to cache-efficient element scanning via the memory wall theory [5].

Modern Sorting in C++:

- Musser (1997): Developed Introsort, the foundation of std::sort, ensuring worst case performance of $O(n \log n)$ through hybridization [21].
- Peters (2021): Proposed PDQSort with adaptive pivot reselection for modern adversarial workloads [22].

Project Schedule

Phase	Duration	Key Activities	Deliverables
Literature Review	24 Oct 2025 – 8 Nov 2025	Review foundational and modern sorting algorithms, dual-pivot quicksort theory, and C++23 enhancements.	Literature summary for interim report
Implementation	9 Nov 2025 – 13 Dec 2025	Develop and test Dual Pivot Quicksort in C++23; modular and template-based, with version control.	Source code and documentation
Benchmarking	14 Dec 2025 – 5 Jan 2026	Conduct experimental tests on all datasets and platforms; gather empirical performance data for analysis.	Benchmark results and raw data
Analysis & Interim Reporting	6 Jan 2026 –	Analyze performance, compare with theory, prepare interim report	Interim report and presentation video

	9 Jan 2026	and presentation video for submission on 9 Jan 2026.	
Further Analysis & Refinement	10 Jan 2026 – 5 Apr 2026	In-depth analysis, code refinement, additional experiments as needed; address feedback from interim review.	Finalized analysis
Final Reporting	6 Apr 2026 – 10 Apr 2026	Compile results, document methods and findings, complete final report for submission on 10 Apr 2026.	Final project report
Final Presentation Preparation & Delivery	11 Apr 2026 – 22 Apr 2026	Prepare slides, rehearse defense, deliver final presentation during 18, 20–22 Apr 2026 windows.	Final presentation slides and defense

Submission and Presentation Milestones

- Project Proposal Submission: 24 Oct 2025
- Interim Report & Presentation Video: 9 Jan 2026
- Final Report Submission: 10 Apr 2026
- Final Presentation: 18, 20–22 Apr 2026

Resources Estimation

Hardware

- Apple MacBook Air (M2, 2022)
    - 10-core CPU (8 performance + 2 efficiency cores)
    - 8-core integrated GPU
    - 16 GB unified LPDDR5 memory
    - 256 GB SSD storage
    - Operating System: macOS Sequoia 15.0

    - Compiler: Apple Clang with full C++23 support

  Rationale: This platform provides insights into energy efficiency, ARM-based unified memory behavior, and modern macOS development environments, making it ideal for evaluating cross-platform algorithm performance, particularly in memory-bound tasks.

- Desktop PC
    - Intel Core i5 12600KF (10 cores / 16 threads, Alder Lake: 6 performance + 4 efficiency cores)
    - Motherboard: Colorful B760M-AYW WIFI D5
    - RAM: Yingchi DasPro Armor DDR5 32 GB (2×16 GB, 6000 MHz)
    - Storage: NVMe SSD
    - Operating System: Windows 11 Pro (with WSL2 Ubuntu for cross-comparison)
    - Compilers: MSVC 19.39 and GCC 14.2 (C++23 mode)

  Rationale: This x86-64 system provides a modern, high-performance baseline for scalability testing, cache behavior analysis, and compatibility with both major operating systems.

Software

- Operating Systems
    - macOS Sequoia 15.0
    - Windows 11 Pro (and WSL2 Ubuntu 24.04 LTS)
- Compilers and Development Environments
    - Apple Clang (C++23 mode) [23]
    - GCC 14.2 (C++23) [24]
    - MSVC 19.39 (C++23)
    - CMake/Ninja for build automation [25]
- Benchmarking and Profiling Tools:
-

References

https://www.perplexity.ai/search/according-to-the-description-r-tLlAjPitSpCsFLDSWqZvcA

https://www.perplexity.ai/search/turn-what-ai-generated-writing-dbQji7MdQ46yafL_XkXIBg

https://www.perplexity.ai/search/turn-what-ai-generated-writing-dbQji7MdQ46yafL_XkXIBg

https://www.perplexity.ai/search/find-official-documentation-of-j0LSUnb_TC6512ykjVYc9g