Umm Al-Qura University

College of computing

Department of Computer Science and Artificial Intelligence

جامعة أم القرى
UMM AL-QURA UNIVERSITY

CS4612 – Parallel & Distributed Computing

Final Project Report

Confirmed Quality Score (CQS) Analysis of Mobile Phones & Accessories Reviews

Team Members:

Mada Alotaibi – 444001223

Retaj Mohammed – 444006831

Shahad Altalhi - 444001817

Noora Saad – 444001562

Arwa Sultan – 444001818

Date: 8\12\2025

1. Introduction

This project focuses on applying both parallel and distributed computing techniques to analyze large scale customer review data. The goal is to compute a Confirmed Quality Score (CQS) for mobile phone products and accessories using the Wilson Lower Bound statistical method. The project includes three implementations: a serial CPU version, a shared-memory parallel version using OpenMP, and a distributed memory version using MPI. The main objective is to evaluate performance, scalability, and efficiency across different execution models.

2. Problem Description & Dataset

Problem Description:

The goal of this project is to compute a Confirmed Quality Score (CQS) for products and their corresponding categories using customer reviews. The CQS is calculated using the Wilson Lower Bound (95% confidence) based on the proportion of positive reviews (ratings greater than or equal to 4 stars). Neutral reviews (3 stars) may be excluded or handled separately.

For each product:

- Count the number of positive, negative, and total reviews.

- Compute $p = \frac{\text{positives}}{\text{total reviews}}$.

- Apply the Wilson Lower Bound formula to calculate the CQS score.

Dataset Description:

Dataset: Amazon Cell Phones & Accessories Reviews
Source: Amazon Reviews Dataset
Size: Approximately 580 MB (~3.4 million reviews, 346K products)
Notes: This dataset is open-source and publicly available for academic use.

3. System & Hardware Specifications

The experiments were conducted on a laptop equipped with the following specifications:

Component	Details
CPU Model	Intel Core i9-13900H
Number of Cores	14 cores / 20 threads
RAM Size	16.00 GB RAM
Operating System	64-bit Windows
Distributed Environment	Microsoft MPI

4. Design & Implementation

4.1 Serial Implementation

In the serial version, the program reads the entire file line by line and stores all lines in the **`lines`** array.
For each line, it extracts the **ASIN** and overall rating using the **`extract_asin_overall`** function.

If a product already exists in the products array, its review count is updated; otherwise, it is added as a new entry.
After processing all lines, the **CQS** for each product is calculated using the **`wilson_lower_bound`** function.

Computation time is measured specifically for the processing section using **`omp_get_wtime().`**

$$\text{Wilson}_{\text{lower}}\left(\hat{p}, n, \tfrac{\alpha}{2}\right) \equiv w^{-} = \frac{1}{1 + z_{\alpha}^{2}/n}\left(\hat{p} + \frac{z_{\alpha}^{2}}{2n} - \frac{z_{\alpha}}{2n}\sqrt{4n\hat{p}(1-\hat{p}) + z_{\alpha}^{2}}\right)$$

This approach works well for small files and a limited number of products but becomes very slow when handling large datasets.

4.2 OpenMP Parallel Implementation

The OpenMP parallel version follows the same logic as the serial implementation, but lines are distributed across threads using **`#pragma omp for`**.

Each thread maintains a local array **`local_lists[tid]`** to prevent race conditions during processing.
Once all threads finish, the local lists are merged into the main products array on the master thread.

Computation time is measured before and after the processing section using `t_compute_start` and `t_compute_end.`
This approach significantly improves performance for large files and multiple threads, reducing execution time compared to the serial version.

4.3 MPI Distributed Implementation

In the MPI distributed version, the file is divided among processes (ranks) based on the rank number (`line_index % size == rank`), so each process reads only a portion of the lines.

Each process computes a local list of products using the same method as in the serial version.
All local lists are then gathered at the master process (Rank 0) using **MPI_Gatherv**, which merges them into the final product list and calculates the overall **CQS**.

Both computation time and communication time between processes are measured using `MPI_Wtime.`
This approach is highly suitable for very large datasets across multiple devices or cores, although communication overhead between processes must be considered.

5. Experimental Setup & Methodology

- **Number of trials:** Each experiment was repeated three times to account for variability in execution time.
- **OpenMP:** Thread counts tested: 17, 18, 20, 21, 22 with different **scheduling types** (`static, dynamic, guided`).
- **MPI:** Number of ranks (size) depends on the experiment; each rank reads a part of the file.
- **Tuning Parameters:** Key parameters include the chunk size for OpenMP scheduling, initial size of the product arrays, and local arrays allocated per thread to prevent race conditions.
- **Strong Scaling:** We used the same dataset size while increasing the number of threads/processes to evaluate performance improvement.

6. Results

Threads	Schedule	Chunk	Run1	Run2	Run3	Avg sec	Mun	Speedup=Ts/Tp	Efficiency=S/p
20	static	998	246.375	260.519	258.731	255.203	4.253	43.55	2.177
20	dynamic	998	248.534	267.053	228.154	247.914	4.132	44.83	2.241
20	guided	998	227.981	229.704	226.784	228.156	3.803	48.72	2.435
18	static	1109	258.449	231.034	244.157	244.547	4.076	45.45	2.525
18	dynamic	1109	235.229	234.711	231.793	233.911	3.899	47.52	2.625
18	guided	1109	227.448	230	248.177	235.208	3.920	47.26	2.625
17	static	1175	229.09	227.556	224.353	227.000	3.783	48.96	2.88
17	dynamic	1175	243.277	255.469	284.934	261.227	4.354	42.55	2.502
17	guided	1175	233.737	232.009	228.081	231.276	3.855	48.06	2.826
21	static	951	256.117	224.432	234.285	238.278	3.971	46.65	2.221
21	dynamic	951	120.614	225.322	223.969	189.968	3.166	58.51	2.786
21	guided	951	236.655	226.126	243.546	235.442	3.924	47.21	2.248
22	static	908	238.738	226.486	227.65	230.958	3.849	48.12	2.187
22	dynamic	908	231.778	223.622	224.441	226.614	3.777	49.05	2.229
22	guided	908	223.783	223.076	223.047	223.302	3.722	49.77	2.262

Table 1: OpenMP Tuning Results (Execution Time, Speedup, and Efficiency)

P	Run1	Run2	Run3	Avg
comm 20	0.067	0.085	0.082	0.078
comp 20	90.149	90.345	90.216	90.237
total 20	871.983	1142.856	1067.459	1027.432
comm 16	0.032	0.038	0.041	0.037
comp 16	337.809	105.111	105.245	182.722
total 16	1276.316	1110.055	1158.140	1181.504
comm 8	0.012	0.014	0.011	0.012
comp 8	245.556	133.125	130.610	169.763
total 8	535.957	551.878	436.692	508.176
comm 4	0.007	0.007	0.007	0.007
comp 4	258.796	256.214	253.431	256.147
total 4	491.208	492.733	496.344	493.428

Table 2: MPI Performance Results (Computation, Communication, and Total Time)

Time (sec)	Run Serie
3617.156	Run1
21256.492	Run2
8470.828	Run3
11114.82533	Average

Table 3: Serial Execution Time Results

7. Performance Analysis & Discussion

7.1 Serial Baseline

The serial implementation serves as the baseline for performance comparison. The execution time varies depending on the dataset, with an average of approximately 11115 seconds. This version reads and processes the entire file line by line without parallelization. While it works adequately for small datasets, it becomes very slow for large files. This baseline is used to calculate speedup and efficiency for OpenMP and MPI implementations.

7.2 OpenMP Parallel Performance

OpenMP parallelization was tested with thread counts of 17, 18, 20, 21, and 22, using static, dynamic, and guided scheduling. For each configuration, multiple runs were executed, and average execution times were measured.

The average efficiency for each thread count was approximately:

273.7% for 17 threads, 259.7% for 18 threads, 228.5% for 20 threads, 241.8% for 21 threads, and 222.7% for 22 threads**.**

Observations:

The highest average efficiency was observed with 17 threads, while the lowest occurred with 22 threads. Efficiency does not decrease uniformly as the number of threads increases, which indicates the presence of overhead such as thread management, memory bandwidth limitations, and synchronization costs. Therefore, careful tuning of thread counts and scheduling policy is essential to achieve optimal performance.

7.3 Comparison of Serial, OpenMP, and MPI

The **serial version** is the baseline it works correctly but takes a very long time, especially with large datasets there is no parallel overhead, but processing millions of records can be too slow.

The **OpenMP version** reduces execution time by splitting work across multiple threads when tuned properly (for example, 17 threads with the right scheduling), it reaches high efficiency (~274%), showing that CPU cores are well used using more threads may lower efficiency slightly due to thread management, memory limits, and synchronization.

The **MPI version** works well on multiple nodes or cores by dividing the dataset between ranks it also speeds up processing a lot compared to serial efficiency is a bit lower because of communication between ranks, but it still works well for very large datasets.

In summary, parallelization greatly improves performance. OpenMP is ideal for single, multi-core machines, while MPI is better suited for distributed systems with multiple machines proper tuning of threads, scheduling, and workload is essential to achieve the best performance and reduce overhead.

7.4 Throughput Analysis

In addition to raw runtime, we also report throughput, defined as the number of products processed per second (total products / total runtime). For the full dataset of 319,678 products, the serial implementation achieved a throughput of only 28.76 products/second. The OpenMP version significantly improved to 1681.99 products/second, while the distributed MPI version reached 647.96 products/second. This confirms that both parallel implementations can handle large datasets much more efficiently than the serial baseline, with OpenMP providing the highest throughput on our multi core machine.

Throughput	Products	Time	Method
28.76	319678	11114.82	Serial
1681.99	319678	189.968	Parallel
647.96	319678	493.427	Distributed

Table 4: Throughput Analysis Results

7.5 Correctness Validation Between Implementations

We compared the output files generated by:

- Serial version
- OpenMP version
- MPI version

By matching the rows ASIN – Total – Positive – CQS, All three versions produced identical results.

ASIN	Total	Positive	CQS
011040047X	1	0	0.0000
0110400550	31	12	0.2373
0195866479	3	1	0.0615
0214514706	1	0	0.0000
0214614700	1	1	0.2065
0214714705	1	1	0.2065
0594033918	6	4	0.3000
0641554214	6	6	0.6097
0989890945	2	1	0.0945
103824174X	1	1	0.2065
103984068X	1	1	0.2065
1049830512	1	1	0.2065
1059130386	1	1	0.2065
1059236222	1	0	0.0000
1059274930	3	3	0.4385
1059359189	1	1	0.2065
1059370514	1	0	0.0000
1059375362	1	1	0.2065

ASIN	Total	Positive	CQS
011040047X	1	0	0.0000
0110400550	31	12	0.2373
0195866479	3	1	0.0615
0214514706	1	0	0.0000
0214614700	1	1	0.2065
0214714705	1	1	0.2065
0594033918	6	4	0.3000
0641554214	6	6	0.6097
0989890945	2	1	0.0945
103824174X	1	1	0.2065
103984068X	1	1	0.2065
1049830512	1	1	0.2065
1059130386	1	1	0.2065
1059236222	1	0	0.0000
1059274930	3	3	0.4385
1059359189	1	1	0.2065
1059370514	1	0	0.0000
1059375362	1	1	0.2065
1059556936	1	1	0.2065

ASIN	Total	Positive	CQS
011040047X	1	0	0.0000
0110400550	31	12	0.2373
0195866479	3	1	0.0615
0214514706	1	0	0.0000
0214614700	1	1	0.2065
0214714705	1	1	0.2065
0594033918	6	4	0.3000
0641554214	6	6	0.6097
0989890945	2	1	0.0945
103824174X	1	1	0.2065
103984068X	1	1	0.2065
1049830512	1	1	0.2065
1059130386	1	1	0.2065
1059236222	1	0	0.0000
1059274930	3	3	0.4385
1059359189	1	1	0.2065
1059370514	1	0	0.0000
1059375362	1	1	0.2065
1059556936	1	1	0.2065

ASIN	Total	Positive	CQS
0110400550	31	12	0.2373
0195866479	3	1	0.0615
0214514706	1	0	0.0000
0214614700	1	1	0.2065
0214714705	1	1	0.2065
0594033918	6	4	0.3000
0641554214	6	6	0.6097
0989890945	2	1	0.0945
103824174X	1	1	0.2065
103984068X	1	1	0.2065
1049830512	1	1	0.2065
1059130386	1	1	0.2065
1059236222	1	0	0.0000
1059274930	3	3	0.4385
1059359189	1	1	0.2065
1059370514	1	0	0.0000
1059375362	1	1	0.2065
1059556936	1	1	0.2065
1059598515	1	1	0.2065

Figure 1: Output Consistency Across All Implementations

7.6 Summary of Tuning Decisions

- Based on the full tuning results, the best OpenMP performance was achieved with 21 threads using dynamic scheduling, which produced the lowest average execution time (189.968 seconds).
- Increasing the thread count beyond this point reduced efficiency due to memory-bandwidth limitations and cache contention.
- For MPI, the best total execution time was obtained with 4 processes (≈493.4 seconds). Increasing the number of ranks to 8, 16, and 20 did not improve performance; instead, the additional communication and merging overhead outweighed the benefits of extra processes.
- Adjusting the chunk size in OpenMP improved load balancing by distributing uneven JSON line workloads more effectively across threads.

8. Conclusion

This project showed that parallel computing gives much better performance than the serial version which takes a very long time with large data. OpenMP worked very well on a single machine when the number of threads was chosen correctly. MPI was also effective for processing large datasets on multiple machines even though it needs extra time for communication. Overall using parallel methods clearly helped improve speed and efficiency.

9. References

- *Amazon Reviews Dataset. (n.d.). Retrieved from https://nijianmo.github.io/amazon/index.html*
- *Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. Journal of the American Statistical Association, 22(158), 209–212.*
- *Zhou, X. H. (2008). Improving interval estimation of binomial proportions. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences.*
- *Binomial proportion confidence interval. (n.d.). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval*
- *ChatGPT. (2025). Used to assist in researching ideas and identifying suitable datasets.*