CMPE 655 Assignment 2
Parallel Implementation of a Ray Tracer

Robert Relyea
Submitted: November 23, 2016

Professor: Dr. Muhammad Shaaban
TAs: Akshay Yembarwar
Mohit Sathawane

Abstract

The purpose of this assignment was to observe and compare the sequential and parallel performances of ray tracing. The ray tracing rendering technique will render an image by calculating the color for each individual pixel based on the scene being rendered. The color given to a pixel is determined by what objects are within the view and reflections of light on them. Depending on the composition of the scene, some pixels may require more work to render than others. This can lead to load imbalances in parallel implementations with static work assignment. A dynamic work assignment implementation can improve performance with an unpredictable load. In this assignment, three different static assignment algorithms were implemented to demonstrate the effect of an unpredictable load on performance. A dynamic assignment algorithm was also implemented to show the benefits of run-time load balancing for ray tracing.

Design Methodology

Ray tracing performance benefits greatly from parallel execution. Each pixel in an image can be rendered independently from one another so multiple processes can handle different parts of an image simultaneously. In order to handle which parts of the image were given to different processes, static and dynamic task assignment algorithms were designed.

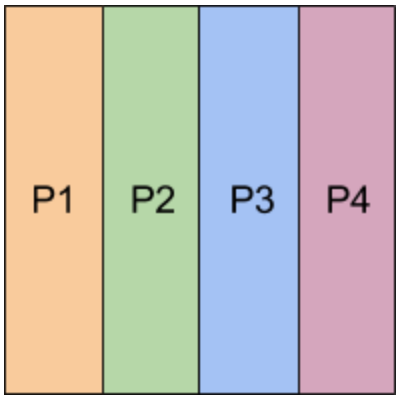

Figure 1: Static Vertical Strip Partitioning Scheme.

The first static task assignment scheme involved splitting the image into equally sized vertical strips. An illustration of this technique is shown in Figure 1, where P1, P2, P3 and P4 represent processes that have been given strips to render. The number of vertical strips was equivalent to the number of processes. The advantage with this implementation was the simplicity in the orchestration and memory management. A disadvantage was the poor load balancing inherent with the strips allocation. Strips that encompass a simpler region in the

scene will render much faster than strips over more complex regions. This resulted in processes that finished early and remained idle while others continued to work for much longer periods of time. As the number of processes increased, the size of the strips decreased. This resulted in better load balancing since the complex regions of the image were split among more processors. If the image was not evenly divided amongst the processors due to the resolution, the remaining pixels were assigned by incrementing the amount of columns each processor rendered.

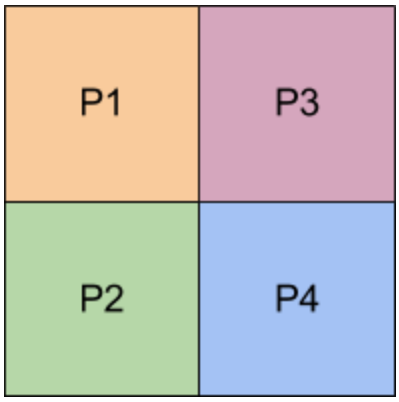

Figure 2: Static Block Partitioning Scheme.

The second static task assignment scheme involved splitting the image into equally sized square blocks. An illustration of this technique is shown in Figure 2 where an image is divided into boxes corresponding to the number of processors. The number of processors determined the dimensions of each box. If the resolution did not divide evenly by the number of boxes, the remaining pixels were incorporated into the rightmost and bottommost boxes. This was a simple way to handle the remaining pixels. Unfortunately the implementation of the static block partitioning scheme suffers from some unremedied indexing issues due to the time constraints of the assignment. In practice, the static block partitioning scheme falls short when load balancing is considered. This is due to the same reason as the static vertical strips partitioning scheme. This scheme will also benefit from an increased number of processors for the same reason as the static vertical strips scheme as well. Another constraint to consider is the number of processors that can be utilized with this method must be a square number.

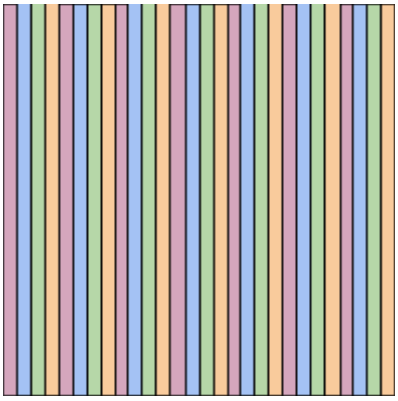

Figure 3: Static Cyclical Column Partitioning.

The final static task assignment scheme involved cyclically assigning columns of the image to processes. The implementation of this method allowed for a variable column width for testing purposes. If the image did not divide evenly into the specified column width, the extra columns would be given to the processor next in line for cyclic column assignment. Communication of the rendered columns was performed by packing all columns into one message and sending this message to the master. This allowed communication to remain constant across varying cyclical column sizes. Communication costs can be modulated with changing the process count. This method benefits from inherent load balancing depending on the cyclic columns size. If the column size is relatively small, then more complex parts of the scene would be split among more processes resulting in better load balancing. If the column size is relatively large, then the more complex parts of the scene would be split among fewer processes resulting in poor load balancing. Out of the three static task assignment schemes, static cyclical column partitioning yielded the best load balancing.

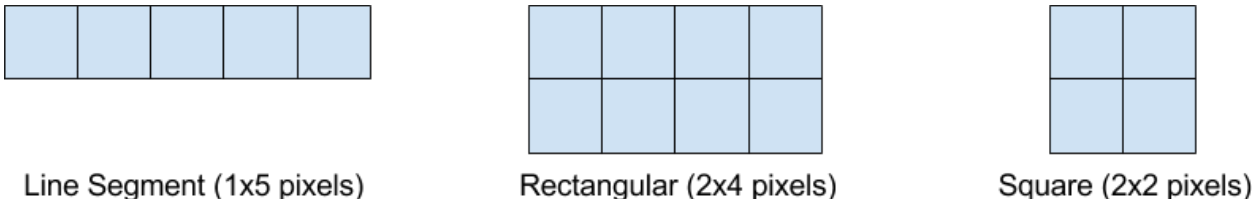

Figure 4: Examples of Work Units for Dynamic Partitioning.

The dynamic task assignment implementation consisted of a centralized task queue managed by the master process that would distribute tasks to the slave processes. This assignment technique determined the work assigned to each process during run-time to efficiently balance an unpredictable workload. The image was split into work units such as the ones in Figure 4. The work units were placed into a task queue handled by the master process.

In some scenarios the image would not evenly divide into the specified work unit dimensions. In these cases, a new work unit was generated based on the size and dimensions of the unrendered regions. These work units were then placed into the task queue for processing.During run-time, the master process will accept any messages from the slave processes and respond accordingly. When a process needs work to do, the master process will give the slave the next available work unit on the queue. When a process has finished rendering, the master will take the slave's pixels and transfer them into the final image. A process taking a long time to render one work unit would not prevent the other processes from rendering the rest of the image as the master will still distribute other work units. This technique attempted to keep each processor busy for the entire execution duration.

Results/Data Analysis

Unfortunately due to time constraints and cluster congestion every test run dictated by the assignment was not performed. All implementations other than static block partitioning would successfully render the final image. Another consequence of cluster congestion was inconsistent execution times amongst separate runs. Each ray traced scene render was at 5000 by 5000 pixels.

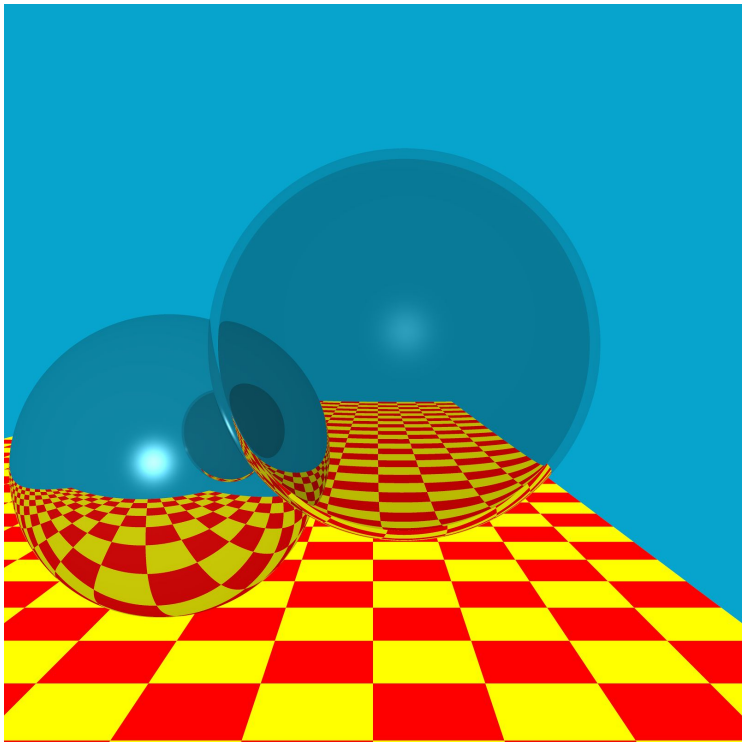

Figure 5: A Simple Ray Traced Scene by Turner Whitted.

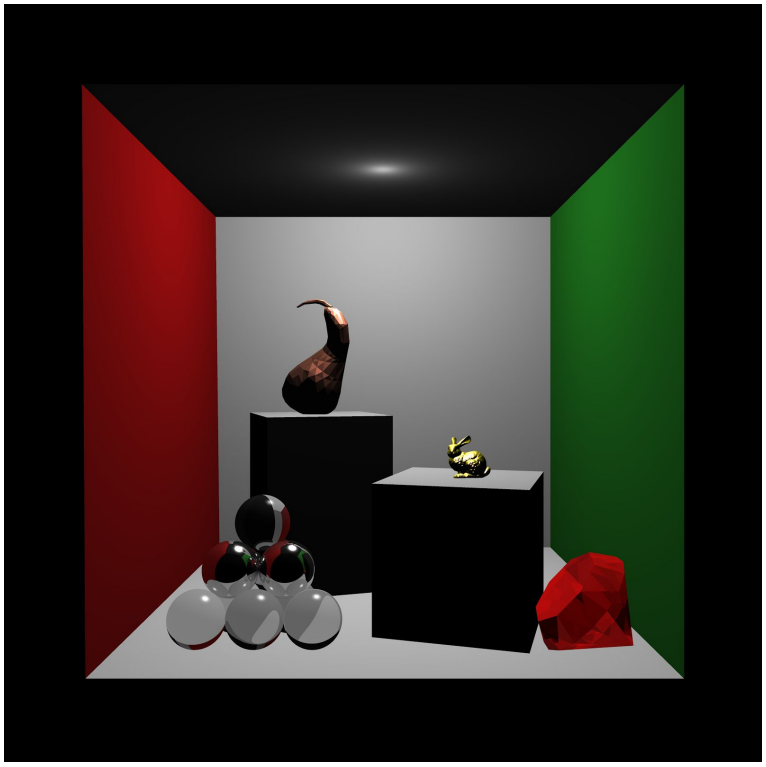

Figure 6: A More Complex Ray Traced Scene.

Figures 5 and 6 show the two different scenes that were to be rendered. Figure 5 is a considerably simpler image to ray trace in comparison to Figure 6.

Sequential Execution (Simple Image)	
raytrace_seq	179.08 Seconds

Table 1: Sequential Execution of Ray Tracing a Simple Image

Table 1 shows the execution time for a purely sequential implementation of the ray tracing program. The execution time will be a baseline for the speedup of the parallel implementations.

Static Strip Allocation (Simple Image)

Number of Processes	Number of Strips	Execution Time (s)	Speedup	C-to-C Ratio
1 (mpirun -np 1)	1	179.251	0.9990460304	N/A
2 (mpirun -np 2)	2	104.943	1.706450168	0.00399322
4 (mpirun -np 4)	4	68.9248	2.598193974	0.46021
9 (mpirun -np 9)	9	37.6875	4.751708126	0.647561
16 (mpirun -np 16)	16	22.0611	8.117455612	0.7404
20 (mpirun -np 20)	20	17.7109	10.1112874	0.752111
25 (mpirun -np 25)	25	14.5085	12.34310921	0.801097
36 (mpirun -np 36)	36	11.1402	16.07511535	0.908951
49 (mpirun -np 49)	49	8.83821	20.26202138	1.01197
55 (mpirun -np 55)	55	5.93485	30.17430938	0.809222
64 (mpirun -np 64)	64	7.35857	24.33625011	1.10502

Table 2: Parallel Execution Times for Rendering the Simple Ray Traced Scene Utilizing Static Strip Task Allocation.

Static strip partitioning renders were performed on the simple ray traced scene with a resolution of 5000 by 5000 pixels. The resulting execution times and communication to computation ratios of these runs are shown in Table 2. For these runs, the process count was varied to illustrate the difference in execution times depending on the number of strips. As more processes and strips were used, the overall execution time decreased. The decreased execution time is due to the exploited parallelism of the ray tracing problem. There is a point around 55 processors where the incurred communication costs due to increased processor count drives up the overall execution time. The communication and computation ratio increased with the processor count due to a widened scope of orchestration.

Static Cyclical Allocation (Simple)

Number of Processes	Height of Strip in Pixels	Execution Time (s)	Speedup	C-to-C Ratio
4 (mpirun -np 4)	1	48.8061	3.669213479	0.0218514
4 (mpirun -np 4)	5	48.439	3.697020995	0.0194352
4 (mpirun -np 4)	10	48.2663	3.71024918	0.0171124
4 (mpirun -np 4)	20	48.432	3.697555335	0.0175624
4 (mpirun -np 4)	80	48.7563	3.672961238	0.0165554
4 (mpirun -np 4)	320	51.0546	3.507617335	0.0860019
4 (mpirun -np 4)	640	51.5446	3.474272766	0.0123821
4 (mpirun -np 4)	1280	Unrun	Unrun	Unrun
9 (mpirun -np 9)	1	22.9025	7.819233708	0.0701139
9 (mpirun -np 9)	5	30.8502	5.804824604	0.315595
9 (mpirun -np 9)	10	40.0787	4.468208799	0.485013
9 (mpirun -np 9)	20	22.7335	7.877361603	0.0584204
9 (mpirun -np 9)	80	22.8199	7.84753658	0.0775887
9 (mpirun -np 9)	320	26.6169	6.728056235	0.0979825
9 (mpirun -np 9)	640	37.3788	4.79095102	0.557634
9 (mpirun -np 9)	1280	67.4444	2.655224155	0.461112
16 (mpirun -np 16)	1	15.1442	11.8249891	0.324379
16 (mpirun -np 16)	5	13.8052	12.97192362	0.146312
16 (mpirun -np 16)	10	13.9188	12.86605167	0.155948
16 (mpirun -np 16)	20	14.6117	12.25593189	0.203631
16 (mpirun -np 16)	80	14.4893	12.35946526	0.226238
16 (mpirun -np 16)	320	25.2117	7.103051361	0.750345
16 (mpirun -np 16)	640	36.0431	4.968496051	0.520522
16 (mpirun -np 16)	1280	67.3777	2.657852672	0.44836

Table 3: Parallel Execution Times for Rendering the Simple Ray Traced Scene Utilizing Static Cyclical Task Allocation With Varying Strip Sizes.

Static cyclical task partitioning renders were performed on the simple ray traced scene with a resolution of 5000 by 5000 pixels. The resulting execution times and communication to computation ratios of these runs are shown in Table 3. For these runs, the process count and strip height were varied to illustrate the difference in execution times depending on the number of processes and strip size. As more processes were used, the overall execution time decreased. The decreased execution time is also due to the exploited parallelism of the ray tracing problem. With cyclical strip assignment, inherent load balancing helps to drive down execution time. The effectiveness of this load balancing is directly related to the size of the cyclic strips. If the strips are large, the execution time will increase. This is due to load imbalancing as certain processes may be given strips that contain a large amount of complex imagery. With smaller strips, the complex portions of the image are divided up more evenly. The execution times in Table 3 show a decrease in overall execution time until a the strip sizes pass a certain point. At that point the execution time increases as certain processes are rendering for a longer period of time than others.

Static Cyclical Allocation

Number of Processes	Height of Strip in Pixels	Execution Time (s)	Speedup	C-to-C Ratio
1 (mpirun -np 1)	27	177.235	1.010409908	N/A
2 (mpirun -np 2)	27	89.8286	1.99357443	0.011396
4 (mpirun -np 4)	27	48.9007	3.662115266	0.0267065
9 (mpirun -np 9)	27	23.4555	7.634883076	0.106857
16 (mpirun -np 16)	27	13.6516	13.11787629	0.117269
20 (mpirun -np 20)	27	12.3192	14.53665822	0.248776
25 (mpirun -np 25)	27	9.81338	18.24855452	0.27201
36 (mpirun -np 36)	27	12.0593	14.84994983	0.517545
49 (mpirun -np 49)	27	8.45712	21.17505723	0.828353
55 (mpirun -np 55)	27	8.90381	20.11273825	0.834893
64 (mpirun -np 64)	27	6.58929	27.1774349	0.628046

Table 4: Parallel Execution Times for Rendering the Simple Ray Traced Scene Utilizing Static Cyclical Task Allocation With a Consistent Strip Size.

Table 4 shows additional rendering execution times for the static cyclical allocation scheme. In these runs, the strip size was maintained over a differing amount of processes. As the number of processes increase the overall execution times decrease. This is due to the load being balanced over more processes. The computation to communication ratio will also increase with process count due to a wider scope of orchestration.

Conclusion

Unpredictable loads can be mitigated with effective task assignment schemes. Static assignment schemes with poor inherent load balancing such as static strip or static block allocation will experience a speedup over sequential execution. A static assignment scheme with better inherent load balancing such as cyclic strip assignment will give an even better speedup than static strip or static block allocation. The best performance improvements for a problem with an unpredictable load will be seen in dynamic task allocation schemes where the work will be balanced during runtime.