Parallel Ray Tracing Using MPI

1. Introduction

This project asks you to implement a parallel ray tracer using MPI (Message Passing Interface).

The goals of this project are:
(1) Build a MPI implementation of a raytracer.
(2) Implement different data partitioning schemes
(3) Test different communication topologies
(4) Speedup measure and experimental comparisons on different partitioning schemes

The basic idea of the a parallel ray tracing program can be described in Figure 1. It divides the pixel matrix consisting the figure into partitions, and assign each partition to a process running on the cluster. Each one will generate a partial pixel result and finally combine them together to get the final result.

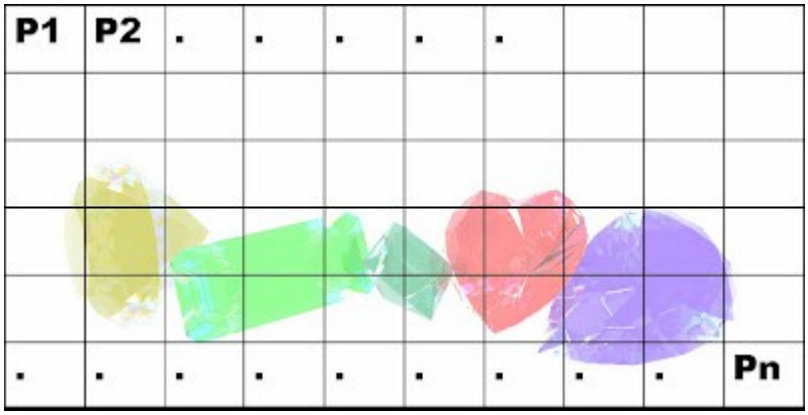

Figure 1, Basic Idea of Parallel Ray Tracing,(where P_n means the n-th process/processor)

2. Key Techniques

(1) Pixels Partition Scheme

First implement a static partitioning scheme (no load balancing) which sequentially assign (groups of) rows of the pixel matrix to processes/processors. As shown in Figure 2, P_1 to P_n are fairly assigned a group of rows in order.

Figure 2. Static Pixels Partitioning Scheme

One disadvantages of static partitioning scheme is the lack of load balancing between processors. One can tell from figure 2 that P_1 and P_2 may finish much quicker than, say, P_{n-1}, so the total running time will be constrained by the running time of P_{n-1}.

To improve the static scheme, implement load balancing schemes in which (groups of) pixels are assigned to processes/processors in a round robin way. One possible load balancing policy is to extend the static allocation of rows based on the process/processor rank, such that each process/processor gets to process the rows whose number equals the processor rank mod n. Another solution, shown in figure 3, would be to refine even more the “coverage” of the image by processes/processors by assigning (groups of) pixels to processors in a row-wise, snake-like traversal of the pixels of the image matrix.

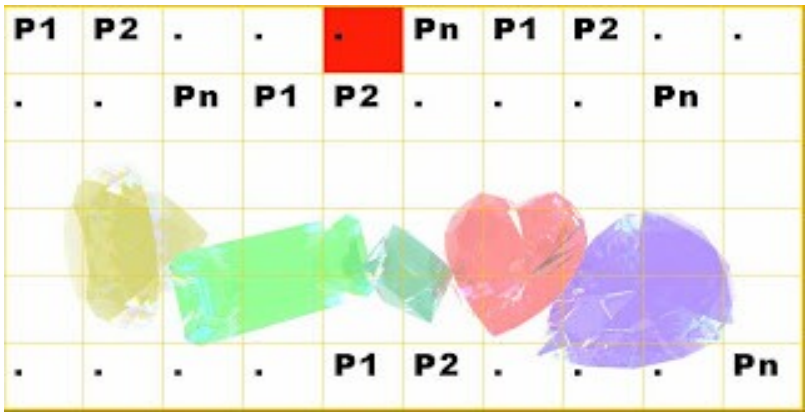

Figure 3. Load Balancing Pixel Partitioning Scheme: Each square (shown as red) can be a group of adjacent pixels or a pixel.

(2) Topologies

Choose different MPI predefined topologies (MPI_GRAPH, MPI_CART) and use them for the processes/processor interconnection topology.

(3) Output Combination

The combination of results from all processes can be illustrated in Figure 4 describing an example case where there is a 4 pixel matrix divided to two processes P1 and P2. At the computation step, P1 and P2 are actually working on their only color value indices as shown in yellow in each process's buffer. They also have according global indices in the receive buffer on rank 0 process (white) and the actual coordinates in pixel matrix. It is not hard to figure out how to transfer them to each other (implement that in your code).

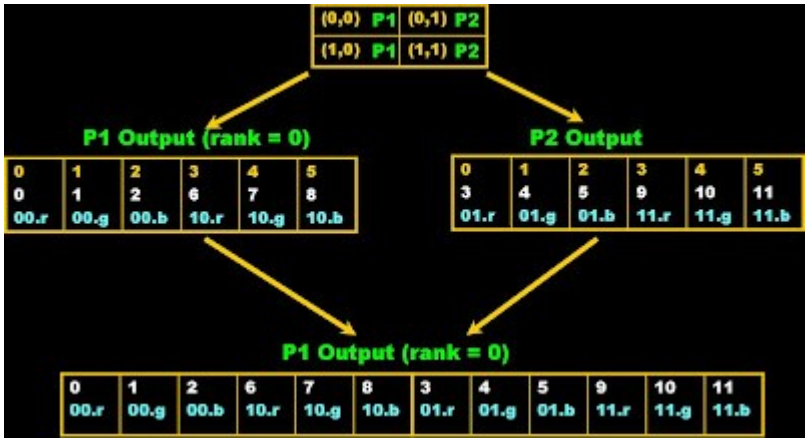

Figure 4: Result Combination Illustration

3. Experiments

(1) Describe your experimental environment

(2) Explain the parameters used in your experiments:
Data Size (image resolution): a x b pixels
Data block (pixel block size)
Scaling Number of Processes from m to n, stride k.
Comparing one-time MPI initialization time and rendering time
Averaging time of 20 runs as the finishing time result

(3) Write a report about the results that should contain the following measurements:

(a) Show speed-up and efficiency results for:

Load Blancing algorithm vs. Sequential solution:
Static partitioning vs. Sequential solution:

(b) Comparison between Load Balancing and static allocation solutions

Compare the methods as the number of processes increases and assess the impact of the communication cost between processes.

(c) Comparison between the network topologies

Compare the results of the load balancing solutions when running on different topologies and assess the impact of the communication cost for each topology used.