UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CPA, Practice 2 Deliverable

11549, Parallel Computing - 2023

Group 2E2

Authors:

Ramis Gutiérrez, Marc

Jiménez Ríos, Francisco Javier

Tutor: Alvarruiz Bermejo, Fernando

Valencia, November 29, 2023

Contents

0 | Introduction

This report endeavors to delve into the analysis and enhancement of performance within parallel versions of a ray tracing program through a series of four exercises. The primary objective is to leverage OpenMP for optimizing the computational efficiency of this program. Throughout these exercises, the focus is placed on enhancing parallelism within the codebase. The exercises encompassed tasks such as parallelizing inner and outer loops, evaluating the impact of different scheduling methodologies, and assessing performance alterations concerning varying thread counts.

1 | Exercise 1: Parallelize Inner Loop (rayp1.c)

The objective of this exercise is to parallelize the inner loop (x loop) within the 'render' function to enhance the efficiency of image generation. OpenMP directives and clauses are employed to effectively utilize parallel processing capabilities.

```
for (y = 0; y < height; ++y) {
    ncalls_line = 0;
    #pragma omp parallel for reduction(+:ncalls_line) private(xx,yy,pixel,raydir,ncalls)
        schedule(runtime)
    for (x = 0; x < width; ++x) {
        pixel = &image[y * width + x];
        xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectratio;
        yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
        Vec3_new(&raydir, xx, yy, -1);
        Vec3_normalize(&raydir);
        ncalls = trace(pixel, &origin, &raydir, size, spheres, 0);
        ncalls_line += ncalls;
    }
    ncalls_line = ncalls_line / width;
    if (ncalls_line < min_ncalls_line)
        min_ncalls_line = ncalls_line;
    if (ncalls_line > max_ncalls_line)
        max_ncalls_line = ncalls_line;
    update_histogram(histo, ncalls_line);
}
```

Listing 1: Parallelized Loop rayp1.c

- **Parallelization Strategy**
  - Workload distribution among threads assigns each thread the task of computing a portion of the line's pixels.
  - Utilizes the 'reduction' clause to aggregate 'ncalls_line' values computed by individual threads, ensuring accurate results.

- **Improvements Due to Parallelization**
  - Significant enhancement in computational efficiency, particularly noticeable for larger dimension images.
  - Capitalizes on parallel processing capabilities within the inner loop, resulting in a substantial speedup in image generation.

- **Conclusion on Optimization Strategy**
  - Successful optimization strategy demonstrated by the observed speedup in image generation.
  - Effective workload distribution among threads and reduction strategy maximize OpenMP's parallel processing capabilities.
  - Particularly beneficial for scenarios with extensive image dimensions, managing substantial computational workloads efficiently.

- **Impact of Parallelization**
  - Resultant images vividly exhibit the positive impact of parallelizing the inner loop.
  - Validates the approach and highlights efficiency gains achieved through parallel processing techniques.

Calls/pixel	Number of lines
[1.0 – 2.5[	201 **********
[2.5 – 4.0[	57 **********
[4.0 – 5.5[	116 **********
[5.5 – 7.0[	137 **********
[7.0 – 8.5[	158 **********
[8.5 – 10.0[	99 **********
[10.0 – +inf[	0

Table 1.1: Complexity histogram ray.c

Least complex line had 1.000 calls to trace per pixel,
Most complex line had 9.816 calls to trace per pixel,
Time: 1.773439

Calls/pixel	Number of lines
[1.0 – 2.5[	201 **********
[2.5 – 4.0[	57 **********
[4.0 – 5.5[	116 **********
[5.5 – 7.0[	137 **********
[7.0 – 8.5[	158 **********
[8.5 – 10.0[	99 **********
[10.0 – +inf[	0

Table 1.2: Complexity histogram rayp1.c

Least complex line had 1.000 calls to trace per pixel,
Most complex line had 9.816 calls to trace per pixel,
Time: 0.400684

2 | Exercise 2: Parallelize Outer Loop (rayp2.c)

The aim of task is to make the inner loop (y loop) in the render function run in parallel to enhance image generation speed using OpenMP directives.

```
#pragma omp parallel for private(x, pixel, xx, yy, raydir, ncalls, ncalls_line) schedule(
    runtime)
    for (y = 0; y < height; ++y)
    {
        ncalls_line = 0;
        for (x = 0; x < width; ++x)
        {
            pixel = &image[y * width + x];
            xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectratio;
            yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
            Vec3_new(&raydir, xx, yy, -1);
            Vec3_normalize(&raydir);
            ncalls = trace(pixel, &origin, &raydir, size, spheres, 0);
            ncalls_line += ncalls;
        }
        ncalls_line = ncalls_line / width;
        if (ncalls_line < min_ncalls_line)
        {
            #pragma omp critical(min)
            if (ncalls_line < min_ncalls_line)
                min_ncalls_line = ncalls_line;
        }
        if (ncalls_line > max_ncalls_line)
        {
            #pragma omp critical(max)
            if (ncalls_line > max_ncalls_line)
                max_ncalls_line = ncalls_line;
        }
        #pragma omp critical(histo)
        update_histogram(histo, ncalls_line);
    }
}
```

Listing 2: Parallelized Loop rayp2.c

- **Precautionary Measures**
  - Introduced *ncalls_line* as a private variable to independently calculate sums for individual lines by each thread.
  - Implemented protective measures to prevent conflicts in accessing global variables (*max_ncalls_line* and *min_ncalls_line*).
  - Ensured data integrity by securing the function call responsible for updating variables to prevent conflicts during simultaneous access to shared resources.

- **Performance Improvements**
  - Noticeable reduction in execution time compared to the original and preceding versions.
  - Parallelizing the outer loop allows threads to compute entire lines rather than individual components, contributing to a substantial speedup in overall performance.
  - Enhanced workload distribution among threads significantly optimizes the program's efficiency.

- **Focus on Further Optimization**
  - Future focus on parallelizing the outer loop to optimize computational resource utilization and improve overall efficiency and speed.
  - Strategic parallelization of the outer loop distributes tasks among processing units, reducing idle times and enhancing overall performance.

- Refined version leverages parallel processing capabilities for a simplified and effective computational workflow.

Calls/pixel	Number of lines
[1.0 – 2.5[	201 **********
[2.5 – 4.0[	57 **********
[4.0 – 5.5[	116 **********
[5.5 – 7.0[	137 **********
[7.0 – 8.5[	158 **********
[8.5 – 10.0[	99 **********
[10.0 – +inf[	0

Table 2.1: Complexity histogram rayp2.c

Least complex line had 1.000 calls to trace per pixel,
Most complex line had 9.816 calls to trace per pixel,
Time: 0.400684

3 | Exercise 3: Evaluate Scheduling Influence

3.1 | Results Analysis

Program: rayp1

Execution Time:

- Static: 9.83 seconds
- Static with chunk size 1: 4.47 seconds
- Dynamic: 3.11 seconds

Program: rayp2

Execution Time:

- Static: 5.66 seconds
- Static with chunk size 1: 4.09 seconds
- Dynamic: 3.17 seconds

3.2 | Comparing Schedule Choices

For Program rayp1:

- **Best Schedule Choice:** Dynamic (3.11 seconds)
- **Worst Schedule Choice:** Static (9.83 seconds)
- **Justification:** The dynamic schedule outperforms the others, likely due to its ability to adapt workload distribution among threads during runtime, enhancing load balance and reducing idle time.

For Program rayp2:

- **Best Schedule Choice:** Dynamic (3.17 seconds)
- **Worst Schedule Choice:** Static (5.66 seconds)
- **Justification:** Similar to rayp1, dynamic scheduling leads to better performance, possibly due to better load balancing, while the static scheduling methods result in increased execution times.

3.3 | Load Balance and Histogram:

Dynamic scheduling seems to enable better load balance, reducing idle time for threads compared to static scheduling. This could be reflected in the histogram produced by the program, showing more evenly distributed workload across threads.

Objective:

The study evaluated the influence of different iteration scheduling methods on the performance of parallel programs (rayp1 and rayp2) utilizing 32 threads each.

Methodology:

Using the Kahan cluster queuing system, three scheduling methods—static, static with chunk size 1, and dynamic—were implemented for each program. Execution times were recorded for analysis.

Results and Analysis:

The results indicate significant performance variations based on scheduling methods. For both rayp1 and rayp2, dynamic scheduling consistently outperformed static scheduling methods. Dynamic scheduling exhibited the shortest execution times, attributed to its adaptive workload distribution among threads, resulting in better load balance. Conversely, static scheduling showed increased execution times due to potential load imbalance among threads.

Conclusion:

The dynamic scheduling method proved to be the most efficient for both rayp1 and rayp2 programs in terms of reducing execution times. This was largely due to its ability to maintain better load balance among threads. The static scheduling methods, especially with default settings, exhibited longer execution times, indicating potential load imbalance.

Recommendation:

Based on these findings, we recommend adopting dynamic scheduling for parallel programs to optimize performance by ensuring better load balance among threads and reducing overall execution time.

Calls/pixel	Number of lines
[1,0 – 2,5[	194 **********
[2,5 – 4,0[	34 *********
[4,0 – 5,5[	143 **********
[5,5 – 7,0[	120 **********
[7,0 – 8,5[	198 **********
[8,5 – 10,0[	46 **********
[10,0 – +inf[	33 ********

Table 3.1: Histogram rayp1

Least complex line had 1,000 calls to trace per pixel,
Most complex line had 11,489 calls to trace per pixel,
Time: 3.111816

Calls/pixel	Number of lines
[1,0 – 2,5[	194 **********
[2,5 – 4,0[	34 *********
[4,0 – 5,5[	143 **********
[5,5 – 7,0[	120 **********
[7,0 – 8,5[	198 **********
[8,5 – 10,0[	46 **********
[10,0 – +inf[	33 ********

Table 3.2: Histogram rayp2

Least complex line had 1,000 calls to trace per pixel,
Most complex line had 11,489 calls to trace per pixel,
Time: 3.168285

Table 3.3: Execution times for different programs with varying schedules

Program	Execution Time (seconds)	Threads	Schedule
rayp1	9.83	32	static
rayp1	4.47	32	static,1
rayp1	3.11	32	dynamic
rayp2	5.66	32	static
rayp2	4.09	32	static,1
rayp2	3.17	32	dynamic

3.4 | Shell code: *sbatch job.sh*

```
gcc -Wall -fopenmp -o rayp1 ~/W/Delivery/rayp1.c -lm
gcc -Wall -fopenmp -o rayp2 ~/W/Delivery/rayp2.c -lm
OMP_NUM_THREADS=32 OMP_SCHEDULE="static" ./rayp1 600
OMP_NUM_THREADS=32 OMP_SCHEDULE="static,1" ./rayp1 600
OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp1 600

OMP_NUM_THREADS=32 OMP_SCHEDULE="static" ./rayp2 600
OMP_NUM_THREADS=32 OMP_SCHEDULE="static,1" ./rayp2 600
OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp2 600
```

Listing 3: Shell file

These commands are compiling two programs, `rayp1` and `rayp2`, using the GCC compiler with flags for OpenMP support and then executing them with different OpenMP environment settings and with 600 circles

- `gcc -Wall -fopenmp -o rayp1 /W/Delivery/rayp1.c -lm`: This compiles the `rayp1.c` source file with OpenMP support and links the math library (`-lm` flag) to generate an executable named `rayp1`.
- `gcc -Wall -fopenmp -o rayp2 /W/Delivery/rayp2.c -lm`: Similar to the previous command, this compiles the `rayp2.c` source file with OpenMP support, linking the math library, and generates an executable named `rayp2`.
- `OMP_NUM_THREADS=32 OMP_SCHEDULE="static" ./rayp1 600`: This command sets the number of threads to 32 (`OMP_NUM_THREADS=32`) and the scheduling policy to "static" (`OMP_SCHEDULE="static"`) for the `rayp1` program. It executes `rayp1` with 600 circles
- `OMP_NUM_THREADS=32 OMP_SCHEDULE="static,1" ./rayp1 600`: Similar to the previous command but know it is not default, it is executed with 1 chunk (static,1") for `rayp1`.
- `OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp1 600`: This command sets the scheduling policy to "dynamic" (`OMP_SCHEDULE="dynamic"`) for the `rayp1` program while keeping the number of threads at 32. It executes `rayp1` with 600 circles

These commands seem to test different scheduling policies (static, dynamic) with different parameters (if applicable) and the same number of threads (32) for both `rayp1` and `rayp2` programs, likely to observe their performance characteristics under various OpenMP scheduling settings.

4 | Exercise 4: Varying Threads and Optimal Scheduling

4.1 | Observations

Speed-Up and Efficiency Trends:

- Both p2 and p1 exhibit an initial increase in speed-up as the number of threads grows. However, beyond a certain point, this increase plateaus or decreases, suggesting diminishing returns with more threads.
- Efficiency decreases as thread count increases, indicating inefficiencies due to increased overheads in synchronization and communication among threads.

Comparison between p2 and p1:

- Initially, both versions (p2 and p1) show similar improvements in performance with increasing thread counts.
- However, at higher thread counts (64 and 128), p2 demonstrates a decline in speed-up and efficiency compared to p1.

4.2 | Conclusion

Based on the data and analysis:

1. **Similar Performance Gains Initially:** Both p2 and p1 versions demonstrate comparable improvements in speed-up and efficiency as thread counts increase up to a certain point.
2. **Decline in p2's Performance at Higher Thread Counts:** At 64 and 128 threads, p2 experiences a noticeable decline in speed-up and efficiency compared to p1. This decline suggests that p2 might struggle with higher thread counts, potentially due to increased contention for resources or inefficiencies in its parallelization strategy.
3. **Optimal Performance for p1:** p1 appears to handle higher thread counts more effectively, maintaining better speed-up and efficiency compared to p2, especially at 64 and 128 threads.

4.3 | Justification for Maximum Thread Consideration

The decision to consider a maximum of 128 threads aligns with the cluster's capabilities and scalability testing. However, the decrease in performance observed at 128 threads indicates that the hardware (likely optimized for a maximum of 64 threads) reaches its limit beyond this point. It highlights the importance of considering hardware limitations when scaling parallel programs, as excessive thread counts can lead to diminished performance due to resource contention and inefficiencies in thread management.

Nthreads	p2	p1	p0	SpeedUp p2	Efficiency p2	SpeedUp p1	Efficiency p1
2	38.72	38.73	77.16	1.993	0.996	1.992	0.996
4	19.54	19.50	77.16	3.949	0.987	3.957	0.989
8	9.91	9.89	77.16	7.788	0.973	7.802	0.975
16	5.92	5.91	77.16	13.030	0.814	13.060	0.816
32	3.17	3.11	77.16	24.353	0.761	24.795	0.775
64	2.70	2.70	77.16	28.593	0.447	28.540	0.446
128	2.94	2.99	77.16	26.210	0.205	25.810	0.202

Table 4.1: Execution times, Speed-Up, and Efficiency for varying threads

4.4 | Charts

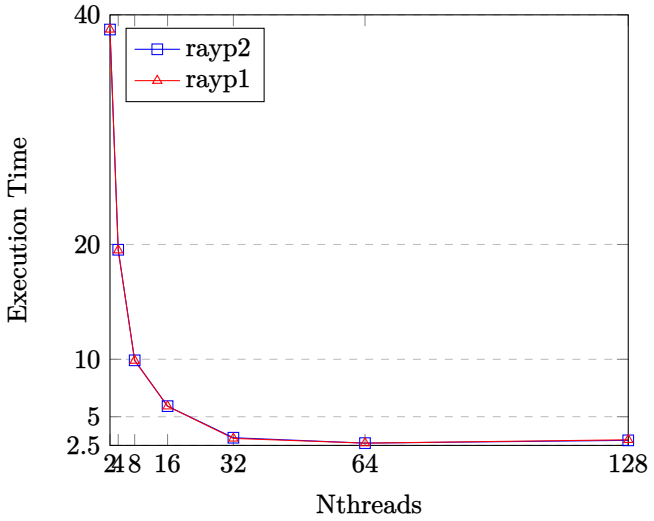

Figure 4.1: Execution Time vs. Nthreads

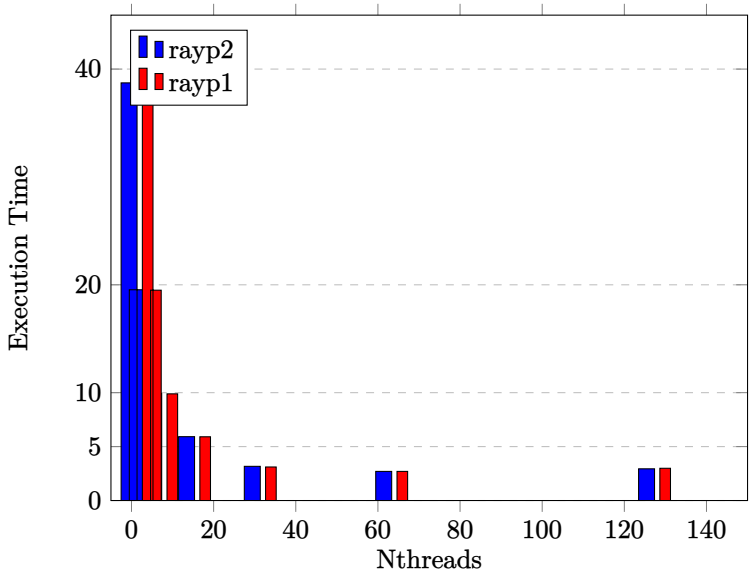

Figure 4.2: Execution Time vs. Nthreads (Bar Chart)

4.5 | Shell code: *sbatch job.sh*

```
gcc -Wall -fopenmp -o rayp1 ~/W/Delivery/rayp1.c -lm
gcc -Wall -fopenmp -o rayp2 ~/W/Delivery/rayp2.c -lm
OMP_NUM_THREADS=2 OMP_SCHEDULE="dynamic" ./rayp1 600
OMP_NUM_THREADS=4 OMP_SCHEDULE="dynamic" ./rayp1 600
OMP_NUM_THREADS=8 OMP_SCHEDULE="dynamic" ./rayp1 600
OMP_NUM_THREADS=16 OMP_SCHEDULE="dynamic" ./rayp1 600
OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp1 600
OMP_NUM_THREADS=64 OMP_SCHEDULE="dynamic" ./rayp1 600
OMP_NUM_THREADS=128 OMP_SCHEDULE="dynamic" ./rayp1 600

OMP_NUM_THREADS=2 OMP_SCHEDULE="dynamic" ./rayp2 600
OMP_NUM_THREADS=4 OMP_SCHEDULE="dynamic" ./rayp2 600
OMP_NUM_THREADS=8 OMP_SCHEDULE="dynamic" ./rayp2 600
OMP_NUM_THREADS=16 OMP_SCHEDULE="dynamic" ./rayp2 600
OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp2 600
OMP_NUM_THREADS=64 OMP_SCHEDULE="dynamic" ./rayp2 600
OMP_NUM_THREADS=128 OMP_SCHEDULE="dynamic" ./rayp2 600
```

Listing 4: Shell file

These lines are commands that set the number of threads and the scheduling policy for OpenMP and then execute two different programs (`rayp1` and `rayp2`) with varying thread counts and with 600 circles

- **OMP_NUM_THREADS**: This environment variable sets the number of threads that OpenMP will use for parallel execution.
- **OMP_SCHEDULE**: This environment variable sets the scheduling policy for distributing loop iterations among the threads. In this case, it's set to "dynamic," which means that the iterations are dynamically divided among threads at runtime.

The commands are executed for different thread counts (2, 4, 8, 16, 32, 64, and 128) for both `rayp1` and `rayp2` programs, likely to observe how the execution time varies with increasing thread counts using a dynamic scheduling policy. The specified 600 circles indicate the numbers of circles created.

5 | Exercise 5: Threads, cluster

In parallel computing, the relationship between the number of threads and performance isn't always linear. It's not uncommon for doubling the number of threads to not result in a similar doubling of performance due to various factors such as resource contention, overheads, and hardware limitations.
In your case, the Kahan cluster has a maximum capacity of 64 threads. When you run the program with 64 threads, the performance might be optimized to efficiently utilize the available hardware resources. The system might efficiently distribute tasks among these 64 threads, ensuring better load balance and minimizing overheads related to synchronization and communication between threads.
However, when you scale up to 128 threads, it's beyond the physical capacity of the hardware. This can lead to performance degradation due to several reasons:

- **Resource Contention**: With 128 threads, there might be a contention for limited resources like CPU time, memory, or I/O bandwidth, causing performance bottlenecks.
- **Synchronization Overheads**: More threads can lead to increased overheads related to synchronizing their actions, leading to inefficiencies in task coordination.
- **Hardware Limitation**: The hardware itself might not efficiently handle such a high number of threads, resulting in degraded performance beyond its optimal capacity.

Therefore, the discrepancy in performance between 64 and 128 threads could be due to the hardware's limitations and the inefficiency introduced by trying to exceed the system's optimal thread capacity. It's essential to consider the hardware's capabilities when determining the optimal number of threads for parallel execution to achieve the best performance.

6 | Exercise 6: Enhanced Profiling and Line Identification (rayp3.c)

The objective of this exercise is to augment the profiling capabilities of the ray tracer program (rayp3.c) by incorporating code to display the number of function calls to 'trace' for each thread. Additionally, the program is enhanced to identify and print the most complex and least complex lines processed during execution.

- **Array Implementation for 'trace' Function Calls**
  - Introduction of an array to store the number of 'trace' function calls made by each thread, associating each index of the array with a specific thread.
  - Utilization of 'max_ncalls_line' and 'min_ncalls_line' variables to track and print the index of the 'y' loop corresponding to the most and least complex lines.
  - Implementation of critical sections ('pragma omp critical') to ensure correctness when updating 'max_ncalls_line', 'min_ncalls_line', and when calling the 'update_histogram()' function, preventing race conditions and maintaining accuracy in profiling.

- **Dynamic Schedule and Workload Distribution**
  - Implementation of a dynamic schedule enabling threads to transition quickly between lines, ensuring a balanced workload distribution and minimizing idle times for optimized parallel execution.

- **Identifying Complex and Simple Lines**
  - Identification of line complexity based on the number of reflective spheres; more spheres indicate complexity, while fewer spheres denote simplicity.
  - Correlation analysis between the most and least complex lines and the threads executing the most and least 'trace' calls, providing valuable insights into program behavior.

- **Analysis and Performance Strategies**
  - Detailed analysis providing insights into the ray tracer program's parallel behavior, informing strategies for better performance in scenarios with varying workloads.
  - Monitoring workload distribution among threads and assessing the impact of different schedule types, observing connections between complex/less complex lines and threads with high/low 'trace' function calls.

Computing... Thread 4: 1002360 Thread 5: 848943 Thread 6: 621789 Thread 7: 202755 Thread 8: 283547 Thread 2: 665351 Thread 3: 808826 Thread 1: 544438

Calls/pixel	Number of lines
[1,0 – 2,5[	194 **********
[2,5 – 4,0[	34 *********
[4,0 – 5,5[	143 **********
[5,5 – 7,0[	120 **********
[7,0 – 8,5[	198 **********
[8,5 – 10,0[	46 **********
[10,0 – +inf[	33 ********

Table 6.1: Histogram rayp3

Least complex line had 1.000 calls to trace per pixel. And the number of the line is 711. Most complex line had 11.489 calls to trace per pixel. And the number of the line is 392. Time: 16.705725

```
#pragma omp parallel private(x, ncalls_line, pixel, xx, yy, ncalls, raydir, counter)
    {
        counter = 0;
        #pragma omp for schedule(dynamic)
        for (y = 0; y < height; ++y)
        {
            ncalls_line = 0;
            for (x = 0; x < width; ++x)
            {
                pixel = &image[y * width + x];
                xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectratio;
                yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
                Vec3_new(&raydir, xx, yy, -1);
                Vec3_normalize(&raydir);
                counter++;
                ncalls = trace(pixel, &origin, &raydir, size, spheres, 0);
                ncalls_line += ncalls;
                counter += ncalls;

            }
            // Each thread indicates the number of calls to trace that the thread makes (
                including recursive calls) in total

            ncalls_line = ncalls_line / width;
            if (ncalls_line < min_ncalls_line)
            {
                #pragma omp critical(lock1)
                if (ncalls_line < min_ncalls_line)
                {
                    min_ncalls_line = ncalls_line;
                    lineMin = y;
                }
            }

            if (ncalls_line > max_ncalls_line)
            {
                #pragma omp critical(lock2)
                if (ncalls_line > max_ncalls_line)
                {
                    max_ncalls_line = ncalls_line;
                    lineMax = y;
                }
            }
            update_histogram(histo, ncalls_line);
        }

        printf("Thread %d: %d\n", omp_get_thread_num(), counter);
    }
```

Listing 5: Parallelized Loop rayp3.c