EECC-756 Spring 2010 Assignment 2: Parallel Implementation of a Ray Tracer

Rochester Institute of Technology, Department of Computer Engineering
Instructor: Dr. Shaaban (meseec@rit.edu), TA: Dan Brandt (dtb4355@rit.edu)
Due: Tuesday, May 11, 2010 at noon

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.

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 and job allocation methods 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 partitioning schemes:

- Static partitioning using contiguous strips of rows
- Static partitioning using square blocks
- Static partitioning using cyclical assignments of rows
- Dynamic partitioning using a centralized task queue

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 are to ship their time measurements to the master in the same packet in which they transmit their related computation. The master must calculate the communication to computation 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 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 cluster.ce.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.
- Your program must be called `raytrace`.
- 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.

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.

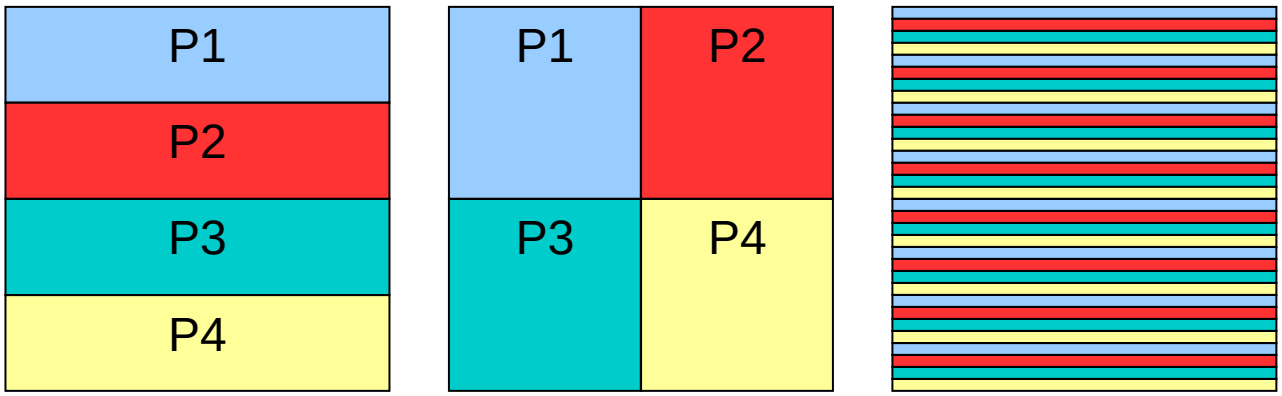


Figure 1: Static Partitioning Schemes

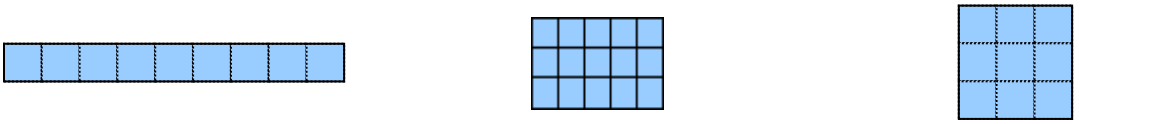
Figure 1 illustrates the three static partitioning schemes. On the left is the partitioning into contiguous blocks that span the width of the image. For this scheme, the blocks span the entire width of the image and evenly divide the height of the image.

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 queue must be used. The master process 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 and the complex .

Sequential	
mpirun –np 1	

Table 1: Sequential runtime

Static Strip Allocation				
Number of Processes	Number of Strips	Execution Time (s)	Speedup	C-to-C ratio
5 (mpirun –np 5)	5			
10 (mpirun –np 10)	10			
16 (mpirun –np 16)	16			

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
5 (mpirun –np 5)	5			
10 (mpirun –np 10)	10			
16 (mpirun –np 16)	16			

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

Static Cyclical Allocation				
Number of Processes	Height of strip in pixels	Execution Time (s)	Speedup	C-to-C ratio
16 (mpirun –np 16)	1			
16 (mpirun –np 16)	5			
16 (mpirun –np 16)	10			
16 (mpirun –np 16)	20			
16 (mpirun –np 16)	50			
16 (mpirun –np 16)	100			
16 (mpirun –np 16)	400			
10 (mpirun –np 10)	1			
10 (mpirun –np 10)	5			
10 (mpirun –np 10)	10			
10 (mpirun –np 10)	20			
10 (mpirun –np 10)	40			
10 (mpirun –np 10)	160			
10 (mpirun –np 10)	640			
5 (mpirun –np 5)	1			
5 (mpirun –np 5)	5			
5 (mpirun –np 5)	10			
5 (mpirun –np 5)	20			
5 (mpirun –np 5)	80			
5 (mpirun –np 5)	320			
5 (mpirun –np 5)	1280			

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

Dynamic Allocation				
Number of processes	Work Unit Size (rows x columns)	Execution Time (s)	Speedup	C-to-C Ratio
16, (mpirun –np 16)	1x1			
16, (mpirun –np 16)	10x10			
16, (mpirun –np 16)	30x30			
16, (mpirun –np 16)	50x50			
16, (mpirun –np 16)	100x100			

Table 5: 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, (mpirun –np 16)	11x1			
16, (mpirun –np 16)	10x1			
16, (mpirun –np 16)	7x13			
16, (mpirun –np 16)	5x29			
16, (mpirun –np 16)	11x43			
16, (mpirun –np 16)	67x1			

Table 6: The effect of work unit shape on computational efficiency as well as test for arbitrary work unit size

Report

Within your report you must:

- 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 16 processes, one should be for 10 processes, and one should be for 5 processes.
- 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, and mention which scheme you would recommend for general use.

Extra Credit

You can gain bonus points by implementing one of the following tasks. Your bonus work will be considered for grading only if you are done with the main part of the assignment.

If you do the bonus work, your program must provide a means of compiling without the bonus work and with the bonus work.

5 point bonus

For the dynamic partitioning model, implement the master using immediate sends and receives, overlapping communication and computation on the master (meaning, the master must be computing blocks as well as the slaves).

Investigate, in your report, the effect of this implementation on grain size, comparing it to the case of dynamic assignment using blocking sends and receives.

10 point bonus

Perform the work for the 5 point bonus, but also implement the slaves using immediate sends and receives, overlapping their communications with computations. To keep the slaves busy, you may need to have a job queue on each slave.

Investigate, in your report, the effect of this implementation on grain size, comparing it to the case of dynamic assignment using blocking sends and receives.

Grading

Grades are out of 100 points.

- 60 points for correct program execution. Of this 60:
  - 6 points for strip allocation
  - 6 points for block allocation
  - 6 points for cyclical allocation
  - 42 points for dynamic partitioning
- 20 points for design, implementation, coding style, and performance of your program
- 20 points for writeup and analysis
- Any additional bonus points

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 raytracer 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 four things:

- Initializes MPI by calling `MPI_Init`
- 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 two actions:

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

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, float *pixels, int width, int height)

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

- 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.
- Get started early. This project is much more programming intensive than the previous project.

- Modularize your code. Avoid having one monolithic function. This will make testing and debugging easier.
- 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.
- Develop at home. The provided code should compile on any x86 system (i386, i686, x86_64). If it doesn't, let us know!

Cluster Usage Tips

- Do not use Ctrl+Z to stop a running job. If you wish to cancel a job, use Ctrl+C. If you want to stop a job—don't. Stopped jobs take up resources that you should be letting other students use.
- Whenever you cancel a job or whenever your program crashes, run the following set of commands:

```
/opt/rocks/sbin/cluster-kill raytrace
/opt/rocks/bin/cluster-fork /opt/mpich/gnu/sbin/cleanipcs
```

  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.
- Add the following line to the bottom of your `~/.bashrc` file:

```
export P4_GLOBMEMSIZE=33554432
```

  This will ensure your program is able to allocate enough memory when transferring the very large segments necessary to send for the static partitioning models.

Rendered Scenes

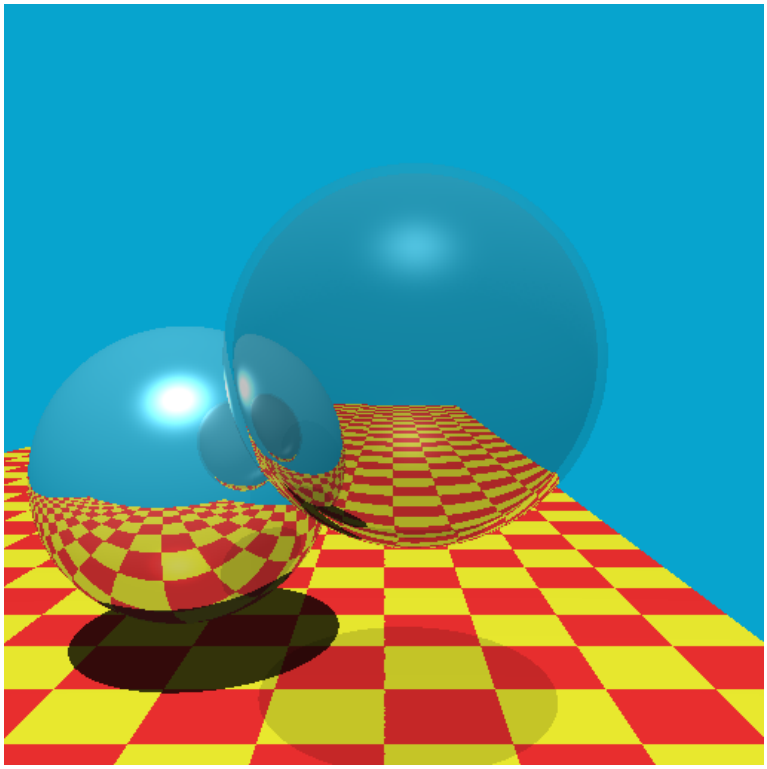


Figure 3: Turner Whitted's classic ray trace scene

Figure 4: Complex ray traced scene

Credit

The models rendered within the complex scene include the Stanford Bunny and the Stanford Dragon. These models are 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.