GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Parallel Ray Tracing

Pranay Bhatia, Karan Sharma

MatrNr: 17935037, 21032486

Supervisor: Zoya Masih

Practical course in High-Performance Computing,
Georg-August-Universität Göttingen
Institute of Computer Science

September 29, 2024

Abstract

Ray tracing is a powerful rendering technique capable of producing photorealistic images by simulating the complex interactions of light with objects in a scene. Despite its visual fidelity, traditional ray tracing methods are computationally expensive, making real-time applications like video games and simulations challenging. This report explores the implementation of parallel ray tracing as a means to overcome these limitations, focusing on techniques that reduce rendering time while maintaining high image quality.

We begin by discussing the foundational concepts of ray tracing, including ray casting, path tracing, and acceleration structures like Bounding Volume Hierarchies (BVH). The report also provides Python code for the parallel ray tracing implementation, which is optimized to leverage task and data independence for significant performance improvements. Additionally, different types of materials, such as Lambertian, metallic, and dielectric, are explored to understand their impact on rendering.

To evaluate the performance gains from parallelization, we compare serial and parallel implementations using 1, 4, and 8 CPU cores. The results highlight substantial reductions in rendering times without compromising image quality. Furthermore, an additional experiment is conducted to analyze sphere intersection using geometric methods, providing deeper insights into the computational efficiency of ray tracing algorithms.

Benchmarking results confirm the effectiveness of parallelization, demonstrating its potential for real-time applications. The report concludes with a discussion on the implication

Contents

List of Figures **iii**

1 Introduction **1**
1.1 Problem Statement 2

2 Background **4**
2.1 Components of a Ray Tracing Scene 4
2.2 Algorithm: The Pseudo Code Behind Ray Tracing 5
2.3 The Scene Setup 6
2.4 Ray Intersection & Definition 7
2.5 Sphere Intersection 7
2.6 Blinn Phong Reflection 8
2.7 Sphere Intersection: Geometric Considerations 8

3 Parallelization with MPI **9**
3.1 MPI Setup and Process Distribution 9
3.2 Handling Variable-Sized Image Segments to Prevent Tiling 10
3.3 Performance Benefits of Parallelization 10
3.4 Setting Up the Environment and Running the Script 11

4 Results **12**
4.1 Scene Complexity Results 12
4.2 Light Source Position Results 13
4.3 Performance Results Based on Scene Complexity 13
4.3.1 Observations 15

5 Conclusion **16**

List of Figures

1 different ways of interaction of light: [Joh14] 2
2 Observation: independence of task 3
3 Components of ray tracing scene source: [Afl21] 5
4 Scene setup source: [Afl21] 6
5 Sphere Discriminant 8
6 Low Complexity Scene (2 spheres) 12
7 Medium Complexity Scene (4 spheres) 12
8 High Complexity Scene (7 spheres) 12
9 Light Source on the Left 13
10 Light Source on the Right 13
11 Light Source on the Top 13
12 Execution Time vs. Number of Processes (Low Complexity) 14
13 Execution Time vs. Number of Processes (Medium Complexity) 14
14 Execution Time vs. Number of Processes (High Complexity) 15

Listings

1 sudo_code 6
2 MPI Initialization and Row Distribution 9
3 Handling Variable-Sized Image Segments to Prevent Tiling 10

1 Introduction

Ray tracing is a rendering technique that fundamentally relies on the physics of light to produce images with a high degree of realism. The basic premise of ray tracing involves tracing the paths of light rays as they travel through a scene, interacting with objects and materials. The ultimate goal of ray tracing is to produce images that can be indistinguishable from those captured by a camera, even in animated films. This technique has become a cornerstone in modern visual effects, utilized extensively in both non-real-time applications such as film and television, and real-time applications like video games.

When you observe the world around you, the objects you see are illuminated by beams of light. Ray tracing works by following the path of these beams backward, from your eye to the objects that the light interacts with. This approach allows for the simulation of various optical phenomena, such as reflections, refractions, and shadows, which are essential for creating photorealistic images. [Nvi18] These effects are critical in modern movies, where ray tracing is used to generate or enhance special effects, creating lifelike reflections on surfaces, accurate refractions through transparent materials, and realistic shadowing in complex scenes.

Ray tracing is not just limited to visual effects in media. It is also employed in various other fields such as lighting design, architecture, and engineering. In these areas, ray tracing helps professionals visualize how light interacts with different materials and environments, enabling them to design spaces and objects that are both aesthetically pleasing and functionally effective.

Definition and Purpose

As defined by Brian Caulfield from Nvidia,

"*Ray tracing is a rendering technique that can realistically simulate the lighting of a scene and its objects by rendering physically accurate reflections, refractions, shadows, and indirect lighting.*" [Nvi18]

This definition encapsulates the core purpose of ray tracing: to simulate the behavior of light in a way that is true to the physical world. By calculating how light rays interact with objects, ray tracing can produce images that are visually indistinguishable from reality.

Applications

Ray tracing's ability to simulate complex light interactions makes it invaluable in various applications:

- **Non-real-time applications:** In film and television, ray tracing is used to create high-quality visual effects that enhance the realism of scenes. This includes everything from the subtle reflections on a wet street to the dramatic lighting of an alien spaceship. [Vas+18]

- **Real-time applications:** Video games use ray tracing to enhance the visual fidelity of virtual environments, making them more immersive for players. Recent advancements in hardware have made real-time ray tracing feasible, bringing cinematic quality graphics to interactive media. [Nvi18]

- **Lighting design:** Professionals in the field of lighting design use ray tracing to simulate how light sources will interact with a space. This helps in optimizing lighting setups for both aesthetics and energy efficiency. [KK08]
- **Architecture and engineering:** In these fields, ray tracing is used to visualize the effects of light on building designs and materials, aiding in the creation of structures that are both visually striking and functionally sound.

In summary, ray tracing is a versatile and powerful tool that has transformed how we create and interact with visual content. Its ability to simulate realistic light interactions makes it a critical technology in fields ranging from entertainment to engineering. As we continue to push the boundaries of what is possible in computer graphics, ray tracing will undoubtedly play a central role in shaping the future of digital imagery.

1.1 Problem Statement

While ray tracing offers unparalleled realism in rendering images, it comes with significant computational challenges. The technique's complexity arises from its fundamental approach of simulating the interactions of light rays with objects in a scene. In the real world, a light source emits an infinite number of rays, each of which can interact with various surfaces, materials, and other rays in countless ways. [Jun+21] Accurately modeling these interactions is computationally expensive, making ray tracing a resource-intensive process.

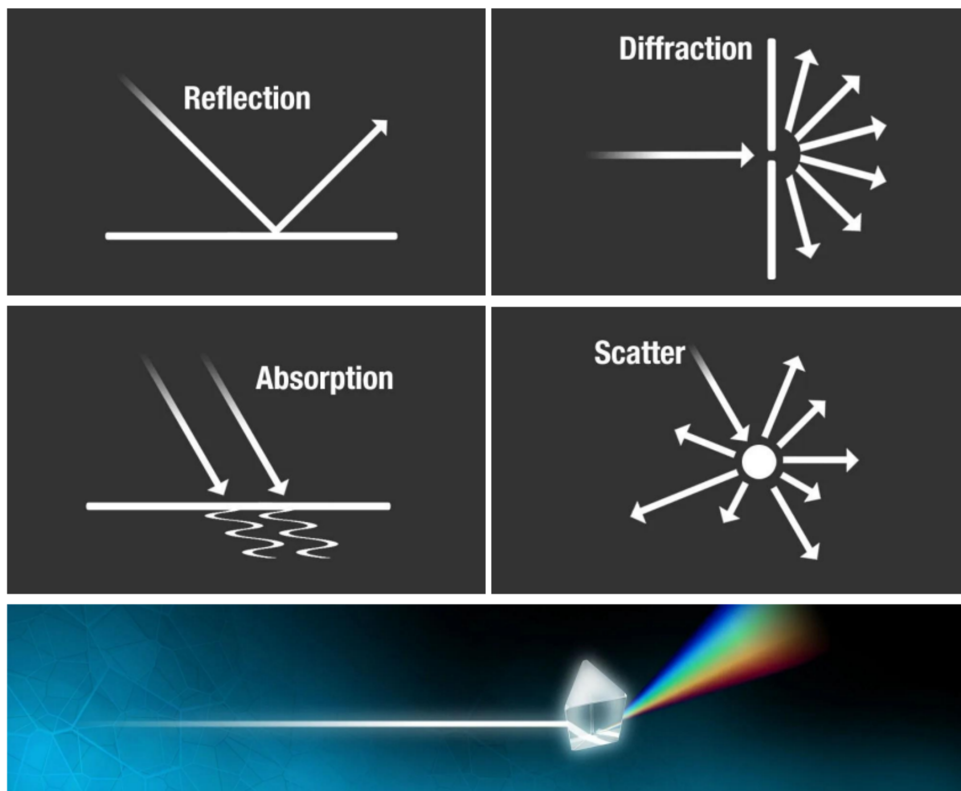


Figure 1: different ways of interaction of light: [Joh14]

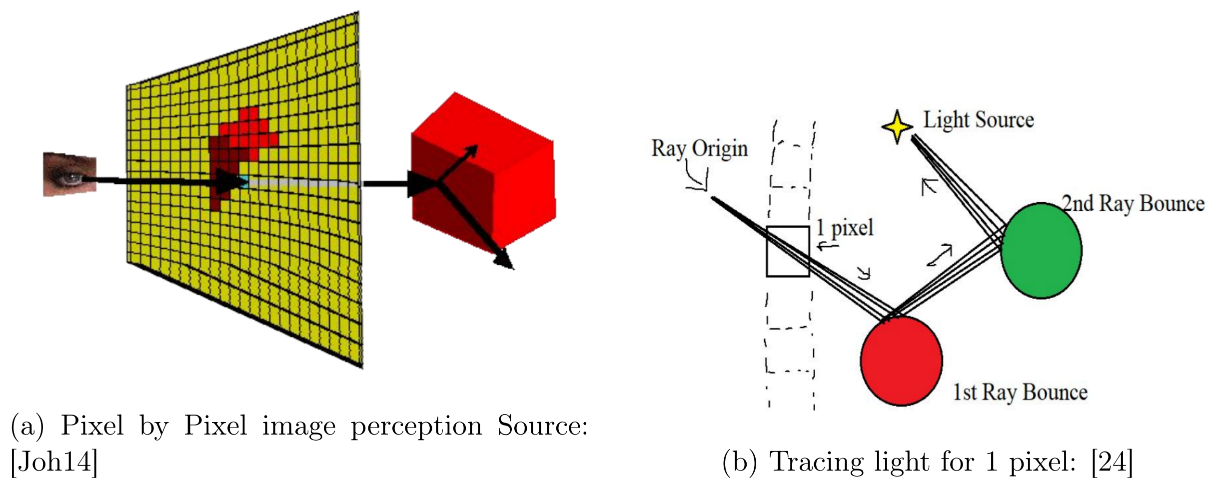
The primary challenge in ray tracing is the high computational cost associated with rendering images. Each pixel in an image requires the simulation of numerous light

interactions, including reflections, refractions as shown in figure: 1, [Vas+18] and shadows. As the number of rays and interactions increases, so does the time required to produce a high-quality image. [Joh14] This presents a significant barrier to using ray tracing in real-time applications, such as video games and interactive simulations, where rendering speed is critical.

Given these challenges, the objective of this work is to explore methods for reducing rendering time while maintaining image quality. One promising approach is the parallelization of ray tracing algorithms, which distributes the computational workload across multiple processing units. By leveraging parallelization techniques, it is possible to achieve significant reductions in rendering time, making ray tracing more viable for real-time and resource-constrained environments. [Vas+18]

Parallelization Opportunities

The potential for parallelization in ray tracing is supported by several key factors:



(a) Pixel by Pixel image perception Source: [Joh14]

(b) Tracing light for 1 pixel: [24]

Figure 2: Observation: independence of task

- **Independence of Task:** Each pixel in the image is independent of the others (see Figure 2b, meaning the computation for one pixel does not affect the computation of another. This independence allows for parallel processing, where each pixel's ray tracing can be handled simultaneously by different processing units.
- **Data Independence:** The same tracing computation must be performed across the entire image data. Since each pixel requires similar calculations, the workload can be evenly distributed across multiple processors, further enhancing parallelization efficiency.
- **Speed-up Potential:** Given a large enough problem size, such as high-resolution images or complex scenes, the potential for speed-up through parallelization becomes significant. Each pixel can be individually calculated for its illumination and is independent of other pixels. (see 2) Larger problems benefit more from parallel processing, as the workload can be divided more effectively.
- **Resource Optimization:** By distributing the computational load across multiple processors or cores, the utilization of available resources can be maximized. This not

only reduces rendering times but also improves the overall efficiency of the system, allowing for faster and more cost-effective rendering.

These factors make ray tracing a suitable candidate for parallelization, offering the potential to overcome the inherent computational challenges and make high-quality, real-time rendering more accessible.

2 Background

Ray tracing is a powerful and flexible technique for rendering images by simulating the behavior of light as it interacts with objects in a scene. To implement ray tracing effectively, it's essential to understand the mathematical foundations, components involved in the scene setup, and the algorithm that drives the process. This section delves into these prerequisites and foundational concepts, setting the stage for a deeper exploration of parallelization techniques.

Mathematical Basics

At the core of ray tracing are vector operations, which are fundamental to computing the paths of light rays and their interactions with objects. The following are key mathematical concepts necessary for understanding and implementing ray tracing:

- **Vector Operations:** These include addition, subtraction, and calculating the length and direction of vectors. Vector operations are used to determine the position and movement of rays in 3D space. [Byjnd]
- **Unit Vector:** A unit vector has a magnitude of one and is used to indicate direction. In ray tracing, unit vectors are crucial for normalizing direction vectors, ensuring that calculations of light paths are accurate.[Byjnd]
- **Dot Product:** The dot product of two vectors is a scalar value that can be used to determine the angle between the vectors. This is particularly useful in calculating reflections and lighting interactions.[Byjnd]
- **Solving Quadratic Equations:** Many intersections between rays and objects, such as spheres, involve solving quadratic equations. The solutions to these equations determine the points at which rays intersect with objects. [Bri21]

These mathematical basics are the building blocks of ray tracing algorithms. A solid grasp of these concepts is necessary to implement the complex calculations involved in simulating realistic lighting and shading effects.

2.1 Components of a Ray Tracing Scene

In addition to the mathematical foundation, a ray tracing setup requires an understanding of the basic components that define a scene 3. Each component plays a vital role in how rays are generated, traced, and rendered. The key components include:

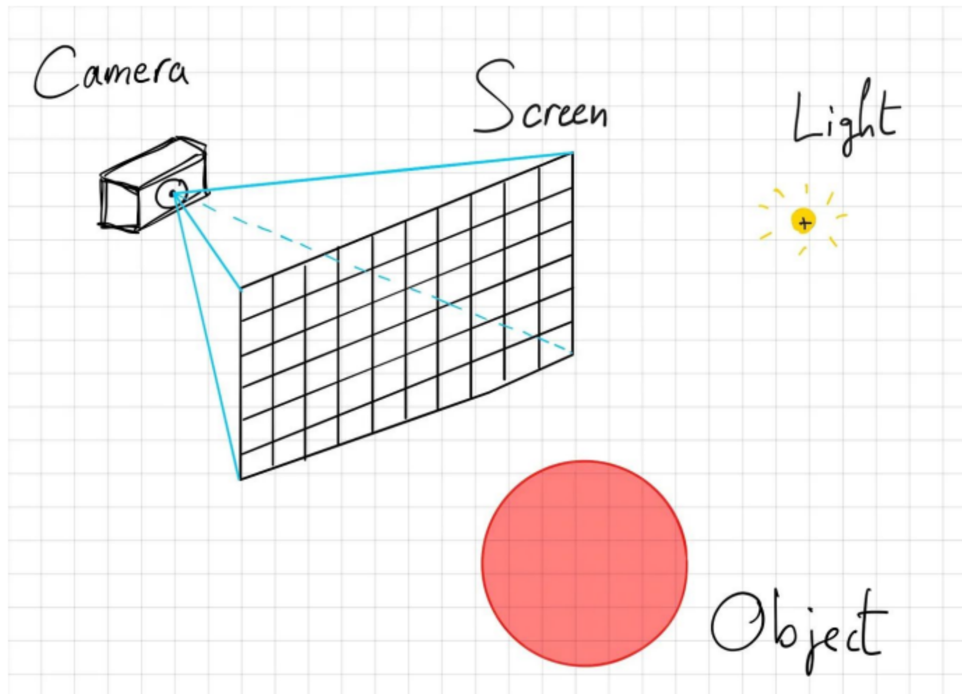


Figure 3: Components of ray tracing scene source: [Afl21]

- **3D Space:** The environment in which all objects are positioned using three coordinates (x, y, z). This space provides the framework for all ray tracing calculations.
- **Objects:** These are the entities within the 3D space that rays interact with. For simplicity, spheres are often used in basic ray tracing implementations due to their straightforward geometric properties. In more complex scenes, objects can include various shapes and materials.
- **Light Source:** A point within the scene that emits light in all directions. The light source is critical for determining how objects are illuminated and how shadows are cast.
- **Camera:** The position from which the scene is observed. The camera defines the viewpoint and the direction in which rays are cast through the scene.
- **Screen:** A rectangular plane through which the camera views the objects. The screen is composed of pixels, and each pixel represents a potential ray that can be traced from the camera through the scene.

These components together create the framework of a ray tracing scene, allowing for the simulation of light as it interacts with objects from the perspective of the camera.

2.2 Algorithm: The Pseudo Code Behind Ray Tracing

The ray tracing process is driven by an algorithm that simulates the behavior of light rays as they interact with objects in the scene. The following pseudocode outlines the basic steps involved in this process:

```
for each pixel of the screen:
    pixel.color = black
    ray <direction> = generateRay(camera, pixel)
    if extended ray at pixel intersects any object of the scene then:
        calculate the intersection point (i) to the nearest object
        if intersection == true && no object
            between point of intersection (i) and Light (l):
                pixel.color = calculateColor(object, i, l)
```

Listing 1: sudo_code

In this algorithm, rays are generated from the camera through each pixel on the screen. The rays are then extended into the scene to check for intersections with objects. If an intersection occurs, the algorithm calculates the exact point of intersection and determines whether any objects obstruct the path to the light source. If no obstruction is found, the pixel's color is updated based on the calculated lighting and shading effects.

2.3 The Scene Setup

To provide a concrete example of how the ray tracing algorithm is applied, consider a simple scene setup:

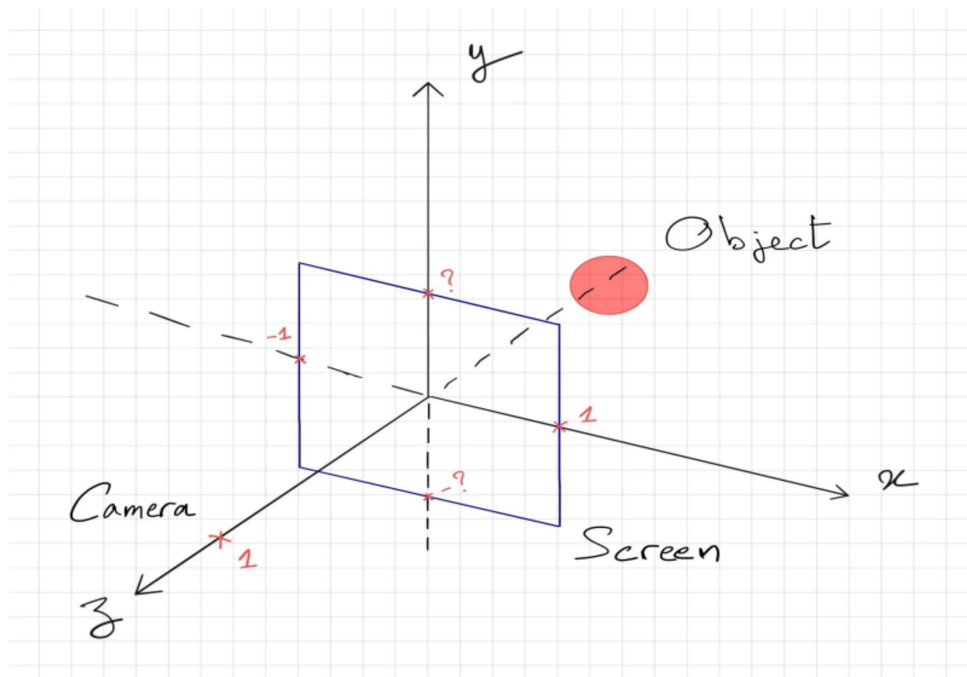


Figure 4: Scene setup source: [Afl21]

- **Camera Location:** The camera is positioned at the point (x=0, y=0, z=1), providing a view of the scene from above.
- **Screen Position:** The screen is part of the plane formed by the x and y axes, through which the camera observes the objects in the scene.

- **Ray Intersection:** For each pixel on the screen, a ray is generated from the camera and extended through the scene. The algorithm checks for intersections between the ray and any objects, calculating the point of intersection and determining whether the pixel should be illuminated.

This simple setup allows for the demonstration of the core concepts of ray tracing, such as ray generation, intersection calculation, and lighting determination.

2.4 Ray Intersection & Definition

The next step of the algorithm is to determine whether the ray (line) originating from the camera and directed towards point p intersects any object in the scene.

Since the ray starts at the camera and extends in the direction of the currently targeted pixel, a unit vector should point in a similar direction. Therefore, a "ray that starts at the camera and goes towards the pixel" can be defined as follows:

$$\text{ray}(t) = \text{camera} + \frac{\text{pixel} - \text{camera}}{\|\text{pixel} - \text{camera}\|} t$$

The ray moves further away from the camera in the direction of the pixel. This formulation represents a parametric equation yielding a point along the line for a given t.

In a similar manner, a ray can also be defined that starts at the origin **O** and extends towards a destination **D**:

$$\begin{aligned} \text{ray}(t) &= O + \frac{D - O}{\|D - O\|} t \\ &= O + d \cdot t \end{aligned}$$

For convenience, **d** is defined as the direction vector. Subsequently, the code can be completed by adding the computation of the ray. The `normalize(vector)` function, which returns a normalized vector, has been included. Additionally, the computation of the origin and direction, which together define a ray, has been established. It is noteworthy that the pixel has $z = 0$ since it lies on the screen, which is contained in the plane formed by the x and y axes. [Afl21]

2.5 Sphere Intersection

The ray equation and the condition for a point to lie on a sphere are known. Thus, substituting the ray equation into the sphere equation allows for the determination of t. This involves addressing the question of for which t the expression ray(t) lies on the sphere:

$$\|\mathbf{r}(t) - \mathbf{C}\|^2 = r^2 \tag{1}$$

This results in an ordinary quadratic equation solvable for t. The coefficients associated with t^2, t^1, and t^0 are denoted as a, b, and c respectively. The discriminant of this equation can be calculated as follows:

$$D = b^2 - 4ac \tag{2}$$

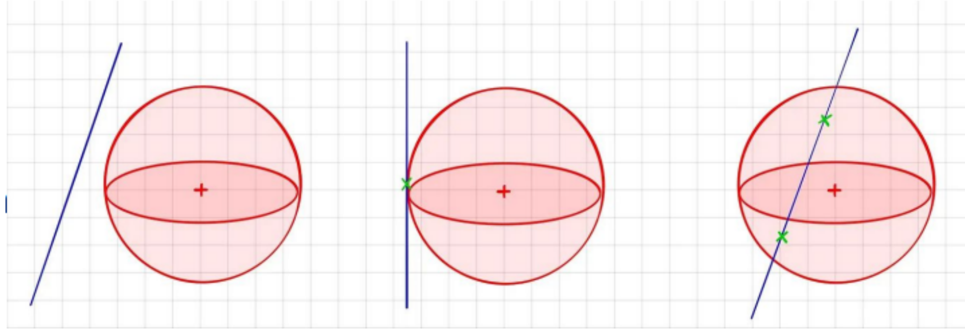


Figure 5: Sphere Discriminant

Given that **d** (the direction) is a unit vector, a equals 1. Upon calculating the discriminant, three possibilities arise:

Only the third case will be utilized to detect intersections from Figure 5. The following function can detect intersections between a ray and a sphere. It will return t, the distance from the origin of the ray to the nearest intersection point, if the ray intersects the sphere. If no intersection occurs, it will return None. It is essential to note that only the nearest intersection is returned (as there are two potential intersections) when both t_1 and t_2 are positive.

2.6 Blinn Phong Reflection

To simulate realistic lighting, ray tracing often incorporates an illumination model. One such model is the Blinn Phong reflection model [Con20], which defines how light interacts with the surface of objects to produce different visual effects. According to this model, any material in the scene has the following properties:

$$I_p = k_a * i_a + k_d * i_d * L \cdot N + k_s * i_s * \left(N \cdot \frac{L + V}{\|L + V\|} \right)^{\frac{\alpha}{4}}$$

- **Ambient Color:** The color an object appears in the absence of light.
- **Diffuse Color:** The color closest to what we typically perceive as the object's color, resulting from diffuse reflection.
- **Specular Color:** The color of the shiny part of an object when light strikes it, often appearing white.
- **Shininess:** A coefficient representing the object's reflective properties, determining how sharp or blurred the reflections appear [Par19].

The illumination of a point on an object's surface is calculated using the Blinn Phong model by combining the ambient, diffuse, and specular components, each modulated by the light's properties and the object's material coefficients.

2.7 Sphere Intersection: Geometric Considerations

In the context of ray tracing, accurately calculating the intersection between rays and spheres is crucial. The process involves determining the distance t from the origin of the

ray to the nearest point of intersection. This distance is found by solving a quadratic equation that yields two potential values for t, denoted as t_1 and t_2. The smaller of these values represents the first point of contact between the ray and the sphere.

To determine whether a pixel should be illuminated, the algorithm checks if the light source is directly visible from the intersection point. If no objects obstruct the path between the intersection point and the light source, the pixel is illuminated; otherwise, it remains in shadow.

This background section provides a comprehensive overview of the fundamental concepts, mathematical foundations, and components involved in ray tracing. By understanding these elements, we can appreciate the challenges and opportunities in optimizing ray tracing through parallelization, which will be explored in subsequent sections of the report.

3 Parallelization with MPI

Parallelization is a crucial aspect of the ray tracing implementation in this project. To achieve efficient parallelization, we use the `mpi4py` library, which provides bindings for the Message Passing Interface (MPI) in Python. The use of MPI allows the workload to be distributed across multiple processes, significantly reducing the computation time for rendering high-resolution images.

3.1 MPI Setup and Process Distribution

One key challenge in parallelizing the ray tracing process is ensuring that the image is evenly distributed among the available processes, especially when the number of rows in the image is not divisible by the number of processes. In early versions of the implementation, this mismatch resulted in a tiling effect in the final image, where some rows were duplicated or misaligned. To resolve this issue, we adjusted the distribution of rows and used `MPI.Gatherv` to correctly gather variable-sized segments from each process.

The main strategy for parallelizing the ray tracing process is to divide the image into horizontal strips, where each process is responsible for rendering a specific portion of the image. This is achieved by assigning each process a range of rows to compute.

```
1  comm = MPI.COMM_WORLD
2  size = comm.Get_size()  # Number of processes
3  rank = comm.Get_rank()  # Process rank (ID)
4
5  # Calculate rows assigned to each process
6  rows_per_proc = config["height"] // size
7  remainder = config["height"] % size
8
9  if rank < remainder:
10     start_row = rank * (rows_per_proc + 1)
11     end_row = start_row + rows_per_proc + 1
12 else:
13     start_row = rank * rows_per_proc + remainder
14     end_row = start_row + rows_per_proc
15
```

```
16 image_segment = np.zeros((end_row - start_row, config["width"], 3))
```

Listing 2: MPI Initialization and Row Distribution

In this updated row distribution logic, extra rows (if the image height is not divisible by the number of processes) are distributed among the first few processes. This ensures that each process handles a balanced number of rows and no part of the image is duplicated or skipped.

3.2 Handling Variable-Sized Image Segments to Prevent Tiling

Once the row distribution is set, each process computes its assigned rows of the image. The next challenge is gathering the computed image segments correctly on the root process. Since each process may handle a different number of rows, we use `MPI.Gatherv`, which allows us to gather variable-sized data from each process and place it correctly in the final image.

```
1
2  # Prepare to gather the image segments with variable sizes
3  sendcounts = np.array([((config["height"] // size) + (1 if r < remainder
4      else 0)) * config["width"] * 3 for r in range(size)])
5  displacements = np.array([sum(sendcounts[:r]) for r in range(size)])
6
7  combined_image = None
8  if rank == 0:
9      combined_image = np.zeros((config["height"], config["width"], 3))
10
11 # Gather the image segments into the combined image
12 comm.Gatherv(image_segment, [combined_image, sendcounts, displacements,
13              MPI.DOUBLE], root=0)
```

Listing 3: Handling Variable-Sized Image Segments to Prevent Tiling

The use of `sendcounts` and `displacements` ensures that each process sends the correct number of rows, and these rows are placed in the appropriate positions in the final image. This adjustment fixes the tiling effect and ensures that the image is seamless when gathered from multiple processes.

3.3 Performance Benefits of Parallelization

The independence of pixel calculations makes ray tracing highly suitable for parallelization. Each pixel can be computed independently, allowing the work to be easily divided among multiple processors. By using MPI, we can take advantage of multiple CPU cores to reduce the overall rendering time.

This parallelization approach results in significant performance improvements, particularly for larger and more complex scenes. For example, by dividing the workload among 20 processes, the rendering time is drastically reduced compared to running the program serially. The scalability of the solution ensures that it can handle even higher resolutions and more complex scenes efficiently as the number of available processors increases.

3.4 Setting Up the Environment and Running the Script

Before running the ray tracing script, it is important to set up the environment, install the necessary dependencies, and configure the system for parallel execution. This involves creating a Python virtual environment, installing required packages, and then running the script using either `mpiexec` or `srun` for parallel processing.

To begin, a Python virtual environment is created to isolate the dependencies, ensuring consistency across different runs of the experiment. You can create and activate the environment using the following commands:

```
# Load the necessary Python module (if required by the HPC environment)
module load python/3.9

# Create a virtual environment named "mpi"
python3 -m venv mpi

# Activate the virtual environment
source mpi/bin/activate
```

Once the environment is activated, the dependencies for the project can be installed via a `requirements.txt` file. This file lists the Python libraries needed to run the ray tracing script, such as `mpi4py`, `numpy`, and `matplotlib`. The following is an example of the `requirements.txt` file used for this project:

```
# requirements.txt
mpi4py
numpy
matplotlib
```

To install these dependencies, use the following command inside the activated virtual environment:

```
# Install all required packages
pip install -r requirements.txt
```

With the environment set up and the necessary packages installed, the ray tracing script can be run in two ways: manually using `mpiexec` or via job submission using `srun` in a Slurm-managed environment.

If you prefer to run the script manually, you can use `mpiexec` to control the number of MPI processes, as shown below:

```
# Activate the virtual environment
source mpi/bin/activate

# Run the Python script with 4 MPI processes
mpiexec -n 4 python ray_tracer.py low

# For medium and high complexity scenes, you can specify different levels:
mpiexec -n 4 python ray_tracer.py medium
mpiexec -n 4 python ray_tracer.py high
```

In this example, the -n option is used to specify the number of MPI processes, while the complexity of the scene (low, medium, or high) is passed as an argument to the Python script.

Alternatively, the ray tracing script can be executed via Slurm using the srun command inside a Slurm batch script. You can submit the batch script with sbatch as follows:

```
# Submit the batch script using Slurm
sbatch your_batch_script.sh
```

Inside the batch script, the srun command is used to run the Python script for different scene complexities:

```
srun python ray_tracer.py low
srun python ray_tracer.py medium
srun python ray_tracer.py high
```

After running the experiments, the final images are generated and saved for review. The results are validated by checking for correctness in the output (e.g., no tiling artifacts) and measuring performance metrics, including total execution time and speedup from parallelization.

4 Results

In this section, we present the visual output of the ray tracing algorithm for various scene complexities and light source positions. These images were generated during the experiments using different levels of scene complexity and configurations for the light source position, and they illustrate the correctness and performance of the algorithm.

4.1 Scene Complexity Results

The following images show the output of the ray tracing algorithm for three different levels of scene complexity: low, medium, and high. Each image is rendered with an increasing number of spheres and objects, demonstrating the algorithm's ability to handle scenes of varying computational difficulty.

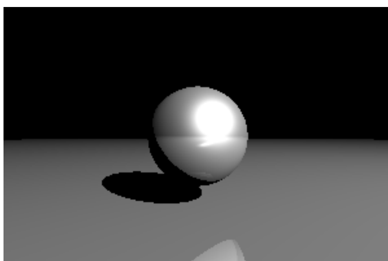


Figure 6: Low Complexity Scene (2 spheres)

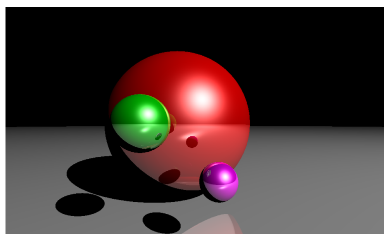


Figure 7: Medium Complexity Scene (4 spheres)

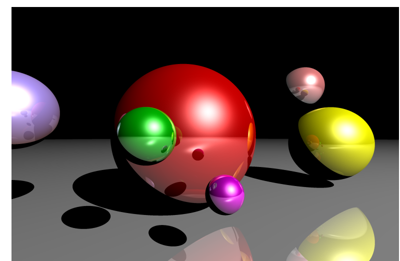


Figure 8: High Complexity Scene (7 spheres)

These images illustrate the progression in complexity, starting from the simplest scene (two spheres) to more complex scenes with additional spheres and reflections. The increase

in the number of objects results in more complex interactions between light, shadows, and reflections, demonstrating the efficiency of the ray tracing algorithm.

4.2 Light Source Position Results

In this experiment, we varied the position of the light source to test the algorithm's accuracy in generating shadows. The light source was positioned to the left, right, and top of the scene, and the results were compared by inspecting the shadows cast by the spheres. The following images show the output of the scene with the light source in different positions.

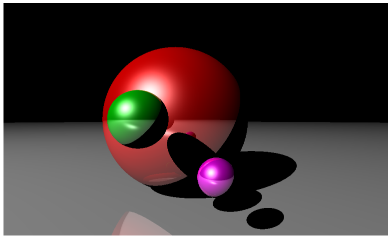


Figure 9: Light Source on the Left

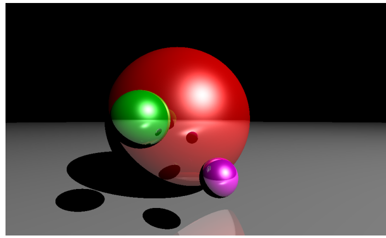


Figure 10: Light Source on the Right

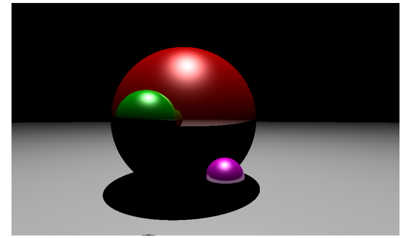


Figure 11: Light Source on the Top

As shown in the images, the shadows cast by the spheres correctly correspond to the position of the light source. For example:

- When the light source is positioned to the **left**, the shadows are cast to the right of the spheres.
- When the light source is positioned to the **right**, the shadows move to the left of the spheres.
- With the light source placed at the **top**, the shadows appear below the spheres.

This confirms the correctness of the ray tracing algorithm in calculating the direction and position of shadows based on the light source's position.

4.3 Performance Results Based on Scene Complexity

The following graphs display the relationship between the number of processes and the execution time for three different scene complexities: low, medium, and high. The experiments were performed with varying numbers of processes, and the execution times were recorded to analyze the scalability and performance of the parallelized ray tracing algorithm.

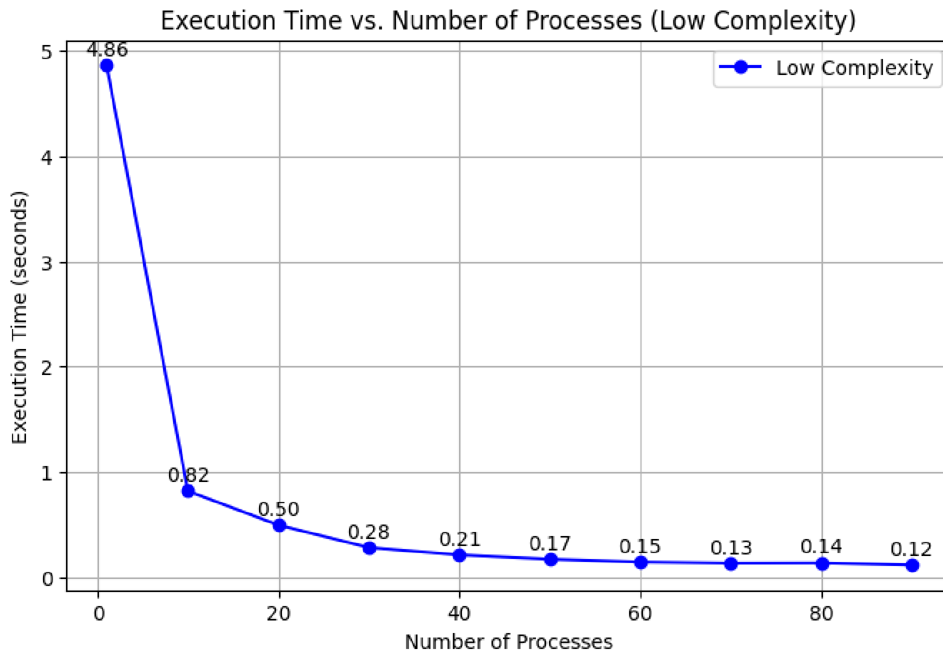


Figure 12: Execution Time vs. Number of Processes (Low Complexity)

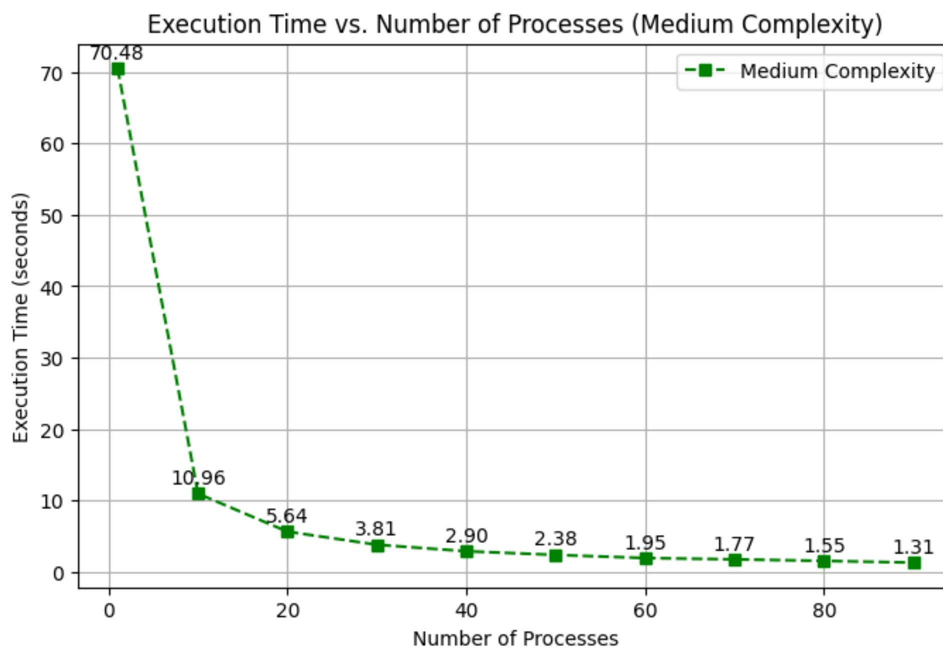


Figure 13: Execution Time vs. Number of Processes (Medium Complexity)

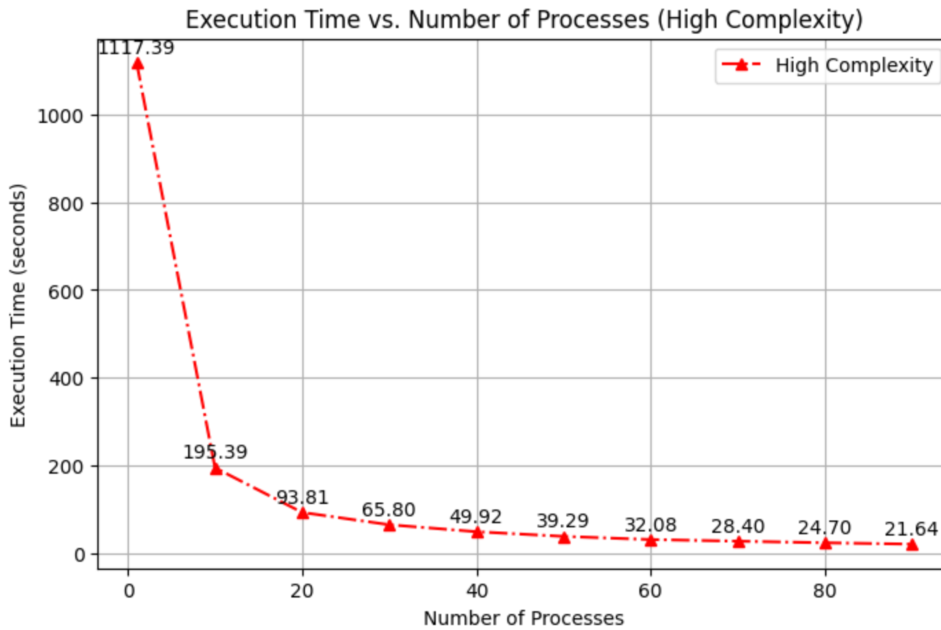


Figure 14: Execution Time vs. Number of Processes (High Complexity)

4.3.1 Observations

The results from the three graphs show a consistent reduction in execution time as the number of processes increases, demonstrating the effectiveness of parallelization. Across all scene complexities—low, medium, and high—the execution time roughly halves each time the number of processes is doubled, particularly in the early stages of parallelization. This behavior illustrates the strong scalability of the ray tracing algorithm, especially for more complex scenes.

In the initial stages, as the number of processes is increased from 1 to 10, there is a substantial reduction in execution time for all scene complexities. Doubling the number of processes from 10 to 20 results in a further noticeable decrease in execution time. For instance, the execution time consistently drops by a factor of two or more as the number of processes doubles, reflecting the linear scalability of the parallelized algorithm.

However, as the number of processes continues to increase, the rate of improvement diminishes. This is particularly evident in the low complexity scene, where the execution time begins to level off after about 20 processes, and further increases in process count provide only marginal gains. This behavior is expected in simpler scenes where the computational workload is not sufficient to fully utilize a larger number of processes, and the overhead of managing more processes begins to counteract the performance benefits.

For medium and high complexity scenes, the trend of execution time halving continues for a larger range of process counts, reflecting the greater computational demands of these scenes. Even as the number of processes reaches 80 or 90, there are still measurable reductions in execution time, particularly in the high complexity scene. This indicates that for more demanding workloads, parallelization remains effective at distributing the computation across a larger number of processes, though the rate of improvement slows as the number of processes increases.

5 Conclusion

This report investigated the implementation and performance of a parallelized ray tracing algorithm using MPI, focusing on how varying the number of processes impacts execution time across different scene complexities. By distributing the computational workload among multiple processes, the algorithm was able to significantly reduce rendering times, particularly for more complex scenes. The results showed that in the initial stages of parallelization, doubling the number of processes led to a near-halving of execution time, demonstrating strong scalability. However, as the number of processes increased further, the benefits diminished, particularly for simpler scenes where the computational workload was not sufficient to fully utilize all the available processes.

For more complex scenes, such as those with higher object counts and reflections, the algorithm continued to benefit from additional processes, though at a slower rate. While the high complexity scene saw measurable reductions in execution time even with 90 processes, the results emphasized the importance of balancing the number of processes with the scene's complexity. Over-parallelization can lead to diminishing returns, especially when the overhead of managing more processes outweighs the performance gains. Future work could explore optimizing the algorithm for more heterogeneous workloads and implementing dynamic load balancing to ensure even better resource utilization across different complexity levels.

References

[24] active tonight active. *Reddit - Dive into anything*. Reddit.com, 2024. URL: https://www.reddit.com/r/raytracing/comments/rv1er1/sample_per_pixel_and_ray_per_pixel_in_ray_and/ (visited on 09/29/2024).

[Afl21] Omar Aflak. *Ray Tracing From Scratch in Python*. Medium, Aug. 2021. URL: https://omaraflak.medium.com/ray-tracing-from-scratch-in-python-41670e6a96f9.

[Bri21] A. Brijesh. *Quadratics (Quadratic Equation) - Definition, Formula & Solution*. Accessed: 2024-09-29. 2021. URL: https://byjus.com/maths/quadratics/.

[Byjnd] Byju. *Vector Formulas with Solved Examples and Equations*. https://byjus.com/vector-formulas/. Accessed: [insert access date]. n.d.

[Con20] Wikipedia Contributors. *Blinn–Phong reflection model*. Accessed: 2024-09-29. 2020. URL: %5Curl%7Bhttps://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model%7D.

[Joh14] Brian R. Johnson. *Ray trace*. Washington.edu, 2014. URL: https://courses.washington.edu/arch481/1.Tapestry%20Reader/ (visited on 09/29/2024).

[Jun+21] Wang Jun-Feng et al. "Parallel optimization of the ray-tracing algorithm based on the HPM model". In: *The Journal of Supercomputing* 77 (Mar. 2021), pp. 10307–10332. DOI: 10.1007/s11227-021-03680-0. URL: https://d-nb.info/1231681616/34 (visited on 05/05/2023).

[KK08] Syukriah Kadir and Tazrian Khan. "Parallel Ray Tracing using MPI and OpenMP". In: (Jan. 2008).

[Nvi18] Nvidia. *Ray Tracing*. NVIDIA Developer, Aug. 2018. URL: https://developer.nvidia.com/discover/ray-tracing.

[Par19] Paroj. *Blinn-Phong Model*. Accessed: [insert access date]. 2019. URL: %5Curl%7Bhttps://paroj.github.io/gltut/Illumination/Tut11%20BlinnPhong%20Model.html%7D.

[Vas+18] Elena Vasiou et al. "A detailed study of ray tracing performance: render time and energy cost". In: *The Visual Computer* 34 (Apr. 2018), pp. 875–885. DOI: 10.1007/s00371-018-1532-8. (Visited on 11/20/2021).