Scientific Research Publishing

Journal of Computer and Communications, 2024, 12, 120-130
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

The Implementation of Ray Tracing Algorithm with OpenMP Parallelization

Noor Alnasser, Raghad Alabssi, Batool Faran, Latifah Alessa, Naya Nagy

College of Computer Science and IT, Imam Abdulrahman Bin Faisal University, Dammam, Kingdom of Saudi Arabia
Email: 2200004494@iau.edu.sa, 2200002169@iau.edu.sa, 2190002705@iau.edu.sa, 2200004582@iau.edu.sa nmnagy@iau.edu.sa

How to cite this paper: Alnasser, N., Alabssi, R., Faran, B., Alessa, L. and Nagy, N. (2024) The Implementation of Ray Tracing Algorithm with OpenMP Parallelization. *Journal of Computer and Communications*, **12**, 120-130.
https://doi.org/10.4236/jcc.2024.121008

Received: December 4, 2023
Accepted: January 26, 2024
Published: January 29, 2024

Copyright © 2024 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution-NonCommercial International License (CC BY-NC 4.0).
http://creativecommons.org/licenses/by-nc/4.0/

Open Access

Abstract

Ray tracing is a computer graphics method that renders images realistically. As the name suggests, this technique primarily traces the path of light rays interacting with objects in a scene [1], permitting the calculation of lighting and reflecting impact [2]. As ray tracing is a time-consuming process, the need for parallelization to solve this problem arises. One downside of this solution is the existence of race conditions. In this work, we explore and experiment with a different, well-known solution for this race condition. Starting with the introduction and the background section, a brief overview of the topic is followed by a detailed part of how the race conditions may occur in the case of the ray tracing algorithm. Continuing with the methods and results section, we have used OpenMP to parallelize the Ray tracing algorithm with the different compiler directives critical, atomic, and first-private. Hence, it concluded that both critical and atomic are not efficient solutions to produce a good-quality picture, but first-private succeeded in producing a high-quality picture.

Keywords

Parallelization, Ray Tracing, Parallel Computer Architecture, OpenMP

1. Introduction

Consider the prospect of swiftly generating intricate, realistic worlds through computational processes. This is made possible through the help of parallelism in ray tracing. Ray tracing is a technique used in computer graphics that simulates the behavior of real-time rendering [3] [4]. Introducing parallelism has expedited rendering processes, yielding improved visual experiences. Traditionally, ray tracing was time-consuming, often requiring hours or even days to render a single frame (which is one image generated by the ray tracing process). This was

due to the complex calculations involved in tracing rays, determining object intersections, calculating lighting effects, and shading pixels [5]. Nevertheless, the integration of parallelism has significantly decreased rendering times, facilitating the creation of virtual worlds more efficiently [2].

Parallelism in ray tracing leverages the collective power of multiple processors or cores working simultaneously. Processors, or central processing units (CPUs), are the brain of a computer responsible for executing instructions [6]. Conversely, cores are individual processing units within a CPU capable of handling separate tasks concurrently. The overall process becomes significantly faster by dividing the rendering workload into smaller tasks and executing them concurrently across these processors or cores. This concept of parallel execution opens up new dimensions in computer graphics, enabling real-time or near-real-time rendering of highly detailed and dynamic scenes.

One of the techniques used to achieve parallelism is multi-threading, where a program divides its tasks into smaller threads that can be executed simultaneously on different processors or cores. Each thread handles a specific part of the rendering process and can communicate and share data with other threads. This allows for the efficient distribution of the workload and faster completion of rendering tasks [7]. Another avenue for parallelism in ray tracing is the utilization of Graphics Processing Units (GPUs). These specialized processors, designed with numerous cores, excel at handling the intense computational requirements of graphics processing. By leveraging the parallel architecture of GPUs, ray tracing computations can be performed simultaneously, accelerating the rendering process and delivering breath-taking visual results [8].

With parallelism, the possibilities in ray tracing are virtually limitless. Complex scenes with intricate lighting effects, reflections, and shadows can be rendered in real-time, allowing for interactive experiences that push the boundaries of realism. From video games that transport players to magical worlds to movies with awe-inspiring visual effects, parallelism has revolutionized the field of computer graphics, ushering in a new era of immersive digital experiences [9]. Parallelism in ray tracing facilities to a faster rendering time, enabling the creation of realistic graphics. Through the power of multiple processors or cores, multi-threading, and leveraging GPUs, ray tracing has transformed from a time-consuming process to a real-time rendering powerhouse. Prospects for the future abound as parallelism persistently extends the limits of visually attainable outcomes. The impact of parallelism is poised to be revealed in unprecedented ways.

A potential challenge introduced by parallel computing is the race condition. Race conditions occur in parallel computing when multiple jobs or threads access and modify the same data simultaneously, potentially resulting in uncertain or incorrect outcomes. Picture two people attempting to change the same document simultaneously without prior communication. This could lead to alterations or errors in the final text that do not make sense [10].

In ray tracing, race conditions may occur when multiple rays or threads at-

tempt to modify the same piece of data simultaneously, such as a pixel's color [10]. This can result in issues like stuttering, inaccurate colors, or inconsistent shadows, ultimately compromising the quality of the final image [2]. Different processes, including critical, private, and atomic, are employed to mitigate race conditions [2]. These methods guarantee that only one thread can access the shared data at any given time, preserving the output image's structure and preventing issues [2]. Atomic operations execute as a single, indivisible unit, preventing interference from other parallel processes [2].

On the other hand, critical sections are code portions that must be executed by only one thread at a time to prevent concurrent access to shared data [11]. Protecting shared variables with locks or mutexes ensures that only one thread can enter the critical section [12] simultaneously, preventing race conditions [11]. In contrast, the "first private" method involves creating private copies of variables for each thread to ensure that modifications in one thread do not affect others. Ray tracing can be used in many areas, such as construction, virtual reality, games, and movie production.

2. Methods

OpenMP makes shared-memory multiprocessing programming easier, improving the computational power needed for realistic image creation through ray tracing. It facilitates cross-platform shared-memory multiprocessing programming in C, C++, and Fortran, accommodating various platforms, instruction-set architectures, and operating systems such as Solaris, AIX, HP-UX, Linux, macOS, and Windows. Comprising compiler directives, library routines, and environment variables, OpenMP influences runtime behavior. The "#pragma" directive, conforming to the C standard, serves to convey additional information to the compiler beyond the language itself [13]. This directive, considered a special purpose, enables the activation or deactivation of specific features, and its usage is compiler-specific, varying from one compiler to another. In computer graphics, ray tracing is a technique [14] that simulates how light interacts with virtual objects to generate realistic images [2]. Through the parallelization capabilities of OpenMP, the computational demands of ray tracing can be efficiently distributed across multiple processors, enhancing rendering speed and overall performance [2].

The program starts by initializing variables and data structures, Reading the input parameters, and then creating an empty image buffer; that image is divided into smaller blocks. We start the parallel region using OpenMP; inside the parallel region, a for loop is created for each block in the parallel region. We set up the picture parameters, and then inside this for loop, another for loop is created for each pixel in a block. We compute the ray direction and find the closest intersection. If an intersection is found, we compute the shading at the intersection point and update the pixel color in the image buffer. If the intersection is not found, we set the pixel to the background color. After setting the pixel

color in both situations (intersection found and not found), the inner for loop asks if this is the last pixel in the block. If not, it goes back to the start of the loop; if yes, it is the last pixel in a block reached; the outer for loop asks if the last block is reached or not; if the condition is not met, it will go back to the outer for loop, if the last block is reached, We reach the end to the parallel region, the image blocks are merged into the final image, and finally output the image. **Figure 1** demonstrates the flowchart of the ray tracing program with OpenMP.

To execute the code [15] efficiently, we utilized an Intel(R) Core(TM) i7-10510U CPU with 4 cores and 8 logical processors, operating at a clock speed of 1.80 GHz. The code utilizes three crucial variables, directionX, directionY, and directionZ, pivotal to a ray tracing algorithm that defines rays based on their origin and direction [5]. These three variables play a crucial role in calculating the direction of each ray for specific pixels in the rendered image. The methodology involves a detailed breakdown of:

How these direction variables are utilized:

1) directionX: Calculated as (i + 0.5) − rows/2., where i is the current pixel's X coordinate. This centers the rays horizontally, directing them towards the image center.

2) directionY: Calculated as −(j + 0.5) + cols/2., where j is the current pixel's Y coordinate. This flips the image vertically, ensuring rays are directed towards the image center.

3) directionZ: Calculated as −cols/(2. * tan(fov/2.)), where cols is the image height and fov is the field of view. This determines the depth of rays in the scene, aligning them appropriately based on the field of view.

After computing the direction components, a Vec3f vector is normalized using the normalize() method on the Vec3 f (directionX, directionY, directionZ) vector. This vector represents the normalized direction of the ray. Subsequently, the cast ray function is invoked with the origin (0, 0, 0) and the calculated direction vector to determine pixel properties. The results are stored in the image array at the corresponding index. To enhance computational efficiency, the code incorporates OpenMP (#pragma omp parallel for) for parallelizing the ray tracing algorithm, allowing concurrent computation of multiple rays for different pixels [5].

Moreover, the methodology encompasses the resolution of obstacles of race conditions in variables that are globally specified. Our primary objective was to comprehend race conditions' effect on images' visual quality. We systematically assessed numerous solutions utilizing varying quantities of processing threads to achieve this. Additionally, this project examines how the number of threads impacts the time the program runs on a 4-core device. The effectiveness of our parallelization approach is clear even with race conditions present, and the code runs fastest when using eight threads. This highlights our systematic approach to addressing race conditions, evaluating visual results, and optimizing performance with different thread counts and core setups. The three solutions to the

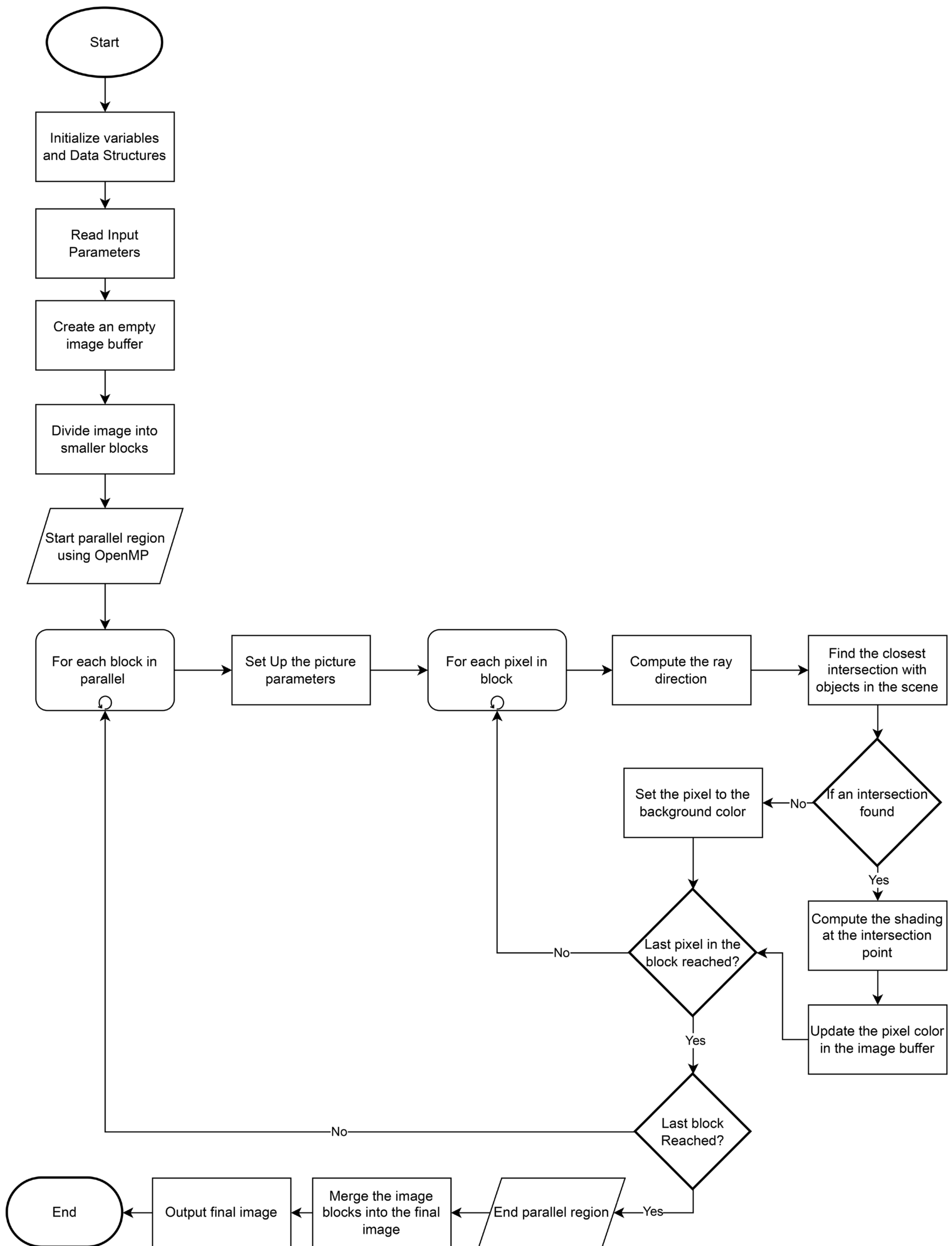

Figure 1. Parallel ray tracing with OpenMP.

race conditions issue will be introduced in more detail in the next section.

3. Results

The Results of our parallel solutions to have best picture results and with no race conditions using OpenMP compiler directives in a C++ programming language were as follows: we calculated the execution time of the parallel region of our code that is in Figure 2, and the resulting Picture of the race condition is in Figure 3 now, applying the solutions, first, we applied a critical compiler directive solution, and we found that the time of executing the parallel solution with critical on the default number of threads (that is, without forcing the threads to have a specific number using the omp set num threads()), does not reduce the execution time, Figure 4 nor that it provides a picture with no race condition (race condition, in our case, indicates the noise and the many dots in the Picture). Figure 5 means that the critical solution is inefficient for solving the race

```
PS C:\cygwin64\home\Noorf> g++ -fopenmp ParallelRaytracingWithOpenMP.cpp -o ParallelRaytracing
PS C:\cygwin64\home\Noorf> ./ParallelRaytracing
Parallel RayTracing is Running....
Raytracing Algorithm Started..........
Raytracing Algorithm Finished......
Time required to run ray tracing algorithm in parallel ray tracing program.....5.80742 seconds.
```

Figure 2. Execution time of the code having a race condition.

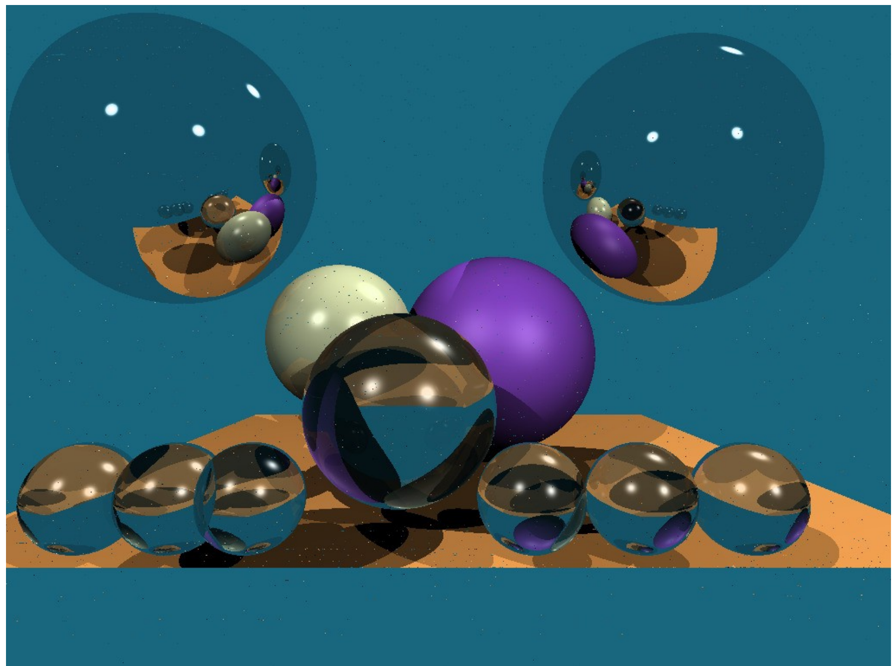

Figure 3. Output picture of the race condition.

```
PS C:\cygwin64\home\Noorf> g++ -fopenmp ParallelRaytracingWithOpenMP.cpp -o ParallelRaytracing
PS C:\cygwin64\home\Noorf> ./ParallelRaytracing
Parallel RayTracing is Running....
Raytracing Algorithm Started..........
Raytracing Algorithm Finished......
Time required to run ray tracing algorithm in parallel ray tracing program.....5.81585 seconds.
```

Figure 4. Execution time of critical.

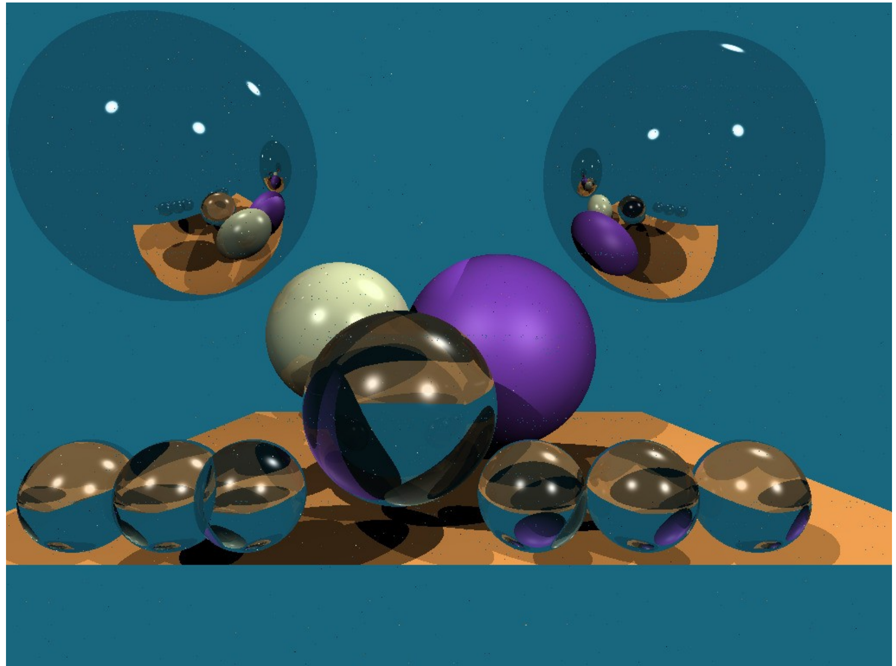

Figure 5. Resulted picture after applying the critical solution.

```
PS C:\cygwin64\home\Noorf> g++ -fopenmp ParallelRaytracingWithOpenMP.cpp -o ParallelRaytracing
PS C:\cygwin64\home\Noorf> ./ParallelRaytracing
Parallel RayTracing is Running....
Raytracing Algorithm Started..........
Raytracing Algorithm Finished......
Time required to run ray tracing algorithm in parallel ray tracing program.....6.05409 seconds.
```

Figure 6. Execution time of atomic.

condition problem with the Ray tracing algorithm. Furthermore, we have implemented a solution that uses an atomic solution. As critical, it did not reduce the execution time Figure 6 or the noise in the Picture, as demonstrated in Figure 7 indicating that we need to find another solution to this problem. Last but not least, we have used the first-private solution to ensure that the variable that gets copied is not assigned wrong values. Even though it did not reduce the execution time in Figure 8 with the default number of threads, it gave us a nice picture with no noise in it, as displayed in Figure 9 outstanding to be the best solution in our case, considering the number of cores, which is four that we have used, a comparison of the execution time of the code that produces race conditions with the other solutions, is displayed in Figure 10.

We have made a comparison between the execution time of the code having race conditions with the code of the best solution that is with firstprivate, and the results appeared as follows in Figure 11 and Figure 12.

4. Discussion

This project aims to resolve issues associated with using the ray tracing algorithm in conjunction with OpenMP parallelization. Implementing OpenMP directives and race conditions within the algorithm presented considerable obstacles that

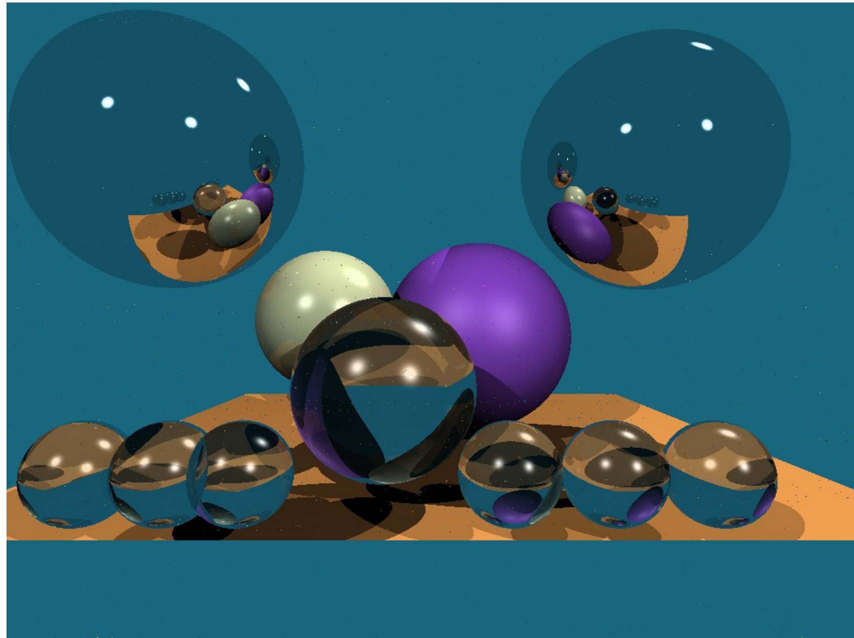

Figure 7. Resulted picture of atomic solution.

```
PS C:\cygwin64\home\Noorf> g++ -fopenmp ParallelRaytracingWithOpenMP.cpp -o ParallelRaytracing
PS C:\cygwin64\home\Noorf> ./ParallelRaytracing
Parallel RayTracing is Running....
Raytracing Algorithm Started..........
Raytracing Algorithm Finished......
Time required to run ray tracing algorithm in parallel ray tracing program.....5.94563 seconds.
```

Figure 8. Execution time of firstprivate.

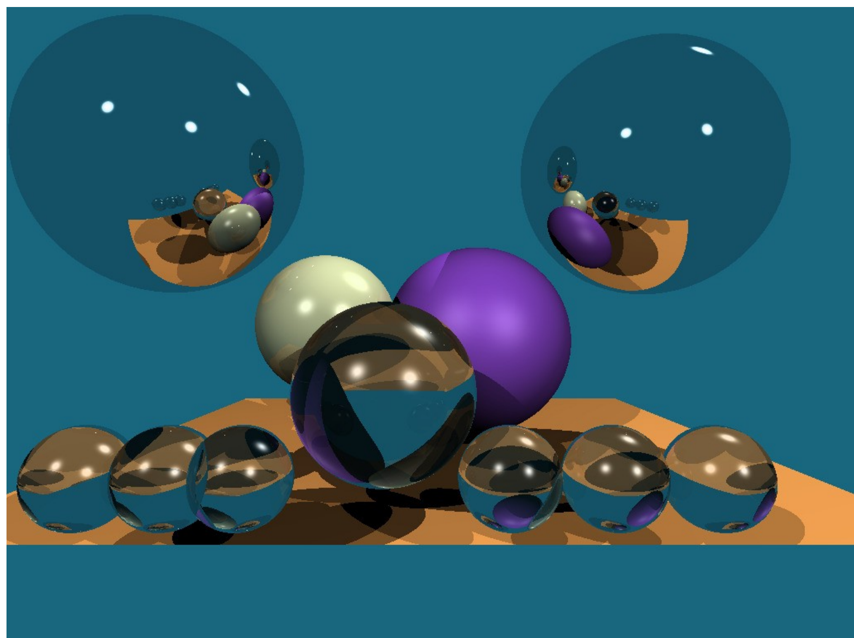

Figure 9. Resulted picture of the firstprivate solution.

needed to be addressed. The results indicate neither the critical nor the atomic compiler directive solutions reduced the execution time nor did they provide a noise-free image; on the other hand, the first private solution was found to be

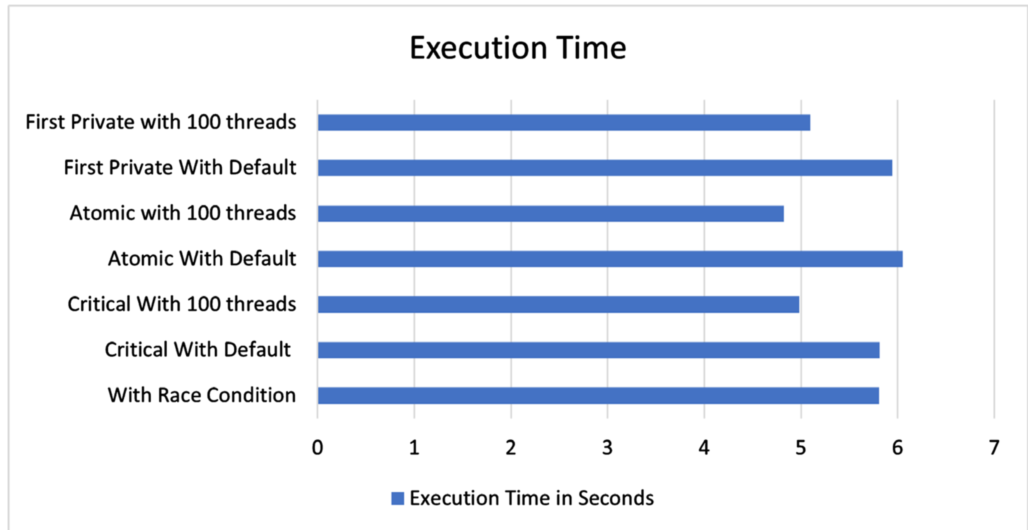

Figure 10. Execution time of all the solutions as well as race condition, all in seconds.

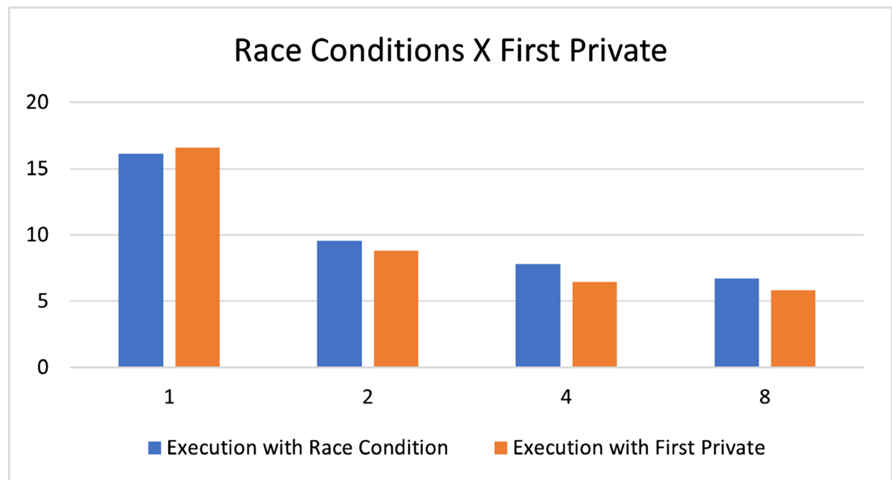

Figure 11. Comparison made with 1, 2, 4, and 8 threads.

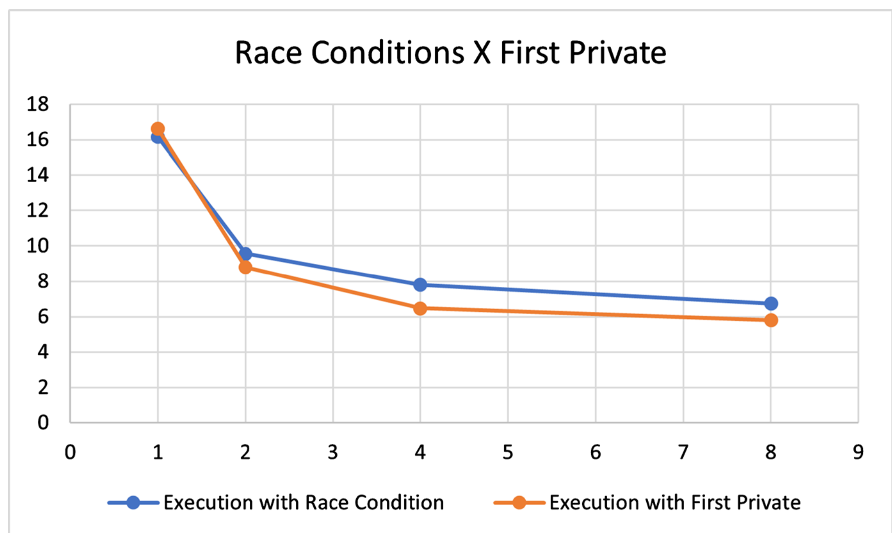

Figure 12. Comparison made with 1, 2, 4, and 8 threads.

the one that produced the best results among all.

The research findings clearly correlate the quantity of threads utilized and the execution duration. This confirmed that augmenting the number of threads reduces execution time, which is consistent with the generally acknowledged principle of employing parallelism.

In this project, we were limited to the execution of only 3 race condition solutions: critical, atomic, and first-private; it was not feasible to execute.

A 4th solution is the reduction, as the equations used in the code are complex, and this solution is only available to a specific operation. Also, some other possible solutions, such as locks, mutex, and data synchronization strategies, were not explored; furthermore, due to this limitation, we can conclude that it is not possible to determine what would be the possible effect on both the execution time and the image noise if these solutions were applied to the presented problem. Also, the presented results may vary.

As all of the presented explorations and findings in this project are based on executing a 4-core device with different thread counts, the solution's scalability to a larger system and a higher number of cores was not discussed.

Acknowledgments

We thank Naya Nagy for her invaluable guidance and support throughout this research. Her expertise and feedback have been crucial in shaping the direction of our study. Additionally, we extend our appreciation to our university, Imam Abdulrahman Bin Faisal (IAU), especially our College of Computer Science and Information Technology (CCSIT), for providing us with a valuable and informative course on Parallel Computer Architecture. In conclusion, completing this research was only possible with the help of Naya Nagy and the colleagues who participated in the success of this project.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Nicks, S. (2023, February 17). RTX Cards Information and Review 2023. StoneAgeHacks. https://stoneagehacks.com/rtx/

[2] Pharr, M. and Humphreys, G. (2016) Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann, Burlington.

[3] Breitenmoser, S. (2023) The Ray Tracing Update Is Finally Coming to Elden Ring. EarlyGame. https://earlygame.com/gaming/ray-tracing-update-elden-ring

[4] Geeks, L. (2023) Ray Tracing Techniques: A Comprehensive Guide for Stunning Visuals. https://lambdageeks.com/ray-tracing-techniques/

[5] Pearsoncmg.com (n.d.). https://ptgmedia.pearsoncmg.com/images/9780321399526/samplepages/0321399528.pdf

[6] Ahmad, S. (2023, February 17) Technology. https://expertsadvices.net/technology/page/3/

[7] Kirvan, P. (2022, May 26) What Is Multithreading? WhatIs. https://www.techtarget.com/whatis/definition/multithreading

[8] Awati, R., Gillis, A.S. and Steele, C. (n.d.) Graphics Processing Unit (GPU). https://www.techtarget.com/searchvirtualdesktop/definition/GPU-graphics-processing-unit

[9] Sanderson, J. (2023, September 6) The Evolution of Metaverse Gaming and Its Impact. Textually. https://textually.org/the-evolution-of-metaverse-gaming-and-its-impact/

[10] Larus, J. and Kozyrakis, C. (2008) Transactional Memory. *Communications of the ACM*, **51**, 80-88. https://doi.org/10.1145/1364782.1364800

[11] Slusallek, C. and Daniel, P. (2003) PVG 2003 (Parallel and Large-Data Visualization and Graphics). *Computers & Graphics*, **27**, 662. https://doi.org/10.1016/S0097-8493(03)00098-0

[12] User (n.d.) Java Core Technology Volume 18. Java Concurrency. https://topic.alibabacloud.com/a/java-core-technology-volume-18-java-concurrency_1_27_30242925.html

[13] Trobec, R. (2018) Introduction to Parallel Computing: From Algorithms to Programming on State. Springer International Publishing. https://doi.org/10.1007/978-3-319-98833-7

[14] TNW (n.d.) Ray Tracing (Graphics) News. https://thenextweb.com/topic/ray-tracing-graphics

[15] Manjunath, A. (n.d.) Adarshkoppmanjunath/Raytracing. GitHub. https://github.com/AdarshKoppManjunath/Raytracing