**Report on Parallelization Optimization using OpenMP**

**Introduction**

The objective of this report is to analyze and optimize the performance of parallel versions of a ray tracing program using OpenMP across four exercises. The exercises involved parallelizing inner and outer loops, evaluating scheduling influence, and analyzing performance with varying thread counts.

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

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



Rayp1.c

The parallelization strategy involves distributing the workload among threads, with each thread responsible for computing a portion of the line's pixels. The 'reduction' clause ensures the correct aggregation of the **ncalls\_line** values computed by individual threads.

The parallelization significantly improves computational efficiency, especially for images with larger dimensions. By parallelizing the inner loop, the program exploits parallel processing capabilities, resulting in a noticeable speedup in image generation.

In conclusion, the parallelization of the inner loop proves to be a successful optimization strategy, as evidenced by the observed speedup in image generation. The workload distribution among threads and the reduction strategy effectively make the most of the parallel processing capabilities provided by OpenMP. This enhancement is particularly beneficial for scenarios involving large image dimensions, where the workload is considerable.

The resulting images demonstrate the positive impact of parallelizing the inner loop, validating the approach and highlighting the efficiency gains achieved through parallel processing.

A screenshot of a computer

Description automatically generated

Ray.c

A screenshot of a computer screen

Description automatically generated

Rayp1.c

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

* The aim of this task is to make the inner loop (y loop) in the render function run in parallel, enhancing the speed of image generation. OpenMP directives and clauses are used to efficiently tap into the capabilities of parallel processing.
* We've introduced **ncalls\_line** as a private variable, ensuring each thread independently calculates sums for individual lines. To protect against conflicts in accessing global variables, particularly 'max\_ncalls\_line' and 'min\_ncalls\_line', we've implemented protective measures. Similarly, we've secured the function call responsible for updating variables to maintain data integrity. These precautions are crucial for a flawless parallel execution, preventing potential conflicts during simultaneous access to shared resources.
* We observe a noticeable reduction in execution time compared to both the original and the preceding versions of the program. By parallelizing the outer loop, we achieve a more efficiently scheduled workload for threads, enabling them to compute entire lines rather than individual components. This modification results in a larger portion of the code falling within the parallelized region, contributing to a substantial speedup in overall performance. The enhanced workload distribution among threads plays a big role in optimizing the program's efficiency.
* The implemented modifications have led to a noticeable reduction in the program's execution time. Consequently, the finalized version of this program should focus on parallelizing the outer loop. This adjustment optimizes the utilization of computational resources, resulting in improved efficiency and overall speed. The strategic parallelization of the outer loop allows for a more balanced distribution of tasks among processing units, reducing idle times and enhancing the program's overall performance. The refined version not only accelerates execution but also takes advantage of the parallel processing capabilities to achieve a more simplified and effective computational workflow.

A screenshot of a computer screen

Description automatically generated



**Exercise 3: Evaluate Scheduling Influence**

* **Objective:** Analyze performance with different scheduling choices using 32 threads.
* **Approach:** Tested static and dynamic scheduling with varying chunk sizes.
* **Analysis:** Dynamic scheduling slightly outperformed static scheduling, with default chunk size being optimal.
* **Conclusion:** Load balancing and histogram update mechanisms affected performance, indicating the need for a balanced approach.

| **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 |

**Exercise 4: Varying Threads and Optimal Scheduling**

* **Objective:** Assess performance with varying thread counts, using the best scheduling choices from Exercise 3.
* **Approach:** Ran parallel versions with powers of 2 threads and optimal scheduling.
* **Analysis:** **p2** consistently outperformed **p1**, showcasing better scalability and efficiency.
* **Conclusion:** **p2** demonstrated superior performance across different thread counts, emphasizing its efficiency in leveraging parallelism.

**The data outlines the correlation between 'Nthreads', 'p2', and 'p1'. As the number of threads increases, there's a general decrease in 'p1' and 'p2' values. Exceptionally, 'p1' and 'p2' plateau between 64 and 128 threads.**

**Highlights 📊**

- 'p2' and 'p1' averages are remarkably similar, signaling close performance metrics.

- Lowest 'p1' and 'p2' values at 64 threads, interestingly value increases after this 📈.

- Most variation occurs in 'Nthreads' with a large range of 126.

**Recommended Next Steps**

- Examine why results plateau and increase again after 64 threads.

- Test whether the pattern continues beyond 128 threads.

- Customise thread levels based on whether 'p1' or 'p2' is a higher priority for optimisation.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nthreads | p2 | p1 | p0 | SpeedUp | Efficency |
| 2 | 38,72 | 38,73 | 77,16 | 1,992196 | 0,996098 |
| 4 | 19,54 | 19,50 | 77,16 | 3,957157 | 0,989289 |
| 8 | 9,91 | 9,89 | 77,16 | 7,801516 | 0,975189 |
| 16 | 5,92 | 5,91 | 77,16 | 13,06033 | 0,816271 |
| 32 | 3,17 | 3,11 | 77,16 | 24,79515 | 0,774848 |
| 64 | 2,70 | 2,70 | 77,16 | 28,54031 | 0,445942 |
| 128 | 2,94 | 2,99 | 77,16 | 25,8095 | 0,201637 |

A graph with a line

Description automatically generatedA screen shot of a graph

Description automatically generated

**Overall Conclusion**

* **Best Parallel Version:** **p2** consistently exhibited superior performance in execution times, demonstrating more effective parallelization optimizations compared to **p1**.
* **Scheduling Impact:** Optimal scheduling choices significantly influenced performance, with careful consideration required for load balancing and shared resource updates.
* **Thread Count and Scalability:** Diminishing returns were observed with increased thread counts, emphasizing the need to balance speed-up and efficiency.

**Recommendations**

* Focus further optimization efforts on **p2**, leveraging its efficient parallelization strategies.
* Experiment with scheduling and chunk sizes to balance load and optimize resource utilization.
* Consider trade-offs between speed-up and efficiency when scaling parallel implementations.

**Conclusion**

The exercises underscored the significance of thoughtful parallelization strategies and scheduling choices in optimizing performance. **p2** emerged as the superior parallel version due to its consistent efficiency and scalability. Future enhancements should aim at refining scheduling strategies and resource management for even better performance gains.

**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.

To do so, we introduce an array to store the number of 'trace' function calls made by each thread. Each index of the array corresponds to a specific thread.

Print the index of the 'y' loop corresponding to the most and least complex lines by tracking the variables 'max\_ncalls\_line' and 'min\_ncalls\_line'. Use critical sections ('#pragma omp critical') to ensure the correctness of updates to these variables.

Implement critical sections for updating 'max\_ncalls\_line', 'min\_ncalls\_line', and for calling the 'update\_histogram()' function. This prevents race conditions and ensures accurate profiling.

With a dynamic schedule in play, threads quickly transition between lines, ensuring a balanced workload distribution. This adaptive scheduling minimizes idle times and optimizes overall parallel execution.

Identify the most and least complex lines based on the number of reflective spheres. The correlation between these lines and the threads executing the most and least 'trace' calls provides valuable insights into the program's behavior.

When measuring line complexity, we identify complexity based on reflective sphere counts. More reflective spheres make a line complex, while fewer spheres denote simplicity.

This detailed analysis provides a complete view of the ray tracer program's parallel behavior, informing strategies for better performance in scenarios with varying workloads.

With this code implementation, we can monitor the workload distribution among threads, allowing us to assess the impact of various schedule types. Additionally, we might observe a connection between the most/least complex lines and the threads with the highest and lowest trace function calls, respectively.A screenshot of a computer

Description automatically generated