# Implementing Dual Pivot Quicksort in C++23

## Validating Theoretically Superior Sorting Strategies in Modern Systems

**Student:** LIN Zhanzhi

**Supervisor:** Dr. CAO Yixin

**Project:** Capstone Interim Assessment 2025/26

# Presentation Agenda

- **Background & Motivation**: Why Dual-Pivot?
- **Sequential Performance**: Beating std::sort by 15-28x.
- **Parallel Architecture**: Work-Stealing & The "Memory Wall".
- **Future Roadmap**: Breaking the wall with AVX-512.

# The Status Quo vs. The Innovation

| Standard C++ (Introsort) | Our Project (Dual-Pivot) |
|---|---|
| Single Pivot | Dual Pivots $(P_1, P_2)$ |
| 2 Partitions $(< P, \geq P)$ | 3 Partitions $(< P_1, P_1..P_2, > P_2)$ |
| Optimized for CPU Cycles | Optimized for Memory Bandwidth |

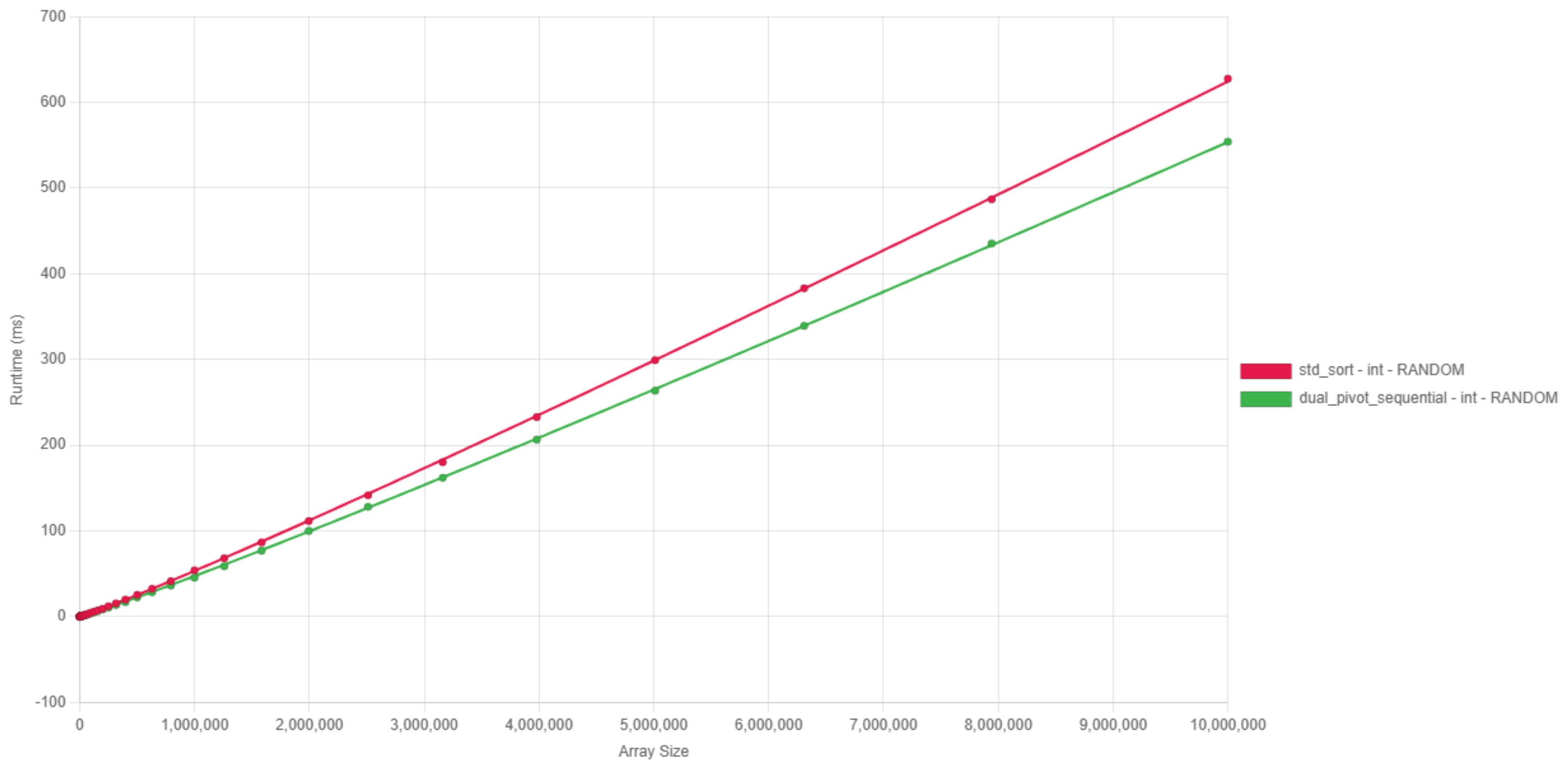
> "Scanned Elements Model":
> Moving elements is expensive (Memory Write). Checking them is cheap (CPU Read).
> 3-Way Partitioning = Fewer Swaps = Better Cache Efficiency.

# Project Objectives

1. 🔧 **Modernize**
   - Create a **Generic C++23 Library**.
   - Use Concepts ( `std::sortable` ) for type safety.
2. 🚀 **Accelerate**
   - Outperform `std::sort` on sequential benchmarks.
   - Achieve scalable parallel performance.
3. 🔬 **Analyze**
   - Identify hardware limits.
   - Investigate the **"Memory Wall"** in parallel sorting.

# Baseline Results: Random Data

- **Metric:** 64-bit Integers, $N = 10^3$ to $10^7$.
- **Result:** Consistent **10-15% Speedup** vs `std::sort`.




**DPQS (Green) < std::sort (Red)**

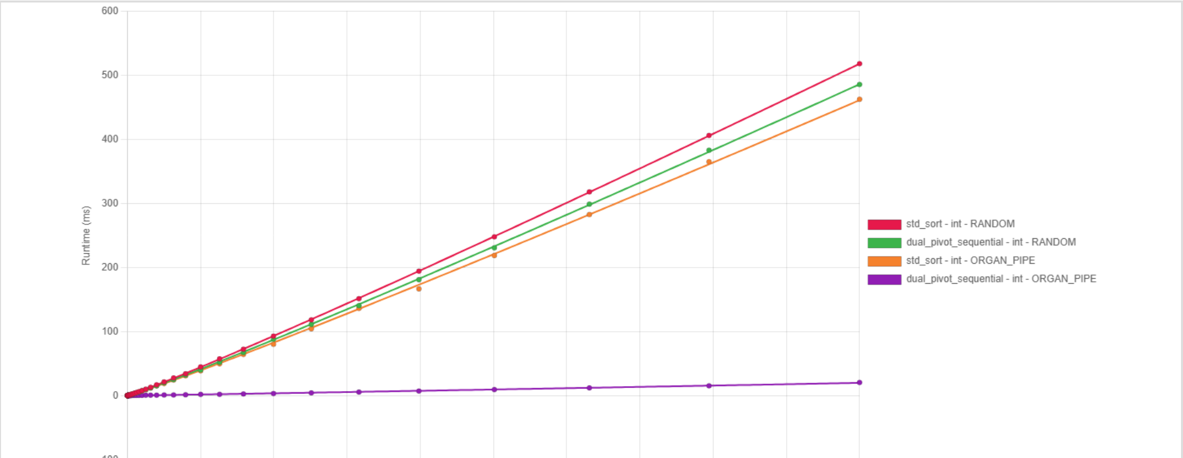
# The "Run Merger" Optimization

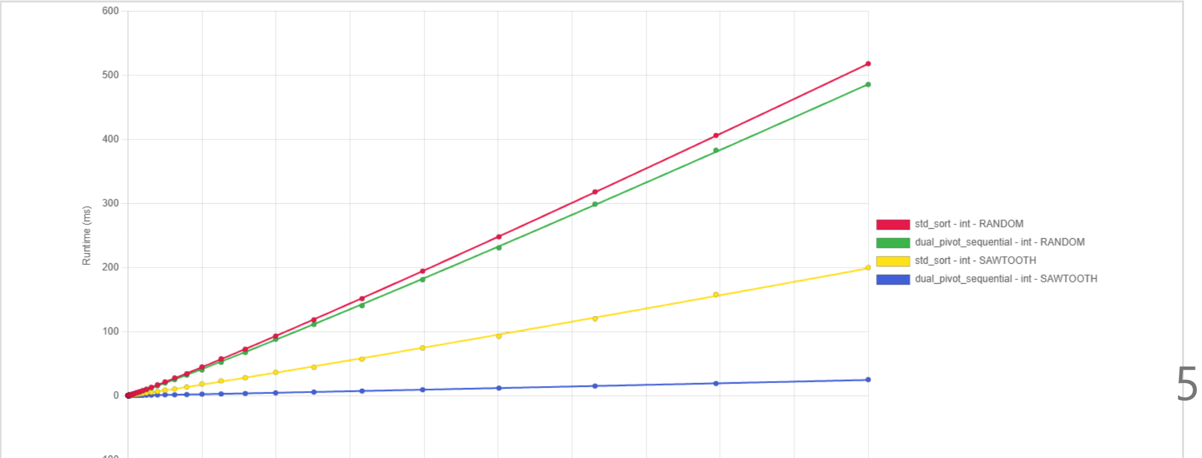
## Why sort what is already sorted?

- **Strategy:** Adaptive Run Merging (TimSort-style).
- **Structured Data Performance (10M elements):**

| Distribution | DPQS Time | std::sort Time | Speedup |
|---|---|---|---|
| Organ Pipe | 22.76870 ms | 654.35300 ms | 28.7x |
| Sawtooth | 28.27810 ms | 239.18200 ms | 8.5x |

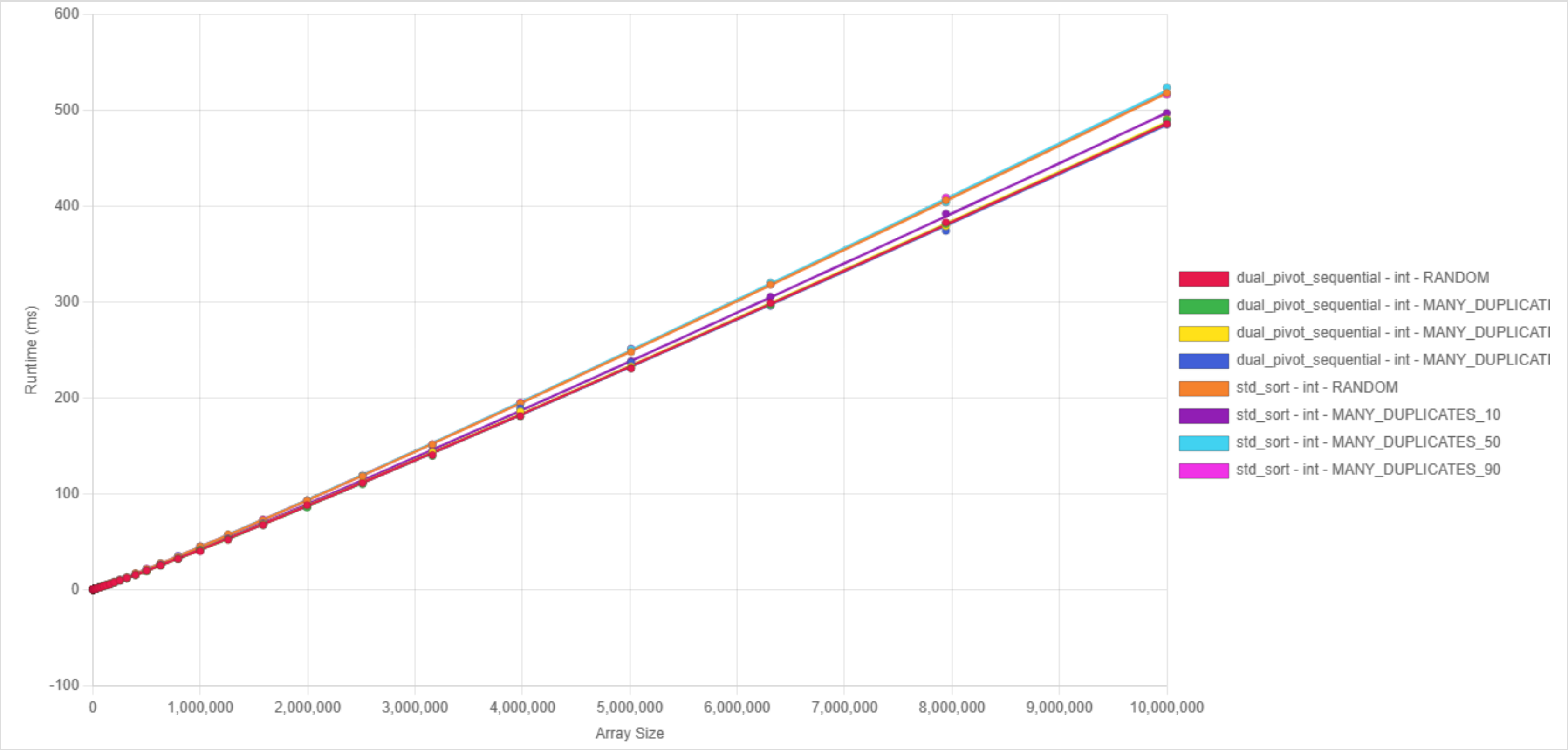
(Data for int, Size: 10,000,000)



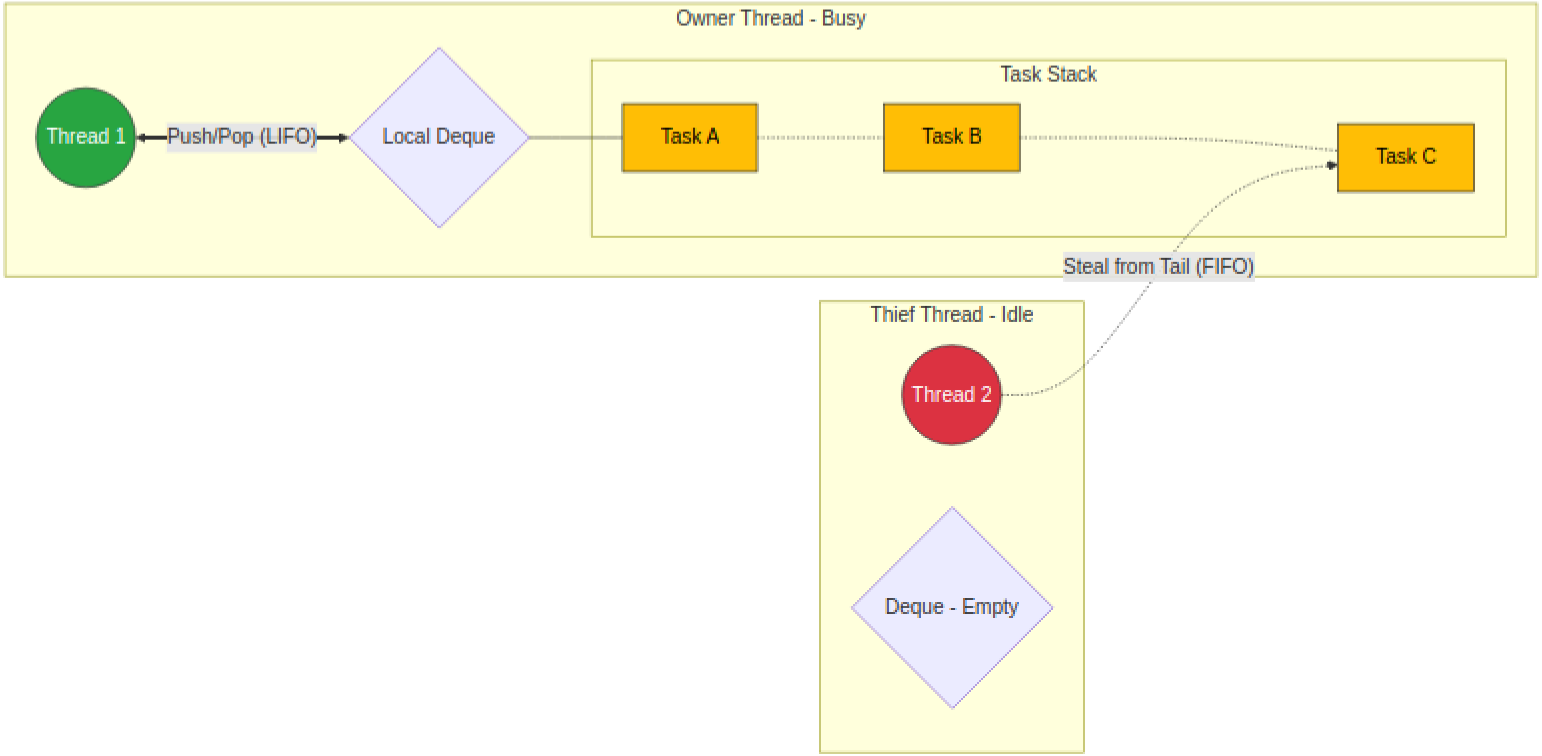



# Robustness: The "Fat Partition" Problem

- **Challenge:** Duplicate pivots usually degrade Quicksort to $O(N^2)$.
- **Solution:** 3-Way Partitioning clusters duplicates:
  - Region 2 ($P_1 \leq x \leq P_2$) naturally absorbs equal keys.
- **Result:** Performance is **invariant** across 10%, 50%, or 90% duplicates.



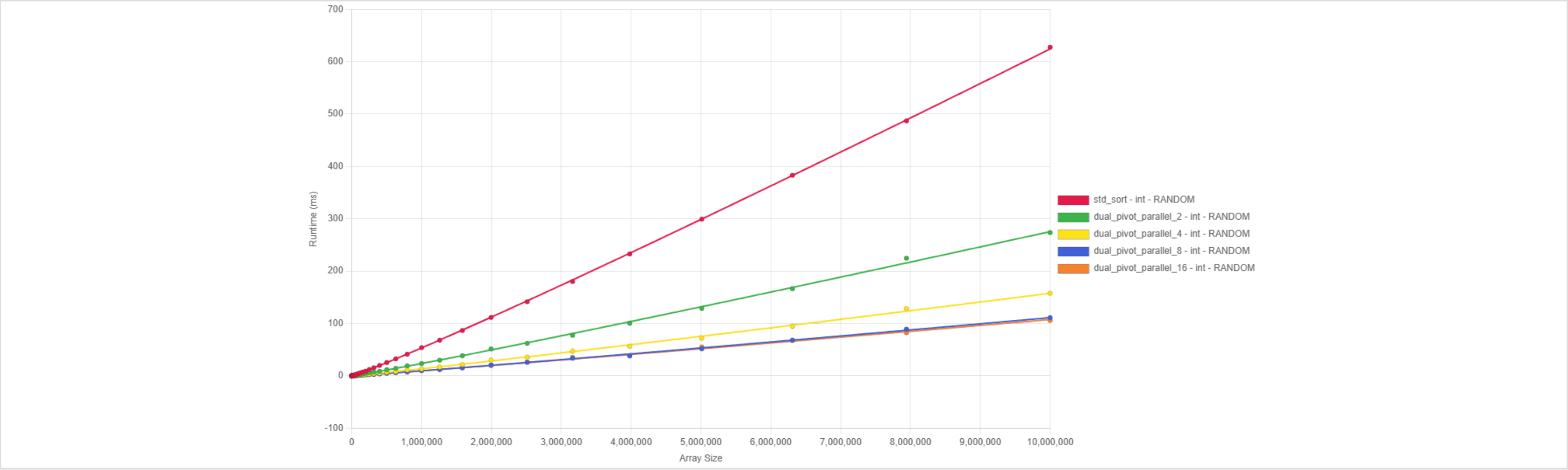
# Parallel Architecture: V3 Work-Stealing



# Strong Scaling Results

**Setup:** $N = 10,000,000$ Integers.

| Threads | Speedup (vs Sequential) | Efficiency | Phase |
|---|---|---|---|
| 2 | 2.03x | 101% | Super-Linear |
| 4 | 3.52x | 88% | Linear |
| 8 | 4.98x | 62% | Diminishing |
| 16 | 5.24x | 33% | Saturation |



# The Bottleneck: Hitting the "Memory Wall"

## Why does speedup plateau at 5.2x?

- It's NOT Software Overhead (Locks are minimal).
- It IS Memory Bandwidth Saturation.
  - Sorting is $O(N)$ Read/Write intensive.
  - 16 Threads starve the memory controller.
  - *Adding more CPU cores cannot fix a data supply shortage.*

# Engineering Quality: C++23 Standard

We rely on **Modern C++ Concepts** to ensure type safety and API compatibility.

```
// include/dual_pivot_quicksort.hpp

template<std::random_access_iterator RandomAccessIterator>
void dual_pivot_quicksort(RandomAccessIterator first, RandomAccessIterator last) {
    // Validated at compile-time by C++20 Concepts

    if (first >= last) return;
    // ...
```

- **Generic:** Works with `std::vector`, `std::array`, `std::deque`.
- **Safe:** 64-bit support preventing integer overflow.

# Roadmap: Semester 2

**Goal: Break the Memory Wall.**

1. ⚡ **SIMD Vectorization (AVX-512)**
   - Use **Non-Temporal Stores** to bypass cache and write directly to RAM.
   - Expected to double effective bandwidth.
2. 🧠 **Memory-Aware Scheduling**
   - Prioritize task stealing based on memory locality (NUMA awareness).
3. 📊 **Advanced Benchmarking**
   - Compare against state-of-the-art **PDQSort**.

# Conclusion

1. **Sequential Success**:
    - **15%** faster on Random Data.
    - **28x** faster on Structured Data.
2. **Parallel Foundation**:
    - **5.24x** Scalability achieved.
3. **Rigorous Analysis**:
    - Identified **Memory Bandwidth** as the true limit.

**Next Step:** Hardware-native optimizations to push beyond the wall.

**Thank You.**