

COMSW4995 Parallel Functional Programming:

Final Project Proposal - Parallel Ray Tracer in Haskell

Team: Mst Nayma (mzn2002), Becky Dinh (bd2818)

1. Overview

We will implement a ray tracer that renders a 3D scene of spheres and planes to an image. For each pixel, a ray is cast from the camera into the scene, the nearest intersection is found, and the pixel is shaded using the surface normal and light position. Since each pixel can be calculated independently, ray tracing provides us a natural way for parallel execution. We plan to divide the rendering workload across multiple CPU cores. We plan on first developing a simple sequential implementation, optimizing it, then parallelizing it and finally measuring speedup from 1 to 8 cores. During this process, we will explore different approaches to distributing the work, such as processing rows or smaller sections of the image concurrently, and then evaluating how these approaches affect overall performance.

Our main objective is to achieve meaningful speedup while still keeping the implementation efficient and manageable.

2. Algorithm

We will implement a ray tracing algorithm that generates a 2D image from a 3D scene by tracing rays from a virtual camera and then calculating their interactions with objects and light sources. The algorithm will consist of the following high level steps:

- Camera generates one ray per pixel (later: several per pixel for anti-aliasing).
- Ray-sphere and ray-plane intersection; keep the closest hit.
- Shading: The color at each intersection is calculated using Lambertian diffuse plus a specular term (Phong). Shadow rays toward each light.
- Reflection rays for mirrored surfaces, limited to a fixed bounce depth.
- The image is written to a PPM file (no external image libraries)

3. Sequential implementation and optimization

- Initial Implementation
 - First version with plain data types and lists. This is a baseline sequential ray tracer using standard Haskell features.
- Memory and Performance Optimization:
 - Improve computation by introducing strict, unpacked `Vec3` fields; unboxed vectors for the pixel buffer; avoid laziness leaks in the per-pixel loop.
 - Measure runtime, memory allocation, and garbage collection overhead using `+RTS -s`.
 - Use additional profiling as needed to identify expensive operations.

4. Parallelization

We plan to parallelize the optimized sequential implementation using Haskell's `Control.Parallel.Strategies`. Each pixel's color is calculated independently of the other pixels, so parallelization mainly changes when and where the rendering work runs, rather than how the colors are calculated.

- Divide the image into groups of rows or rectangular tiles that can be rendered in parallel.
- Compare chunk sizes and row vs tile decomposition.
 - Some pixels may take longer to render than others, some experimentation will need to be done to keep cores busy without creating too much scheduling overhead.
- Use ThreadScope and runtime statistics to analyze spark conversion and core utilization.
 - *Will need to adjust accordingly to further improve performance*
- Possibly a second level of parallelism inside expensive pixels (multiple samples per pixel) to study granularity.
- Assemble results in the correct pixel order to render the full image

5. Input Data

The input to the ray tracer is a scene description of the 3D scene that our program will render into an image. (We're essentially giving the program instructions about what objects exist, where they are, etc.)

- Generate 3D scenes with spheres and planes with their positions, sizes, and colors.
- Vary the number of objects, placement of objects, and image resolution to test different workloads.
 - Image resolution and samples per pixel will be command-line arguments.
- We will define a few small test scenes directly in Haskell.
 - For benchmarking, we will generate larger scenes programmatically (for example, N spheres placed on a grid or at random positions with a fixed seed) so we can control the number of objects and measure how performance scales.
- No external datasets are required.

6. Evaluation

- Test scenes of increasing complexity (varying number of objects, reflection depth, samples per pixel).
- Measure runtime on 1, 2, 4, 8 cores; speedup curve; estimated parallel fraction via Amdahl's law.
- Identify performance bottlenecks such as scene setup, image output, garbage collection, and parallel overhead.
- Verify that sequential and parallel versions produce the same output.

7. Testing

- Unit tests for vector math and intersection routines (known hit/miss cases).
- Rendered output compared against a reference image to confirm the parallel version is pixel-identical to the sequential one.
- Test edge cases such as rays missing all objects, overlapping objects etc
- Test different numbers of objects and image resolutions settings to make sure results are consistent
- Run the parallel version on 1,2,4, and 8 cores to check for correctness and stability

8. Timeline

- Week 1: Implement the sequential tracer with spheres, planes, diffuse shading, PPM output.
- Week 2: Add shadows, reflections; sequential optimization and profiling.
- Week 3: Parallelize the ray tracer, experiment with different chunk sizes, ThreadScope to analyze performance
- Week 4: Run benchmarks measurement, report, optional features / stretch goals (BVH, anti-aliasing) only if the core is solid.

9. References

- Marlow, *Parallel and Concurrent Programming in Haskell*
- Shirley, *Ray Tracing in One Weekend*
- NVIDIA. *Ray Tracing*. <https://developer.nvidia.com/discover/ray-tracing>
- Scratchapixel. Implementing the Raytracing Algorithm.
<https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/implementing-the-raytracing-algorithm.html>
- The pfp-nqueens example repository