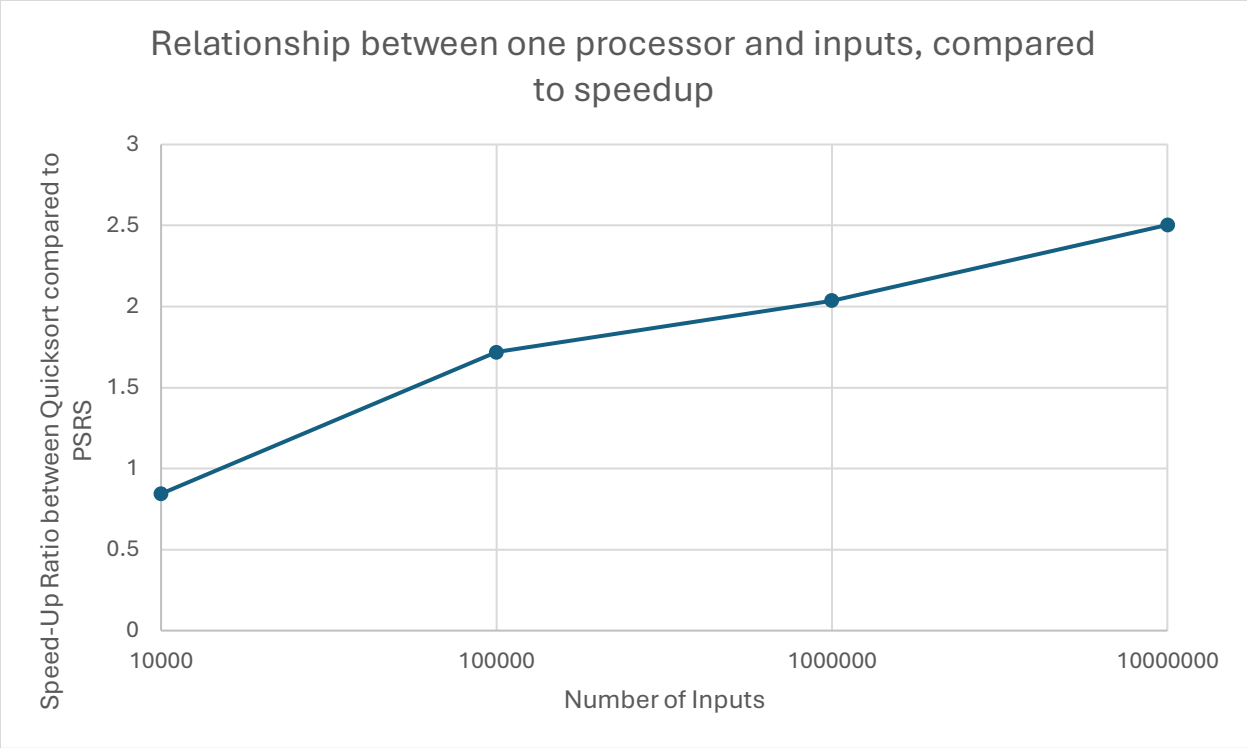
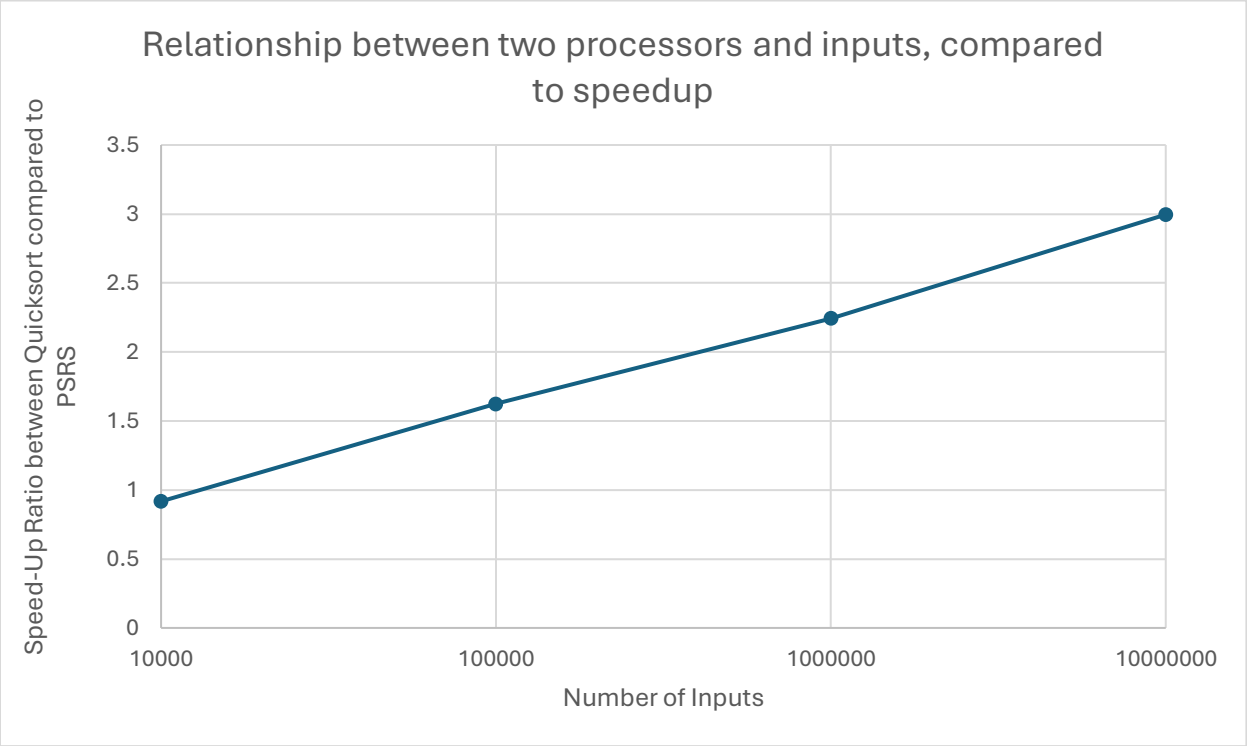
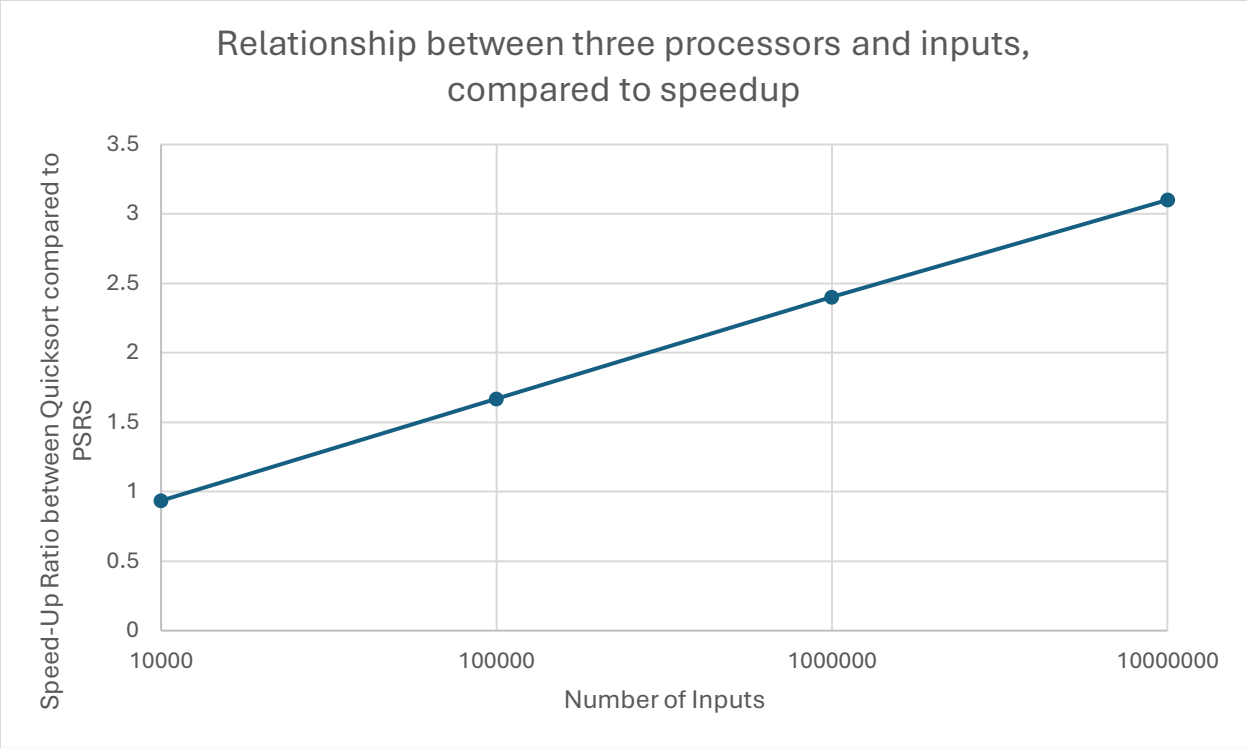
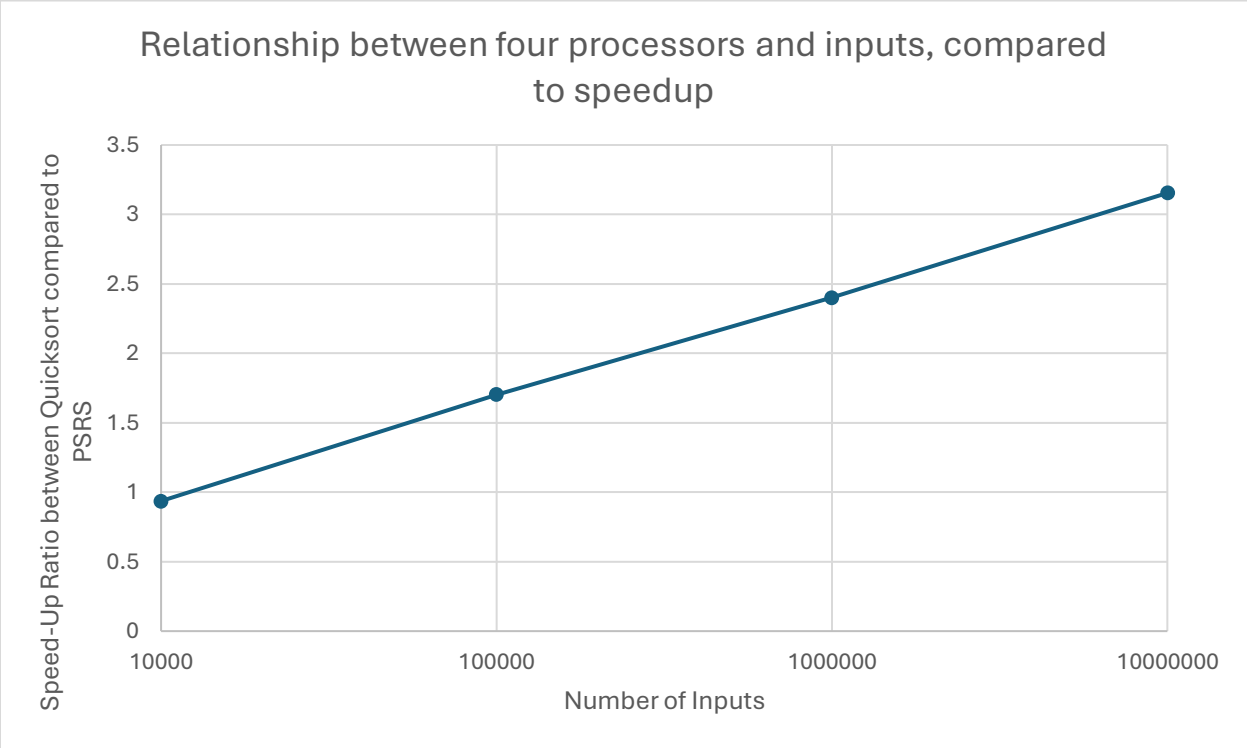
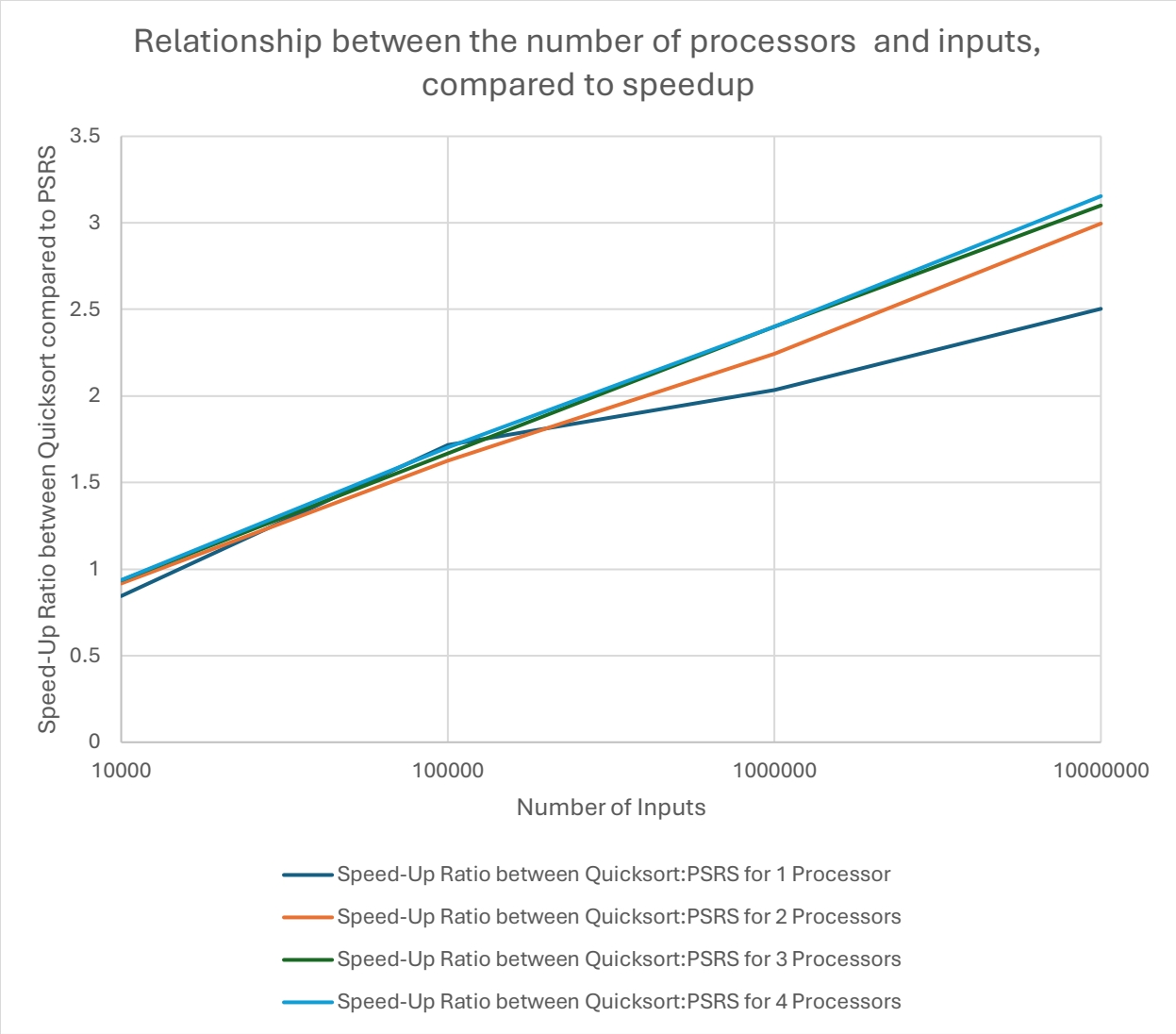

Relationship between one processor and inputs, compared to speedup
Speed-Up Ratio between Quicksort compared to PSRS
3
2.5
2
1.5
1
0.5
0
10000
100000
1000000
10000000
Number of Inputs


Relationship between two processors and inputs, compared to speedup
Speed-Up Ratio between Quicksort compared to PSRS
3.5
3
2.5
2
1.5
1
0.5
0
10000
100000
1000000
10000000
Number of Inputs


Relationship between three processors and inputs, compared to speedup
Speed-Up Ratio between Quicksort compared to PSRS
3.5
3
2.5
2
1.5
1
0.5
0
10000
100000
1000000
10000000
Number of Inputs


Relationship between four processors and inputs, compared to speedup
Speed-Up Ratio between Quicksort compared to PSRS
3.5
3
2.5
2
1.5
1
0.5
0
10000
100000
1000000
10000000
Number of Inputs


Relationship between the number of processors and inputs, compared to speedup
Speed-Up Ratio between Quicksort compared to PSRS
3.5
3
2.5
2
1.5
1
0.5
0
10000
100000
1000000
10000000
Number of Inputs
Speed-Up Ratio between Quicksort:PSRS for 1 Processor
Speed-Up Ratio between Quicksort:PSRS for 2 Processors
Speed-Up Ratio between Quicksort:PSRS for 3 Processors
Speed-Up Ratio between Quicksort:PSRS for 4 Processors

Results by processor count (analysing speedup)

1 process (p=1).
Speedup is < 1 across all input sizes (~0.84–0.94). With no parallelism, PSRS still pays its fixed costs (sampling, pivot handling, partition bookkeeping), so it is slower than straight quicksort.

2 processes (p=2).
Speedup improves to ~1.62–1.70 as N grows. Once each rank has enough local work, the cost of one all-to-all exchange and the final merge is properly distributed. This is the first routine where PSRS consistently beats sequential quicksort.

3 processes (p=3).
Speedup rises to ~2.04–2.40. Communication volume grows with p, but for $N \geq 10^5$ the extra concurrency outweighs it. Curves begin to bend, as because pivot selection, all-to-all, and local merge phases do not scale perfectly.

4 processes (p=4).
Best overall times: ~2.50–3.15× speedup at $N=10^6$–10^7. Scaling is good but not linear (ideal would be 4×). Diminishing returns come from:

1) More messages in the all-to-all
2) Imperfect load balance from regular sampling
3) The final k-way merge, which is O(n log p) locally.

Interpretation for how PSRS steps affect speedup

- **Step 1 (local sort).** Helpful to scale: each rank sorts ~N/p elements in O((N/p) log(N/p)). This is where most of the parallel savings come from.
- **Step 2–3 (sampling + pivoting).** Overhead grows with p (rank 0 sorts p^2 samples and broadcasts p−1 pivots). For small N this dominates; for large N it's negligible but still caps speedup as p increases.
- **Step 4 (partition + all-to-all).** Dominant communication cost. Each rank sends/receives O(N/p) data split across p peers. Latency (many small messages) and bandwidth (total bytes) both increase with p.

- **Step 5 (k-way merge).** Each rank merges p sorted runs totaling ~N/p. Even with an O(n log p) merge, this is a serial section on each process and contributes to the non-linear scaling at higher p.

Takeaways

1. PSRS is slower than sequential quicksort for small N due to fixed overheads (sampling, pivoting, all-to-all, merge).
2. For moderate–large N, PSRS outperforms sequential quicksort and strong scaling up to 4 processes (≈3.15x speedup at 10^7 inputs).
3. Sub-linear scaling is expected: Amdahl's serial fractions (pivoting/merge) and the communication pattern (All to all) limit ideal speedup. Load imbalance from regular sampling further reduces efficiency.

This aligns with the PSRS design goal for MIMD systems: good performance for large inputs with a simple, analyzable communication pattern, and predictable diminishing returns as p increases.