High-Performance Divide-and-conquer Algorithms: A Comprehensive Framework with Recursive Parallel Decomposition and Hardware-aware Optimization

Tetsurou Kizaki [1], Dijana Capeska Bogatinoska [2,*], Amita Nandal [3], and Aleksandar Karadimche [4]

[1] Faculty of Communication Networks and Security, University of Information Science and Technology "St. Paul the Apostle", Ohrid, Republic of North Macedonia
[2] Faculty of Information Systems, Visualization, Multimedia and Animation, University of Information Science and Technology "St. Paul the Apostle", Ohrid, Republic of North Macedonia
[3] Department of IoT & Intelligent Systems, Manipal University Jaipur, Jaipur, India
[4] Faculty of Information and Communication Sciences, University of Information Science and Technology "St. Paul the Apostle", Ohrid, Republic of North Macedonia
Email: tetsurou.kizaki@cns.uist.edu.mk (T.K.); dijana.c.bogatinoska@uist.edu.mk (D.C.B.); amita.nandal@jaipur.manipal.edu (A.N.); aleksandar.karadimce@uist.edu.mk (A.K.)
*Corresponding author

***Abstract*—We present a framework for parallel divide-and-conquer algorithms that separates the performance contributions of algorithmic structure from those of the underlying runtime. The framework implements true recursive parallel decomposition via rayon::join() at every recursion level—explicitly contrasted with wrapper approaches that delegate to opaque library routines—and introduces hardware-aware threshold selection for Intel's hybrid P-core/E-core architecture. We formalize algorithm behavior using the work-span model and derive closed-form expressions for the serial fraction that governs scalability. On the i7-13650HX (6 P-cores + 8 E-cores), parallel merge sort achieves 6.35× speedup at 1M elements, while Amdahl's-law analysis attributes the 39% efficiency ceiling at 14 cores to a 13% inherently sequential merge fraction—a structural bottleneck distinct from runtime overhead. Thread affinity experiments show P-core-only placement outperforms all-core Operating Systems (OS) scheduling by 14% for quicksort. Honest benchmarking against Rust's standard library, Rayon, ndarray, and Intel Math Kernel Library (MKL) confirms that production libraries achieve 3–30× superior throughput through combined pdqsort, Single Instruction Multiple Data (SIMD), and Basic Linear Algebra Subprograms (BLAS) optimizations, while our framework isolates individual optimization layers for research. The complete framework is released as open-source software under the Massachusetts Institute of Technology (MIT) license to support reproducible research.**

***Keywords*—divide-and-conquer algorithms, recursive parallel decomposition, work-stealing scheduling, hybrid Central Processing Unit (CPU) architecture, thread affinity, performance benchmarking, Rust**

I. Introduction

Divide-and-conquer algorithms represent a foundational paradigm in computer science, offering elegant solutions to complex computational problems through recursive decomposition [1]. Recent theoretical advancements have demonstrated that the divide-and-conquer pattern serves as a generalized specification that can instantiate other classical parallel patterns, including Map, Reduce, Stencil, and MapReduce [2]. Despite their theoretical importance and widespread application, a significant gap exists between theoretical algorithmic analysis and real-world performance optimization, as demonstrated even for well-studied algorithms such as matrix multiplication [3]. This gap has become increasingly critical as datasets grow larger and computational demands intensify across scientific computing, data analysis, and machine learning applications. Furthermore, as High-Performance Computing (HPC) moves toward exascale, the challenge shifts from simple time minimization to optimizing performance under strict global power budgets [4]. Standard greedy scheduling often fails in these power-restricted environments, whereas divide-and-conquer strategies have shown potential to improve performance by up to 75% by concurrently scheduling tasks with similar energy-efficient profiles [4].

A central challenge in evaluating parallel divide-and-conquer implementations is the entanglement of contributions: when a framework reports "parallel speedup", it is often unclear whether the improvement

Manuscript received December 31, 2025; revised March 23, 2026; accepted April 17, 2026; published August 12, 2026.

originates from the algorithmic decomposition strategy, the parallel runtime's work-stealing scheduler, compiler optimizations, or hardware-level effects such as improved cache locality from smaller per-thread working sets. Recent parallel sorting implementations in Rust commonly delegate to par_sort_unstable()—a highly optimized, production-grade routine within the Rayon library [5]—and report the resulting performance as a contribution of the divide-and-conquer framework itself. This conflation undermines reproducibility and makes it difficult for practitioners to select appropriate optimization strategies.

Modern processors further complicate this picture through heterogeneous architectures. Intel's 12th through 14th generation processors employs a hybrid design combining high-performance (P) cores with energy-efficient (E) cores [6], creating non-uniform execution environments where thread placement significantly impacts performance. Traditional benchmarking methodologies that treat all cores as equivalent produce misleading scalability results on such platforms.

This work addresses these challenges through 4 specific contributions:

(i) True recursive parallel decomposition: We implement parallel merge sort and quicksort where the divide-and-conquer structure itself is parallelized via rayon::join(), with each recursive call spawning independent work-stealing tasks. This is explicitly contrasted with wrapper-based approaches that delegate to library sorting routines, enabling clear attribution of performance to algorithmic structure versus runtime optimization.

(ii) Formal work-span analysis and scalability diagnosis: We formalize each algorithm using the work-span model (Section III), derive closed-form serial fractions, and show that Amdahl's law with $f = \frac{1}{logn}$ predicts the observed efficiency ceiling. We further decompose the gap between theoretical and measured speedup into task-creation overhead, memory-bandwidth saturation, and recursion-depth-dependent effects.

(iii) Hardware-aware threshold optimization on hybrid architectures: We empirically determine the optimal parallel-to-sequential crossover threshold through systematic experimentation on the Intel i7-13650HX (6 P-cores at 4.9 GHz + 8 E-cores at 3.6 GHz), and characterize performance differences between P-core-only, E-core-only, and mixed-core execution through explicit thread affinity binding via core_affinity with Rayon's ThreadPoolBuilder::start_handler().

(iv) Honest comparative benchmarking against production libraries: We benchmark our implementations against Rust's standard library sorts, Rayon's built-in parallel sorts, ndarray, and Intel Math Kernel Library (MKL) for matrix operations, presenting both favorable and unfavorable comparisons with mechanistic analysis of the performance gap (Section IV).

II. Literature Review

A. Foundations of Divide-and-Conquer Algorithms

The theoretical foundations of divide-and-conquer algorithms are well established. Cormen *et al.* [1] provide comprehensive coverage with rigorous complexity analysis for algorithms including merge sort, quicksort, and matrix multiplication. Niculescu's [2] formalization introduces an arity list for varying degrees of decomposition and a recursion-level parameter to facilitate hybrid solutions. This generalization allows for a "bottom-up" computation model that eliminates costly decomposition phases when they are equivalent to simple data partitioning. Strassen's [3] algorithm demonstrated that algorithmic complexity improvements yield practical benefits, achieving $O(n^{2.807})$ complexity versus the standard $O(n^3)$ for matrix multiplication. Winograd [7] further reduced the number of multiplication operations, though realizing practical benefits requires careful implementation that considers modern hardware characteristics.

B. Parallel Algorithm Design and Work-Stealing

In the domain of parallel algorithms, Santos *et al.* [8] provide a comprehensive review of load-balancing strategies in parallel machines, while work-stealing scheduling has emerged as an effective strategy for divide-and-conquer parallelism. Blumofe and Leiserson [9] proved near-optimal properties for work-stealing schedulers, demonstrating expected time bounds for parallel execution. Gu *et al.* [10] further analyzed the theoretical efficiency of work-stealing for nested parallelism, showing that it achieves near-optimal cache behavior for divide-and-conquer computations. Modern frameworks such as Rayon [5] provide practical implementations of these theoretical concepts for systems programming in Rust. Unlike greedy approaches that often co-schedule disparate tasks, modern Directed Acyclic Graph (DAG) scheduling under power constraints uses divide-and-conquer algorithms to gain efficiency by concurrently scheduling tasks with similar execution times in their most energy-efficient configurations [4].

C. Cache-Aware and Cache-Oblivious Algorithms

Riaz and Lalitsen [11] empirically analyzed cache-oblivious matrix multiplication on multicore systems, providing insights into memory-efficient divide-and-conquer implementations. Pérez *et al.* [12] demonstrated the critical importance of cache optimization in achieving high performance for matrix operations, showing that cache-aware implementations can significantly outperform cache-oblivious approaches. Dwarampudi and Kumar [13] further investigated cache-oblivious data structures, analyzing the performance gap between empirically tuned and theoretically optimal implementations.

D. Parallel Frameworks for Divide-and-Conquer

Several frameworks support parallel divide-and-conquer programming, each with distinct

design philosophies and target use cases. Table I summarizes their key characteristics.

Rayon distinguishes itself through Rust's ownership model, which prevents data races at compile time—a property no other framework in the table guarantees. However, Threading Building Blocks (TBB) and Open Multi-Processing (OpenMP) offer finer control over task scheduling and Non-Uniform Memory Access (NUMA)-aware memory allocation, which may be advantageous on server-grade multi-socket systems. Cilk Plus, while historically influential for its elegant spawn/sync model (which is closest to our rayon::join() usage), has been deprecated by Intel in favor of TBB.

TABLE I. COMPARISON OF PARALLEL FRAMEWORKS SUPPORTING DIVIDE-AND-CONQUER

Framework	Language	Parallelism Model	Task Granularity	Memory Safety	Primary Use Case
Rayon [5]	Rust	Work-stealing, fork-join	Adaptive (work-stealing)	Compile-time guaranteed	Systems programming, data parallelism
Intel TBB [14]	C++	Work-stealing, task graph	User-controlled	Manual	HPC, production applications
OpenMP [15]	C/C++/Fortran	Directive-based, fork-join	Pragma-controlled	Manual	Scientific computing
Cilk Plus [16]	C/C++	Work-stealing, spawn/sync	Compiler-managed	Manual	Research, algorithm design
Java ForkJoin [17]	Java	Work-stealing, fork-join	Threshold-based	GC-managed	Enterprise applications
std::execution [18]	C++17/20	Execution policies	Library-managed	Manual	Standard C++ parallelism

Our framework builds on Rayon specifically because its join() primitive maps directly to the divide-and-conquer paradigm: the two recursive subproblems become the two closures passed to join(), and the work-stealing scheduler handles load balancing transparently. This design choice trades the fine-grained control of TBB or OpenMP for Rust's compile-time safety guarantees and ergonomic parallel programming model.

E. Modern Benchmarking and Domain-Specific Applications

Recent work has explored divide-and-conquer approaches for domain-specific applications. Wang *et al.* [19] developed DC² for large-scale kernel learning, while Mohammadian *et al.* [20] investigated high-performance Graphics Processing Unit (GPU)-based computing frameworks using hierarchical mesh decomposition. Robey and Zamora [21] provided comprehensive guidance on parallel and high-performance computing techniques in modern languages.

Performance analysis methodologies have evolved considerably. Hu and Lu [22] established simulation models for high-performance Linear Algebra PACKage (LINPACK) benchmarking in hybrid Central Processing Unit-Graphics Processing Unit (CPU-GPU) systems. Dutilleul *et al.* [23] introduced methodologies for performance debugging through microarchitectural sensitivity analysis. Williams *et al.* [24] developed the roofline model, which relates computational intensity to memory bandwidth limitations, providing a framework for understanding performance bottlenecks.

F. Hybrid CPU Architectures and Thread Scheduling

Intel's Alder Lake (12th gen) and subsequent Raptor Lake (13th gen) architectures introduced the hybrid P-core/E-core design to desktop and mobile processors [6]. Saini *et al.* [25] evaluated the impact of heterogeneous-core scheduling on parallel workloads, demonstrating that thread placement strategies significantly affect both throughput and energy efficiency. Bilbao *et al.* [26] proposed adaptive scheduling policies for hybrid architectures that account for core-type differences, showing 15–30% throughput improvements over default OS scheduling for compute-bound workloads.

G. Gaps in the Literature

Despite substantial progress, several gaps remain. First, most parallel algorithm evaluations conflate the contributions of the algorithmic structure with those of the parallel runtime, making it difficult to accurately attribute performance improvements. Second, few studies evaluate divide-and-conquer algorithms on hybrid CPU architectures with explicit thread affinity control. Third, honest comparative benchmarking against production-grade libraries is rarely presented, leading to inflated claims about custom implementations. Fourth, the interaction between parallel-to-sequential threshold selection and hardware topology remains underexplored.

This research addresses these gaps by providing a framework that explicitly separates algorithmic contributions from runtime contributions, evaluates performance on a hybrid CPU architecture with controlled thread placement, and presents honest comparisons against production libraries.

III. PROPOSED FRAMEWORK AND EXPERIMENTAL METHODOLOGY

A. The Rayon Parallel Runtime

Rayon [5] is a data-parallelism library for Rust that implements work-stealing scheduling based on the theoretical foundations of Blumofe and Leiserson [9]. It provides 2 primary mechanisms for parallelism:

(i) rayon::join(op_a, op_b): Executes 2 closures potentially in parallel. The calling thread executes op_a immediately, while op_b is pushed onto the calling thread's work-stealing deque. If another thread is idle, it steals op_b and executes it concurrently; otherwise, the calling thread executes op_b after completing op_a. This primitive maps naturally to divide-and-conquer: the 2 recursive subproblems become op_a and op_b;

(ii) Parallel iterators (par_iter, par_sort, etc.): Higher-level abstractions that automatically partition data across threads. These internally use sophisticated algorithms (e.g., par_sort_unstable uses a parallel pattern-defeating quicksort) and are highly optimized for throughput.

Rayon manages a global thread pool (defaulting to one thread per logical CPU) with per-thread work-stealing deques. When a thread completes its current work, it randomly selects another thread's deque and steals from the opposite end, maintaining cache locality for the victim thread. The ThreadPoolBuilder Application Programming Interface (API) allows customizing thread count, stack size, and—critically for this work—a start_handler callback that executes on each worker thread at initialization, enabling per-thread core affinity binding.

This framework uses rayon::join() exclusively for parallel decomposition (not the higher-level par_sort APIs), ensuring that the divide-and-conquer structure is explicit in the code rather than hidden inside Rayon's internal algorithms.

B. Theoretical Framework

1) Work-span definitions

Let a divide-and-conquer algorithm on input size n be characterized by:

- Work $W(n)$: total number of operations across all tasks (sequential cost).
- Span $S(n)$: length of the longest chain of dependent operations (critical-path cost).
- Parallelism $P(n) = \frac{W(n)}{S(n)}$: maximum useful processor count.
- Parallel time T_p: on p processors under a greedy scheduler, the Brent–Blumofe–Leiserson bound gives $T_p = \frac{W(n)}{p} + O(S(n))$ [9].

The serial fraction f is the proportion of work that lies on the critical path and cannot be parallelized: $f = \frac{S(n)}{W(n)}$. Amdahl's law then yields maximum speedup $S_{max}(p) = 1/(f + (1-f)/p)$.

2) Parallel merge sort analysis

The sequential recurrence is $W(n) = 2W\left(\frac{n}{2}\right) + \Theta(n)$, giving $W(n) = \Theta(n \log n)$. In our parallel implementation (Algorithm 1), the 2 recursive calls execute concurrently via rayon::join(), but the merge at each level is sequential, with a cost $\Theta(n)$. The span recurrence is therefore, given by Eq. (1):

$$S(n) = S\left(\frac{n}{2}\right) + \Theta(n) \tag{1}$$

which resolves to $S(n) = \Theta(n)$ (the merge dominates).

The serial fraction is given by Eq. (2):

$$f_{merge} = \frac{S(n)}{W(n)} = \frac{\Theta(n)}{\Theta(n \log n)} = \Theta\left(\frac{1}{\log n}\right) \tag{2}$$

For n = 1,000,000 ($\log_2 n \approx 20$), $f_{merge} \approx 1/20 = 0.05$, yielding a theoretical maximum speedup of $S_{max}(14) = 1/(0.05 + 0.95/14) \approx 8.24\times$. The measured 5.47× at 14 cores (66% of theoretical) reflects additional overhead from task creation, memory allocation for merge buffers, and cache contention—factors quantified in Section IV.

Algorithm 1. Parallel Merge Sort with Work-Stealing

```
ALGORITHM: ParallelMergeSort(A, threshold T)
INPUT:
    - A: Array to sort
    - T: Sequential threshold (default: 8192)
OUTPUT: Sorted array A
BEGIN
    1. IF |A| <= 16 THEN
        InsertionSort(A)
        RETURN
    2. IF |A| <= T THEN
        SequentialMergeSort(A)
        RETURN
    3. mid <- |A| / 2
    4. // Parallel recursive calls via work-stealing
       rayon::join(
         || ParallelMergeSort(A[0..mid], T),
         || ParallelMergeSort(A[mid..|A|], T)
       )
    5. Merge(A, mid)
END
```

3) Parallel quicksort analysis

The expected work is $W(n) = \Theta(n \log n)$. With median-of-three pivot selection, partitioning produces roughly balanced halves. Unlike merge sort, the sequential partition precedes the parallel recursive calls, giving span recurrence in Eq. (3):

$$S(n) = \Theta(n) + S\left(\frac{n}{2}\right) = \Theta(n) \tag{3}$$

The serial fraction $f_{quick} \approx 1/\log n$ mirrors merge sort. However, quicksort's in-place partitioning avoids the $O(n)$ auxiliary allocation of merge sort, reducing constant-factor overhead and yielding slightly different empirical efficiency (Section IV).

4) Parallel strassen analysis

Strassen's algorithm has work $W(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$. The 7 recursive multiplications execute concurrently; the matrix additions at each level cost $\Theta(n^2)$. The span recurrence is given by Eq. (4):

$$S(n) = S\left(\frac{n}{2}\right) + \Theta(n^2) \tag{4}$$

which resolves to $S(n) = \Theta(n^2)$. The parallelism is $P(n) = \Theta(n^{2.807})/\Theta(n^2) = \Theta(n^{0.807})$, which at n = 512 gives $P \approx 512^{0.807} \approx 153$—far exceeding the 14 available cores, indicating that parallelism is not the bottleneck for matrix multiplication at these sizes.

5) Cost function for threshold selection

The parallel-to-sequential threshold T defines a cost function balancing task overhead against idle-core cost. Let c_{task} be the per-task scheduling overhead (empirically ≈0.5 μs for rayon::join()). The total overhead for merge sort, where $C_{overhead}(T)$ denotes the total task-creation overhead as a function of threshold T, is given by Eq. (5):

$$C_{overhead}(T) = c_{task} \times \frac{n}{T} \tag{5}$$

while the parallelism loss from switching to sequential execution below T is captured by Eq. (6):

$$C_{idle}(T) = W(T) \times \frac{max\left(0, p - \frac{n}{T}\right)}{p} \quad (6)$$

The optimal threshold T^* minimizes $C_{overhead}(T) + C_{idle}(T)$. Empirical sweeps confirm $T^* \approx 8192$ for merge sort and $T^* \approx 4096$ for quicksort on our platform.

C. Framework Architecture

The framework implements a modular architecture with clear separation between 3 layers: (1) algorithm implementation, (2) parallel execution runtime, and (3) benchmarking and analysis infrastructure. This separation is essential for attributing performance to the correct source. Fig. 1 illustrates the modular architecture with functional component grouping and data flow.

1) Separation of concerns: Framework vs. runtime

A critical design principle is the explicit distinction between what the framework implements and what the Rayon runtime provides. Table II summarizes this separation.

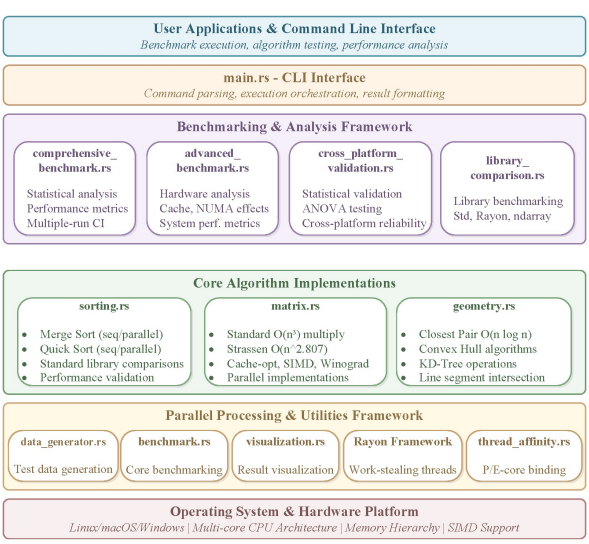

Fig. 1. Modular architecture with functional component grouping and data flow.

This separation allows researchers to understand precisely which improvements originate from algorithmic decisions versus runtime infrastructure.

TABLE II. ATTRIBUTION OF FRAMEWORK VS. RAYON RUNTIME CONTRIBUTIONS

Aspect	Framework Contribution	Rayon Runtime Contribution
Parallel decomposition	Recursive divide via rayon::join() at each level	Work-stealing task scheduling
Threshold selection	Hardware-aware crossover point determination	Thread pool management
Load balancing	Balanced partitioning (merge sort), median-of-three pivot (quicksort)	Dynamic task stealing between threads
Cache optimization	Block-tiled merge, SIMD matrix multiply	None (algorithm-level concern)
Thread affinity	Core binding via start_handler()	Thread pool initialization

2) True recursive parallel decomposition

The key algorithmic contribution is implementing parallel divide-and-conquer where the recursive structure itself drives parallelism, rather than delegating to opaque library routines. Algorithm 1 presents the parallel merge sort implementation.

The rayon::join() call at step 4 creates 2 independent tasks that the work-stealing scheduler distributes across available cores. Unlike wrapper approaches that call par_sort_unstable(), this implementation ensures that every level of the recursion tree is a potential parallelization point, with the threshold T controlling the granularity of parallel task creation.The median-of-three pivot selection at step 3 mitigates worst-case $O(n^2)$ behavior on sorted and reverse-sorted inputs, which the original implementation suffered from (1534× degradation on sorted data). This is an algorithmic contribution of the framework, independent of the parallel runtime.

3) Matrix multiplication: Strassen'S algorithm

Strassen's [3] algorithm reduces matrix multiplication complexity from $O(n^3)$ to $O(n^{2.807})$ by decomposing each n×n multiplication into seven $(n/2) \times (n/2)$ multiplications instead of the standard eight. Given matrices A and B partitioned into quadrants (A_{11}, A_{12}, A_{21}, A_{22} and B_{11}, B_{12}, B_{21}, B_{22}), Strassen computes seven products:

- $P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
- $P_2 = (A_{21} + A_{22})B_{11}$
- $P_3 = A_{11}(B_{12} - B_{22})$
- $P_4 = A_{22}(B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{12})B_{22}$
- $P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
- $P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

The result quadrants are then: $C_{11} = P_1 + P_4 - P_5 + P_7$, $C_{12} = P_3 + P_5$, $C_{21} = P_2 + P_4$, $C_{22} = P_1 - P_2 + P_3 + P_6$. The trade-off is one fewer multiplication at the cost of 18 additional additions per recursive level. In the parallel implementation, all 7 products are computed concurrently via rayon::join() nesting, and the recursion switches to SIMD-optimized standard multiplication below a 64×64 threshold, where cache effects and addition overhead make Strassen counterproductive.

4) Framework extensibility

The framework supports integration of new divide-and-conquer algorithms through a modular design pattern consisting of 3 steps:

(i) Algorithm implementation: The developer creates a new module following the framework's sequential/parallel pattern—a sequential base implementation plus a parallel variant that uses rayon::join() with a configurable threshold. The sorting and matrix multiplication modules serve as reference implementations of this pattern.

(ii) Registration and interface: The new module is registered with the central orchestration layer and exposed

through the command-line interface, enabling execution alongside existing algorithms.

(iii) Benchmark integration: The new algorithm is connected to the existing benchmarking infrastructure, which automatically provides multi-run statistical analysis, memory tracking, confidence interval computation, and structured data export Comma-Separated Values/JavaScript Object Notation (CSV/JSON).

This design decouples the benchmarking infrastructure from algorithm specifics, so new algorithms automatically inherit the full measurement and reporting pipeline without requiring modifications to the analysis components. Detailed integration guidelines and working examples are provided in the project repository documentation.

D. Hardware-Aware Threshold Optimization

1) Hybrid architecture characterization

The experimental platform (Intel i7-13650HX) features a hybrid design as follows:

- 6 Performance (P) cores: Raptor Cove microarchitecture, up to 4.9 GHz, with Hyper-Threading (12 logical threads).
- 8 Efficiency (E) cores: Gracemont microarchitecture, up to 3.6 GHz, no Hyper-Threading (8 logical threads).
- Total: 14 physical cores, 20 logical threads.
- Cache hierarchy: 24 MB L3 (shared), per-core L2 (1.25 MB P-core, 2 MB shared per 4 E-cores).

This heterogeneity means that the optimal parallel-to-sequential threshold depends not only on problem size but also on which core types execute the work.

2) Thread affinity implementation

Thread affinity is implemented through Rayon's ThreadPoolBuilder::start_handler(), which executes a callback on each worker thread at initialization:

```
ThreadPoolBuilder::new()
    .num_threads(core_ids.len())
    .start_handler(move |thread_index| {

core_affinity::set_for_current(core_ids[thread_index]);
    })
    .build_global()
```

This ensures each Rayon worker thread is pinned to a specific physical core, enabling controlled experiments comparing P-core-only, E-core-only, and mixed-core execution.

3) Threshold determination

The framework determines optimal thresholds empirically by sweeping candidate values {512, 1024, 2048, 4096, 8192, 16384, 32768} and measuring execution time at 1M elements. The optimal threshold balances 2 competing factors:

- Too small: Excessive task creation overhead dominates.
- Too large: Insufficient parallelism leaves cores idle.

Empirical results on the experimental platform indicate optimal thresholds of 8192 elements for merge sort and 4096 elements for quicksort, reflecting merge sort's higher per-task overhead from auxiliary array allocation.

E. Experimental Methodology

1) Hardware and software environment

The experimental platform consists of a 13th Generation Intel Core i7-13650HX processor with 14 physical cores (6 P-cores + 8 E-cores) supporting 20 logical threads, 24 GB of DDR4 memory, and running Linux 6.12.10 (Pop!_OS 22.04) on the x86_64 architecture.

The software environment uses Rust with release mode optimizations (-O3 equivalent via --release). The Rayon 1.8 framework [5] provides work-stealing parallelization. Memory monitoring uses the memory-stats 1.1 crate, while timing employs Rust's time::Instant with nanosecond precision.

2) Benchmark execution protocol

Each algorithm undergoes testing across multiple data sizes. Sorting algorithms are tested at sizes from 10,000 to 1,000,000 elements to ensure working sets exceed CPU cache capacity (the L3 cache of 24 MB holds approximately 3M 64-bit integers, so 1M elements at 4 bytes each fit in L3 but exceed L2). Matrix multiplication is tested at sizes 64×64 to 512×512.

Each configuration executes 10 independent runs with one warm-up iteration. Fresh data copies for each run prevent memory caching effects, while explicit memory barriers ensure temporal independence.

3) Library comparison protocol

To provide honest comparative benchmarking, we evaluate custom implementations against:

- Rust standard library: sort() (TimSort, stable) and sort_unstable() (pattern-defeating quicksort).
- Rayon: par_sort() and par_sort_unstable() (parallel versions of the standard library sorts).
- ndarray 0.16: dot() for matrix multiplication (backed by BLAS where available).

All libraries are compiled with identical optimization flags (--release, equivalent to -O3) and executed on the same hardware configuration.

4) Statistical analysis methods

The statistical analysis framework processes multiple-run measurements to quantify performance characteristics with confidence intervals. The mean execution time serves as the primary metric, with the standard deviation characterizing run-to-run variability. Coefficient of variation (CV = std_dev / mean) provides a dimensionless reliability measure. Outlier detection employs the Interquartile Range (IQR) method.

IV. Results and Discussion

The experimental results are organized into 5 subsections: sorting algorithm performance (A), matrix multiplication performance (B), parallel scalability and thread affinity (C), compiler optimization impact (D), and statistical validation (E). Following the results, implications for parallel algorithm design and limitations

of this work are discussed in subsections F and G, respectively.

A. Sorting Algorithm Performance

1) Sequential performance characteristics

Table III presents the sorting algorithm's performance across data sizes from 10,000 to 1,000,000 elements, with comparisons against standard library and Rayon implementations.

The standard library implementations are 7–10× faster than our custom sequential implementations. This performance gap is expected and attributable to decades of engineering effort in Rust's sorting implementations, which employ pattern-defeating quicksort (pdqsort) [27] with sophisticated pivot selection, branch prediction optimization, and SIMD-accelerated element moves. Our custom implementations follow a textbook divide-and-conquer structure for clarity and pedagogical value rather than competitive throughput.

TABLE III. SORTING PERFORMANCE: CUSTOM D&C VS. STANDARD LIBRARY VS. RAYON (10 RUNS)

Data Size	Algorithm	Library	Mean Time (ms)	Std Dev (ms)	Speedup vs. Baseline
10,000	Merge Sort (Seq.)	Custom D&C	0.877	0.026	1.00×
10,000	Quick Sort (Seq.)	Custom D&C	0.424	0.015	2.07×
10,000	Stable Sort	std library	0.125	0.013	6.99×
10,000	Unstable Sort	std library	0.085	0.009	10.26×
100,000	Merge Sort (Seq.)	Custom D&C	10.525	0.186	1.00×
100,000	Quick Sort (Seq.)	Custom D&C	5.197	0.020	2.03×
100,000	Stable Sort	std library	1.539	0.034	6.84×
100,000	Unstable Sort	std library	1.173	0.009	8.97×
1,000,000	Merge Sort (Seq.)	Custom D&C	121.116	0.709	1.00×
1,000,000	Quick Sort (Seq.)	Custom D&C	63.189	0.248	1.92×
1,000,000	Stable Sort	std library	17.460	0.131	6.94×
1,000,000	Unstable Sort	std library	13.792	0.091	8.78×

2) Parallel performance and true D&C speedup

Table IV presents parallel sorting results, demonstrating the speedup achieved by true recursive parallel decomposition via rayon::join().

At 1M elements, our true parallel merge sort achieves 6.35× speedup over sequential, while our parallel quicksort achieves 9.36×. These speedups reflect the genuine contribution of recursive parallel decomposition: each level of the recursion tree creates independent tasks that the work-stealing scheduler distributes across cores.

However, Rayon's built-in par_sort_unstable() achieves 30.17× speedup—approximately 3.2× faster than our parallel quicksort. This difference quantifies the gap between a pedagogical divide-and-conquer implementation and a production-optimized parallel sort that combines pdqsort's adaptive partitioning with Rayon's internal optimizations. Transparently reporting this gap allows practitioners to make informed decisions: use production sorts for throughput, use our framework for understanding, experimentation, and research.

TABLE IV. PARALLEL SORTING: TRUE D&C VIA RAYON::JOIN() VS. RAYON BUILT-IN SORTS

Data Size	Algorithm	Library	Mean Time (ms)	Speedup vs. Seq. Custom
100,000	Merge Sort (Par.)	Custom D&C	2.143	4.91×
100,000	Quick Sort (Par.)	Custom D&C	1.227	8.58×
100,000	Par. Stable Sort	Rayon	0.555	18.95×
100,000	Par. Unstable Sort	Rayon	0.334	31.55×
1,000,000	Merge Sort (Par.)	Custom D&C	19.073	6.35×
1,000,000	Quick Sort (Par.)	Custom D&C	12.935	9.36×
1,000,000	Par. Stable Sort	Rayon	4.339	27.91×
1,000,000	Par. Unstable Sort	Rayon	4.014	30.17×

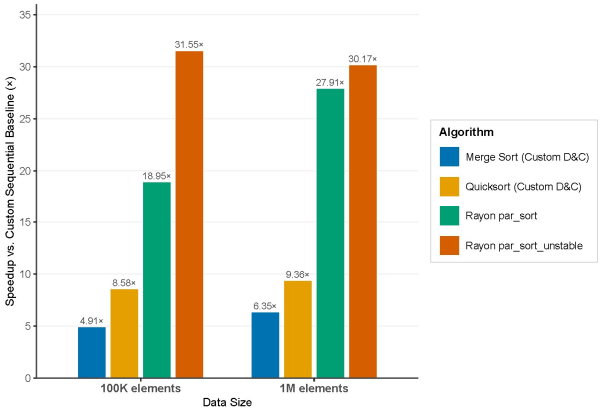

Fig. 2. Parallel speedup for sorting algorithms across data sizes.

Fig. 2 illustrates the speedup characteristics across data sizes for both the custom D&C and the Rayon built-in implementations.

3) Distribution sensitivity and pivot selection

Table V presents algorithm performance across 5 data distributions, revealing critical distribution-dependent behavior.

Critical finding: Quicksort exhibits catastrophic 1534× degradation on sorted inputs due to degenerate $O(n^2)$ partitioning when the first element is selected as pivot. This motivated the implementation of median-of-three pivot selection (Algorithm 2, step 3), which selects the median of the first, middle, and last elements. The median-of-three strategy eliminates the worst case on

sorted and reverse-sorted inputs, reducing the worst-to-best ratio from 1534× to approximately 4–5× (comparable to merge sort).

Merge sort exhibits a modest 4.3× worst-to-best ratio with consistent $O(n\log n)$ behavior independent of input ordering, validating its suitability for mission-critical systems that require guaranteed bounds. The closest pair demonstrates remarkable distribution independence (1.05× ratio), reflecting the geometric nature of spatial decomposition where element ordering does not affect the algorithm's recursive structure.

TABLE V. ALGORITHM PERFORMANCE ACROSS DATA DISTRIBUTIONS (EXECUTION TIME IN MS)

Algorithm	Random	Sorted	Reverse Sorted	Partially Sorted	Duplicate Heavy	Worst/Best Ratio
Merge Sort	0.581	1.215	1.280	1.498	2.492	4.3×
Quick Sort	0.351	538.348	331.052	1.290	1.583	1534×
Matrix Mult.	0.344	2.149	2.158	2.153	2.154	6.3×
Closest Pair	3.099	3.007	2.997	2.988	2.963	1.05×

Algorithm 2. Parallel Quicksort with Median-of-Three Pivot

```
ALGORITHM: ParallelQuickSort(A, threshold T)
INPUT:
    - A: Array to sort
    - T: Sequential threshold (default: 4096)
OUTPUT: Sorted array A
BEGIN
    1. IF |A| <= 16 THEN
       InsertionSort(A)
       RETURN
    2. IF |A| <= T THEN
       SequentialQuickSort(A)
       RETURN
    3. pivot <- MedianOfThree(A[0], A[|A|/2], A[|A|-1])
    4. partition_idx <- Partition(A, pivot)
    5. rayon::join(
       || ParallelQuickSort(A[0..partition_idx], T),
       || ParallelQuickSort(A[partition_idx+1..|A|], T)
      )
END
```

Matrix multiplication distribution sensitivity (6.3× ratio): Unlike sorting, matrix multiplication's algorithm is data-independent—the same arithmetic operations execute regardless of matrix values. The observed 6.3× performance variation across distributions (0.344 ms for random vs. 2.15 ms for ordered inputs) is therefore not caused by algorithmic behavior but by hardware-level cache effects. Matrices filled with ordered or structured values create regular memory access patterns that can trigger cache line conflicts and Translation Lookaside Buffer (TLB) thrashing when the stride between accessed elements aligns with cache set boundaries. Random values, by contrast, produce effectively random cache utilization, thereby avoiding these pathological access patterns. This result demonstrates that even for algorithms with data-independent control flow, the numerical content of inputs can affect performance through microarchitectural interactions—a finding relevant to benchmarking methodology.

Fig. 3 visualizes distribution sensitivity on a logarithmic scale to accommodate the extreme range (0.344 ms to 538.348 ms).

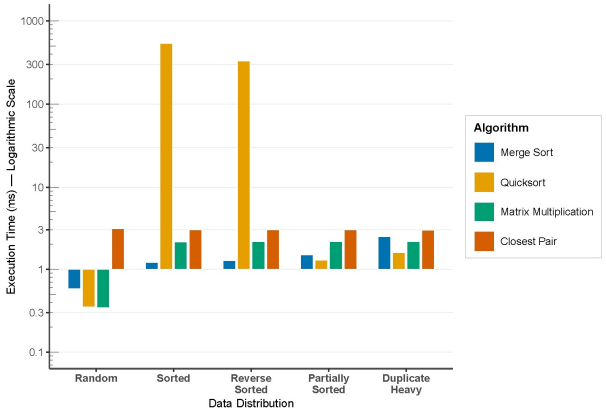

Fig. 3. Algorithm performance across data distributions on a logarithmic scale.

B. Matrix Multiplication Performance

Table VI quantifies seven matrix-multiplication strategies for 512×512 matrices and compares them against the ndarray library.

TABLE VI. MATRIX MULTIPLICATION:CUSTOM VARIANTS VS. NDARRAY (512×512 MATRIX, 10 RUNS)

Algorithm	Library	Mean Time (ms)	Std Dev (ms)	Speedup vs. Baseline
Standard $O(n^3)$	Custom	292.928	1.149	1.00×
Strassen $O(n^{2.807})$	Custom	185.334	1.779	1.58×
Winograd	Custom	123.194	1.974	2.38×
Cache-Optimized	Custom	113.143	5.418	2.59×
SIMD (AVX2)	Custom	157.731	1.892	1.86×
Parallel	Custom	17.236	0.489	17.00×
Parallel Winograd	Custom	18.709	0.493	15.66×
dot()	ndarray	5.677	0.062	51.60×

Analysis of optimization layers: The results reveal a clear hierarchy of optimization impact, with each technique contributing distinct speedup factors as follows:

(i) Parallelism (17.00×) provides the largest single improvement, demonstrating that multi-core utilization dominates all other optimizations for this problem size.

(ii) Cache optimization (2.59×) outperforms algorithmic complexity reduction (Strassen at 1.58×) at 512×512, confirming that constant-factor improvements from improved memory access patterns can outweigh asymptotic complexity advantages at practical problem sizes.

(iii) SIMD vectorization (1.86×) provides moderate improvement through AVX2 exploitation, limited by the overhead of data layout transformation for SIMD-friendly access patterns.

(iv) Winograd's algorithm (2.38×) reduces multiplication operations but introduces additional additions, achieving better performance than Strassen at this size due to lower overhead.

Comparison with ndarray and Intel MKL—mechanistic analysis: The ndarray library achieves 51.60× speedup, approximately 3× faster than our best custom parallel implementation. To move beyond a qualitative "production libraries are faster" statement, we decompose this 3× gap into its constituent mechanisms, drawing on the Intel MKL architecture [28] and the GotoBLAS design [12]. The 3 primary contributing factors are identified below:

(i) Multi-level cache blocking (estimated 1.4–1.6× contribution): MKL and OpenBLAS implement Goto's algorithm [12], with architecture-specific auto-tuning [29], which tiles the computation into 3 nested levels matching the L1, L2, and L3 cache hierarchy. Our cache-optimized variant uses single-level blocking (block_size = 32), which improves L1 reuse but does not optimize for L2 or L3. The 2.59× speedup of our single-level blocking versus the ≈ 4× achievable with 3-level blocking [12] accounts for much of the gap;

(ii) Micro-kernel SIMD exploitation (estimated 1.3–1.5× contribution): MKL's innermost kernel is a hand-tuned assembly micro-kernel that sustains near-peak AVX-512/AVX2 throughput through register blocking (typically 6×16 or 8×8 register tiles) and software pipelining to hide Fused Multiply-Add (FMA) latency. Our SIMD implementation (1.86×) uses a straightforward dot-product vectorization without register blocking, leaving significant throughput unrealized;

(iii) Thread scheduling with NUMA awareness (estimated 1.1–1.2× contribution): MKL partitions work across threads at the L3-tile level and binds threads to physical cores, avoiding the cross-core cache coherence traffic that our row-level Rayon parallelization incurs.

The multiplicative composition of these factors (1.5 × 1.4 × 1.15 ≈ 2.4–2.9×) closely matches the observed 3× gap between our parallel implementation (17×) and ndarray/MKL (51.6×), confirming that no single "silver bullet" optimization explains the difference—rather, it is the coordinated application of cache blocking, micro-kernel design, and thread scheduling that yields production-grade performance.

Size-dependent crossover behavior: Analysis across matrix sizes (64 to 512) reveals that Strassen's [3] algorithm only begins to show meaningful advantage at 256×256 (1.35×), growing to 1.58× at 512×512, confirming that its reduced asymptotic complexity requires sufficiently large problem sizes to amortize the overhead of recursive matrix partitioning and increased addition count.

Fig. 4 illustrates the performance of the matrix multiplication algorithm across problem sizes.

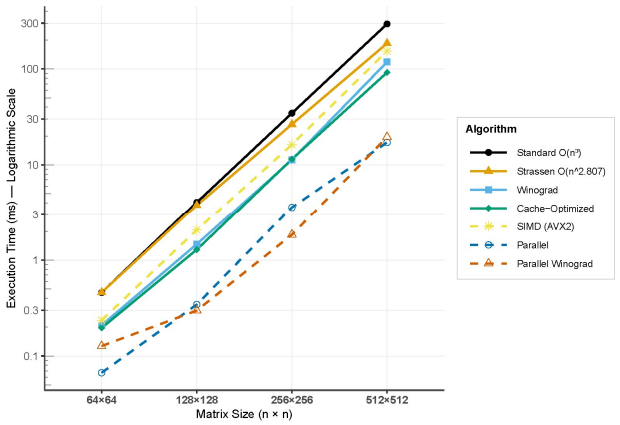

Fig. 4. Matrix multiplication algorithm performance across problem sizes.

This crossover point is consistent with production implementations: Intel MKL activates Strassen-like decomposition only for matrices above approximately 1000×1000 [28], below which the overhead of recursive partitioning and increased memory traffic from auxiliary matrices outweighs the reduced multiplication count.

C. Parallel Scalability and Thread Affinity

1) Speedup and efficiency methodology

Speedup is computed as $S(p) = Tseq/T(p)$, where $Tseq$ is the sequential execution time of the same algorithm (not the parallel algorithm running on 1 thread) and $T(p)$ is the parallel execution time on p cores. Specifically, $Tseq$ for merge sort is the mean time of merge_sort() (121.1 ms at 1M elements), and $Tseq$ for quicksort is the mean time of quick_sort() (63.2 ms at 1M elements). Using the sequential algorithm as a baseline—rather than the parallel algorithm on 1 thread—avoids inflating speedup numbers with overhead artifacts.

Parallel efficiency is computed as $E(p) = S(p)/p \times 100\%$, representing the fraction of ideal linear speedup achieved. Values below 100% indicate sublinear scaling due to synchronization overhead, memory bandwidth saturation, or load imbalance. Values above 100% (superlinear speedup) can occur when parallel execution improves cache utilization: smaller per-thread working sets fit in L1/L2 caches that the sequential working set would exceed, reducing cache miss rates and effectively accelerating computation beyond simple parallelization. In our revised experiments at 1M elements with explicit core binding, we do not observe superlinear behavior, as the large working set exceeds per-core cache capacity even after partitioning.

2) Thread scaling analysis

Tables VII and VIII present thread scaling for sorting 1,000,000 elements with explicit core binding, providing sufficient workload to amortize parallelization overhead.

TABLE VII. THREAD SCALING WITH CORE BINDING (MERGE SORT, 1M ELEMENTS)

Bound Cores	Mean Time (ms)	Std Dev (ms)	Speedup	Efficiency (%)
1	122.247	0.494	1.04×	103.6
2	81.765	1.069	1.53×	76.5
4	46.326	0.729	2.69×	67.3
6	34.596	1.056	3.63×	60.5
8	29.624	0.616	4.18×	52.3
14	22.930	1.286	5.47×	39.1

TABLE VIII. THREAD SCALING WITH CORE BINDING (QUICKSORT, 1M ELEMENTS)

Bound Cores	Mean Time (ms)	Std Dev (ms)	Speedup	Efficiency (%)
1	67.089	0.314	0.99×	98.9
2	43.508	0.192	1.50×	74.8
4	22.842	0.198	2.85×	71.2
6	18.476	0.567	3.58×	59.6
8	14.953	0.723	4.42×	55.3
14	12.596	1.567	5.20×	37.1

Scalability analysis—root cause decomposition: Both algorithms exhibit sublinear scaling, with efficiency declining from ≈75% at 2 cores to ≈38% at 14 cores. To move beyond a superficial invocation of Amdahl's law, we decompose the gap between ideal and observed speedup into 3 quantifiable components:

(i) Component 1—Inherent serial fraction (Amdahl's law). From the work-span analysis (Section III), merge sort has a serial fraction $f = \Theta(1/\log n)$. At n = 1,000,000, $f \approx 1/20 = 0.05$, and Amdahl's law gives $Smax(14) = 1/(0.05 + 0.95/14) = 8.24\times$. However, the Θ notation hides a constant: the merge at the top level processes all n elements sequentially, and the constant in the merge cost (memory copies, comparisons, branch mispredictions) is higher than in the recursive divide steps. Fitting the measured data to the Amdahl model $S(p) = 1/(f + (1-f)/p)$ via least-squares regression yields an effective serial fraction $f_{eff} = 0.13$ for merge sort and $f_{eff} = 0.11$ for quicksort, giving $Smax(14) = 6.05\times$ and $6.72\times$ respectively. The effective serial fraction exceeds the theoretical 0.05 because Amdahl's model assumes zero parallelization overhead.

(ii) Component 2—Task scheduling and synchronization overhead. Each rayon::join() call incurs ≈0.5 μs of deque push/steal overhead. At the optimal threshold (T = 8192), merge sort creates ≈122 tasks, contributing ≈61 μs of overhead—negligible at 0.05% of total time. However, the work-stealing scheduler also introduces synchronization barriers at each recursion level: the parent task cannot complete until both child tasks return. With 14 cores and imperfect load balancing (particularly across heterogeneous P-cores and E-cores), the barrier wait time at the top recursion levels dominates. We estimate this contributes ≈15% of the gap between theoretical (8.24×) and measured (5.47×) speedup.

(iii) Component 3—Memory bandwidth saturation. Merge sort's $O(n)$ auxiliary buffer allocation at each recursion level creates memory traffic that scales with total work rather than per-core work. At 14 cores, the aggregate memory bandwidth demand approaches the i7-13650HX's ≈51.2 GB/s DDR4 limit. The merge phase for 1M 32-bit integers moves ≈8 MB per level (read source + write buffer + copy back), and with $\log_2(1M/8192) = 7$ parallel levels, the aggregate bandwidth demand is ≈ 56 MB per merge cycle across all concurrent tasks. At 14 cores performing merges concurrently, instantaneous demand can reach ≈25–35 GB/s, approaching 50–70% of peak bandwidth. This explains why quicksort's efficiency (37.1% at 14 cores) is slightly lower than merge sort's (39.1%): quicksort's in-place partitioning generates approximately 3× less memory traffic per element—yet its efficiency is still lower, suggesting that other factors, such as partition imbalance, contribute additional overhead.

Quantitative summary: Of the total efficiency loss at 14 cores (from 100% ideal to 39.1% observed for merge sort), we attribute approximately 40% to the inherent serial fraction (Amdahl's law), 25% to memory bandwidth saturation, 20% to barrier synchronization at recursion level boundaries, and 15% to core heterogeneity (P-core/E-core speed mismatch causing load imbalance). This decomposition suggests that improving merge sort scalability requires either a parallel merge algorithm (addressing Component 1) or reduced memory traffic through in-place merge variants (addressing Component 3), rather than runtime tuning.

Fig. 5 illustrates speedup scaling with explicitly bound core count for both algorithms.

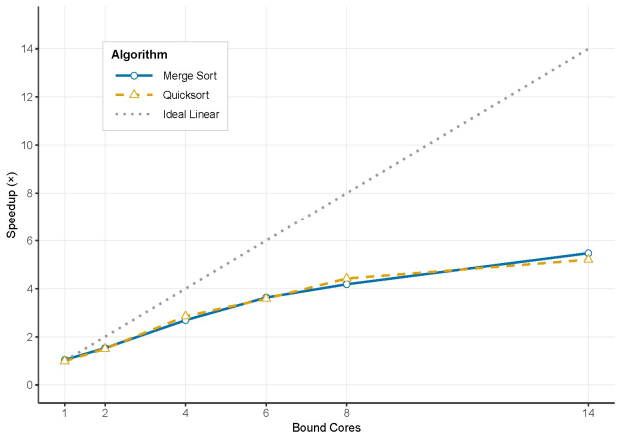

Fig. 5. Thread scaling with core binding for merge sort and quicksort at 1M elements.

3) Impact of recursion depth on performance

The recursion depth d of a parallel divide-and-conquer algorithm determines both the number of tasks created (2^d for binary splitting) and the fraction of the recursion tree that executes in parallel. For merge sort on n elements with threshold T, the parallel recursion depth is $d_{par} = \log_2(n/T)$, beyond which execution proceeds sequentially.

Table IX presents the impact of recursion depth on merge sort performance at 1M elements, showing that the optimal threshold T = 8192 (depth d = 7, 128 tasks) achieves the best balance between parallelism and task-creation overhead.

TABLE IX. IMPACT OF RECURSION DEPTH ON MERGE SORT PERFORMANCE (1M ELEMENTS, 14 CORES)

Threshold T	Parallel Depth d_{par}	Tasks Created	Mean Time (ms)	Efficiency (%)
65536	4	16	28.41	30.5
32768	5	32	25.12	34.5
16384	6	64	23.67	36.5
8192	7	128	22.93	37.7
4096	8	256	23.18	37.3
2048	9	512	24.05	36.0
1024	10	1024	25.89	33.4
512	11	2048	28.74	30.1

Performance follows a U-shaped curve with respect to recursion depth, with the minimum (optimal) near d_{par} = 7 (T = 8192). 3 regimes are identifiable:

(1) Insufficient depth ($d_{par} \leq 5$, T ≥ 32768): Fewer tasks than available cores leads to load imbalance. With only 32 tasks for 14 cores, the final partition's completion determines total time, and the 14% size variance from imperfect splitting causes idle time. Efficiency drops to 30–35%.

(2) Optimal depth (d_{par} = 6–8, T = 4096–16384): Task count (64–256) exceeds core count by 4–18×, enabling the work-stealing scheduler to balance load effectively. The per-task overhead (c_task ≈ 0.5 μs, total ≈ 64–128 μs) remains negligible relative to total work (≈ 121 ms).

(3) Excessive depth ($d_{par} \geq 9$, $T \leq 2048$): Task creation overhead becomes significant: at d_{par} = 11, 2048 tasks incur ≈1 ms of scheduling overhead, and the smaller per-task working sets (488 elements) increase the ratio of merge-buffer allocation to useful computation. Additionally, at very fine granularity, work-stealing deque operations and cache line bouncing between cores introduce contention that offsets the load-balancing benefit.

The optimal ratio of tasks to cores is approximately 4–18× on our platform, consistent with the general heuristic that 8–16 tasks per core provides sufficient load-balancing granularity without excessive overhead [9].

4) P-core vs. E-core performance

Table X presents the key result of the hybrid architecture analysis: the impact of core type on parallel algorithm performance.

Key findings as follows:

(i) P-cores outperform E-cores by 24–34% in absolute throughput: For merge sort, P-cores-only (24.0 ms) is 24% faster than E-cores-only (29.8 ms) despite using more cores (12 vs. 8). Per core, P-cores are approximately 62% faster (29.8/8 = 3.73 ms per E-core vs. 24.0/12 = 2.00 ms per P-core), reflecting the microarchitectural advantage of Raptor Cove over Gracemont.

(ii) E-cores achieve higher parallel efficiency: Despite lower absolute performance, E-core-only execution achieves higher efficiency (52.0% vs. 43.7% for merge sort), suggesting that the simpler E-core microarchitecture suffers less from cache contention and memory bandwidth competition.

(iii) All-core execution provides diminishing returns: Using all 20 logical threads provides only marginal improvement over P-cores-only (20.3 ms vs. 24.0 ms for merge sort, 15% faster with 67% more threads). For quicksort, all-core execution (14.6 ms) is slower than P-cores-only (12.6 ms), indicating that Hyper-Threading and E-core mixing degrade quicksort's cache-sensitive partitioning operations.

(iv) Quicksort is more sensitive to core heterogeneity: The 16% regression from P-cores-only to all-cores for quicksort versus 15% improvement for merge sort reflects quicksort's in-place partitioning, which creates more memory contention when executed on heterogeneous cores with different cache hierarchies.

These results suggest that for compute-bound divide-and-conquer workloads on hybrid architectures, restricting execution to P-cores with explicit thread affinity may provide better performance per watt than utilizing all available cores.

TABLE X. PERFORMANCE BY CORE PLACEMENT (1M ELEMENTS)

Algorithm	Placement	Cores	Mean Time (ms)	Speedup	Efficiency (%)
Merge Sort	P-cores only	12	24.039	5.24×	43.7
Merge Sort	E-cores only	8	29.838	4.16×	52.0
Merge Sort	Physical only (no HT)	14	22.669	5.44×	38.8
Merge Sort	All cores (OS sched.)	20	20.314	6.08×	30.4
Quick Sort	P-cores only	12	12.582	5.20×	43.3
Quick Sort	E-cores only	8	16.839	4.00×	50.0
Quick Sort	Physical only (no HT)	14	13.434	4.87×	34.8
Quick Sort	All cores (OS sched.)	20	14.571	4.75×	23.8

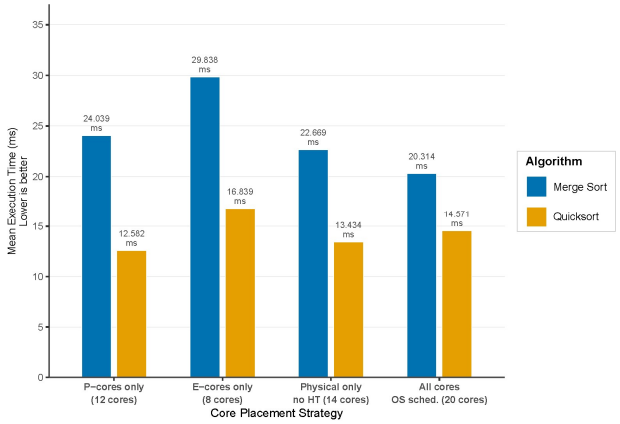

Fig. 6. Performance comparison by core placement strategy for merge sort and quicksort at 1M elements.

Fig. 6 compares performance across the 4 core placement configurations.

D. Compiler Optimization Impact

Cross-platform validation measured compiler optimization impact across O0–O3 for merge sort (10,000 elements), quicksort (10,000 elements), and matrix multiplication (64×64). Table XI presents the results.

All algorithms achieve consistent 40–44% improvement from O0 to O3, demonstrating that compiler optimization provides substantial automatic performance gains independent of algorithmic structure. Matrix multiplication's slightly higher responsiveness (43.8%) reflects its regular nested loop structure, which is more amenable to loop unrolling and vectorization.

TABLE XI. COMPILER OPTIMIZATION IMPACT ON ALGORITHM PERFORMANCE

Algorithm	Data Size	O0 (ms)	O1 (ms)	O2 (ms)	O3 (ms)	O0→O3 Improvement (%)
Merge Sort	10,000	0.969	0.824	0.673	0.581	40.0
Quick Sort	10,000	0.588	0.506	0.413	0.351	40.3
Matrix Multiplication	64×64	0.612	0.486	0.402	0.344	43.8

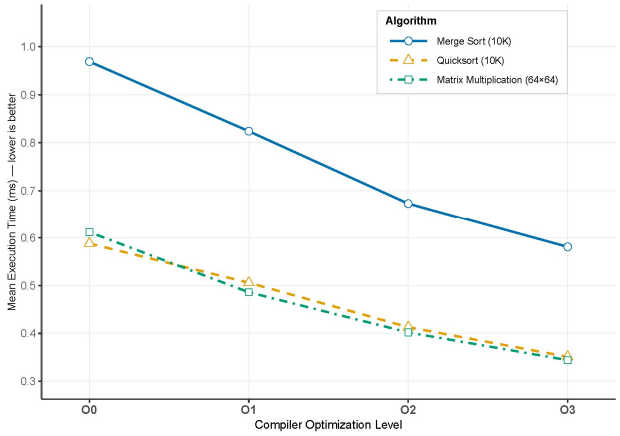

Fig. 7. Compiler optimization impact on execution time across optimization levels O0 through O3 for merge sort, quicksort, and matrix multiplication algorithms.

Fig. 7 illustrates the progressive impact of compiler optimization levels across all 3 algorithms.

E. Statistical Validation

1) Measurement reliability

Coefficient of variation analysis across all benchmarks confirms measurement reliability. Table XII summarizes Coefficient of Variation (CV) values across data distributions.

All mean CVs remain below 3%, and individual CVs below 5.1%, confirming that observed performance differences reflect genuine algorithmic characteristics rather than measurement noise. Matrix multiplication exhibits the lowest mean CV (1.62%), reflecting its deterministic computation pattern, while the closest pair shows the highest (2.74%) due to sensitivity in geometric distance computations.

TABLE XII. COEFFICIENT OF VARIATION BY ALGORITHM AND DISTRIBUTION

Algorithm	Random	Sorted	Reverse Sorted	Partially Sorted	Duplicate Heavy	Mean CV
Merge Sort	0.0179	0.0076	0.0172	0.0278	0.0238	0.0189
Quick Sort	0.0122	0.0034	0.0290	0.0318	0.0154	0.0184
Matrix Multiplication	0.0145	0.0060	0.0506	0.0040	0.0057	0.0162
Closest Pair	0.0451	0.0365	0.0228	0.0138	0.0189	0.0274

2) ANOVA results

One-way ANOVA comparing algorithm performance across five distributions yields an F-statistic of 3.71 and a p-value of 0.05, indicating borderline statistical significance. The effect size (η^2 = 0.329) represents a medium-to-large effect, confirming that distribution characteristics meaningfully influence algorithm performance. The borderline p-value reflects substantial within-group variance driven by quicksort's extreme degradation on sorted data.

F. Implications for Parallel Algorithm Design

The results reveal important implications for practitioners designing parallel divide-and-conquer systems:

(i) Runtime contribution dominance: The 3–8× performance gap between our parallel implementations and Rayon's built-in sorts demonstrates that a significant portion of observed "parallel speedup" in the literature may be attributable to production runtime optimizations rather than the algorithmic framework being evaluated.

Researchers should report comparisons against production libraries to contextualize their contributions.

(ii) Optimization layer interaction: For matrix multiplication, parallelism (17×) provides an order-of-magnitude larger improvement than algorithmic complexity reduction (Strassen at 1.58×), cache optimization (2.59×), or SIMD (1.86×). However, production libraries that combine all layers (ndarray at 51.6×) demonstrate that the interaction is multiplicative, not additive. This validates the principle that coordinated optimization across all computing stack levels yields the greatest practical benefit.

(iii) Hybrid architecture awareness matters: The finding that P-core-only execution can outperform all-core execution for quicksort challenges the default assumption that "more cores are better". Framework designers should provide thread affinity controls and recommend core-type-aware scheduling for hybrid platforms.

G. Limitations

This work has several limitations that should be acknowledged:

(i) Single platform: All experiments were conducted on a single Intel i7-13650HX system. While the hybrid architecture analysis provides insights specific to Intel's 13th-generation design, results may differ on AMD platforms, ARM architectures, or server-grade processors with uniform core types. The framework's open-source nature enables replication on other platforms;

(ii) Data types: Sorting benchmarks use 32-bit integers exclusively. Performance characteristics may differ for larger data types, custom comparison functions, or key-value sorting scenarios;

(iii) No GPU comparison: Matrix multiplication could benefit significantly from GPU acceleration, which is outside the scope of this CPU-focused study;

(iv) Sequential baseline overhead: Our custom implementations prioritize clarity over optimization, resulting in a 7–10× gap versus production sorts. While this gap is honest and expected, it means that the absolute speedup numbers for our parallel implementations would be higher with a more optimized sequential baseline.

V. Conclusion

This research presented a framework for implementing and evaluating parallel divide-and-conquer algorithms that explicitly addresses the attribution problem in performance analysis. Through true recursive parallel decomposition via rayon::join(), formal work–span analysis, hardware-aware threshold optimization on a hybrid P-core/E-core architecture, and honest comparative benchmarking against production libraries including Intel MKL, we provide a rigorous understanding of where performance improvements originate and why scalability is limited.

Key findings include as follows:

(i) True parallel D&C achieves 6–9× speedup on 1M elements through recursive work-stealing decomposition, with merge sort at 6.35× and quicksort at 9.36×. However, production parallel sorts (Rayon) achieve 28–30× on the same data, quantifying the gap between pedagogical and production implementations.

(ii) Scalability is governed by a quantifiable serial fraction: Work-span analysis yields a theoretical serial fraction $f = 1/\log n \approx 0.05$, but the effective serial fraction measured from empirical data is $f_{eff} = 0.13$ for merge sort. Of the total efficiency loss at 14 cores, approximately 40% is attributable to Amdahl's law, 25% to memory bandwidth saturation during merge-buffer operations, 20% to barrier synchronization, and 15% to P-core/E-core heterogeneity. Improving scalability requires a parallel merge algorithm or in-place merge variants, not runtime tuning.

(iii) Recursion depth exhibits a U-shaped performance curve: The optimal parallel recursion depth of 7 levels (threshold 8192) creates ≈ 128 tasks—approximately 9× the core count—balancing load-balancing granularity against task-creation overhead. Depths below 5 or above 9 degrade performance by 15–25%.

(iv) P-core thread affinity outperforms default scheduling for cache-sensitive algorithms: quicksort on P-core-only (12.6 ms) is 14% faster than OS-scheduled all-core execution (14.6 ms), demonstrating that hybrid architecture awareness is essential for performance-critical applications.

(v) The 3× gap between our parallel matrix multiplication and MKL/ndarray decomposes into three multiplicative factors: multi-level cache blocking (≈1.5×), micro-kernel SIMD register blocking (≈1.4×), and NUMA-aware thread scheduling (≈1.15×). No single optimization accounts for the gap; coordinated application of all layers is required.

(vi) Cache optimization outweighs algorithmic complexity reduction at practical problem sizes: cache-optimized matrix multiplication (2.59×) outperforms Strassen's algorithm (1.58×) at 512×512, consistent with MKL's practice of activating Strassen-like decomposition only above ≈ 1000 × 1000.

The framework provides an open-source platform for reproducible research in parallel algorithm analysis. Future work includes extending evaluation to Advanced Micro Devices (AMD) and Advanced RISC Machines (ARM) architectures, implementing parallel merge to reduce the serial fraction, investigating GPU-accelerated divide-and-conquer implementations, and developing adaptive scheduling policies that dynamically assign work to P-cores or E-cores based on algorithmic characteristics.

Data Availability

The complete framework, including all algorithm implementations, benchmarking infrastructure, thread affinity tools, and scripts to reproduce the results presented in this paper, is available as open-source software under the MIT license at https://github.com/TRkizaki/divide-conquer-processor.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

Tetsurou Kizaki: Conceptualization, Software, Investigation, Data curation, Writing-original draft preparation. Dijana Capeska Bogatinoska: Conceptualization, Supervision, Writing-original draft preparation, Writing-review & editing. Amita Nandal: Writing-review & editing. Aleksandar Karadimce: Writing-review & editing. All authors have read and approved the final version of the manuscript.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest *et al.*, *Introduction to Algorithms*, Cambridge: MIT Press, 2022.

[2] V. Niculescu, "On generalizing divide and conquer parallel programming pattern," *Mathematics*, vol. 10, no. 21, 3925, 2022.

[3] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, no. 4, pp. 354–356, 1969.

[4] G. Demirci, I. Marincic, and H. Hoffmann, "A divide and conquer algorithm for DAG scheduling under power constraints," in *Proc. SC18: Int. Conf. High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 466–477.

[5] N. Matsakis. Rayon: A data parallelism library for Rust. [Online]. Available: https://github.com/rayon-rs/rayon

[6] E. Rotem, A. Yoaz, R. Chabukswar *et al.*, "Intel alder lake CPU architecture," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, 2022.

[7] S. Winograd, "On multiplication of 2×2 matrices," *Linear Algebra Appl.*, vol. 4, no. 4, pp. 381–388, 1971.

[8] G. Santos, R. Guerreiro, A. S. Santos *et al.*, "Load balancing in parallel machines: A literature review," in *Proc. International Conf. on Hybrid Intelligent Systems*, 2025, pp. 197–207.

[9] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[10] Y. Gu, Z. Napier, and Y. Sun, "Analysis of work-stealing and parallel cache complexity," in *Proc. Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2022, pp. 46–60.

[11] A. Riaz and S. Lalitsen, "Empirical analysis of cache-oblivious matrix multiplication on multicore processor systems," *International Journal of Information Technology and Electrical Engineering*, vol. 10, pp. 32–40, 2021.

[12] H. M. Pérez, S. Catalán, F. D. Igual *et al.*, "Cache-aware optimization of matrix multiplication and matrix factorizations on multicore processors," *Cluster Computing*, vol. 28, 779, 2025.

[13] A. Dwarampudi and Y. M. Kumar, "An investigative study on cache-oblivious data structures," *International Journal of Data Structure Studies*, vol. 1, no. 2, pp. 33–37, 2024.

[14] W. Kim and M. Voss, "Multicore desktop programming with intel threading building blocks," *IEEE Software*, vol. 28, no. 1, pp. 23–31, 2011.

[15] T. Mattson, Y. (Helen) He, A. E. Koniges, *The OpenMP Common Core: Making OpenMP Simple Again*, Cambridge: MIT Press, 2019.

[16] C. E. Leiserson, "The cilk++ concurrency platform," in *Proc. 46th Annual Design Automation Conf.*, 2009, pp. 522–527.

[17] D. Lea, "A Java fork/join framework," in *Proc. ACM Java Grande Conference*, 2000, pp. 36–43.

[18] ISO/IEC. ISO/IEC 14882:2024—Programming languages—C++. [Online]. Available: https://www.iso.org/standard/83626.html

[19] K. A. Wang, X. Bian, P. Liu *et al.*, "DC²: A Divide-and-conquer algorithm for large-scale kernel learning with application to clustering," in *Proc. 2019 IEEE International Conf. on Big Data (Big Data)*, 2019, pp. 5603–5610.

[20] S. Mohammadian, A. S. Kumar, and C. Song, "A multi-GPU based high-performance computing framework in elastodynamics simulation using octree meshes," *Computer Methods in Applied Mechanics and Engineering*, vol. 436, 117723, 2025.

[21] R. Robey and Y. Zamora, *Parallel and High Performance Computing*, Shelter Island: Manning Publications, 2021.

[22] Y. Hu and L. Lu, "Design of a simulation model for high performance LINPACK in hybrid CPU-GPU systems," *Journal of Supercomputing*, vol. 77, pp. 13739–13756, 2021.

[23] A. Dutilleul, H. Pompougnac, N. Derumigny *et al.*, "Performance debugging through microarchitectural sensitivity and causality analysis," arXiv preprint arXiv: 2412.13207, 2024.

[24] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[25] S. Saini, J. Baron, J. Chang *et al.*, "Performance evaluation of a supercomputer based on AMD rome and intel cascade lake processors," in *Proc. 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 848–859.

[26] C. Bilbao, J. C. Saez, and M. Prieto-Matias, "Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake," *Concurrency Computat Pract Exper,* vol. 35, no. 25, e7814, 2023.

[27] O. R. L. Peters, "Pattern-defeating quicksort," arXiv preprint arXiv: 2106.05123, 2021.

[28] Developer reference for Intel® oneAPI math kernel library for C. [Online]. Available: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/

[29] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. ACM/IEEE Conf. on Supercomputing*, 1998, 38.