Parallel Ray Tracing

Alexander Kazeka and John Stevens

December 11, 2007

1 Introduction

Ray tracing is a technique for rendering images from a three dimensional model of a scene by projecting it on to a two dimensional image plane. It works by calculating the direction of the ray that strikes each pixel of the image plane, and tracing that ray back into the scene to determine the interaction of lights and surfaces that produced that ray.

Raster graphics systems, such as OpenGL, use approximations and computational short cuts to allow rendering scenes in real time at the cost of image quality. Ray tracing, on the other hand, produces higher quality images (see Figure 1), but is too slow to be used in real time.

Since the calculations for each pixel are independent, ray tracing can be easily parallelizable. Parallelization is one way to close the gap in execution time between ray tracing and traditional raster graphics systems.

The algorithm we are implementing is not so sophisticated; it is little more than the basic naive implementation. Therefore, we expect to achieve almost perfect speedup - this is the hypothesis we will prove.

2 Ray Tracing

2.1 Algorithm

The basic ray tracing algorithm is as follows:

```
for every pixel in the image
  calculate trajectory of ray striking that pixel
  find closest intersection point of ray with scene geometry
  calculate contribution of all lights at intersection point
  recursively trace specularly reflected ray
end for
```

Finding the closest intersection of a ray with the scene is done like so:

```
for every object in the scene
  find intersection point of ray with object
```

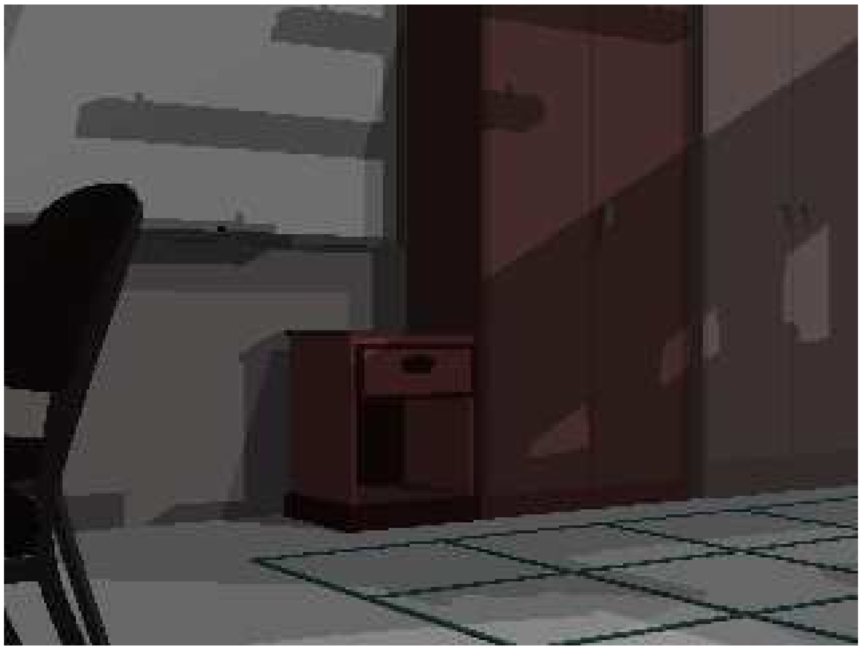

Figure 1: A ray traced image of a model of the robot area of the North CS Lab produced by our OpenMP parallel implementation. Note the shadows cast by objects.

```
  if intersection is closer to image plane than current closest intersection
    store new intersection point as closest
  end if
end for
```

The recursion of the ray tracer terminates once a certain number of recursions is reached, or once the contribution of the specular component (determined by the material properties of surfaces in the scene) is below a certain threshold.

2.2 Parallelizability

The naive ray-tracing algorithm of intersecting each ray with every surface without any optimization is pleasantly parallel. The iterations of the `for` loop in the first block of pseudo-code above is executed in parallel. All threads need only read the shared data structure, and the calculations for each pixel are independent. However, some of optimizations done on the serial ray tracing algorithm can inroduce difficulty when parallelizing.

One optimization takes advantage of similar trajectories of rays form neighboring pixels in the image. When the ray from one pixel is traced, the object it intersects and other nearby objects in the world are cached and checked first when tracing nearby rays. Since the trajectories of these rays will be similar, it is likely that they will intersect the same objects, and so this can allow scenes to be rendered much faster. The downside of this technique for parallelization is that it requires that the scene data structure be modified each time a ray is traced to include the cached information.

Other ray tracing techniques do not simply trace a single ray for each pixel. Instead, they initially trace rays at non-uniformly sampled positions on the image plane, and adaptively trace more rays in areas with high intensity variance. The idea is that scenes are not uniformly detailed: some areas of a scene are uniform in color and simple in shape, and require only a few samples to render accurately. Other areas are more detailed, and require a higher number of samples. Adaptive non-uniform sampling allows for more efficient rendering by tracing rays densely when they strike areas of high detail while saving computation time and tracing a smaller number in areas of low detail. While this results in an increase in the performance of serial ray tracers, it complicates parallelization. Each ray is no longer independent of every other, since new rays are traced based on the values returned by tracing old ones. This kind of ray tracing is not impossible to parallelize, as Notkin and Gotsman show in their paper "Parallel Progressive Ray-Tracing" [1], but it is not as simple as the algorithm described above and is beyond the scope of this project.

[1]Notkin, Irena and Gotsman, Craig. "Parallel Progressive Ray-Tracing". Computer Graphics Forum, volume 16, pp. 43-55, 1997.

3 Implementation

3.1 OpenMP

Since each thread in our ray tracer needs read-only access to a large scene data structure, out initial plan was to parallelize with OpenMP. With OpenMP, we can keep the scene data structure in shared memory, and avoid the overhead of distributing it to multiple processors. We can also avoid the overhead of combining the subsection of the scene rendered by each processor into the final image by having all threads write into different sections of the same shared image data structure. To implement this, we place a `parallel for` pragma with proper clauses outside the outer for-loop iterating over image columns while the inner for-loop iterates over the pixels in that column - this achieves larger grain size than parallelization of the inner loop.

In addition to implementation, however, there were several other issues that needed to be resolved before OpenMP parallelization was complete. First, since our ray tracer was developed using C++ and OOP and the actual ray tracing algorithm was implemented inside a class method, the program needed to be built with C++ compiler/linker that supported OpenMP directives inside class methods. Although Sun C++ version 5.5 compiler installed on CS host faure (development/initial testing workstation) had the necessary support, Sun C++ version 7.4 compiler installed on CoGrid (final experiment environment) had not. A solution was to use CS host mraz for building, testing, and final experiments. Mraz is a 8-core 64-bit Xenon-based workstation that has GCC 4.2 installed with full support for necessary OpenMP functionality.

3.2 MPI

The MPI implementation is similar to the OpenMP implementation. Columns of pixels are divided as evenly as possible among processors. Each processor does the computation for a sequential block of rows. This is not necessarily the most load balanced scheme, but it is easy to implement.

Each processor separately loads its own copy of the model file from the disk. Since this disk access is read-only, this is safe, although it may be more efficient to have a single processor read and broadcast the data.

After each processor computes its section of the image, an `MPI_Gatherv` operation is performed to aggregate the image on the first processor. The first processor then writes the image to disk.

4 Results

4.1 Overview

We did not measure the algorithm complexity based on different input sizes. This is because we do not expect the complexity to vary uniformly with the number of pixels in the image. The time to trace a ray from a single pixel is a function of the complexity of the area of the scene that that ray passes through and the number of recursions required for that pixel. Therefore, we cannot easily predict the the amount of computation required for any pixel.

We gathered data on how speedup varied with the number of processors for both our OpenMP and MPI implementations. The OpenMP experiments were run on mraz, an 8-core machine in the CS department. The MPI experiments were run on the machines in the HP classroom. We rendered the same 64 by 64 pixel image on each of one through eight processors with the OpenMP implementation, and each of one through sixteen processors with the MPI version. Ten trials of each program on each number of processors were run. To explore the effects of scheduling on program performance in OpenMP, we ran five trails each using various schedules on 8 processors. The schedules tested were `static`, `static,1`, `dynamic`, and `guided`. For the scheduling experiments, the image size was 128 by 64 to increase the total run time and make the differences among scheduling types more prominent.

4.2 Data

The speedups of both implementations are shown in Figure 2. Our speed up calculations are based on our own initially serial ray tracer and not on an optimized sequential ray tracer. We did not have time to set up and run timing tests on an efficient sequential ray tracer using the same scene file. OpenMP achieves nearly perfect speedup up to six processors, but plateaus at seven and eight processors. This is due to the residual load on the testing machine: other processes were running on two of the eight cores, allowing our program to fully utilize only six cores.

The MPI version acheves moderate speedup up to sixteen processors. One thing to note on the MPI graph are the plateaus in speedup at eight, eleven, and thirteen processors. This is due to poor load balancing. The MPI partitioning is static: each processor is allocated a section of rows to compute at the start of execution. With a problem size of 64 rows and thirteen, fourteen, or fifteen processors, eahc processor is allocated four or five rows to compute. Thus, since some processor is computing five rows for each of these problem sizes, there is no observable speedup. With sixteen processors, each processor is allocated four rows, and the speedup once again increases.

The following table shows the run times for the different OpenMP scheduling schemes on eight processors:

Schedule	Runtime
static	26.9523
static,1	26.3865
dynamic	25.73104
guided	30.89894

This corroborates our earlier assertion about algorithm complexity: since the complexity of ray tracing each pixel varies, the optimal load balancing is achieved with dynamic scheduling with the smallest chunk sizes.

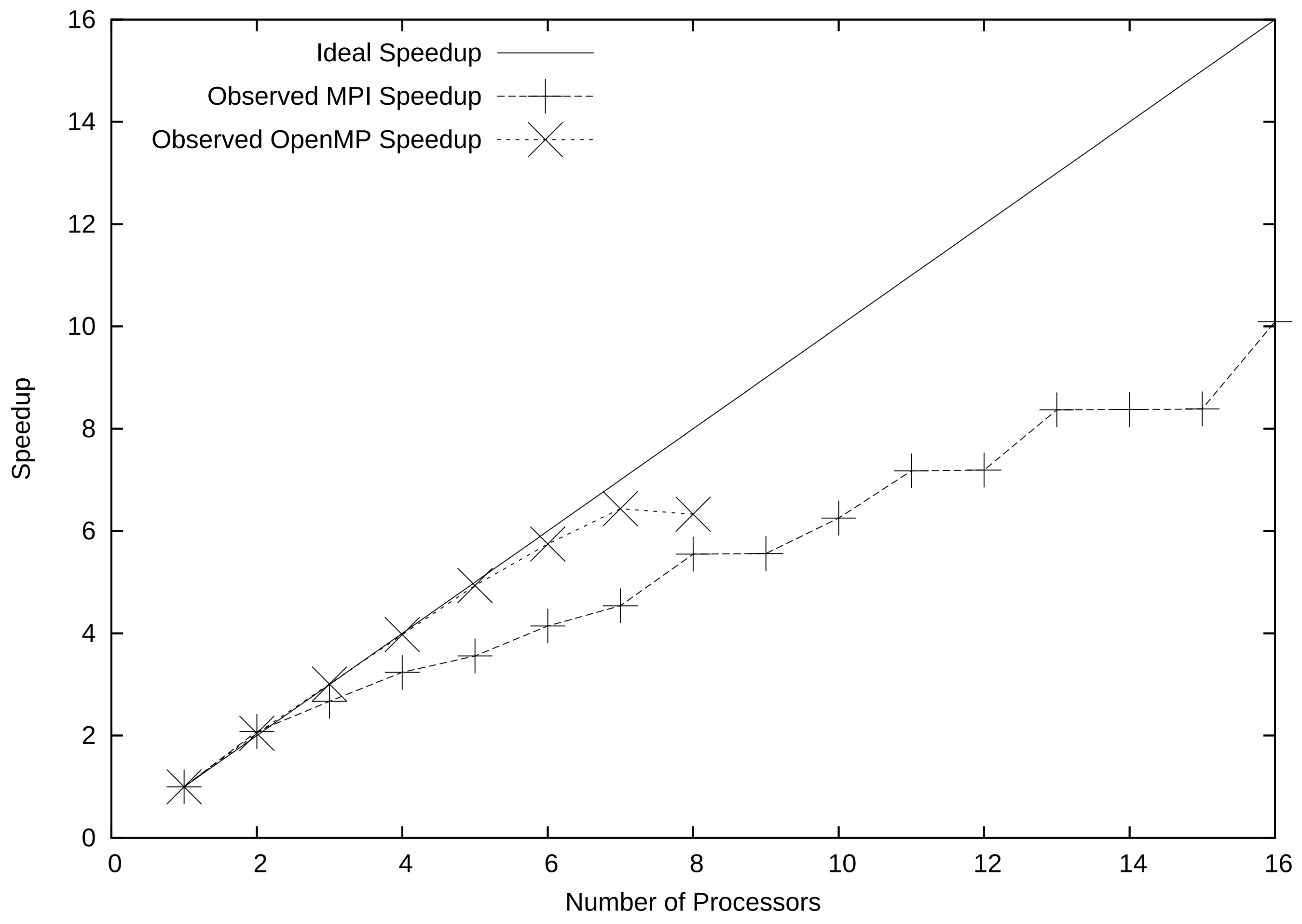

Figure 2: Speedup vs. Number of Processors for both OpenMP and MPI.

4.3 Karp-Flatt Data Analysis

Since we have experimental data, we can use the Karp-Flatt metric to estimate the fraction of the algorithm that must run sequentially.

Processors	MPI Speedup	MPI serial fraction	OMP Speedup	OMP serial fraction
2	2.081842	-0.03931230	2.043273	-0.021178276
3	2.671881	0.06140225	3.009054	-0.001504460
4	3.237206	0.07854448	3.974286	0.002156698
5	3.559631	0.10115999	4.933027	0.003394113
6	4.142629	0.08967112	5.745767	0.008849402
7	4.539544	0.09033418	6.436022	0.014604725
8	5.550360	0.06304971	6.328262	0.037738595
9	5.559570	0.07735378	N/A	N/A
10	6.252266	0.06660223	N/A	N/A
11	7.177297	0.05326104	N/A	N/A
12	7.192123	0.06077200	N/A	N/A
13	8.368685	0.04611751	N/A	N/A
14	8.374394	0.05167406	N/A	N/A
15	8.388266	0.05630087	N/A	N/A
16	10.093213	0.03901491	N/A	N/A

One thing to note in the above table is that there are negative values for the serial fraction for small numbers of processors. This is caused by noise in our data. We observed speedups of better than ideal in certain cases, which causes the serial fraction calculation to not be meaningful. Another thing to note is that the OpenMP version has a very low serial fraction up to six processors. This corroborates our hypothesis that ray tracing is a pleasantly parallel algorithm.

4.4 Comparison

Our experiments show that OpenMP is the clear winner when it comes to speedup, at least on fewer than six processors. The OpenMP implementation is also much simpler than the MPI implementation (initial compilation problems excepted): it is a single preprocessor directive placed around the outermost for loop. The advantage MPI has over OpenMP is in the hardware it runs on. OpenMP is a shared memory model of parallelization; it requires specialized hardware that allows multiple processors to access the same memory. In contrast, MPI is a distributed memory model which can be implemented on commodity hardware. Although the OpenMP implementation more efficiently utilizes available processors, the number of processors available is typically limited. MPI systems don't have such hard limits on the number of processors. This can be seen in our experiments: we ran our MPI implementation on sixteen processors (and were able to run it on more), and we achieved a speedup of approximately ten on that many processors, higher than any speedup obtained with OpenMP.

5 Conclusions and Future Work

There are many other meaningful experiments that could be run if we had time. Different partitions of the computation could be tried, which might improve the laod balancing issues we experienced with our MPI implementation. With the OpenMP version, we could try parallelizing the inner loop (over each pixel in a row instead of over each row) to achieve finer grain parallelism. Running the OpenMP on a machine with more processors (such as CoGrid) would give us a better idea of how the performance of the OpenMP implementation scales with the number of processors.

One thing lacking from our experiments is a comparison to a highly optimized sequential ray tracer. Our speed up is calculated based on the performance of the unoptimized sequential ray tracer we had readily available. There exist open source ray tracers, such as POV ray (http://www.povray.org/), that we could compare to.