CSC 4005 Distributed and Parallel Programming
Assignment 3 N-body Simulation

Ryan Christopher – 119010507

School of Data Science

The Chinese University of Hong Kong, Shenzhen

119010507@link.cuhk.edu.cn

Project Introduction

N-Body Simulation is the problem we will use as the basis of this program and computations. We must model the motions of some items that are brought about by the gravitational pull between them in order to solve this problem. Only two dimensions of the simulation will be shown in order to make the issue simpler. The assumption that the objects are too small to collide with one another is another one we make. Any object, nevertheless, will always rebound if it strikes the simulation window's wall. As a result, the quantity of items won't alter over the course of the simulation.

This computing assignment tries to familiarize us with parallel programming through this problem. Considering the importance of parallel computing in solving practical problems, this paper will examine the performance of four different parallelization methods along with a sequential algorithm. These 4 methods of parallelization include Message Passing Interface (MPI), POSIX Threads (PThreads), and Open Multi-Processing (OpenMP) and CUDA. The kernel version I am currently using is version **3.10.0-862.el7.x86_64**.

Design Approach

All implementations will have a similar computing component. The simulation's items will initially be created at random by each application. We will generate a predetermined number of objects thanks to the randomness, but each object's characteristics will vary depending on the simulation. The duration of the simulation will then be determined by the number of computing iterations, during which the force experienced by each object will be determined using the mathematical expression displayed in Figure 1. Where F represent the force between two objects, G represent the gravitational constant which is used for the calculation. M1 and M2 for mass of each object correspondingly. Finally, R for the distance between object 1 and object 2

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

Figure 1. Formula for force

The formula in figure 1 is only used to calculate the force occurs between two objects. If we want to mimic a lot more things in our software. Thus, the following equation represents the total force experienced by any one object. Fi is the resultant force for object i. According to the formula in figure 2, the combined force of item I and all other objects is equal to the resultant force for object i. Using the following formula, we can determine the instantaneous velocity from the obtained resultant force.

$$\vec{F_i} = \sum_{j}^{n} G\frac{m_i \times m_j}{r^2}$$

Figure 2. Formula for Fi

We can ultimately determine each object's future position using the formula in figure 4, after getting the acceleration and velocity in figure 3. Every iteration of our program will include applying every formula we covered to every item, giving it an O(n2) time complexity. Only the implementation, which allows us to paralleled the computation of the objects, differs between each program. We shall go into further depth about each program's specific implementation after the formula below.

$$a_i = \frac{\vec{F_i}}{m_i}$$

$$v_i = a_i \times dt$$

Figure 3. Formula for Acceleration and Velocity

$$\text{pos}_{new}^{i} = \text{pos}_{old}^{i} + (a_i \times dt)$$

Figure 4. Formula for position computation

Note:

A is the acceleration of object i

V is the velocity of object i

Dt is the short period of time used

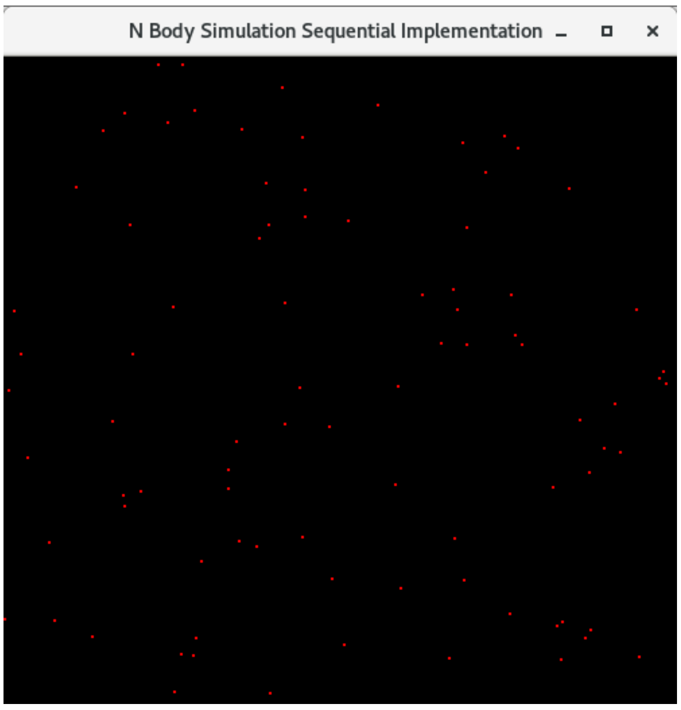
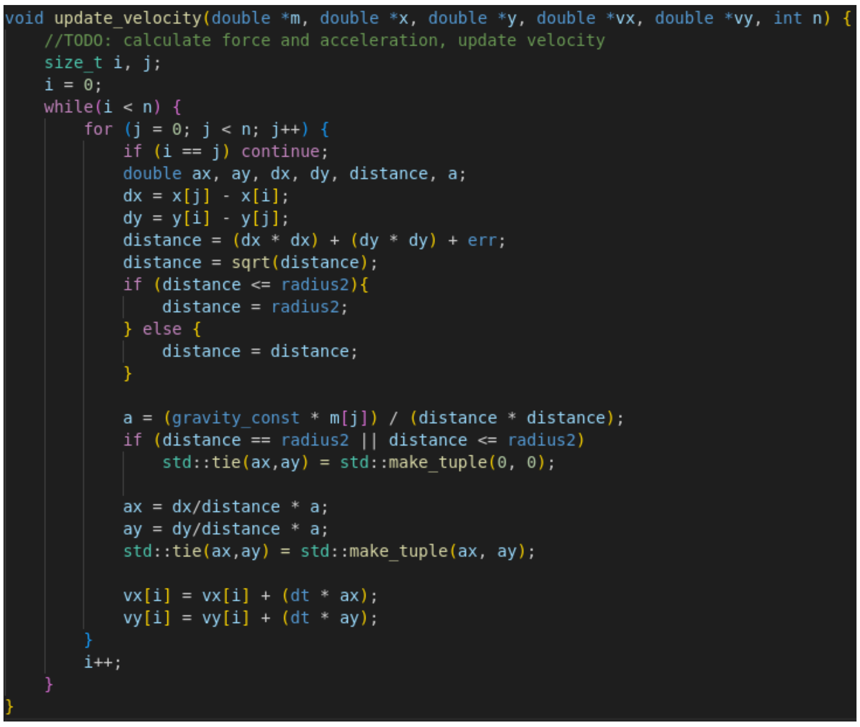


Figure 5. Program Output

Program Implementation

Sequential Programming:

Since we won't be parallelizing any code with a sequential technique, we may use this implementation as the benchmark to contrast with parallel implementations. After the objects are generated at random, the simulation is conducted sequentially using two nested loops (for performance evaluation, we clock the simulation before and after the outer loop). To denote the simulation iteration, use the outer loop (or simulation time). Another loop that iterates as many times as there are objects is included in this loop. Each body's calculation in a given iteration is carried out by this inner loop. To prevent position updates before all of the calculations are complete, the results of each body's calculations will be temporarily stored into another variable. We have another brief loop that iterates through all of the objects to update their positions after computing the future positions for each in the current iteration.



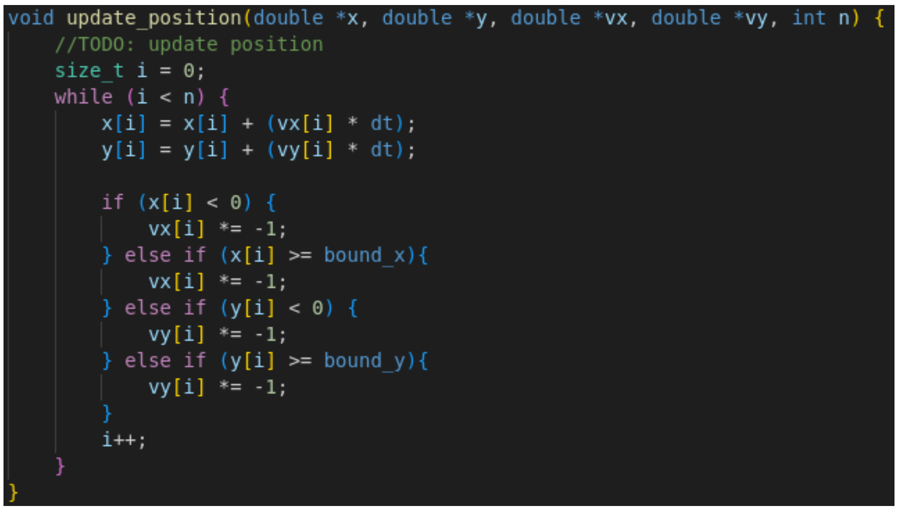
```
void update_velocity(double *m, double *x, double *y, double *vx, double *vy, int n) {
    //TODO: calculate force and acceleration, update velocity
    size_t i, j;
    i = 0;
    while(i < n) {
        for (j = 0; j < n; j++) {
            if (i == j) continue;
            double ax, ay, dx, dy, distance, a;
            dx = x[j] - x[i];
            dy = y[i] - y[j];
            distance = (dx * dx) + (dy * dy) + err;
            distance = sqrt(distance);
            if (distance <= radius2){
                distance = radius2;
            } else {
                distance = distance;
            }

            a = (gravity_const * m[j]) / (distance * distance);
            if (distance == radius2 || distance <= radius2)
                std::tie(ax,ay) = std::make_tuple(0, 0);

            ax = dx/distance * a;
            ay = dy/distance * a;
            std::tie(ax,ay) = std::make_tuple(ax, ay);

            vx[i] = vx[i] + (dt * ax);
            vy[i] = vy[i] + (dt * ay);
        }
        i++;
    }
}
```

Figure 6. Benchmark algorithm used to get acceleration value and update velocity



```
void update_position(double *x, double *y, double *vx, double *vy, int n) {
    //TODO: update position
    size_t i = 0;
    while (i < n) {
        x[i] = x[i] + (vx[i] * dt);
        y[i] = y[i] + (vy[i] * dt);

        if (x[i] < 0) {
            vx[i] *= -1;
        } else if (x[i] >= bound_x){
            vx[i] *= -1;
        } else if (y[i] < 0) {
            vy[i] *= -1;
        } else if (y[i] >= bound_y){
            vy[i] *= -1;
        }
        i++;
    }
}
```

Figure 7. Benchmark algorithm used to update particle position after collision

Message Passing Interface (MPI) Programming:

MPI is a parallelization method that is based on the distributed-memory architecture; hence, it will be quite hard to implement this simulation. Simply put, each object must access the characteristics of other objects in order to determine its next location. Therefore, the master process must transmit the generation result to all processes after randomly creating all of the objects (the creation occurs only in the master process); doing so also initiates the timer. In addition, various processors will be given the objects in this situation. Fortunately, this simplistic simulation does not take into account any item collisions that would cause objects to fragment into additional smaller things. As a result, the initial distribution of object masses to all processes is the sole one. The locations of the items, on the other hand, are different because they will all be modified after each cycle. All processes will start their simulation iteration, during which they compute the upcoming positions of all the objects assigned to them (the number of objects is regarded as being more than the number of processes), when the master process has completed all distributions. We won't require a synchronization barrier once we've completed the calculations for all objects in the current iteration because we'll be exchanging data among all processes. Blocking receive during this data sharing will inevitably synchronize all processes. Additionally, unlike a sequential technique, employing this function does not require us to update the position data from a temporary variable. The temporary variable can be used as the function's input, and the output can be sent back to the original variable. After this data exchange, we can carry on with the following simulation iteration till the simulation is finished (where we end the time measurement).

Note: The code for MPI Programming is not completely finished. However, the theory and implementation are understandable.

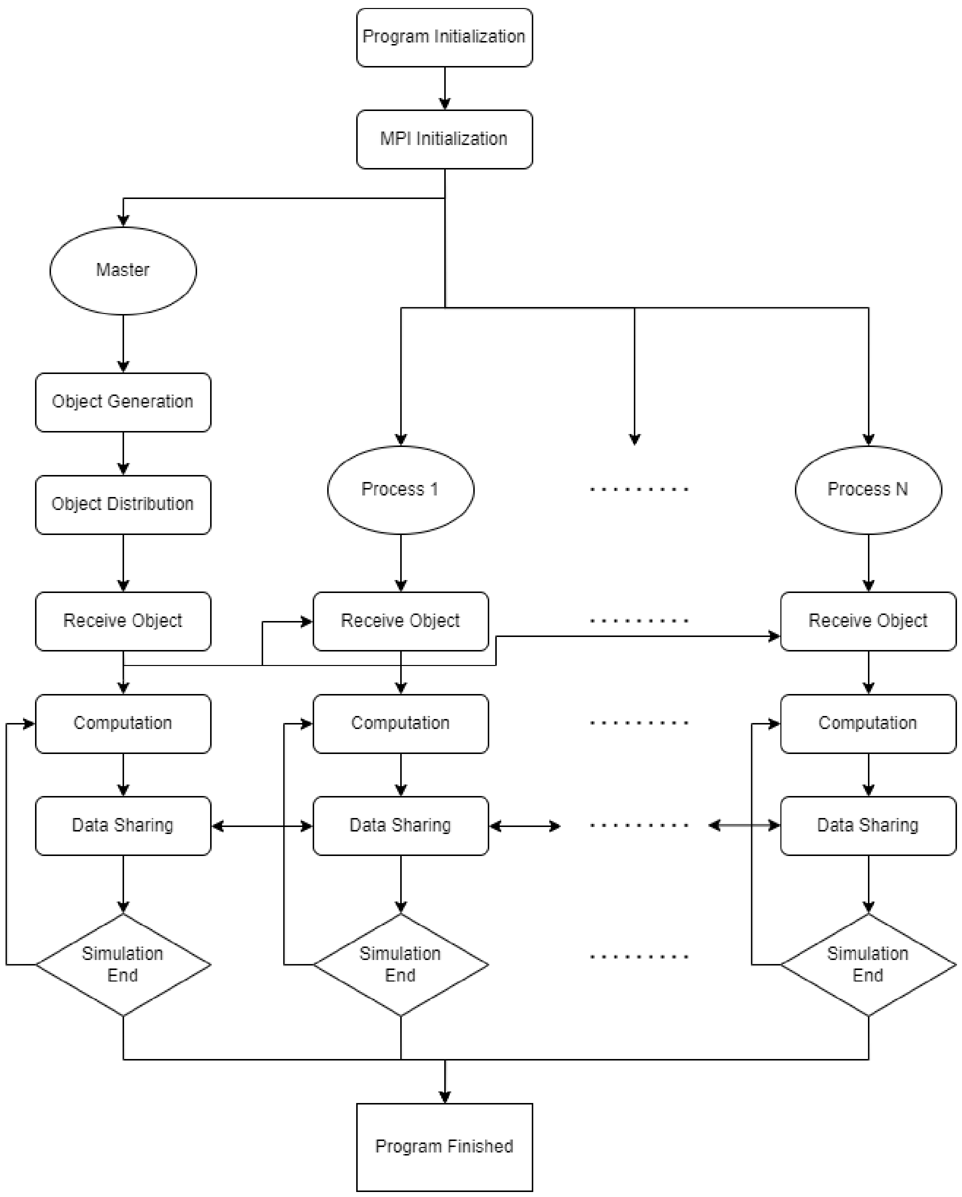
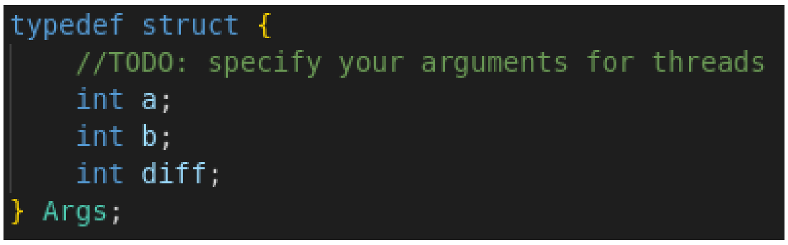


Figure 7 MPI Flow Diagram Implementation

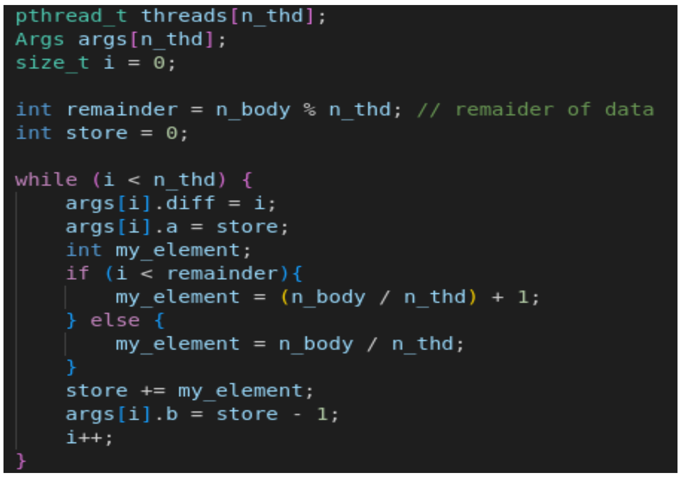
PThread Programming:

PThreads is a parallelization technique that uses shared memory architecture, as opposed to MPI. We may share data between threads thanks to this PThreads functionality. As a result, we can immediately fork some threads and assign the objects to the threads following object formation (we start to measure the time here). The simulation iteration can then begin on each thread right away without the requirement to distribute the items first. The data for the objects was placed in shared memory so that each thread could easily access it. The location calculations for all the objects assigned to each thread are then performed during each iteration. To prevent any calculation errors, the results of all calculations must still be saved in temporary variables (same reason as it is in sequential implementation). Any thread that has completed its calculations must then wait for all other threads to complete their calculations. Here, we use mutex and barrier to create a certain type of barrier. An initialized primitive integer variable with the value zero is required by the barrier operation. Any thread will lock the variable after completing its computations, increase it by one, unlock the variable, and then wait for an incoming signal. All threads follow this protocol, with the exception of the master thread (which is granted thread id 0). It has a loop that will continue until the variable's value equals the number of threads before breaking. It will transmit a signal to all threads after it has passed the loop, which signifies that all threads have arrived at the same location. Following this broadcasting, each thread will continue its iterations until the simulation's conclusion, at which point they are all brought together and the timer is turned off.



```c
typedef struct {
    //TODO: specify your arguments for threads
    int a;
    int b;
    int diff;
} Args;
```

Figure 8 Threads arguments in Struct



```c
pthread_t threads[n_thd];
Args args[n_thd];
size_t i = 0;

int remainder = n_body % n_thd; // remaider of data
int store = 0;

while (i < n_thd) {
    args[i].diff = i;
    args[i].a = store;
    int my_element;
    if (i < remainder){
        my_element = (n_body / n_thd) + 1;
    } else {
        my_element = n_body / n_thd;
    }
    store += my_element;
    args[i].b = store - 1;
    i++;
}
```

Figure 9. PThread Algorithm

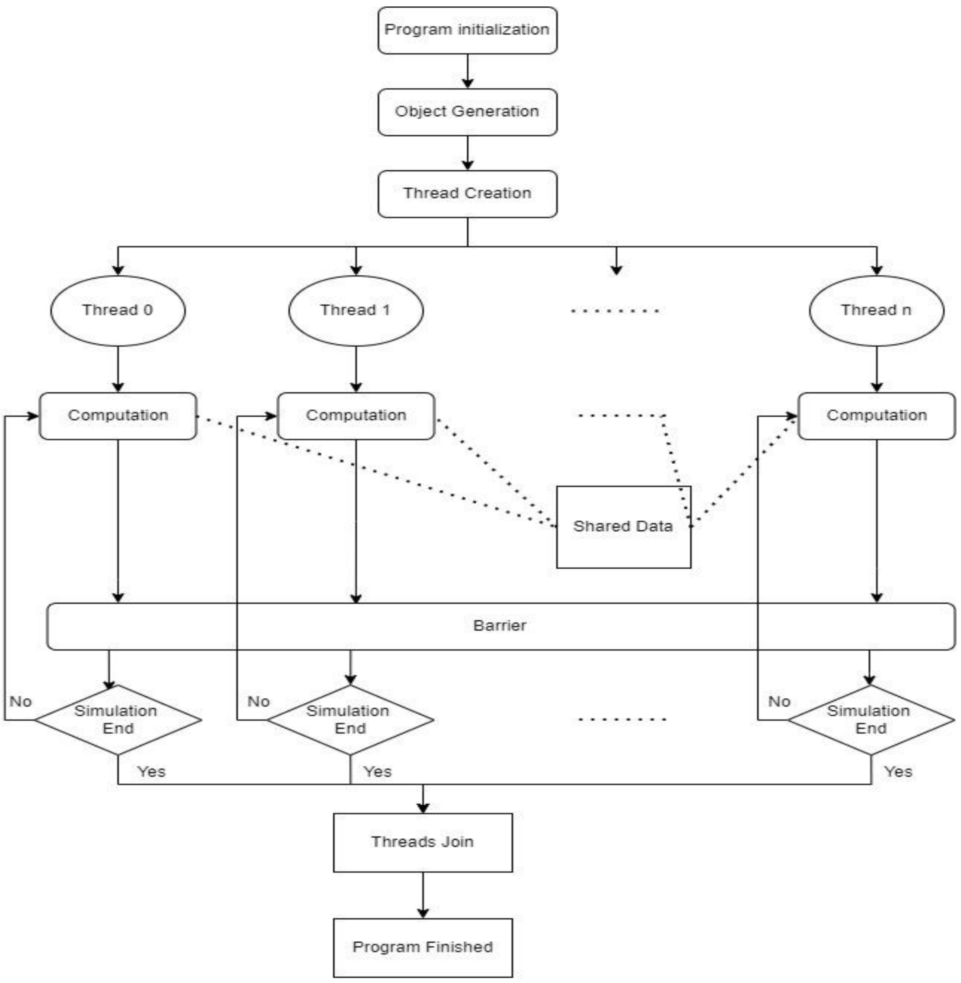
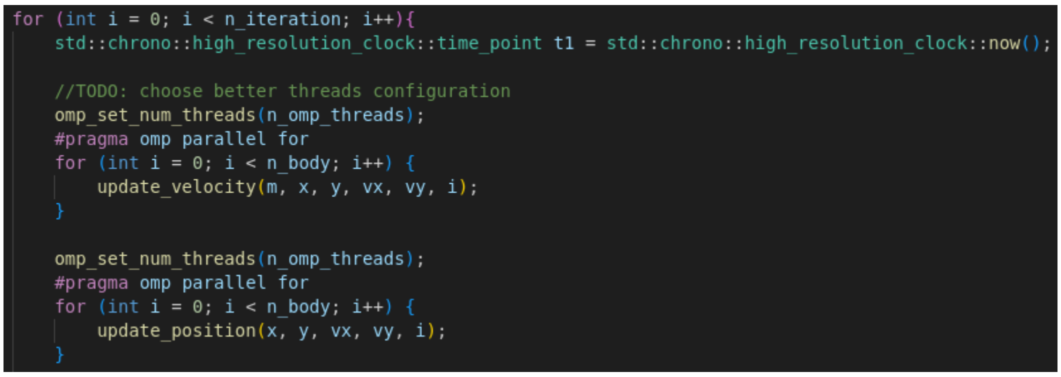


Figure 10. Pthread Flow Diagram Implementation

OpenMP Programming:

The implementation of OpenMP, a parallelization that shares the same architectural foundation as PThreads, is strikingly comparable to that of a sequential application. The reason is that OpenMP places a greater emphasis on parallelizing loops and some other portions that can be parallelized (using directive). Thus, OpenMP primarily executes the program in sequential fashion aside from those areas (or using only 1 thread). We create the outer loop after producing the objects and beginning the timer (simulation iteration). Because the computation inside this outer loop depends on the data computed in the previous iteration (data of the future position depend on data of the present position), we shouldn't parallelize this loop. The parallelization pragma is then placed inside the outer loop to parallelize the inner loop. The inner loop can be parallelized since it is used to compute numerous data (objects) that are independent of one another (no object's location is directly derived from the locations of other items). Since OpenMP performs these tasks automatically, we do not need to explicitly assign the specific duties for each thread when employing parallelization. A temporary variable will be used to store the outcomes of all computations, just like in sequential and PThreads implementations. So, we must update the object placements when each simulation iteration is complete. We don't need to employ a barrier with OpenMP, unlike PThreads, because the parallelization will terminate when the inner loop is completed, and the threads will then be connected. All threads must first arrive at the same position before they may be joined. As a result, we can think of this as a thread synchronization barrier. Then, in order to boost performance even more, we added parallelization to the process of updating object positions. Following the update, the program will keep running the simulation until it is finished (at which point the timer is also stopped).



```
for (int i = 0; i < n_iteration; i++){
    std::chrono::high_resolution_clock::time_point t1 = std::chrono::high_resolution_clock::now();

    //TODO: choose better threads configuration
    omp_set_num_threads(n_omp_threads);
    #pragma omp parallel for
    for (int i = 0; i < n_body; i++) {
        update_velocity(m, x, y, vx, vy, i);
    }

    omp_set_num_threads(n_omp_threads);
    #pragma omp parallel for
    for (int i = 0; i < n_body; i++) {
        update_position(x, y, vx, vy, i);
    }
```

Figure 11. OpenMP Algorithm differentiator

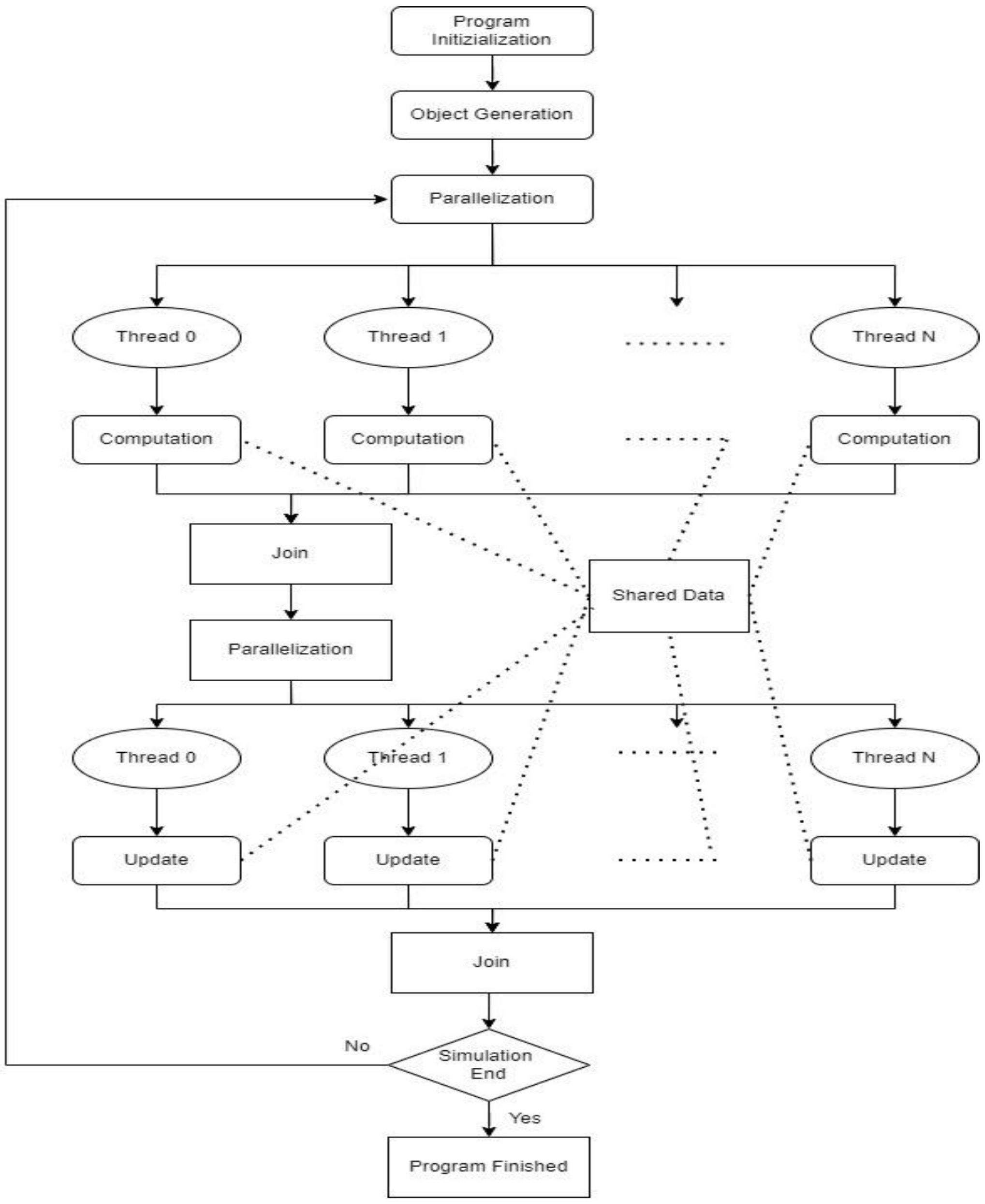


Figure 12. OpenMP Flow Diagram Implementation

CUDA Programming:

The final parallel implementation makes advantage of the CUDA programming model from Nvidia. Comparing CUDA to earlier parallel implementations reveal how distinctive and diverse it is. The Graphical Processing Units (GPU) with their massive core count are used by CUDA. These cores are slower than standard CPU cores and can only handle rudimentary tasks. CUDA supports running several threads at once. A CUDA thread block is a collection of these threads. The block size in our software is set to 512. By using the distinct Blockidx and Threadidx, we may index each individual thread separately. But only 32 threads can be active at once (called a warp). We are only able to pass kernel processes to the GPU for its special calculation. A unique feature of CUDA is memory.

For each thread, the GPUs use registers that are chosen by the compiler. Threads do not share register memory. However, CUDA also makes use of shared memory known as the L1 cache. Shareable among all CUDA blocks is this memory. The read-only memory, which includes an instruct cache and constant memory, comes next. It is not available to users and is read-only for kernel code. The L2 cache follows, which performs similar tasks to the L1 cache but is slower and has greater storage. Finally, there is the global memory, which consists of the GPU's internal DRAM and framebuffer size. The CUDA application will not have a fixed number of threads we can run like the previous paradigms because GPU programming and CUDA are unique. Instead, every participant in the simulation will be assigned a thread in this implementation due to the enormous number of threads it can handle. The same procedures will next be followed (sequentially) to update the velocities and locations of each body. For the simulation, CUDA will perform 32 body calculations every block.

Due to its usage of the most threads, CUDA will serve as the gold standard for parallel execution (most parallelized). Its general structure resembles Pthread's quite a bit.

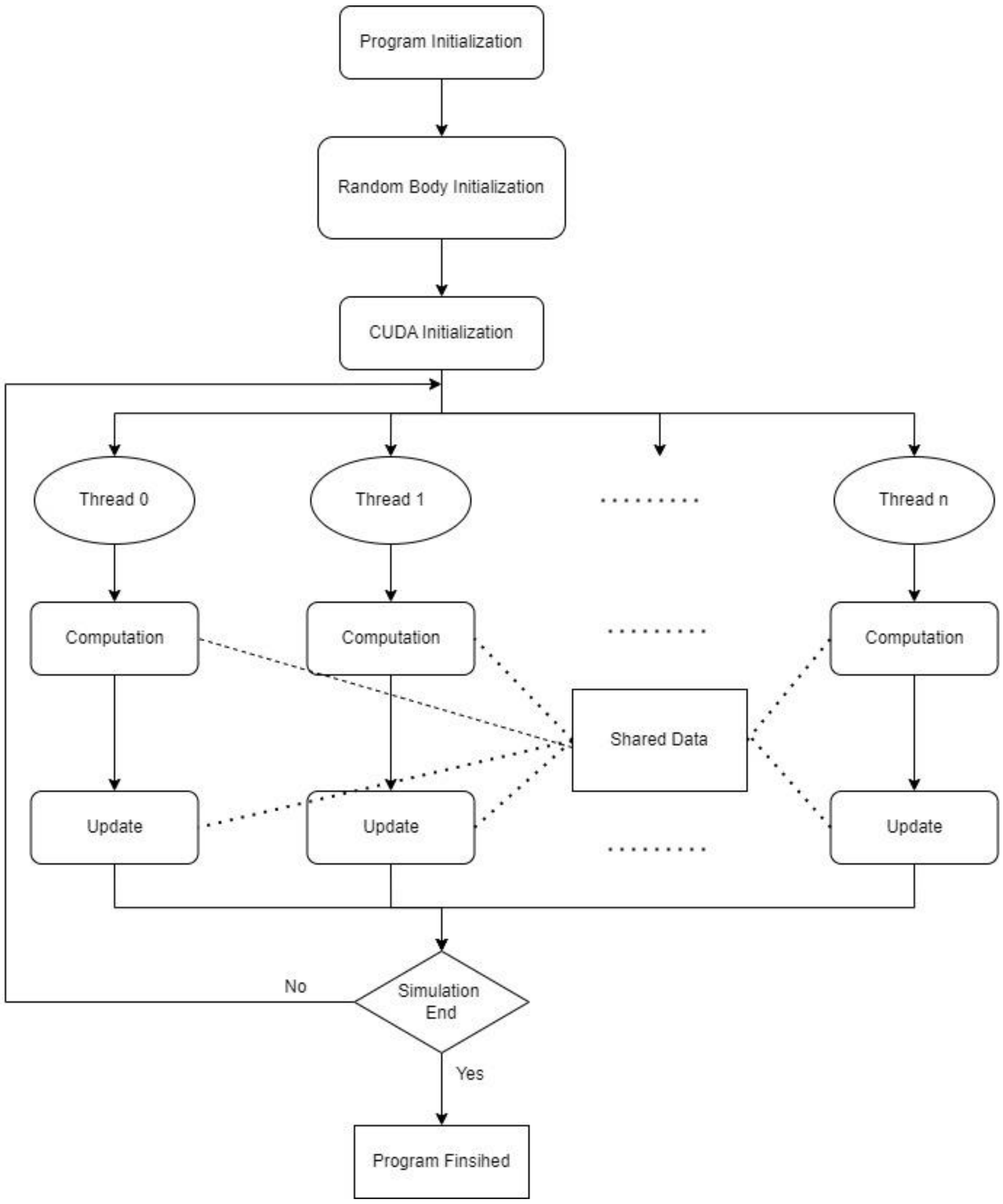


Figure 13. CUDA Flow Diagram Impelementation

Code Experiment and Data Visualization

Sequential Programming

Processor	100	500	1000	5000	10000
1	0.0227	0.2033	0.7734	19.3473	77.3888

MPI Programming – Have not got any result yet since the code did not work

PThread Programming

Threads	100	500	1000	5000	10000
3	0.0342	0.2004	0.7678	7.9751	27.3741
5	0.0605	0.1309	0.4741	5.4512	17.1232
7	0.0668	0.1063	0.3575	4.3768	12.7626
15	0.0766	0.1269	0.3310	3.9746	11.2547
30	0.1268	0.1182	0.3364	3.6994	9.4666

OpenMp Programming

Threads	100	500	1000	5000	10000
3	0.01021	0.39102	0.60301	11.915	44.9217
5	0.40312	1.03193	0.88078	8.94171	29.448
7	0.01944	0.83025	0.42401	7.14093	27.1695
15	0.02249	0.69302	0.40402	6.92442	25.5533
30	0.01672	0.58684	0.36189	5.08712	21.3011

Cuda Programming

N	100	500	1000	5000	10000
1	0.0732	0.4427	0.78916	3.1864	7.13873

We may infer from the data in the tables and figures that parallel implementations typically result in a reduction in execution time as the number of processors/threads increases for an increase in the number of particles. We also observe that, as anticipated based on our prior studies, the sequential program outperforms the parallel implementations in the 500-body data column. We can speculate that using the

Hypercube data model with 30 processors results in a significant increase in communication overhead because the data from the 30 processors must be repeatedly sent to every other processor during each iteration of the global arrays. There will be breakpoints, though, where the 30 processors will accelerate the execution time.

It's interesting to observe that after 5 threads for particle counts of 100 and 500, Pthread and OpenMP implementations don't much speed up. After 10 threads for particle counts of 1000 and 15 threads for particle counts of 10,000, the same behavior may be seen. There are instances in our experiment where they sense a slowdown rather than a speedup. The construction and joining overhead that was addressed in an earlier section may, however, be the cause of this in the case of Pthreads. Given that these two parallel threading paradigms struggle to accelerate after a certain number of threads for various input sizes, adding more threads will eventually cause the overhead to catch up to the speedup. As a result, our returns have been drastically reduced. We can surmise that either this is a characteristic of multithreaded programming architectures like Pthread and OpenMP, or that it is a result of the program's implementation (Pthread and OpenMP implementation is similar).

Finally, when we examine CUDA, we can observe that it performs worse than the sequential program for tiny input sizes. This is because the GPU employs a greater number of degenerate CPUs, which operate more slowly than the CPU's standard processing units. However, as we increase the particle count to 1000, we can see that CUDA is now beginning to outperform the sequential program. This can be seen as the breakpoint. This demonstrates the GPU's incredible capacity for parallel programming thanks to its enormous arrays of CPUs and threads. As a result, we can conclude that, in contrast to other parallel paradigms, which can speed up even with medium input sizes, the CUDA architecture should only be used with extremely large jobs (such as data mining and crypto mining).

Project Summary

This project helped us understand how different parallel computing models, notably MPI, Pthread, OpenMP, and CUDA, may speed up computations to different degrees. We can observe that MPI speeds up naturally up to a certain number of processors, after which the overhead outpaces the speedup from parallel execution. The two threaded paradigms Pthread and OpenMP struggle to accelerate after a certain number of threads, much like MPI does. However, the input size also affects this figure. We speculate that this is caused by factors with diminishing returns that are comparable to those in MPI. Multithreaded architectures might have this as a feature. Finally, we see that the CUDA speedup pattern is the simplest. We observe that it is slower than the sequential approach at small input sizes because of the slower GPU cores. Because there are so many GPU cores available, the speedup from CUDA is only noticeable at enormous input sizes. The speedup for the CUDA paradigm is likewise the greatest compared to the other implementations. Thus, we draw the conclusion that CUDA should only be used for very big activities, MPI for medium- to large-sized jobs, and threaded architectures for small to medium-sized jobs.