Parallel Programming

Group K

Group K - Team Members

- Purushothaman, Logeshwari
  Matrikelnummer - 1536313 (fd0001575)
  Fox Algorithm

- Purushothaman, Pavithra
  Matrikelnummer - 1535949 (fd0001571)
  Cannon Algorithm

- Basu, Spandan
  Matrikelnummer - 1452130 (fdai7670)
  DNS Algorithm

Agenda

1. Introduction
2. Row Wise Algorithm
3. Fox Algorithm
4. Cannon Algorithm
5. DNS Algorithm
6. Performance Analysis
7. Conclusion
8. References

1. Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and various engineering applications. When dealing with large matrices, sequential execution becomes inefficient, leading to the need for parallel algorithms. **MPI (Message Passing Interface)** allows us to distribute the computation across multiple processes, significantly improving performance. In this presentation, we will discuss different parallel matrix multiplication approaches, including the row-wise method, Fox, Cannon, and DNS algorithms. We will also analyze their efficiency and compare them to determine the best approach under different conditions.

2. Row Wise Algorithm

The row-wise method is a straightforward parallel approach where each process is assigned one or more rows of the first matrix (A). Each process then computes the corresponding rows of the result matrix (C) by multiplying them with the entire second matrix (B), which is broadcasted to all processes. While this method is easy to implement, it can become inefficient due to excessive communication overhead and load imbalance when the number of processes increases.

2.1 Implementation of Row Wise Algorithm

The row-wise algorithm follows a unique approach. Instead of using a grid, it **divides the work row by row**. Each process is responsible for a few rows of matrix A, while the entire matrix B is broadcasted to all processes. Each process computes its assigned portion of the result and then sends it back to the root process for gathering.

While the row-wise method is simpler to implement, it has several disadvantages:

- **More communication overhead** - Since matrix B is broadcasted to all processes, it leads to redundant data transfers.
- **Load imbalance** - Some processes may finish their work quickly, while others are left waiting.
- **Not well-suited for large matrices** - As the matrix size increases, communication dominates the execution time, making it inefficient.

2.1 Implementation of Row Wise Algorithm (Cont...)

```
int main(int argc, char* argv[]){
    if(rank == 0){

        for(int i= 0; i<MATRIX_SIZE; i++){
            ierr = MPI_Bcast(matrix_b[i], MATRIX_SIZE, MPI_DOUBLE, 0, MPI_COMM_
        }

        int row_a_counter=0;
        for(int i= 1; i<size; i++){
            ierr = MPI_Send(matrix_a[i-1], MATRIX_SIZE, MPI_DOUBLE, i, i, MPI_(
            row_a_counter++;
        }
        for(int i=0; i<MATRIX_SIZE; i++){
            ierr = MPI_Recv(answer, MATRIX_SIZE, MPI_DOUBLE, MPI_ANY_SOURCE, MP
            int row = status.MPI_TAG;
            int sender = status.MPI_SOURCE;
            memcpy(matrix_c[row], answer, MATRIX_SIZE * sizeof(double));

            if(row_a_counter<MATRIX_SIZE){
                ierr = MPI_Send(matrix_a[row_a_counter], MATRIX_SIZE, MPI_DOUBL
                row_a_counter++;
            }
            else{
                //send message with tag 0 so that all the processes can stop ca
                ierr = MPI_Send(&matrix_c[0], MATRIX_SIZE, MPI_DOUBLE, sender,
            }
        }
```

Rank 0 broadcast the entire matrix B to all the other processes with **MPI_Bcast.**

Each row of matrix A is sent individually with **MPI_Send**.

Each rank receives one row of matrix A with **MPI_Recv** to perform the multiplication.

*** This causes load imbalance, delay, performance degradation

3. Fox Algorithm for Parallel matrix Multiplication

- The Fox algorithm is a parallel matrix multiplication technique that divides matrices into submatrices and uses a Cartesian topology for communication.
- It involves distributing submatrices to processes arranged in a grid.

Steps involved

1. Initialization
2. Local Multiplication and Shifting
3. Iteration
4. Result Gathering

3.1 Fox Algorithm Implementation in Task A

1. In Task A, the Fox algorithm was implemented for matrix multiplication using individual elements.
2. Each process handled a single element of the matrices A and B.

```
MPI_Scatter(A, 1, MPI_INT, &local_A, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

3. The algorithm used a **Cartesian topology** with row and column communicators for efficient data movement.
4. Elements were shifted and multiplied locally using **MPI_Bcast** and **MPI_Sendrecv_replace**.
5. Results were gathered at the root process to form the product matrix C.
6. Its implementation is limited by its handling of individual matrix elements, which **becomes inefficient for larger matrices**. This is because each process only handles a single element, leading to more communication between processes. For example, the code uses MPI_Scatter to distribute single elements.

3.2 Handling Larger Matrices in Task B - Fox

1. **Matrix Division:** Matrices are divided into submatrices to reduce communication overhead.Car
2. **Memory Allocation**: Allocating memory for submatrices instead of individual elements improves efficiency:

```
local_A = (int *)malloc(submatrix_size * submatrix_size * sizeof(int));
```

3. **Custom MPI Datatype:** A custom MPI datatype is created for efficient submatrix Communication:

```
MPI_Type_vector(submatrix_size, submatrix_size, N, MPI_INT, &block_type);
```

4. **Scattering Submatrices:** Submatrices are scattered to processes using the custom datatype:

```
MPI_Send(temp_buffer, submatrix_size * submatrix_size, MPI_INT, dest_rank, 0, cart_comm);
```

5. **Cartesian Topology:** Processes are arranged in a Cartesian topology to facilitate structured communication and computation which is implemented in task A aswell.
6. **Local Multiplication and Shifting:** Each process performs matrix multiplication on its submatrices and shifts them using MPI_Sendrecv_replace to maintain data locality

```
MPI_Sendrecv_replace(local_B, submatrix_size * submatrix_size, MPI_INT, dest, 0, src, 0, col_comm, MPI_STATUS_IGNORE);
```

3.3 Why this implementation for Task B- Fox ?

1. Cartesian Topology (MPI_Cart_create) - Ensures structured process communication, making it optimal for matrix operations. **Alternative:** Linear topology uses a sequential process arrangement, leading to inefficient data transfer, while graph topology lacks an inherent structured pattern.

2. Derived Data Types (MPI_Type_vector) - Reduces communication overhead and enhances memory efficiency. **Alternative:** Using **MPI_Send** for each element increases communication calls, while manual packing/unpacking requires additional memory handling. (Tried and difficult for me to debug when result matrix produced wrong values)

3. Blocking Point-to-Point Communication (MPI_Sendrecv_replace) - Guarantees reliable data transfer and simplifies execution. **Alternative:** Non-blocking communication (MPI_Isend/Irecv) requires explicit synchronization, while collective communication (MPI_Bcast) may introduce unnecessary overhead.

4. Suitability for Large Matrices - Enhances performance and scalability for large datasets **Alternative:** Element-wise transmission increases synchronization costs, and a centralized distribution approach (e.g., MPI_Scatter from a single rank) can create bottlenecks.

5. Efficiency and Maintainability - Improves code clarity and long-term maintainability. **Alternative:** Unstructured approaches require complex data reordering, while excessive manual memory management makes debugging harder.

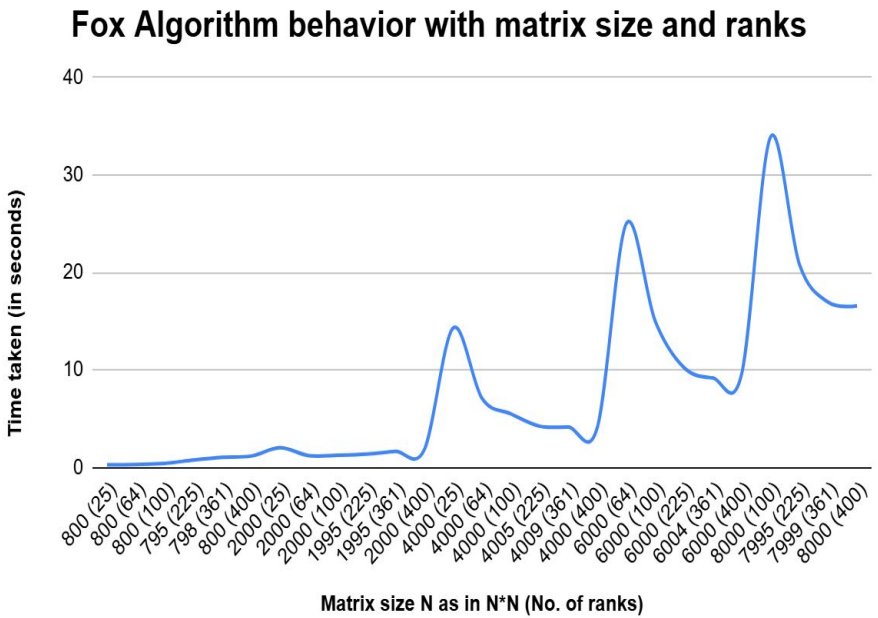
This approach optimizes performance, scalability, and maintainability, making it ideal for Fox algorithm implementation.

3.3.1 Performance of task b - Fox algorithm

Large Matrices & High Ranks: Large matrices (e.g., 8000x8000) perform better with **more ranks** due to even workload distribution, reducing computation time. **Communication overhead** increases with ranks, diminishing performance beyond a certain point (e.g., 400 ranks).

Overhead in Smaller Matrices: Smaller matrices (e.g., 800x800) show diminishing returns with **excessive ranks** due to increased communication overhead.**Optimal performance** with fewer ranks, reducing communication costs.

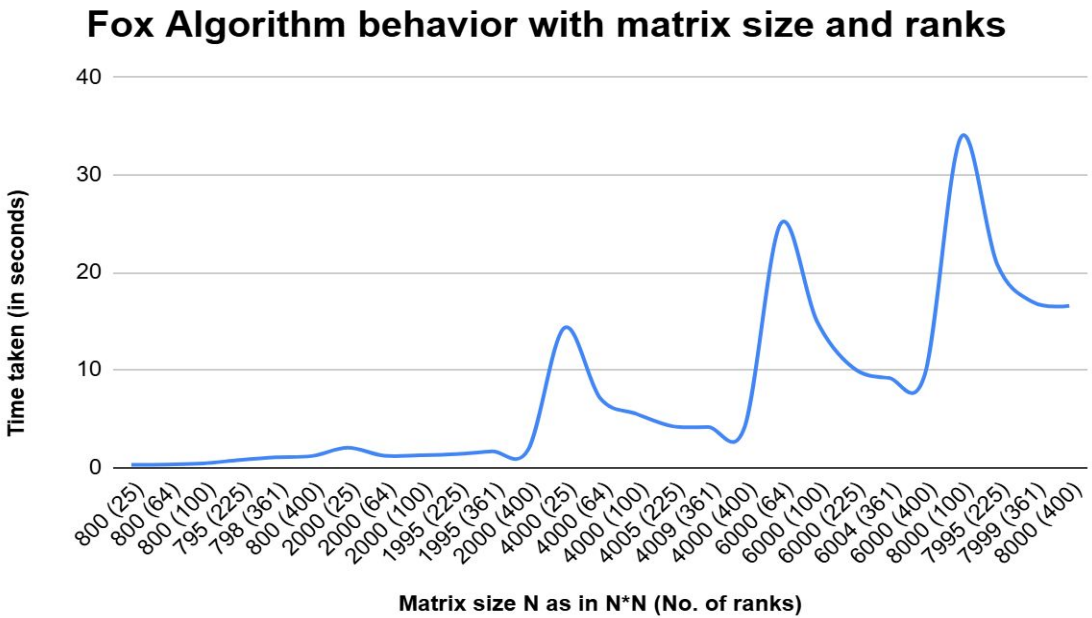
Non-Square Matrix Sizes: Irregular matrix sizes (e.g., 7999, 6004) cause slight inefficiencies due to load imbalance among ranks.



3.3.1 Performance of task b - Fox algorithm

Scaling Efficiency: Increasing ranks significantly improves execution time initially, but beyond a point (e.g., 400 ranks for 8000x8000), communication costs offset the benefits.

Low Rank Impact: With fewer ranks (e.g., 25, 64), performance drops significantly as each rank handles a larger computation, increasing execution time.



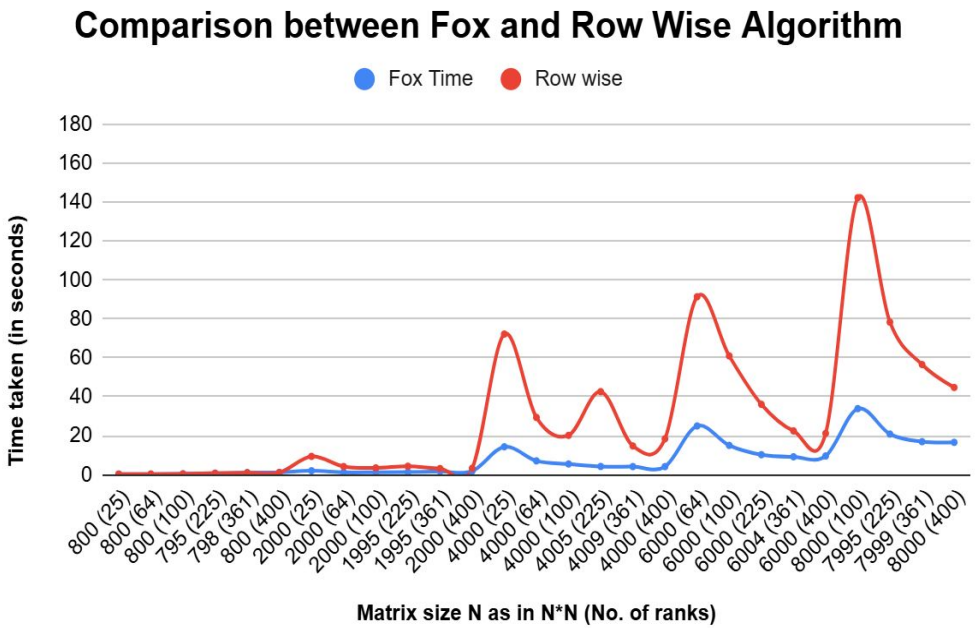
3.4 Performance - Fox Algorithm vs Row Wise Multiplication

Scalability: The Fox algorithm **scales better** with an increasing number of processes due to efficient matrix block distribution. Row-wise multiplication faces scaling limitations due to overhead and load imbalance.

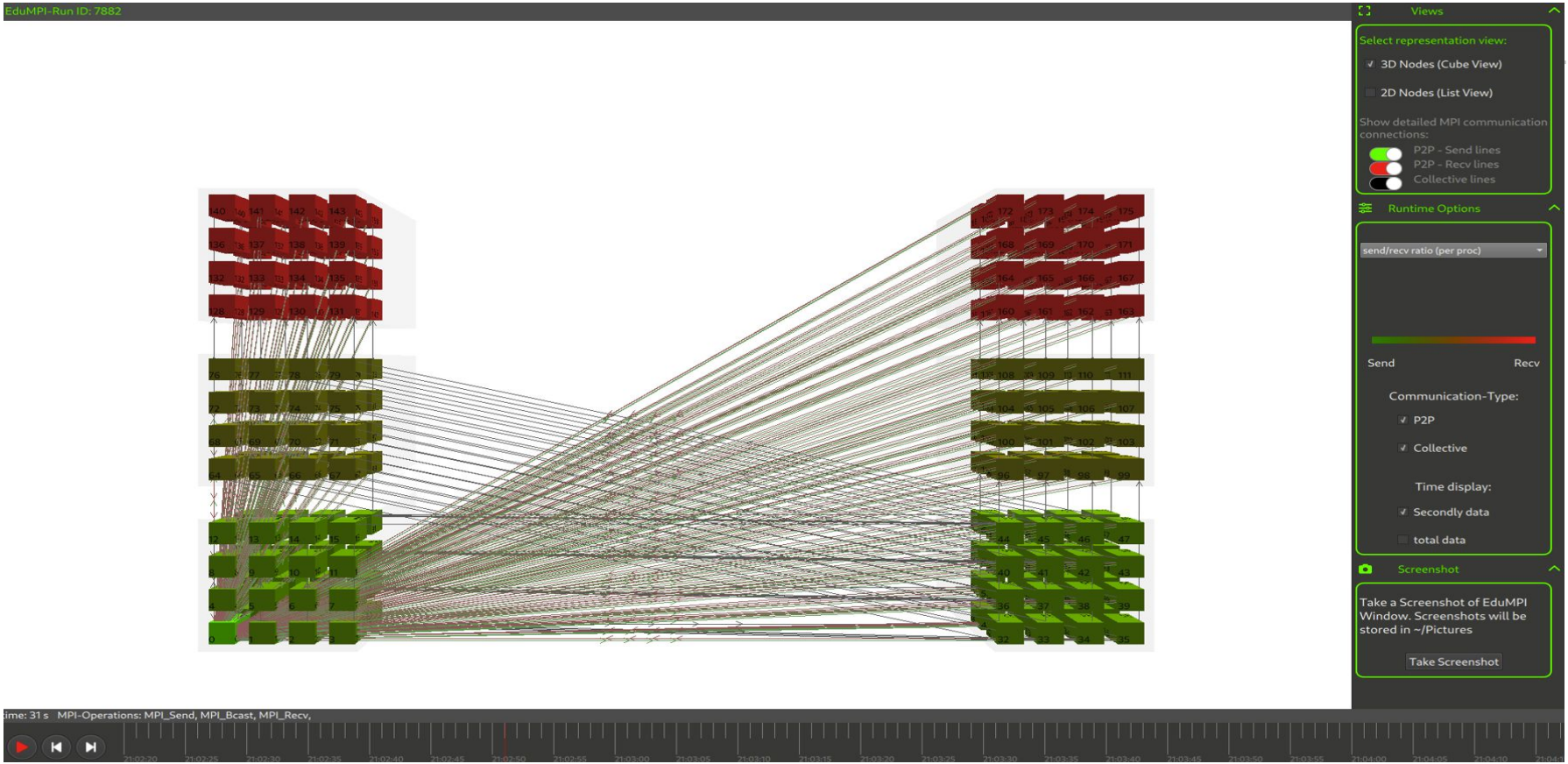
Speedup: Fox algorithm maintains **consistent speedup** as processes increase, while row-wise multiplication plateaus more quickly due to less efficient parallelization.

Memory Efficiency: Fox algorithm is slightly more memory-efficient, especially with large matrices, as it reduces the memory footprint through block-wise distribution compared to row-wise multiplication.

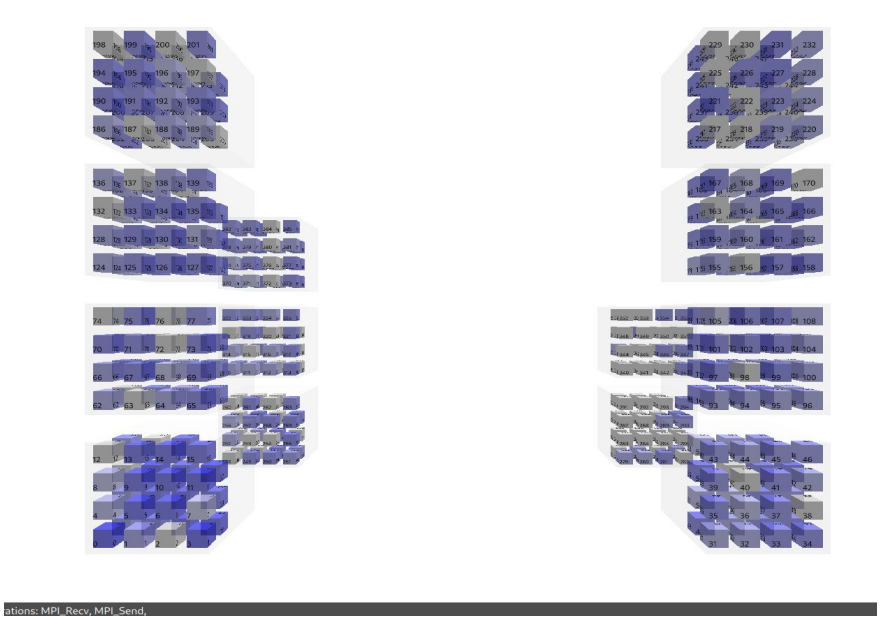
Overhead: Fox algorithm incurs higher communication overhead from block broadcasting and shifting, but it is spread across the grid and minimized with large matrices. Row-wise multiplication suffers from overhead during row transfers and may experience underutilization as processes exceed row numbers.

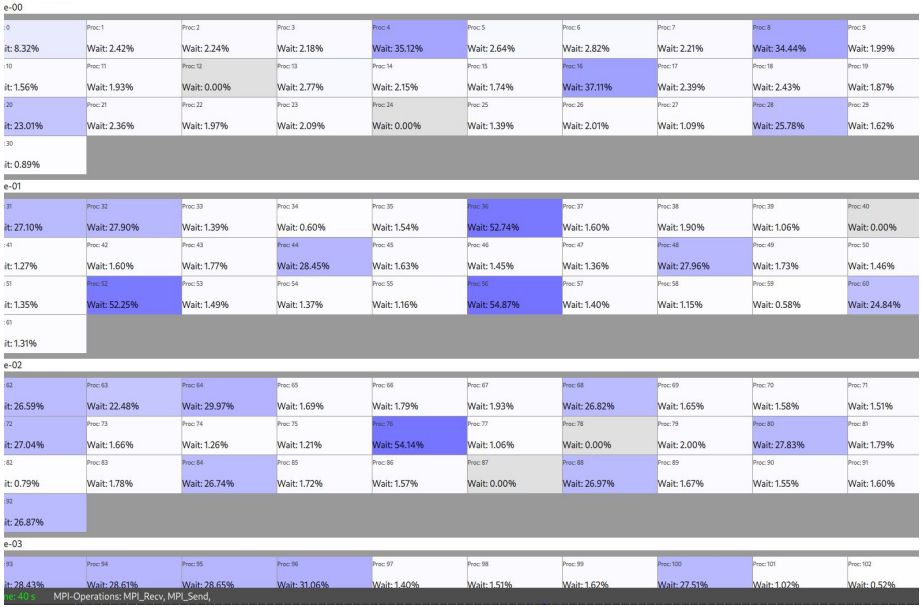


3.4.1 Visualization with EduMPI - Fox Algorithm 8000x8000 with 400 Ranks



3.4.2 Row Wise Algorithm Execution (8000*8000 with 400 ranks)



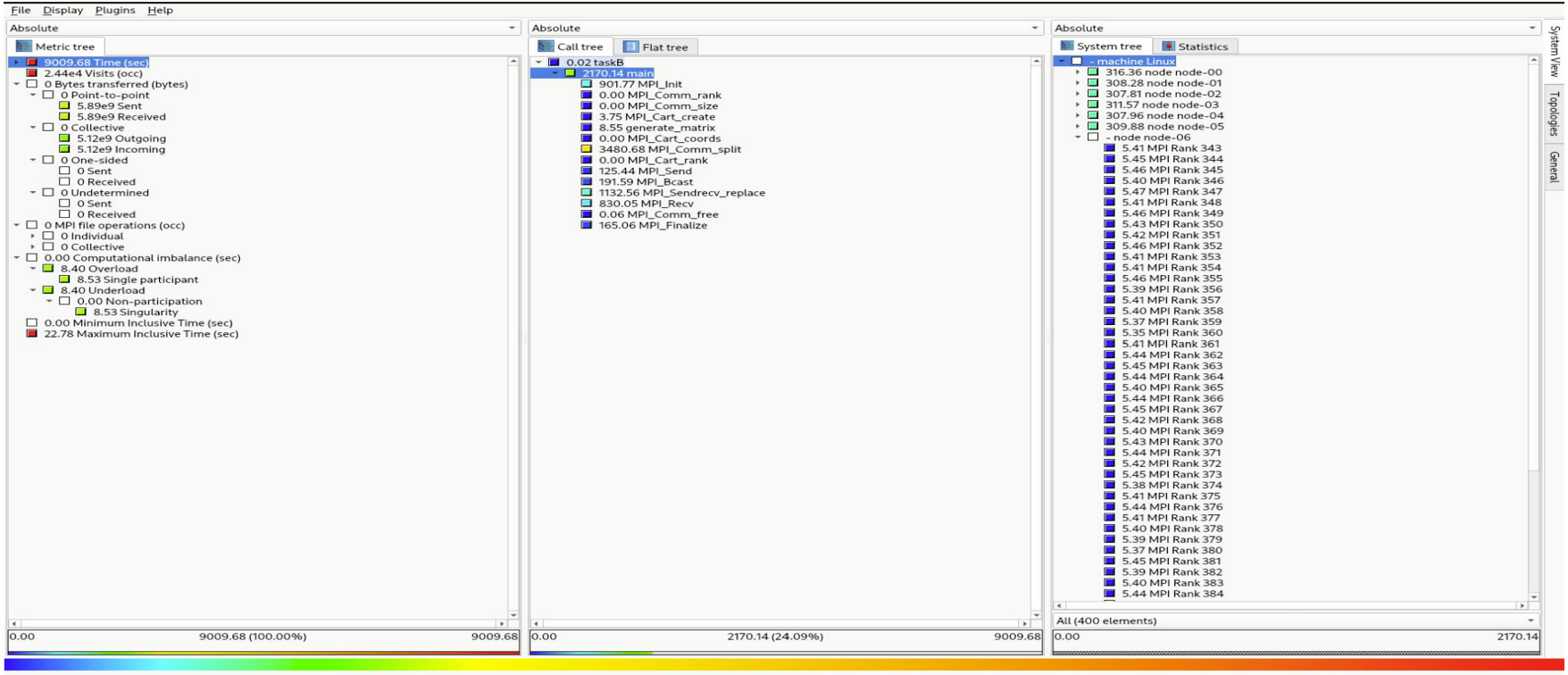


3.4.3 Which is better and Why?

Matrix size N (N * N)	No. of ranks	Fox Time	Row wise
8000 (400)	400	16.58215	44.768384
7999 (361)	361	16.943193	56.552937
7995 (225)	225	20.848334	78.333554
8000 (100)	100	33.897512	142.177188
6000 (400)	400	9.583721	21.200251
6004 (361)	361	9.179071	22.457988
6000 (225)	225	10.23975	36.111402
6000 (100)	100	15.032589	60.94756
6000 (64)	64	25.000698	91.340023
4000 (400)	400	4.131044	18.4258
4009 (361)	361	4.146263	14.747109
4005 (225)	225	4.220373	42.612454
4000 (100)	100	5.491957	20.188319
4000 (64)	64	7.074584	29.369019
4000 (25)	25	14.30934	72.17804
2000 (400)	400	1.816385	3.274246
1995 (361)	361	1.643487	3.153115
1995 (225)	225	1.359905	4.321926
2000 (100)	100	1.242776	3.461536
2000 (64)	64	1.200642	4.155797
2000 (25)	25	2.008233	9.378607
800 (400)	400	1.161203	1.085054
798 (361)	361	1.030522	0.998942
795 (225)	225	0.758797	0.711037
800 (100)	100	0.402773	0.384559
800 (64)	64	0.295318	0.299248
800 (25)	25	0.253592	0.289613

- For large matrices, the **Fox algorithm** will generally be more efficient, offering better scalability and performance, especially as the number of processes grows.
- **Row-wise multiplication** is simpler but has a limited scalability and speedup compared to Fox, making it more suited for smaller problems or environments with limited processing power.

3.5 Visualization with Cube- Fox Algorithm



4. Cannon Algorithm

Cannon’s algorithm is a parallel method for matrix multiplication designed to efficiently distribute work across multiple processes. It works on a **square process grid**, meaning the number of processes must be a perfect square. The matrices are divided into smaller blocks, which are initially aligned using a shifting mechanism. Each process multiplies its assigned submatrices locally and then communicates with neighboring processes using a **circular shift** to exchange data. This structured communication reduces unnecessary data movement and ensures better load balancing.

4.1 Implementation of Task A - Cannon Algorithm

In **Task A**, I implemented the basic version of Cannon's algorithm with the focus on **handling individual elements**. The main drawbacks of this approach was:

```
// Distributing initial matrices
if (rank == 0) {
    for (int i = 0; i < sq_size; i++) {
        for (int j = 0; j < sq_size; j++) {
            MPI_Send(&A[IDX(i*block_size, j*block_size, N)], 1, block_type,
            MPI_Send(&B[IDX(i*block_size, j*block_size, N)], 1, block_type,
        }
    }
}
MPI_Recv(local_A, block_size * block_size, MPI_INT, 0, 0, cart_comm, MPI_ST
MPI_Recv(local_B, block_size * block_size, MPI_INT, 0, 1, cart_comm, MPI_ST
```

- **Data distribution was not fully optimized.** Each process received individual matrix elements instead of blocks, leading to inefficient communication.
- **Processes had to wait longer for data.** This increased synchronization overhead, making the computation slower.
- **Limited scalability.** The method struggled with larger matrices due to excessive communication.

4.2 Improvements in Task B Over Task A

For **Task B**, I redesigned the approach to work with **submatrices** instead of **individual elements**. The main advantage of this approach was:

- This **distributes large matrix blocks** instead of sending one value at a time, reducing the number of communications.
- **Distributing the workload more evenly** across all processes to prevent any single rank from becoming a bottleneck.
- **Reducing unnecessary communication delays** by aligning the matrices properly before multiplication starts.

These changes **significantly improved performance and scalability**. The execution time dropped because each process handled a larger chunk of data at once, reducing the number of messages exchanged.

4.3 Implementation of Task B Cannon Algorithm

A **Cartesian topology** is a way to organize MPI processes in a structured, grid-like manner. It helps manage communication between processes efficiently by **defining ranks in a 2D (or higher-dimensional) grid** rather than a simple linear arrangement.

In **Cannon's algorithm**, matrix multiplication is performed **on a 2D processor grid**, so using a **Cartesian topology** is the most natural fit. Instead of treating MPI processes as a simple list (ranks 0, 1, 2, ...), we structure them into **rows and columns** to match the submatrices' layout.

Cannon's algorithm involves:

1. **Dividing the matrix** into blocks and assigning them to processes.
2. **Shifting matrix blocks** in a structured way (A left, B up).
3. **Communicating with neighbors** efficiently.

4.3 Implementation of Task B Cannon Algorithm (Cont...)

A **Cartesian topology** allows us to:

1. **Assign each process a unique 2D coordinate** instead of a simple rank (e.g., rank 5 → (row=1, col=1)).
2. **Easily determine neighbors** using MPI_Cart_shift, so we don't have to manually compute which process to send/receive data from.
3. **Enable wraparound shifting** (circular communication), ensuring that every process gets all the necessary matrix blocks.

Using **MPI Cartesian topology** makes Cannon's algorithm:

- **More efficient** – shifting and communication are easier.
- **More scalable** – it works for larger matrices without manual adjustments.
- **Less error-prone** – built-in MPI functions handle neighbors automatically.

4.3 Implementation of Task B Cannon Algorithm (Cont...)

Alternative to Cartesian Topology?

Instead of MPI_Cart_create, we could:

- **Use Manual Indexing:**
  Compute neighbors manually, but **error-prone**.
- **Use MPI Graph Topology:**
  Allows custom communication patterns, but **not ideal for structured shifts**.

Final Summary:

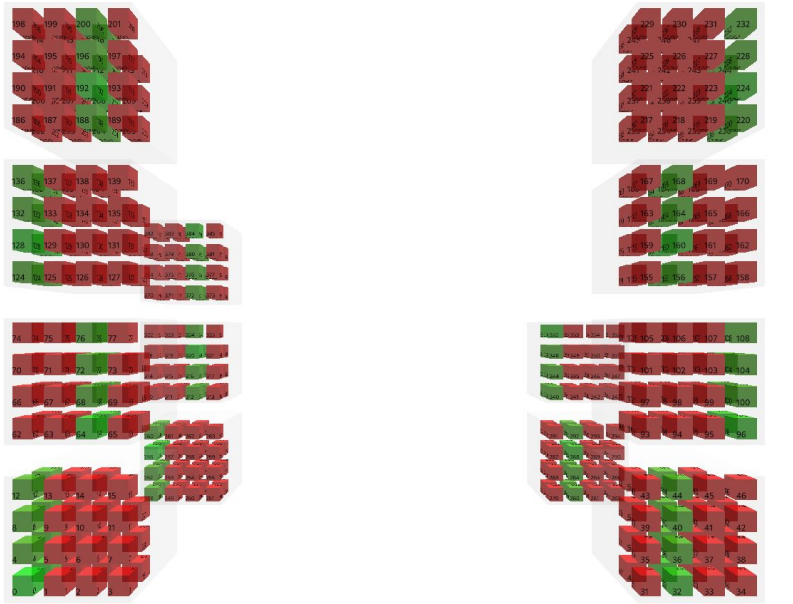
- **Cartesian topology** efficiently organizes **processes into a grid**.
- **Shifting A left and B up** ensures every process gets required blocks.
- **Communication stays localized**, reducing overhead.
- **MPI_Cart_shift automates neighbor identification**, avoiding manual rank calculations.

4.3 Implementation of Task B Cannon Algorithm (Cont...)

```
// Scatter A and B
MPI_Scatterv(A, sendcounts, displs, blocktype, localA, sub_n * sub_n, MPI_INT, 0, comm);
MPI_Scatterv(B, sendcounts, displs, blocktype, localB, sub_n * sub_n, MPI_INT, 0, comm);
```

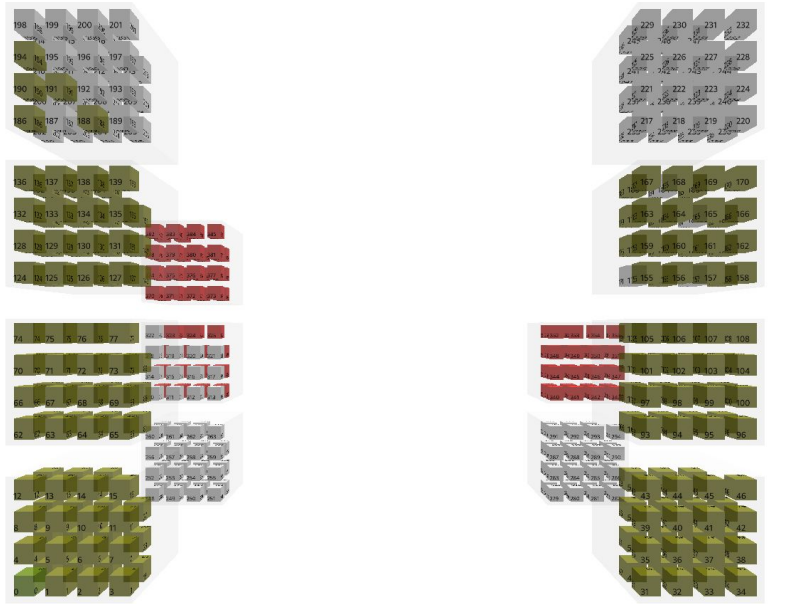
- This distributes large matrix blocks instead of sending one value at a time, reducing the number of communications.
- MPI_Scatterv allows all processes to receive data **simultaneously**.
- **Instead of rank 0 doing all the work**, MPI_Scatterv ensures even distribution.
- We could use MPI_Bcast for matrix B but **MPI_Scatterv** is better for non-contiguous data.
- **Task B** improved performance by using MPI_Scatterv, better communication, and more efficient data handling.

4.3.1 Row Wise Algorithm Execution (8000*8000 with 400 ranks)



MPI_Bcast (knomial),

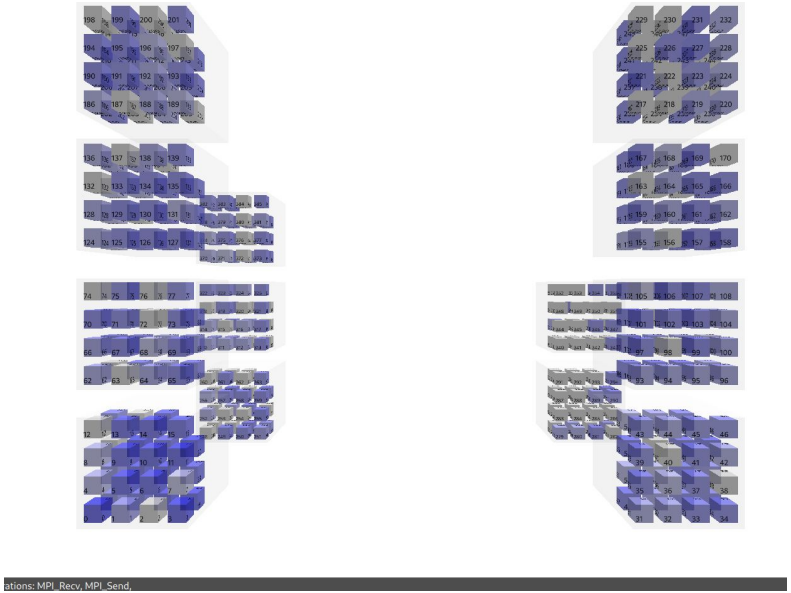
1. MPI_Bcast()



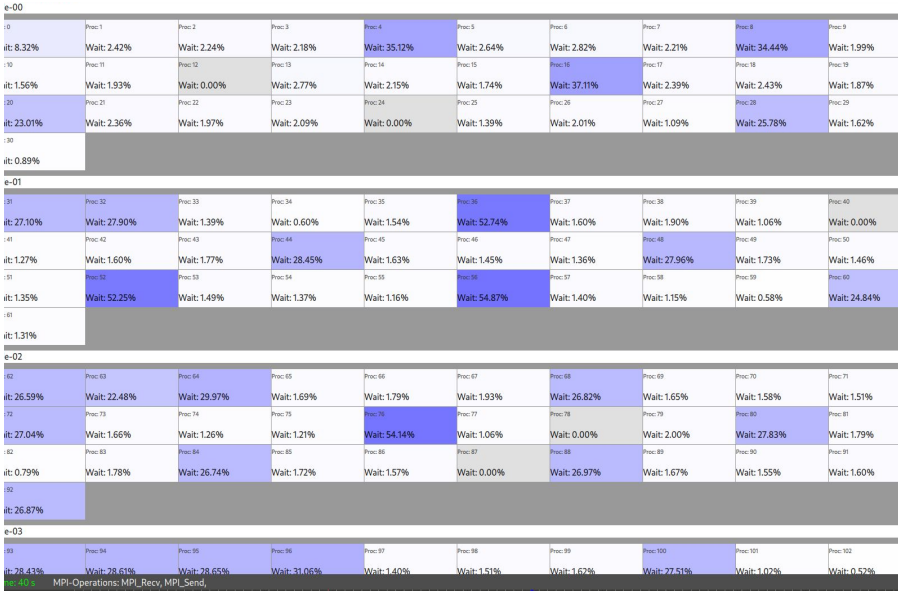
MPI_Recv, MPI_Send,

2. MPI_Send(), MPI_Recv()

4.3.1 Row Wise Algorithm Execution (8000*8000 with 400 ranks)

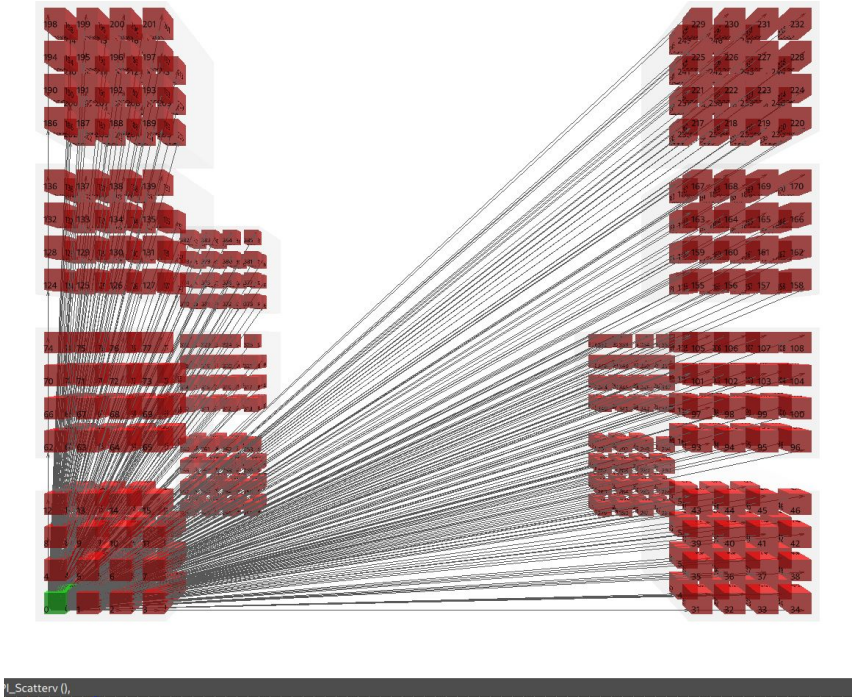


3. MPI_Send(), MPI_Recv()

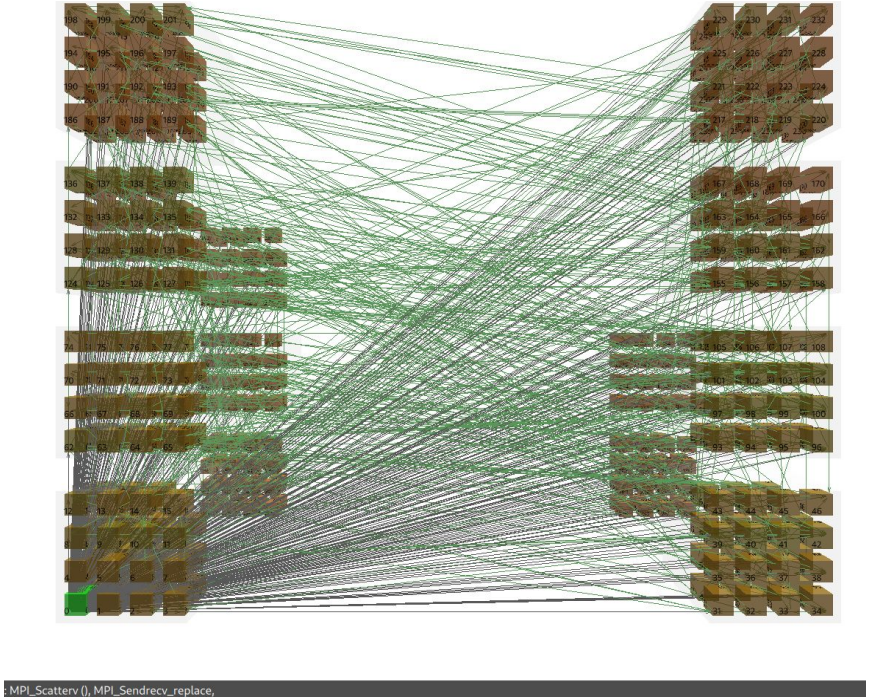


2. MPI_Send(), MPI_Recv()

4.3.2 Cannon Algorithm Execution (8000*8000 with 400 ranks)

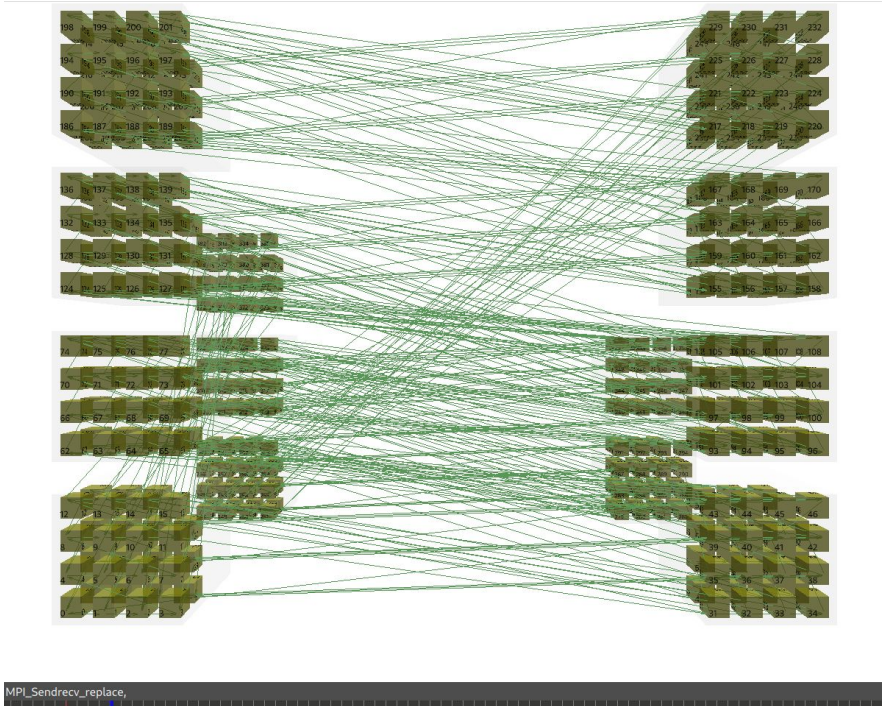


1. MPI_Scatterv()



2. MPI_Scatterv(), MPI_Sendrecv_replace()

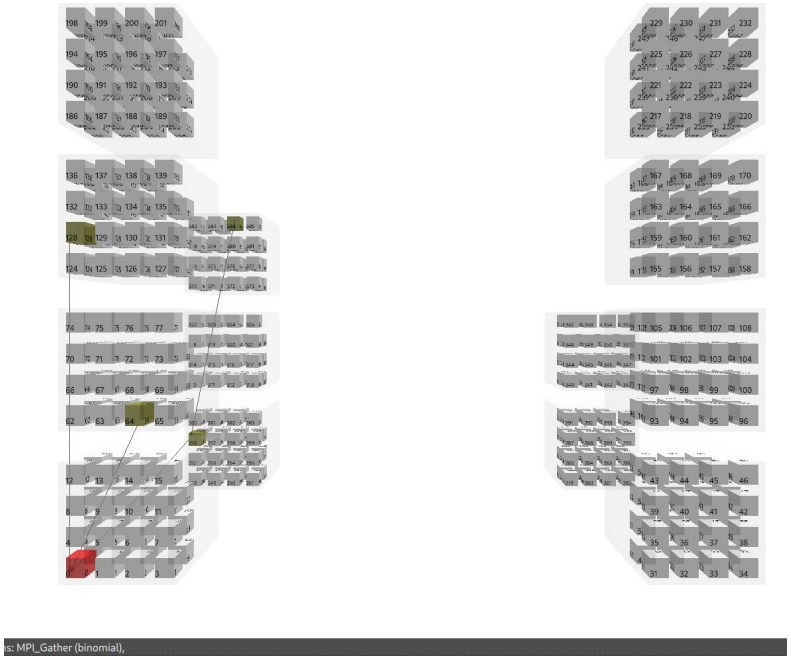
4.3.2 Cannon Algorithm Execution (8000*8000 with 400 ranks)



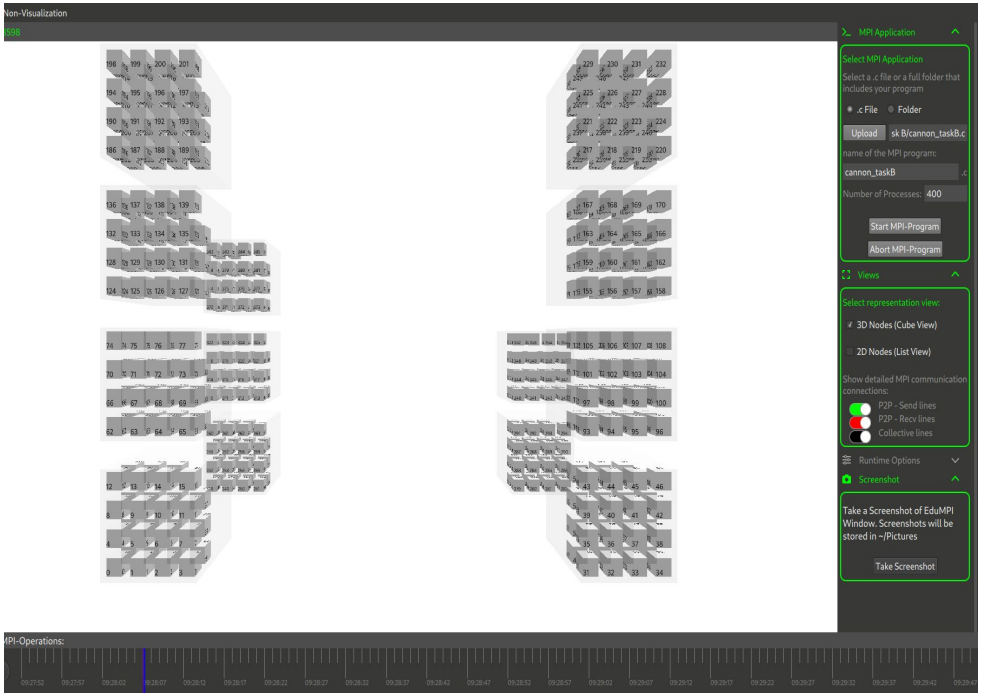
3. MPI_Sendrecv_replace()

4. MPI_Sendrecv_replace(), MPI_Gather()

4.3.2 Cannon Algorithm Execution (8000*8000 with 400 ranks)

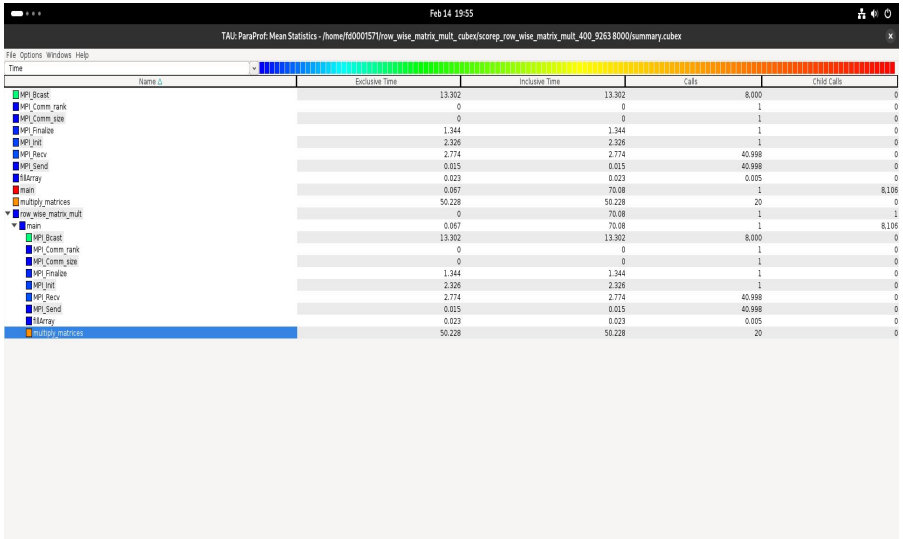


5. MPI_Gather()

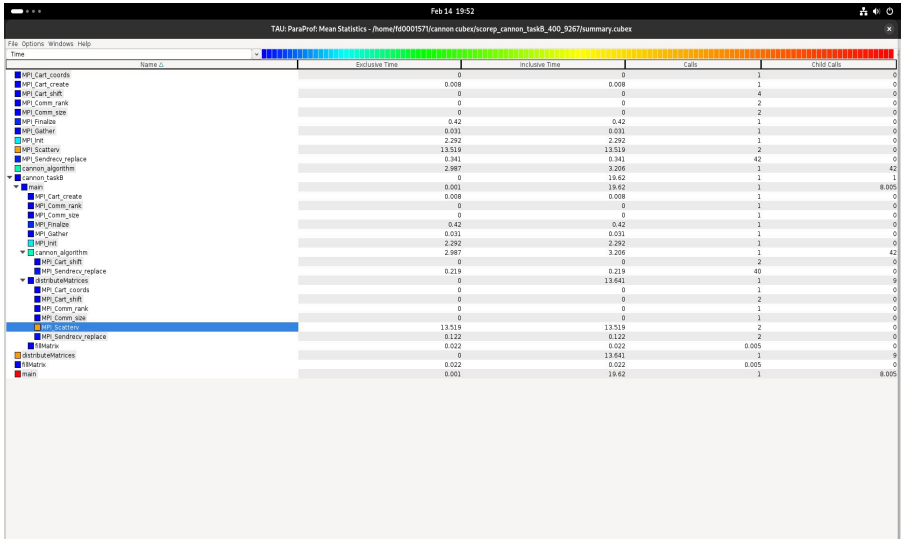


6. No MPI Operations.

4.3.3 Execution comparison using Tau

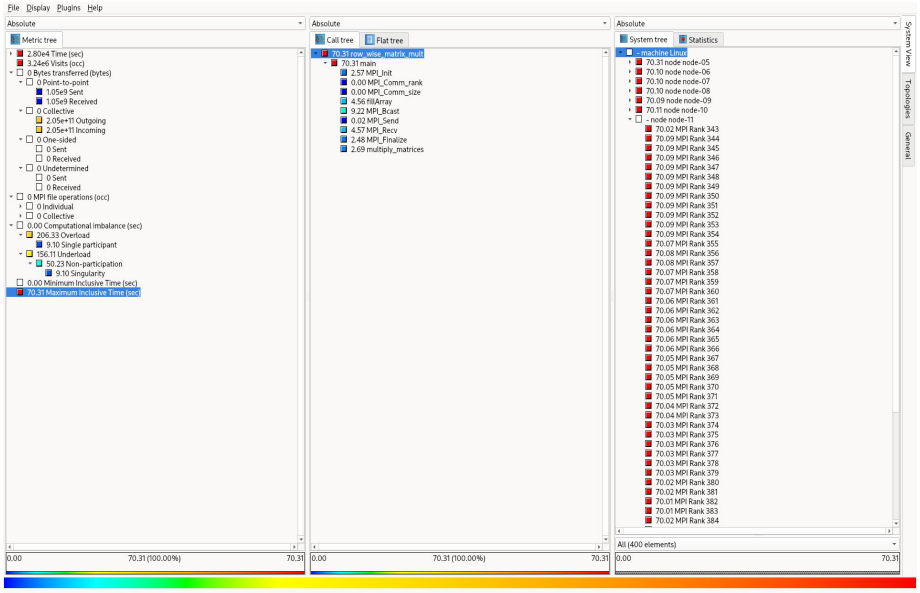


Row Wise Algorithm

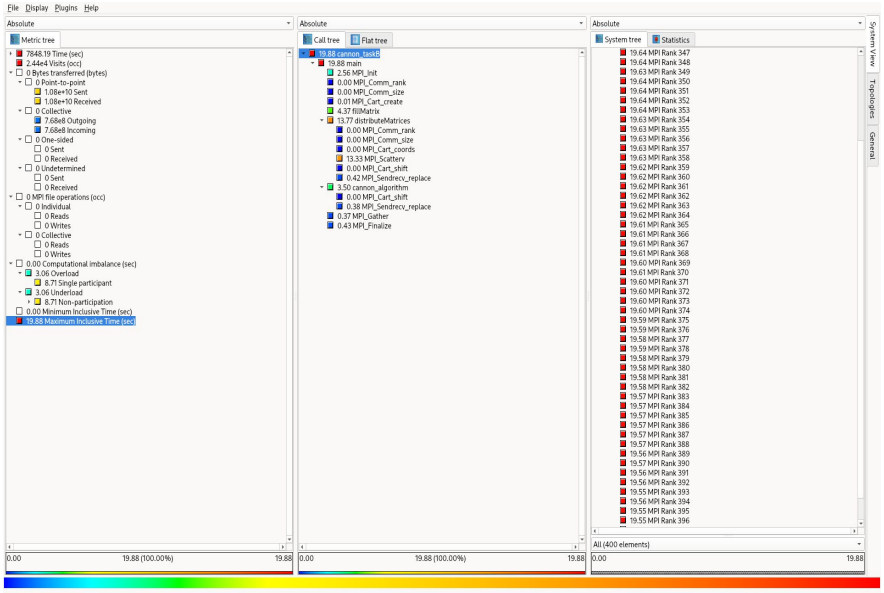


Cannon Algorithm

4.3.3 Execution comparison using Cube



Row Wise Algorithm



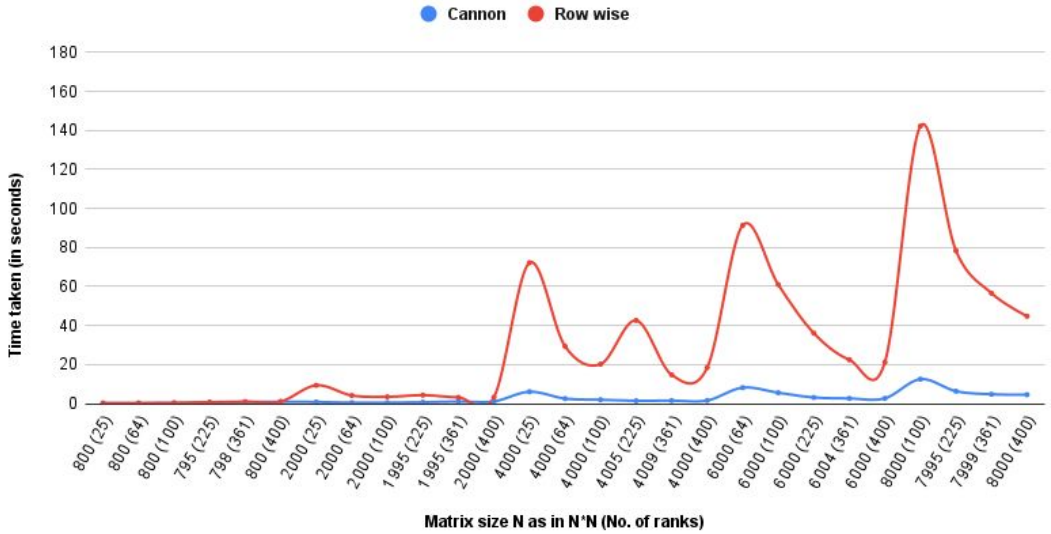
Cannon Algorithm

4.4.1 Performance comparison between Cannon & Row Wise Algorithm

Matrix size n (n *n)	No. of ranks	Cannon (time in seconds)	Row Wise (time in seconds)
8000	400	4.556065	44.768384
7999	361	4.780669	56.552937
7995	225	6.346554	78.333554
8000	100	12.519516	142.177188
6000	400	2.667808	21.200251
6004	361	2.6597	22.457988
6000	225	3.127458	36.111402
6000	100	5.565125	60.94756
6000	64	8.178098	91.340023
4000	400	1.571628	18.4258
4009	361	1.536143	14.747109
4005	225	1.427675	42.612454
4000	100	1.965064	20.188319
4000	64	2.548688	29.369019
4000	25	6.082837	72.17804
2000	400	1.097408	3.274246
1995	361	1.034949	3.153115
1995	225	0.699891	4.321926
2000	100	0.484458	3.461536
2000	64	0.471185	4.155797
2000	25	0.829536	9.378607
800	400	0.951239	1.085054
798	361	0.797747	0.998942
795	225	0.574432	0.711037
800	100	0.252218	0.384559
800	64	0.146558	0.299248
800	25	0.098584	0.289613

The maximum time taken for cannon algorithm is **12.51 seconds** overall whereas for the row wise it is 142.17 seconds both for the same matrix size 8000*8000 with number of ranks as 100.

4.4.2 Performance comparison between Cannon & Row Wise Algorithm



Cannon's algorithm in **Task B is** clearly **more efficient** because it minimizes communication overhead and evenly distributes work. The row-wise approach, while simple, becomes inefficient as matrix sizes grow.

4.5 Comparison: Task B vs. Row-Wise Algorithm

Feature	Task B (Cannon's Algorithm)	Row-Wise Algorithm
Work Distribution	Each process handles a submatrix	Each process handles a few rows
Communication	Efficient, structured shifting	High overhead due to broadcasting
Scalability	Scales well with increasing ranks	Struggles with large matrices
Load Balancing	Even workload for all processes	Some processes may finish early and wait
Performance	Faster for large matrices	Slower as communication increases

4.6 Final Thoughts - Cannon Algorithm

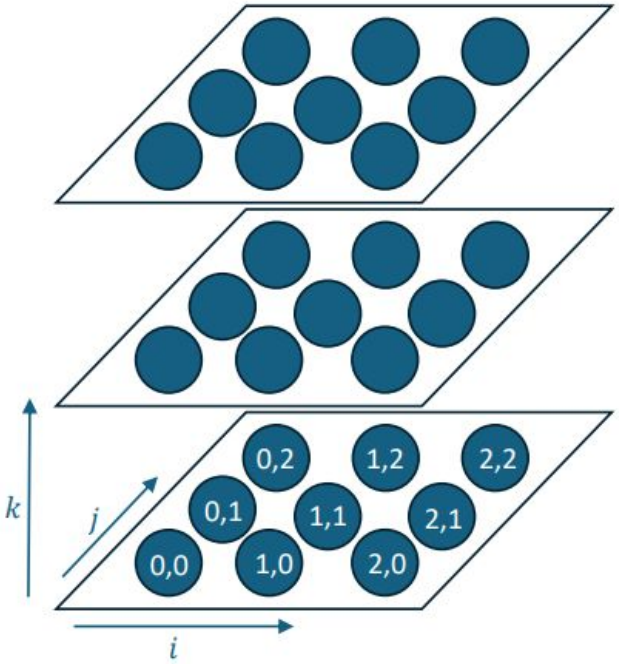
Cannon Algorithm's Task B successfully improved on the original Cannon's implementation by introducing **block-based communication** instead of row-wise transfers. This made the algorithm **faster and more scalable**, especially for large matrices. Compared to the row-wise method, it achieved better **load balancing and communication efficiency**, making it the **preferred choice for high-performance parallel matrix multiplication**.

5. DNS

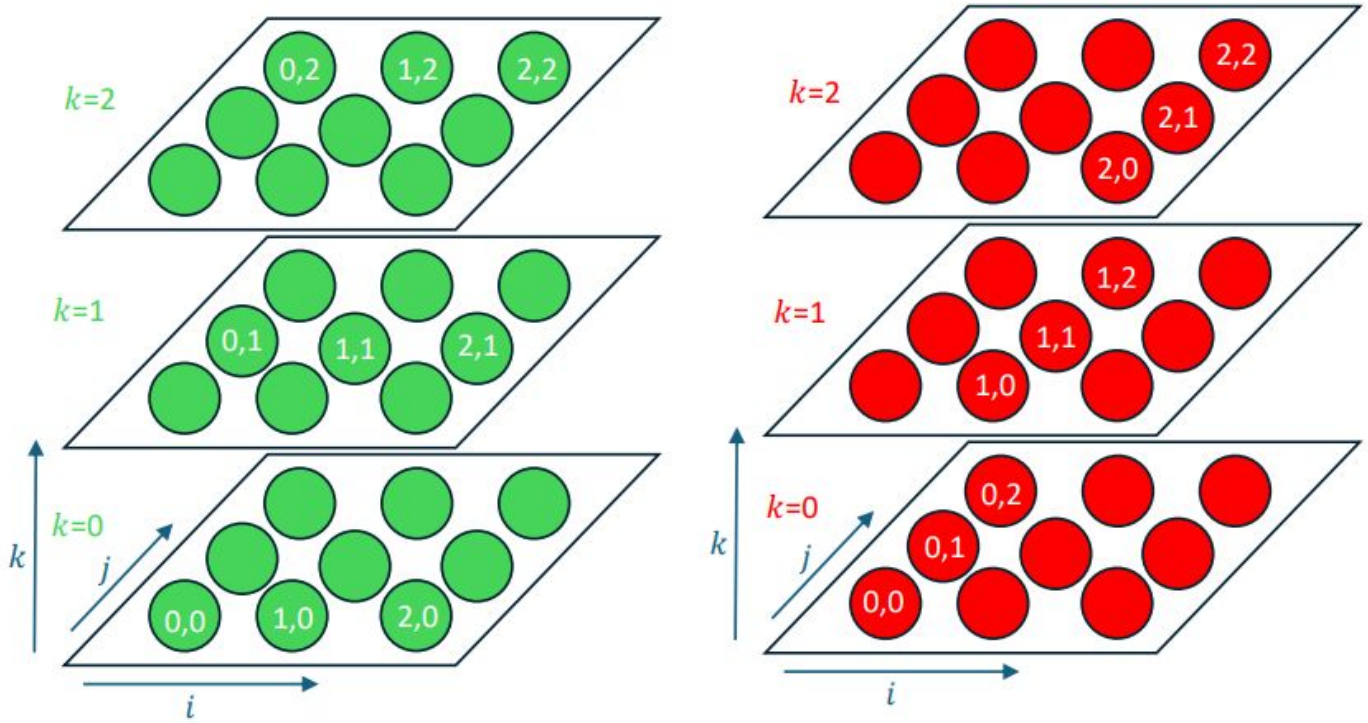
DNS(Deckel, Nassimi, Sahni) Algorithm

So number of processors (p) must be a perfect cube and the matrix size must be divisible by the root of p. The Algorithm uses this cube to distribute the matrix array in a certain manner and then do multiplication.

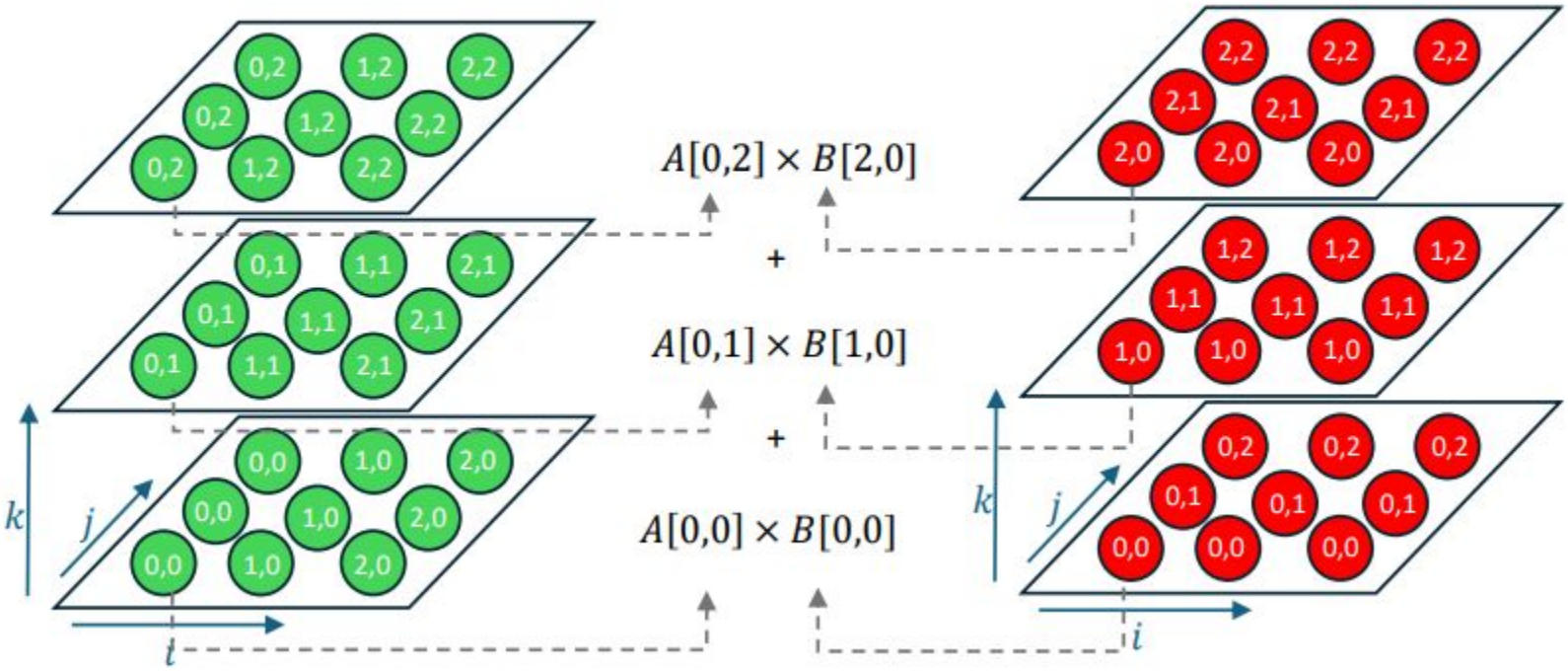
DNS Algorithm: How it Works?



Movement of the elements in the Matrices



Broadcasting the values along j and i axis

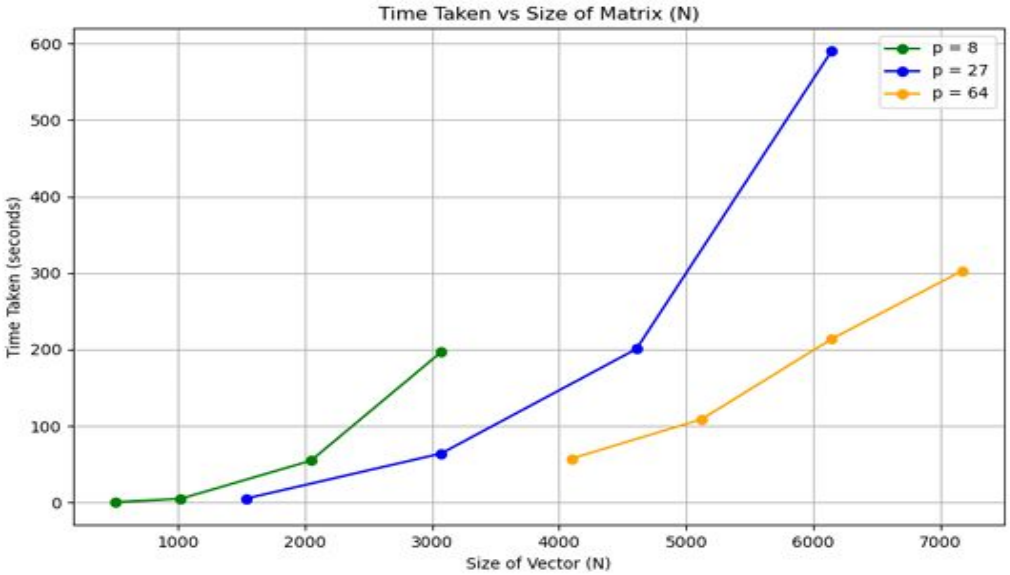


5.1 DNS(Dekel, Nassimi, Sahni) Algorithm Task A

The distribution of matrices is achieved using MPI_Bcast, a collective communication operation. Once Rank 0 generates the matrices, it broadcasts them to all other processes, ensuring each process receives a copy of the necessary matrix elements. This is done using the following MPI function:

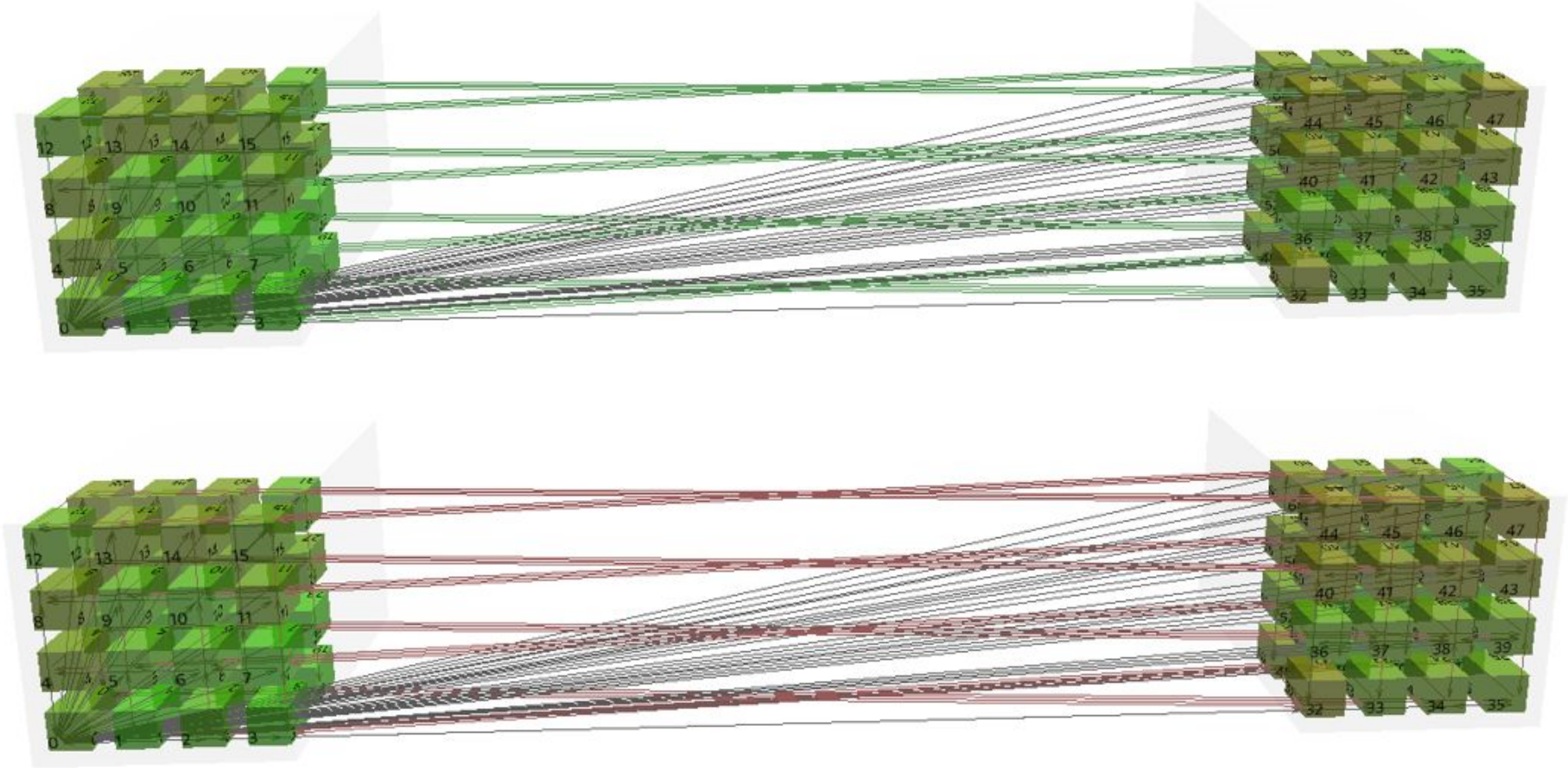
```
// Broadcast the matrices A and B to all processes
MPI_Bcast(A, n * n, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(B, n * n, MPI_INT, 0, MPI_COMM_WORLD);
```

5.2 Time taken for the DNS Algorithm For different Matrix Sizes with Different Processors



5.3 Analysis of the Graph - DNS

- The above graph shows the difference in runtimes of DNS algorithm with different combinations of matrix size and number of processes.
- As the matrix size increases the submatrix size for each process to compute increases so the exponential increase in run time is expected.
- And as the number or processes increases the number of elements per process decreases hence the yellow line of 64 processes is below the blue line.



Send and Recv Lines for Matrix size 6000 and 64 processors. Tool Used EduMPI

5.4 Send-Recv Communications used:

5.5 DNS(Dekel, Nassimi, Sahni) Algorithm Task B

For Task B, I worked on optimizing the DNS Algorithm for larger matrices by improving communication efficiency using MPI. The main refactoring involved the use of both blocking and non-blocking MPI operations to handle data movement effectively across processes.

MPI Scatter and Gather were used to distribute and collect sub-matrices before and after calculations. Scatter was used to send sub-matrix data to different processes, ensuring each process got its respective chunk of matrices A and B:

5.6 Send-Recv used in the Final Code:

For shifting sub-matrices efficiently, I used MPI Isend and MPI Irecv, which are non-blocking

communication methods. Sub-matrix data was received into nextA and nextB from the right

and down using Irecv, while Isend sent data from localA and localB up and left:

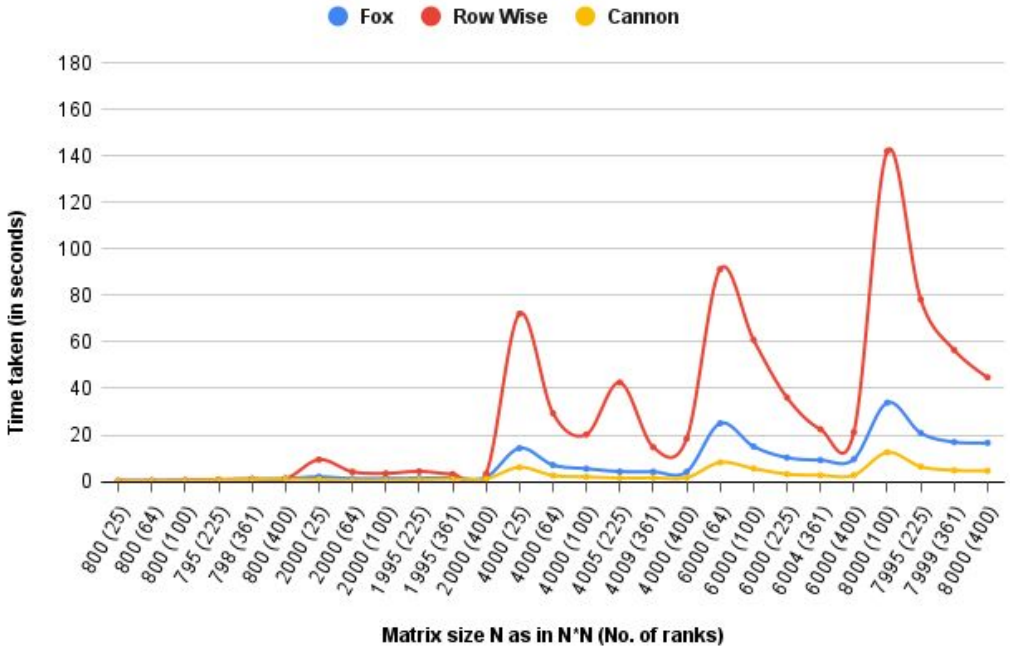
```
MPI_Irecv(nextA, block_elems, MPI_DOUBLE, right_rank, 0, cart_comm, &reqs[0]);
MPI_Irecv(nextB, block_elems, MPI_DOUBLE, down_rank, 1, cart_comm, &reqs[1]);
MPI_Isend(localA, block_elems, MPI_DOUBLE, left_rank, 0, cart_comm, &reqs[2]);
MPI_Isend(localB, block_elems, MPI_DOUBLE, up_rank, 1, cart_comm, &reqs[3]);
```

6. Performance Analysis

A	B	C	D	E	F	G	H	I
Matrix size n (n *n)	**No. of ranks**	**Fox (time in seconds)**	**Cannon (time in seconds)**	**Row Wise (time in seconds)**		**Matrix Size n (n*n)**	**No. of ranks**	**DNS(time)**
8000	400	16.58215	4.556065	44.768384		8000	125	407.5698
7999	361	16.943193	4.780669	56.552937		7992	216	321.5632
7995	225	20.848334	6.346554	78.333554		7987	343	254.4589
8000	100	33.897512	12.519516	142.177188		6000	125	168.2569
6000	400	9.583721	2.667808	21.200251		6000	64	220.2356
6004	361	9.179071	2.6597	22.457988		4000	125	75.6987
6000	225	10.23975	3.127458	36.111402		4000	64	110.0025
6000	100	15.032589	5.565125	60.94756		2000	125	8.2417
6000	64	25.000698	8.178098	91.340023		2000	64	10.756
4000	400	4.131044	1.571628	18.4258		801	27	0.9768
4009	361	4.146263	1.536143	14.747109		800	125	0.8606
4005	225	4.220373	1.427675	42.612454		800	64	0.5154
4000	100	5.491957	1.965064	20.188319		792	216	1.4144
4000	64	7.074584	2.548688	29.369019				
4000	25	14.30934	6.082837	72.17804				
2000	400	1.816385	1.097408	3.274246				
1995	361	1.643487	1.034949	3.153115				
1995	225	1.359905	0.699891	4.321926				
2000	100	1.242776	0.484458	3.461536				
2000	64	1.200642	0.471185	4.155797				
2000	25	2.008233	0.829536	9.378607				
800	400	1.161203	0.951239	1.085054				
798	361	1.030522	0.797747	0.998942				
795	225	0.758797	0.574432	0.711037				
800	100	0.402773	0.252218	0.384559				
800	64	0.295318	0.146558	0.299248				
800	25	0.253592	0.098584	0.289613				

Table with execution time of different algorithms with different ranks and matrix size compared with Row Wise here.

6.1 Performance Analysis



From the graph, we could see that the **Cannon algorithm** works better offering a balanced trade-off between performance and complexity.

6.2 Comparing Matrix Multiplication Algorithms

- **Cannon Algorithm** is the most efficient for balanced grids, offering low communication overhead and good scalability.
- **Fox Algorithm** is effective but slightly slower due to additional communication.
- **DNS Algorithm** is computationally expensive, requiring high memory and large message exchanges.
- **Row-wise Algorithm** performs poorly due to excessive communication between ranks.
- Overall, **Cannon** and **Fox** scale better than DNS and Row-wise, making them more practical for large problems.

6.3 Choosing the Best Algorithm Based on Conditions

- Use Cannon for general-purpose large-scale matrix multiplication.
- Use Fox when processor grids are limited.
- Avoid DNS unless memory and computation resources are abundant.
- Avoid Row-wise for large-scale parallelization due to high communication overhead.
- Optimize based on specific hardware (GPU, cloud, or distributed systems).

7. Conclusion

Parallel matrix multiplication is essential for handling large-scale computations efficiently. While the row-wise method is simple, it struggles with communication overhead. Fox, Cannon, and DNS provide better alternatives, with Cannon offering a balanced trade-off between performance and complexity. The results suggest that **Cannon is the best choice for general-purpose parallel matrix multiplication**, while **Fox is effective** but slightly slower and **DNS is optimal** for very large matrices on powerful clusters. Future work can focus on hybrid approaches that combine the strengths of these different methods.

8. References

- Matrix Multiplication - Fox Algorithm
- Derived data types Reference MPI - how it can be used for fox.
- MPI Communication - stack overflow - for verifying broadcasting necessity.
- Tutorial on Cannon's algorithm for understanding, here.
- Cannon’s Algorithm for Matrix Multiplication wikipedia article.
- Used lecture slides for understanding different communication mechanisms.
- Used code files uploaded in exercises for reference to structure the program.

Thank you!

Tschüss "👋