Parallel Ray Tracing on the BlueGene/L

Ben Boeckel[*]   Artem Kochnev[†]   Abhishek Mukherjee[‡]   Taro Omiya[§]

Abstract

While libraries such as Nvidia's CUDA can greatly optimize the graphical application, it's pipeline structure causes inaccuracies to occur in lighting physics. As such, an efficient, accurate graphical application is required Ray-tracing can render lightings very accurately, but falls short in efficiency With a massively parallel system such as BlueGene/L (BG/L), however, ray-tracing can be rendered at a much faster speed.

Keywords: parallel, raytracing, bluegene

1   Introduction

Several attributes about ray-tracing lends itself well to networked computers such as the BG/L, regardless of whether it uses threads or Message Passing Interface (MPI). Each ray in ray-tracing acts independently from each other, and communication between each calculation occurs only in the end of tracing. Finally, ray-tracing can be rendered using the CPU, only.

Through this project, we will prove that it is possible to generate realistic graphics on a highly parallel system created originally for scientific uses. The libraries we plan to use is the MPI library for the BG/L. The results will be a PPM image file.

The program will be tested using the object models from Advanced Computer Graphics. Various measurements will be used to test the performance of our program: the number of processors, number of models and texture, number of time rays can bounce, and number of samples for shadows.

In addition, a few extensions were added to create several filters similar to those found in image editors such as Adobe Photoshop. With a little tweaking, we can give graphics a bit of an artistic taste.

2   Related Works

Several papers has already delved into the topic of ray tracing for highly parallel systems. One notable work is from Benthin and his work will ray tracing on the Cell processor, most commonly found in Sony's Playstation 3. His careful consideration in the architecture of the processor demonstrates the importance of the system's structure in calculations. [Benthin et al. 2006]

A more thorough analysis is found in Badouel's paper, where he mentions the specific structures, algorithms, and varies strategies to avoid costly data transfers on the Blue Gene L. He uses scheduling and data managing to most effectively balance each processors' workload and minimize latency. Furthermore, careful ways of avoiding common parallel problems, such as deadlocks, are addressed from this paper. This will act as the main basis for our project. [Badouel et al. 1994]

---

[*]e-mail: boeckb@rpi.edu
[†]e-mail: kochna@rpi.edu
[‡]e-mail: mukhea2@rpi.edu
[§]e-mail: omiyat@rpi.edu

3   Parallelism

Ray tracing is inherently a very parallel operation. For each pixel in the image, the algorithm traces out what will be hit by a ray coming from the camera, and then tries to figure out what color that object will be. The work for one pixel on the resulting image is entirely independent from the work done on another pixel on the screen. Therefore, it is possible to have N processors working on its own pixel and simply joining the resulting colors together to get a complete image.

3.1   Load Balancing

A couple interesting problems arise when going to multi-scalar processors. One important problem is load balancing. Properly using multiple processors cannot be done unless every processor is always doing work. Having times where processors have to wait for another processors results will cause a program's efficiency to plummet. Thus work has to be distributed in such a way where each processor is doing about the same amount of work. There are a few ways to do this for ray tracing as discussed by [Badouel et al. 1994]. These include things like preprocessing the image into chunks and distributing it to the processors. However, we felt this early preprocessing would not make sense for super-scalar architectures like the BG/L because it would just take too much time. [Benthin et al. 2006] also discusses several load balancing schemes that were specifically designed to match the Cell architecture and it's limitations in memory.

3.2   Implementation

We decided to go with an algorithm similar to the one from Baduel et al., However, instead of preprocessing the data, we simply indexed each pixel into a one dimensional array and gave processor N all pixels, i, such that

$$i \mod N \equiv 0 \tag{1}$$

The idea behind this would be that each region would be shared between the processors. Therefore if one region is difficult to compute, all processors should have some share in the region.

4   Filters

In an image editor, a filter is an algorithm that converts the pixel values to a new pixel to either remove unwanted artifacts or add an artistic taste to an image. Often, in an image, a pixel is represented by a red, green, and blue value each corresponding to additive colors in lights. Using these values, we can recalculate and compile a new image that gives a different impression from the original.

In this case, we created a gray-scale filter and a color-limiting filter that recalculates the generated pixel value from each ray in the raytracer.

4.1   Gray Scale

Each pixel in a gray scale image is represented, frequently, by a single gamma value that represents a shade of gray. Finding the

gamma from an RGB value is, incidentally, very simple. We simply have to find the weighted average of each value:

$$\gamma = W_r R + W_g G + W_b B \qquad (2)$$

Where:

$$1 = W_r + W_g + W_b \qquad (3)$$

One can easily convert this back to the RGB format by setting the red, green, and blue value equal to γ. It's worth noting that the weight values must be carefully chosen, as it represents each color's contribution to the image. For example, if we naively give equal weights to red, green, and blue, we get an overly-lit image seen in Figure 1.

To create the most acceptable color-to-gray-scale conversion, we used the weight values from [MathWorks 2009] to get the bottom image in Figure 1.

$$W_r = 0.299 \qquad (4)$$
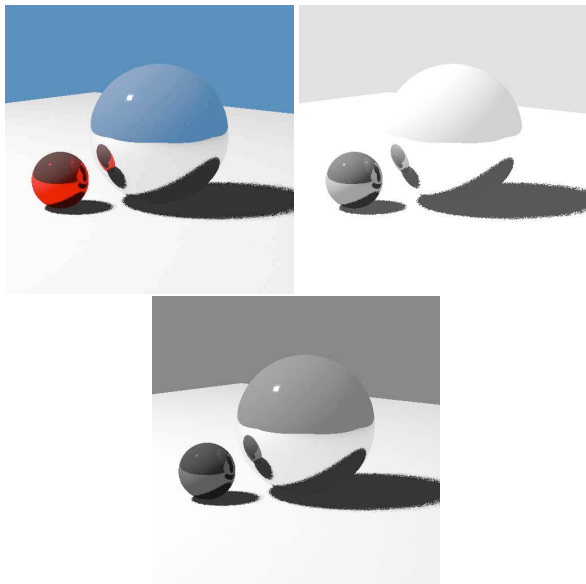$$W_g = 0.587 \qquad (5)$$
$$W_b = 0.114 \qquad (6)$$

Figure 1: *Left image: original image. Right image: overly lit gray scale image. Bottom image, fixed image*

4.2 Limiting Color

Many image editors includes an option to posterize an image, rendering a group of near-colored pixels to be shaded in one color. While our program can not detect neighboring pixels, it can generalize colors to limited shades by calculating where it fall in the spectrum of the RGB value.

Limiting the color spectrum is fairly easy. We divide the color spectrum evenly to the number of shades the user wants. If a pixel's value falls under any of the middle sections, we set it to a precomputed value corresponding to that portion. The only exception is towards the two ends, where they will be set to either minimum or maximum color value.
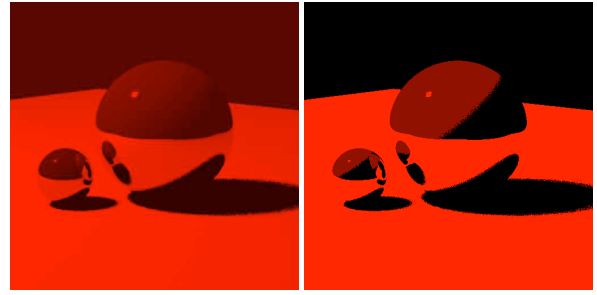

Figure 2: *Left image: full red scale image. Right image: red scale image limited to 3 shades.*

The basic algorithm can be described as follows:

Let D be the quotient of maximum color value, M, divided by n number of shades.

Let $S = M/(n-1)$.

For index i between 0 and n

If color red (r) is less than $i \times D$, set $r = S \times i$

Repeat for green (g) and blue (b).

Figure 3: *Above: red, green, and blue are limited to 3 shades, for a total of 27 colors.*

5 Conclusion

5.1 Parallelism

Our parallel algorithm worked decently well except when the computation for pixels is absurdly larger than the computation for other pixels. For example, if the number of reflective bounces is set to sixty, the computation becomes completely weighted against all the processors that hold these pixels. Specifically for the BG/L, this is an incredibly bad circumstance. The BG/L devotes all it's resources to multiple processors rather than processor speed. We encountered multiple times when it seemed like the computer entered into a state of deadlock, two or more processors are waiting for input from another processor in the cycle so no one can do work, because the processors just could not get any of the work done. It was believed to be deadlock because the same computation could run on a simple 2GHz Intel Core 2™ laptop in a reasonable amount of time. The difference was that the laptop could process a ray that bounces 100 times, while the 700MHz PowerPC processor in the BG/L could not.

5.2 Performance

A few changes in stats were compared. For example, the time it took to complete the ray tracing images in the BG/L are shown in Figure 4 and Figure 5. Figure 4 shows a typical exponential growth in time as the number of bounces calculated increases. This is expected, as more bounces increases the number of times a ray recurses.

Figure 5 is more unusual. One would expect the performance behavior in increasing the number of pixels would cause linear growth in time, since a ray is issued for each pixel. This is not the case, however. There is a sudden increase in performance when the number of pixels is increased to 2560000 units, before it continues off with its usual linear growth. We believe this burst of performance may come from the nature of how BG/L was built. Since 2560000 is a power of 2, it makes it simple for the system to make binary computations.

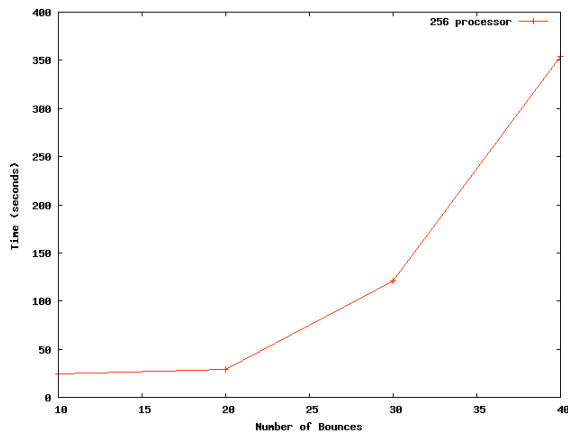For comparison, we decided to test our algorithm on a typical machine in figure 6.

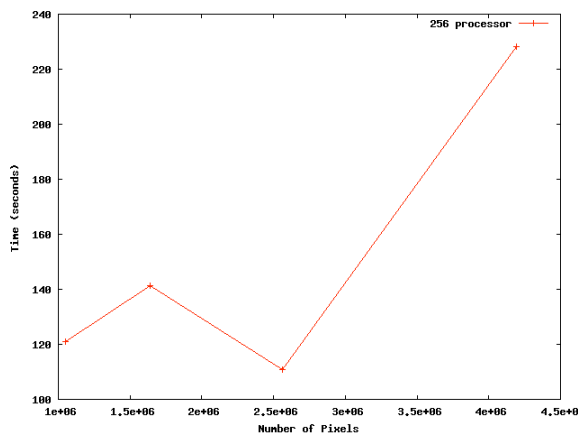**Figure 4:** *Time to render number of bounces on a BG/L. The image size used was 1024x1024.*

Figure 5: *Time to render number of pixels on a BG/L. The number of bounces was 30 and the number of shadow bounces was 100.*
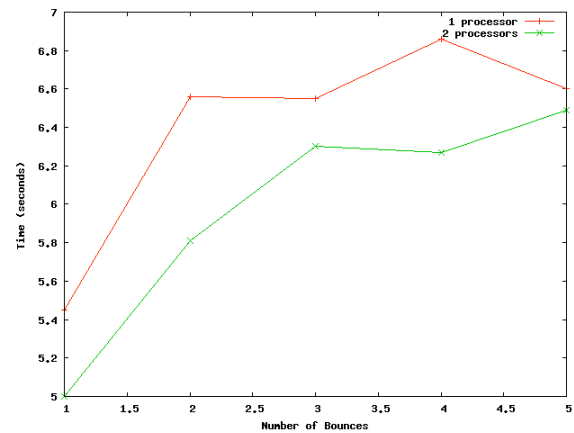

Figure 6: *Time to render number of bounces on a Intel Core 2 Duo, 1.8GHz, 2 GB RAM, customized desktop. The image size used was 200x200, and the number of shadow bounces was 10.*

5.3 Filters

On filters, we were mostly satisfied with the gray scale filtering. The limited colors also faired well to our expectations. However, no pixel value had the ability to reference from their consecutive neighbors. Since our filters had to be solely based on a single pixel value, it created some unnecessary noise from very similar colors, causing the image to look artificial. If we could devise a way to read neighboring pixel values, we could implement a more intelligent, visually pleasing filters to create artistic work. In addition, the ability to format and re-map colors based on an external file could add more expression to our limited and inflexible algorithm.

6 Citation

- **Distributing data and control for ray tracing in parallel** This paper discusses the data structures to be used to minimize data transfer in a parallel ray tracer. This will be an important consideration for us, as we begin writing our program for the BG/L. Although ray tracing is a relatively simple algorithm to implement in a parallel fashion, the only way to get proper efficiency out of the program is to manage the data, and the workload, properly. We hope this paper will relieve some of the burden of creating various parallel data structures to be used. This paper also discusses the common deadlocks that could occur in a ray tracing algorithm, which we also have to avoid.[Badouel et al. 1994]

- **Ray tracing on the cell processor** This paper discusses the special considerations that need to be taken into account for developing a ray tracer on a parallel algorithm. The architecture they were building for, the Cell, is a very different architecture from the BG/L that we will be building our algorithm on. We belove that it will form a good basis to start from in our discussions about how to create the algorithm on the BG/L.[Benthin et al. 2006]

References

BADOUEL, D., BOUATOUCH, K., AND PRIOL, T. 1994. Distributing data and control for ray tracing in parallel. *IEEE computer graphics and applications 14*, 4, 69–77.

BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. 2006. Ray tracing on the cell processor. In *IEEE Symposium on Interactive Ray Tracing 2006*, 15–23.

MATHWORKS, 2009. How do i convert my rgb image to grayscale without using the image processing toolbox? `http://www.mathworks.com/support/solutions/data/1-1ASCU.html`, Apr.