CS 424 : Introduction to High Performance Computing
Final Project Writeup

Luke Marzen

December 13, 2022

Abstract

Ray tracing is a method of rendering that uses ray casting to model light physics. Since each pixel can be calculated by a ray cast independently of any other ray cast, this is an ideal problem for parallelization. This paper explores two different techniques for implementing the parallelization of a ray tracing engine. The first technique focuses on minimal overhead, evenly dividing the pixels across the available resources. The second technique dynamically assigns work with the goal of higher utilization. Each technique is implemented with OpenMP and MPI. The results show that the dynamic work assignment approach with more overhead can scale more optimally than the statically assigned approach.

Implementation

The ray-tracing engine is written in C and uses C standard libraries as well as OpenMP or MPI for parallelized versions. The scene is defined by the coordinates of light sources and spheres. Each surface is assigned a material that will determine how light will interact with the object. The output of the program is a single .ppm bitmap file depicting the rendered scene. The scene is decided before the program is compiled, while the resolution is defined by width and height, which are read in as command-line arguments.

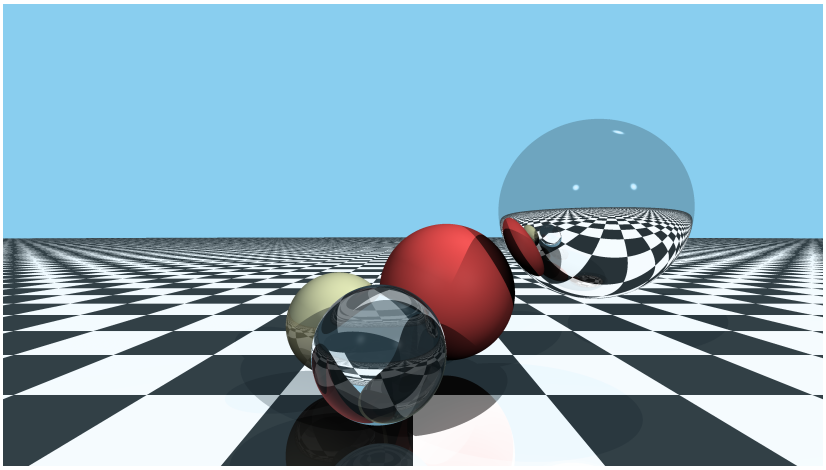
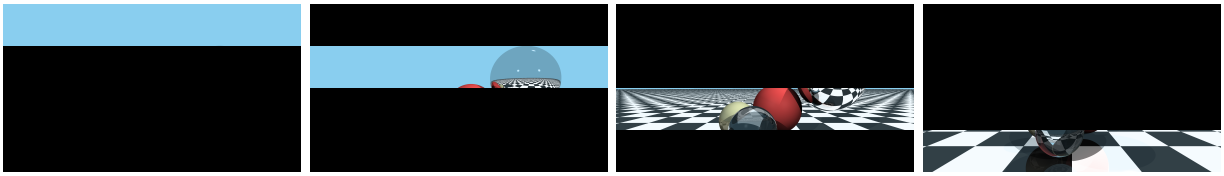


Figure 1: Rendered output.

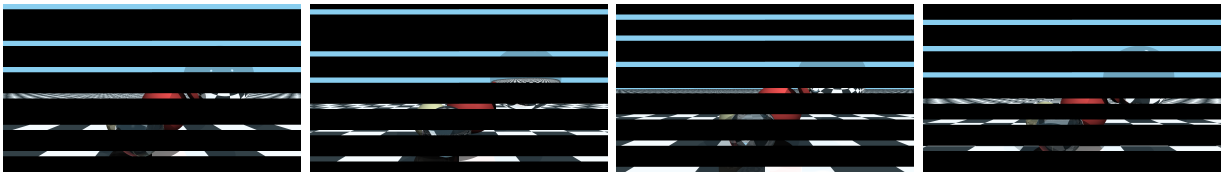
This program was first implemented as a sequential program, then modified to exploit parallelism using two different methods for both OpenMP and MPI, for a total of 5 different implementations.
Two methods of parallelization:

1. Static-assignment. Evenly divide the pixels into N buckets. For simplicity, let N be the number of available threads. Then, each thread is assigned one of the N buckets of pixels. This implementation has the advantage of simplicity and relatively little overhead. However, this technique can only be optimal if every bucket of pixels takes exactly the same time to render. When using ray trace-based rendering, likely, the scene is not of uniform complexity. The sky or ground will be quick to render with minimal or no reflections, while other areas containing complex geometry and reflective surfaces will take much longer to render. Variance in scene complexity may result in a few highly-utilized threads while other threads finish early and sit idle.



Figures 2 through 5. Static-assignment, 4 slices rendered across 4 threads.

2. Dynamic-assignment. Evenly divide the pixels into N buckets, where N is some integer significantly greater than the number of available threads. Begin by assigning each thread one bucket. Whenever a thread finishes rendering a bucket of pixels, assign it another bucket. Repeat this until all buckets have been rendered. This approach has slightly more overhead and is less trivial to implement, but will ensure all threads are highly utilized when presented with scenes of non-uniform complexity.



Figures 6 through 9. Dynamic-assignment, 32 slices rendered across 4 threads.

The OpenMP implementation of the static-assignment method is trivial, using a single parallel for directive.

```
#pragma omp parallel for num_threads(thread_count)
for (unsigned int pixel = 0; pixel < pixel_count; ++pixel) {
  ... // calculate ray origin and direction
  framebuffer[pixel] = cast_ray(origin, dir);
}
```

The MPI implementation of the same method is only slightly more involved, requiring some additional logic for each processor to determine which set of pixels it is responsible for rendering as well as communication logic to sync all the rendered results to the root processor.

```
for (unsigned int pixel = local_start; pixel <  local_end; ++pixel) {
  ... // calculate ray origin and direction
  local_buf[local_index] = cast_ray(origin, dir);
  ++local_index;
}
// sync results to root thread
if (my_rank != 0) {
  MPI_Send(local_buf, local_buf_sz, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
} else {
  memcpy(framebuffer, local_buf, local_buf_sz);
  for (unsigned int i = 1; i < comm_sz; ++i) {
    MPI_Recv(local_buf, local_buf_sz, MPI_CHAR, i, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    memcpy(framebuffer + i * local_buf_sz, local_buf, local_buf_sz);
  }
}
```

Implementation of the dynamic-assignment methods is significantly different between OpenMP and MPI. Since OpenMP uses shared memory, a shared counter can be used to indicate the next bucket to be rendered. The shared counter must be in a critical block to lock out other threads while it is being read/updated.

```
#pragma omp parallel num_threads(thread_count)
{
  unsigned int my_region = omp_get_thread_num();
  while (my_region < num_regions) {
    for (unsigned int pixel = my_region * pixels_per_region;
         pixel < (my_region + 1) * pixels_per_region; ++pixel) {
      ... // calculate ray origin and direction
      framebuffer[pixel] = cast_ray(origin, dir);
    }
    #pragma omp critical
    {
      ++current_region;
      my_region = current_region;
    }
  } // end while
} // end #pragma omp parallel num_threads(thread_count)
```

In the MPI implementation, the root thread will be reserved as a central communicator, keeping track of which buckets each thread has rendered and assigning a new bucket of pixels to each thread once it has completed its previous assignment. Once all buckets have been rendered, the root thread must retrieve each bucket of rendered pixels from the thread that was assigned to render it.

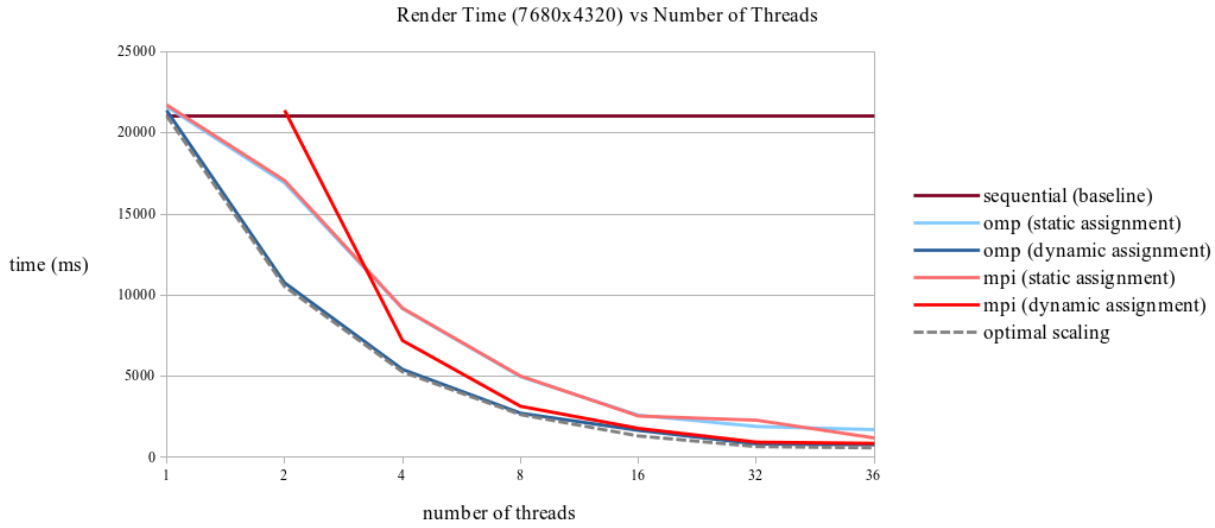
```
if (my_rank != 0) {
```

```
  unsigned int my_region = my_rank - 1;
  while (my_region < num_regions) {
    for (unsigned int pixel = my_region * pixels_per_region;
         pixel < (my_region + 1) * pixels_per_region; ++pixel) {
      ... // calculate ray origin and direction
      framebuffer[pixel] = cast_ray(origin, dir);
    }
    MPI_Send(&my_rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&my_region, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
} else { // root thread is central communicator and will assign work
  unsigned int next_region;
  unsigned int recv_rank = 0;
  for (next_region = 0; next_region < comm_sz - 1; ++next_region) {
    bucket_map[next_region] = next_region + 1; // initial work is predetermined
  }
  while (next_region < num_regions + comm_sz - 1) {
    MPI_Recv(&recv_rank, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
    MPI_Send(&next_region, 1, MPI_INT, recv_rank, 0, MPI_COMM_WORLD);
    bucket_map[next_region] = recv_rank;
    ++next_region;
  }
}
// sync results to root thread, UINT_MAX from root indicates all done.
if (my_rank != 0) {
  unsigned int bucket_index = 0;
  MPI_Recv(&bucket_index, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  while (bucket_index != UINT_MAX) {
    MPI_Send(framebuffer + bucket_index, bucket_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&bucket_index, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
} else {
  for (unsigned int i = 0; i < num_regions; ++i) {
    MPI_Send(&bucket_index, 1, MPI_INT, bucket_map[i], 0, MPI_COMM_WORLD);
    MPI_Recv(framebuffer + bucket_index, bucket_size, MPI_CHAR, bucket_map[i], 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    bucket_index += bucket_size;
  }
  for (unsigned int i = 1; i < comm_sz; ++i) {
    bucket_index = UINT_MAX;
    MPI_Send(&bucket_index, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
```

```
}
```

Results

Each version of the rendering engine was benchmarked on a single node of Iowa State University's Nova HPC cluster. The node used was equipped with two 18-Core Intel Skylake 6140 Xeon processors. Each version of the program was ran using 1, 2, 5, 8, 16, 32, and 36 threads, with the exception of the dynamic assignment MPI version, which needs a minimum of 2 threads. The output (shown in Figure 1) was rendered at a resolution of 7680x4320(8k), and the dynamic assignment versions of the program used 512 buckets.



Figures 10. Line plot of render time for an 8k(7680x4320) output vs number of threads.

All parallelized versions of the rendering engine showed scaling, see Figure 10. The static assignment OpenMP and MPI versions performed nearly identically across all samples, with the OpenMP version being only 1.3% faster on average. As expected, the dynamic assignment versions of the program showed more optimal scaling than the static assignment versions, with the OpenMP version showing near-optimal scaling. The MPI dynamic assignment version requires one thread to be the central communicator instead of performing work and hence shows poor scaling when only using a few threads, but with more threads scaling becomes more optimal.