PARALLELIZATION OF RAY TRACING

GNANI PASUPULA

CSE 633 - Parallel Algorithms

Instructor - Dr. Russ Miller

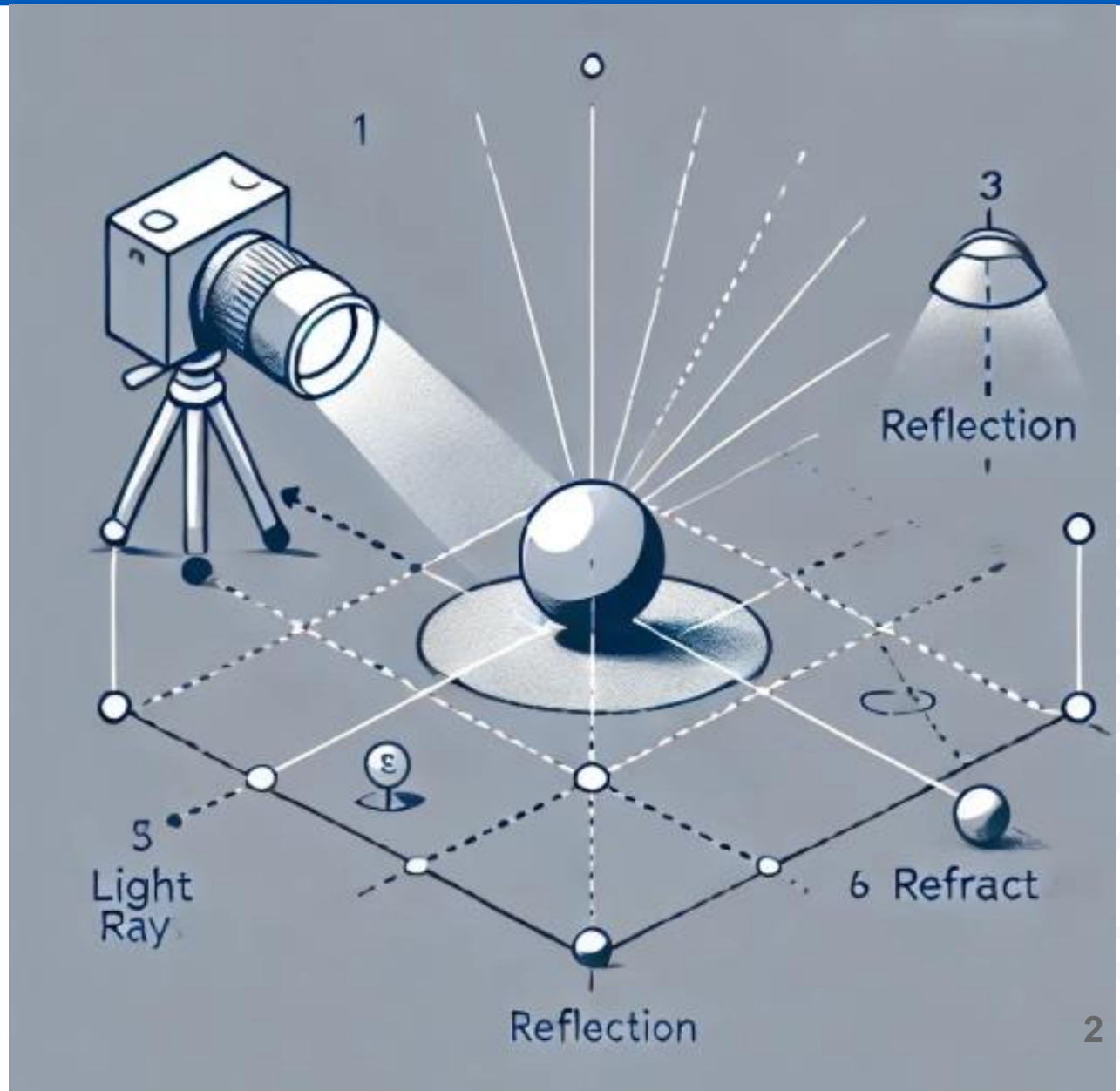
University at Buffalo The State University of New York

What is Ray Tracing?

Ray tracing is a rendering technique that simulates how light interacts with objects in a scene. It works by tracing rays of light from the camera through each pixel into the scene, determining how they interact with surfaces, and calculating the final color.

Process Breakdown

- **Ray Casting** – A ray is cast from the camera through each pixel into the scene.
- **Intersection Testing** – The ray is checked for intersections with objects. The closest intersection determines the visible surface.
- **Color Calculation** – The surface color is computed based on material properties, light sources, and shading models.
- **Reflection & Refraction** – If the material is reflective or transparent, secondary rays are traced to compute reflections and refractions, adding realism.



Why is Ray Tracing Computationally Intensive?

- Each pixel requires **tracing multiple rays**, performing **intersection tests**, and computing **lighting effects** (shadows, reflections, refractions).
- Complexity increases with **scene detail**, **object count**, and **light interactions**.
- Rendering high-resolution images can involve **millions of rays**, making real-time ray tracing extremely demanding.

Limitations of the Sequential Approach

- **Linear Execution:** Each pixel is computed one at a time, leading to long render times.
- **Exponential Complexity:** Advanced effects (global illumination, soft shadows) require more rays, increasing computational load.
- **Not Scalable:** As resolution and scene complexity grow, rendering time increases drastically.

Parallel Computing With MPI

- **Tasks:** A unit of work that needs to be performed. In ray tracing, a task could be rendering a portion of the image (e.g., a row, a tile, or a set of pixels).
- **Processes:** Independent units of execution that can run concurrently. In the MPI implementation, each process will be assigned a portion of the rendering task.
- **Communication (Message Passing):** How different processes exchange information and coordinate their work? We use MPI to send and receive data (e.g., the rendered portions of the image) between processes.
- **Synchronization :** Mechanisms to coordinate the execution of processes to ensure correct results. We use barriers (`MPI_Barrier`) to ensure all processes complete a certain stage before proceeding.

Parallelization Strategy – Task Decomposition

- **Image Decomposition**: Task/Image divided into a few horizontal or vertical strips, with each strip labeled "Process 1," "Process 2," etc.
- **Pixel-Level Decomposition:** Each Task/Image with individual pixels or small blocks highlighted and labeled as being worked on by different processes.

Our Approach: Image Decomposition

Why? We opted for **Image Decomposition**, specifically dividing the final image. Each process in our MPI implementation is responsible for rendering a specific portion of the image.

- **Justification:**
  - **Simplicity:** This approach is relatively straightforward to implement using MPI's communication primitives (like `MPI_Gather`).
  - **Data Locality:** Each process primarily works on a contiguous block of pixels, which can improve data locality and reduce the need for complex inter-process communication during the core ray tracing calculations.
  - **Load Balancing :** For a well-defined image, the workload for each strip is generally balanced, as each strip contains a similar number of pixels.

With MPI:

- **Few Key MPI Concepts Utilized in Our Project:**
- **`MPI_Init()`:** Initializes the MPI environment.
- **`MPI_Comm_rank()`:** Determines the unique rank (ID number) of the current process within a communicator.
- **`MPI_Comm_size()`:** Determines the total number of processes in the communicator.
- **`MPI_Gather()`:** Collects data from all processes and sends it to a designated root process. This was used to assemble the final rendered image.

Sequential Ray Tracing Algorithm

- **Generate Ray:**
- Determine the screen coordinates of the current pixel.
- Construct a ray originating from the camera position and passing through the pixel.
- **Ray-Scene Intersection:**
- For each object in the scene (e.g., spheres):
  - Calculate the intersection point of the ray with the object.
  - If an intersection occurs and is closer than the current closest intersection, record it.

```python
def trace_ray_sequential(ray_origin, ray_direction, scene_objects, lights):
    """
    Traces a single ray through the scene and returns the color of the hit object
    or the background color.

    """
    closest_hit = float('inf')
    closest_object = None

    # 1. Ray-Scene Intersection
    for obj in scene_objects:
        if obj['type'] == 'sphere':
            hit_distance = intersect_sphere(ray_origin, ray_direction, obj['center'], obj['radius'])
            if hit_distance is not None and hit_distance > 1e-6 and hit_distance < closest_hit:
                closest_hit = hit_distance
                closest_object = obj

    # 2. Calculate Color
    if closest_object:
        hit_point = ray_origin + ray_direction * closest_hit
        normal = normalize(hit_point - closest_object['center'])
        color = np.array([0.0, 0.0, 0.0])

        # Lighting
        for light in lights:
            light_direction = normalize(light['position'] - hit_point)
            # Shadow check
            shadowed = False
            for obj in scene_objects:
                if obj != closest_object and obj['type'] == 'sphere':
                    shadow_hit = intersect_sphere(hit_point + 1e-6 * light_direction, light_direction, obj['center'], obj['radius'])
                    if shadow_hit is not None and shadow_hit > 1e-6:
                        shadowed = True
                        break
            if not shadowed:
                diffuse = max(0, np.dot(normal, light_direction))
                color += closest_object['color'] * light['intensity'] * diffuse

        return np.clip(color, 0, 1)
    else:
        # Background color (e.g., sky blue)
        return np.array([0.5, 0.7, 1.0])
```

Parallel Ray Tracing with MPI

- **MPI as the Framework:** We utilized the Message Passing Interface (MPI) to parallelize the ray tracing process across multiple processors.
- **Image Distribution Strategy:** We employed **Image Decomposition** by dividing the final image into horizontal strips.
- **Work Assignment:** Each MPI process is responsible for rendering a specific portion of the image. The master process (rank 0) distributes the work, and each worker process renders its assigned pixels independently.

```python
def parallel_render_scene(width, height, spheres, lights, camera):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    image = np.zeros((height, width, 3))
    rows_per_process = height // size
    start_row = rank * rows_per_process
    end_row = (rank + 1) * rows_per_process if rank < size - 1 else height

    for y in range(start_row, end_row):
        for x in range(width):
            x_screen = (2 * (x + 0.5) / width - 1) * camera['aspect_ratio'] * camera['scale']
            y_screen = (1 - 2 * (y + 0.5) / height) * camera['scale']
            ray_direction = normalize(np.array([x_screen, y_screen, -1]))
            image[y, x] = trace_ray_parallel(camera['position'], ray_direction, spheres, lights)

    gathered_image = comm.gather(image[start_row:end_row], root=0)

    if rank == 0:
        final_image = np.zeros((height, width, 3))
        current_row = 0
        for part in gathered_image:
            rows = part.shape[0]
            final_image[current_row:current_row + rows] = part
            current_row += rows
        return final_image
    else:
        return None
```

Communication & Synchronization

- **Communication Mechanisms (using MPI):**
- **Data Exchange:** The primary communication happens when each process finishes rendering its assigned portion of the image.
- **`MPI_Gather()` for Image Assembly:** We used the `MPI_Gather()` function to collect the rendered image segments from all the worker processes and send them to the root process (rank 0). This allows the root process to assemble the complete final image.
- **Synchronization Mechanisms:**
- **`MPI_Barrier()` for Coordination:** We utilized `MPI_Barrier()` at strategic points in our code to ensure that all processes reach a certain point before any of them proceed further. This was particularly important:
  - **After work assignment:** To ensure all processes know their assigned image section before starting rendering.
  - **Before image collection:** To ensure all processes have finished rendering their parts before the `MPI_Gather()` operation.

Communication & Synchronization

- **Communication Bottlenecks Considered:**
- **`MPI_Gather()` Overhead:** The `MPI_Gather()` operation, while essential for assembling the final image, can become a bottleneck if the number of processes is very large or if the image size is extremely high. The root process needs to receive and combine data from all other processes.

Overcoming Bottlenecks

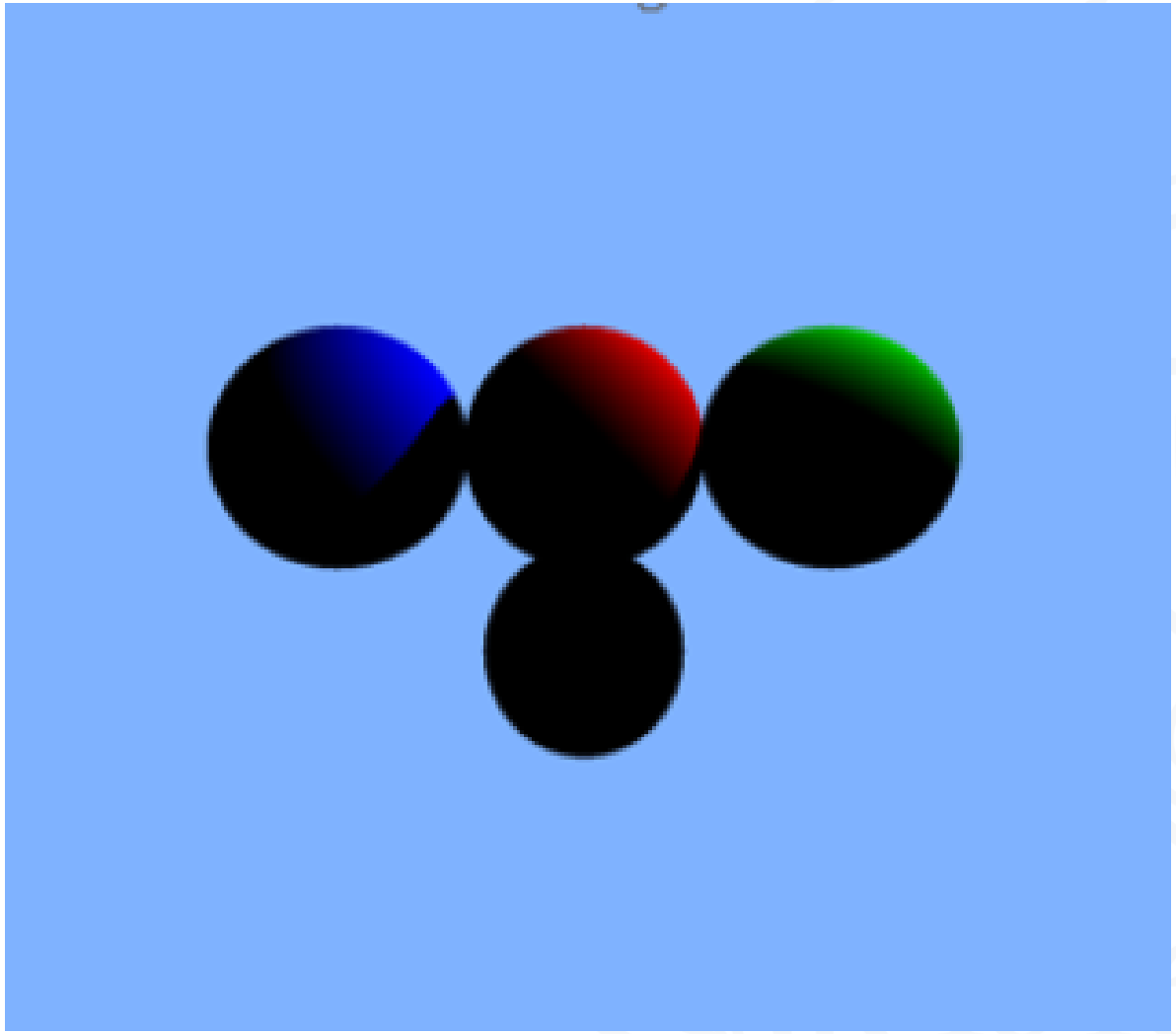
- **1. `MPI_Gather()` Overhead:**
- **Reduce Data Volume:**
  - **Lossy Compression :** If acceptable for your application, consider compressing the image data before gathering. However, be mindful of the added computational cost of compression/decompression.
- **Alternative Communication Patterns (for very large scale):**
  - **Scatter-Gather:** Instead of a single gather, consider a more distributed approach where intermediate results are gathered in stages or within subgroups of processes.
  - **Non-Blocking Communication:** Use non-blocking `MPI_Igather` and `MPI_Wait` to overlap communication with computation, potentially hiding some of the communication latency. However, this adds complexity to the code.

MPI Functions used

- MPI_Init(&argc, &argv)
- MPI_Comm_rank(MPI_COMM_WORLD, &rank)
- MPI_Comm_size(MPI_COMM_WORLD, &size)
- MPI_Get_processor_name(hostname, &name_len)
- MPI_Wtime()
- MPI_Bcast(&num_images, 1, MPI_INT, 0, MPI_COMM_WORLD)
- MPI_Send(&count, 1, MPI_INT, proc, 0, MPI_COMM_WORLD)
- MPI_Recv(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
- MPI_Barrier(MPI_COMM_WORLD)
- MPI_Reduce(&processing_time, &max_processing_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD)
- MPI_Finalize()

Scene A: Simple Scene

- Objects at similar distances: Simplifies intersection calculations.
- Non-overlapping objects: Reduces multi-object intersections, simplifying shadow/reflection rays
- Few objects: Fewer ray-object intersection tests per pixel.



Sequential vs Parallel Execution Times

- Simple scene

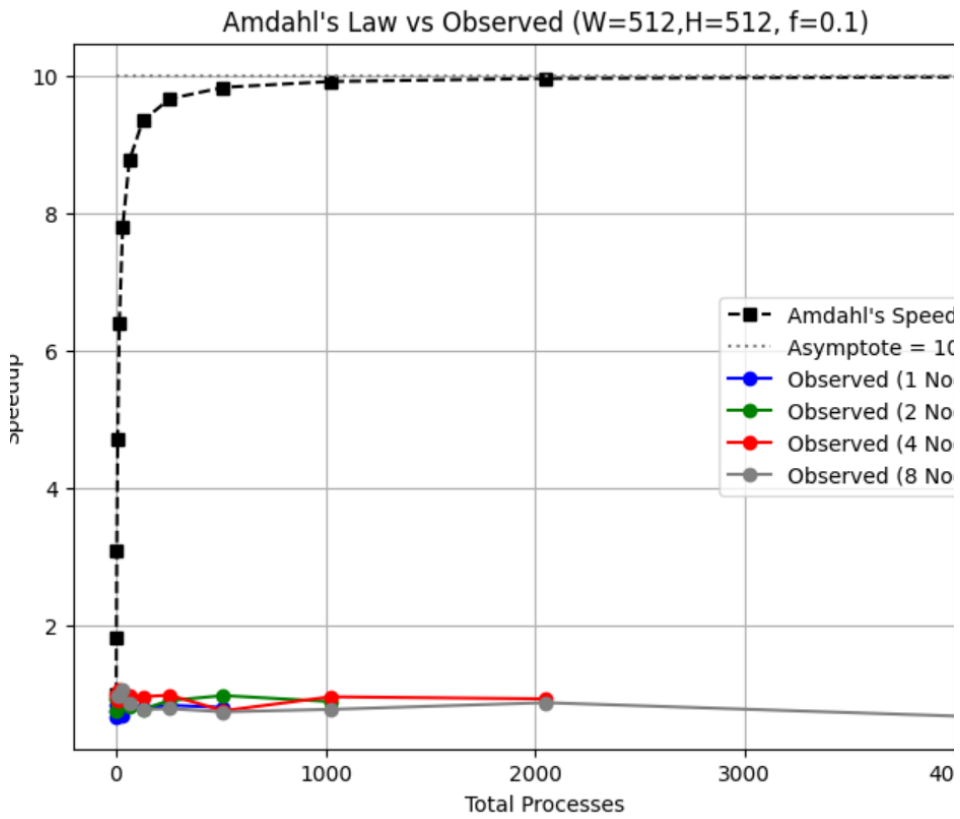
	Width	Height	Execution Time (s)
0	512	512	65.493668

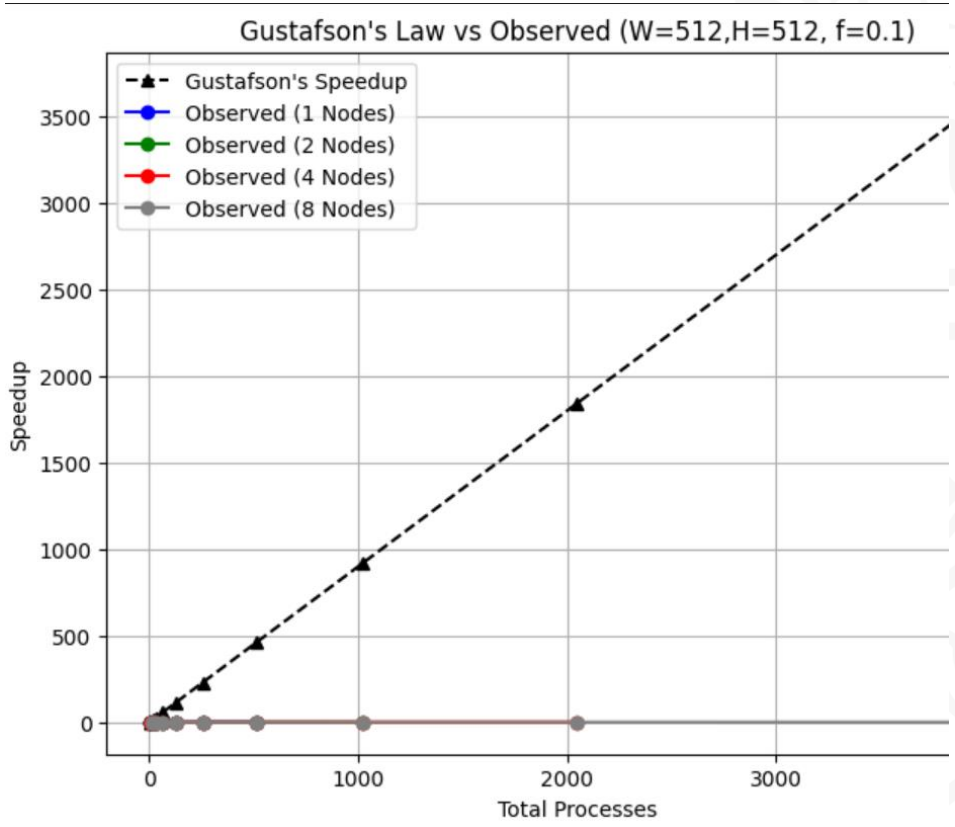
Parallel Table:

	Width	Height	Nodes	Processes/Node	Total Processes	Execution Time (s)
0	512	512	1	1	1	96.700848
1	512	512	1	2	2	77.361295
2	512	512	1	4	4	86.648971
3	512	512	1	8	8	83.363113
4	512	512	1	16	16	86.803929
5	512	512	1	32	32	94.492543
6	512	512	1	64	64	76.952073
7	512	512	1	128	128	82.202159
8	512	512	1	256	256	77.996892
9	512	512	1	512	512	81.022257
10	512	512	2	1	2	86.352805
11	512	512	2	2	4	69.606936
12	512	512	2	4	8	73.760027
13	512	512	2	8	16	78.458730
14	512	512	2	16	32	74.301440
15	512	512	2	32	64	79.027986
16	512	512	2	64	128	83.558853
17	512	512	2	128	256	72.121904
...						
36	512	512	8	64	512	88.056235
37	512	512	8	128	1024	83.964876
38	512	512	8	256	2048	74.634773


Observed Speedup vs. Total Processes (Width=512, Height=512)
1 Nodes
2 Nodes
4 Nodes
8 Nodes
Observed Speedup (T_sequential / T_parallel)
1.05
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
1000
2000
3000
4000
Total Processes

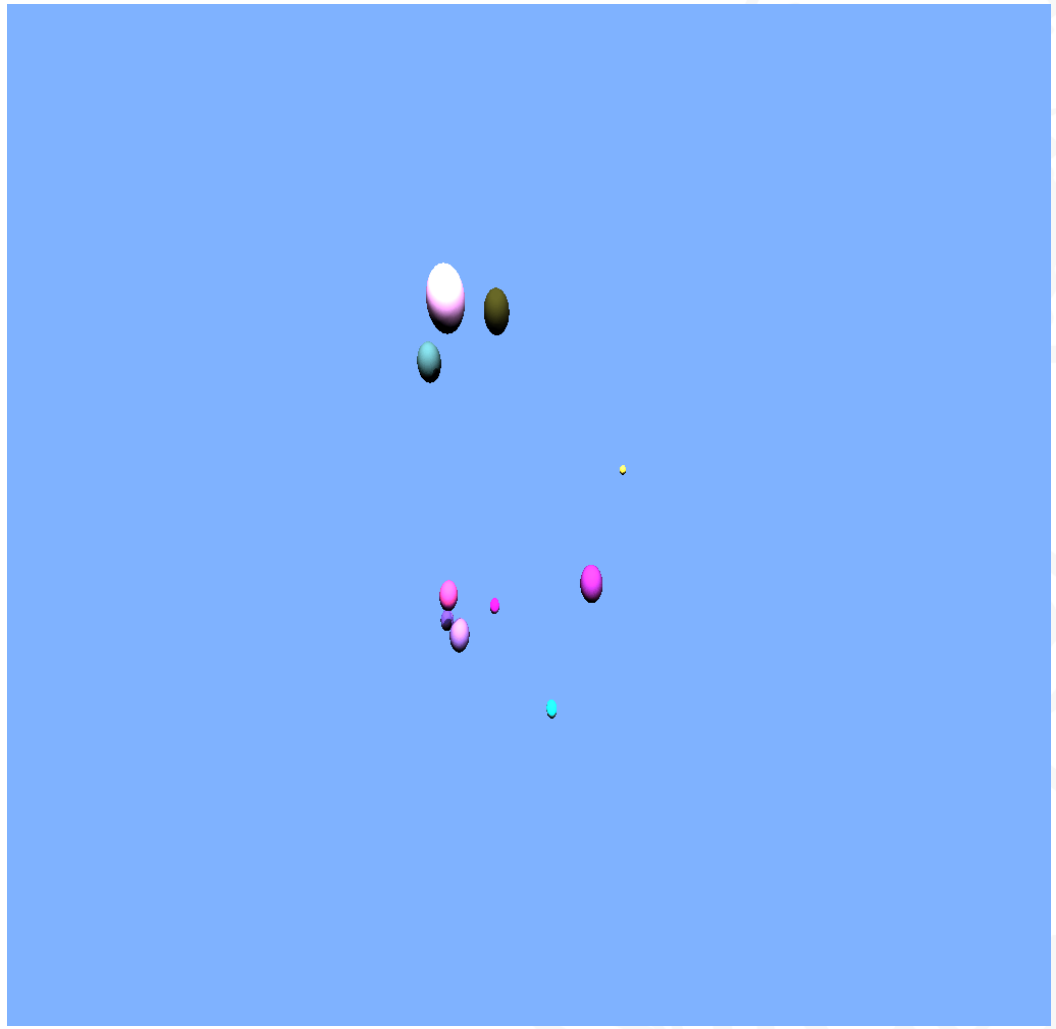
Output:


Amdahl's Law vs Observed (W=512,H=512, f=0.1)
Speedup
Total Processes
Amdahl's Speed
Asymptote = 10
Observed (1 No
Observed (2 No
Observed (4 No
Observed (8 No
0
1000
2000
3000
40
2
4
6
8
10


Gustafson's Law vs Observed (W=512,H=512, f=0.1)
Gustafson's Speedup
Observed (1 Nodes)
Observed (2 Nodes)
Observed (4 Nodes)
Observed (8 Nodes)
Speedup
Total Processes
0
500
1000
1500
2000
2500
3000
3500
0
1000
2000
3000

Scene B: Complex scene

- Objects at varying distances: Requires complex intersection calculations, potentially depth sorting.
- Overlapping objects: Leads to multi-object intersections, complex shadow, reflection, and refraction calculations.
- Many objects: Increases ray-object intersection tests per pixel.



Sequential vs Parallel Execution Times

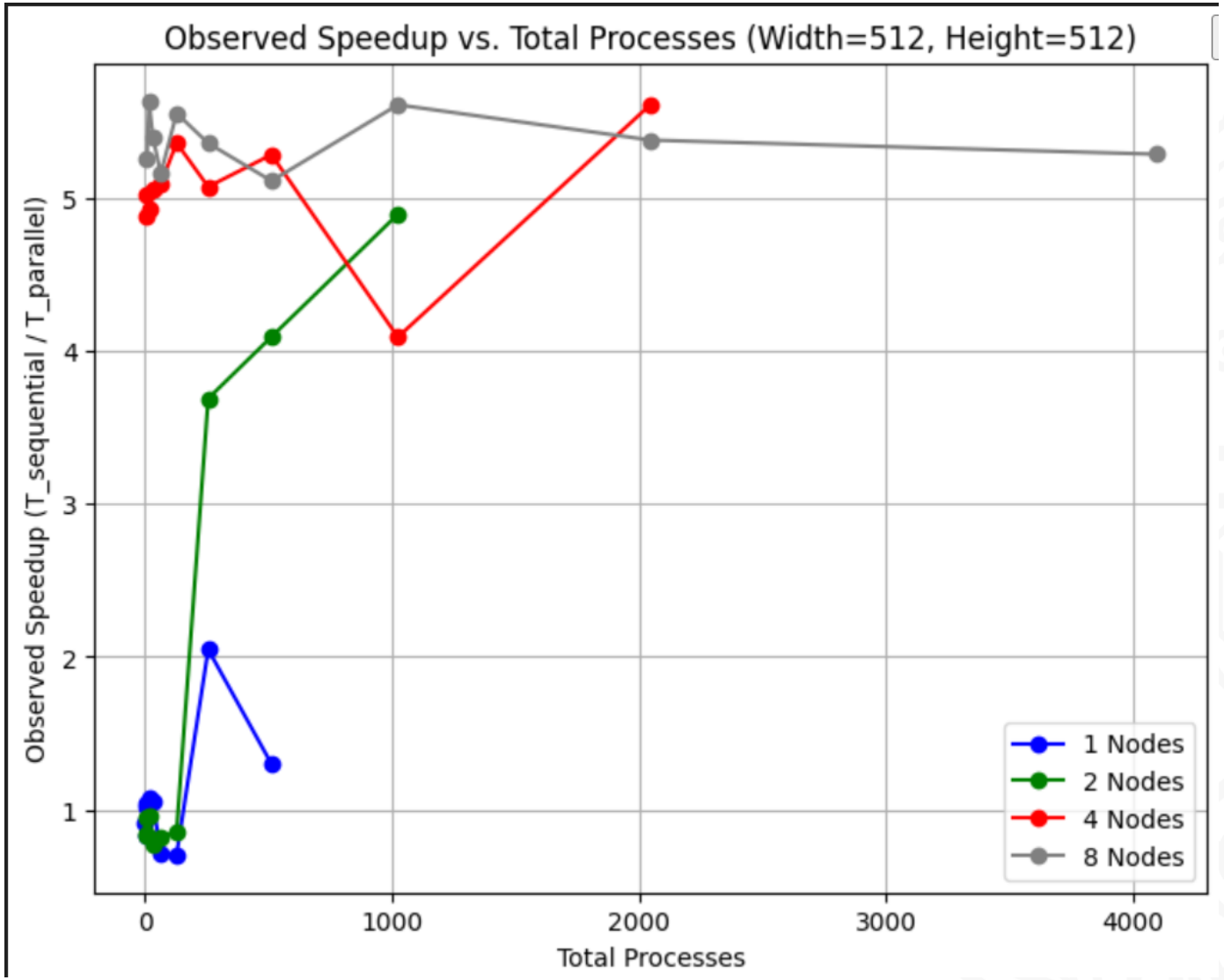
- Complex scene

Sequential Table:

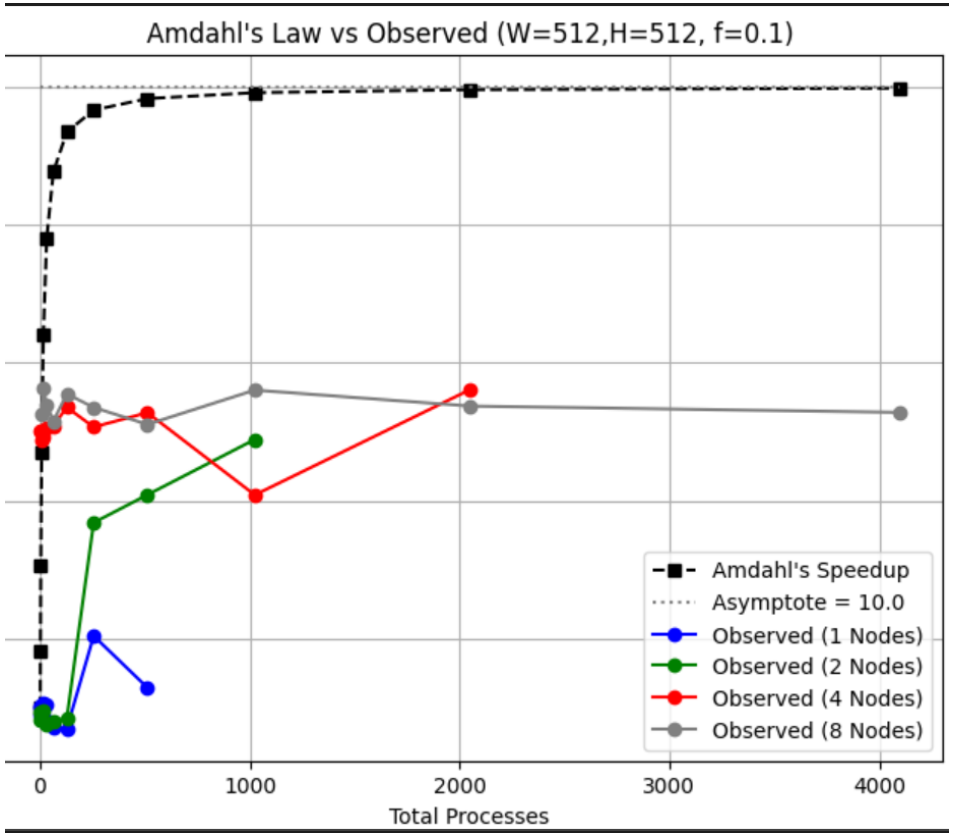
	Width	Height	Execution Time (s)
0	512	512	137.684937

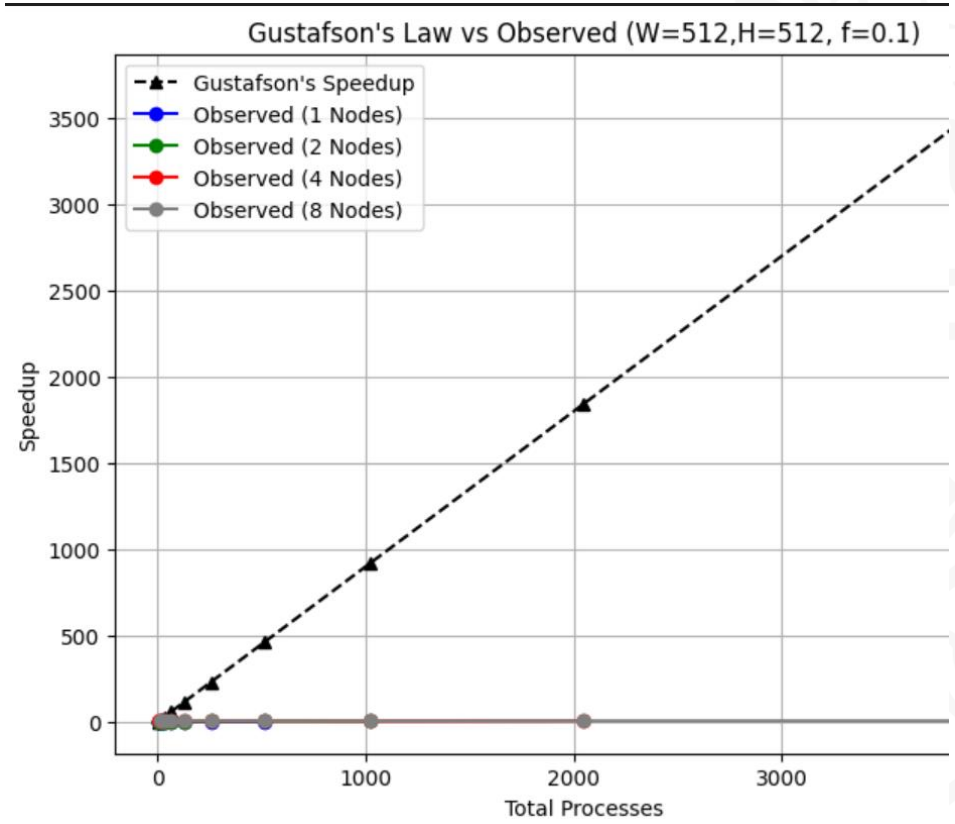
Parallel Table:

	Width	Height	Nodes	Processes/Node	Total Processes	Execution Time (s)
0	512	512	1	1	1	151.409585
1	512	512	1	2	2	135.559252
2	512	512	1	4	4	132.302810
3	512	512	1	8	8	166.304848
4	512	512	1	16	16	128.197487
5	512	512	1	32	32	131.208627
6	512	512	1	64	64	191.811984
7	512	512	1	128	128	195.033434
8	512	512	1	256	256	67.230955
9	512	512	1	512	512	105.921425
10	512	512	2	1	2	144.807324
11	512	512	2	2	4	164.637087
12	512	512	2	4	8	163.298362
13	512	512	2	8	16	144.046878
14	512	512	2	16	32	177.887940
15	512	512	2	32	64	167.932984
...						
36	512	512	8	64	512	26.944391
37	512	512	8	128	1024	24.555116
38	512	512	8	256	2048	25.609209
39	512	512	8	512	4096	26.055134


Observed Speedup vs. Total Processes (Width=512, Height=512)
Observed Speedup (T_sequential / T_parallel)
Total Processes
0
1000
2000
3000
4000
1
2
3
4
5
1 Nodes
2 Nodes
4 Nodes
8 Nodes

Output:


Amdahl's Law vs Observed (W=512,H=512, f=0.1)
Amdahl's Speedup
Asymptote = 10.0
Observed (1 Nodes)
Observed (2 Nodes)
Observed (4 Nodes)
Observed (8 Nodes)
0
1000
2000
3000
4000
Total Processes


Gustafson's Law vs Observed (W=512,H=512, f=0.1)
Gustafson's Speedup
Observed (1 Nodes)
Observed (2 Nodes)
Observed (4 Nodes)
Observed (8 Nodes)
Speedup
0
500
1000
1500
2000
2500
3000
3500
0
1000
2000
3000
Total Processes

Conclusion:

- As demonstrated with Scene 2 (Complex Scene), parallelization excels at handling computationally demanding tasks. The significant workload in complex scenes allows parallel processing to effectively distribute the rendering calculations, leading to substantial performance gains.
- In contrast, Scene 1 (Simple Scene) illustrated the limitations of parallelization for less intensive tasks. The communication overhead associated with managing multiple processes can outweigh the benefits of distributing the relatively small amount of work, resulting in diminished returns or even performance degradation.
- When designing a ray tracing system, it's crucial to consider the complexity of the scenes that will be rendered. Parallelization is better for applications involving complex geometry, numerous light sources, and advanced rendering effects.

Thank You