Assignment 2 - Mandelbulb Set

M11315016 林品安

1 Implementation

1.1 Scheduling Algorithm

In this implementation, we use the OpenMP directive `#pragma omp parallel for schedule(dynamic) collapse(2)`. The `dynamic` scheduling algorithm was chosen because:

- The workload for each pixel may vary due to the complexity of the ray-traced scene.
- `dynamic` scheduling allows load balancing across threads by redistributing work at runtime.
- This scheduling method helps prevent threads from being idle while other threads process more complex regions.

The `collapse(2)` clause allows the OpenMP runtime to parallelize both loops over image rows and columns, giving more flexibility in distributing the workload across threads.

1.2 Task Partitioning

The tasks are partitioned at two levels:

- At the **MPI level**, each process is responsible for a portion of the image (a specific number of rows). This partitioning is determined by:

```
    int rows_per_rank = height / size;
    int remainder = height % size;
    int start_row = rank * rows_per_rank + (rank < remainder ? rank : remainder);
    int end_row = start_row + rows_per_rank + (rank < remainder ? 1 : 0);
```

  This approach ensures that each MPI process gets an equal number of rows, with the first few processes handling any extra rows.

- At the **OpenMP level**, each MPI process further parallelizes the work across multiple threads. Each thread is responsible for rendering multiple pixels (anti-aliasing and color computation) within its assigned portion of the image.
- After rendering, two arrays(sendcounts and displs) are created to manage how data will be gathered from each process. These arrays will store:
  - sendcounts: The number of pixels (in bytes) each process sends to rank 0.
  - displs: The displacement (offset in the final array) where each process' s data will be placed in the final image on rank 0.

```
        // Create an array to store the number of elements each rank is sending
    int* sendcounts = new int[size];
    int* displs = new int[size];

    // Calculate sendcounts and displacements
    for (int i = 0; i < size; i++) {
        int r = height / size;
        int start = i * r + (i < remainder ? i : remainder);
        int end = start + r + (i < remainder ? 1 : 0);
        sendcounts[i] = (end - start) * width * 4;
        displs[i] = (i == 0) ? 0 : displs[i-1] + sendcounts[i-1];
    }
```

- `MPI_Gatherv` gathers the pixel data from all processes and assembles them in `raw_image` on rank 0.
  - `local_image`: The buffer from each process containing its portion of the image.
  - `local_height * width * 4`: The number of bytes in each process' s local image buffer.
  - `raw_image`: The final image buffer on rank 0.
  - `sendcounts`: The number of bytes each process contributes to raw_image.
  - `displs`: The displacement (starting offset) for each process' s data in raw_image.

```
    MPI_Gatherv(local_image, local_height * width * 4, MPI_UNSIGNED_CHAR,
            raw_image, sendcounts, displs, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

    // Save image on rank 0
    if (rank == 0) {
        write_png(argv[10]);
        delete[] raw_image;
    }
```

1.3 Techniques to Reduce Execution Time

Several optimization techniques were used to reduce execution time:

- **Dynamic scheduling in OpenMP:** This balances the workload dynamically, improving performance by reducing idle time for threads handling complex regions.
- **Efficient task partitioning:** By using MPI for process-level partitioning and OpenMP for thread-level parallelization, we maximized the use of all available CPU cores.
- The `#pragma omp parallel sections` directive divides the work into three independent sections. Each section computes one of dx, dy, or dz in parallel, allowing these computations to happen simultaneously on separate threads.

```
// use gradient to calc surface normal
vec3 calcNor(vec3 p) {
    vec2 e = vec2(eps, 0.);
    double dx, dy, dz;

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            dx = map(p + e.xyy()) - map(p - e.xyy());  // Compute dx in one thread
        }
        #pragma omp section
        {
            dy = map(p + e.yxy()) - map(p - e.yxy());  // Compute dy in another
                thread
        }
        #pragma omp section
        {
            dz = map(p + e.yyx()) - map(p - e.yyx());  // Compute dz in another
                thread
        }
    }

    return normalize(vec3(dx, dy, dz));
}
```

1.4 Other Efforts

Other efforts to optimize the program include:

- **Memory management:** Each process only allocates memory for the portion of the image it renders, reducing memory overhead.
- **Load balancing:** The `dynamic` scheduling reduces the risk of load imbalance between threads, ensuring that the workload is spread evenly even when some parts of the image require more computation.

2 Analysis

2.1 Load Balancing

As the number of threads increases, execution time decreases for both 6 and 9 processes, indicating improved parallelism. However, 6 processes consistently outperform 9 processes, and the performance gains diminish as threads increase beyond 6, suggesting an optimal thread count for efficiency.

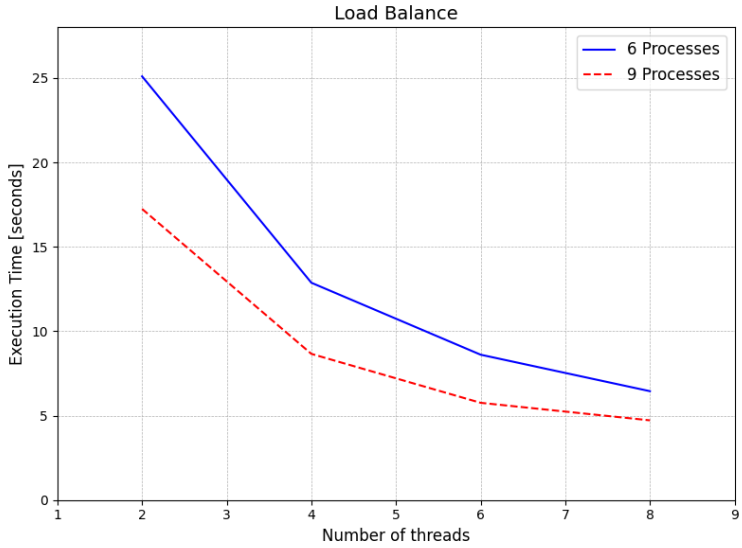

Figure 1: Execution time versus number of threads for image of 512*512 on 3 nodes

2.2 Scalability Analysis

We analyze scalability in three key aspects:

- **Number of CPU cores per process:** By increasing the number of threads per MPI process, we achieve better parallelism for complex scenes.
- **Number of processes per node:** Increasing the number of MPI processes per node reduces the memory footprint per process but can lead to contention for resources like memory bandwidth and cache, especially for high-resolution images.
- **Number of nodes:** Scaling to multiple nodes improves performance significantly, especially for large-scale problems, since it reduces the amount of work per node. However, network communication (e.g., `MPI_Gatherv`) can become a bottleneck as the number of nodes increases.

3 Conclusion

3.1 Lessons Learned

From this assignment, I learned:

- The importance of balancing workload distribution in parallel programming.
- How to combine MPI for distributed computing with OpenMP for shared-memory parallelism.
- Effective use of scheduling algorithms like `dynamic` in OpenMP to balance uneven workloads.

3.2 Difficulties Encountered

Some difficulties I encountered include:

- Handling load balancing between processes for irregular workloads (e.g., complex regions of the Mandelbulb fractal).
- Managing memory allocation efficiently when distributing work across MPI processes and OpenMP threads.

3.3 Feedback and Suggestions

The assignment was both challenging and rewarding, offering practical experience in hybrid parallel programming.