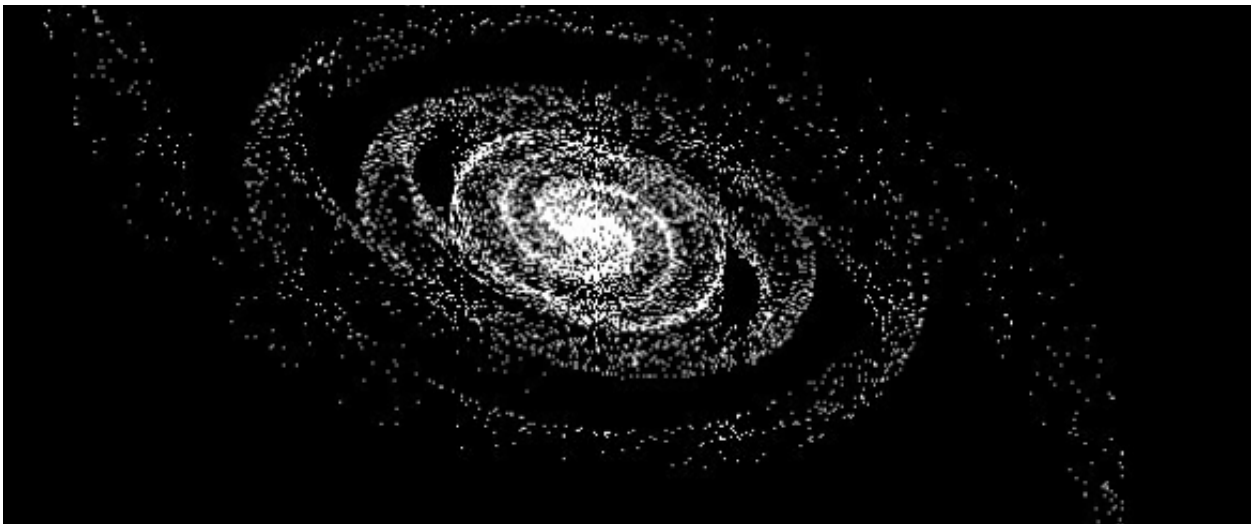


Final Project: N-Problem Simulation

Anas Al Baqeri: 202004427

Mohammad Shahin : 202105493

Chapter 1: Introduction

What’s it all about?

The N-body problem refers to the challenge of predicting the long-term motion of a system containing three or more gravitating objects, such as planets in a solar system. While equations can accurately describe the **gravitational forces** between two bodies (thanks to Newton's laws), extending this to systems with more bodies introduces complexity. The problem becomes analytically unsolvable due to the increasing number of unknown variables, leading to chaotic behavior where small differences in initial conditions result in vastly different outcomes over time. The restricted three-body problem is a simplified approach, treating one body as negligible, and advancements in computer simulations help approximate solutions for more accurate predictions.

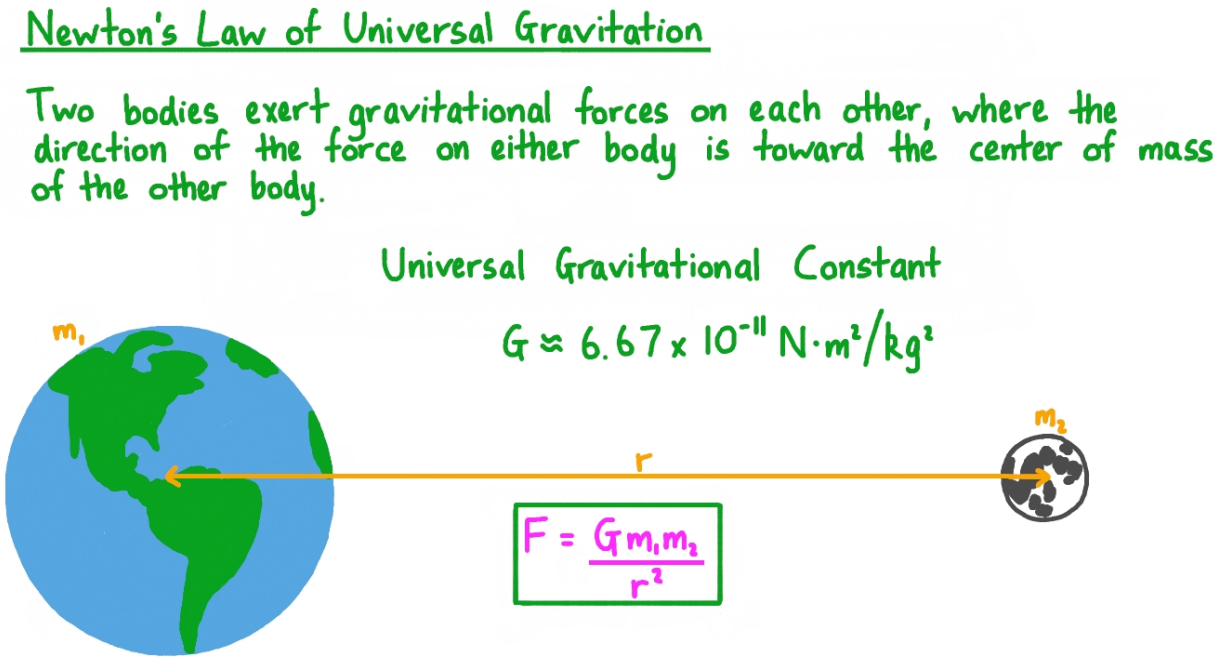
Gravitational forces:

According to Newton's laws of gravitation, any two objects in the universe exert a gravitational pull on each other. This gravitational interaction affects both the position and velocity of the two objects. The larger object exerts a more substantial gravitational force, pulling the smaller object towards it. This law explains why planets, including Earth, orbit around the sun. The sun, being much more massive than the planets, exerts a larger gravitational force, effectively pulling them into an orbit around it.

$$F = G\frac{m_1 m_2}{r^2}$$

F = force
G = gravitational constant
m_1 = mass of object 1
m_2 = mass of object 2
r = distance between centers of the masses

The N-body problem presents a significant challenge when more than two objects are involved. The complexity of this problem increases exponentially as the number of objects increases, making it an extremely hard task to compute and predict the future trajectories of these objects. The primary difficulty lies in the fact that many unknown variables are at play in the situation. Thus, the task of determining the behavior of these objects becomes incredibly tedious, given the limited number of known equations available to calculate these unknowns. The result is an intricate puzzle that requires advanced computational methods to solve, further adding to the problem's difficulty.



Mathematical solution

In order to accurately predict the trajectory of a variety of different objects, our common approach is to implement the Euler method. This method, named after the Swiss mathematician Leonhard Euler, is a numerical procedure for solving ordinary differential equations with a given initial value. It generates a curve, which is a numerical approximation of the true trajectory. This curve is constructed by a series of short straight lines that approximate the object's path in the physical world. This is particularly useful in physics and engineering simulations where precise predictions of object paths are necessary.

Certainly! The Euler method is a numerical technique used to estimate the solution to ordinary differential equations (ODEs), and it can be applied to approximate the motion of celestial bodies in an N-body system. Here's a simplified explanation of how the Euler method can be used for the N-body problem:

Euler Method for N-Body Problem:

Write down the differential equations that govern the motion of each body based on Newton's laws of motion and gravity. These equations describe how the position and velocity of each body change over time.

1. **Discretize Time:**
   - Divide the time interval of interest into small steps or intervals. Let's call the time step Δt.
   - In our case, each step is defined at 0.01f.
2. **Initialize Positions and Velocities:**
   - Start with initial positions and velocities for each body.
3. **Iterative Calculation:**
   - For each time step:
     - Calculate the gravitational forces acting on each body due to the presence of other bodies.
     - Update the velocities and positions of each body using the Euler method.

   Euler Method Update Equations:

- Velocity Update:

$$v_i^{(n+1)} = v_i^{(n)} + a_i^{(n)} \cdot \Delta t$$

- Position Update:

$$r_i^{(n+1)} = r_i^{(n)} + v_i^{(n)} \cdot \Delta t$$

Where:

$$v_i^{(n)} is\ the\ velocity\ of\ the\ i_{th}\ body\ at\ time\ step\ n.$$

$$r_i^{(n)}\ is\ the\ position\ of\ the\ i^{th}\ body\ at\ time\ step\ n.$$

$$a_i^{(n)}\ is\ the\ acceleration\ of\ the\ i^{th}\ body\ attime\ step\ n.$$

And delta t is the step size we moving and calculating after each time. Repeat the iterative calculation for the desired number of time steps or until reaching a specific endpoint.

Limitations and Considerations:

- The Euler method is a simple numerical approximation and may introduce errors, especially for long-term predictions or systems with rapidly changing dynamics.
- Using smaller time steps delta t can improve accuracy but may increase computational cost.
- Advanced numerical methods, such as higher-order Runge-Kutta methods, may be employed for more accurate results.

In summary, the Euler method provides a straightforward approach to numerically estimate the motion of celestial bodies in an N-body system by updating positions and velocities at discrete time steps based on gravitational forces.

Chapter 2: Sequential Implementation

This process involves the execution of a simulation model for the n-body problem in a sequential version. This version is derived from the example provided at the link https://github.com/harrism/mini-nbody/blob/master/nbody.c

Particle generation:

This program generates a specified number of particles with random properties, stores the particle data in a binary file, and measures the elapsed time for this process.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <errno.h>
#include <limits.h>
#include <time.h>

typedef struct {
    float mass;
    float x, y, z;
    float vx, vy, vz;
} Particle;
```

- **Header Inclusions and Struct Definition:** Include necessary header files and define a struct named `Particle` representing properties of a particle, including mass, position (x, y, z), and velocity (vx, vy, vz).

```
/* Function Declarations */
int convertStringToInt(char *str);
void randomizeParticles(Particle *particles, int n);
```

- **Function Declarations:** Declare two functions used later in the code. `convertStringToInt` converts a string to an integer, and `randomizeParticles` initializes the state of particles with random values.

```
int main(int argc, char *argv[]) {
    int num_particles = 100000; // Default number of particles if no parameter is provided on the command line
    if (argc > 1) {
        // If a command-line parameter indicating the number of particles is provided, convert it to an integer
        num_particles = convertStringToInt(argv[1]);
    }
```

- **Main Function and Command-Line Argument Handling:** Set the default number of particles to 100,000. If a command-line parameter is provided, convert it to an integer using `convertStringToInt` and update the number of particles.

```
    // Allocate memory for particles
    Particle *particles = NULL;
    particles = (Particle *)malloc(num_particles * sizeof(Particle));
```

- **Memory Allocation:** Allocate memory dynamically to store the particle data based on the determined number of particles.

```
    // Start timer
    clock_t start = clock();
```

- **Timer Start:** Record the starting time of the process using the `clock` function.

```
    // Randomize particles
    srand(0);//set seed
    randomizeParticles(particles, num_particles);
```

- **Particle Randomization:** Seed the random number generator with 0 and call `randomizeParticles` to initialize the particle data with random values.

```
    // Write particles to file in binary mode
    FILE *file = fopen("particles.txt", "wb");
    if (file != NULL) {
        fwrite(particles, sizeof(Particle), num_particles, file);
        fclose(file);
```

- **File Writing:** Open a file named "particles.txt" in binary mode for writing. If successful, use `fwrite` to write the particle data to the file, then close the file.

```
        // Stop timer
        clock_t end = clock();
        double elapsed_time = ((double)(end - start)) / CLOCKS_PER_SEC;

        printf("%d particles have been created with random values and written to file: particles.txt in binary format.\\n", num_particle
s);
        printf("Elapsed time for generating %d particles is: %f seconds.\\n", num_particles, elapsed_time);
    }
```

- **Timer Stop and Output:** Stop the timer, calculate the elapsed time, and print the number of particles generated and the time taken to the console.

```
    // Free allocated memory
    free(particles);
}
```

- **Memory Deallocation:** Free the dynamically allocated memory for particle data before program termination.

```
/* Convert string to integer */
int convertStringToInt(char *str) {
    char *endptr;
    long val;
    errno = 0; // To distinguish success/failure after the call

    val = strtol(str, &endptr, 10);

    /* Check for possible errors */
    if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN)) || (errno != 0 && val == 0)) {
        perror("strtol");
        exit(EXIT_FAILURE);
    }

    if (endptr == str) {
        fprintf(stderr, "No digits were found\\n");
        exit(EXIT_FAILURE);
    }

    /* If we are here, strtol() successfully converted a number */
    return (int)val;
}
```

- **String to Integer Conversion Function:** Convert a string to an integer using `strtol`. Check for conversion errors and handle them appropriately. This function is used in the main function to convert the command-line argument.

```
/* Initialize particle state with random values */
void randomizeParticles(Particle *particles, int n) {
    for (int i = 0; i < n; i++) {
        particles[i].mass = 2.0; // Arbitrarily chosen value for particle mass -> 2.0
        particles[i].x = 2.0f * (rand() / (float)RAND_MAX) - 1.0f; // Random number between -1 and 1
        particles[i].y = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        particles[i].z = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        particles[i].vx = 2.0f * (rand() / (float)RAND_MAX) -

1.0f;
        particles[i].vy = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        particles[i].vz = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
    }
}
```

- **Particle Randomization Function:** Assign random values to the properties of each particle. The mass is set to an arbitrarily chosen value of 2.0, and other properties are set to random floating-point values within the range of -1.0 to 1.0. This function is called in the main function to initialize particle data.

Main code:

This particular n-body problem simulation has been expertly adapted to be compatible with the Linux operating environment. The n-body problem, which is a classic issue in physics, requires careful and precise simulation, and this particular example provides a robust model for understanding this complex dynamical system as the one shown below:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include <errno.h>
```

```
#define SOFTENING 1e-9f

typedef struct {
    float mass;
    float x, y, z;
    float vx, vy, vz;
} Particle;

/* Function declarations */
int convertStringToInt(char *str);
void bodyForce(Particle *p, float dt, int n);

int main(int argc, char* argv[]) {

    int nBodies = 10; // Number of bodies if no parameters are given from the command line
    if (argc > 1) nBodies = convertStringToInt(argv[1]);
```

- **Header Inclusions:** Include necessary header files for mathematical operations, file I/O, memory allocation, time measurements, and error handling.
- **Macro Definition:** Define a macro `SOFTENING` with a value of `1e-9` to avoid division by zero in force calculations.
- **Struct Definition:** Define a struct named `Particle` to represent the properties of a particle in the simulation (mass, position, velocity).
- **Function Declarations:** Declare two functions: `convertStringToInt` for converting strings to integers and `bodyForce` for computing inter-body forces.
- **Main Function Start:** Begin the main function.

```
    clock_t startIter, endIter;
    clock_t startTotal = clock(), endTotal;
    double totalTime = 0.0;
```

- **Clock Variables:** Declare variables to measure execution time using the `clock` function. `startTotal` is initialized to the current clock time.

```
    Particle *particles = NULL;
    particles = (Particle *)malloc(nBodies * sizeof(Particle));
```

- **Memory Allocation:** Allocate memory for an array of particles using `malloc`.

```
    FILE *fileRead = fopen("particles.txt", "r");
    if (fileRead == NULL) {
        /* Unable to open the file */
        printf("\\nUnable to open the file.\\n");
        exit(EXIT_FAILURE);
    }
```

- **File Reading:** Attempt to open a file named "particles.txt" for reading. If unsuccessful, print an error message and exit the program.

```
    int particlesRead = fread(particles, sizeof(Particle) * nBodies, 1, fileRead);
    if (particlesRead == 0) {
        /* The number of particles to read is greater than the number of particles in the file */
        printf("ERROR: The number of particles to read is greater than the number of particles in the file\\n");
        exit(EXIT_FAILURE);
    }
    fclose(fileRead);
```

- **Binary File Reading:** Use `fread` to read particle data from the file into the allocated memory. Check if the expected number of particles are read; if not, print an error message and exit.

```
    for (int iter = 1; iter <= nIters; iter++) {
        startIter = clock();

        bodyForce(particles, dt, nBodies); // Compute inter-body forces

        for (int i = 0; i < nBodies; i++) { // Integrate position
            particles[i].x += particles[i].vx * dt;
            particles[i].y += particles[i].vy * dt;
            particles[i].z += particles[i].vz * dt;
        }

        endIter = clock() - startIter;
        printf("Iteration %d of %d completed in %f seconds\\n", iter, nIters, (double)endIter / CLOCKS_PER_SEC);
    }
```

- **Simulation Loop:** Perform a simulation loop with a specified number of iterations (`nIters`). Inside the loop:
  - Measure the iteration start time.
  - Call the `bodyForce` function to compute inter-body forces.
  - Update particle positions based on their velocities.
  - Measure the iteration end time and print the elapsed time.

```
    endTotal = clock();
    totalTime = (double)(endTotal - startTotal) / CLOCKS_PER_SEC;
    double avgTime = totalTime / (double)(nIters);
    printf("\\nAvg iteration time: %f seconds\\n", avgTime);
    printf("Total time: %f seconds\\n", totalTime);
    printf("Number of particles: %d ", nBodies);
```

- **Total Time Calculation:** Measure the total simulation time, calculate the average iteration time, and print the results.

```
    /* Write the output to a file to evaluate correctness by comparing with parallel output */
    FILE *fileWrite = fopen("sequential_output.txt", "w");
    if (fileWrite != NULL) {
        fwrite(particles, sizeof(Particle) * nBodies, 1, fileWrite);
        fclose(fileWrite);
    }

    free(particles);
}
```

- **File Writing:** Write the final state of particles to a file named "sequential_output.txt" for later evaluation of correctness in comparison with parallel output.
- **Memory Deallocation:** Free the allocated memory for the particle array before program termination.

```
/* Conversion from string to integer */
int convertStringToInt(char *str) {
    char *endptr;
    long val;
    errno = 0;  // To distinguish success/failure after the call

    val = strtol(str, &endptr, 10);

    /* Check for possible errors */
    if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN)) || (errno != 0 && val == 0)) {
        perror("strtol");
        exit(EXIT_FAILURE);
    }

    if (endptr == str) {
        fprintf(stderr, "No digits were found\\n");
        exit(EXIT_FAILURE);
    }
```

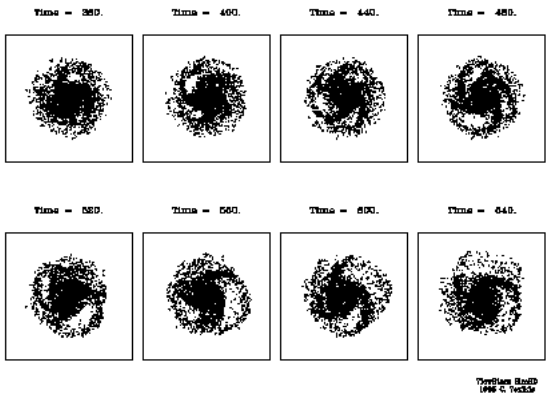
```
    /* If we are here, strtol() has converted a number correctly */
    return (int)val;
}
```

- **String to Integer Conversion Function:** Convert a string to an integer using `strtol`. Check for conversion errors and handle them appropriately.

```
/* Function that performs computation */
void bodyForce(Particle *p, float dt, int n) {
    for (int i = 0; i < n; i++) {
        float Fx = 0.0f;
        float Fy = 0.0f;
        float Fz = 0.0f;

        for (int j = 0; j < n; j++) {
            float dx = p[j].x - p[i].x;
            float dy = p[j].y - p[i].y;
            float dz = p[j].z - p[i].z;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }

        p[i].vx += dt * Fx;
        p[i].vy += dt * Fy;
        p[i].vz += dt * Fz;
    }
}
```

- **Body Force Computation Function:** Perform the computation of inter-body forces. For each particle, calculate the force contributions from all other particles and update the particle's velocity based on the computed forces.

This code represents a sequential implementation of the N-body problem simulation, where particles interact with each other through gravitational forces over a specified number of iterations. The program reads initial particle data from a file, performs the simulation, measures and prints timing information, optionally prints the initial and final states of particles, writes the final state to a file, and deallocates memory before termination.



The following image does not present the actual output of the decode but the graphical implementation of an N-Body simulation

Read.c

The following text pertains to a read file that is specifically designed to decode the binary output that is generated by the prior code. This decoded output is then converted into results that are easily readable and understandable, allowing users to make sense of the

processed data with greater ease. The primary purpose of this read file is to bridge the gap between complex binary code and user-friendly text, thereby ensuring that the key findings and outcomes of the code are accessible to all.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    float mass;
    float x, y, z;
    float vx, vy, vz;
} Particle;
```

- **Header Inclusions and Struct Definition:** Include necessary header files and define a struct named `Particle` to represent the properties of a particle in the simulation (mass, position, velocity).

```
int main() {
    FILE *inputFile = fopen("particles.txt", "rb");
    if (inputFile == NULL) {
        perror("Error opening file");
        return 1;
    }
```

- **File Opening:** Open a binary file named "particles.txt" for reading (`"rb"` mode). If unsuccessful, print an error message using `perror` and exit the program with a return code of 1.

```
    FILE *outputFile = fopen("readable_output.txt", "w");
    if (outputFile == NULL){
        perror("Error opening file");
        return 1;
    }
```

- **Output File Opening:** Open a text file named "readable_output.txt" for writing. If unsuccessful, print an error message using `perror` and exit the program with a return code of 1.

```
    Particle particle;
    size_t particlesRead;

    while ((particlesRead = fread(&particle, sizeof(Particle), 1, inputFile)) > 0) {
```

- **Particle Reading Loop:** Declare a `Particle` variable named `particle` and a variable `particlesRead` to store the number of particles read. Enter a loop that reads particles from the binary file using `fread`. The loop continues as long as particles are successfully read.

```
        fprintf(outputFile,"Mass: %f\\n", particle.mass);
        fprintf(outputFile,"Position: (%f, %f, %f)\\n", particle.x, particle.y, particle.z);
        fprintf(outputFile,"Velocity: (%f, %f, %f)\\n", particle.vx, particle.vy, particle.vz);
        fprintf(outputFile,"\\n");
    }
```

- **Output Formatting:** Inside the loop, write the particle data to the text file using `fprintf` to format the output with mass, position, and velocity information.

```
    fclose(inputFile);
    fclose(outputFile);

    printf("the particles file has been transferred from binary to readable data at readable_output.txt\\n");

    return 0;
}
```

- **File Closures and Program Exit:** Close both input and output files using `fclose`. Print a message indicating the successful transfer of the particle data from binary to readable format. Finally, return 0 to indicate successful program execution.

This program reads binary particle data from a file, converts it to human-readable format, and writes the results to a text file. It also provides informative error messages in case of file opening failures.

> 💡 The choice to store particle data in binary format at the beginning of the program is primarily motivated by efficiency and space considerations. Binary files are more compact and faster to read and write compared to human-readable text files. Storing data in binary allows for a direct mapping of the struct's memory layout, resulting in a straightforward and efficient transfer of particle information. This is crucial in scenarios like simulations, where large datasets are common, and the performance impact of file operations can be significant. While binary files may not be human-readable, their efficiency makes them suitable for storing and exchanging structured data efficiently in applications like scientific simulations.

Chapter 3: Parallelization

Parallelizing sequential N-body simulations is crucial for enhancing computational performance and addressing the increasing demands of large-scale simulations in various scientific and engineering domains. Various parallelization techniques are implemented, such as OpenMP, MPI, CUDA C, and pthreads, which provide solutions to expedite these simulations by distributing the workload among multiple processors, cores, or GPUs. This not only reduces the overall simulation time but also enables the simulation of more extensive datasets, leading to more accurate and realistic results.

> 💡 In the context of the N-body problem, dynamic parallelization is favored over static parallelization due to the unpredictable and dynamic nature of gravitational interactions between particles. The irregular workload distribution inherent in the N-body problem, where particles vary in proximity and gravitational influence, benefits from a dynamic approach. Dynamic parallelization allows processors or threads to adaptively handle tasks based on the evolving state of particle interactions, ensuring better load balancing. This flexibility is crucial for optimizing performance in simulations where gravitational forces dynamically change over time, preventing inefficiencies associated with statically assigned workloads. Overall, dynamic parallelization enhances efficiency by keeping processors consistently engaged in meaningful computations throughout the simulation.

OpenMP:

Allows for shared-memory parallelism by employing multiple threads to concurrently compute particle interactions. Well-suited for multi-core processors.

1. **Command-Line Arguments:**

```
if (argc < 3) {
    printf("Usage: %s <num_particles> <num_threads>\\n", argv[0]);
    exit(EXIT_FAILURE);
}
```

   - **Explanation:** The program expects two command-line arguments - the number of particles (`num_particles`) and the number of threads (`num_threads`). If these parameters are not provided, the program displays a usage message and exits.

2. **Initialization and Memory Allocation:**

```
int nBodies = convertStringToInt(argv[1]);
int numThreads = convertStringToInt(argv[2]);

omp_set_num_threads(numThreads);
```

```
Particle* particles = (Particle*)malloc(nBodies * sizeof(Particle));
```

- **Explanation:** The number of particles and threads are extracted from the command-line arguments. OpenMP's `omp_set_num_threads` function is used to set the number of threads for parallel execution. Memory is allocated for the particle array.

3. **Reading Initial Particle Data:**

```
FILE* fileRead = fopen("particles.txt", "r");
```

- **Explanation:** The program attempts to open the file "particles.txt" to read initial particle data.

4. **Parallel Particle Interaction Calculation:**

1. Force Calculation Loop:

```
#pragma omp parallel for
for (int i = 0; i < nBodies; i++) {
    float Fx = 0.0f;
    float Fy = 0.0f;
    float Fz = 0.0f;

    for (int j = 0; j < nBodies; j++) {
        if (i != j) {
            // Calculate gravitational forces between particles i and j
            float dx = particles[j].x - particles[i].x;
            float dy = particles[j].y - particles[i].y;
            float dz = particles[j].z - particles[i].z;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }
    }

    // Update velocities after the inner loop
    particles[i].vx += dt * Fx;
    particles[i].vy += dt * Fy;
    particles[i].vz += dt * Fz;
}
```

- **Outer Loop (`#pragma omp parallel for`):**
  - The outer loop iterates over all particles (`nBodies` iterations).
  - OpenMP parallelizes this loop by distributing iterations across available threads.
  - Each thread processes a subset of particles independently.
- **Inner Loop (Force Calculation):**
  - The inner loop iterates over all particles again to calculate gravitational forces between particle `i` and other particles (`nBodies` iterations).
  - Forces are accumulated in local variables `Fx`, `Fy`, and `Fz` for each particle `i`.
- **Data Dependencies:**
  - No data dependencies within the inner loop, allowing for parallel execution.
  - Forces are accumulated independently for each particle.
- **Velocity Update:**
  - Velocities (`vx`, `vy`, `vz`) are updated after the inner loop using the accumulated forces.

summery of the force calculation loop

1. **Outer Loop (i)**: The outer loop represents different particles. Each iteration of the outer loop deals with one specific particle, and the calculations for different particles are independent of each other. Therefore, we can parallelize this loop by assigning different particles to different threads. Each thread works on a separate particle, and they can work concurrently without interfering with each other.
    a. **Inner Loop (j)**: The inner loop represents the calculation of gravitational forces between the current particle (i) and all other particles (j). The forces are accumulated for a single particle (i) during this process. The reason we typically don't parallelize this loop is that the forces being calculated for one particle depend on the positions of all other particles. If we parallelize this loop, threads might interfere with each other's calculations, leading to incorrect results or increased complexity in managing dependencies.

2. Position Update Loop:

```
cCopy code
#pragma omp parallel for
for (int i = 0; i < nBodies; i++) {
    // Integrate position
    particles[i].x += particles[i].vx * dt;
    particles[i].y += particles[i].vy * dt;
    particles[i].z += particles[i].vz * dt;
}
```

Details:

- **Outer Loop (`#pragma omp parallel for`):**
    - Similar to the first loop, the outer loop iterates over all particles (`nBodies` iterations).
    - OpenMP parallelizes this loop to distribute iterations across threads.
- **Position Update:**
    - Inside the loop, the position of each particle is updated based on its velocity and the time step.
- **Data Dependencies:**
    - No data dependencies within the loop, allowing for parallel execution.
    - Each particle's position is updated independently.

Parallelization Strategy:

- Both loops use `#pragma omp parallel for` to parallelize the outer loops.
- Data dependencies within the loops are managed appropriately to avoid race conditions.
- The parallelization strategy is focused on distributing the workload across threads by assigning subsets of particles to each thread for independent processing.

```
#pragma omp parallel for
for (int i = 0; i < nBodies; i++) {
    // ... (inner loop calculations)
}
```

- **Explanation:** The outer loop, responsible for updating particle velocities, is parallelized using OpenMP. Each thread works on a subset of particles, calculating the forces exerted by other particles in parallel.

5. **Parallel Particle Position Update:**

```
#pragma omp parallel for
for (int i = 0; i < nBodies; i++) {
    // ... (position update calculations)
}
```

- **Explanation:** Similar to the previous step, the loop responsible for updating particle positions is parallelized. Each thread works on a subset of particles.

6. **Timing Measurements:**

```
double startTotal = omp_get_wtime();
// ... (simulation and timing code)
double endTotal = omp_get_wtime();
```

- **Explanation:** The wall-clock time is measured before and after the simulation to calculate the total time taken. Timing code is wrapped around the main simulation loop.

7. **File Output and Memory Cleanup:**

```
FILE* fileWrite = fopen("openmp_output.txt", "w");
if (fileWrite != NULL) {
    fwrite(particles, sizeof(Particle) * nBodies, 1, fileWrite);
    fclose(fileWrite);
}
free(particles);
```

- **Explanation:** The final particle data is written to "openmp_output.txt" in binary format. Memory allocated for particles is freed.

```
Number of threads used: 16
-------------------------------------------------------
Iteration 1 of 10 completed in 1.004642 seconds
Iteration 2 of 10 completed in 0.997752 seconds
Iteration 3 of 10 completed in 1.018317 seconds
Iteration 4 of 10 completed in 0.989729 seconds
Iteration 5 of 10 completed in 1.018777 seconds
Iteration 6 of 10 completed in 1.029764 seconds
Iteration 7 of 10 completed in 1.007296 seconds
Iteration 8 of 10 completed in 0.976287 seconds
Iteration 9 of 10 completed in 0.987650 seconds
Iteration 10 of 10 completed in 0.954465 seconds

Avg iteration time: 0.998580 seconds
Total time: 9.985801 seconds
Number of particles: 9000
Number of threads used: 16
-------------------------------------------------------
Iteration 1 of 10 completed in 1.311341 seconds
Iteration 2 of 10 completed in 1.188607 seconds
Iteration 3 of 10 completed in 1.237679 seconds
Iteration 4 of 10 completed in 1.121422 seconds
Iteration 5 of 10 completed in 1.166348 seconds
Iteration 6 of 10 completed in 1.226453 seconds
Iteration 7 of 10 completed in 1.222988 seconds
Iteration 8 of 10 completed in 1.216993 seconds
Iteration 9 of 10 completed in 1.163461 seconds
Iteration 10 of 10 completed in 1.186921 seconds

Avg iteration time: 1.204335 seconds
Total time: 12.043348 seconds
Number of particles: 10000
Number of threads used: 16
-------------------------------------------------------
-------------------------------------------------------
```

This parallelized version aims to distribute the computational workload among multiple threads using OpenMP, improving the efficiency of the N-body simulation.

MPI (Message Passing Interface):

Enables parallel computing in distributed-memory environments by allowing communication between separate processes. MPI is suitable for large-scale clusters or supercomputers.

1. **MPI Initialization:**

```
MPI_Init(&argc, &argv);
MPI_Type_contiguous(7, MPI_FLOAT, &particle_type);
MPI_Type_commit(&particle_type);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
```

- **Explanation:** MPI is initialized, a custom MPI data type for the `Particle` structure is created, and the number of tasks (processors) and the rank of the current task are obtained. The MPI barriers ensure synchronization among processes.

2. **Equal Workload Distribution:**

```
dim_portions = (int*)malloc(sizeof(int) * numtasks);
displ = (int*)malloc(sizeof(int) * numtasks);
compute_equal_workload_for_each_task(dim_portions, displ, num_particles, numtasks);
```

- **Explanation:** The workload is distributed evenly among MPI processes. The `dim_portions` array represents the size of the workload portion for each process, and the `displ` array represents the starting offset of the workload portion for each process.

3. **Particle Initialization:**

```
particles = (Particle*)malloc(num_particles * sizeof(Particle));
my_portion = (Particle*)malloc(sizeof(Particle) * dim_portions[myrank]);
gathered_particles = (myrank == MASTER) ? (Particle*)malloc(sizeof(Particle) * num_particles) : NULL;
```

- **Explanation:** Memory is allocated for the particle arrays. Each process has its own portion (`my_portion`) of particles.

4. **Main Simulation Loop:**

```
for (int iteration = 0; iteration < I; iteration++) {
    // ... (iteration-specific code)
}
```

- **Explanation:** The main simulation loop iterates over a specified number of iterations (`I`).

5. **Reading Initial Particle Data or Broadcasting:**

```
if (iteration == 0) {
    // Read initial state of particles from file in the first iteration
    // ... (file I/O code)
} else {
    // Broadcast particles from MASTER to all other processes
    MPI_Bcast(particles, num_particles, particle_type, MASTER, MPI_COMM_WORLD);
}
```

- **Explanation:** In the first iteration, the MASTER process reads the initial state of particles from a file. In subsequent iterations, the MASTER process broadcasts the particle data to all other processes.

6. **Parallel Particle Computation:**

```
bodyForce(particles, displ[myrank], dt, dim_portions[myrank], num_particles);
```

- **Explanation:** The `bodyForce` function calculates the forces and updates velocities for the particles in the portion assigned to each process in parallel.

7. **Gathering Results:**

```
MPI_Gatherv(particles + displ[myrank], dim_portions[myrank], particle_type, gathered_particles, dim_portions, displ, particle_type,
MASTER, MPI_COMM_WORLD);
if (myrank == MASTER) particles = gathered_particles;
```

- **Explanation:** The results from each process are gathered into the `gathered_particles` array on the MASTER process. The MASTER process then uses this array as the main particle array.

8. **Timing and Output:**

```
MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
if (myrank == MASTER) {
    // Output timing information and write final state of particles to a file
    // ... (output code)
}
```

- **Explanation:** MPI barriers ensure all processes finish their tasks before measuring the end time. The MASTER process calculates and outputs the total time and average iteration time. It also writes the final state of particles to a file.

9. **MPI Finalization:**

```
MPI_Finalize();
```

- **Explanation:** MPI is finalized, and memory allocated for particle arrays is freed.

This parallelized version uses MPI to distribute the workload among multiple processes, improving the efficiency of the N-body simulation.

CUDA C:

Tailored for NVIDIA GPUs, CUDA offloads parallelable tasks to the GPU, taking advantage of its parallel processing capabilities and accelerating simulations. The CUDA c implementation, although it achieved high speeds over the specified iterations, was very inefficient, and the main reason comes from the fact that we were not able to test the code locally, so we used Google Colab to run the program. However, in Google Colab, running CUDA C code may face challenges due to hardware and environment limitations. Colab provides GPU resources, but direct file I/O and memory management between the CPU and GPU can be complex. The platform's temporary file system and potential restrictions on accessing low-level GPU features may affect the execution of CUDA code. In our case, the particles were generated in binary format leading to the most efficient utilization as it involves lower serialization overhead, more compact data (smaller sizes), and more importantly, the binary data was outputted to reflect a direct memory representation of the data that is the particle’s velocity, position and mass. Other reasons include lower bandwidth usage, reduced I/O, and improved cache utilization.

Nevertheless, for this experiment, we did implement a parallel CUDA program for his simulation but read human-readable files instead of binary code. This came at the cost of running the program over the full 1024 so the comparison of speedup is not equivalent as for threads, mpi, and openMP implementation, we are using a fixed number of processors and threads to compare (4 processors for MPI and 4 threads for openMP and threads)

Code:

first we parallelized the particle generation part:

particle_cuda.cu

```
%%writefile particles_cuda.cu

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>
#include <curand.h>

typedef struct {
    float mass;
    float x, y, z;
    float vx, vy, vz;
} Particle;
```

```
__global__ void randomizeParticlesKernel(Particle *particles, int n, unsigned int seed) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        curandState state;
        curand_init(seed, i, 0, &state);

        for (int j = 0; j < 100000; ++j) {  // Increase computation per thread
            particles[i].mass = 2.0;
            particles[i].x = 2.0f * curand_uniform(&state) - 1.0f;
            particles[i].y = 2.0f * curand_uniform(&state) - 1.0f;
            particles[i].z = 2.0f * curand_uniform(&state) - 1.0f;
            particles[i].vx = 2.0f * curand_uniform(&state) - 1.0f;
            particles[i].vy = 2.0f * curand_uniform(&state) - 1.0f;
            particles[i].vz = 2.0f * curand_uniform(&state) - 1.0f;
        }
    }
}

extern "C" void run_cuda_code(int num_particles, Particle *particles, int threads_per_block, int *max_threads, int *block_size) {
    Particle *d_particles;

    // Allocate device memory for particles
    cudaMalloc((void**)&d_particles, num_particles * sizeof(Particle));

    // Randomize particles on the GPU
    int blocks_per_grid = (num_particles + threads_per_block - 1) / threads_per_block;
    unsigned int seed = 0; // Seed for random number generation

    randomizeParticlesKernel<<<blocks_per_grid, threads_per_block>>>(d_particles, num_particles, seed);

    // Copy the results back to the host
    cudaMemcpy(particles, d_particles, num_particles * sizeof(Particle), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_particles);

    // Set the max threads and block size
    *max_threads = blocks_per_grid * threads_per_block;
    *block_size = blocks_per_grid;
}

// Wrapper function to get the max threads per block
extern "C" int get_max_threads_wrapper() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    return prop.maxThreadsPerBlock;
}
```

Here is a detailed explanation of the code

Kernel Function (`randomizeParticlesKernel`):

- **Parallelism Specification:**
  - The kernel is designed to be executed in parallel by multiple threads. Each thread has a unique identifier, calculated as `i = blockIdx.x * blockDim.x + threadIdx.x`. This identifier (`i`) corresponds to the index of the particle that the thread will process.
- **Random Number Generation:**
  - The `curand` library is used to initialize a random number generator state (`curandState`) for each thread. The state is initialized with a seed and the thread index.
- **Particle Randomization:**
  - The kernel then enters a loop that iterates 100,000 times, and during each iteration, it randomizes properties of the particle associated with the thread.
- **Independent Threads:**
  - Importantly, each thread operates independently, handling a separate particle. This is a key aspect of parallelism in CUDA.

Host Code (`run_cuda_code`):

- **Memory Allocation:**
  - Device memory is allocated for the particles using `cudaMalloc`. This memory (`d_particles`) is used for storing the randomized particle data on the GPU.
- **Kernel Launch:**
  - The `randomizeParticlesKernel` is launched on the GPU using `<<<blocks_per_grid, threads_per_block>>>`. This configuration specifies the number of thread blocks (`blocks_per_grid`) and the number of threads per block (`threads_per_block`).
- **Memory Copy:**
  - After the kernel execution, the randomized particle data is copied from the device memory back to the host memory using `cudaMemcpy`.
- **Memory Deallocation:**
  - Device memory is freed using `cudaFree` to prevent memory leaks.
- **Return Values:**
  - The calculated maximum threads per block (`max_threads`) and block size (`block_size`) are returned through pointers. These values provide information about the parallel execution configuration used in the kernel launch.

Device Query Function (`get_max_threads_wrapper`):

- **Device Query:**
  - The function queries the CUDA device properties using `cudaGetDeviceProperties` to obtain information about the device, including the maximum number of threads per block (`maxThreadsPerBlock`).

Overall Workflow:

1. The host prepares data and allocates memory for the GPU (`cudaMalloc`).
2. The host launches the CUDA kernel, specifying the organization of threads (`<<<blocks_per_grid, threads_per_block>>>`).
3. The kernel executes in parallel on the GPU, with each thread handling a distinct particle and performing randomization.
4. The host copies the results (randomized particles) from the GPU back to the host memory (`cudaMemcpy`).
5. The host frees the allocated device memory (`cudaFree`).
6. The host queries and returns device properties, including the maximum threads per block.

In summary, parallelism is achieved through the execution of the `randomizeParticlesKernel` on the GPU, where each thread operates independently on a separate particle. The host manages memory, launches the kernel, and retrieves device properties, orchestrating a coordinated interaction between the CPU and GPU

now that we have our particle production code, we compile and run the code as follows

1. compile:

```
!nvcc -arch=sm_75 -lcudart -lcurand -Xcompiler -fPIC -shared -o particles_cuda.so particles_cuda.cu
```

note: `arch=sm_75` : This flag specifies the target GPU architecture for which the code is compiled. In this case, it's targeting an architecture with compute capability 7.5. You might adjust this flag based on the capabilities of your GPU.

2. run:

```
import ctypes
import numpy as np
import time
```

```
# Load the shared library
particles_cuda = ctypes.CDLL('./particles_cuda.so')

# Set the argument types for the run_cuda_code function
particles_cuda.run_cuda_code.argtypes = [
    ctypes.c_int,
    np.ctypeslib.ndpointer(dtype=np.dtype([('mass', np.float32), ('x', np.float32), ('y', np.float32), ('z', np.float32), ('vx', np.float32
    ctypes.c_int,  # threads_per_block
    ctypes.POINTER(ctypes.c_int),  # max_threads (output)
    ctypes.POINTER(ctypes.c_int)   # block_size (output)
]


# Define the number of particles
num_particles = 10000

# Initialize variables to store output values
max_threads = ctypes.c_int()
block_size = ctypes.c_int()

# Choose a number of threads per block (adjust as needed)
threads_per_block = 256

# Create an array to store particle data
particles = np.empty((num_particles,), dtype=np.dtype([('mass', np.float32), ('x', np.float32), ('y', np.float32), ('z', np.float32), ('vx'


for threads_per_block in [128, 256, 512, 1024]:
    for particles_per_thread in [1, 2, 4]:
      # Measure start time
        start_time = time.time()
        num_particles_per_block = threads_per_block * particles_per_thread
        blocks_per_grid = (num_particles + num_particles_per_block - 1) // num_particles_per_block

        particles_cuda.run_cuda_code(
            num_particles, particles, threads_per_block, ctypes.byref(max_threads), ctypes.byref(block_size)
        )



        # Print relevant information
        print(f"Threads per Block: {threads_per_block}")
        print(f"Particles per Thread: {particles_per_thread}")
        print(f"Number of Particles Generated: {num_particles}")
        print(f"Number of Threads Used: {max_threads.value}")
        print(f"Block Size: {block_size.value}")
        # Save the output to a text file
        output_filename = 'particle_output.txt'
        np.savetxt(output_filename, particles, fmt='%f', header='mass x y z vx vy vz', comments='')

        # Measure end time
        end_time = time.time()

        # Calculate elapsed time
        elapsed_time = end_time - start_time

        print(f"Elapsed Time for Generation: {elapsed_time} seconds")

        print()


# Print a confirmation message
print(f"\nParticle data saved to {output_filename}")
```

this Python script simply explores different configurations for generating random particle data on the GPU using CUDA. It interacts with the CUDA code in the shared library, measures the performance for different thread configurations, and outputs the results to a text file. It repeats the test for specified number of iterations and block sizes if given. However, these times are irrelevant in our main concern; the actual simulation.

Cuda simulaiton

```
%%writefile cuda_simulation_modified.cu

#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <cmath>
#include <chrono>
#include <iomanip> // for std::setprecision

#define SOFTENING 1e-9f
#define I 10

typedef struct {
    float mass;
    float x, y, z;
    float vx, vy, vz;
} Particle;

const float dt = 0.01f; // time step

__global__ void bodyForce(Particle *particles, int num_particles) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < num_particles) {
        float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;

        for (int j = 0; j < num_particles; j++) {
            float dx = particles[j].x - particles[i].x;
            float dy = particles[j].y - particles[i].y;
            float dz = particles[j].z - particles[i].z;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }

        particles[i].vx += dt * Fx;
        particles[i].vy += dt * Fy;
        particles[i].vz += dt * Fz;
    }
}

void readFile(const std::string &filename, std::vector<Particle> &particles) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        exit(EXIT_FAILURE);
    }

    std::string line;
    while (std::getline(file, line)) {
        std::istringstream iss(line);
        Particle p;
        iss >> p.mass >> p.x >> p.y >> p.z >> p.vx >> p.vy >> p.vz;
        particles.push_back(p);
    }

    file.close();
}

void saveParticleData(const std::vector<Particle> &particles, int iteration, std::ofstream &outputFile) {
    for (int i = 0; i < particles.size(); i++) {
        if (iteration == 1) {
            outputFile << "Particle " << i + 1 << ":\n";
            outputFile << "  Original data (before running the simulation): mass=" << particles[i].mass << ", position=(" << particles[i].x
        } else {
            outputFile << "  Data after iteration " << iteration - 1 << ": mass=" << particles[i].mass << ", position=(" << particles[i].x
        }
    }
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
```

```
        std::cerr << "Usage: " << argv[0] << " <number_of_particles>\n";
        return EXIT_FAILURE;
    }

    int num_particles = std::stoi(argv[1]);
    std::cout << "Number of particles: " << num_particles << std::endl;

    // Allocate host memory
    std::vector<Particle> particles;
    particles.resize(num_particles);

    // Read particle data from file
    readFile("particle_output.txt", particles);

    // Allocate device memory
    Particle *d_particles;
    cudaMalloc((void **)&d_particles, sizeof(Particle) * num_particles);

    // Copy data from host to device
    cudaMemcpy(d_particles, particles.data(), sizeof(Particle) * num_particles, cudaMemcpyHostToDevice);

    // Launch kernel
    dim3 blockDim(4);
    dim3 gridDim((num_particles + blockDim.x - 1) / blockDim.x);

    std::cout << "Block size: " << blockDim.x << ", Grid size: " << gridDim.x * blockDim.x << " (number of threads)\n";

    // Timing variables
    auto start_time = std::chrono::high_resolution_clock::now();

    // Open output file for writing particle data
    std::ofstream outputFile("particle_simulation_output.txt");
    if (!outputFile.is_open()) {
        std::cerr << "Error opening output file for particle data" << std::endl;
        exit(EXIT_FAILURE);
    }

    for (int iteration = 1; iteration <= I; iteration++) {
        auto iter_start_time = std::chrono::high_resolution_clock::now();

        bodyForce<<<gridDim, blockDim>>>(d_particles, num_particles);
        cudaDeviceSynchronize();

        // Copy data from device to host
        cudaMemcpy(particles.data(), d_particles, sizeof(Particle) * num_particles, cudaMemcpyDeviceToHost);

        // Save particle data for the current iteration
        saveParticleData(particles, iteration, outputFile);

        auto iter_end_time = std::chrono::high_resolution_clock::now();
        auto iter_duration = std::chrono::duration_cast<std::chrono::microseconds>(iter_end_time - iter_start_time).count() / 1e6; // Conve
        std::cout << "Iteration " << iteration << " of " << I << " completed in " << std::fixed << std::setprecision(5) << iter_duration <<
    }

    // Close output file
    outputFile.close();

    // Calculate and print average time
    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time).count() / 1e6; // Convert to seconds
    std::cout << "Avg iteration time: " << std::fixed << std::setprecision(5) << duration / I << " seconds\n";
    std::cout << "Total time: " << std::fixed << std::setprecision(5) << duration << " seconds\n";

    // Free device memory
    cudaFree(d_particles);

    return 0;
}
```

here is the full breakdown of the code:

Constants and Data Structures:

```
cppCopy code
#define SOFTENING 1e-9f
#define I 10

typedef struct {
    float mass;
    float x, y, z;
    float vx, vy, vz;
} Particle;
```

- `SOFTENING`: A constant used to prevent singularities in the force calculation.
- `I`: The number of iterations for the simulation.
- `Particle`: A struct representing a particle's mass, position (x, y, z), and velocity (vx, vy, vz).

Kernel Function (`bodyForce`):

```
cppCopy code
__global__ void bodyForce(Particle *particles, int num_particles) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < num_particles) {
        float Fx = 0.0f, Fy = 0.0f, Fz = 0.0f;

        for (int j = 0; j < num_particles; j++) {
            float dx = particles[j].x - particles[i].x;
            float dy = particles[j].y - particles[i].y;
            float dz = particles[j].z - particles[i].z;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }

        particles[i].vx += dt * Fx;
        particles[i].vy += dt * Fy;
        particles[i].vz += dt * Fz;
    }
}
```

- **Parallelism Specification:**
  - The kernel is executed in parallel by multiple threads. Each thread processes one particle, and the total number of threads is determined by the grid and block dimensions.
- **Parallel Execution:**
  - Each thread (identified by `i`) calculates the gravitational forces acting on the particle at index `i` due to all other particles in the system.
- **Force Calculation:**
  - The gravitational force between particles is calculated using the standard formula: force=distance3massj×distance_vector.

$$\text{force} = \frac{\text{mass}_j}{\text{distance}^3} \times \text{distance\_vector}$$

  - The loop iterates over all particles to calculate the total force on the particle at index `i`.
- **Integration:**
  - The calculated forces are used to update the velocity of the particle in the x, y, and z directions.

File I/O and Helper Functions:

```
cppCopy code
void readFile(const std::string &filename, std::vector<Particle> &particles) {
    // ...
}

void saveParticleData(const std::vector<Particle> &particles, int iteration, std::ofstream &outputFile) {
    // ...
}
```

- `readFile`: Reads particle data from a file into a vector of `Particle` structs.
- `saveParticleData`: Saves particle data to an output file for each iteration of the simulation.

Main Function:

```
cppCopy code
int main(int argc, char *argv[]) {
    // ...
}
```

- **Command Line Argument Parsing:**
  - The main function expects the number of particles as a command line argument.
- **Host Memory Allocation and Data Initialization:**
  - Host memory is allocated for storing particle data, and the data is read from a file.

```
cppCopy code
// Read particle data from file
readFile("particle_output.txt", particles);
```

- **Device Memory Allocation and Data Transfer:**
  - Device memory is allocated for particles (`d_particles`), and particle data is copied from the host to the device.

```
cppCopy code
// Allocate device memory
Particle *d_particles;
cudaMalloc((void **)&d_particles, sizeof(Particle) * num_particles);

// Copy data from host to device
cudaMemcpy(d_particles, particles.data(), sizeof(Particle) * num_particles, cudaMemcpyHostToDevice);
```

- **Kernel Launch Configuration:**
  - The grid and block dimensions are configured to organize threads for efficient parallel execution.

```
cppCopy code
// Launch kernel
dim3 blockDim(4);
dim3 gridDim((num_particles + blockDim.x - 1) / blockDim.x);
```

- **Timing and Performance Measurements:**
  - The code measures the time taken for each iteration and calculates the average iteration time and total time.

```
cppCopy code
// Timing variables
auto start_time = std::chrono::high_resolution_clock::now();
```

- **Simulation Loop:**
  - The main loop iterates through the specified number of iterations (`I`).
  - For each iteration, the `bodyForce` kernel is launched to calculate forces on each particle.
  - Data is transferred back from the device to the host.
  - Particle data for the current iteration is saved to an output file.

```
cppCopy code
// Simulation Loop
for (int iteration = 1; iteration <= I; iteration++) {
    // ...
    bodyForce<<<gridDim, blockDim>>>(d_particles, num_particles);
    cudaDeviceSynchronize();

    // Copy data from device to host
    cudaMemcpy(particles.data(), d_particles, sizeof(Particle) * num_particles, cudaMemcpyDeviceToHost);

    // Save particle data for the current iteration
    saveParticleData(particles, iteration, outputFile);

    // ...
}
```

- **Device Memory Deallocation:**
  - Device memory is freed before exiting the program.

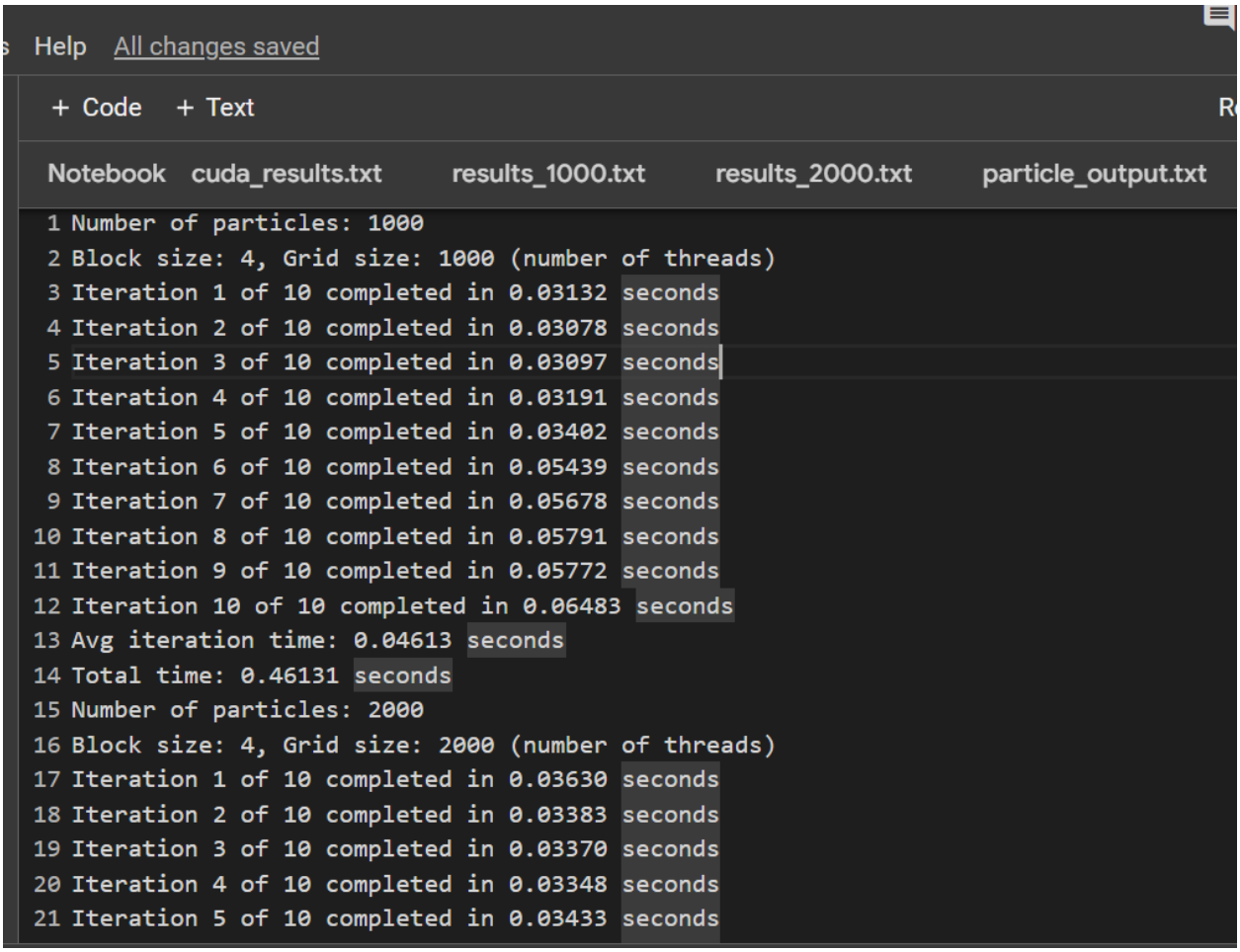
```
cppCopy code
// Free device memory
cudaFree(d_particles);
```

In this work, we have implemented a straightforward gravitational simulation that harnesses the power of parallel computing on the GPU. The primary goal of parallelization is to effectively distribute the computational load of gravitational force calculations among numerous threads, thereby capitalizing on the parallel processing prowess of the GPU. The decision to employ parallelism is in harmony with the inherent nature of gravitational force calculations, where each interaction operates independently, rendering it well-suited for concurrent execution. Additionally, the code incorporates timing measurements to offer a comprehensive understanding of the simulation's performance.

for running the code execute the following python script, which would save results to resutls.txt

```
for i in range(1000, 11000, 1000):
    compile_command = f"nvcc -o cuda_simulation_modified_{i} cuda_simulation_modified.cu"
    os.system(compile_command)

    run_command = f"./cuda_simulation_modified_{i} {i} >> results.txt"
    os.system(run_command)
```

results:



Pthreads (POSIX threads):

A threading library for parallelizing tasks in a multi-threaded environment. It can be utilized to parallelize computations within a shared-memory system.

1. **Struct Definitions:**

```
typedef struct {
    float mass;
    float x, y, z;
    float vx, vy, vz;
} Particle;

typedef struct {
    Particle* particles;
    float dt;
    int start;
    int end;
} ThreadData;
```

   - **Explanation:** Two structures are defined, one for representing particles (`Particle`) and another (`ThreadData`) for passing data to threads.

2. **Function Prototypes:**

```
int convertStringToInt(char* str);
void* bodyForceThread(void* arg);
void bodyForce(Particle* p, float dt, int start, int end);
void computeForces(Particle* particles, float dt, int n, int num_threads);
double getCurrentTime();
```

- **Explanation:** Prototypes for various functions used in the program.

3. **Main Function:**

```
int main(int argc, char* argv[]) {
    // ... (variable declarations)

    for (int iter = 1; iter <= nIters; iter++) {
        double startIter = getCurrentTime();
        computeForces(particles, dt, nBodies, num_of_threads);
        // ... (position integration code)
        double endIter = getCurrentTime();
        printf("Iteration %d of %d completed in %f seconds\\n", iter, nIters, endIter - startIter);
    }

    // ... (output and cleanup code)
}
```

- **Explanation:** The main function iterates through simulation steps, calling `computeForces` and updating particle positions.

4. **Compute Forces Function:**

```
void computeForces(Particle* particles, float dt, int n, int num_threads) {
    pthread_t threads[num_threads];
    ThreadData thread_data[num_threads];
    int chunk_size = n / num_threads;

    // Create threads to compute forces in parallel
    for (int i = 0; i < num_threads; i++) {
        // ... (thread data setup and thread creation)
    }

    // Wait for threads to finish
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }
}
```

- **Explanation:** This function creates threads to parallelize the computation of forces using the `bodyForceThread` function.

5. **Body Force Thread Function:**

```
void* bodyForceThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    bodyForce(data->particles, data->dt, data->start, data->end);
    return NULL;
}
```

- **Explanation:** This function is executed by each thread and calculates forces for a specified range of particles.

6. **Body Force Function:**

```
void bodyForce(Particle* p, float dt, int start, int end) {
    for (int i = start; i < end; i++) {
        // ... (force calculation and velocity update)
    }
}
```

- **Explanation:** This function calculates forces and updates velocities for particles in a specified range.

7. **Conversion and Timing Functions:**

```
int convertStringToInt(char* str);
double getCurrentTime();
```

   - **Explanation:** Utility functions for string to integer conversion and getting the current time.

8. **Timing and Output:**

```
// ... (timing and output code)
```

   - **Explanation:** Timing information is printed, and the final state of particles is written to an output file.

9. **Current Time Function:**

```
double getCurrentTime() {
    // ... (getting the current time)
}
```

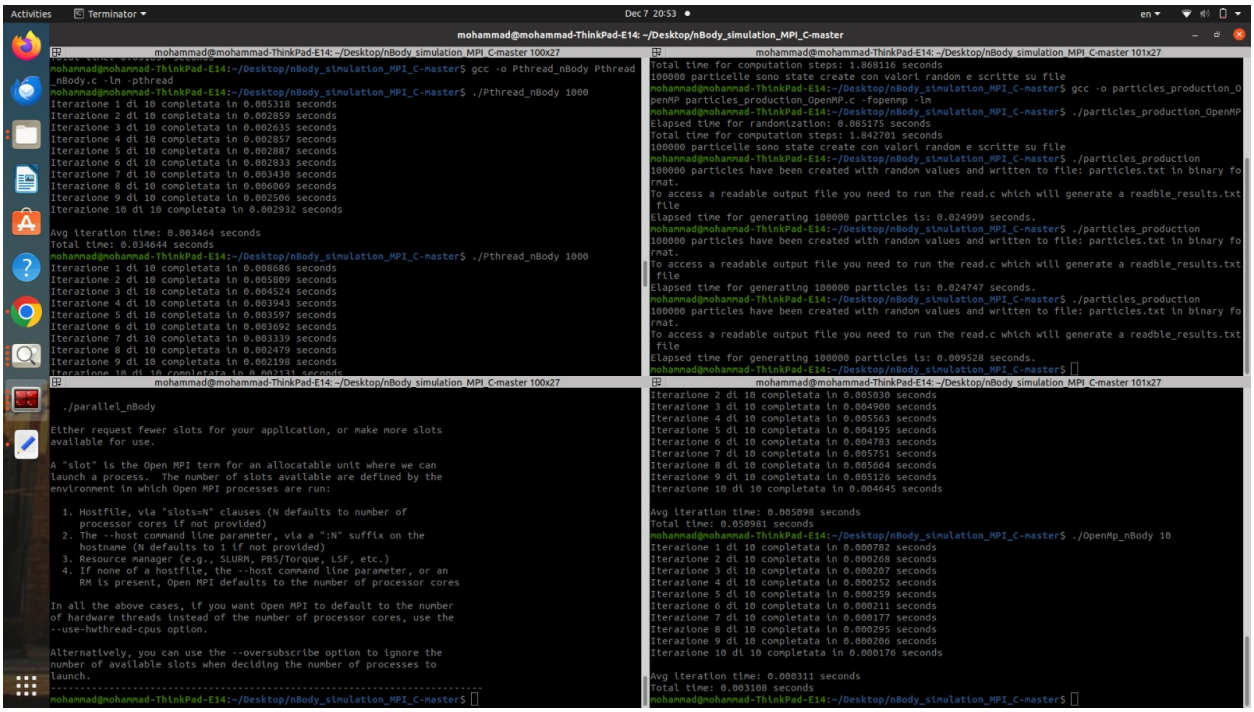
   - **Explanation:** Returns the current time using POSIX functions.

This code parallelizes the force computation step using pthreads, aiming to improve the efficiency of the N-body simulation by leveraging multiple threads for concurrent execution.

```
-----------------------------------------------------------
Iteration 1 of 10 completed in 0.564997 seconds
Iteration 2 of 10 completed in 0.521189 seconds
Iteration 3 of 10 completed in 0.597381 seconds
Iteration 4 of 10 completed in 0.602161 seconds
Iteration 5 of 10 completed in 0.539103 seconds
Iteration 6 of 10 completed in 0.590582 seconds
Iteration 7 of 10 completed in 0.512180 seconds
Iteration 8 of 10 completed in 0.542167 seconds
Iteration 9 of 10 completed in 0.546232 seconds
Iteration 10 of 10 completed in 0.535199 seconds

Avg iteration time: 0.555249 seconds
Total time: 5.552495 seconds
Number of particles: 9000
Number of threads used: 16

-----------------------------------------------------------
Iteration 1 of 10 completed in 0.711219 seconds
Iteration 2 of 10 completed in 0.599233 seconds
Iteration 3 of 10 completed in 0.718621 seconds
Iteration 4 of 10 completed in 0.702502 seconds
Iteration 5 of 10 completed in 0.704030 seconds
Iteration 6 of 10 completed in 0.699890 seconds
Iteration 7 of 10 completed in 0.720858 seconds
Iteration 8 of 10 completed in 0.780018 seconds
Iteration 9 of 10 completed in 0.611682 seconds
Iteration 10 of 10 completed in 0.533985 seconds

Avg iteration time: 0.678316 seconds
Total time: 6.783158 seconds
Number of particles: 10000
Number of threads used: 16

-----------------------------------------------------------
```

Parallelization techniques are crucial to utilize modern computing power and tackle complex simulations. They allow larger, higher resolution simulations, advancing fields like astrophysics and climate modeling. Thus, parallelization is key in enhancing simulation capabilities and scientific understanding.

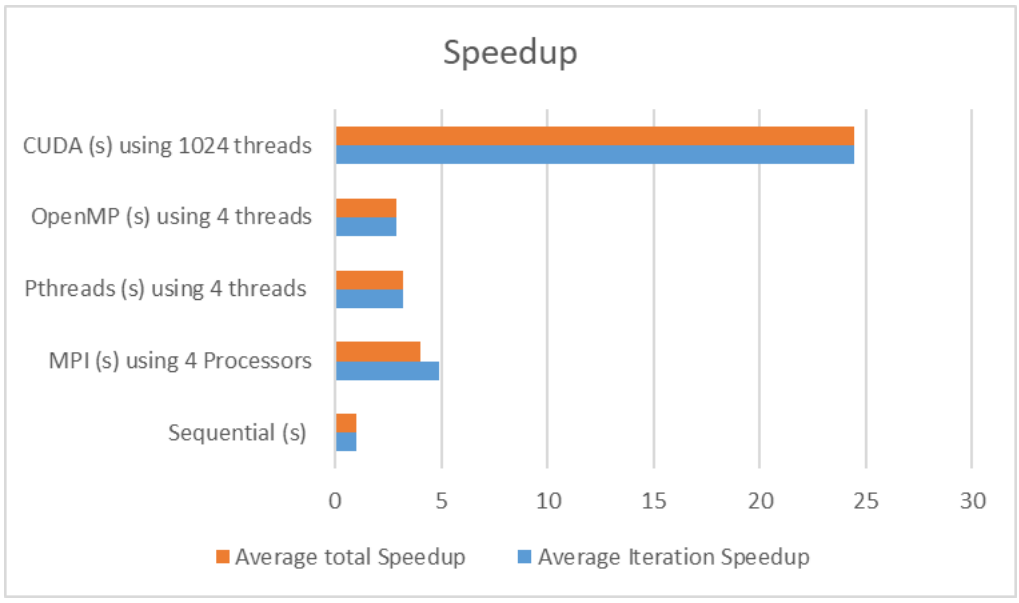
Chapter 4: Analysis

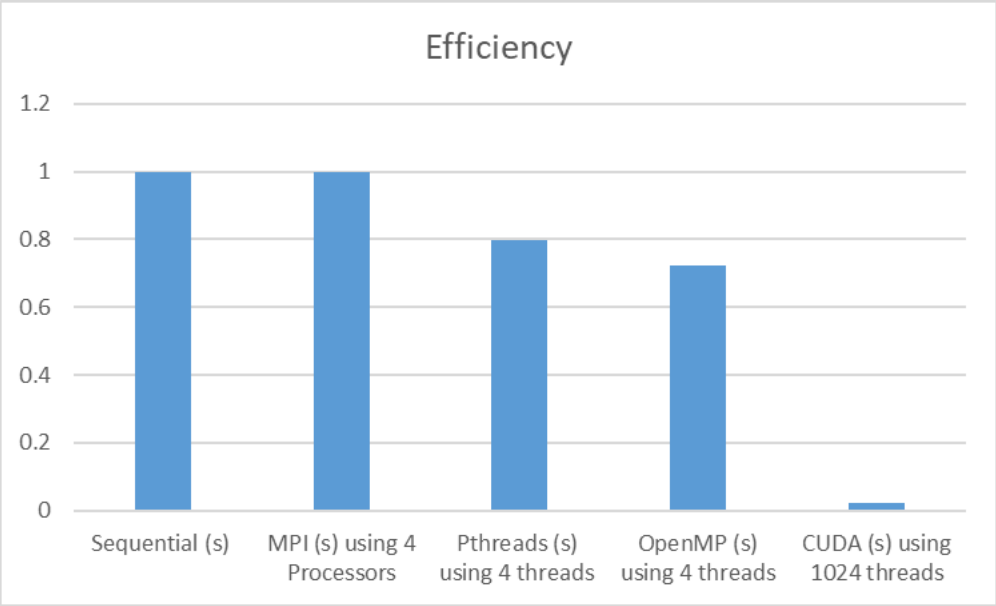


Particle Count		Sequential (s)	MPI (s) using 4 Processors	Pthreads (s) using 4 threads	OpenMP (s) using 4 threads	CUDA (s) using 1024 threads
	Total time	0.389752	0.071815	0.159906	0.166878	0.29439
1000	Average Iteration time	0.038975	0.007182	0.015991	0.016688	0.02944
2000	Average Iteration time	0.148888	0.024278	0.047949	0.052919	0.06049
	Total time	1.488884	0.242785	0.479492	0.529194	0.60488
3000	Average Iteration time	0.31054	0.024278	0.107405	0.106893	0.03487
	Total time	3.105401	0.242785	1.074049	1.068927	0.34872
4000	Average Iteration time	0.54639	0.075332	0.18277	0.179868	0.03806
	Total time	5.463896	0.753318	1.827695	1.79868	0.38055
5000	Average Iteration time	0.847457	0.183784	0.291604	0.21492	0.04177
	Total time	8.474565	1.837844	2.916043	2.149197	0.41774
6000	Average Iteration time	1.244638	0.279431	0.404155	0.39479	0.08564
	Total time	12.446375	2.794313	4.041546	3.947899	0.85639
7000	Average Iteration time	1.679806	0.364554	0.509514	0.61204	0.04626
	Total time	16.798058	3.645542	5.095143	6.1204	0.46257
8000	Average Iteration time	2.192645	0.464945	0.667515	0.766638	0.05128
	Total time	21.926448	4.649452	6.675146	7.666379	0.51278
9000	Average Iteration time	2.780224	0.565704	0.856375	1.013748	0.05408

	Total time	27.80224	5.657043	8.563749	10.137479	0.5408
10000	Average Iteration time	3.41919	0.70354	1.057683	1.215874	0.09908
	Total time	34.191901	7.035402	10.576826	12.158737	0.99077

	Sequential (s)	MPI (s) using 4 Processors	Pthreads (s) using 4 threads	OpenMP (s) using 4 threads	CUDA (s) using 1024 threads
Average Iteration Speedup	1	3.918329963	3.189779619	2.887551706	24.41679391
Average total Speedup	1	3.991313706	3.189780533	2.887552119	24.41728856
	Sequential (s)	MPI (s) using 4 Processors	Pthreads (s) using 4 threads	OpenMP (s) using 4 threads	CUDA (s) using 1024 threads
Efficiency	1	0.997828427	0.797445133	0.72188803	0.023845008





Average Iteration Time and Total Time

The average iteration time $T_{\text{iter}}, T_{\text{total}}$ for each parallelization paradigm are integral performance indicators. As the number of particles or iterations increases, these times naturally escalate. The CUDA implementation consistently outperforms other methods, as evidenced by its significantly lower iteration and total times, showcasing its capability to efficiently handle large-scale parallel tasks.

Speedup Analysis

Average Iteration Speedup:

The average iteration speedup S_{iter} is calculated as the ratio of the average iteration time for the sequential approach to the average iteration time for the parallel approach

$$S_{\text{iter}} = \frac{T_{\text{seq\_iter}}}{T_{\text{par\_iter}}}$$

The results demonstrate substantial speedup across all parallel paradigms, with CUDA exhibiting the most significant improvement. The formula provides a quantitative measure of the parallel efficiency for a single iteration.

Average Total Speedup:

The average total speedup S_{total} considers the overall performance improvement for the entire simulation duration:

$$S_{\text{total}} = \frac{T_{\text{seq\_total}}}{T_{\text{par\_total}}}$$

Here, T_{seq_total} and T_{par_total}represent the total time for the sequential and parallel approaches, respectively. CUDA stands out with an exceptional average total speedup, indicating its scalability and efficiency over the entire simulation.

Efficiency Analysis

Efficiency (*E*) measures the utilization of available resources and is calculated as the ratio of the speedup to the number of processors or threads:

$E = \frac{S}{P}$

Where *P* is the number of processors or threads. The efficiency values for MPI, Pthreads, OpenMP, and CUDA are generally less than 1, suggesting that these parallel paradigms are not fully utilizing available resources. The declining efficiency for CUDA with

larger simulations raises considerations about its scalability for extremely large datasets.

Scalability Inference

Scalability evaluates the program's ability to handle an increasing workload. It is often inferred from the speedup and efficiency metrics:

$$S_{\text{scalability}} = S_{\text{total}} \times E$$

The speedup and efficiency results suggest strong scalability for all parallel methods compared to the sequential approach. However, the decreasing efficiency for CUDA with larger simulations indicates potential limitations in scalability for extremely large datasets.

In summary, the analysis incorporating formulas and numerical values reinforces the superiority of CUDA in terms of speedup, emphasizing its efficiency in handling parallel gravitational simulations. However, the diminishing efficiency with larger simulations signals the need for further exploration and optimization in CUDA for enhanced scalability. This comprehensive examination provides valuable insights for further refinement and advancement of parallel gravitational simulations.

Recommendations:

- Different parallelization techniques can be more suitable depending on the nature of the hardware and the specific requirements of the task. The chosen technique can significantly influence the system's efficiency and performance.
- When choosing a parallelization technique, consider factors like implementation complexity, potential for future scalability, and the type of parallel architecture available—be it shared-memory, distributed-memory, or GPU-based systems.
- Additionally, optimizing and fine-tuning parameters within each parallelization approach can often lead to further performance gains. This can result in more efficient computational processes, maximizing the potential of the existing hardware and software.

Bibliography:

https://github.com/nicolaDeCristofaro/nBody_simulation_MPI_C

https://github.com/harrism/mini-nbody/blob/master/nbody.c