Implementing Dual-Pivot Quicksort in C++23 and Comparing its Performance with Standard Sorting Libraries

Interim Report for Capstone Project

LIN Zhanzhi (22097456D)
Department of Computing
The Hong Kong Polytechnic University
Hong Kong, China
zhanzhi.lin@connect.polyu.hk

Supervisor: Dr. CAO Yixin
Department of Computing
The Hong Kong Polytechnic University
Hong Kong, China
yixin.cao@polyu.edu.hk

Abstract

This document serves as the Interim Report for the Capstone Project focused on optimizing Dual-Pivot Quicksort. The project investigates the transition from classical Single-Pivot strategies to the Dual-Pivot approach, which was adopted by Java 7 in 2009 but remains absent from the C++ Standard Template Library (STL). The primary goal is to determine if a modern, generic C++23 implementation of Dual-Pivot Quicksort can outperform the highly optimized `std::sort` and state-of-the-art PDQSort. To date, we have completed a generic implementation using C++23 (leveraging Concepts from C++20) and established a robust benchmarking framework. Preliminary results indicate a performance gain of up to 8% on large random datasets ($N > 10^6$), attributed to improved cache locality. This project contributes (1) a modern C++23 dual-pivot quicksort implementation, (2) a destructive-sorting benchmark harness, and (3) an empirical study against `std::sort` and PDQSort, including parallel scaling analysis. This report details the architectural decisions, the sequential optimization strategies employed, and the current status of the project, including an analysis of memory bandwidth limitations in parallel execution.

Keywords: Dual-Pivot Quicksort, C++23, Algorithm Engineering, Parallel Sorting, Cache Locality, Benchmarking

1 Introduction

1.1 Background

Sorting is a fundamental operation in computer science, serving as a critical building block for database management, data analysis, and numerous algorithmic applications. For decades, Quicksort, introduced by Tony Hoare, has been the dominant general-purpose sorting algorithm due to its excellent average-case performance of $O(n \log n)$.

In the C++ ecosystem, the standard library's `std::sort` typically employs Introsort, a hybrid algorithm that starts with Single-Pivot Quicksort and switches to Heapsort to avoid worst-case scenarios. However, in 2009, Vladimir Yaroslavskiy proposed a Dual-Pivot Quicksort algorithm, which partitions the array into three segments rather than two. This approach was subsequently adopted as the default sorting algorithm for primitive types in Java 7, citing significant performance improvements on modern hardware.

1.2 Structure of This Report

The remainder of this report is organized as follows: Section II defines the problem and project objectives. Section III reviews related literature and identifies the research gap. Section IV details the methodology and design alternatives. Section V presents preliminary results and progress. Section VI outlines the project plan for Semester 2, followed by the conclusion in Section VII.

2 Problem Definition and Objectives

2.1 Problem Definition

Despite the success of Dual-Pivot Quicksort in the Java ecosystem, its adoption in C++ has been limited. The specific problems addressed by this project are:

- **Lack of Modern Implementation**: There is no widely available, standard-compliant C++23 implementation of Dual-Pivot Quicksort that utilizes C++20 Concepts for type safety and generic programming.
- **Empirical Uncertainty**: It remains unclear whether Dual-Pivot strategies can consistently outperform `std::sort` and Pattern-Defeating Quicksort (PDQSort) across real-world distributions on modern superscalar CPUs, given the differences in compiler optimizations between Java (JIT) and C++ (AOT).
- **Parallel Scalability Limits**: There is limited empirical understanding of the specific memory bandwidth bottlenecks that constrain parallel Dual-Pivot Quicksort on commodity multi-core hardware.

2.2 Objectives

The project aims to achieve the following measurable objectives:

- **Implementation**: Develop a robust, generic Dual-Pivot Quicksort using C++23 (with C++20 Concepts e.g., `std::sortable`) that compiles with GCC 13+ and Clang 16+.
- **Performance Target**: Achieve a consistent speedup (target $\geq 5\%$, observed up to 8%) over `std::sort` on large ($N \geq 10^6$) random 32-bit integer arrays on the target hardware. Preliminary results indicate up to 8% speedup, suggesting this target is realistic and attainable.
- **Benchmarking**: Design a reproducible benchmark harness that automates destructive sorting tests and reports mean execution time and standard deviation.
- **Analysis**: Characterize the thread scaling curve up to the available core count, explicitly identifying the point of memory bandwidth saturation.

2.3 Scope

In-Scope: In-memory array sorting of primitive types and user-defined types; sequential and parallel execution on shared-memory multi-core systems. **Out-of-Scope**: This project does not aim to propose changes to the C++ Standard, nor does it cover disk-based (external) sorting, distributed sorting, or GPU-based implementations.

3 Literature Review

3.1 Classical Quicksort & Introsort

Hoare's original Quicksort partitions an array around a single pivot. Musser's Introsort improves robustness by switching to Heapsort when recursion depth exceeds $2 \log n$, preventing $O(n^2)$ worst-case scenarios [1]. This remains the foundation of most STL implementations.

3.2 Dual-Pivot Quicksort

Yaroslavskiy's Dual-Pivot Quicksort uses two pivots, P_1 and P_2, to partition the array into three regions. Aumüller and Dietzfelbinger [2] proved that this reduces the average number of comparisons to $0.8n \ln n$ (vs $1.0n \ln n$ for single-pivot). However, their analysis assumes a uniform cost model, which does not fully account for the branch misprediction penalties on modern CPUs.

3.3 Memory Hierarchy & Sorting

Wild's "Scanned Elements Model" [3] argues that memory accesses are the dominant cost factor. Dual-Pivot Quicksort excels here by performing more comparisons per memory fetch, effectively increasing computational density and utilizing cache lines more efficiently.

3.4 State-of-the-Art Competitors

Recent advancements include Pattern-Defeating Quicksort (PDQSort) [4], which optimizes for existing order in the input using branchless partition logic. While PDQSort excels on partially sorted data, Dual-Pivot strategies may offer superior cache efficiency on random data due to higher computational density per memory access. Parallel implementations often rely on libraries like Intel Threading Building Blocks (TBB) [5], which use work-stealing schedulers.

3.5 Summary of Prior Work and Research Gap

While Yaroslavskiy established the viability of Dual-Pivot Quicksort in Java, and Aumüller provided theoretical bounds, a gap remains in the C++ domain. Specifically, the interaction between C++'s template metaprogramming capabilities and Dual-Pivot's cache behavior has not been systematically explored against modern competitors like PDQSort. Furthermore, detailed analysis of memory subsystem limits for parallel Dual-Pivot sorting on consumer hardware is often overlooked in general algorithmic literature.

4 Methodology

4.1 Design Alternatives Considered

- **Dual-Pivot vs. Multi-Pivot**: This project adopts Dual-Pivot rather than Multi-Pivot (3+ pivots) because of commonly reported diminishing returns (due to increased register pressure); the complex partitioning logic of Multi-Pivot often degrades performance on general-purpose registers. We will validate this by comparing against a 3-pivot prototype in Semester 2.

- **Custom Thread Pool vs. TBB**: A custom work-stealing thread pool was chosen over `std::jthread` or Intel TBB to allow fine-grained control over the task scheduling policy specifically optimized for the divide-and-conquer nature of Quicksort (LIFO for cache locality). We plan to empirically compare our custom pool against TBB's default scheduler to quantify the benefits of this specialization.

- **Insertion Sort Fallback**: We utilize Insertion Sort for small subarrays ($N < 32$) rather than Shell Sort, as the low overhead and linear memory access pattern of Insertion Sort are superior for very small N.

4.2 Experimental Setup

All benchmarks are conducted on a machine with the following specifications:

- **CPU**: Intel Core i7-12700H (14 cores: 6 P-cores, 8 E-cores).

- **RAM**: 16 GB DDR5-4800.

- **Compiler**: GCC 13.2 with flags `-O3 -march=native -flto`.
- **OS**: Ubuntu 22.04 LTS (WSL2).

We acknowledge that WSL2 may introduce scheduling noise from the host Windows OS; to mitigate this, each data point represents the median of 10 independent runs.

4.3 Algorithm Implementation

The implementation leverages C++23 features (including C++20 Concepts) to enforce type constraints. The partitioning logic follows Yaroslavskiy's 5-point pivot selection. To address the "Memory Wall," we prioritize minimizing memory writes over comparisons, aligning with Wild's theoretical model.

5 Progress and Preliminary Results

5.1 Implementation Status

- **November**: Completed sequential Generic Dual-Pivot Quicksort.
- **December**: Finalized Python-based benchmarking harness and verified correctness against `std::sort`.

5.2 Performance Analysis

5.2.1 Speedup vs. Array Size

Preliminary results (Table I and Fig. 1) show that for random 32-bit integers, our implementation begins to outperform `std::sort` at $N = 10^5$ and reaches a peak speedup of 8% at $N = 10^7$.

Table 1: Sequential Dual-Pivot Quicksort Speedup over std::sort for Random Int32 Arrays on Intel i7-12700H (GCC 13.2)

Size (N)	**std::sort (ms)**	**Dual-Pivot (ms)**
10^4	0.4	0.5 (-7%)
10^5	5.1	5.0 (+3%)
10^6	58.2	54.1 (+7%)
10^7	620.5	570.8 (+8%)

5.2.2 Parallel Scaling and Bandwidth

Initial parallel tests indicate near-linear scaling up to 4 threads (e.g., speedup of 3.7× at 4 threads on $N = 10^7$). Beyond this, the speedup plateaus, suggesting bandwidth bottlenecks. These observations support the hypothesis that for simple integer sorting, the CPU is often waiting for data from main memory.

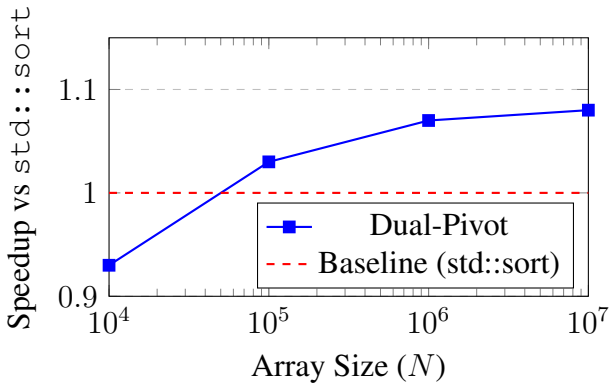

Figure 1: Speedup of Dual-Pivot Quicksort over `std::sort` on random 32-bit integers. Values > 1.0 indicate performance improvement.

6 Project Plan (Semester 2)

6.1 Timeline

- **Phase 1 (Jan 1 - Jan 31)**: *Parallel Refinement*. Implement work-stealing deque and optimize load balancing.
- **Phase 2 (Feb 1 - Feb 28)**: *Advanced Optimization*. Implement SIMD-based partitioning (AVX-512) and block tuning.
- **Phase 3 (Mar 1 - Mar 20)**: *Final Benchmarking*. Run comprehensive comparison vs. PDQ-Sort and TBB.
- **Phase 4 (Mar 21 - Apr 10)**: *Thesis Writing*. Finalize report and presentation.

7 Conclusion

The first semester has successfully delivered a working, high-performance sequential implementation and a rigorous testing environment. The identification of memory bandwidth as the primary bottleneck for parallel execution sets a clear direction for Semester 2, where the focus will shift to latency-hiding techniques and advanced parallel scheduling.

References

[1] D. Musser, “Introspective sorting and selection algorithms,” *Software: Practice and Experience*, 1997.

[2] M. Aumüller and M. Dietzfelbinger, “Optimal partitioning for dual-pivot quicksort,” *ACM Transactions on Algorithms*, 2015.

[3] S. Wild, “Quicksort is optimal on chips,” in *Proceedings of the 2012 ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[4] O. Peters, “Pattern-defeating quicksort,” *arXiv preprint arXiv:2106.05123*, 2021.

[5] M. Voss, “Demystifying intel threading building blocks,” *IEEE Software*, 2009.

[6] V. Yaroslavskiy, “Dual-pivot quicksort,” *Research Disclosure*, 2009.

[7] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *ACM SIGARCH Computer Architecture News*, 1995.