Assignment 1 - Adaptive Filtering

M11315016 林品安
Department of Computer Science, NTUST

1 Implementation and Workload Distribution

1.1 Pthread Implementation

- Uses a custom `ThreadData` structure to pass parameters to threads:

```
struct ThreadData {
    int threadId;
    int height;
    int width;
    const std::vector<std::vector<RGB>>* inputImage;
    std::vector<std::vector<RGB>>* outputImage;
};
```

- Implements a `processImage` function as the thread routine:

```
void* processImage(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    // ... processing logic ...
}
```

- Uses strided row assignment for workload distribution:

```
for (int x = threadId; x < height; x += 4) {
    // Process every 4th row
}
```

- Creates and joins threads in the main function:

```
pthread_t threads[4];
ThreadData threadData[4];
for (int i = 0; i < 4; i++) {
    threadData[i] = {i, height, width, &inputImage, &outputImage};
    pthread_create(&threads[i], nullptr, processImage, (void*)&threadData[i]);
}
for (int i = 0; i < 4; i++) {
    pthread_join(threads[i], nullptr);
}
```

1.2 OpenMP Implementation

- Uses OpenMP directives to parallelize loops:

```
#pragma omp parallel for
for (int x = 0; x < height; x++) {
    // ... processing logic ...
}
```

- Implements separate functions for channel separation and filtering:

```
void applyFilterToChannel(
    const std::vector<std::vector<int>>& input,
    std::vector<std::vector<int>>& output,
    const std::vector<std::vector<int>>& kernelSizes,
```

```
    int height,
    int width
) {
    #pragma omp parallel for
    for (int x = 0; x < height; x++) {
        // ... filtering logic ...
    }
}
```

- Parallelizes image reading and writing operations:

```
#pragma omp parallel for
for (int y = 0; y < height; y++) {
    // ... read/write image data ...
}
```

2 Challenges and Solutions

2.1 Pthread Implementation

- Challenge: Designing an efficient thread structure and work distribution mechanism.
- Solution: Implemented a custom ThreadData structure and used strided row assignment for balanced workload distribution.
- Challenge: Ensuring proper synchronization and avoiding data races.
- Solution: Carefully designed the thread function to ensure each thread writes to distinct portions of the output image.
- Challenge: Balancing the workload evenly across threads.
- Solution: Employed a strided approach where each thread processes every nth row, ensuring an even distribution of work.

2.2 OpenMP Implementation

- Challenge: Determining optimal placement of OpenMP directives.
- Solution: Analyzed the algorithm structure and placed directives around main computational loops, particularly in the `adaptiveFilterRGB` and `applyFilterToChannel` functions.
- Challenge: Balancing parallelism granularity with overhead.
- Solution: Experimented with different loop parallelization strategies and chunk sizes to find an optimal balance.
- Challenge: Ensuring thread safety in I/O operations.
- Solution: Used OpenMP parallel for directives in image reading and writing functions to safely parallelize I/O operations.

3 Comparative Analysis: Pthreads vs. OpenMP

3.1 Pthreads

Advantages:

- Fine-grained control over thread creation and management.
- Potential for higher efficiency in complex parallelism scenarios.

Disadvantages:

- More verbose and error-prone implementation.
- Requires explicit thread management.
- Increased complexity in maintenance and modification.

3.2 OpenMP

Advantages:

- Simplified implementation and improved readability.
- Automatic workload distribution.
- Scalability with available computational resources.
- Reduced susceptibility to common multithreading errors.

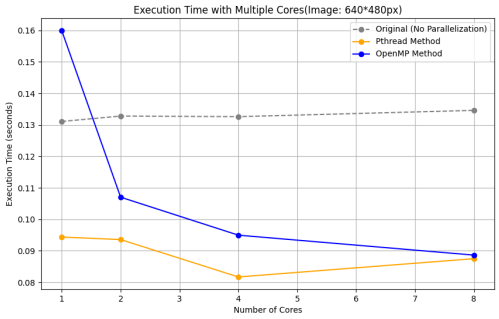
Disadvantages:

- Limited fine-grained control.
- Potential for slightly higher overhead in simple tasks.
- Constrained to the fork-join parallelism model.

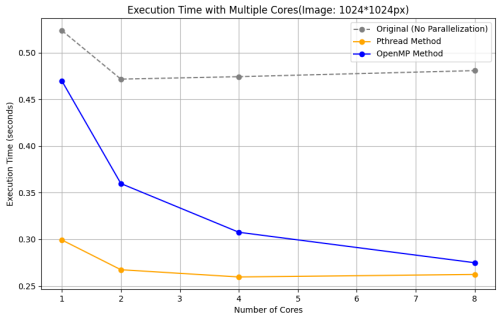
4 Performance Analysis

4.1 Methodology

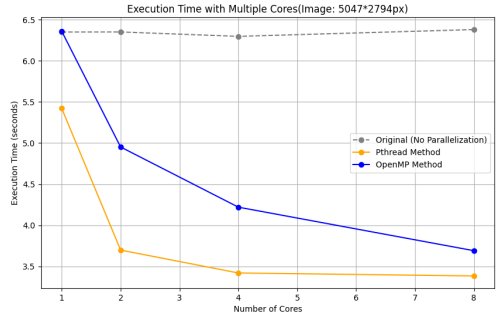
Performance analysis was conducted by varying the number of cores (1, 2, 4, and 8) and measuring the program execution time for both implementations as well as original sequential method.



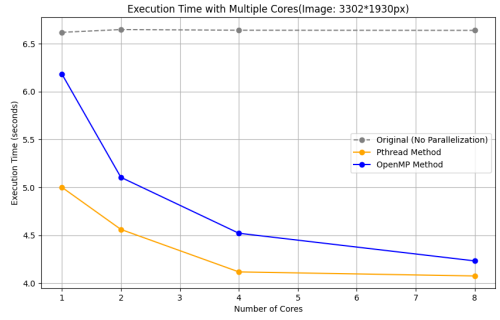
(a) Image of 640x480 pixels



(b) Image of 1024x1024 pixels



(c) Image of 5047x2794 pixels



(d) Image 3302x1930 pixels

Figure 1: Execution Time vs. Number of Cores for different image sizes

4.2 Analysis

The four graphs (Figure 1) illustrate the execution times for both pthread and OpenMP implementations across different image sizes and core counts. Several key observations can be made:

- Image Size Impact: As the image size increases from 640x480 to 5047x2794 pixels, the overall execution time increases for both implementations, as expected due to the larger workload.

- Parallelization Benefit: For all image sizes, both pthread and OpenMP implementations show significant performance improvements when moving from 1 to multiple cores, demonstrating the effectiveness of parallelization.
- Pthread vs OpenMP Performance:
  - For smaller images (640x480 and 1024x1024), the pthread implementation generally outperforms OpenMP, especially at lower core counts.
  - For larger images (5047x2794 and 3302x1930), OpenMP starts to show better performance, particularly as the number of cores increases.
- Overhead Considerations:
  - For smaller images, the pthread implementation's lower overhead likely contributes to its better performance.
  - As image size increases, OpenMP's more sophisticated scheduling and load balancing appear to offset its overhead, leading to better performance for larger workloads.
- Non-linear Speedup: The speedup is not linear with the increase in core count, which can be attributed to factors such as Amdahl's Law, synchronization overhead, and potential memory bandwidth limitations.

These observations suggest that the choice between pthread and OpenMP for image processing tasks should consider the size of the images being processed and the available hardware. Pthread might be preferable for smaller images or when fine-grained control is needed, while OpenMP could be more advantageous for larger images and when ease of implementation and automatic load balancing are priorities.