CMPE-655 Fall 2021 Assignment 2: Parallel Implementation of a Ray Tracer

Rochester Institute of Technology, Department of Computer Engineering
Instructor: Dr. Shaaban (meseec@rit.edu)
TAs: Prangon Das (pcd3897@rit.edu)
Rohan Challa (rc6831@rit.edu)

First Submission (Code only) Due: *Thursday, November 11 , 2021, 23:59:00 (week 12)*
Second Submission (Code only) Due: *Thursday, November 18 , 2021, 23:59:00 (week 13)*
Final Submission Due: ***Thursday, December 2 , 2021, 23:59:00 (week 15)***

Contents

Introduction

Ray tracing is an image rendering technique that is often referred to an as “embarrassingly parallel” problem because all pixels in the rendered image can be rendered independently of each other. There can easily be millions of pixels in a rendered image, each requiring its own ray tracing result. This means that there is a significant amount of work, nearly all of which can be done in parallel.

But ray tracing does have a catch, though. In ray tracing, the transport of light within a scene is calculated by tracing the path of the light backwards from the camera. Whenever an object intersects the ray being traced, a color value is calculated. If the intersected object is transparent or reflective, more rays are generated and traced, and the color values of those rays are applied to the original ray. In this way, ray tracing is a recursive problem. However, that recursion can lead to **a fairly unpredictable execution time per ray**, especially in scenes where there are many reflective or transparent objects.

Objectives

In this assignment you will be exploring the effect that grain size (task size) and job allocation methods (partitioning) have on the performance of a parallel program.

You will be given the core of a ray tracing program, and your task will be to implement several different partitioning schemes for the ray tracing problem, determine which scheme gives the best performance increase over a sequential version of the program, and explain why each performed as it did.

You must implement the following four partitioning schemes:

- **1- Static partitioning using contiguous strips of columns**
- **2- Static partitioning using square blocks**
- **3- Static partitioning using cyclical assignments of n rows**
- **4- Dynamic partitioning using a centralized (one) task queue (maintained by master process)**

To test each of these schemes, two scenes will be given to render. The first will be a sparse, simple scene, containing very few shapes with which rays can interact. The second scene will contain significantly more objects to render. Additionally, you will be testing to see which task grain size fares best in the dynamic partitioning implementation.

Program Requirements

- All time recordings should be taken on the master as well as the slaves using the function calls below:

```
double comp_start, comp_stop, comp_time;
comp_start = MPI_Wtime();
       // Pure computation, no communication.
comp_stop = MPI_Wtime();
comp_time = comp_stop – comp_start;
```

- Slaves (workers) are to ship (send) their time measurements to the master in the same packet (message) in which they transmit their related computation. The master must calculate the communication to computation (C-to-C) ratio using similar measurements—the master must record the amount of time it spends within communication functions, and the amount of time it spends within computation functions.
- When using one of the static partitioning models, the master (e.g. rank =0) must perform as much work as the slaves.
- All partitions must be of uniform size (or as close to uniform as possible).
- Everything must be contained within a single program.
- Your program must work on **mpssubmit.rc.rit.edu**, regardless of where you develop it. All timing measurements must be taken on the cluster.
- For static allocation methods, there must be only one communication: the sending from the slaves to the master of computation results.
- All parallel implementations using static partitioning methods must work with any number of processes, including 1. Implementations using dynamic partitioning must work with any number of processes greater than or equal to 2.
- Your programs must be called **`raytrace_seq`** and **`raytrace_mpi.`**
- Your program must accept arbitrary sizes; both the size of the image and the size of the partitioning segments must not have a fixed size.
- The formatting of the program output must not be changed.

Partitioning

You are required to implement both static and dynamic partitioning models in this assignment.

Static partitioning

When using static partitioning, both the master and slave computing elements must perform rendering work—for static partitioning, the only difference between master and slaves is that the master saves the final rendered image.

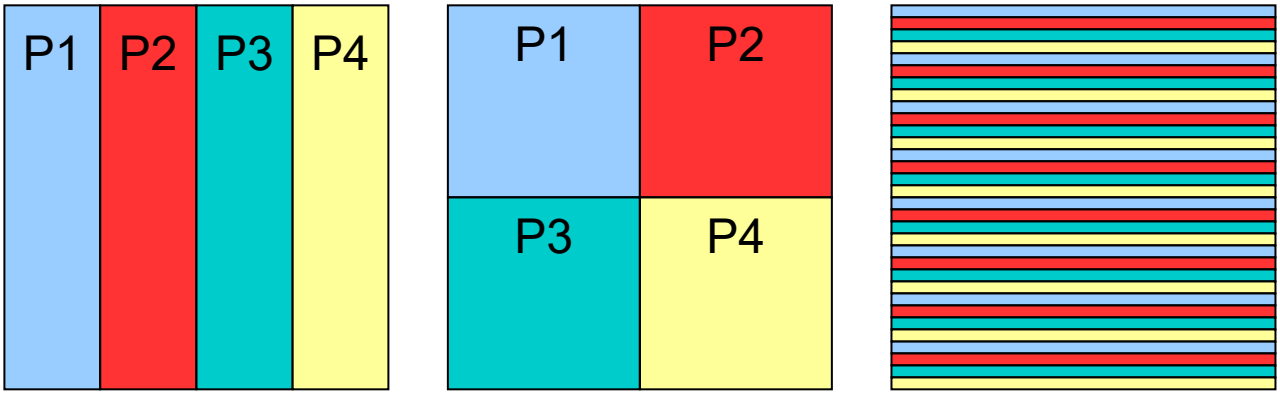


Figure 1: Static Partitioning Schemes

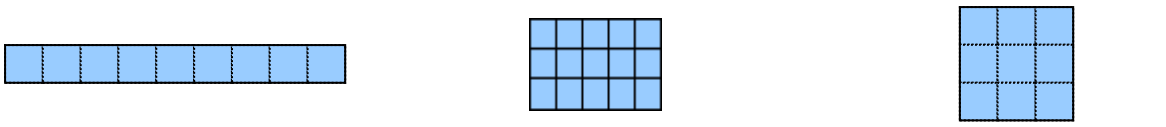
Figure 1 illustrates the three static partitioning schemes. On the left is the partitioning into **contiguous blocks (or columns)** that span the height of the image. For this scheme, the blocks span the entire height of the image and evenly divide the width of the image.

In the middle of Figure 1 is the partitioning into **square blocks** that tile the image. For this scheme you must figure out a good way to handle the case where the scene is not evenly divided.

On the right is the **cyclical partitioning into rows**. In this scheme, the blocks span the entire width of the image and are ***n* rows** tall.

Dynamic partitioning

For dynamic partitioning a centralized single queue must be used. The master process (e.g. rank =0) will handle giving out and collecting work units (**the master will not perform any rendering**), while the slaves handle all the rendering.



Line Segment (1x9 pixels) Rectangular (3x5 pixels) Square Block (3x3 pixels)

Figure 2: Example work unit sizes

It is your responsibility to decide on an appropriate method for managing work units, communicating data to and from the slaves, handling termination of the program, etc.

Your program must accept an arbitrary work unit size (given by the height/width of a block). You are to decide the best option to handle left over space from an uneven division of picture dimensions, however, you will be required to justify your choice. A variety of block sizes will be tested.

Required Measurements

- You must gather data to fill in the following tables.
- All rendered images **must be 5000x5000 pixels**.
- You must gather data for the following tables for **both the simple scene and the complex scene.**

Sequential	
raytrace_seq	

Table 1: Sequential runtime

Static Strip of Columns Allocation				
Number of Processes	Number of Strips	Execution Time (s)	Speedup	C-to-C ratio
1 (srun –n 1)	1			
2 (srun –n 2)	2			
4 (srun –n 4)	4			
9 (srun –n 9)	9			
16 (srun –n 16)	16			
20 (srun –n 20)	20			
25 (srun –n 25)	25			
36 (srun –n 36)	36			
49 (srun –n 49)	49			
55 (srun –n 55)	55			
64 (srun –n 64)	64			

Table 2: The effect of work unit size on computational efficiency using strip allocation

Static Block Allocation				
Number of Processes	Number of Blocks	Execution Time (s)	Speedup	C-to-C ratio
1 (srun –n 1)	1			
4 (srun –n 4)	4			
9 (srun –n 9)	9			
16 (srun –n 16)	16			
25 (srun –n 25)	25			
36 (srun –n 36)	36			
49 (srun –n 49)	49			
64 (srun –n 64)	64			

Table 3: The effect of work unit size on computational efficiency using block allocation

Static Cyclical n Row Allocation				
Number of Processes	Height of strip in pixels	Execution Time (s)	Speedup	C-to-C ratio
4 (srun –n 4)	1			
4 (srun –n 4)	5			
4 (srun –n 4)	10			
4 (srun –n 4)	20			
4 (srun –n 4)	80			
4 (srun –n 4)	320			
4 (srun –n 4)	640			
4 (srun –n 4)	1280			
9 (srun –n 9)	1			
9 (srun –n 9)	5			
9 (srun –n 9)	10			
9 (srun –n 9)	20			
9 (srun –n 9)	40			
9 (srun –n 9)	160			
9 (srun –n 9)	350			
9 (srun –n 9)	650			
16 (srun –n 16)	1			
16 (srun –n 16)	5			
16 (srun –n 16)	10			
16 (srun –n 16)	20			
16 (srun –n 16)	50			
16 (srun –n 16)	100			
16 (srun –n 16)	250			
16 (srun –n 16)	400			

Table 4: The effect of work unit size on computational efficiency using cyclical allocation

Static Cyclical n Row Allocation				
Number of Processes	Height of strip in pixels	Execution Time (s)	Speedup	C-to-C ratio
1 (srun –n 1)	27			
2 (srun –n 2)	27			
4 (srun –n 4)	27			
9 (srun –n 9)	27			
16 (srun –n 16)	27			
20 (srun –n 20)	27			
25 (srun –n 25)	27			
36 (srun –n 36)	27			
49 (srun –n 49)	27			
55 (srun –n 55)	27			
64 (srun –n 64)	27			

Table 5. The effect of increasing the number of processors on a fixed allocation size

Dynamic Allocation				
Number of processes	Work Unit Size (rows x columns)	Execution Time (s)	Speedup	C-to-C Ratio
9 (srun –n 9)	1x1			
9 (srun –n 9)	15x15			
9 (srun –n 9)	25x25			
9(srun –n 9)	50x50			
9 (srun –n 9)	75x75			
9 (srun –n 9)	100x100			
16 (srun –n 16)	1x1			
16 (srun –n 16)	15x15			
16 (srun –n 16)	25x25			
16 (srun –n 16)	50x50			
16 (srun –n 16)	75x75			
16 (srun –n 16)	100x100			

Table 6: The effect of work unit size on computational efficiency using dynamic block allocation

Dynamic Allocation				
Number of processes	Work Unit Size (rows x columns)	Execution Time (s)	Speedup	C-to-C Ratio
16 (srun –n 16)	11x1			
16 (srun –n 16)	1x10			
16 (srun –n 16)	7x13			
16 (srun –n 16)	5x29			
16 (srun –n 16)	9x110			
16 (srun –n 16)	70x1			
36 (srun –n 36)	11x1			
36 (srun –n 36)	1x10			
36 (srun –n 36)	7x15			
36 (srun –n 36)	5x29			
36 (srun –n 36)	9x110			
36 (srun –n 36)	70x1			

Table 7: The effect of work unit shape on computational efficiency as well as test for arbitrary work unit size

Report

Within your report you must:

- Include an abstract, a detailed design of your solution, a results section where you present your findings, in addition to your analysis, and a conclusion.
- Explain how you performed the partitioning and how you handled the extra work if there was some leftover area.
- Discuss the effects of parallel implementation and explain why these effects are present.
- Compare and contrast the efficiency of each method.
- From Table 4 obtain a graph with three curves with work unit size on the X-axis and execution time on the Y-axis. One curve should be for each quantity of processes.

- From Table 5, create a graph showing the speedup of the work distribution. The x-axis will contain the number of processors and the y-axis will be the speedup. This plot needs to be made for both the simple and the complex image. Discuss the results that you see in the graph.
- Graphs and tables should be **properly** labeled and introduced in the text before they appear. Afterwards, they **must** be analyzed.
- Discuss the effect of different grain shapes/sizes when using the centralized task queue, with appropriate performance measurements to support your discussion.
- Explain why each partitioning scheme performed as well/poorly as it did, mention which scheme you would recommend for general use, and justify your answers.
- The idea for the report is to provide a complete and concise overview of the entire assignment. We should not have to start looking through the code in order to find out how you did things. Additional diagrams should be placed in your report to aid the understanding of image partitioning when it cannot be broken up perfectly.

Submission Schedule

1. Due Thursday, **November 11th** at 23:59:00 – **One** of the four partitioning schemes (of your choice). Be clear on the scheme you choose.
2. Due Thursday, **November 18th** at 23:59:00 – **Two** of the four partitioning schemes (**one more**).
3. Due Thursday, **December 2nd** at 23:59:00 – **Final project submission** (all four partitioning schemes and report).

- **NOTE:** Submissions 1 and 2 are **CODE ONLY submissions**. Create a bzipped tarball of the *project* directory that contains all of the provided files and folders. The submission will be made to a dropbox on myCourses.
- Grading of submissions 1 and 2 will be based on the effort put into the method. The partitioning scheme does not need to be perfectly correct however the code must compile and run and be mostly correct. This is mostly for you to keep on track and start early.
- A report is required to get any credit for the assignment. If no report is submitted, the overall grade for the assignment will be a zero regardless of how much code is completed.

Grading

- 5 points for Code Submission 1 (November 11th)
- 5 points for Code Submission 2 (November 18th)
- 50 points for correct program execution (final submission)
    - 8 points for strip allocation
    - 12 points for block allocation
    - 12 points for cyclical allocation
    - 18 points for dynamic partitioning
- 10 points for implementation, coding style, and performance
    - Implementation – Does the code *look like* it will run correctly?
    - Coding Style – Readability and commenting
    - Performance – Memory footprint and speedup achieved
- 30 points for the report

Testing and Debugging

- Test your code before submitting. If your submission does not compile, no credit will be given for program execution. Corrected submissions will be accepted and graded according to the standard late policy.
- A utility program is provided for you to compare two images. The program will tell you the number of differences between the two images. This program will be used to grade your submissions, so it may be worth your time to verify that you are rendering the images properly.
- Do not attempt to render the complex scene until you have verified that you correctly render the simple scene.
- Start rendering with small scenes. If your program does not correctly render a 200x200 pixel image, then it will not render a 5000x5000 pixel image.
- Don't test everything all at once. Develop in stages. There's no reason for you to try rendering the scene when you haven't yet verified that the master and slave are communicating properly.
- Use the *debug* partition of the cluster to test your progress. This is a good way to avoid hogging resources that other students may be using to collect their actual results.
- After verifying that your intermediate submissions are correct on the *debug* partition, you can then run all of your jobs on the *class* partition while continuing development of the remaining partitioning methods.

The Ray Tracer Code

A ray tracing engine is provided to you in this assignment. You are not required to know any of the internal logistics of the ray tracing algorithm, except that its execution time can vary greatly depending on the makeup of the rendered scene. You will be provided with an object file archive, a set of header files, and a basic sequential implementation, all of which you will use when implementing your parallel solutions.

The provided code was designed to require as little knowledge as possible about the inner workings of the ray tracer. There are only four functions and one struct needed to use the raytracer, and one additional function provided for convenience.

struct ConfigData

All data needed by your program and the ray tracer itself is contained within the `struct ConfigData` struct type, defined in the `RayTrace.h` file. This struct contains a few fields of interest, all of which are described in the `RayTrace.h` file.

int initialize(int *argc, char ***argv, ConfigData *data)

The initialize function does three things:

- Parses the command line arguments
- Loads the scene to render
- Fills in the given ConfigData struct with the information your program needs.

This function must be called by all processes, otherwise the ray-tracer engine will not know what to render.

void shutdown(CondigData *data)

The shutdown function must be one of the last functions called by all processes. This function performs one action:

- Cleans up resources allocated by the call to `initialize()`

void shadePixel(float *color, int x, int y, ConfigData *data)

The shadePixel function is the core of the ray-tracer. Given an `x` and a `y` coordinate, and the scene information contained with the ConfigData struct, the shadePixel function will calculate the color to assign to the (`x, y`) pixel and will store that color in the `color` parameter. Note that coordinate (`0,0`) corresponds to the bottom left of the image. The given `color` parameter must point to an area of memory where three single-precision floating point values may be written. These values are the RGB triplet representing the color at the (`x, y`) coordinate.

short savePixels(char *filename, ConfigData* data)

The savePixels function saves the rendered image to the file specified by `filename`. The `width` and `height` parameters should be the same as those specified on the command line and stored within the ConfigData struct.

The `pixels` parameter must contain all of the image pixel values, thus, the `pixels` parameter must contain (`width * height * 3`) floating point values. These values must be stored in **row-major** order, starting with the bottom row. In other words, the RGB color rendered for:

- Coordinate (`0, 0`) must be stored in
  - `pixels[0]`
  - `pixels[1]`
  - `pixels[2]`
- Coordinate (`0, 1`) must be stored in
  - `pixels[width * 3 + 0]`
  - `pixels[width * 3 + 1]`
  - `pixels[width * 3 + 2]`
- Coordinate (`x, y`) must be stored in
  - `pixels[(y * width + x) * 3 + 0]`
  - `pixels[(y * width + x) * 3 + 1]`
  - `pixels[(y * width + x) * 3 + 2]`

Programming Tips

- Get started early. This project is much more programming and resource intensive than the previous project.
- Modularize your code. Avoid having one monolithic function. This will make testing and debugging easier.
- Try developing at home. The provided code should compile on any x86-64 version of Linux.

Cluster Usage Tips

- Whenever you cancel a job or whenever your program crashes, run the following set of commands:

  ```
  scancel –u username
  orte-clean
  ```

  This will help keep the cluster working for everyone.

- Don't hog the cluster. Yes, you *can* run all your jobs simultaneously, but please don't. It prevents other people from running their jobs.
- Monitor your jobs – if more than 12 heavy worker threads are running on the same node, your **results will be incorrect**. Kill the job and try again. (Hint: Start early and avoid problems!)
- Do NOT change the $SLURM_NPROCS environment variable in the provided sbatch scripts.

Rendered Scenes

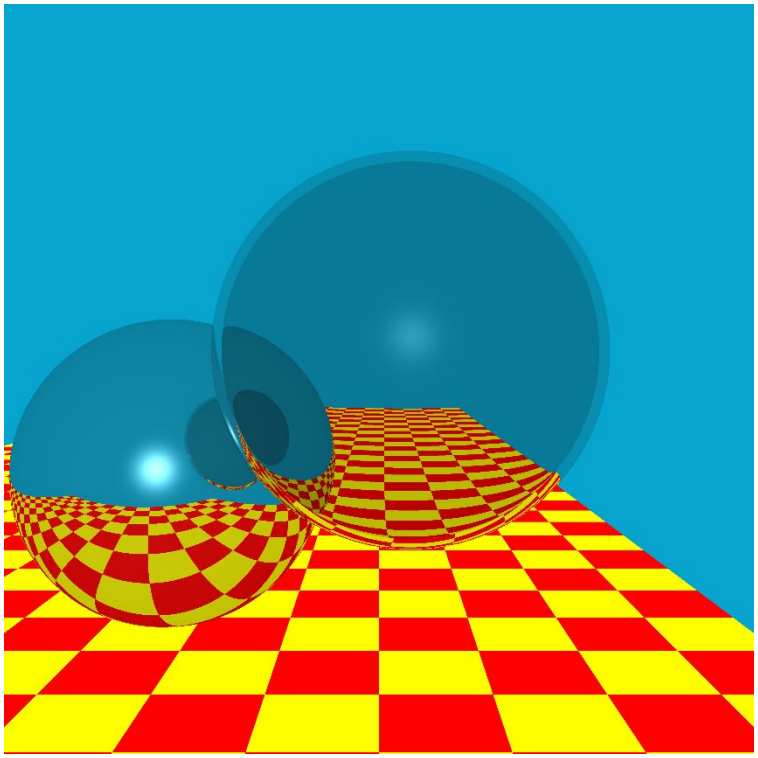


Figure 3: Turner Whitted's classic ray trace scene

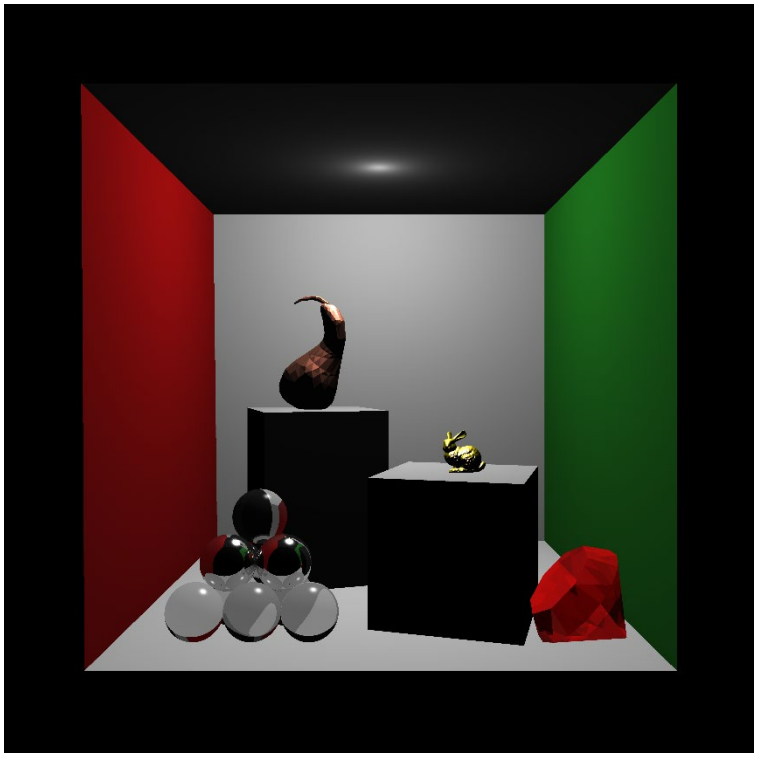


Figure 4: Complex ray traced scene

Credit

This ray tracing engine was written by Jason Lowden. The model rendered within the complex scene included the Stanford Bunny. This model is used very often to demonstrate the capabilities of a computer graphics system. You can find more information at http://graphics.stanford.edu/data/3Dscanrep/

To Learn More

If you're interested in learning more about ray tracing, other rendering methods, or just computer graphics in general, RIT has a very good computer graphics course line that covers all aspects of the computer graphics pipeline, and there is a computer graphics seminar that meets weekly to learn about interesting work in the field. The seminar's website: http://www.cs.rit.edu/~rjb/cgseminar.htm.