CS677-Parallel Distributed 3D Volume Rendering: Group 10

1 Introduction

This report describes the implementation of Parallel Distributed Volume Rendering using MPI (Message Passing Interface) with 3D decomposition. The program performs volume rendering by decomposing a 3D scalar dataset across multiple processes along all three dimensions (X, Y, Z) to execute the ray-casting algorithm concurrently.

2 Code Overview

The program reads a 3D scalar dataset and distributes it across ranks. Each rank processes its subdomain for ray casting along the Z direction, implementing front-to-back compositing. The results are then stitched together to form the final image output.

2.1 Key Functions

2.1.1 `Main`

- MPI Initialization: Initializes MPI, retrieves rank, size, and parses command-line arguments.
- Dataset Loading and Distribution: Rank 0 reads and distributes the dataset to all ranks using `MPI_Bcast`.
- Volume Rendering: Uses 3D domain decomposition where each process handles a unique subdomain. The rendering is managed by `volumeRendering`.

2.1.2 `readDataset`

Loads a binary 3D dataset file. The dataset is reshaped into a vector for distribution to all ranks.

2.1.3 `rayCasting`

The `rayCasting` function is responsible for performing ray traversal and compositing along the Z-axis within each subdomain assigned to a process. For each pixel in the X and Y dimensions, rays are cast through the Z direction. As each ray progresses, it accumulates color and opacity values based on interpolated data from the 3D dataset, with transfer functions applied to assign color and opacity for each intensity value.

A critical feature of the function is the communication between processes along the Z-axis. Each process, after processing its assigned Z range, sends the accumulated opacity value of each ray to its child process in the Z direction. This communication ensures that each process continues the compositing work without interruption. The process at the front of the Z direction initializes the ray accumulation, while each subsequent process builds upon the accumulated values received from its parent. This step-by-step handoff avoids idle time, as each process operates on its specific subdomain and communicates intermediate results promptly. By maintaining this direct parent-to-child communication, the function achieves parallel processing along the Z direction, ensuring continuity and consistency across the subdomains and ultimately enabling efficient parallel computation with minimized latency due to idle waiting.

2.1.4 `volumeRendering`

The `volumeRendering` function coordinates the entire rendering pipeline across all processes. It begins by decomposing the dataset into subdomains, distributing them along X, Y, and Z dimensions according to the specified process grid. The function then calls `rayCasting` for each subdomain, with each process rendering its portion of the data. Compositing along the Z-axis is managed by grouping processes with the same X and Y coordinates, allowing front-to-back compositing along the Z direction within each group. The results from each subdomain are gathered by the front-most process in the Z direction for final image assembly. Once all subdomains have been composited, the primary process gathers and saves the complete image. This function also measures compute and communication times to analyze performance across different configurations, providing insight into the scalability and efficiency of the parallel volume rendering approach.

2.1.5 `saveImageFromVector`

Saves the generated image in PNG format. This function normalizes the pixel data and converts it to an 8-bit format.

3 Performance Observations

The performance was analyzed for various test cases by measuring computation, communication, and total execution times.

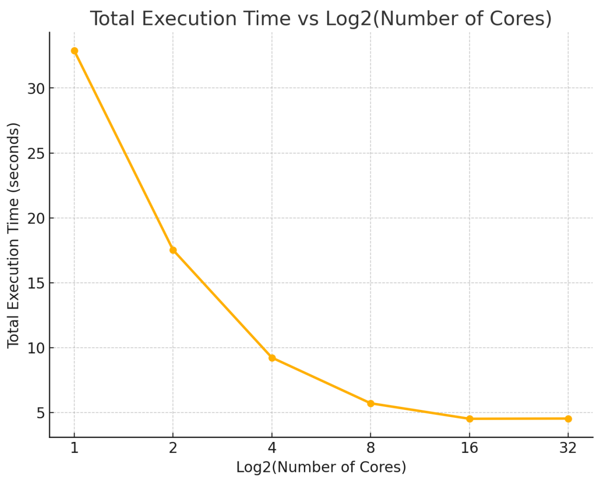

Figure 1: Runtime variation with increasing cores

We can clearly see in Figure 1. the time reduces with increase in number of cores. It saturates after core 16, and we no longer see improvement in runtime. This could be attributed to the machine maximum cores (ie. 16) and the communication time.

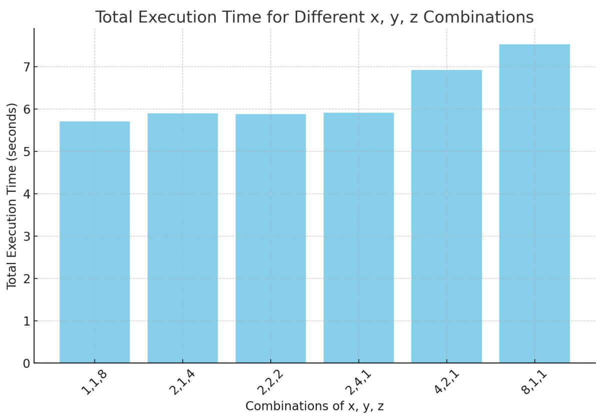

Figure 2: Runtime vs Permutations of cores

In Figure 2, we observe the effect of different combinations of processes along the x, y, and z dimensions on total execution time. The configuration with $x = 1$, $y = 1$, $z = 8$ shows the best performance, achieving the lowest execution time. However, as we increase the number of processes along the x or y dimensions, such as in combinations $x = 4, y = 2, z = 1$ and $x = 8, y = 1, z = 1$, we see a gradual increase in execution time.

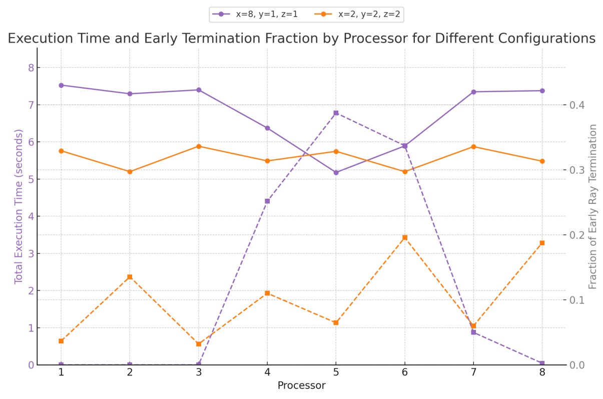

Figure 3: Visualization for Load Balancing

The figure illustrates that, across various configurations (e.g., $x = 8, y = 1, z = 1$ and $x = 2, y = 2, z = 2$), each permutation achieves efficient load balancing, as indicated by relatively stable execution times and early ray termination fractions across processors. This balance ensures that work is evenly distributed and minimizes idle time, enabling effective parallel processing across different axis allocations. Some processors have lower time taken but that can be accounted to the higher early ray termination in them.

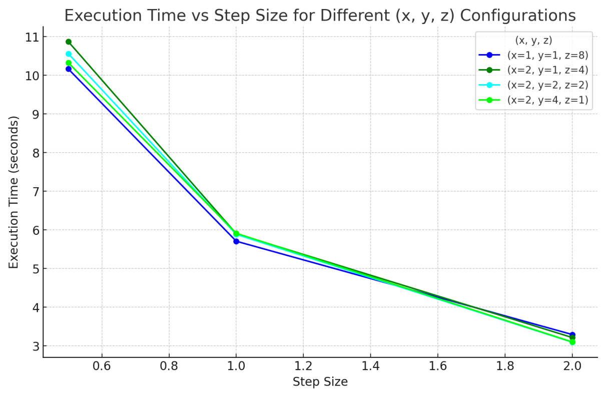

Figure 4: Runtime variation with step size of different permutations

This figure suggest that there is little to no difference between working over permutations of cores and the graph is close to linear with time getting halved at each step.

4 Test Cases and Results

The following test cases were executed with specified parameters and decompositions:

- **Test Case 1**: `mpirun -np 8 -f hostfile ./executable Isabel_1000x1000x200_float32.raw 2 2 2 0.5`
- **Test Case 2**: `mpirun -np 16 -f hostfile ./executable Isabel_1000x1000x200_float32.raw 2 2 4 0.5`
- **Test Case 3**: `mpirun -np 32 -f hostfile ./executable Isabel_1000x1000x200_float32.raw 2 2 8 0.5`

4.1 Execution Times and Early Ray Termination

Summary of Total Execution Times

Test Case	Total Execution Time (s)
1	10.5606
2	7.61328
3	7.82157

Table 1: Overall execution times for each test case

Detailed Processor-Specific Computation Times and Early Ray Termination Fractions

Processor	Computation Time (s)
1	10.5606
2	9.33531
3	9.72529
4	9.78459
5	10.533
6	9.31773
7	9.71582
8	9.81908

Table 2: Test Case 1: Computation times for each processor (2x2x2 decomposition)

Processor	Computation Time (s)
1	6.01999
2	6.72505
3	7.53515
4	7.03262
5	6.03796
6	6.74436
7	7.557
8	7.05452
9	6.07809
10	6.83469
11	7.61328
12	7.06692
13	6.04828
14	6.73308
15	7.54631
16	7.04568

Table 3: Test Case 2: Computation times for each processor (2x2x4 decomposition)

Early Ray Termination Fractions for Processors

Processor	Early Ray Termination Fraction
1	0.25932
2	0.154546
3	0.439239
4	0.129461
5	1.00401
6	1.00401
7	1.00401
8	1.00401

Table 4: Test Case 1: Early ray termination fractions (2x2x2 decomposition)

Processor	Early Ray Termination Fraction
1	0.0302368
2	0.121819
3	0.0261123
4	0.0957948
5	0.21963
6	0.153758
7	0.394095
8	0.128879
9	0.435862
10	0.189718
11	0.560223
12	0.181256
13	1.00401
14	1.00401
15	1.00401
16	1.00401

Table 5: Test Case 2: Early ray termination fractions (2x2x4 decomposition)

5 Conclusion

The results demonstrate that (apart from that of 2D and 1D decompositions)

- performance improves with increased process counts up to saturation.
- there is no significant difference of runtime over permutations of cores with respect to total cores.
- load imbalance can be handled using techniques like by communication over z processes and binary swap algorithm

. .

6 Group Information

Group Number: 10
Group Members:

- Divyansh Mittal (Roll Number: 210358, Email: dmittal21@iitk.ac.in)
- Lakshvant Balachandran (Roll Number: 210557, Email: lakshvant21@iitk.ac.in)
- Parthapratim Chatterjee (Roll Number: 210705, Email: partha21@iitk.ac.in)