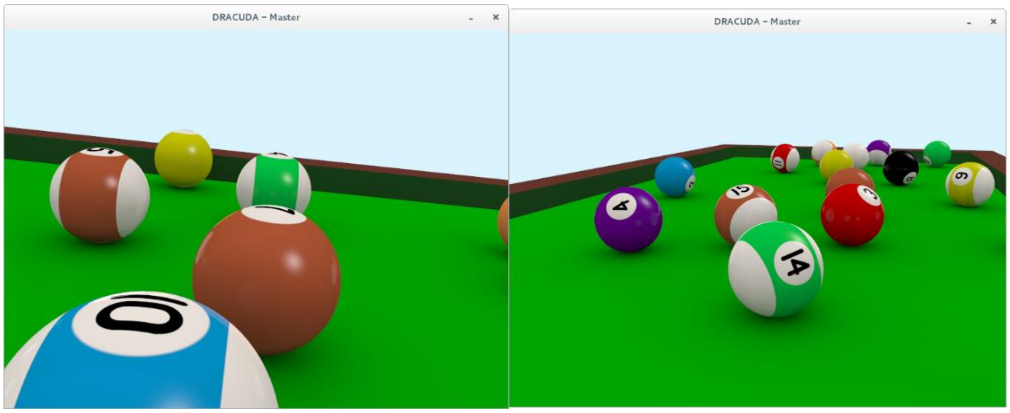
Muhammad Hilman Beyri (mbeyri), Zixu Ding (zixud)

Parallel Computer Architecture and Programming Final Project

Summary

We have developed a distributed interactive ray tracing application in OpenMP and SIMD on the CPU and in CUDA on the GPU in a heterogeneous cluster.



Background

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. As such, ray tracing is able to simulate necessary effects to create photorealistic images. However, a high-quality ray tracing can take very long time to render, and is not suitable for real-time applications.

Essential operations in ray tracing, such as intersection tests, transformations, vector math and shading are all data-parallel operations, and therefore are SIMD friendly. Making use of SIMD processors like GPU or CPU SIMD instructions can immensely improve the speed of ray tracing.

Many computers nowadays have both CPU and GPU installed. However, most of the time they are idle or poorly utilized. This presents a potential performance improvement if we can parallelize ray tracing application using both CPU and GPU.

With the advent of cloud computing, it is possible to access massive computing resources without actually purchasing them. Even GPU computing instances have become available in the past few years. Because of this, we can also parallelize over multiple connected computers to further increase performance of ray tracing application.

Different performance characteristics on CPU, GPU, and network make workload imbalance the major obstacle of parallelization. In order to balance the workload, we need to use an adaptive load balancing algorithm. This algorithm will then assign the work load of each node accordingly.

The system is implemented using master slave architecture. User directly controls the master application and master will assign work to slaves. Slave receives work from master, does the work in CPU using OpenMP or SIMD or in GPU using CUDA. Once the work finished, slave sends back the result back to master which then will stitch back together the image sent by all the slaves. Master would also need to adjust the workload to better match the slave's computing capability to better balance the workload.

Approach

We are using OpenMP and SIMD instruction for CPU, CUDA for GPU, and Boost Asio for networking. We are targeting linux machines.

Ray tracing algorithm essentially is about shooting rays from each pixel. The rays from different pixel is completely unrelated and therefore would be a perfect way to parallelize in GPU which has a very wide vector width SIMD architecture. Each pixel would be handle by one CUDA thread. Moving computation to GPU make ray tracer application goes faster than in CPU. CPU itself nowadays contains multiple cores. Computing some part of the image using multiple threads and using vector instruction in multi-core CPU would also potentially helps. Loop iteration for all the pixels could be distributed among multiple threads.

Load balancing algorithm is needed in order to match workload with a component's computation power and network latency. We model slave's response time as addition of network latency and rendering time. Based on slave's response time data that master collects every frame, we can have an idea of both slave's computation power and network latency / data transfer. Based on these data, we can then assign workload to slave as a function of network latency and rendering time.

We made a pool game as a ray tracing application. User can move around and control a white ball. The ball can roll and collide with each other. We chose to make this game because it is simple, easy to understand and let us concentrate on the parallelization, and also, of course, because we love pool game.

Master and slave needs a way to communicate. We are using Boost Asio, a networking library part of boost. Master holds multiple TCP connection with the slaves. The responsibility of master is to divide the work among the slaves and give enough information to the slaves so that the slaves know what to render. Based on that, we implemented a structure that holds information of all the balls data, camera data, and image part data that slave needs to render. Slave receives this structure, sets up the scene described structure, and render the image part that master asked. After rendering finished, slave sends back the image part, along with any useful information like rendering time. Master receives the image part from the slave, and once it has received all part of the image from other slave, it stitches all parts of the image and display it to the user. Aside from the image part, master also collects useful statistics from slave's message such as average response time, average network latency, rendering time, etc. This data then is used by master to determine the workload for next frame. Because of the way it works, it is possible that master determines no workload for a certain slave. This could be caused by long response time from the slave.

We started with existing ray tracer code from CMU's Computer Graphics class. It is using CPU and OpenMP.

Results

We tested our project mainly on GHC cluster machines and latedays. The application's screen dimension is set to 768 x 576. The application runs on linux.

CUDA Ray Tracer

We implemented CUDA Ray tracer. It supports specular, reflection, and shadow. CUDA Ray tracer achieves 174x speedup over single threaded CPU implementation on latedays.

SIMD Ray Tracer

We implemented SIMD Ray tracer. It supports the same features as in CUDA. The SIMD Ray tracer using AVX instruction achieved 2.5x speedup compared to single threaded CPU implementation. Using OpenMP to add threading boosted the performance 11.4x speedup. SIMD and CPU are tested on latedays machine.

Distributed Ray tracer

We implemented distributed ray tracer. We achieved 15-20 fps with 5 slaves using GHC41, GHC45, GHC26, GHC30, and GHC46. We achieved ~1 fps with 5 SSE SIMD slaves on the same computers. We achieved

Load Balancer

We implemented load balancer algorithms.

- Equal Division

  In the beginning of the program, for a certain number of frames, master will divide the workload equally to all nodes. This is because master does not yet know anything about the connected slaves. After certain number of frames, master then started to divide the work dynamically.

- Naïve Division

  This algorithm will simply give bigger workload to node with smaller response time. This algorithm is not stable. The response times could vary wildly based on current frame's sudden network latency hike.

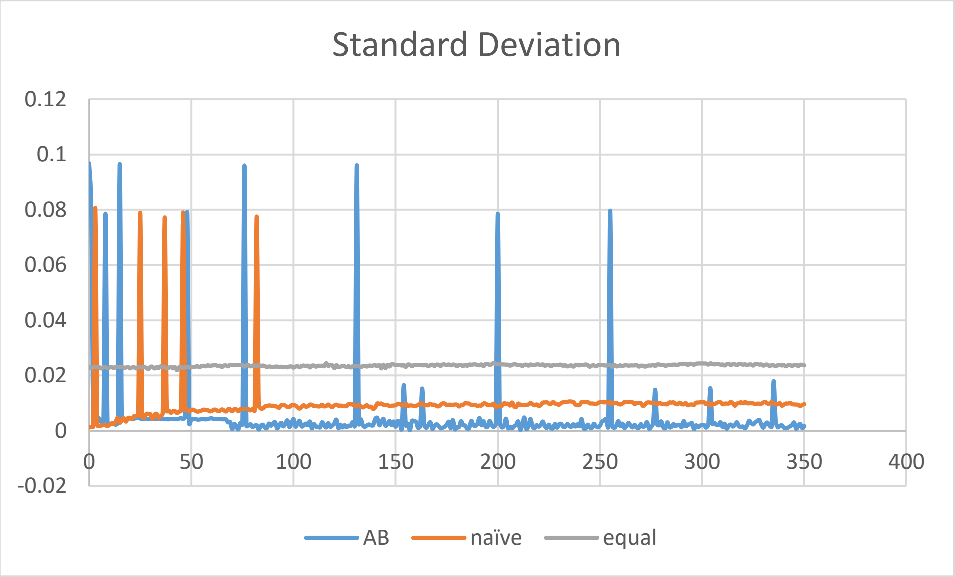
- Naïve Mean Division

  This is our earliest try of implementing load balancer. We end up not using this. This algorithm will simply give bigger workload to node with smaller response time. It achieves better response time in a heterogeneous setting compared to equal division.

- AB Division
  Instead of just using the response time to determine the workload, we model the workload as a linear function of network latency and rendering factor. This function model the relationship between workload and response time more accurately. Rendering factor is the result of dividing known rendering latency by the known workload. The network latency and rendering factor are averaged since the first frame.

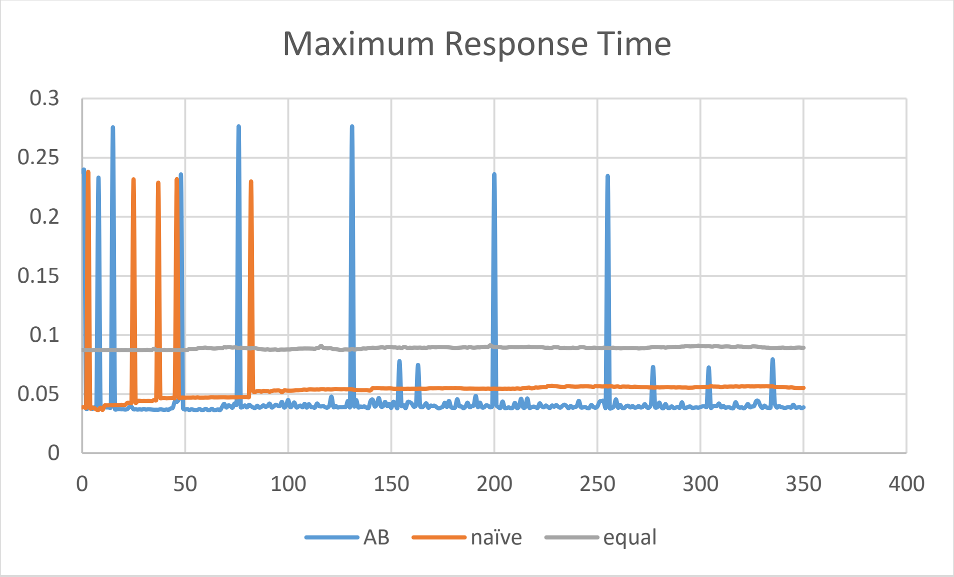
The goal of AB Division is to better reduce the standard deviation of all slave's response time. A minimal standard deviation illustrates a better workload balance which usually leads to better performance. This algorithm runs in O(n) time.

We tested these three algorithms on five slaves, GHC41, GHC26, GHC45, GHC30, and GHC46. Here is the standard deviation comparison of the three algorithms:



The standard deviation at frame i is the standard deviation of all the slave's response times in that frame. As you can see from the chart, AB division has the lowest standard deviation as expected. We speculated the network hiccup that happens on naïve and AB is caused because we were sending too fast and perhaps the network buffered our messages sometime in between.

Next, we compare the maximum response time of all three algorithms:



The maximum response time at frame i is the maximum of all slave's response times in that frame. As you can see from the chart, AB division has the lowest maximum response time as expected.

We decided not to use image compression to send the image back to master. On closer look of the response time, the network latency in average, takes about 3 millisecond. The fastest image compression, using jpeg-turbo, takes about the same time to compress.