ARTI 503 – Parallel Computer Architecture and Programming

Term 1 – 2024/2025
ARTI 503: Project Stage 1

Multiple Player Game State Synchronization

Section: ARTI 503- 9FS2(3823)
Group: Group 2 – "Muneerh Alfaleh"
Supervised By: Rabab Alkhalifa

#	Name	ID	Role
1	Muneerh Alfaleh	2210002858	Leader
2	Zahra AlEid	2210002616	Member
3	Hawra Alsedrah	2210003421	Member
4	Rose Hummusani	2210002644	Member
5	Jenan Albuzaid	2210003173	Member
6	Zainab al mousa	2210003212	Member

Introduction:

Multiplayer games generate a vast amount of real-time data as players interact and alter the game state continuously. Efficient handling is required in synchronizing data to multiple clients so that gaming can be carried out effectively with consistency and minimum lag. Parallelization can be fruitful in the case of MPI for state distribution and synchronization of clients in games by offering concurrent processing of data, resulting in faster updates with decreased loads on the network as well as enhanced scalability. By leveraging MPI, game state data can be divided and handled in parallel, their updates optimized in network usage to a high-performance multiplayer experience of users.

Objective

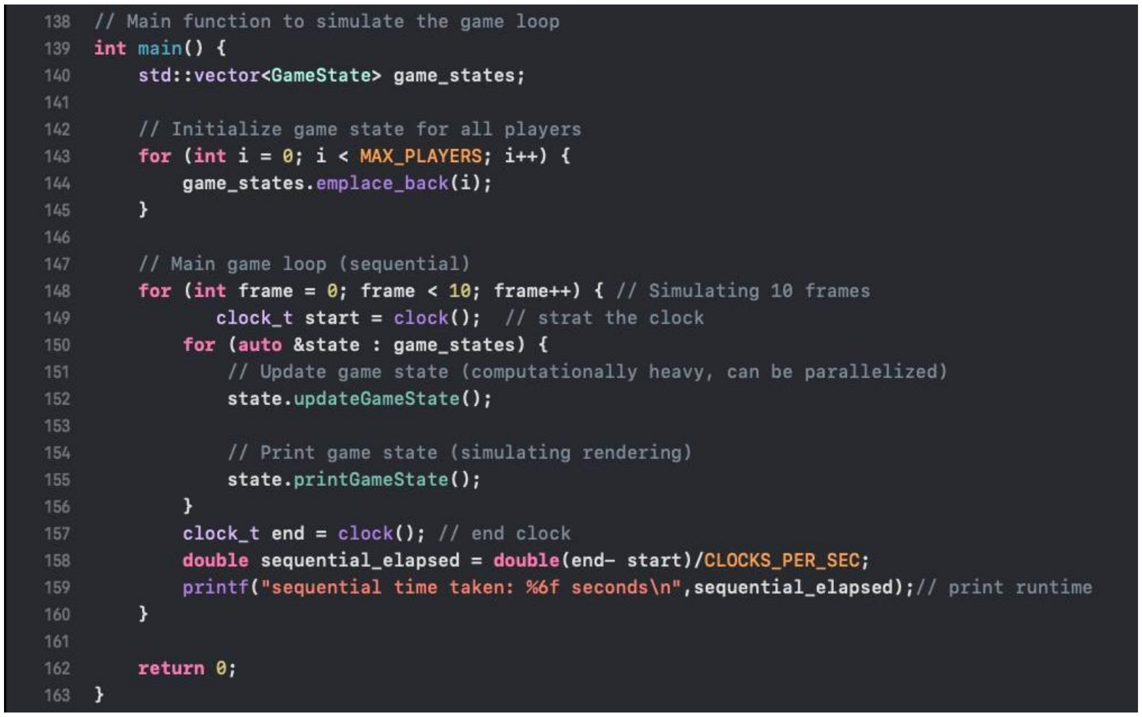
Our purpose is to enhance the real-time process of game rendering to improve frame rates by parallelizing a sequential code; the given code is written by ChatGPT, which simulates a multiplayer large game world with textures, lights, and 3D objects such as the players. However, this code results in performance bottlenecks, especially if we plan on increasing the number of the previously mentioned constants. By parallelizing the code we'll be dividing the tasks into multiple threads with multithreading and leveraging GPU acceleration by offloading tasks such as lighting and position updates to the GPU, we aim to reduce the execution time per frame, which in turn will improve the frame rates and performance.

Justification

This code consists of many huge constants that require heavy computations that take a long time to calculate. Parallelizing such a problem can make it very efficient as it includes many loops, especially nested loops that can be parallelized without dependencies between its iterations. Loops are included for updating the game object, lighting calculation, and texture updates that occur numerous times throughout the game. The texture operations are represented as a 2D array which can be divided into separate chunks that can work independently. Additionally, the problem can be scaled up to increase its complexity and handle more players, objects, and textures which can benefit from dividing it into smaller parallelizable. Moreover, the intensive heavy commutations of the code can benefit from parallel processing as the light computation function involves floating-point operations which are considered computationally expensive. And lastly, minimal conditional logic is included for lighting calculations.

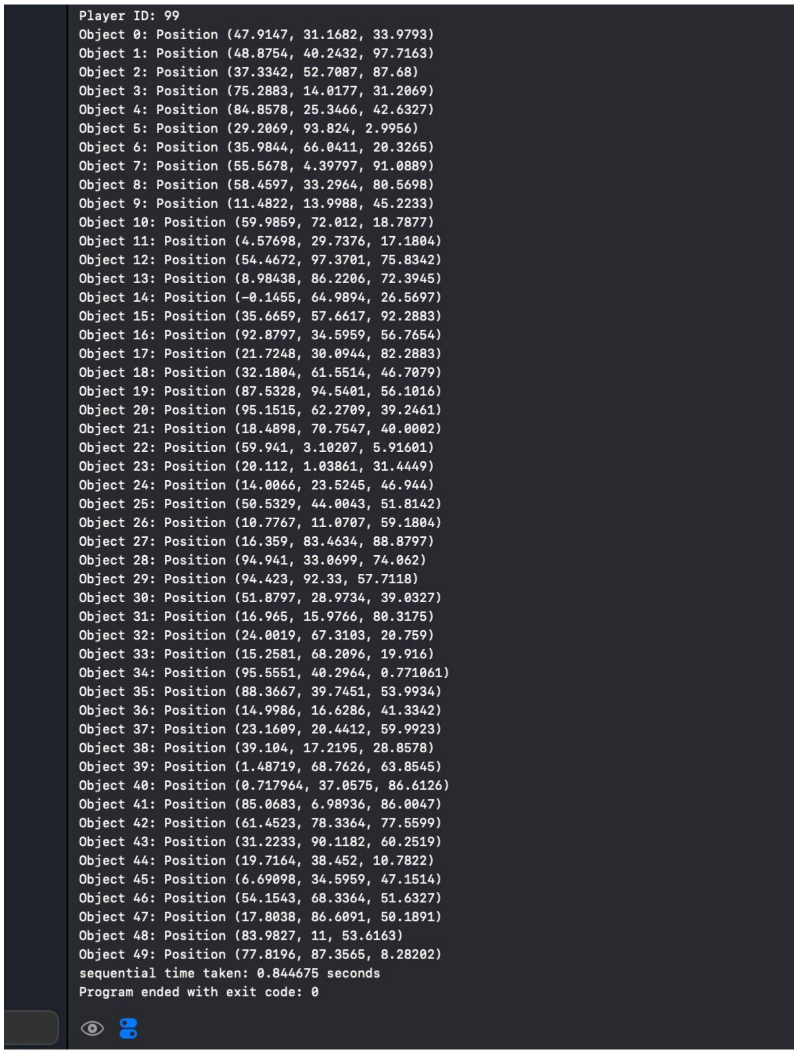
Sequential Benchmarking

Benchmarking with use of clock() mothed:



```cpp
// Main function to simulate the game loop
int main() {
    std::vector<GameState> game_states;

    // Initialize game state for all players
    for (int i = 0; i < MAX_PLAYERS; i++) {
        game_states.emplace_back(i);
    }

    // Main game loop (sequential)
    for (int frame = 0; frame < 10; frame++) { // Simulating 10 frames
           clock_t start = clock();  // strat the clock
        for (auto &state : game_states) {
            // Update game state (computationally heavy, can be parallelized)
            state.updateGameState();

            // Print game state (simulating rendering)
            state.printGameState();
        }
        clock_t end = clock(); // end clock
        double sequential_elapsed = double(end- start)/CLOCKS_PER_SEC;
        printf("sequential time taken: %6f seconds\n",sequential_elapsed);// print runtime
    }

    return 0;
}
```

a. The output with benchmarking for the last player:



```
Player ID: 99
Object 0: Position (47.9147, 31.1682, 33.9793)
Object 1: Position (48.8754, 40.2432, 97.7163)
Object 2: Position (37.3342, 52.7087, 87.68)
Object 3: Position (75.2883, 14.0177, 31.2069)
Object 4: Position (84.8578, 25.3466, 42.6327)
Object 5: Position (29.2069, 93.824, 2.9956)
Object 6: Position (35.9844, 66.0411, 20.3265)
Object 7: Position (55.5678, 4.39797, 91.0889)
Object 8: Position (58.4597, 33.2964, 80.5698)
Object 9: Position (11.4822, 13.9988, 45.2233)
Object 10: Position (59.9859, 72.012, 18.7877)
Object 11: Position (4.57698, 29.7376, 17.1804)
Object 12: Position (54.4672, 97.3701, 75.8342)
Object 13: Position (8.98438, 86.2206, 72.3945)
Object 14: Position (-0.1455, 64.9894, 26.5697)
Object 15: Position (35.6659, 57.6617, 92.2883)
Object 16: Position (92.8797, 34.5959, 56.7654)
Object 17: Position (21.7248, 30.0944, 82.2883)
Object 18: Position (32.1804, 61.5514, 46.7079)
Object 19: Position (87.5328, 94.5401, 56.1016)
Object 20: Position (95.1515, 62.2709, 39.2461)
Object 21: Position (18.4898, 70.7547, 40.0002)
Object 22: Position (59.941, 3.10207, 5.91601)
Object 23: Position (20.112, 1.03861, 31.4449)
Object 24: Position (14.0066, 23.5245, 46.944)
Object 25: Position (50.5329, 44.0043, 51.8142)
Object 26: Position (10.7767, 11.0707, 59.1804)
Object 27: Position (16.359, 83.4634, 88.8797)
Object 28: Position (94.941, 33.0699, 74.062)
Object 29: Position (94.423, 92.33, 57.7118)
Object 30: Position (51.8797, 28.9734, 39.0327)
Object 31: Position (16.965, 15.9766, 80.3175)
Object 32: Position (24.0019, 67.3103, 20.759)
Object 33: Position (15.2581, 68.2096, 19.916)
Object 34: Position (95.5551, 40.2964, 0.771061)
Object 35: Position (88.3667, 39.7451, 53.9934)
Object 36: Position (14.9986, 16.6286, 41.3342)
Object 37: Position (23.1609, 20.4412, 59.9923)
Object 38: Position (39.104, 17.2195, 28.8578)
Object 39: Position (1.48719, 68.7626, 63.8545)
Object 40: Position (0.717964, 37.0575, 86.6126)
Object 41: Position (85.0683, 6.98936, 86.0047)
Object 42: Position (61.4523, 78.3364, 77.5599)
Object 43: Position (31.2233, 90.1182, 60.2519)
Object 44: Position (19.7164, 38.452, 10.7822)
Object 45: Position (6.69098, 34.5959, 47.1514)
Object 46: Position (54.1543, 68.3364, 51.6327)
Object 47: Position (17.8038, 86.6091, 50.1891)
Object 48: Position (83.9827, 11, 53.6163)
Object 49: Position (77.8196, 87.3565, 8.28202)
sequential time taken: 0.844675 seconds
Program ended with exit code: 0
```

b. After reviewing and testing the code in many approaches, we concluded that the function updateGameState() is the one that appears to be the heaviest in terms of time. This includes:

- **Lighting Computation**: In this case, light sources must be considered for every object of the scene hence the inclusion of operations such as the floating-point operations of division as well as sqrtf.
- **Texture Updates**: Once the light sources have been made, texture pixels are adjusted in subsequent project tasks.

Because these operations are computationally intensive and are performed on a fixed basis for each object, this section turns out to be the choking point of the entire simulation.

The code

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdlib>

#define MAX_PLAYERS 100
#define GRID_SIZE 100  // Simulating a large game world
#define OBJECT_COUNT 50
#define LIGHT_COUNT 10
#define TEX_SIZE 256

// Class to represent 3D vectors (for positions, light sources, etc.)
class Vector3 {
public:
    float x, y, z;

    // Constructor for Vector3
    Vector3(float x = 0.0f, float y = 0.0f, float z = 0.0f) : x(x), y(y), z(z) {}
};

// Class to represent textures (2D arrays)
class Texture {
public:
    float pixels[TEX_SIZE][TEX_SIZE];

    // Initialize the texture with random values
    Texture() {
        for (int i = 0; i < TEX_SIZE; i++) {
            for (int j = 0; j < TEX_SIZE; j++) {
                pixels[i][j] = static_cast<float>(rand() % 256) / 255.0f;
            }
        }
    }
};

// Class to represent a 3D object in the game (e.g., a player or an environment element)
class GameObject {
public:
    Vector3 position;
    Vector3 rotation;
    Texture texture;

    // Constructor to initialize a game object with random position and rotation
    GameObject() {
        position = Vector3(static_cast<float>(rand() % GRID_SIZE),
                           static_cast<float>(rand() % GRID_SIZE),
                           static_cast<float>(rand() % GRID_SIZE));

        rotation = Vector3(static_cast<float>(rand() % 360),
                           static_cast<float>(rand() % 360),
                           static_cast<float>(rand() % 360));
    }

    // Update position based on rotation (dummy transformation)
    void updatePosition() {
        position.x += cosf(rotation.x) * 0.1f;
        position.y += sinf(rotation.y) * 0.1f;
        position.z += cosf(rotation.z) * 0.1f;
    }
};

// Class to represent lighting in the game
class Light {
public:
    Vector3 position;
    float intensity;

    // Constructor to initialize a light source with random position and intensity
    Light() {
        position = Vector3(static_cast<float>(rand() % GRID_SIZE),
                           static_cast<float>(rand() % GRID_SIZE),
                           static_cast<float>(rand() % GRID_SIZE));
        intensity = static_cast<float>(rand() % 100) / 100.0f;
    }
};

// Class to represent the game state for each player
class GameState {
public:
    int player_id;
```

```
std::vector<GameObject> objects;
std::vector<Light> lights;

// Constructor to initialize the game state for a player
GameState(int id) : player_id(id) {
    objects.resize(OBJECT_COUNT);
    lights.resize(LIGHT_COUNT);
}

// Function to compute lighting for an object (computationally heavy)
float computeLighting(GameObject &obj, Light &light) {
    float dx = obj.position.x - light.position.x;
    float dy = obj.position.y - light.position.y;
    float dz = obj.position.z - light.position.z;
    float distance = sqrtf(dx * dx + dy * dy + dz * dz);
    return light.intensity / (distance + 1.0f); // Avoid division by zero
}

// Update the game state (computationally heavy)
void updateGameState() {
    for (auto &obj : objects) {
        obj.updatePosition();

        // Apply lighting from all lights
        float total_light = 0.0f;
        for (auto &light : lights) {
            total_light += computeLighting(obj, light);
        }

        // Modulate texture brightness based on lighting
        for (int ti = 0; ti < TEX_SIZE; ti++) {
            for (int tj = 0; tj < TEX_SIZE; tj++) {
                obj.texture.pixels[ti][tj] *= total_light; // Brighten texture
            }
        }
    }
}

// Print the game state (to simulate rendering)
void printGameState() const {
    std::cout << "Player ID: " << player_id << std::endl;
    for (size_t i = 0; i < objects.size(); i++) {
            const GameObject &obj = objects[i];
            std::cout << "Object " << i << ": Position (" << obj.position.x << ", "
                      << obj.position.y << ", " << obj.position.z << ")" << std::endl;
        }
    }
};

// Main function to simulate the game loop
int main() {
    std::vector<GameState> game_states;

    // Initialize game state for all players
    for (int i = 0; i < MAX_PLAYERS; i++) {
        game_states.emplace_back(i);
    }

    // Main game loop (sequential)
    for (int frame = 0; frame < 10; frame++) { // Simulating 10 frames
        for (auto &state : game_states) {
            // Update game state (computationally heavy, can be parallelized)
            state.updateGameState();

            // Print game state (simulating rendering)
            state.printGameState();
        }
    }

    return 0;
}
```