CS677: TOPICS IN LARGE DATA ANALYSIS AND VISUALIZATION

Assignment 2 - Parallel Distributed Axis-aligned Volume Rendering using MPI

Report

By

Group-1

Parjanya Aditya Shukla - 241110046
Divyansh Chaurasia - 241110022
Ansh Makwe - 241110010

Email: { anshakwe24, cdivyansh24, paditya24 } @cse.iitk.ac.in

Indian Institute of Technology, Kanpur

Parallel Volume Rendering Techniques

1. Explanation of key concepts used

1.1 3D Domain Decomposition

One process handles each of the smaller subdomains (or chunks) that are created by splitting the data along the x, y, and z axes. Faster computation and parallel processing result from each process being able to handle a smaller portion of the dataset.

1.2 Binary Tree Load Balancing

Data aggregation and transmission across processes are managed by forming a binary tree-like structure for inter-process communication.

This is done by pairing the neighbouring processes in consecutive rounds; this tree-based, recursive method reduces conflict and communication costs. Also, at each iteration, only a subset of processes take part, ensuring effective data transport free from bottlenecks.

1.3 Front-to-Back Compositing

Each process calculates color and opacity for each voxel it meets, adding up these values as it moves through the volume because rays are cast along the z-axis (aligned with the screen depth). When opacity hits a threshold (near 1), more computations can be halted, saving computational time. This front-to-back strategy works well for early termination.

1.4 Transfer Functions

Transfer functions assign visual attributes depending on voxel values, controlling the rendering process by mapping scalar values in the dataset to colors and opacities.

1.5 Load Balancing

By distributing the work evenly, load balancing helps to prevent hotspots—areas where specific processes handle an excessive amount of work—and lowers communication conflict. This is crucial in large parallel systems to avoid delays and enable each process to execute an equal amount of work and synchronize well.

Along with binary tree based load balancing, we have also implemented a Binary-Swap image composting algorithm. In our previous implementation for image compositing and load balancing, at each iteration, half of the processes drops out of communication, which reduces the communication bottleneck, but only fewer processes are now active for image composting.

Due to this reason we have also implemented a Binary-Swap image compositing algorithm. Since, in binary-swap there is communication between adjacent pairs or groups holding the same image group, all the processes are active throughout the compositing, being responsible for generating their respective graphic primitive.

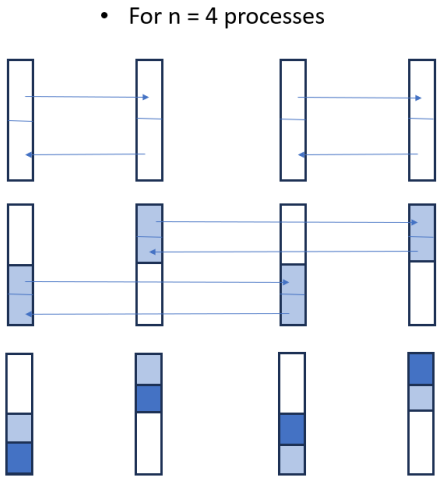

Figure 1: Binary Swap

The only issue is that using binary swap will increase the communication bottleneck but improves the load balancing.

We have also provided the computation results using both algorithms further down the report.

1.6 Tree-based Gather

Effective aggregation of partial results across processes is made possible by the use of binary tree technique and tree-based gather. By dividing the data collection process into hierarchical steps, the data is first captured on the surface processes using the tree base gather function of our implementation.

A similar function with same functionality named tree based gather after function is then implemented to capture the data at process 0 for final image generation.

In addition to producing a final composited image, this methodology provide scalable and effective parallel volume rendering, fulfilling the assignment's requirements for computation and communication time tracking.

Similarly, in our implementation of binary-swap, after applying the binary swap algorithm along z-axis, tree based hierarchical is then used to capture data on surface processes. And then the same tree based gather after function is responsible for capturing the data at process 0 for final image generation

2. Is this fully parallel ?

2.1 Data Exchange Between Processes

Following the first data decomposition, each process is in charge of a certain subset of the data. However, the outcomes of each step must be integrated into a single final image when individual ray casting is completed. Because processes must wait for one another to finish before final compositing, this aggregation necessitates communication between them, adding a sequential element.

2.2 Tree-based Collection

Despite their efficiency, the binary swap and tree-based gather techniques create hierarchical communication. Because each process in the hierarchy must wait for the others at each stage before moving forward, this indicates that processes are not totally independent. As a result, the gather operation is not carried out simultaneously across all processes, but rather in steps.

2.3 Opacity Accumulation

The front-to-back compositing method allows for the computationally efficient early termination of each ray when opacity reaches a threshold. It does, however, result in an imbalance in load since some

processes finish their work more quickly than others. Parallel efficiency may be limited if processes with fewer calculations are idle, waiting for others to finish before synchronization takes place.

2.4 Load Imbalance

Even using the hierarchical compositing technique, early termination may result in uneven workloads for different processes, especially in areas of the volume with different data density. Wait times may be further increased if processes treating dense regions—which contain a large number of non-transparent voxels—complete later than those managing sparser regions.

2.5 Image Sequential Stitching

After collecting data from separate processes, rank 0 (also known as the root process) stitches the outcomes together to create the finished image. Because it entails retrieving input from several processes and putting it in the right sequence to produce a cohesive image, this stage is sequential. To guarantee proper composition, this is usually handled by a single operation, creating a bottleneck.

2.6 I/O Reliance on Final Output

The root process writes the resulting image to a file in the last image-saving step, adding a sequential element as well. This process can nonetheless lead to non-parallel behavior even though it is not computationally demanding, especially in contexts with limited input/output.

3. Evaluation of Results -:

3.1 For Smaller dataset -

No. of Processors	Step Size	Communication Time (s)	Computation Time (s)	Total Time (s)
8	0.5	0.151	25.46	25.611
16	0.5	1.89	15.62	17.51
32	0.5	3.10	12.53	15.63

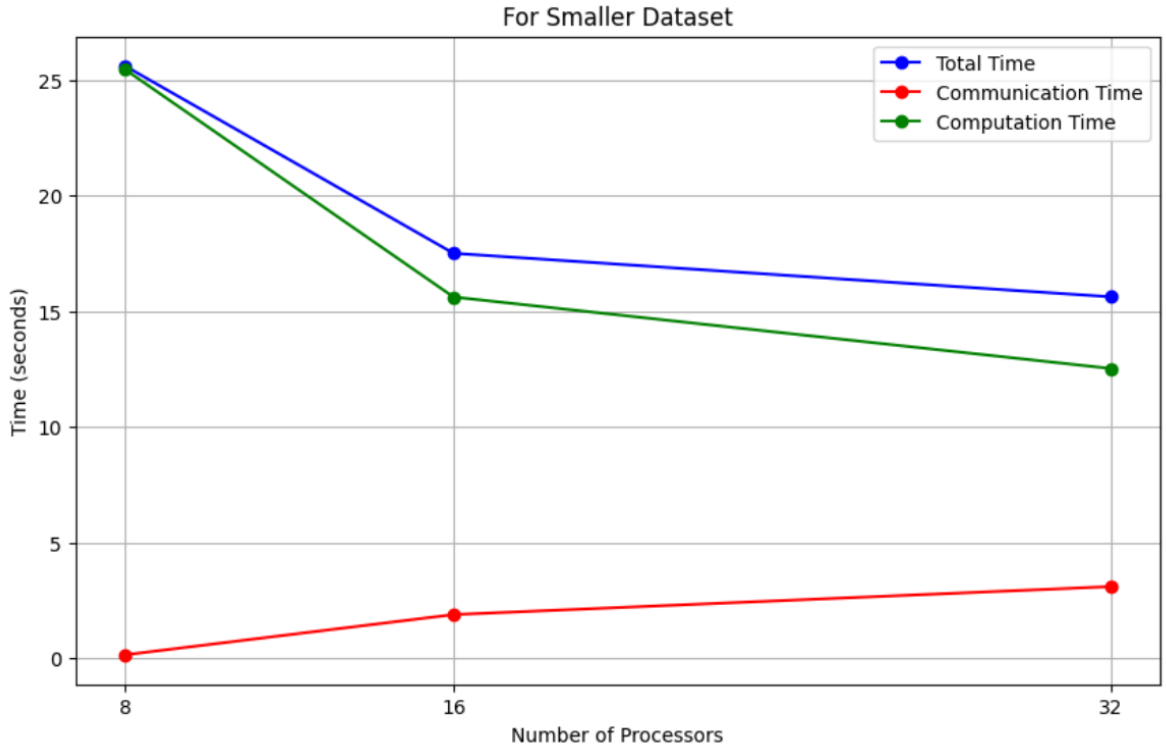

Figure 2: Time taken for different no. of processes for step size = 1.0

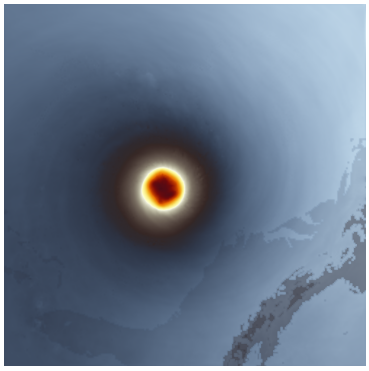

Figure 3: Image Rendered with 8 processes

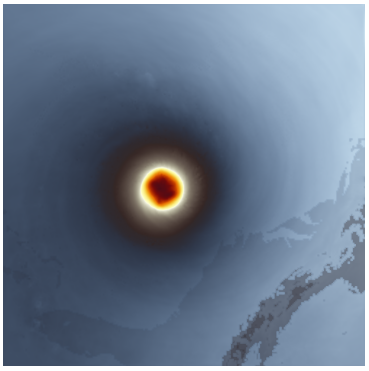

Figure 4: Image Rendered with 16 processes

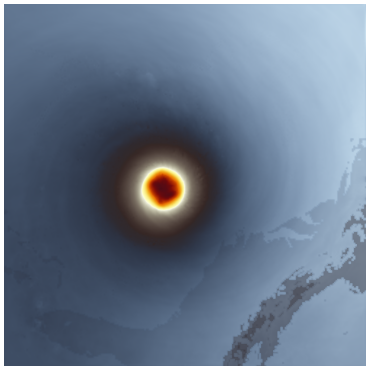

Figure 5: Image Rendered with 32 processes

3.2 For Smaller dataset - Implementing Binary Swap

No. of Processors	Step Size	Communication Time (s)	Computation Time (s)	Total Time (s)
8	0.5	0.36	21.17	21.53
16	0.5	0.79	14.77	15.56
32	0.5	3.76	12.01	15.77

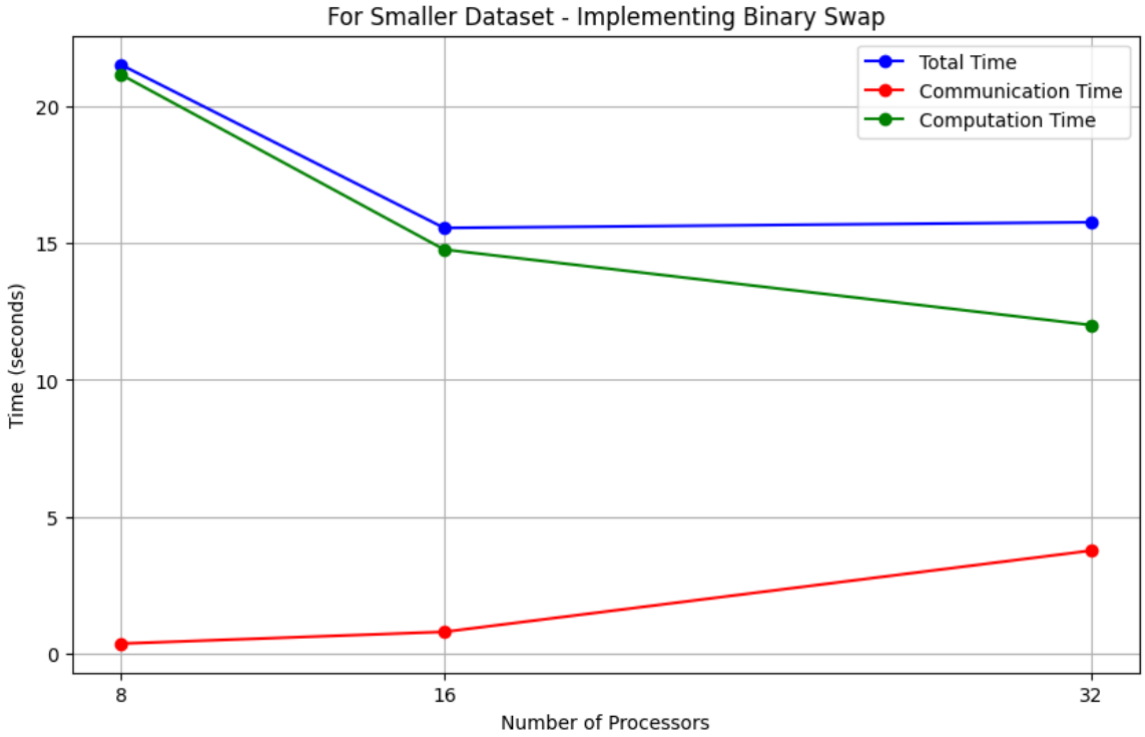

Figure 6: Time taken for different no. of processes with Binary Swap for step size = 1.0

4. For Large Dataset -

No. of Processors	Step Size	Communication Time (s)	Computation Time (s)	Total Time (s)
8	0.5	2.13	135.19	137.32
16	0.5	4.11	90.43	94.54
32	0.5	15.12	65.72	80.84

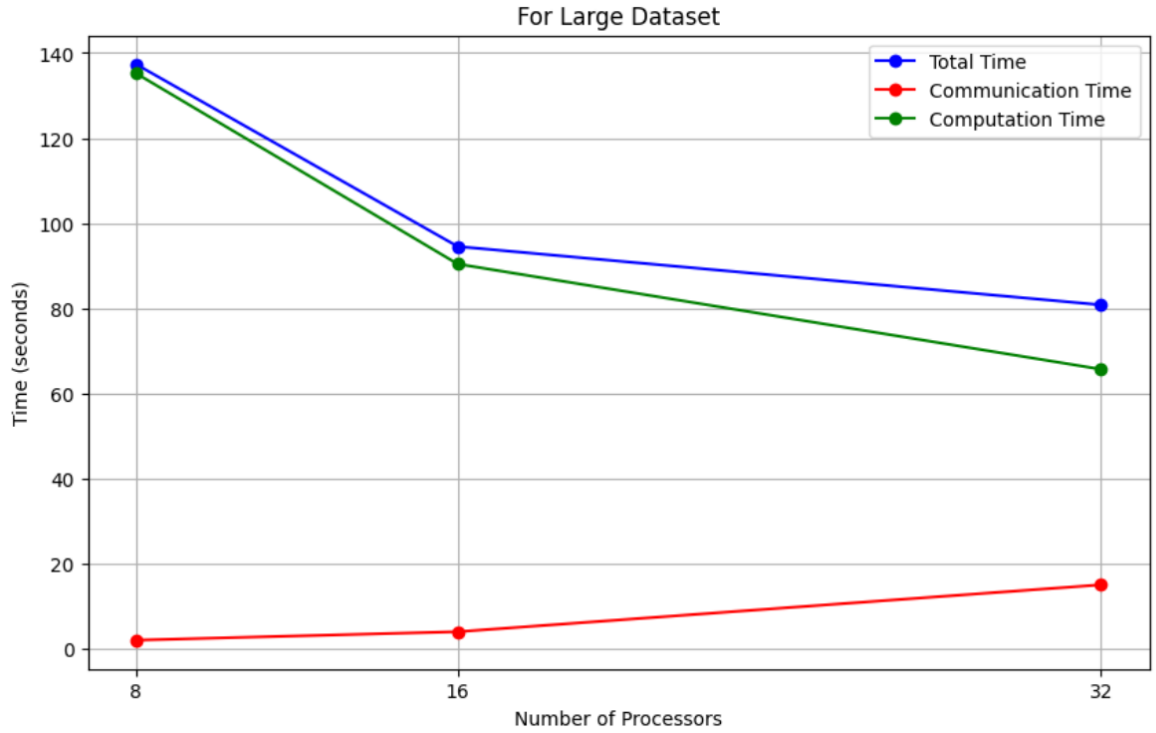

Figure 7: Time taken for different no. of processes for step size = 1.0

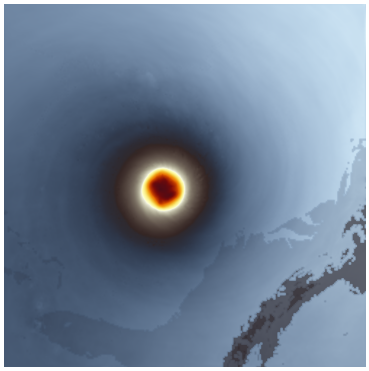

Figure 8: Image Rendered with 8 processes

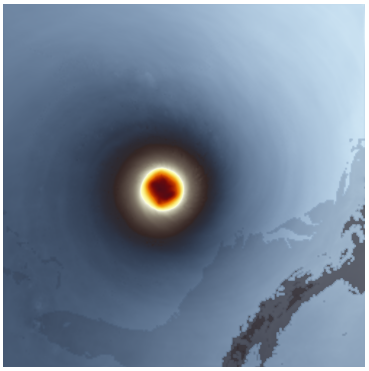

Figure 9: Image Rendered with 16 processes

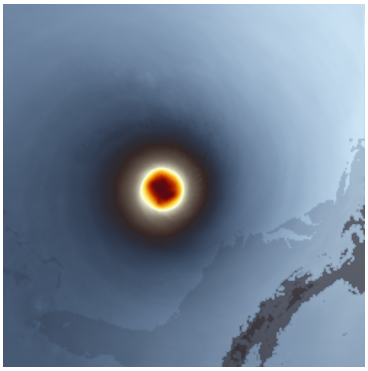

Figure 10: Image Rendered with 32 processes

5. For Large Dataset - Implementing Binary Swap

No. of Processors	Step Size	Communication Time (s)	Computation Time (s)	Total Time (s)
8	0.5	3.70	132.91	136.61
16	0.5	7.11	92.12	99.23
32	0.5	25.55	66.73	92.28

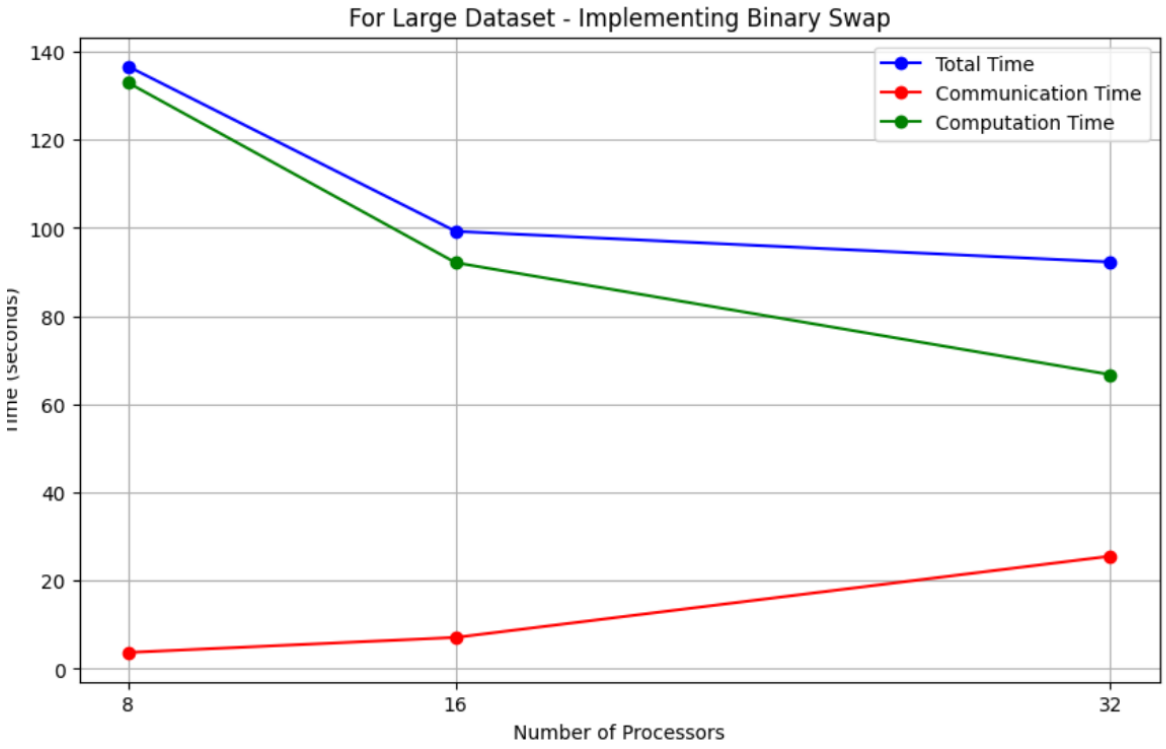

Figure 11: Time taken for different no. of processes for step size = 1.0

6. Observations

6.1 For Smaller Dataset (No Binary Swap)

- **8 Processes**: The total time is the highest among all three setups. Although communication time is low, the computation time is relatively high due to each process handling a larger portion of data.
- **16 Processes**: Increasing the number of processes reduces both computation and communication times. This setup has a balanced reduction in total time, with a significant drop in computation time.
- **32 Processes**: With further parallelism, the computation time decreases further, but communication time increases. The total time is close to the 16-process setup, showing diminishing returns in speedup.

6.2 For Smaller Dataset (Using Binary Swap)

- **8 Processes**: With binary swap, the communication time increases slightly compared to the previous non-swap configuration, but the total time is reduced due to more efficient compositing.
- **16 Processes**: The total time is minimized at this process count, indicating an effective balance between computation and communication.
- **32 Processes**: The communication time increases substantially due to the added complexity of binary swap, leading to a higher total time than with 16 processes. This suggests that for this dataset size, the performance gains from parallelism level off beyond 16 processes.

6.3 For Larger Dataset (No Binary Swap)

- **8 Processes**: High computation and communication times are observed, with the total time being the highest among all configurations, as each process has to manage a significant data portion.
- **16 Processes**: Adding more processes reduces both computation and communication times, lowering the total time significantly.
- **32 Processes**: The computation time decreases further, but the communication time rises, reflecting the increased overhead of inter-process communication for a larger number of processes. The diminishing returns on total time reduction suggest a possible threshold for efficiency.

6.4 For Larger Dataset (Using Binary Swap)

- **8 Processes**: Communication time is higher with binary swap than without, though total time remains close to the non-binary swap setup due to improved load balancing.
- **16 Processes**: The communication time grows with binary swap, but total time remains reasonable, benefiting from the balanced workload.
- **32 Processes**: Binary swap introduces a significant communication overhead here, which overshadows the benefits of reduced computation time. As a result, the total time exceeds that for 16 processes, showing that higher process counts may not be as effective for this dataset size with binary swap.

6.5 Summary

These observations reveal that binary swap improves load balancing but increases communication time, and that adding more processes does not always lead to improved total time due to communication overheads.

7. Critical Note

The small reduction in times in the case of larger dataset in the case of 32 and 16 processes are probably because of the following two reasons:

1. The loading of the larger dataset in the rank 0 process is a costly operation which consumes a large portion of the computation time.
2. More the number of processes, more time will be required for communication of the data chunks.

The binary swap consumes more communication time probably because in each iteration, all nodes are participating in communication, creating a communication bottleneck ,however, in each iteration all processes, do the compositing task, but after all processes need to communicate their final partial image to rank 0 process in stages.

8. Command to execute the code

To run the code, use the following command:

```
mpirun –mca btl_tcp_if_include eno1 -hostfile hostfile -np 8 python3 execuatble.py 8 Isabel_1000x1000x200_float32.raw 2 2 2 1 opacity_TF.txt color_TF.txt
```

```
mpirun –mca btl_tcp_if_include eno1 -hostfile hostfile -np 8 python3 binary_swap.py 8 Isabel_1000x1000x200_float32.raw 2 2 2 1 opacity_TF.txt color_TF.txt
```