CS 677 Assignment 2: Parallel 3D Volume Rendering

Introduction

This assignment aims to perform parallel volume rendering on a 3D scalar dataset using the Message Passing Interface (MPI). The rendering uses a ray casting algorithm that simulates an orthogonal projection where the view is aligned with the XY plane. The input data is a 3D scalar field, which we decompose into subdomains in 3 dimensions and distributed among multiple processes. Each process handles a subdomain and performs ray casting through it, accumulating color and opacity values along the Z-axis. The results from each process are gathered and composited to form the final output image.

Methodology

1. **Input Parameters:**
   Number of processes P: Total number of processes across all servers. Specified using -np.
   Dataset: The input 3D scalar dataset, which is specified as a file (e.g., Isabel_1000x1000x200_float32.raw).
   PX: Number of processes in X-dimension
   PY: Number of processes in Y-dimension
   PZ: Number of processes in Z-dimension
   Stepsize: The distance between two sample points along the ray, which defaults to 0.5 but can be set to any value by the user.
   Opacity TF: Relation between the voxel values and corresponding opacity.
   Color TF: Relation between the voxel values and corresponding color.

2. **Domain Decomposition:**
   a. The domain is decomposed along all 3 dimensions (X, Y and Z).
   b. The number of processes in each dimension is given as input.

3. **Parallel Volume Rendering:**
   a. **File Reading (Rank 0):** The root process (Rank 0) reads the input data file and distributes the necessary subdomains to all processes. Each process receives the portion of data it needs to perform ray casting within its assigned subdomain.

   b. **Ray Casting in Parallel:** Each process performs ray casting using the front-to-back compositing method. This involves casting rays along the Z-direction through the grid points in the XY plane, accumulating color and opacity values while applying the **early ray termination** technique to stop ray traversal when opacity reaches a certain threshold.

   c. **Data Exchange:** After processing, each process sends its partial results back to the root process (Rank 0).

d. **Image Stitching and Output (Rank 0):** The root process gathers all partial images from the worker processes, combines them into the final unified output image, and saves it as a PNG file. This assembly uses the same front-to-back compositing technique to accumulate color and opacity values consistently along the Z-dimension.

4. **Output:**
   a. **Rendered Image:** The final volume-rendered image of the dataset is saved as a PNG file.
   b. **Time for each Subprocess:** The time for the various subprocesses, namely, data reading, domain decomposition, ray casting, image recomposition and communication are recorded.
   c. **Total Execution Time:** The maximum execution time taken by any process is recorded and reported as the total execution time.

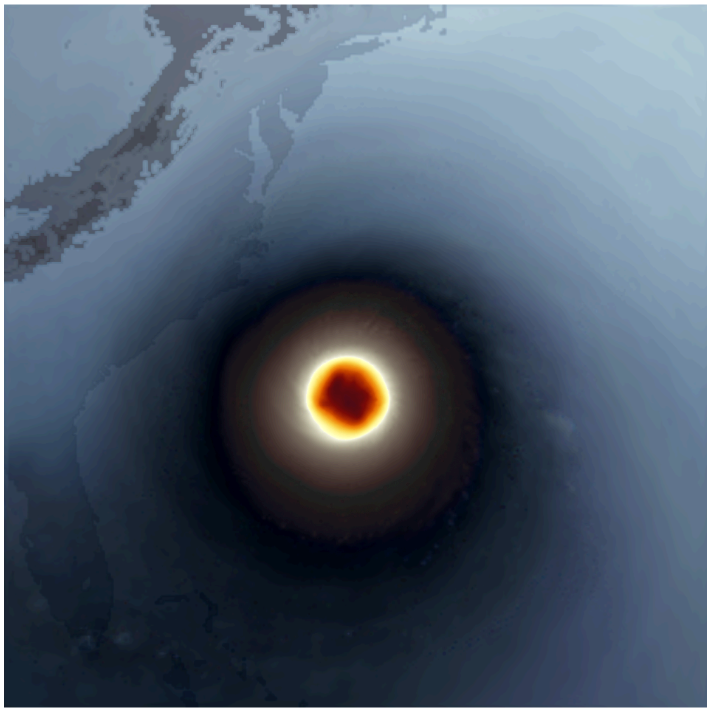
Results

- **Total Execution Time:** The time taken by each process is recorded, and the maximum time among all processes is reported as the total execution time, indicating the overall performance of the parallel rendering.

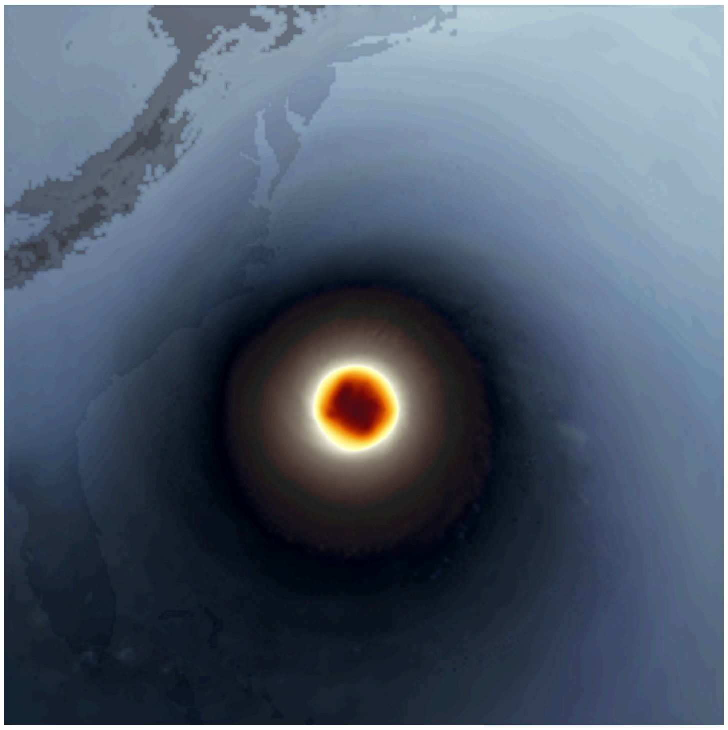
Command	Total execution time		Computation time		Communication Time	
Test Case 1 mpirun -np 8 -f hostfile ./executable Isabel_1000x1000x200_float32.raw 2 2 2 0.5 opacity.json color.json	41.0225	43.8449	40.9631	43.7780	1.4413	1.6572
Test Case 2 mpirun -np 16 -f hostfile ./executable Isabel_1000x1000x200_float32.raw 2 2 4 0.5 opacity.json color.json	46.0719	46.0660	27.9931	29.6712	5.5912	5.4954
Test Case 3 mpirun -np 32 -f hostfile ./executable Isabel_1000x1000x200_float32.raw 2 2 8 0.5 opacity.json color.json	47.0539	47.1104	18.7178	18.4636	17.0387	17.4386

- **Output Images**

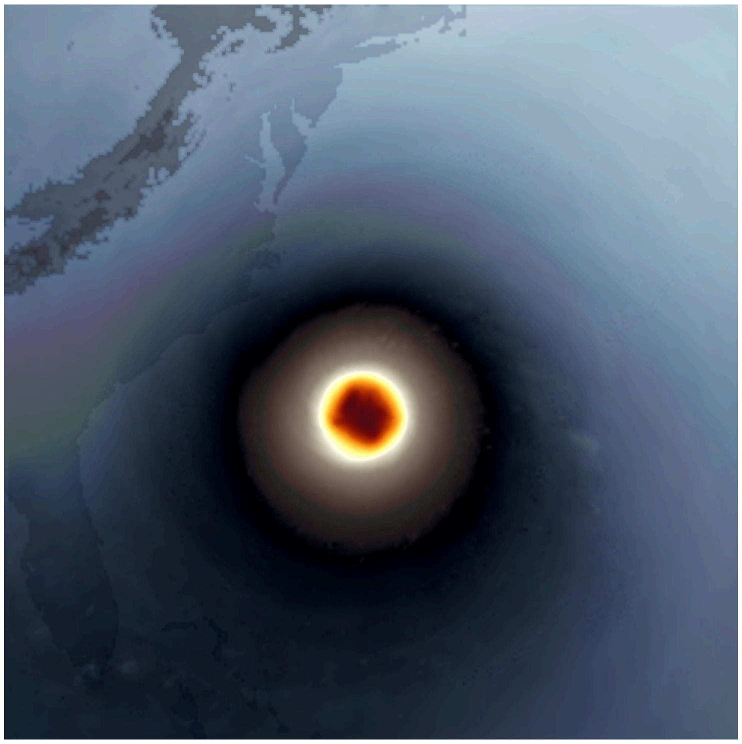
1 i)



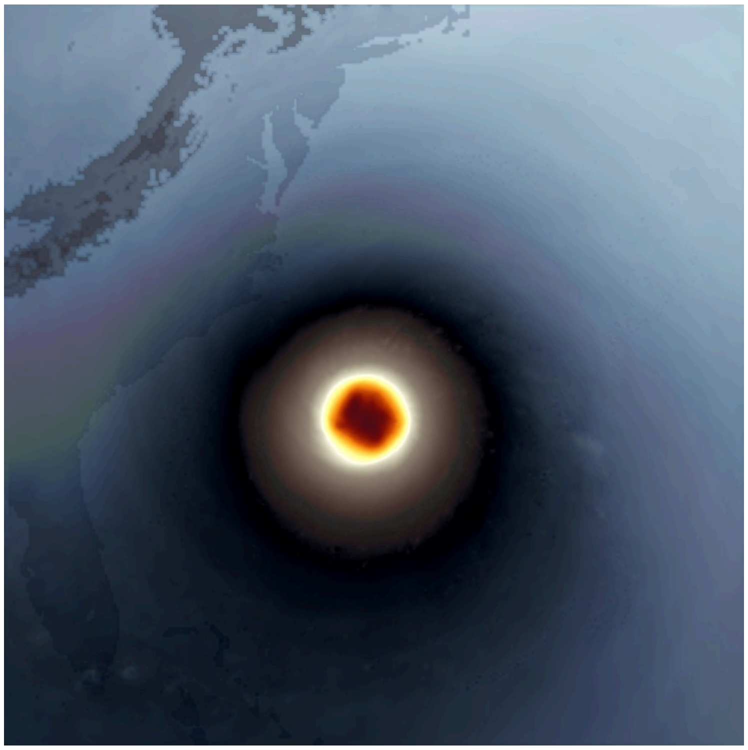
1 ii)



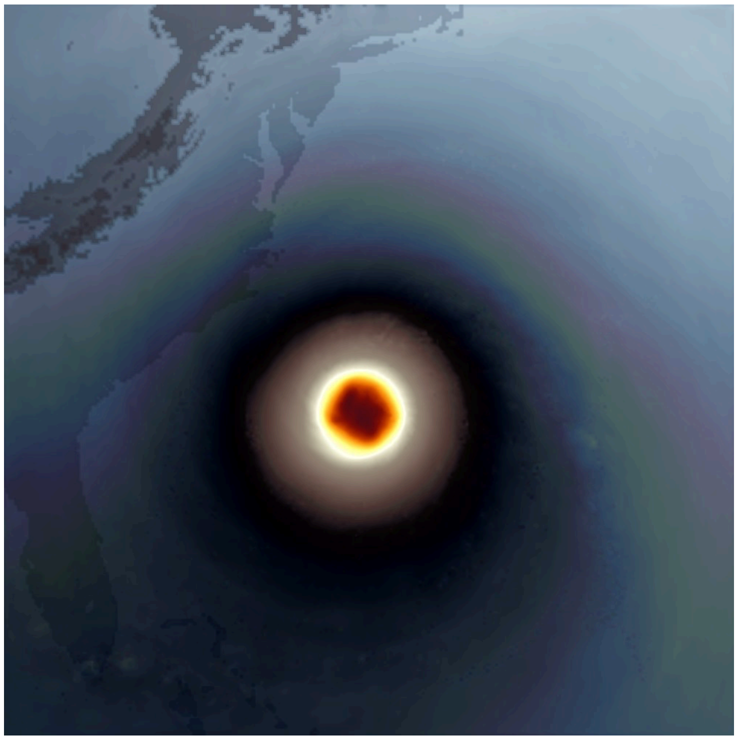
2 i)



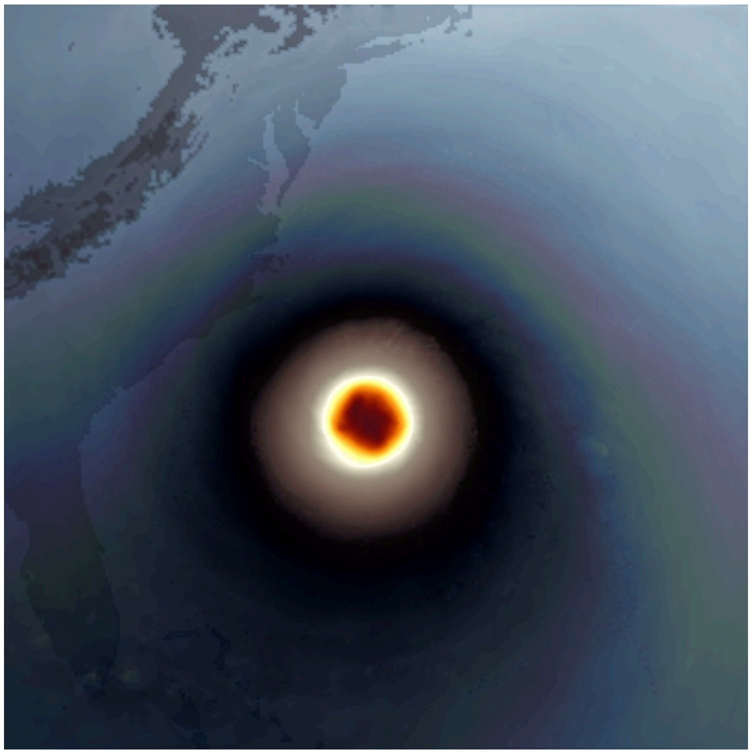
2 ii)



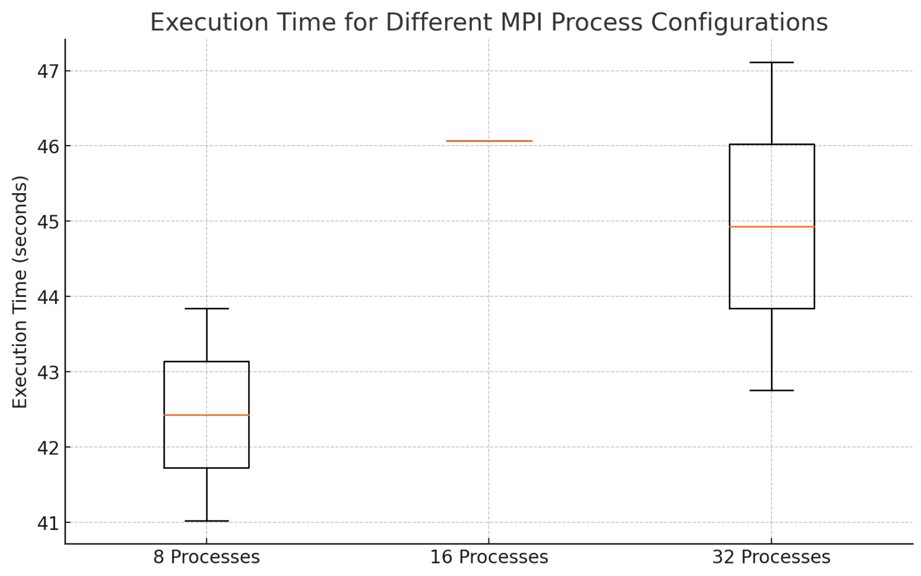
3 i)

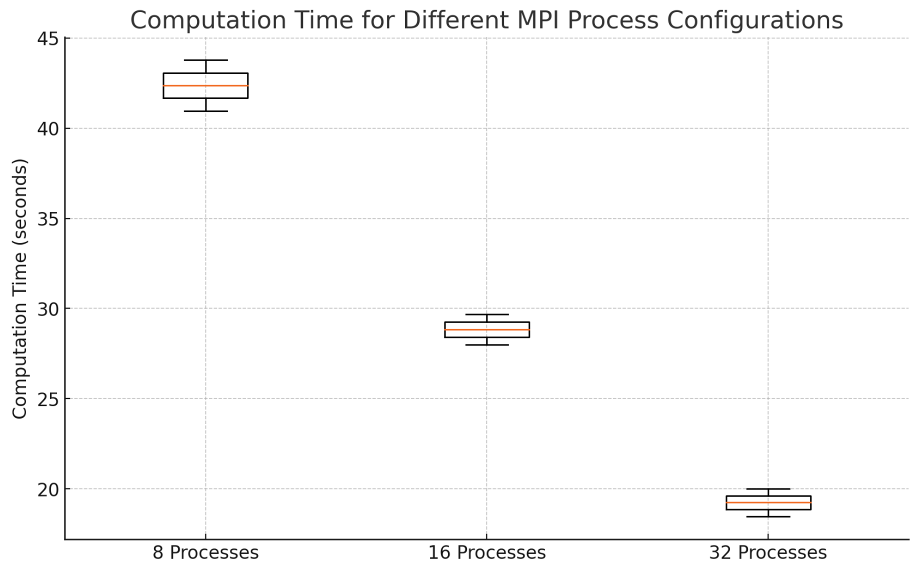


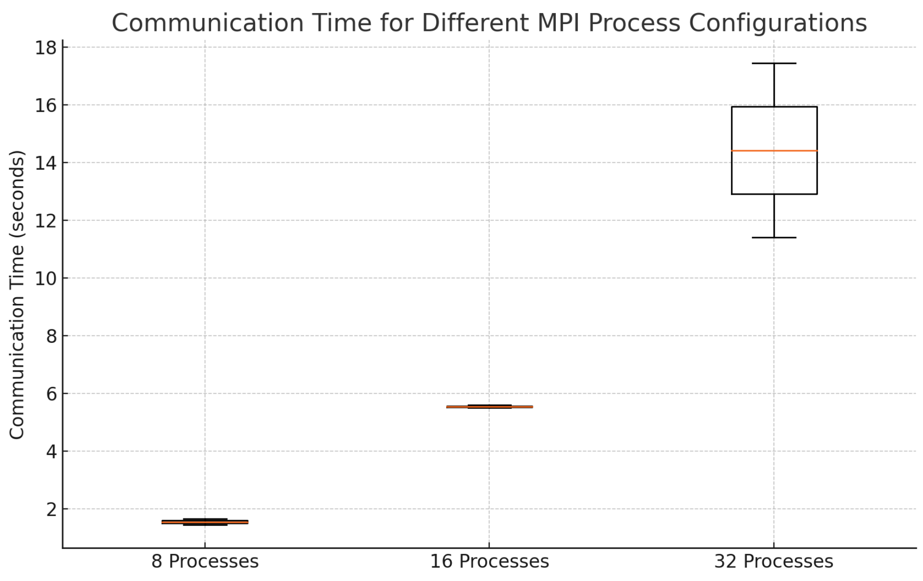
3 ii)



- **Scalability:**



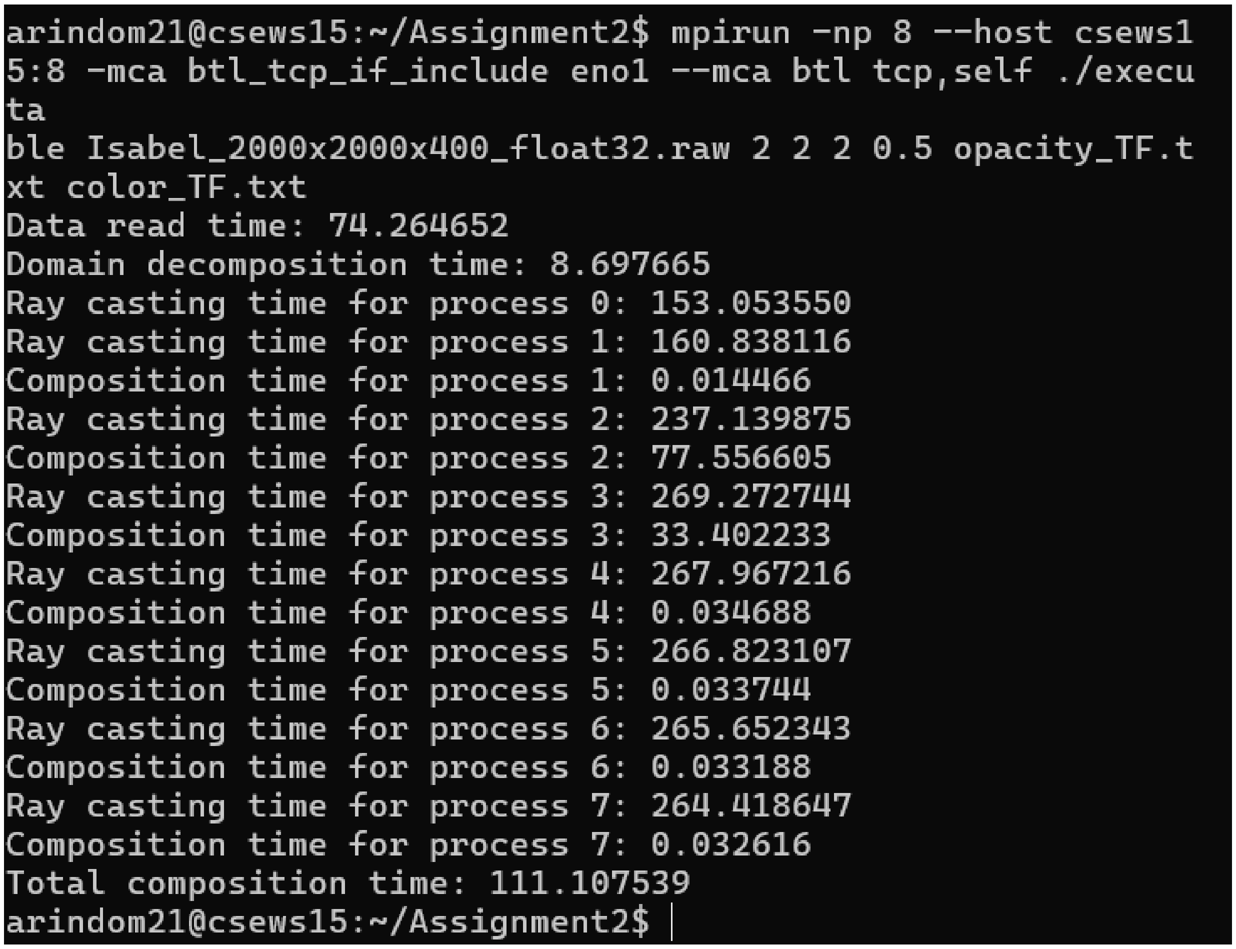




Scalability & Load imbalance on Larger Dataset

Command	Total execution time	Computation time	Communication Time
16 processes	225+90+46 = 361 seconds	225 seconds	46 seconds
8 processes	46.0719	27.9931	5.4954

Total observed to be decreased from 16 processes (361 sec) to 8 processes (46 sec). The dataset could not be used for other processes due to unavailability of nodes. For the load imbalance, some nodes had higher time with ray casting and compositing time both contributing to the increase.



```
arindom21@csews15:~/Assignment2$ mpirun -np 8 --host csews1
5:8 -mca btl_tcp_if_include eno1 --mca btl tcp,self ./execu
ta
ble Isabel_2000x2000x400_float32.raw 2 2 2 0.5 opacity_TF.t
xt color_TF.txt
Data read time: 74.264652
Domain decomposition time: 8.697665
Ray casting time for process 0: 153.053550
Ray casting time for process 1: 160.838116
Composition time for process 1: 0.014466
Ray casting time for process 2: 237.139875
Composition time for process 2: 77.556605
Ray casting time for process 3: 269.272744
Composition time for process 3: 33.402233
Ray casting time for process 4: 267.967216
Composition time for process 4: 0.034688
Ray casting time for process 5: 266.823107
Composition time for process 5: 0.033744
Ray casting time for process 6: 265.652343
Composition time for process 6: 0.033188
Ray casting time for process 7: 264.418647
Composition time for process 7: 0.032616
Total composition time: 111.107539
arindom21@csews15:~/Assignment2$
```

Example snippet for load on 8 processes for the larger dataset.

Challenges Faced

During the execution of the parallel volume rendering tasks, we encountered a notable issue related to the configuration of network interfaces across the hosts used. Specifically, an "unexpected process identifier" error appeared, leading to failed communication between MPI processes on different hosts. This error typically arises when non-routable or redundant network interfaces, such as virtual bridge interfaces (`eno1, virtbr0`), or inconsistent primary network interfaces are present across the nodes.

After investigation, we determined that the presence of different network interfaces on each host was causing MPI to misinterpret the correct path for inter-node communication. The solution involved explicitly specifying the `btl_tcp_if_include` parameter to limit MPI to the `eno1` interface, which was common and routable on all hosts.

To implement this, we modified the command as follows:

```
mpirun -np 16 --host csews16:8,csews19:8 --mca btl_tcp_if_include eno1
--mca btl tcp,self ./executable .. .. ..
```

This adjustment resolved the communication issues by ensuring that MPI only used the specified eno1 interface which was **common amongst all csews hosts**, thus bypassing any non-routable or inconsistent interfaces across the hosts. This experience highlights the importance of network interface consistency for successful MPI-based communication across distributed nodes in a high-performance computing environment.

Using the following github discussion:
https://github.com/open-mpi/ompi/issues/6240

Discussion

This project demonstrates the effective use of MPI for high-performance parallel volume rendering in a distributed memory setup. By decomposing the dataset into subdomains and applying the front-to-back compositing method, the program balances the workload across multiple processes, scaling efficiently with larger datasets. The results highlight the adaptability of the algorithm, which can accommodate various domain decomposition strategies and process configurations. This flexibility is crucial for rendering large datasets on different hardware configurations, achieving both scalability and efficiency.

Conclusion

The implemented parallel 3D volume rendering technique successfully leverages MPI to achieve efficient visualization of large volumetric datasets. The algorithm's design allows flexible customization for domain decomposition, step sizes, and transfer functions, making it a versatile tool for scientific visualization. The performance metrics, including early ray termination fractions and total execution times, confirm the effectiveness of the method in delivering high-quality renderings while managing computational resources efficiently.

Future Improvements:

Future improvements could focus on dynamic load balancing based on data density, optimizing domain decomposition to account for heterogeneous voxel distributions, and exploring alternative compositing methods to further reduce inter-process communication overhead.

Future work could explore enhanced load balancing, dynamic domain decomposition, and further optimization of ray traversal paths based on voxel characteristics. This would refine the performance further and make the approach even more suitable for larger-scale, complex datasets in high-performance computing environments.