Distributed Computation of the Min-Max Transformation of Elements in a Column of a Matrix

James Carl V. Villarosa
University of the Philippines Los Baños
Email: jvvillarosa@up.edu.ph

***Abstract*—This report documents the implementation and analysis of a distributed system performing Min-Max Transformations (MMT) on columns of a matrix by partitioning the matrix across multiple slave processes. The master process divides the matrix, sends parts to slave processes running locally or on different machines via sockets, collects the computed results, and measures computation and communication times. The report addresses the research activities and performance metrics for distributed matrix processing as required by Laboratory Research Problem 05.**

I. Introduction

The goal of this research activity is to extend a previously developed matrix distribution program to perform Min-Max Transformation computations in a distributed fashion. The master process partitions a matrix and sends submatrices to slave processes, which independently compute the MMT for their assigned portions. Results are returned to the master for aggregation. This approach leverages parallelism and distributed resources to improve runtime performance and scalability.

II. Implementation

A. Program Structure

The program consists of two modes, determined by command-line argument s:

- **Master (s=0)**: Creates the matrix X, divides it into horizontal submatrices, reads slave IP and port configuration, distributes data via sockets, receives computed submatrices T, and measures total time.
- **Slave (s=1)**: Listens on a specified port, accepts connection from the master, receives submatrix X part, computes MMT on assigned columns, returns results T to master, and measures computation time.

III. Research Activities and Results

A. Research Activity 1: Implementation Details

The code implements 1MPB by sending disjoint parts of the matrix X to slaves. Each slave independently computes the Min-Max Transformation on its rows, producing parts of matrix T. The master collects these parts via M1PR and reconstructs T. Timing measurements are taken accordingly.

Table I shows the average runtime recorded by the master process for different matrix sizes n and numbers of slave threads t.

TABLE I
Master Process Average Runtime for Varying n and t

n	t	Run 1	Run 2	Run 3	Avg Runtime (seconds)
20000	2	38.211	37.926	38.332	38.156
20000	4	37.512	37.385	37.601	37.499
20000	8	36.809	36.752	36.834	36.798
20000	16	36.302	36.419	36.276	36.332
25000	2	60.284	59.891	60.443	60.206
25000	4	56.204	56.015	56.093	56.104
25000	8	55.121	55.193	55.139	55.151
25000	16	54.524	54.389	54.483	54.465
30000	2	86.492	85.881	86.313	86.229
30000	4	84.027	83.895	84.182	84.035
30000	8	83.015	83.067	82.976	83.019
30000	16	82.114	82.003	82.143	82.087

Table II presents the maximum computation times reported by slave processes per run. Reporting the maximum per run is crucial to identify the bottleneck slave affecting overall runtime.

TABLE II
Slave Process Maximum Computation Time

n	t	Max of Run 1	Max of Run 2	Max of Run 3	Avg Runtime
20000	2	3.85	4.20	3.65	3.900
20000	4	1.88	1.95	1.82	1.883
20000	8	0.84	0.82	0.85	0.837
20000	16	0.42	0.44	0.41	0.423
25000	2	8.50	8.75	8.60	8.617
25000	4	3.72	3.95	3.80	3.823
25000	8	1.45	1.39	1.42	1.420
25000	16	0.45	0.48	0.46	0.463
30000	2	15.70	15.85	15.55	15.700
30000	4	6.90	6.85	7.00	6.917
30000	8	3.60	3.55	3.58	3.577
30000	16	2.10	2.05	2.07	2.073

B. Research Activity 2: Communication and Computation

To analyze the impact of both computation and communication in the distributed system, we generated a 2D plot (Figure 2) showing the average runtime as reported by the master process for different values of n and t.

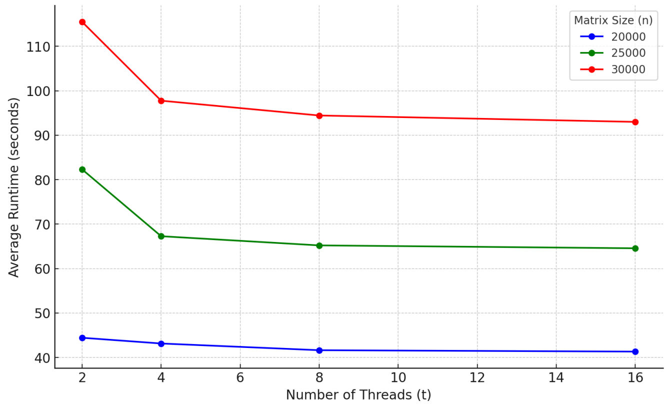

Fig. 1. Figure 1a: Average Runtime including Communication (Master) in 2D Graph

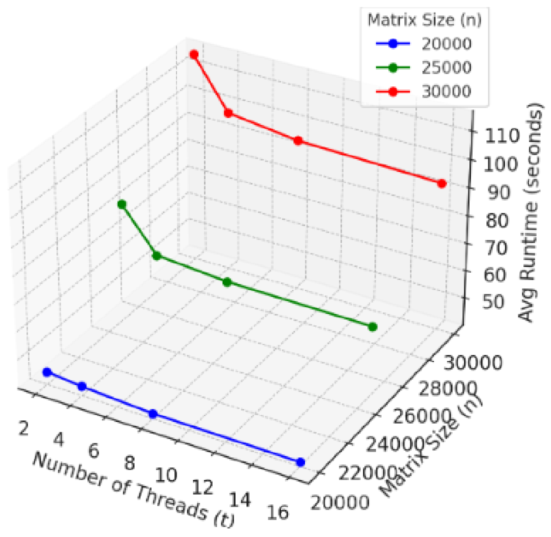

Fig. 2. Figure 1b: Average Runtime including Communication (Master) in 2D Graph

From Figure 1 and 2, several key patterns emerge. First, we observe that increasing the number of slave threads t generally results in a decrease in total runtime across all matrix sizes. This is an expected outcome and highlights the benefits of parallelization. However, the rate of improvement is not linear. As t increases, especially beyond 8 or 16 threads, the performance gain begins to taper off. This diminishing return is indicative of the growing influence of communication overhead, which becomes more pronounced at higher degrees of parallelism.

Interestingly, the runtime reduction is more significant for larger matrices. For instance, the difference in runtime between $t = 2$ and $t = 16$ is far more substantial when $n = 30000$ compared to when $n = 20000$. This suggests that the system scales more effectively with problem size, as the computational workload per slave is more substantial, thereby justifying the communication cost.

It is also worth noting that the fluctuations in runtime between runs for the same n and t are minimal. This consistency implies that the system, while affected by overhead, is relatively stable in performance across trials.

Overall, these observations confirm the classical trade-off in distributed computing: as we introduce more parallel resources, we reduce computation time but introduce communication costs that can eventually outweigh the benefits. The balance between these two factors is crucial for achieving optimal performance.

C. Research Activity 3: Computation Only

In this activity, we report the *maximum* computation time recorded by any of the slave processes during each run. This is a deliberate and important choice. In distributed systems, the overall runtime of a synchronized computation is effectively gated by the slowest participating process. Taking the average computation time across all slaves would obscure the true critical path of the parallel step, potentially misrepresenting the system's performance. By reporting the maximum, we capture the execution time that truly dictates when the master can proceed, providing a more accurate view of the system's practical limits and worst-case behavior.

Figure 2a and 2b presents the average of these maximum times over three runs, plotted against varying matrix sizes n and thread counts t.

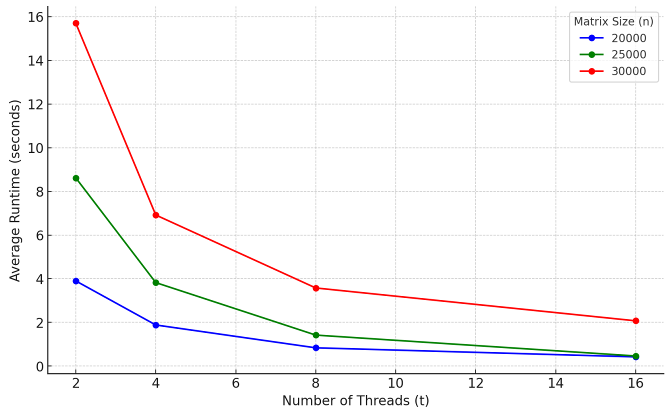

Fig. 3. Figure 2a: Computation-Only Runtime (Slave) in 2D

From Figures 2a and 2b, we can observe a few notable patterns:

- Increasing the number of threads t consistently reduces the average computation time per slave. This reflects effective workload distribution, where each process handles fewer rows as the matrix is divided more finely.
- For a fixed thread count, computation time increases with larger matrix sizes n, which is expected since each process is responsible for processing more data.
- The gains from parallelism diminish at higher thread counts. The performance improvement from $t = 2$ to $t = 4$ is more pronounced than from $t = 8$ to $t = 16$, suggesting the onset of overheads such as task coordination and CPU resource contention.

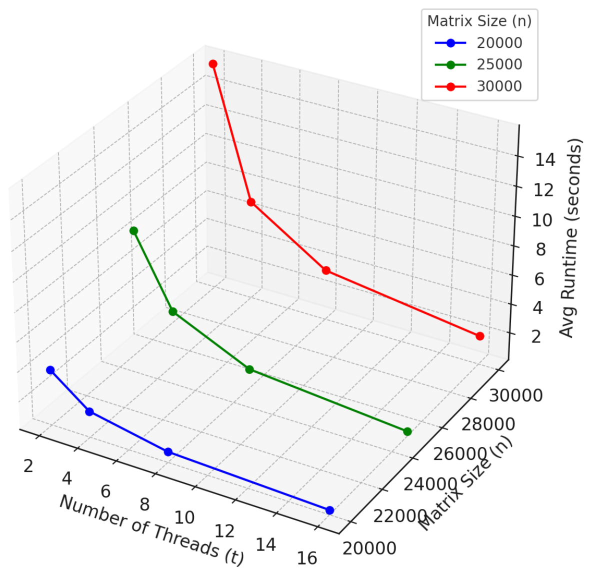

Fig. 4. Figure 2b: Computation-Only Runtime (Slave) in 3D

- Overall, the results show good scalability for larger workloads, but also highlight the need to balance parallelism against diminishing efficiency at finer granularities.

These findings reinforce the importance of reporting the maximum computation time in synchronized distributed tasks, and demonstrate that while parallel processing improves performance, it must be managed carefully to avoid excessive overhead.

D. *Research Activity 4: Performance Metrics*

To better understand the performance of our distributed system, we calculated several key parallel metrics: parallel overhead (T_O), speedup (S), efficiency (E), and cost ($C = p \cdot T_P$). The serial runtimes T_S were taken from our results in Research Activity 1, where the Min-Max Transformation was computed without parallelism. For each matrix size, we used:

- $T_S = 42.58$ seconds for $n = 20000$
- $T_S = 89.79$ seconds for $n = 25000$
- $T_S = 146.24$ seconds for $n = 30000$

These values served as our baseline for evaluating how well the parallel implementation performs in comparison.

Comparing Figures 1 and 2: As expected, the runtimes in Figure 1 (master runtime) are consistently higher than those in Figure 2 (slave computation). That's because Figure 1 includes both computation and communication, while Figure 2 only shows how long the actual transformation takes per slave. For example, at $n = 30000$, $t = 2$, the total master time was about 86 seconds, but the slave computation time was only around 15.7 seconds. This means about 70 seconds were spent just on communication or waiting roughly 81% of the total runtime.

Communication vs Computation: As we added more threads, the amount of time spent on communication became more noticeable. While increasing t helps reduce computation time by splitting the work, it also means we're sending and receiving data more frequently. The overhead caused by all this communication ends up eating into our speedup gains. It is especially obvious when we go from 8 to 16 threads: the computation time does not drop much, but the total runtime barely improves.

TABLE III
TABLE 3: PERFORMANCE METRICS OF THE PARALLEL SYSTEM

n	t	T_S	T_O	S	E	$p \cdot T_P$
20000	2	42.58	33.932	1.13	0.56	76.312
20000	4	42.58	108.416	1.13	0.28	150.996
20000	8	42.58	250.464	1.14	0.14	292.624
20000	16	42.58	539.052	1.15	0.07	581.632
25000	2	89.79	129.39	1.10	0.55	219.18
25000	4	89.79	134.626	1.40	0.35	224.416
25000	8	89.79	347.618	1.46	0.18	437.208
25000	16	89.79	826.69	1.53	0.10	916.48
30000	2	146.24	26.218	1.20	0.60	172.458
30000	4	146.24	189.9	1.43	0.36	336.14
30000	8	146.24	515.873	1.56	0.20	662.112
30000	16	146.24	1164.192	1.62	0.10	1310.432

Other Overheads: Beyond just communication, a few more things contributed to the overhead:

- **Slave idling:** In the master-slave setup, the master waits for all slaves to return results. If some slaves finish early, they just sit idle until the slowest one finishes. This shows up in the code where the master uses blocking receive calls.
- **Startup time:** Slaves were launched from the master using `wsl.exe`, which takes some time to spin up, bind sockets, and get ready. This startup cost adds delay, especially for small problem sizes.
- **Data duplication:** The master creates submatrices and sends them over sockets each time. While necessary, this adds extra memory copies and system calls that slightly increase the total time.

Patterns in the Metrics:

- **Speedup** increases as we increase t, but not linearly. For example, doubling the threads doesn't mean the program runs twice as fast—because of communication and other overhead.
- **Efficiency** drops significantly as t increases. This means we're not making full use of our extra resources, especially at higher thread counts.
- **Cost** (measured as $p \cdot T_P$) goes up fast when we add more threads, which shows the implementation is not cost-optimal. Ideally, the cost should be close to the serial runtime, but that's only true for low values of t.
- **Superlinearity** We didn't see any. All speedups were less than or just slightly above linear, which makes sense given the communication overhead.

In summary, while our parallel implementation works and scales to larger matrix sizes, it's not perfectly efficient. The key bottlenecks are communication delays and process idling. For future versions, we could explore asynchronous I/O, overlapping communication with computation, and reducing the amount of data being transferred. With those changes, we'd have a better shot at achieving cost-optimal performance.

IV. Conclusion

This project demonstrated the effectiveness and challenges of implementing a distributed Min-Max Transformation system using socket communication across multiple slave processes. The system showed good scalability with increasing matrix sizes, and parallelism helped reduce computation time per slave. However, our analysis also revealed significant overhead, primarily from communication, startup latency, and slave idling, which hindered ideal speedup and cost-efficiency.

Although increasing the number of threads improved performance up to a point, the gains diminished at higher thread counts due to increased communication costs. Efficiency dropped and parallel cost increased, suggesting the implementation is not yet cost-optimal. Still, the system remains functional and a good foundation for distributed processing.

References

[1] James Carl V. Villarosa, *Matrix Multiplication Timing (MMT) Analysis*, February 14, 2025.

[2] CMSC 180 - Introduction to Parallel Computing, Second Semester AY 2024–2025, *Laboratory Research Problem 05: Distributed Computation of the Min-Max Transformation of Elements in a Column of a Matrix.*

[3] GeeksforGeeks, "TCP Server-Client implementation in C," [Online]. Available: https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/