Performance Analysis – Parallel Matrix Multiplication

Md Fuzail

February 19, 2026

Analysis of row_wise_matrix_mult.c

The baseline `row_wise_matrix_mult.c` implementation utilizes a Master-Worker paradigm. While this pattern is beneficial for irregular workloads, it is highly inefficient for the dense, predictable nature of matrix multiplication and introduces severe bottlenecks.

- **Communication Bottlenecks:** The most detrimental flaw is the communication topology. The Master process (Rank 0) reads and broadcasts Matrix B row-by-row inside a loop (`MPI_Bcast` N times). This induces massive latency penalties because the network must perform handshakes for every single row instead of streaming a single continuous block. Furthermore, Rank 0 becomes a central choke point, serially sending Matrix A rows and receiving Matrix C results one by one. As the cluster scales, the Master process cannot service the workers fast enough.
- **Memory and Serial Initialization:** Initialization is entirely serial. Rank 0 must allocate and populate 100% of the memory for matrices A, B, and C (roughly 1.5 GB for $N = 8000$). This heavily restricts the maximum problem size to the RAM limits of a single node and wastes the compute power of the other cluster nodes during startup.

EduMPI Observation of row_wise_matrix_mult.c

- **Centralized Star Topology:** The EduMPI visualization displays a dense "Star" communication pattern. Traffic is heavily centralized around Rank 0. As demonstrated in Figure 1, the Master process acts as a severe central choke point.
- **High Wait Times (Red State):** The abundance of red execution states indicates that worker processes are starved for data. Because of the row-by-row broadcast, workers spend the majority of their lifecycle blocked in `MPI_Recv`, waiting for the overloaded Master to deliver their workload.
- **Heavy Point-to-Point Overhead:** As demonstrated in Figure 2, a continuous, overwhelming stream of `MPI_Send` and `MPI_Recv` messages. This illustrates the massive network traffic required to distribute the matrices row-by-row, validating that communication latency,rather than actual computation is the primary limiter of the unoptimized implementation.

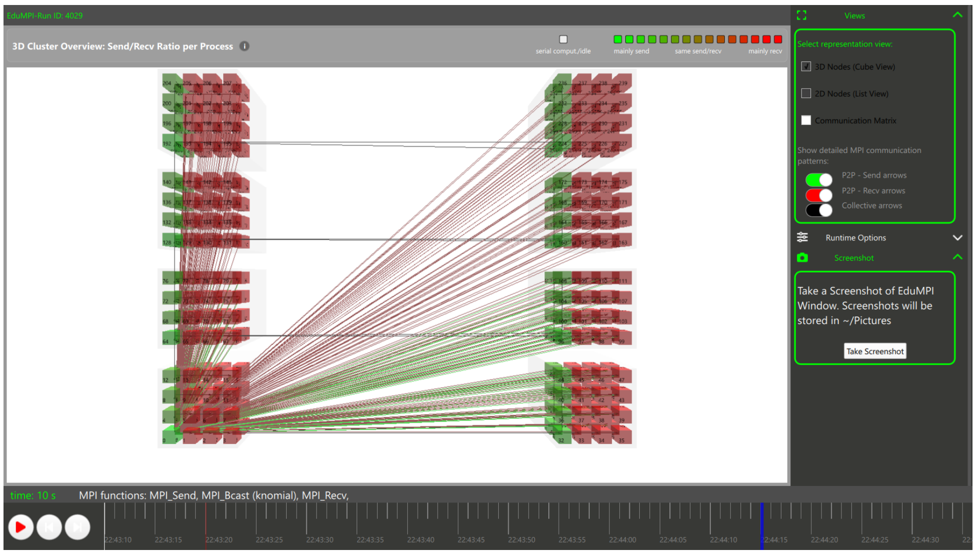

EduMPI-Run ID: 4029
3D Cluster Overview: Send/Recv Ratio per Process
serial comput./idle
mainly send
same send/recv
mainly recv
Views
Select representation view:
3D Nodes (Cube View)
2D Nodes (List View)
Communication Matrix
Show detailed MPI communication patterns:
P2P - Send arrows
P2P - Recv arrows
Collective arrows
Runtime Options
Screenshot
Take a Screenshot of EduMPI Window. Screenshots will be stored in ~/Pictures
Take Screenshot
time: 10 s
MPI functions: MPI_Send, MPI_Bcast (knomial), MPI_Recv,

Figure 1: EduMPI visualization showing the network bottleneck radiating from Rank 0.

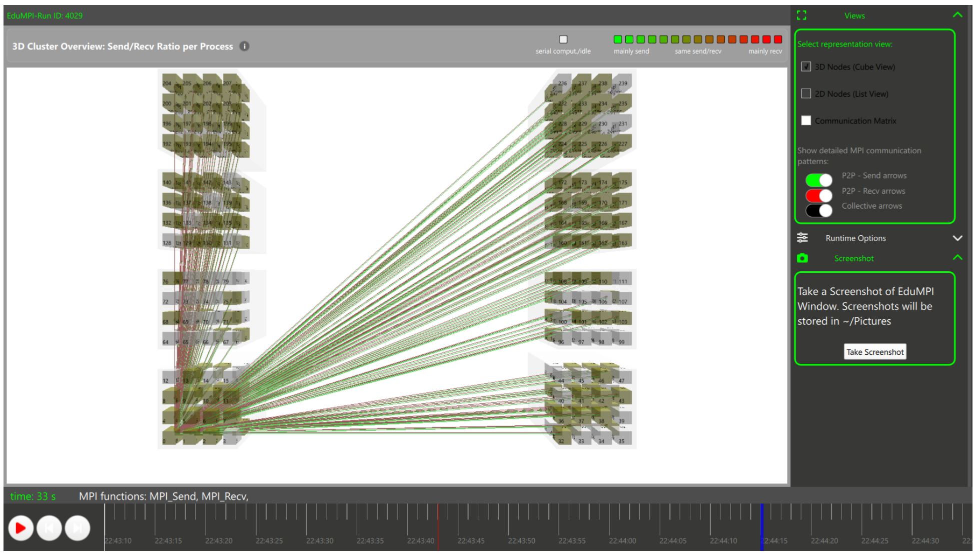


Figure 2: EduMPI visualization showing the heavy point-to-point overhead

Analysis of My Optimized Implementation

The optimized implementation I have used discards the Master-Worker model in favor of a **Static 1D Row-Block Decomposition** combined with a **Zero-Communication Initialization** strategy and **Cache-Efficiency**.

- **Parallel Stateless Initialization:** The most significant architectural change is the complete elimination of startup network communication. Using the deterministic nature of the provided PRNG (`my_rand`), every rank calculates a global offset and generates *only* its required horizontal slice of Matrix A locally. This achieves perfect $\mathcal{O}(N^2/P)$ scaling for Matrix A without a single `MPI_Scatter` call.
- **Redundant Local Generation (Matrix B):** Instead of broadcasting the 512 MB Matrix B over the network, every process concurrently generates the entire Matrix B in its local memory.

- **Cache-Friendly Computation:** To maximize spatial locality, the multiplication loop was refactored from the standard `i-j-k` order to a blocked (tiled) `i-k-j` order. This prevents erratic memory jumping, ensuring that both the CPU's caches are utilized to their maximum potential.
- **Minimal Network Topology:** The only communication step in the entire program occurs at the very end. An `MPI_Gatherv` operation is used to stitch the locally computed chunks of Matrix C back onto Rank 0 for the final checksum verification.

Comparison of Alternate Optimization Strategies

During development, the team evaluated two distinct approaches for handling the initialization of Matrix B. While my teammate implemented **Approach 1** (Distributed Generation with `MPI_Allgatherv`), my implementation explores **Approach 2** (Redundant Local Generation).

- **Approach 1 (Collective Communication):** This approach forces each node to compute a proportional slice of Matrix B and utilizes `MPI_Allgatherv` to stitch it together across all nodes. This scales the initialization workload perfectly to $\mathcal{O}(N^2/P)$, but introduces collective network communication overhead.
- **Approach 2 (Redundant Local Generation - My Implementation):** This approach attempts to circumvent the network bottleneck entirely. By accepting a redundant $\mathcal{O}(N^2)$ computation cost on each node, every process generates the full Matrix B locally, achieving zero network bandwidth consumption during the setup phase.

Conclusion on Strategies: Empirical testing revealed that **Approach 1 ultimately outperformed Approach 2** at higher node counts. While my implementation successfully eliminated network contention during initialization, it exposed a computational bottleneck: the sheer CPU expense of the pseudo-random number generator (`my_rand` and `concatenate`).

Generating an 8000×8000 matrix requires calculating 64 million random numbers. Because these functions involve complex bitwise shifts and 64-bit floating-point math, forcing every single node to perform this full $\mathcal{O}(N^2)$ calculation created a heavy fixed cost that did not scale down as process counts increased (Amdahl's Law).

Conversely, Approach 1 scaled the PRNG workload down to just 125,000 elements per rank at 512 processors. The benchmark results prove that on the Fulda HPC cluster, the high-bandwidth network is highly efficient. The time required to distribute 512 MB via `MPI_Allgatherv` is substantially lower than the CPU time required for a processor to redundantly generate 64 million PRNG values. Therefore, distributing the initialization workload yielded the superior speedup.

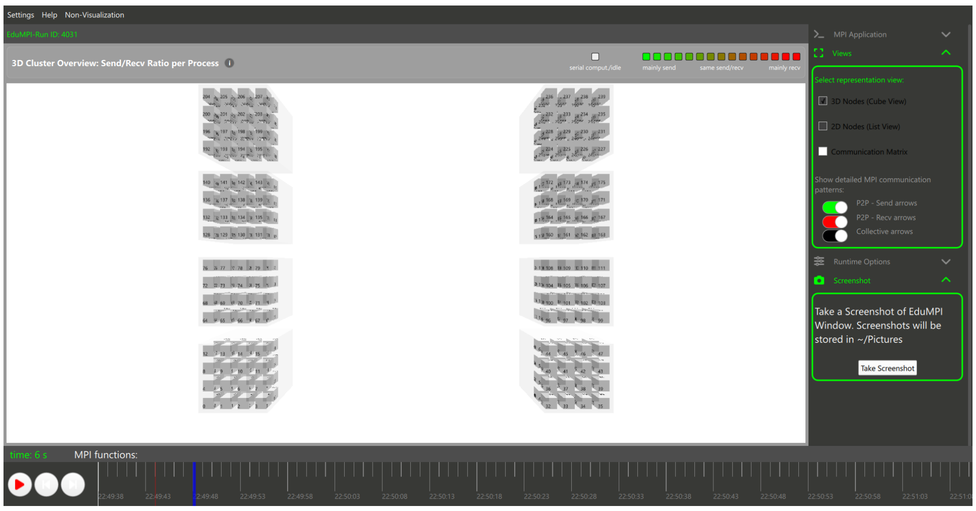
Verification and EduMPI Visualization of Optimized Code

- **Dominant Computation Phase (Grey State):** Prior to the final network operation, the EduMPI visualization demonstrates that processes spend the vast majority of their execution lifecycle in the "grey" state, indicating uninterrupted, isolated computation. This visually confirms that the network bottlenecks have been successfully removed, allowing all ranks to crunch their matrix multiplication workloads at maximum CPU efficiency without waiting for intermediate `MPI_Send` or `MPI_Recv` handshakes.
- **Clean Gather Topology:** As seen in the EduMPI visualizations, the chaotic network traffic is gone. The processes compute in isolation until the very end, where a single, clean `MPI_Gatherv` pulls the final data to Rank 0.

- **Verification:** To ensure mathematical correctness despite the decentralized initialization, the result was verified against the strict assignment constraints using a 64-bit floating-point modulo checksum:

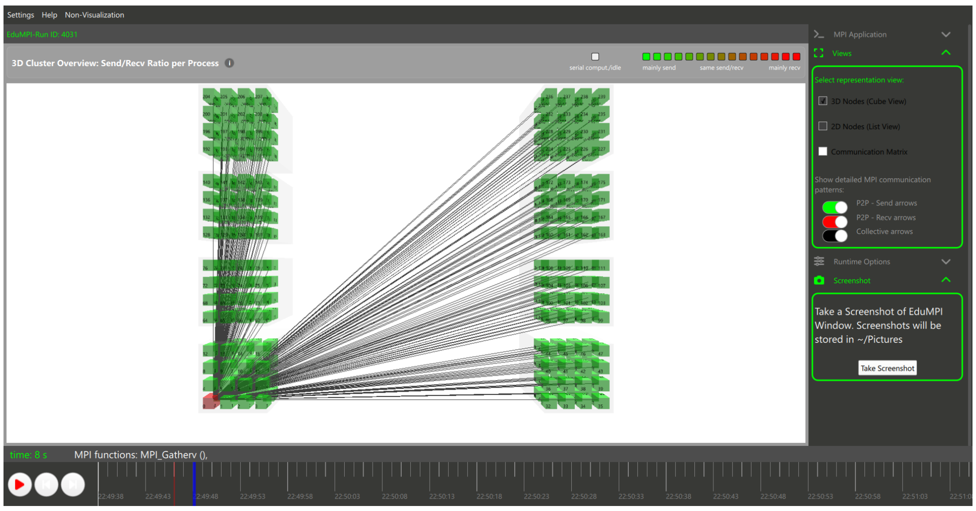
$$\text{checksum} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} C[i][j] \pmod{2^{64}} \tag{1}$$

  This guarantees that the optimized cache-tiling and redundant data generation perfectly match the serial mathematical expectations.



Take a Screenshot of EduMPI Window. Screenshots will be stored in ~/Pictures

Figure 3: EduMPI visualization showing no initial communication



Take a Screenshot of EduMPI Window. Screenshots will be stored in ~/Pictures

Figure 4: EduMPI visualization showing clean gather topology