PSRS Implementation

Parallel Sorting by Regular Sampling
using POSIX Threads

Tayyib Ul Hassan

Student ID: 1888039

CCID: tayyibul

CMPUT 481 – Parallel and Distributed Computing

February 4, 2026

1 Introduction

This report documents my experience implementing the Parallel Sorting by Regular Sampling (PSRS) algorithm using POSIX threads. Rather than describing the algorithm itself, which is well-documented in the original 1993 paper, I focus on the practical challenges encountered during implementation and the insights gained from experimentation. The implementation was developed on macOS, which introduced several platform-specific complications that required creative solutions.

2 Implementation Journey

2.1 The Barrier Problem

My first major roadblock came when the code refused to compile. After some debugging, I discovered that macOS does not provide a native implementation of `pthread_barrier_t`, which is part of the optional POSIX Threads extensions. This was frustrating because barriers are essential for coordinating the four phases of PSRS—each thread must complete its current phase before any thread can proceed to the next.

I initially considered using multiple mutex locks, but this would have been error-prone and difficult to reason about. Instead, I implemented a custom barrier using a combination of `pthread_mutex_t` for mutual exclusion and `pthread_cond_t` for signaling. The implementation maintains a counter that tracks how many threads have reached the barrier. When the last thread arrives, it broadcasts a signal to wake all waiting threads. A generation counter prevents spurious wakeups from causing threads to proceed prematurely.

One subtle issue arose when I placed the barrier function definitions in a header file that was included by multiple source files. This caused duplicate symbol errors during linking. The solution was to declare the functions as `static inline`, which instructs the compiler to generate separate copies for each translation unit rather than creating globally visible symbols.

2.2 Memory Architecture Decisions

Early versions of my implementation allocated large arrays on the stack, which worked fine for small test cases but caused immediate segmentation faults when scaling to millions of elements. The stack size limit on most systems is only a few megabytes, far too small for the data structures PSRS requires.

Moving to heap allocation with `malloc()` solved the immediate problem but introduced new complexity. PSRS requires several multi-dimensional data structures: a 3D array for partitions (indexed by source thread, destination thread, and element), a 2D array for partition sizes, and arrays for samples and pivots. Managing the allocation and deallocation of these nested structures required careful attention to avoid memory leaks.

I chose to use global variables for shared state rather than passing pointers through thread arguments. While this is not ideal from a software engineering perspective, it simplified the code considerably and made the SPMD structure more apparent. Each thread accesses the same global arrays but operates on different indices based on its thread ID.

2.3 The Merge Phase Mistake

Perhaps my most instructive mistake was in Phase 4. My initial implementation simply concatenated the partitions assigned to each thread without actually merging them. The output appeared

reasonable at first glance—the array was mostly sorted—but careful verification revealed that elements were out of order at partition boundaries.

The fix required implementing a proper k-way merge. Each thread receives p partitions (one from each thread's Phase 3 output), and these partitions are individually sorted but not sorted relative to each other. I used a min-heap to efficiently select the smallest element across all partitions at each step. This maintains the $O(n \log p)$ complexity that makes PSRS efficient.

This bug taught me the importance of rigorous correctness checking. I added verification code that scans the entire output array after sorting, confirming that each element is less than or equal to its successor. This check caught several subtle bugs that would have been difficult to detect through manual inspection.

3 Performance Analysis

3.1 Understanding the Results

The speedup results reveal several interesting patterns. First, the break-even point occurs around $p = 2$, where the benefits of parallelism begin to outweigh the overhead. With fewer threads, PSRS performs similar computational work as quicksort but adds overhead from thread management, barrier synchronization, and the additional merge phase.

As thread count increases, speedup improves but at a diminishing rate. With 8-16 threads, I observed speedups of approximately 4-5x rather than the ideal 8-16x. Several factors contribute to this sublinear scaling. The barrier synchronization points introduce idle time—faster threads must wait for slower ones to catch up. Memory bandwidth becomes a bottleneck as multiple threads compete for access to main memory. Additionally, the sample selection and pivot broadcasting in Phase 2 are inherently sequential operations that cannot be parallelized.

Beyond 16 threads, performance actually degrades. At $p = 64$ and $p = 128$, the overhead of managing many threads and the increased synchronization costs outweigh any benefits from parallelism. This demonstrates that blindly adding more processors does not guarantee better performance.

3.2 The Efficiency Perspective

The efficiency metric provides perhaps the clearest view of scalability limitations. Efficiency, defined as speedup divided by processor count ($E = S/p$), measures how effectively each additional processor contributes to performance. Ideal efficiency would be 1.0, meaning each processor provides its full potential speedup.

In practice, efficiency drops steadily as processors are added. At $p = 4$, efficiency hovers around 0.8, indicating reasonably good utilization. By $p = 16$, efficiency falls to around 0.3, meaning more than two-thirds of the theoretical parallel capacity is lost to overhead. This behavior is consistent with Amdahl's Law, which states that the sequential fraction of an algorithm fundamentally limits its scalability.

Interestingly, larger array sizes show slightly better efficiency at high processor counts. This occurs because the parallel work (Phase 1 local sorting and Phase 4 merging) scales with problem size, while the sequential overhead (barrier synchronization, pivot selection) remains relatively constant.

3.3 Practical Implications

These results suggest practical guidelines for using PSRS. For the array sizes tested (32M–164M elements), using 8–16 threads provides a good balance between speedup and efficiency. Beyond 16 threads, the diminishing returns may not justify the increased resource consumption, especially in shared computing environments.

4 Reflections

Working on this project reinforced several principles that are easy to understand theoretically but harder to appreciate until encountered in practice. Synchronization is expensive—every barrier represents a point where the fastest thread waits for the slowest, wasting cycles. Platform portability requires effort—code that compiles cleanly on one Unix system may fail on another due to differences in optional POSIX extensions. Performance measurement requires discipline—single runs are unreliable, and short runs can be dominated by startup overhead rather than actual algorithm performance.

The SPMD programming model, once understood, provides an elegant way to structure parallel code. Each thread executes the same program but operates on different data based on its ID. The barriers serve as synchronization points that divide the program into phases, ensuring all threads have consistent views of shared data at each transition.

Acknowledgements

During implementation, I consulted ChatGPT for assistance with specific technical challenges, particularly the barrier implementation for macOS and the k-way merge algorithm. I also referenced GeeksforGeeks tutorials for refreshers on C programming concepts and the POSIX Threads documentation for understanding the threading API semantics.

Experimental Results

Table 1: Sorting Times (in seconds)

Size	seq	p2	p4	p8	p12	p16	p32	p64
32M	3.04	1.67	0.93	0.75	0.79	0.73	0.74	0.95
48M	4.63	2.60	1.41	1.15	1.10	1.05	1.11	1.46
64M	6.23	3.45	1.89	1.58	1.49	1.38	1.51	1.82
96M	9.55	5.25	2.91	2.34	2.25	2.07	2.11	3.03
128M	12.83	7.13	3.94	3.17	2.98	2.77	2.91	4.45
164M	16.65	9.22	5.17	4.04	3.84	3.60	3.72	4.78

Table 2: Speedup (relative to sequential qsort)

Size	p2	p4	p8	p12	p16	p32	p64	p128
32M	1.83	3.28	4.04	3.87	4.18	4.09	3.19	1.65
48M	1.78	3.29	4.02	4.21	4.40	4.19	3.18	1.51
64M	1.81	3.30	3.95	4.19	4.53	4.13	3.42	1.55
96M	1.82	3.28	4.08	4.25	4.62	4.52	3.15	1.54
128M	1.80	3.26	4.05	4.30	4.64	4.40	2.88	1.31
164M	1.81	3.22	4.12	4.33	4.63	4.47	3.49	1.24

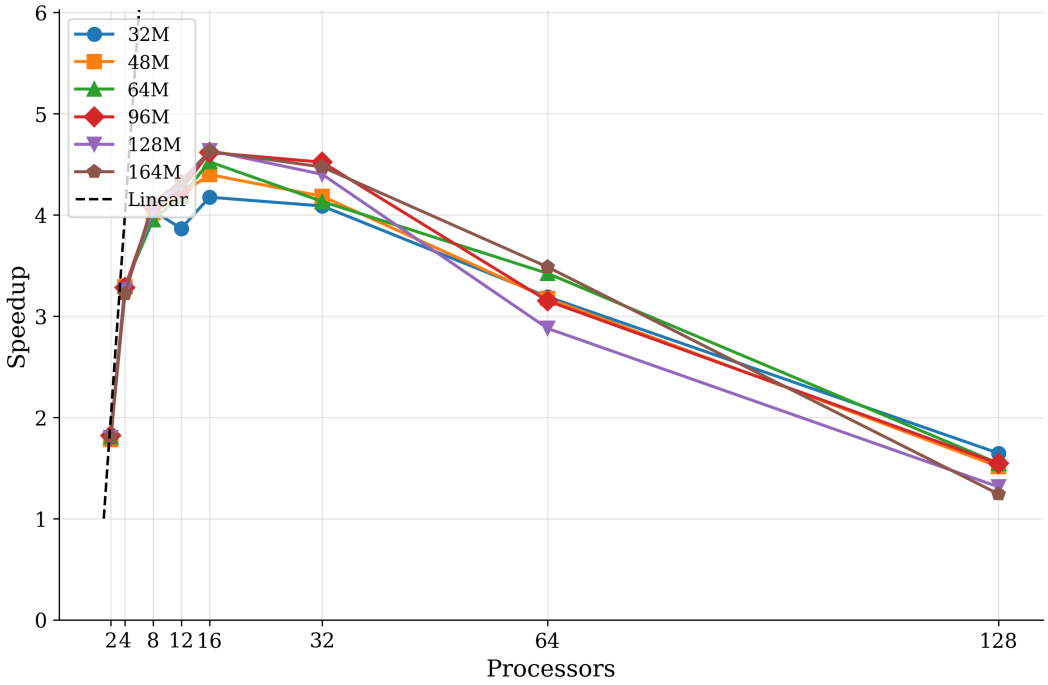

Figure 1: Speedup vs processor count for all tested array sizes.

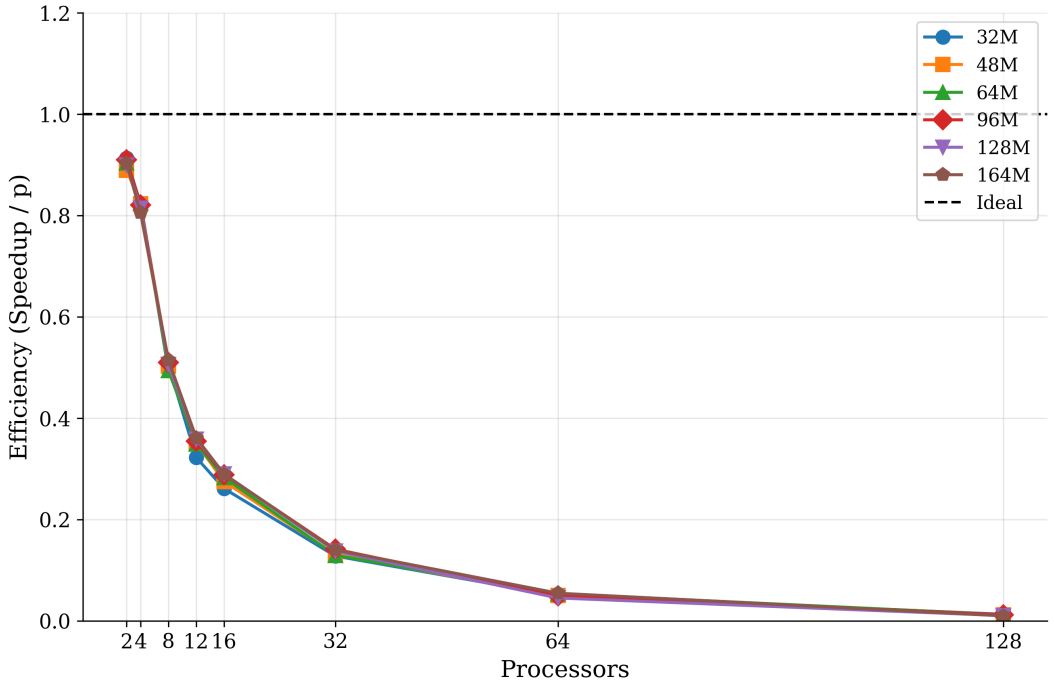

Figure 2: Parallel efficiency (Speedup/p) showing diminishing returns as processors increase.

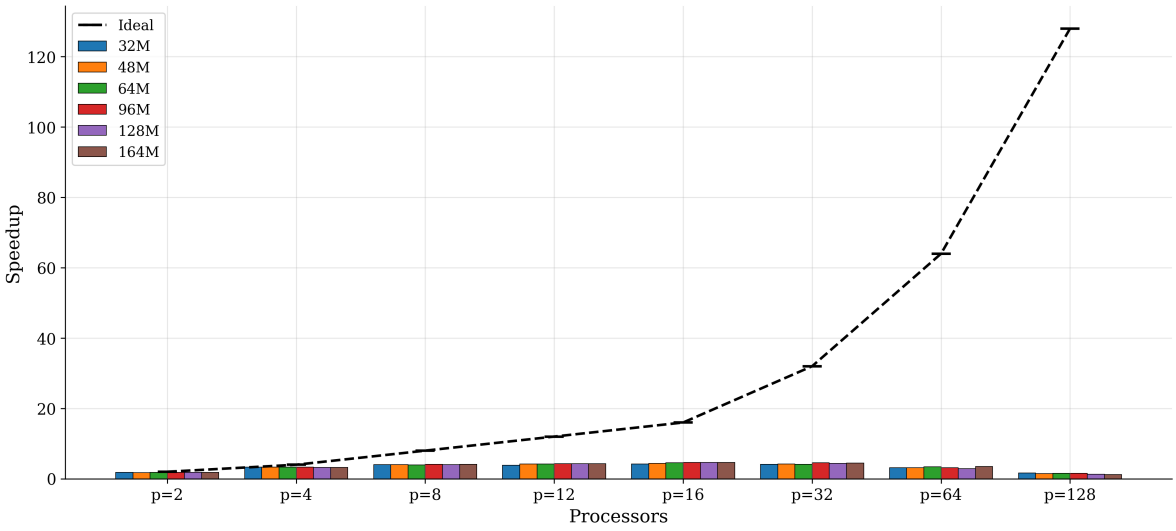

Figure 3: Comparison of achieved speedup vs ideal linear speedup for all array sizes.