Multi-Process Multi-Threaded Raytracer

(MuPMuTr - “Mup Mutter”)

M. J. Coppola, N. S. Shelby, A. G. Towse

Computer Science 335

Siena College

Loudonville, NY, 12211

Abstract

We made a functioning ray tracer. This raytracer used nested parallelization to drastically increase the efficiency of the program. In our implementation we include sphere objects available to be placed in the scene. These objects can be made of multiple types of materials. This project was an overall success and a great start to a ray tracing library.

Overview

Raytracing is a technique used in the film industry and recently becoming a capability of the gaming industry. This technique allows these industries to simulate realistic lighting. Raytracing, although being an incredible step forward in the graphics industry, is quite computationally demanding.

The goals for the project were to create a ray tracer that could produce a high quality image calculating aspects of real life lighting, like shadows, refractions and reflections. Apart from the raytracer, the main aspiration of this project was to create a nested parallelization. What this entailed was not only the parallelization of the light rays for each pixel in a frame, but a level above that, parallelizing the rendering of the different frames. Our results included us building a functional ray tracer with spheres, the parallelization of frame rendering with POSIX threads, and the parallelization of rendering multiple frames with OpenMPI.

During this process we encountered many issues. Some of the problems include banding with the output PPM files, phasing of the objects with the rendered video, stack smashing, and segmentation faults. We were faced with decisions on which types of parallelization to use and

some of the problems we face, and in some situations problems we faced helped us make these decisions.

What we used

POSIX Threads

POSIX Threads were used to divide the ray tracing work for a single frame. This allowed us to render a frame in a much faster time. The beauty of this parallelization is that the work was "embarrassingly parallelizable" since the work to render each pixel is entirely independent of every other pixel. This work was divided into lines of pixels, so if the frame to render was 400x400 pixels we would have 400 lines of work to do, each containing 400 pixels to compute, helping even our workload.

The rendering is effectively the same as the serial version, the only difference is that each thread only renders a portion of the frame and stores it into the frame buffer to be output later. In a later version of this project, we may implement a different division of work - which will be discussed later. The downside to the lines of work being divided amongst the processes in blocks is that some of the lines are easier to compute than others and this means that if a process were to receive a block to render that was easy to compute, it may be done well before other processes.

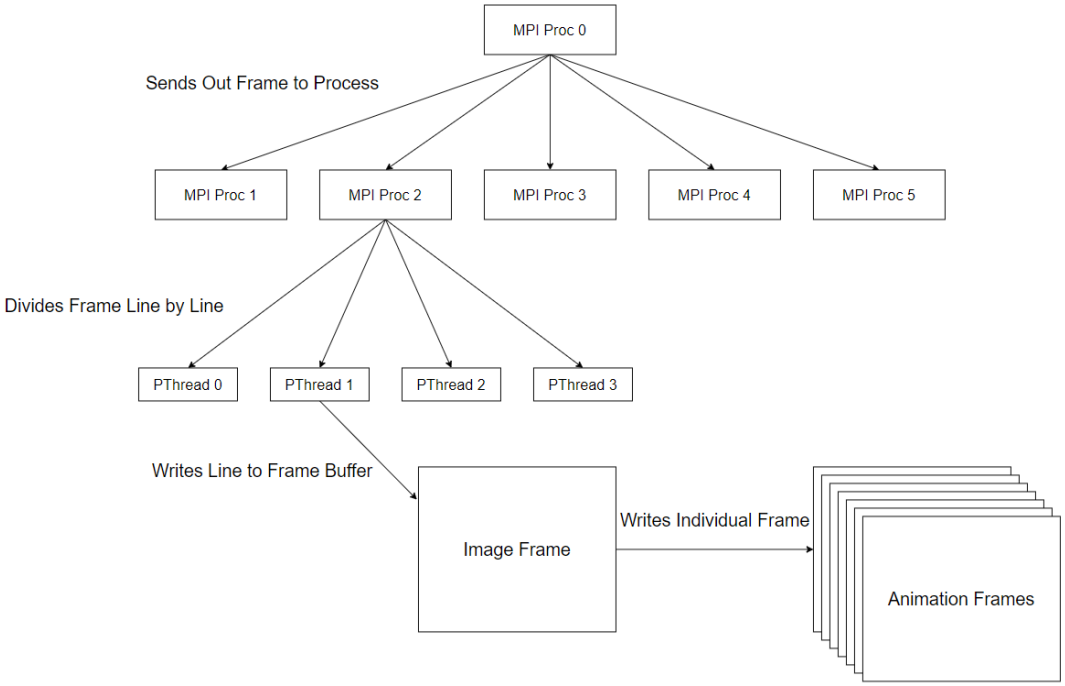
Open MPI

Once the work was well divided amongst a single rendering frame, we had to parallelize the work again in a different spot. At this point, we fell into using Open MPI for message passing as we had experience in using it and it lent itself to the type of work we wanted to do. This was not just an exploration of using two different types of parallelization in a single program, but also as a way to implement a more realistic approach to this type of work. If this work was as computationally intensive as it can get, being able to separate it onto different nodes or processors is essentially required.

The separation of work was nearly "embarrassingly parallelizable" as the division of frame rendering and the division of frames in an animation were effectively entirely separate. The MPI processes would separate and the rank zero process would hand out work, using a bag

of tasks approach to work distribution. As the worker processes received the frame that they were to render - an integer as the animation was simple and required only a transformation of the spheres' positions based on the time in the animation - they then divided up their work amongst the POSIX Threads. Once these frames were fully rendered, they were written to the PPM file, just like the serial implementation, except they then signaled back to the master thread that they were ready for more work. If there was more work, they would get it, otherwise there would be a signal that there was no more work to hand out and that it was time to begin wrapping up.

The fact that calculations weren't shared across processes and there was no need to send anything back to the master thread made the MPI programming quite simple and allowed for a very smooth combination of the MPI processes and POSIX Threads. At this point, we had successfully created a project that used both message passing and threads to divide work.



Notes:

- Discuss what we used for the project

Steps we took

1. Started by making a single frame renderer
    a. Go through each step of added ray tracing showing different output of different phases.
    b. Output to ppm
2. Made script compile multiple frames into video
3. Works towards parallelizing using pthreads

- Problems faced
    - Rendering triangles for meshes
    - Using openmp
        - Faded out glitch with openmp kinda cool
        - Banding problem
    - Aspect ratio bug
    - Allowed for sphere_ray_interesct to find intersections Unusual ray positions
    - Problems with creating checkerboard
- Liberries and sources used
    - CGLM vector math library
    - Different pages on ray tracing math
- What we learned
    - Vector math
    - Different ways of parallelizing this problem
    - Learned about different application of using multi pp techniques
        - Using pthreads for individual frames
        - Using MPI for splitting up frames

FUTURE RESEARCH

- Better data structures for our world projects
- KD trees and how they could help
- Chunking the frames, not lines and such

CONCLUSION

CITATIONS