Query Processing System Report

Jairik "JJ" McCauley

December 28, 2025

1 Samples

For benchmarking query execution runtimes and computing performance measurements, timing samples were collected for each implementation (Serial, OpenMP, and OpenMPI) using dataset sizes of 50,000 and 1,000,000 records. For performance metrics, serial implemented were sampled 3 times to improve overall consistency/edge cases, while parallel implementations were tested using 1, 2, 4, and 6 cores. Additionally, all runtimes are measured in *seconds*.

Serial Runtime Samples

50k	**1M**
0.0219	2.7390
0.0782	2.8672
0.0532	2.7462

Average for 1,000,000 Tuples: $2.7841s$

OpenMP Runtime Samples

Cores	**50k**	**1M**
1	0.0229	0.7121
2	0.0218	0.6795
4	0.0222	0.5826
6	0.0243	0.4979

OpenMPI Runtime Samples

Cores	**50k**	**1M**
1	0.0192	0.5706
2	0.0113	0.3645
4	0.0133	0.2894
6	0.0143	0.2292

Potential Limitations

Due to the complexity of the data (many features and indexes), the dataset sizes were limited when testing. In a real-world scenario (which I unfortunately could not implement), these datasets would be stored on the disk and not in memory, greatly reducing the cold-start runtimes (as data would be saved to disk as B+ trees) and total capabilities (current implementation must load everything into memory). While a 20,000,000 sample dataset was created for benchmarking, it unfortunately took too long when testing to be included within these samples.

This invokes limitations within the results, as smaller datasets may encapsulate more I/O or communication overhead among the different cores. Additionally, these sub-second run-times are significantly more inconsistent. Despite these limitations, however, we can still draw some interesting performance results.

2 Speedup & Efficiency Metrics

OpenMP (1M Dataset)

Cores	Runtime (s)	Speedup	Efficiency
1	0.7121	3.91	3.91
2	0.6795	4.10	2.05
4	0.5826	4.78	1.19
6	0.4979	5.59	0.93

OpenMPI (1M Dataset)

Processes	Runtime (s)	Speedup	Efficiency
1	0.5706	4.88	4.88
2	0.3645	7.64	3.82
4	0.2894	9.62	2.41
6	0.2292	12.15	2.02

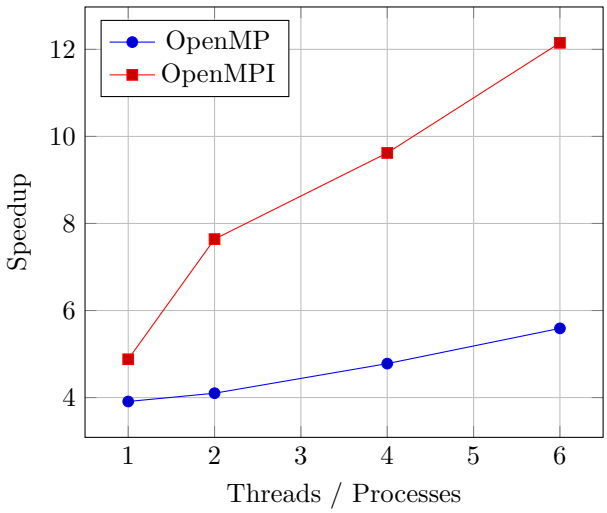

Figure 1: Speedup vs. Parallelism

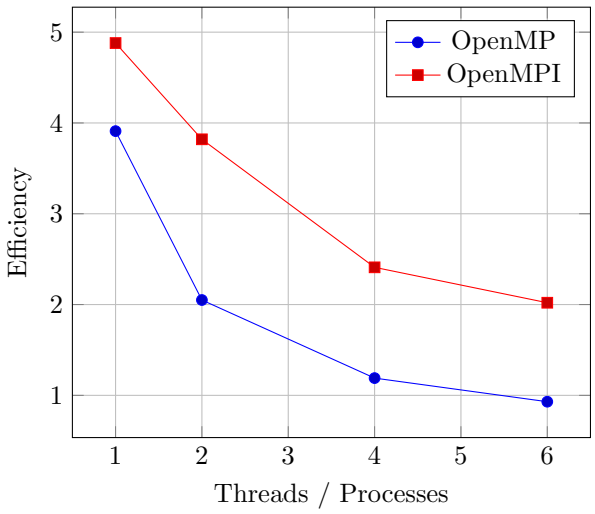

Figure 2: Efficiency vs. Parallelism

3 Parallelism Conclusions Based on Amdahl's Law (Strong Scaling)

From the measured 1 million tuple dataset runs, the OpenMP implementation reached it's optimal speedup at six threads, however reached it's maximum efficiency at 1 thread. Similarly, OpenMP showcased increasing speedup as the process count grew (more substantial of a curve than OpenMP), however showed a declining efficiency. This aligns with Amdhal's law, which states that while parallel performance improves initially, it diminishes due to serial portions and synchronization overhead. Therefore, the optimal thread configuration is likely a lower thread count, likely around 2-3, where speedup remains significant while efficiency is not too degraded.

4 Program Scaling with Thread Count (Weak Scaling)

To answer this question, scaled small datasets (50,000, 100,000, 200,000, 300,000) were generated and paired with the thread counts of 1, 2, 4, and 6. Additionally, sequential executions are noted for reference. Below are the results:

Threads/Processes	**Problem Size**	**Serial (s)**	**OpenMP (s)**	**OpenMPI (s)**
1	50,000	.0136	.0279	.0161
2	100,000	.1504	.0693	.0363
4	200,000	.3295	.1483	.0643
6	300,000	.4509	.2008	.1259

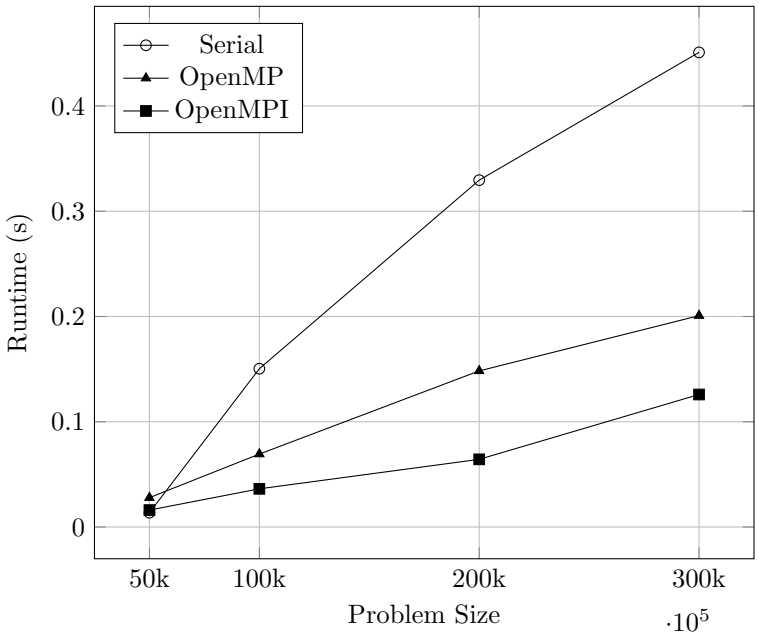

Figure 3: Runtime growth relative to increasing problem size under weak scaling.

In an ideal weak-scaling scenario, runtimes would remain constant while problem size and threads proportionally increase (which they do in this scenario). This is supported by the control serial implementation, which roughly grows linearly. However, the parallel programs do not remain flat, having a small linear increase in rumtime as problem size/threads increase. This points to potential weak-scaling degradation.

5 Conclusions

From these performance results, we can clearly see that both OpenMP and MPI parallel implementations are more efficient than the serial implementations, showing better weak and strong scaling. However, it is clear that the **MPI** implementation is the fastest and most optimized within our scenario.