Introduction

In this assignment, you will be parallelizing a ray tracer using both static and dynamic partitioning. Ray tracing is considered an excellent candidate for parallelization, as each ray may be fired and computed independently of one another. There can easily be millions of pixels to render in a given image, each requiring its own ray tracing results. This means there is a significant amount of work, nearly all which may be done in parallel.

There is one notable catch with ray tracing. In ray tracing, the transport of light in a scene is calculated recursively by tracing the path of the light backwards from the camera. This recursive tracing is allows reflection and refraction to accounted for when rendering the scene. However, this recursive part of the algorithm leads to a fairly unpredictable execution time per ray, especially in scenes where there are many reflective or transparent surfaces (Figure 1, areas with spheres). Your task in this assignment is to implement each of the following partitioning schemes, determine which gives the best performance increase over a sequential version of the program, and explain why each performed as they did:

- Static partitioning, strips of rows
- Static partitioning, block
- Static partitioning, cyclical assignment of rows

Figure 1. An image obtained using ray tracing algorithm.
• Dynamic partitioning, centralized task queue

To test each of these algorithms, two scenes will be given to render. One will be a sparse scene, in which there are few geometries to interact with. The other scene will be one which is significantly denser and evenly distributed over the scene. Additionally, you will be testing to see which task grain size fares best in the dynamic partitioning implementation.

Programming environment

This assignment requires the use of an X11 Server in conjunction with the standard SSH connection to the cluster. For more information and links to download a free X server for Windows, please visit the course website SSH page. To verify that your SSH client is properly forwarding X11 data to your local X server, run the command 'glxgears' while logged into the cluster head node. If you are able to see an X window with gears rotating on the screen, your environment is properly set up to view the assignment. Depending on demand from the class and availability of the TA, an image export function may be added to allow display of output when an X server supporting OpenGL is not available.

Provided files

All files that are supplied for this assignment are located in the /export/home/756-pub/2009_files/assign2/ directory. The files provided include header files to access the ray tracing API, object files for linking your program with, and a sample sequential implementation of the ray tracer. In the sample implementation, a master MPI process performs all rendering of the scene. It is provided to show how to interact with the GLUT windowing system and provides an example of the necessary calls to have your program exit cleanly.

To build your program, first build an object file using the following command:

```
mpicc -O2 -Wall -c -o YourProgram.o YourProgram.c -I/export/home/756-pub/2009_files/assign2/include/
```

Next, build your executable by passing in the supplied object files.

```
mpicc -O2 -Wall -o YourProgram YourProgram.o /export/home/756-pub/2009_files/assign2/*.o -lglut -lGL -lGLU -lm
```

It is recommended practicing this process using the sample code provided and running the resulting executable before going further with the assignment. It will ensure that you have all the files you need for proper compilation, as well as verifying that your development environment is set up correctly. It is also highly recommended to use a build script or makefile to automate the building of your programs. Last, it may be useful to make a symbolic link (ln -s) to the assignment directory in order to cut down on the path lengths. See the ln man page for more information on symbolic links.
Ray Tracer

The 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 a particular part of the scene. You will be provided with a set of header and object files which you will use when implementing your parallel solutions. However, the program was designed to require as little knowledge about the inner workings as possible.

Your implementations will use a few functions which have already implemented for the ray tracer. A sequential version of the program will be provided to demonstrate some of the API calls that are responsible to have the ray tracer function properly. This sequential program first generates a camera and a scene using simple function calls, then fires rays at each pixel in the view window. The rayTrace() function is registered as a callback function with GLUT, and is called whenever a paint request is made by the program (such as a window resize, corrupted window, etc.). More in depth information about the ray tracing API will be provided is on the MPS course website.

MPI

When implementing your parallel renderer in MPI, there are a few things that you need to know about the windowing system that the underlying ray tracer code uses. The program is written in C and uses the OpenGL Utility Toolkit (GLUT), a simple windowing system for OpenGL programs. It achieves most of its functionality by using callback routines, requiring a different approach to MPI programming than normal.

The master process is in charge of showing a GLUT window, to show the results of the ray tracer, as well as coordinating and collecting information from the slave processes. Thus, the master process is required to implement the rayTrace() callback function. In addition, the master process is responsible for making a call to rtuiMain(), which will give control of the execution of the program to GLUT. It is important to realize that the master process will be called by GLUT when a repainting of the window is needed, requiring your program's design to be slightly different than regular MPI program.

Since GLUT takes control of the master process's execution of the program, MPI_Finalize cannot be called during the normal execution of the program. It is necessary to register a function to be called before the program exits which preforms the necessary cleanup for MPI. This can be performed by passing a cleanup function into an atexit() call. This cleanup function should contain your mechanism for terminating the slaves, along with a final MPI_Finalize call to ensure your MPI process is cleaned up appropriately.

The slave code is slightly different than the master for this assignment. The slaves do not use the GLUT functions, and instead are executed like a typical MPI program. Your slave processes will return from main normally, and thus do not need a special cleanup function to be registered at program exit (although care must be taken to ensure MPI_Finalize is still called). Both master and slaves processes will generate the scenes to render by making a call to the generateScene() function to obtain camera and world information. A corresponding call to the destroyScene() function must be made before exiting each process to properly deallocate memory.

The best way to truly grasp what each process must do is to take a look at the sample MPI implementation provided. Its structure allows for all of the mentioned constraints to be satisfied, and should be the basis of your parallel implementations.
Partitioning

You are required to implement both static and dynamic partitioning methods for this assignment. On the static side, you are to implement partitioning in strips of rows, blocks, and cyclical assignment of rows. For static allocation your master has to do computation work as well besides handling extra work related to allocation. The following figures illustrate what these partitions are (with four processes shown as an example):

![Partition Examples]

Figure 2. Examples of static work allocation for ray tracer rendering.

For the dynamic partitioning method, a centralized queue has to be implemented. Your master process will handle giving out and collecting work units (the master does not need to do any rendering in this case). It is your responsibility to decide on an appropriate method for managing work units, communicating data to the slaves and results from the slaves, handling termination, etc... Your program has to accept and handle an arbitrary size (length and width) of work unit. You are to decide what would be the best option to handle the left over from uneven division of picture dimensions by this size. However, you are required to justify your choice. In addition, you will be varying the size and shape of the work units doled out to the slaves for processing. A variety of shapes and sizes should be tested as shown next.

Report

Performance measurements should be taken at the resolution 1600x1600 for both scenes. A graph should be included in your report comparing the performance of each algorithm. For the centralized task queue implementation, the effect of different grain shapes/sizes should be discussed, with appropriate performance measurements included to support such discussion. You should be able to explain why each partitioning scheme performed as well/poorly as it did, and mention which partitioning scheme you would recommend to use for general use.

After you done with development and reach the point when your parallel program is bug-free, fill up the following tables and answer the questions.

Table 1-3 are related to static allocation.
<table>
<thead>
<tr>
<th>Number of processes</th>
<th>Number of strips</th>
<th>Execution Time, sec</th>
<th>Speedup to Sequential</th>
<th>Computation to overhead ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, sequential</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16, (mpirun -np 16)</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. The effect of work unit size on computational efficiency using strip allocation

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>Number of blocks</th>
<th>Execution Time, sec</th>
<th>Speedup to Sequential</th>
<th>Computation to overhead ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, sequential</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16, (mpirun -np 16)</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. The effect of work unit size on computational efficiency using block allocation

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>Width of a strip in px</th>
<th>Execution Time, sec</th>
<th>Speedup to Sequential</th>
<th>Computation to overhead ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>16, (mpirun -np 16)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16, (mpirun -np 16)</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16, (mpirun -np 16)</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16, (mpirun -np 16)</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16, (mpirun -np 16)</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16, (mpirun -np 16)</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, (mpirun -np 10)</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, (mpirun -np 5)</td>
<td>320</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. The effect of work unit size on computational efficiency using cyclic strip allocation

What effects from parallel implementation do you observe? Please explain why these effects are present. Compare and contrast the efficiency of each method. From Table 3 obtain the graphs of execution time: 1) 3 curves in 1 graph with number of processes as a parameter for each work unit size, 2) 3 curves in 1 graph with work unit as a parameter for each number of processes. Feel free to run additional test points if you think they may help you to detect the efficiency, explore more computational power if you have such an opportunity. Analyze the graphs, use ratio values.
Table 4-5 are related to efficiency measurement to be obtained for dynamic block allocation with arbitrary size of a work unit. Figure 3 shows 2 examples of work unit size.

<table>
<thead>
<tr>
<th>Work unit size</th>
<th>Execution Time, sec</th>
<th>Speedup to Sequential</th>
<th>Computation to overhead ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Segment (1x9 pixels)</td>
<td>1x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Block (3x3 pixels)</td>
<td>10x10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4. The effect of work unit size on computational efficiency using dynamic block allocation

<table>
<thead>
<tr>
<th>Work unit shape</th>
<th>Execution Time, sec</th>
<th>Speedup to Sequential</th>
<th>Computation to overhead ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Segment (1x9 pixels)</td>
<td>50x50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Block (3x3 pixels)</td>
<td>100x100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5. The effect of work unit shape on computational efficiency as well as test for arbitrary work unit size using dynamic block allocation

Compare the efficiency of dynamic block allocation to that of static methods. Analyze the effect of work unit size and shape on efficiency. Obtain additional test points if you think they may help supporting your arguments.
Requirements

Start working on this assignment as earlier as possible.

All time recordings should be taken on a master as well as on each slave using the function call below. Slaves are to ship their time measurement values to the master in the same packet as the data related to the computation. Master has to calculate computation to overhead ratio based on these values and similar value of his own.

\[
\begin{align*}
c1 &= \text{(double)}\text{clock}(); \\
&\quad \text{... pure computation, with no communication ...} \\
c1 &= \text{(double)}\text{clock}(); \\
t_{\text{cpu}[0]} &= \text{(double)}(c1 - c0)/(\text{CLOCKS\_PER\_SEC});
\end{align*}
\]

No more than two communication calls are allowed between the master and all slaves: one for master allocating job and another for the slave returning the data.

Your program has to be able to compile within provided source code with 2 compilation options under predefined MPI flag (defined only once): if 0 sequential, original version, if 1 parallel version. You can embed your parallel code within existing code if you wish. Below is example:

```c
#define MPI 1

#if (MPI)
  ... parallel code or part of it...
#else
  ... sequential code or part of it...
#endif
```

All your debugging messages have to be cleaned from compilation by the same method (above). No debugging message output is allowed during collecting data for the report.

You are to use master – slave paradigm in your parallel implementation, where master is responsible for job allocation. Master also prompts the user for parameters.

For static allocation your master has to do computation work as well besides handling extra work related to allocation if any

Your program has to handle an arbitrary size of work unit for dynamic allocation. You have to justify the method that you use for handling left over from uneven work division.

All partitions for static job allocation have to be of uniform size

A single program is required for all modes (static, dynamic). The program has to prompt user: 1) For the mode a user wants to run it in, 2) For additional parameters according to each mode:

- dynamic allocation - work unit size, m x n px
- static allocation – number of blocks, strips
- cyclic allocation – width of a strip in px
Your program must work on cluster.ce.rit.edu, regardless of where you develop it. For taking time measurements for all sequential and parallel tasks, you have to use the cluster.

Coding style and comments have to be at least on the same level as provided in sequential code.

Additional requirements including submission guidelines can be found on http://mps.ce.rit.edu. However, the requirements posted here supersede the ones on the web if they stipulate the same subject-matter.

**Bonus**

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

(5 %) Implement master in dynamic allocation using immediate sends and receives, overlap computation and communication on a master hand. Investigate in your report the effect of this implementation on a grain size comparing to that of in case of dynamic assignment using blocking sends and receives. Show both curves on the same graph in your report.

(10 %) Implement both master and slaves in dynamic allocation using immediate sends and receives and overlap computation and communication. In order to keep slaves busy you may consider having a job queue on each slave. Investigate in your report the effect of this implementation on a grain size comparing to that of in case of dynamic assignment using blocking sends and receives. Show both curves on the same graph in your report.

**Grading**

60% for correct program execution according to the requirements both posted here and on http://mps.ce.rit.edu

- 30 % for static partitioning
  - 10% for strips
  - 10% for blocks
  - 10% for cyclical assignment of rows
- 70% for dynamic partitioning: centralized task queue

20% for design, implementation, coding style, performance of your program.

20% for the write up and analysis.

+% for bonus part.
Additional Resources

If you are interested in learning more about ray tracing, check out the original paper on the subject, Turner Whitted's “An Improved Illumination Model for Shaded Display”. If you are interested in computer graphics in general and wish to create your own ray tracer, it is suggested taking the computer graphics sequence with the RIT Computer Science department. Additional resources may be found on this assignment's page of the MPS course website.

… and Good Luck!