CS 677 Assignment 1: Parallel Distributed Axis-aligned Volume Rendering using MPI

GROUP NO. : 9

Krishna Kumar Bais
Arnika Kaithwas
Pradeep Sahu

Introduction

This report describes an implementation of a volume rendering technique using MPI (Message Passing Interface) to efficiently process and visualize 3D volumetric data. The code reads raw data, applies color and opacity mappings, and performs ray-casting to generate a 2D image representation of the volumetric data. The implementation leverages the `mpi4py` library for parallel computation and the `PIL` library for image handling.

Input Files

The input files required for the program are specified in the `run_rendering_task()` function. The program requires the following input files:

1. Raw Dataset (`Isabel_1000x1000x200_float32.raw`): This binary file contains the 3D volumetric data that will be processed and visualized.
2. Color Transfer Function (`color_TF.txt`): This text file maps data values to specific RGB colors, allowing for the visualization of different data values in color.
3. Opacity Transfer Function (`opacity_TF.txt`): This text file maps data values to opacity levels, determining how transparent or opaque each voxel appears in the final rendered image.

Code Structure

1. MPI Initialization

The code begins by setting up the MPI environment, allowing multiple processes to work together:

```
mpi_comm = MPI.COMM_WORLD
mpi_rank = mpi_comm.Get_rank()
mpi_size = mpi_comm.Get_size()
```

2. Data Loading Functions

- `load_raw_data(filepath, dimensions)`: This function reads the raw volumetric data from a binary file and reshapes it into a 3D numpy array based on the specified dimensions.
- `parse_color_map(filepath)`: This function reads the color mapping file and constructs a dictionary that maps voxel values to RGB color arrays.
- `parse_opacity_map(filepath)`: This function reads the opacity mapping file and constructs a dictionary that maps voxel values to opacity levels.

3. Interpolation Function

- `interpolate_value(value, value_map)`: This function performs linear interpolation to find intermediate color or opacity values based on the provided mappings. It sorts the keys of the mapping and calculates the interpolated value.

4. Data Distribution Function

- `distribute_2d_array(data_array, process_id, total_processes)`: This function distributes a 2D slice of the 3D data array to each MPI process based on its rank. It calculates the dimensions of the blocks to be distributed and extracts the corresponding sub-array for the current process.

5. Volume Rendering Function

- `volume_render(sub_block, color_map, opacity_map, step_size=1.0, termination_threshold=0.95)`: This function implements the ray-casting algorithm to perform volume rendering on a sub-block of the data. It iterates through each pixel, casting rays through the volume and accumulating color and opacity values until the termination threshold is reached.

6. Main Rendering Task Function

- `run_rendering_task()`: This is the main function that orchestrates the entire rendering process. It loads the raw data and mappings, distributes the data among MPI processes, performs volume rendering on each sub-block, and gathers the results. Finally, it combines the rendered sub-images into a final output image and saves it as a PNG file.

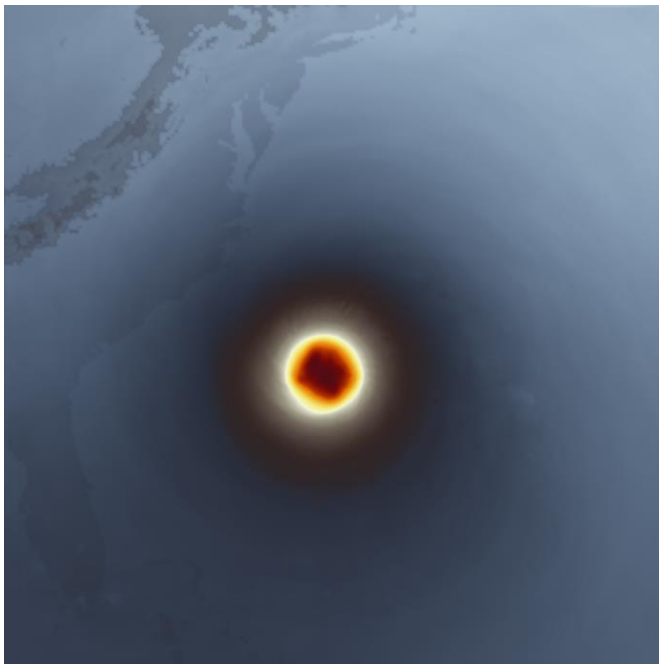
7. Execution

The program is executed by calling the `run_rendering_task()` function within the `if __name__ == "__main__":` block, which ensures that the code runs only when executed as a script.

Results and Discussion

After executing the code, the final rendered image is saved as `final_IMAGE.png`. The mean early termination ratio is printed, indicating the proportion of rays that were terminated early due to reaching the opacity threshold.

Output Image



Instructions for Running the Code

To run the volume rendering code, follow these steps:

- Ensure you have Python 3 installed on your system.

- Install the required dependencies:
  - NumPy: `pip install numpy`
  - mpi4py: `pip install mpi4py`
  - Pillow (PIL): `pip install pillow`

- Save the provided Python script (e.g., `volume_rendering.py`) in a directory of your choice.

- Place the necessary input files in the same directory as the script:
  - Raw dataset: `Isabel_1000x1000x200_float32.raw`
  - Color transfer function: `color_TF.txt`
  - Opacity transfer function: `opacity_TF.txt`

- Open a terminal or command prompt and navigate to the directory containing the script and input files.

- Launch the MPI environment with the desired number of processes. For example, to use 4 processes, run:

  ```
  mpirun -np 4 python volume_rendering.py
  ```

  Replace 4 with the number of processes you want to use.

- The code will execute, and the final rendered image will be saved as `final_IMAGE.png` in the same directory as the script.

- The mean early termination ratio will be printed to the console.

Modifications for Different Raw Dataset

If wants to use a different raw dataset file, such as `Isabel_2000x2000x400_float32.raw`, the following changes need to be made in the code:

- Update the Raw Data Path:
  Change the file path in the `run_rendering_task()` function:

  ```
  raw_data_path = 'Isabel_2000x2000x400_float32.raw'
  ```

- Adjust the Data Dimensions:
  Update the `data_dims` variable to match the dimensions of the new dataset:

  ```
  data_dims = (2000, 2000, 400)
  ```

Conclusion

The implementation successfully demonstrates a parallel volume rendering technique using MPI. The code efficiently loads and processes 3D data, applies color and opacity mappings, and generates a rendered image through ray-casting. The use of MPI allows for scalable performance, making it suitable for large volumetric datasets.