PSRS Implementation

Parallel Sorting by Regular Sampling

Tayyib Ul Hassan

Student ID: 1888039

CCID: tayyibul

CMPUT 681: Parallel and Distributed Computing

February 4, 2026

1 Introduction

Thesis: The PSRS algorithm contains an inherent scalability limitation, that is, Phase 4 (merge) exhibits anti-scaling behavior. While Phase 1 (local sort) parallelizes efficiently, Phase 4's cost grows considerably by increasing the number of threads, as each thread must also merge all the p partitions received from all other threads. My experiments confirm this. At p=128, Phase 4 consumes 73% of total execution time (11.07s of 15.03s for 164M elements) compared to 18.24% with 4 threads.

2 Implementation Notes

The implementation uses the SPMD (Single Program Multiple Data) model where the main thread participates as worker thread 0 and spawns p-1 additional threads. All threads execute the same function with different thread IDs. I ensure correctness of the parallel program by using four barrier synchronization points: after local sorting (Phase 1), after sampling (Phase 2), after partitioning (Phase 3), and after merging (Phase 4). Large arrays are heap-allocated to avoid stack overflow. For Phase 1, I used the standard library `qsort()` for local sorting. For Phase 4, I implemented a k-way merge since merging pre-sorted partitions is more efficient. Correctness was validated by verifying the output array is sorted after every run.

3 Experimental Setup

I tested PSRS on a MacBook Air (Apple M2) with array sizes from 32M to 164M integers and thread counts from 2 to 128. Each configuration ran 7 times; I averaged the last 5 runs to reduce startup noise. All timing used `gettimeofday()` with `-O2` optimization.

4 Results

4.1 Speedup Peaks Then Drops

Figure 1 clearly shows that speedup increases from p=2 to around p=8 to 16, then drops aggressively as very high threads are employed in the psrs program. For 64M elements, speedup peaks at 4.25x with 16 threads (Table 2). At p=128, speedup falls to just 1.20x. This pattern of decrease in speedup with increasing the number of threads holds across all array sizes. But why does adding threads hurt performance? The phase timing data answers this question.

4.2 Phase 4 Dominates at High Thread Counts

Tables 3 to 8 show execution time broken down by phase. The data reveals a clear pattern:

- **Phase 1 (Sort)** timing decreases as expected because more threads mean less work per thread. For 164M elements, 8.60s at p=2 drops to 2.39s at p=128.
- **Phase 4 (Merge)** timing increases dramatically. For 164M elements: 0.63s at p=2 grows to 11.07s at p=128, an 18x increase.

At p=128 with 164M elements, Phase 4 accounts for 73% of total execution time (11.07s of 15.03s). The merge phase dominates because each thread must merge p partitions received from all other threads. More threads mean more partitions to merge.

4.3 Efficiency Collapses at Scale

Figure 2 shows efficiency (E = Speedup/p). At p=2, efficiency is approximately 0.86 to 0.92 across all sizes, indicating good utilization. At p=128, efficiency drops to 0.008 to 0.011. This means over 99% of computational capacity is wasted on overhead when using 128 threads.

5 Why This Matters: Lessons Learned

5.1 More Threads Can Hurt Performance

Before this experiment, I assumed more threads would always help (or at worst, plateau). The data proves otherwise. For 96M elements, going from 32 threads (4.17x speedup) to 128 threads (1.02x speedup) makes the program *4x slower*. This was my most surprising finding.

5.2 Barriers Are Not the Bottleneck

I initially suspected synchronization overhead from barrier waits would limit scalability. The phase data

disproves this. Phase 2 (sampling), which involves barrier synchronization and pivot selection, remains under 0.06s even at p=128. The bottleneck is computational, not synchronization.

5.3 The Optimal Thread Count Exists

For these problem sizes on this hardware, p=8 to 16 consistently gives the best speedup. Beyond this, returns are not just diminishing but become negative. This suggests that blindly maximizing thread count is counterproductive and finding the optimal p requires experimentation.

5.4 Debugging Parallel Code Is Hard

A subtle bug in my merge implementation initially produced mostly-sorted output. It was a very long array so I did not bother examining full array for correct sort. But later while observing some other outputs, I saw that bug. After this, I wrote test cases to always validate the overall output. I learned to always verify correctness programmatically, not visually. Especially for parallelized code, where it is hard to manually observe bugs.

5.5 Phase-Level Timing Is Essential

Without phase-level timing, I would have blamed synchronization or load imbalance for poor scaling. The phase breakdown revealed the true culprit was merge cost. This taught me that 1) aggregate timing hides important details and 2) instrumenting individual phases is necessary to diagnose performance problems in distributed program implementations.

6 Conclusion

My experiments demonstrate that Phase 4 (merge) is the fundamental scalability bottleneck in PSRS. Unlike Phase 1 which benefits from parallelism, Phase 4's cost grows with thread count because each thread must merge p partitions. At p=128, Phase 4 consumes 73% of total execution time. This explains why peak speedup (4.0 to 4.3x) occurs at p=8 to 16 and degrades beyond. The key insight is that PSRS's merge phase has inherent anti-scaling behavior that limits the algorithm's parallel efficiency regardless of problem size.

Acknowledgements

I consulted ChatGPT for help with barrier implementation and k-way merge logic. ChatGPT was also used to fix grammatical mistakes in this report. I referenced GeeksforGeeks for C programming concepts and the POSIX threads documentation.

References

1. Albert Armea, "Using pthread_barrier on Mac OS X," `https://blog.albertarmea.com/post/47089939939/using-pthreadbarrier-on-mac-os-x`
2. GeeksforGeeks, "qsort() Function in C," `https://www.geeksforgeeks.org/c/qsort-function-in-c/`
3. GeeksforGeeks, "Merge K Sorted Arrays," `https://www.geeksforgeeks.org/dsa/merge-k-sorted-arrays/`
4. H. Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," Journal of Parallel and Distributed Computing, vol. 14, no. 4, pp. 361-372, 1992.
5. X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi, "On the Versatility of Parallel Sorting by Regular Sampling," Parallel Computing, vol. 19, no. 10, pp. 1079-1103, 1993.

Appendix

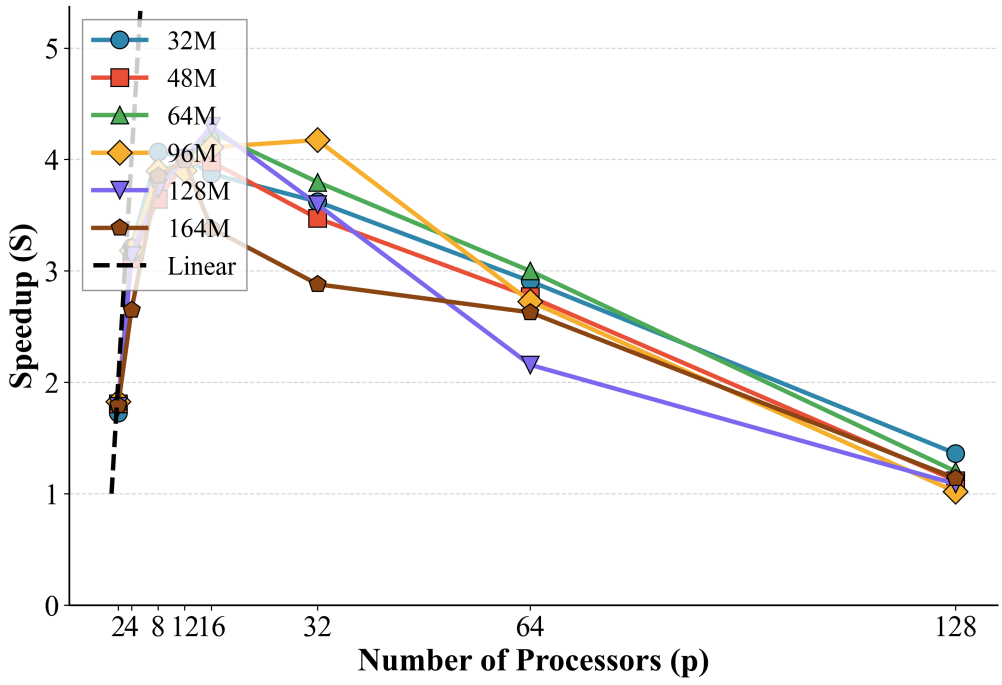

Figure 1: Speedup vs processor count. Peak speedup occurs around p=12-16.

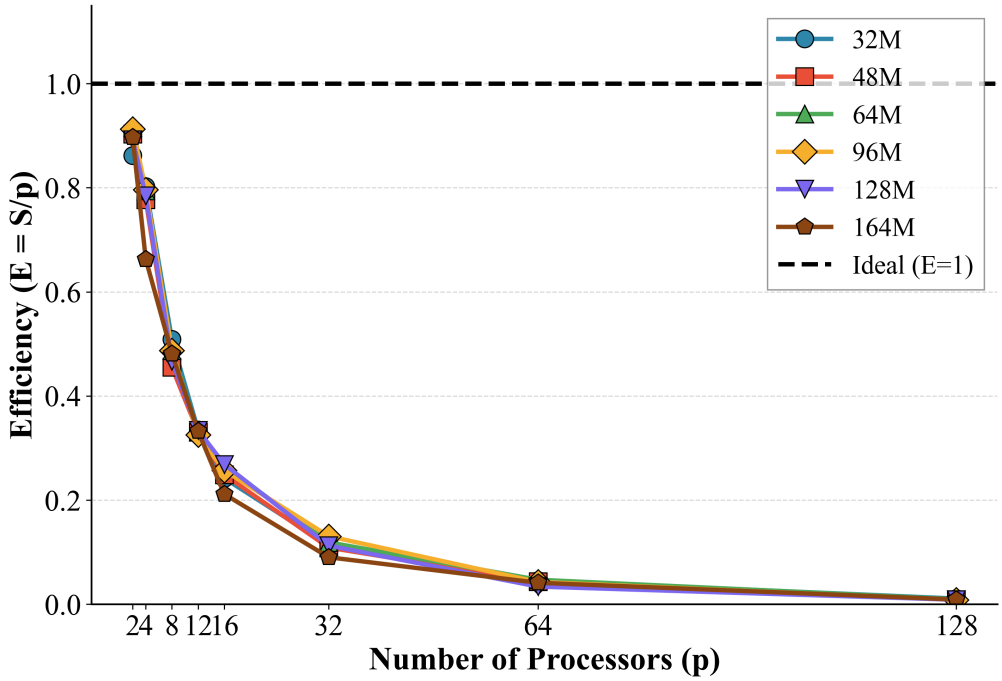

Figure 2: Efficiency (E=S/p) drops rapidly as processors increase.

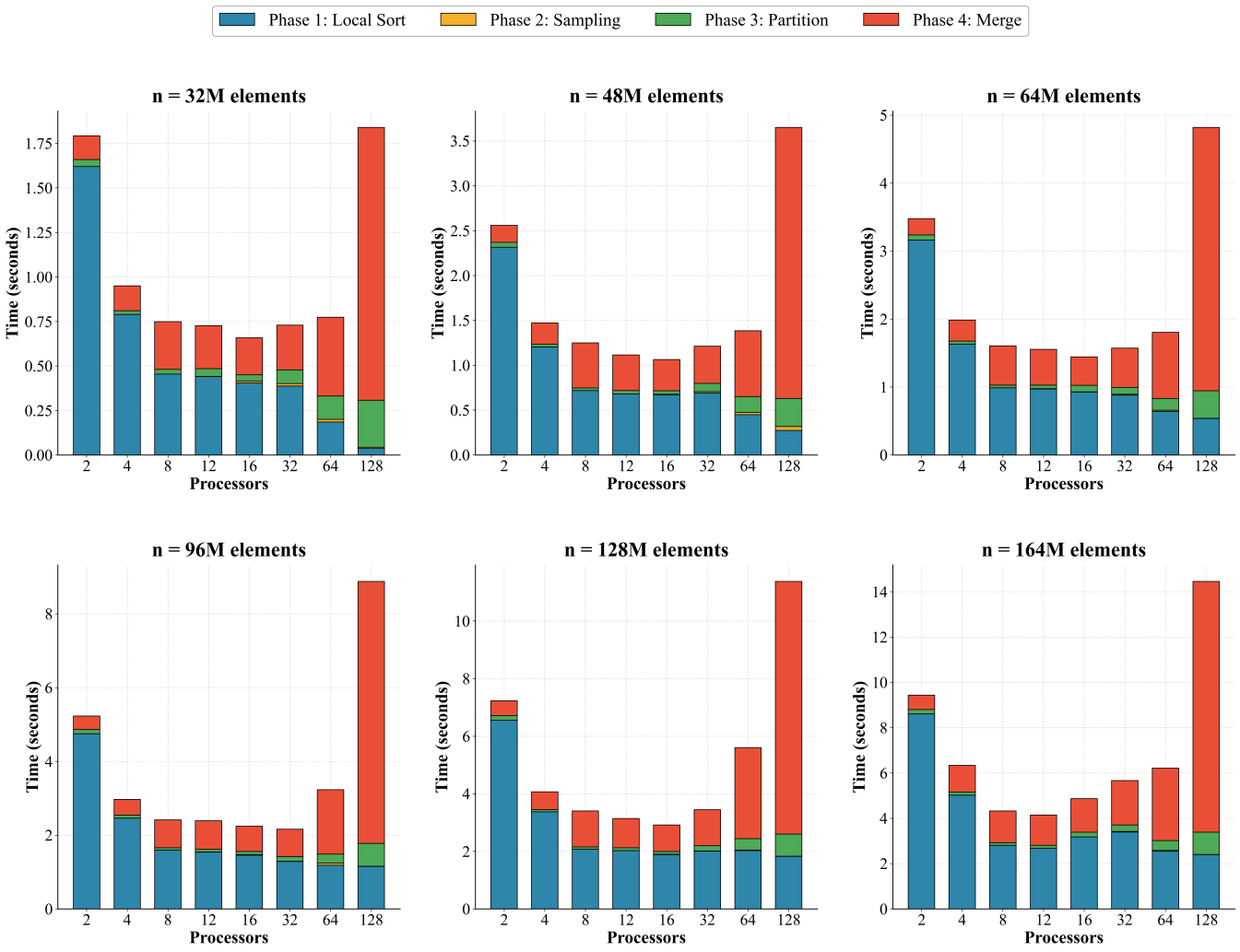

Figure 3: Phase breakdown for all array sizes. Phase 4 (merge) dominates at high p.

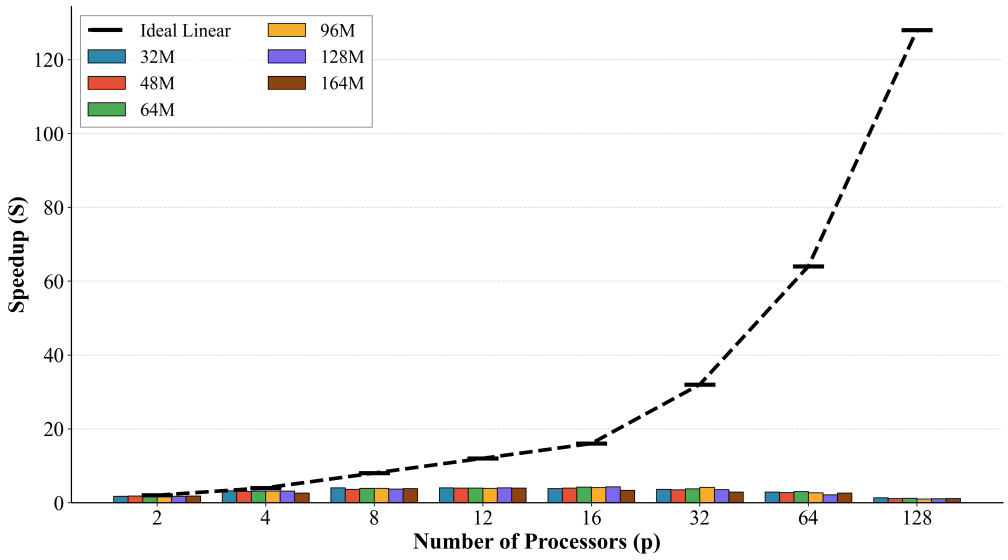

Figure 4: Actual speedup vs ideal linear speedup.

Table 1: Execution Times (seconds)

Size	Sequential	2 thr	4 thr	8 thr	12 thr	16 thr	32 thr	64 thr	128 thr
32M	3.12	1.81	0.97	0.77	0.78	0.81	0.86	1.07	2.29
48M	4.66	2.58	1.50	1.28	1.17	1.17	1.34	1.68	4.16
64M	6.42	3.50	2.02	1.65	1.62	1.51	1.69	2.14	5.34
96M	9.63	5.27	3.02	2.47	2.47	2.34	2.31	3.53	9.45
128M	13.02	7.30	4.15	3.48	3.25	3.03	3.62	6.04	11.97
164M	17.14	9.55	6.47	4.45	4.29	5.06	5.95	6.52	15.03

Note: thr = threads

Table 2: Speedup (relative to sequential)

Size	2 thr	4 thr	8 thr	12 thr	16 thr	32 thr	64 thr	128 thr
32M	1.72	3.21	4.07	4.01	3.87	3.62	2.91	1.36
48M	1.81	3.11	3.64	3.97	3.98	3.47	2.77	1.12
64M	1.83	3.18	3.90	3.97	4.25	3.80	3.00	1.20
96M	1.83	3.19	3.90	3.90	4.11	4.17	2.73	1.02
128M	1.78	3.14	3.74	4.01	4.30	3.60	2.16	1.09
164M	1.79	2.65	3.85	3.99	3.38	2.88	2.63	1.14

Note: thr = threads

Table 3: Phase Times: n=32M (seconds)

# Threads	Phase 1	Phase 2	Phase 3	Phase 4	Total Time
2	1.62	<0.01	0.04	0.13	1.81
4	0.79	<0.01	0.02	0.14	0.97
8	0.45	<0.01	0.03	0.27	0.77
12	0.44	<0.01	0.04	0.24	0.78
16	0.40	0.01	0.04	0.21	0.81
32	0.39	0.01	0.08	0.25	0.86
64	0.19	0.01	0.13	0.44	1.07
128	0.04	0.01	0.26	1.53	2.29

Table 4: Phase Times: n=48M (seconds)

# Threads	Phase 1	Phase 2	Phase 3	Phase 4	Total Time
2	2.31	<0.01	0.06	0.19	2.58
4	1.20	<0.01	0.03	0.24	1.50
8	0.72	<0.01	0.03	0.50	1.28
12	0.68	<0.01	0.04	0.40	1.17
16	0.67	0.01	0.04	0.35	1.17
32	0.69	0.01	0.09	0.41	1.34
64	0.45	0.02	0.18	0.74	1.68
128	0.27	0.04	0.31	3.02	4.16

Table 5: Phase Times: n=64M (seconds)

# Threads	Phase 1	Phase 2	Phase 3	Phase 4	Total Time
2	3.16	<0.01	0.08	0.24	3.50
4	1.63	<0.01	0.04	0.31	2.02
8	0.99	<0.01	0.04	0.58	1.65
12	0.97	0.01	0.05	0.53	1.62
16	0.93	<0.01	0.10	0.42	1.51
32	0.88	0.01	0.10	0.58	1.69
64	0.64	0.02	0.17	0.97	2.14
128	0.54	<0.01	0.40	3.87	5.34

Table 6: Phase Times: n=96M (seconds)

# Threads	Phase 1	Phase 2	Phase 3	Phase 4	Total Time
2	4.75	<0.01	0.12	0.36	5.27
4	2.46	<0.01	0.08	0.43	3.02
8	1.59	<0.01	0.06	0.75	2.47
12	1.54	<0.01	0.08	0.77	2.47
16	1.46	0.01	0.09	0.68	2.34
32	1.29	<0.01	0.13	0.74	2.31
64	1.19	0.06	0.24	1.74	3.53
128	1.15	0.01	0.62	7.10	9.45

Table 7: Phase Times: n=128M (seconds)

# Threads	Phase 1	Phase 2	Phase 3	Phase 4	Total Time
2	6.55	<0.01	0.16	0.52	7.30
4	3.37	<0.01	0.08	0.61	4.15
8	2.07	<0.01	0.08	1.25	3.48
12	2.02	<0.01	0.10	1.02	3.25
16	1.89	0.01	0.10	0.92	3.03
32	2.01	0.01	0.18	1.25	3.62
64	2.02	0.02	0.39	3.17	6.04
128	1.82	0.01	0.77	8.78	11.97

Table 8: Phase Times: n=164M (seconds)

# Threads	Phase 1	Phase 2	Phase 3	Phase 4	Total Time
2	8.60	<0.01	0.20	0.63	9.55
4	5.02	<0.01	0.13	1.18	6.47
8	2.80	<0.01	0.11	1.41	4.45
12	2.67	<0.01	0.13	1.34	4.29
16	3.17	<0.01	0.21	1.48	5.06
32	3.39	0.02	0.29	1.95	5.95
64	2.54	0.04	0.43	3.20	6.52
128	2.39	0.01	0.99	11.07	15.03