Contents

1 Pre-Implementation Considerations

Several aspects are considered in advance to facilitate the implementation process and to improve the overall programming quality.

1.1 Programming Practices

To minimise the risk of programming pitfalls, a set of good practices is adopted. These practices are assessed according to their relative importance to the quality. Below is the list of practices ordered in descending importance level.

1. **Robustness:** The program must be able to handle any user input without crashing. If a given input is invalid, the program should return a descriptive error message.
2. **Maintainability:** Follow a consistent coding style and use the self-documenting function and variable names. Refactor the code so that each function is only responsible for one task. Code with good structure and high cohesion will be able to narrow down the range of potential problems in the future.
3. **Computational Resource Efficient:** Perform the minimum amount of computation in a loop if possible. Minimise overhead by reducing the number of function calls in a loop. If the value of a computationally expensive operation will be used multiple times, then store it into a variable.
4. **Documentation:** The code should be well commented. Each function should have a Javadoc style block comment before the function definition. It is preferred to write self-documenting code, however, inline comments should be used when further clarification is required.

1.2 Unit Testing

Since the program involves user input processing, thus it is required to perform basic unit testing to test its correctness and completeness in handling user inputs.

It is impractical to test every possible combination of user inputs. Therefore, each command argument will be tested independently with their respective edge cases. Testing the program with edge cases is an efficient way to ensure that the problem can handle extreme user inputs.

Test Case	Argument	Expected Result	Status

Table 1: Documentation format of unit testing.

1.2.1 Limitations

Unit testing with edge cases will not catch every error in the program, such as runtime errors or system-level errors. It is possible for the program to produce incorrect behaviour while passing all the test cases. To mitigate this issue, the number of benchmarks has to be increased to reduce the likelihood of unexpected errors.

2 Preliminary Tasks Implementation

This section discusses and compares the various possible techniques for implementing the preliminary tasks before the N-body simulation.

2.1 Arguments Processing

2.1.1 Integer Arguments

Problem: The integer arguments (**N**, **D**, and **I**) are saved as 'string' in `char *argv[]`.

Goal: Parse and convert the 'string' into `unsigned int` (the same data type used in `NBodyVisualiser.c`).

Each input argument **N**, **D**, and **I** can be categorised as follows:

1. **Category 1:** Argument is a string (e.g. "abc"). In this case, the program must exit with an error message since the argument is not a valid integer and cannot be converted.
2. **Category 2:** Argument is a negative number. Since the data type will be `unsigned int`, the program must exit with an error message.
3. **Category 3:** Argument is zero. For **N** and **D**, the value must be larger than 1 so the program will exit with an error message. For **I**, the program will accept the argument but start in **visualisation** mode.
4. **Category 4:** Argument is a positive number. In this category, there are two more cases to be considered:
   (a) argument $\leq$ `INT_MAX`. The program should accept the argument and produce no error. If the number is very large and the machine has limited computational resources, then the program will either return a memory allocation error (discussed in **Section 2.3**) before the simulation or encounter errors including out-of-memory error, system not responding error, etc. during the simulation.
   (b) argument $>$ `INT_MAX`. The is the case where overflow occurs. The program should return an error and exit.

N.B. Changes are made to the definition of **Category 4** in the second iteration. Refer to **Section 2.5.1** for detailed explanation.

Based on the analysis of input categorisation, the problem is now clearly defined and the implementation approach can be chosen. The C standard library `stdlib.h` provides different functions that can convert a string to different integer types. Firstly, the implementation has to choose a function with appropriate return type. However, there are several return types that are not suitable for the use case, as justified below:

1. A function with return type of `int` is not able to satisfy the conditions in **Category 4**. Argument inputs that are larger than `INT_MAX` will cause overflow of `int`.

2. According to the official Microsoft documentation on storage of basic types [5], `long` has the same storage size as `int` (4 bytes). Therefore, it will have the same issue as (1).
3. Function that returns `float`, `double`, or `long double` are not suitable as they are primarily use for floating point numbers.

After applying these restrictions, the available functions for selection are:

- `atoll`: returns `long long int`
- `strtoll`: returns `long long int`
- `strtoul`: returns `unsigned long int`
- `strtoull`: returns `unsigned long long int`

Conclusion

`strtoul` is chosen to be the function for integer arguments processing. This is because:

1. Its return type is `unsigned long int`, which has the same size and range as `unsigned int` [5]. In comparison with a `long long` type, it would use less computational resources and this comply with **Programming Practice 3**.
2. It stops reading the string at the first character it cannot recognize as part of a number [6]. This is helpful for error checking the string.

2.1.2 Operation Mode (M)

There are only two valid enum values for the operation mode: `CPU` and `OPENMP`. The program uses `strcmp()` to compare the argument **M** with the strings `"CPU"` and `"OPENMP"`, and assign the operation mode to the corresponding enum value.

The program will return an error message and exit if the argument **M** is other value.

2.1.3 Input File

The file name is stored into `char *` variable and the file content is processed in N-body data initialisation stage later.

The program will return an error message if the option `-f` is specified but no file name is provided.

2.2 N-Body Data Initialisation

```
#define DEFAULT_X ((float)rand() / RAND_MAX)
#define DEFAULT_Y ((float)rand() / RAND_MAX)
#define DEFAULT_VX 0
#define DEFAULT_VY 0
#define DEFAULT_M (1 / (float)N)
```

Listing 1: Default value of each n-body member variable

Macros are defined for the default value of each n-body member variable. They will be used for generating random data and setting the unspecified variables in an input file.

2.2.1 Generating Random Data

Generating random data is straightforward. The value of each member variable is set to the corresponding macro.

2.2.2 Processing Input File

Problem: The input file contains lines of comma separated values, each representing the initial states of an n-body. The input file may also contain comment lines.

Goal: Read the input file and process only non-comment lines. Default values should be assigned for unspecified variables.

Step 1: Reading the lines

Each line in the input file can be categorised as follows:

1. **Category 1:** The line is a comment line
2. **Category 2:** The line is an empty line
3. **Category 3:** The line do not have the correct number of commas
4. **Category 4:** The line contains invalid values (e.g. a string)
5. **Category 5:** The line contains 5 values and the correct number of commas
6. **Category 6:** The line contains unspecified values and the correct number of commas
7. **Category 7:** The line is longer than the defined `BUFFER_SIZE` of value 64

Lines that belong to either **Category 5** or **Category 6** should only be read and processed by the program. This is implemented by modifying the `readLine()` function provided in the Lab 1 Exercise 5 solution, as shown in **Listing 2**.

```
1       while ((c = (char)getc(f)) != EOF) {
2           // Case 1: ignore any line starting with '#'
3           // Case 2: ignore any blank line
4           if (i == 0 && (c == '#' || c == '\n')) {
5               while (c != '\n') {
6                   c = (char)getc(f);
7               }
8           } else {
9               ...
```

Listing 2: Code snippet of `read_line()` function.

For **Category 3 and 4**, a helper function is created to ensure that only 4 commas appear in a non-comment line and all values in the line can be converted into float. The function will exit the program with an error message if the line does not contain the correct number of commas.

Step 2: Extracting the Values

Now each line will have exactly 4 commas. Some lines will have all 5 values and some of them will have unspecified values. Unspecifed values will then be assigned with the default values defined in **Listing 1**. Firstly, the line has to be split into a series of tokens using comma as delimiter. Each token can then be converted into `float` using the functions provided in C standard library.

Iteration 1

The first attempt to split the line into tokens was using the function `strtok` in C `string.h` library. It is works well when the line is in correct format, and there must be a space after each comma. However, in the case where the line has consecutive commas without space between them (e.g. `0.1f,,0.2f,,`), `strtok` would extract the second token as `0.3f` instead of `NULL`.

After further experiments, it is confirmed that `strtok` is unable to recognise empty tokens. Therefore, a different approach has to be adapted.

Iteration 2

```
1  static char *tokenise(char *buffer) {
2      static char *buffer_start = NULL;
3
4      if (buffer != NULL) buffer_start = buffer;
5
6      // see if we have reached the end of the line
7      if (buffer_start == NULL || *buffer_start == '\0') return NULL;
8
9      // return the number of characters that are not delimiters
10     const unsigned int n = strcspn(buffer_start, ",");
11
12     // return token as NULL for consecutive delimiters
13     if (n == 0) {
14         buffer_start += 1;
15         return NULL;
16     }
17
18     // save start of this token
19     char *p = buffer_start;
20
21     // bump past the delimiters
22     buffer_start += n;
23
24     // remove the delimiters
25     if (*buffer_start != '\0') *buffer_start++ = '\0';
26
27     return p;
28 }
```

Listing 3: Code snippet of `tokenise()` function.

The new approach attempts to address the problem in **Iteration 1** by creating a customised version of `strtok` function (**Listing 3**). Line 13 - 16 allow the `tokenise` to return NULL as the token between two consecutive delimiters.

2.3 Memory Allocation

As mentioned in **Section 2.1.1**, memory error might occur if the integer arguments **N** and **D** are large and the machine does not have sufficient memory for dynamic allocation. This can be avoided by having a `NULL` check on whether the memory allocation has succeeded, as shown in **Listing 4**.

```
1       nbodies = (nbody *)malloc(sizeof(nbody) * N);
2       if (nbodies == NULL) {
3           fprintf(stderr, "error: failed to allocate memory: nbodies\n");
4           exit(EXIT_FAILURE);
5       }
6
7       activity_map = (float *)malloc(sizeof(float) * D * D);
8       if (activity_map == NULL) {
9           fprintf(stderr, "error: failed to allocate memory: activity_map");
10          exit(EXIT_FAILURE);
11      }
```

Listing 4: Dynamically allocating memory.

2.4 Testing

2.4.1 Command-Line Arguments Parsing

Test Case (file)	Expected Result	Status
Missing required arguments	Program exits and prints help message	Pass
Passing undefined option flag	Program exits with an error message	Pass
Passing duplicate option flag	Only the latest value is stored	Pass

Table 2: Test cases for command-line arguments parsing.

2.4.2 Number of Bodies (N)

Test Case	Argument	Expected Result	Status
Not an integer	$\mathbf{N} = 7.5$, abc, ...	Program exits with an error message	Pass
Negative number	$\mathbf{N} = -123$	Program exits with an error message	Pass
Zero	$\mathbf{N} = 0$	Program exits with an error message	Pass
Positive number	$\mathbf{N} = 123$	Program starts the simulation	Pass
Exceeds `INT_MAX`	$\mathbf{N} >$ `INT_MAX`	Program exits with an error message	Pass

Table 3: Test cases for the argument **N**.

2.4.3 Activity Grid Dimension (D)

N.B. See **Section 2.5.2** for the justification about the test case $\mathbf{D} > 46430$.

Test Case	Argument	Expected Result	Status
Not an integer	**D** = 7.5, abc, ...	Program exits with an error message	Pass
Negative number	**D** = −123	Program exits with an error message	Pass
Zero	**D** = 0	Program exits with an error message	Pass
Positive number	**D** = 123	Program starts the simulation	Pass
Exceeds 46430	**D** > 46430	Program exits with an error message	Pass

Table 4: Test cases for the argument **D**.

2.4.4 Operation Mode (M)

Test Case	Argument	Expected Result	Status
CPU mode	**M** = CPU	Simulation starts starts in CPU mode	Pass
OPENMP mode	**M** = OPENMP	Simulation starts in OPENMP mode	Pass
Other inputs	**M** = 123, abc, ...	Program exits with an error message	Pass

Table 5: Test cases for the argument **M**.

2.4.5 Number of Simulation Iterations (-i I)

Test Case	Argument	Expected Result	Status
Not an integer	**I** = 7.5, abc, ...	Program exits with an error message	Pass
Negative number	**I** = −123	Program exits with an error message	Pass
Zero	**I** = 0	Simulation starts in visualisation mode	Pass
Positive number	**I** = 123	Simulation starts in iteration mode	Pass
Exceeds `INT_MAX`	**I** > `INT_MAX`	Program exits with an error message	Pass

Table 6: Test cases for the argument **I**.

2.4.6 Input File (-f F)

Test Case (file)	Expected Result	Status
File name exists in system	Program starts to parse the file	Pass
File name does not exist in system	Program exits with an error message	Pass

Table 7: Test cases for the argument **F**.

Test Case (line in file)	Expected Result	Status
Blank line	Line is ignored by program	Pass

Comment line	Line is ignored by program	Pass
Non-comment line, incorrect format	Program exits with an error message	Pass
Non-comment line, invalid value	Program exits with an error message	Pass
Non-comment line, correct format	Values are extracted and stored	Pass
Line length > BUFFER_SIZE	Program exits with an error message	Pass

Table 8: Test cases for the lines in the input file.

2.5 Changes Introduced in Second Iteration

2.5.1 Maximum Value of Number of Bodies (N) and Iterations (I)

```
1  static void step(void) {
2      unsigned int i;
3      ...
4  #pragma omp parallel for if (M == OPENMP)
5      for (i = 0; i < N; i++) {
6          ...
7  // fixed by changing into
8      int i;
9  #pragma omp parallel for if (M == OPENMP)
10     for (i = 0; i < (int)N; i++) { ...
```

Listing 5: Compile error C3016 [2] for OpenMP `for` statement.

A compile error C3016 was encountered (line 2 - 5 of **Listing 5**) while attempting to parallelise the code. This is because the index variable in OpenMP `for` statement must have signed integral type [2]. This is fixed by changing the type of variable `i` and to `int`. However, the variable `N` is defined as `unsigned int` type. After fixing the compile error, the `i < N` comparison is unsafe as `N` can hold values larger than `INT_MAX`.

N.B. Therefore, to prevent erroneous behaviour in type casting such as `(int)N` or `(int)I`, the maximum possible value for **N** and **I** are changed from `UINT_MAX` to `INT_MAX`. (changes are applied to **Category 4** in **Section 2.1.1**)

2.5.2 Maximum Value of Activity Grid Dimension (D)

```
1      const float normalise = (float)D / (float)N;
2      for (i = 0; i < grid_size; i++) {
3          activity_map[i] *= normalise;
4      }
```

Listing 6: Possible overflow for casting `D * D` into `int`.

In line 2 of **Listing 6**, arithmetic overflow will occur when `D` is larger than 46341 as $46341^2 > 2^{31} - 1$. This is fixed by setting the maximum possible value of `D` to 46340.

3 Serial Implementation

This section describes the implementation process of the N-body simulation and discuss various improvements made in the process.

3.1 `step()` Function

3.1.1 Calculating the Overall Force on a Body (F_i)

```
1     float sum_x = 0, sum_y = 0;
2
3     for (unsigned int j = 0; j < N; j++) {
4         const float dist_x = nbodies[j].x - nbodies[i].x;
5         const float dist_y = nbodies[j].y - nbodies[i].y;
6         const float mag = dist_x * dist_x + dist_y * dist_y;
7         const float m_div_soft = nbodies[j].m / powf(mag + SOFTENING_SQUARE, 1.5f);
8
9         sum_x += m_div_soft * dist_x;
10        sum_y += m_div_soft * dist_y;
11    }
```

Listing 7: Calculation of F_i within the `step()` function.

$$\vec{F_i} = Gm_i \sum_{j=1}^{N} \frac{m_j(\vec{x_j} - \vec{x_i})}{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^{\frac{3}{2}}} \tag{1}$$

Listing 7 is the implementation for the summation part in **Equation (1)**. The calculation of F_i is integrated into **Listing 8**, see **Section 3.1.2** for justification.

At line 6, it is not required to apply square root to the variable `mag` (magnitude) as it will be squared in $||\vec{x_j} - \vec{x_i}||^2$, thus reducing the computational cost inside the loop.

In **Equation (1)**, the calculation for $\frac{m_j}{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^{\frac{3}{2}}}$ will be repeated for `sum_x`, and `sum_y`. Therefore, at line 7, the arithmetic operation is stored into a `const` variable to reduce additional computational cost in the loop.

3.1.2 Calculating the Movement

```
1         // Calculate position vector, do this first as it depends on current velocity
2         nbodies[i].x += dt * nbodies[i].vx;
3         nbodies[i].y += dt * nbodies[i].vy;
4
5         // Calculate velocity vector, force and acceleration are computed together
6         nbodies[i].vx += dt * G * sum_x;
7         nbodies[i].vy += dt * G * sum_y;
```

Listing 8: Calculation of $v_{t+1}^{\vec{}}$ and $x_{t+1}^{\vec{}}$ within the `step()` function.

$$\vec{a_i} = \frac{\vec{F_i}}{m_i} \tag{2}$$

$$\vec{v_{t+1}} = \vec{v_t} + dt * \vec{a} \tag{3}$$

Substituting **Equation (1)** into **Equation (2)**,

$$\begin{aligned}\vec{a_i} &= \frac{Gm_i \sum_{j=1}^{N} \frac{m_j(\vec{x_j}-\vec{x_i})}{(||\vec{x_j}-\vec{x_i}||^2+\epsilon^2)^{\frac{3}{2}}}}{m_i} \\ &= G\sum_{j=1}^{N} \frac{m_j(\vec{x_j}-\vec{x_i})}{(||\vec{x_j}-\vec{x_i}||^2+\epsilon^2)^{\frac{3}{2}}}\end{aligned}$$

Substituting the result into **Equation (3)**,

$$\vec{v_{t+1}} = \vec{v_t} + dt * G\sum_{j=1}^{N} \frac{m_j(\vec{x_j}-\vec{x_i})}{(||\vec{x_j}-\vec{x_i}||^2+\epsilon^2)^{\frac{3}{2}}}$$

The derivation above has concluded that the complete calculation of $\vec{a_i}$ and $\vec{F_i}$ is unnecessary. As demonstrated at line 6-7 in **Listing 8**, only `sum_x` and `sum_y` are required for the calculation of velocity vector ($\vec{v_i}$). This has resulted in a smaller code size and lower computational cost inside the outer N-bodies loop.

3.1.3 Calculating the Activity Map

```
1  // Clear the previous step values (this is before the outer loop)
2  memset(activity_map, 0, grid_size * sizeof(float));
3  ...
4        const unsigned int col = (unsigned int)(nbodies[i].x * (float)D);
5        const unsigned int row = (unsigned int)(nbodies[i].y * (float)D);
6        const unsigned int cell = (unsigned int)(D * row + col);
7
8        // Do not update `activity_map` if n-body is out of grid area
9        if (cell >= 0 && cell < grid_size) {
10           ++activity_map[cell];
11       }
```

Listing 9: Calculation of activity map within the `step()` function.

The variable `activity_map` holds a pointer to a range allocated memory returned by `malloc`. As `activity_map` is a trying to represent a 2D array, the index of a 2D array cell in 1D array can be calculated by `D * row + col` (line 6).

Each N-body is inside the grid if and only if its `x` and `y` are between 0 and 1. The conditional statement at line 9 is use to ensure that the activity map is not updated when an N-body is not within the grid, otherwise erroneous behaviour would occur as line 10 would be writing beyond the bounds of allocated memory.

At line 2, each allocated block of memory pointed by `activity_map` is set to zero at the start of each step. This is to ensure that the counts of each cell in the previous step do not carry over to the current step, otherwise each cell will not be counting the actual number of N-body in the cell.

3.1.4 Normalising the Activity Map

```
1     const float normalise = (float)D / (float)N;
2     for (i = 0; i < grid_size; i++) {
3         activity_map[i] *= normalise;
4     }
```

Listing 10: Normalisation of activity map within the `step()` function.

At the end of the `step()` function, each count in `activity_map` is then normalised, as shown in **Listing 10**. N.B. Since `(float)D / (float)N` is a constant value, it is stored in a variable and reused in the loop (line 1) to save additional CPU cycles.

3.2 Performance Profiling

This section aims to identify potential optimisation opportunities for the serial implementation through the Performance Profiler. The profiling statistics will also be used as a reference for parallel implementation in **Section 4**.

3.2.1 CPU Hot Path

The program is run with arguments: $\mathbf{N} = 20000$, $\mathbf{D} = 10$, $\mathbf{M} = \text{CPU}$, $\mathbf{I} = 10$.

```
                    53          for (unsigned int i = 0; i < I; i++) {
791359 (99.89%)     54              step();
                    55          }
```

Figure 1: Hot path of `main()` function

```
                    87          for (unsigned int j = 0; j < N; j++) {
                    88              const float dist_x = nbodies[j].x - nbodies[i].x;
  4621 (0.67%)      89              const float dist_y = nbodies[j].y - nbodies[i].y;
  5333 (0.78%)      90              const float mag = dist_x * dist_x + dist_y * dist_y;
 31225 (4.56%)      91              const float m_div_soft = nbodies[j].m / powf(mag + SOFTENING_SQUARE, 1.5f);
                    92
 50083 (7.31%)      93              sum_x += m_div_soft * dist_x;
 45828 (6.69%)      94              sum_y += m_div_soft * dist_y;
                    95          }
                    96
                    97          /* Movement */
                    98          // Calculate position vector, do this first as it depends on current velocity
     3 (0.00%)      99          nbodies[i].x += dt * nbodies[i].vx;
     7 (0.00%)     100          nbodies[i].y += dt * nbodies[i].vy;
                   101
                   102          // Calculate velocity vector, force and acceleration are computed together
                   103          nbodies[i].vx += dt * G * sum_x;
     3 (0.00%)     104          nbodies[i].vy += dt * G * sum_y;
                   105
                   106          /* compute the position for a body in the `activity_map`
                   107           * and increase the corresponding body count */
                   108          const unsigned int col = (unsigned int)(nbodies[i].x * (float)D);
     1 (0.00%)     109          const unsigned int row = (unsigned int)(nbodies[i].y * (float)D);
     3 (0.00%)     110          const unsigned int cell = (unsigned int)(D * row + col);
                   111
                   112          // Do not update `activity_map` if n-body is out of grid area
     3 (0.00%)     113          if (cell >= 0 && cell < grid_size) {
    28 (0.00%)     114              ++activity_map[cell];
                   115          }
```

Figure 2: Hot path of `step()` function

3.2.2 Possible Optimisations for Serial Implementation

At line 91 of **Figure 2**, it might be possible to reduce the computation cost of `m_div_soft` by simplifying the equation. `m_div_soft` represents the following part in **Equation (1)**,

$$\frac{m_j}{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^{\frac{3}{2}}}$$

The denominator can be further simplified as follow,

$$\begin{aligned}(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^{\frac{3}{2}} &= \sqrt{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^3} \\ &= \sqrt{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^2 \times (||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)} \\ &= (||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)\sqrt{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)}\end{aligned}$$

Applying the modification to the code,

```
87      for (unsigned int j = 0; j < N; j++) {
88          ...
89          const float mag_add_soft = dist_x * dist_x + dist_y * dist_y + SOFTENING_SQUARE;
90          const float m_div_soft = nbodies[j].m / (mag_add_soft * sqrtf(mag_add_soft));
91          ...
92      }
```

Listing 11: Attempt to optimise the `step()` function.

Number of Bodies (N)	Total Execution Time (seconds)		Speedup
	Before	After	
Number of Bodies (N)			
256	0.268	0.037	7.24
512	1.066	0.144	7.40
1024	4.264	0.569	7.49
2048	17.463	2.300	7.59
4096	68.358	9.353	7.31
		Average	**7.406**
Activity Grid Dimension (D)			
100	4.230	0.572	7.40
1000	4.248	0.599	7.09
2000	4.402	0.736	5.98
5000	5.509	1.651	3.34
10000	8.477	5.020	1.69
		Average	**5.1**

Table 9: Before and after the optimisation attempt of **Listing 11** in iteration mode.

According to the results in **Table 9**, the optimisation attempt in **Listing 11** has improved the performance **significantly**. The optimisation has achieved an average speedup of **7.406**. Moreover, the optimisation also improves the performance in visualisation mode, as shown in **Figure 3**. Therfore, it is confirmed that the performance bottleneck was the `powf()` function.

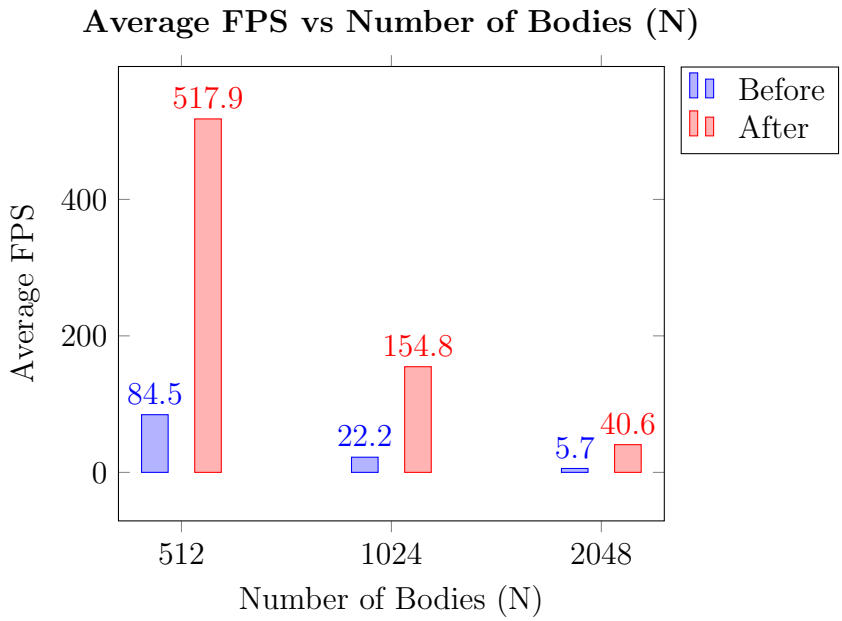

Figure 3: Average FPS before and after the optimisation attempt of **Listing 11** in visualisation mode (FPS recorded using CapFrameX [3]).

3.2.3 Possible Optimisations for Parallel Implementation

1. For the `main()` function, the only possible optimisation is to parallelise the loop.
2. For the `step()` function, the best candidate for optimisation is the summation calculation in the overall force (F_i) equation (line 89-90 in **Figure 2**). There are three possible optimisation approaches for `step()` function:
   (a) Parallelise the outer loop (N-bodies loop)
   (b) Parallelise the inner loop (Overall force F_i calculation loop)
   (c) Parallelise both loops by nested parallelism

3.3 Benchmark Configuration

To minimise the uncontrollable variables in the benchmark environment, all benchmarks are required to be performed under the same configuration to improve consistency.

3.3.1 Hardware Specifications

1. Operating system: Windows 10 64-bit
2. CPU: Intel® Core™ i7-7700HQ, 4 cores, 8 threads
3. RAM: 16GB DDR4-2400MHz

3.3.2 Number of Simulation Iterations

Question: Does the value of argument **I** affect the simulation performance?

Since **I** controls the number of simulations to run, then it is expected that the total execution time would increase as **I** increases. A quick benchmark is conducted to prove this hypothesis:

Fixed arguments: **N** = 1000, **D** = 10

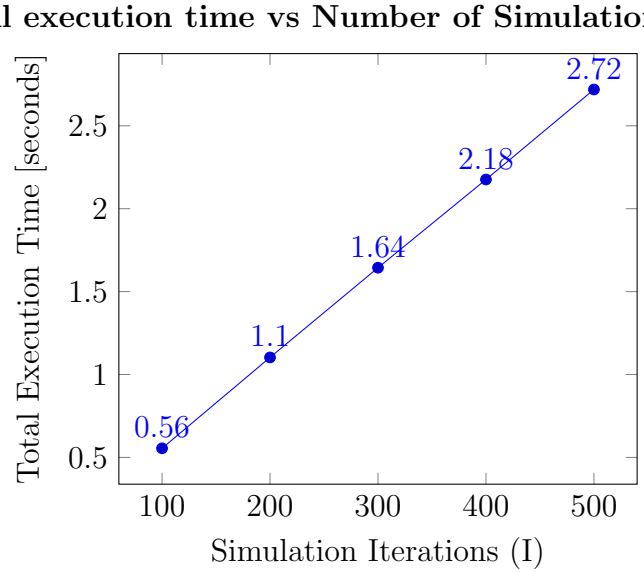

Figure 4: Relationship between the total execution time and number of simulation iterations.

According to **Figure 4**, the total execution time is **directly proportional** to the number of simulation iterations. The value of **I** should only affect the execution time proportionally, but not the simulation performance. Hence, it is decided to exclude **I** from the benchmark variable.

3.3.3 Number of Runs per Benchmark

It is unlikely that the result of every same benchmark run is the same. There will be minimal execution time difference between each run due to uncontrollable environment variables. This can be mitigated by taking the average execution time of multiple benchmark runs to minimise the error rate and randomness. Each benchmark result would be more accurate as well when compared to a single benchmark run. **Conclusion:** It is decided to calculate the average execution time of a single benchmark over 10 runs.

3.4 Benchmark Results

In this section, the benchmark results of serial implementation will be presented and commented. These results act as a baseline to measure potential performance improvement in parallel version.

3.4.1 Baseline Performance

The benchmark is done over a various values of **N** and **D** with the compute bound optimisations applied.

N.B. The benchmarks are performed after the optimisation in **Section 3.2.2**.

Benchmark on Number of Bodies (N)

Fixed arguments: **D** = 10, **I** = 100

N	Average Execution Time of 10 runs (seconds)
256	0.037
512	0.144
1024	0.569
2048	2.300
4096	9.353

Table 10: Baseline performance of program over various values of **N**.

Benchmark on Activity Grid Dimension (D)

Fixed arguments: **N** = 1024, **I** = 100

D	Average Execution Time of 10 runs (seconds)
100	0.572
1000	0.599
2000	0.736
5000	1.651
10000	5.020

Table 11: Baseline performance of program over various values of **D**.

4 Parallel Implementation

N.B. All benchmarks follows the format and fixed arguments used in the baseline performance benchmark (**Section 3.4.1**), where

- **Fixed arguments:** **D** = 10, **I** = 100 for benchmarks on **N**
- **Fixed arguments:** **N** = 1024, **I** = 100 for benchmarks on **D**

Section 4.1 to 4.4 aims to assess the effectiveness of each parallelisation strategy by parallelising one loop at a time in **iteration mode**. Scheduling types and chunk sizes are also included in the benchmarks for each parallelisation strategy.

4.1 `main()`: Simulation Iterations Loop

```
1  int main(const int argc, char *argv[]) {
2      ...
3      // This do not ensure correctness, justification below
4  #pragma omp parallel for default(none) shared(N, D)
5      for (i = 0; i < (int)I; i++) {
6          step();
7      }
8      ...
9  }
```

Listing 12: Parallelising the simulation iterations loop in `main()` function.

Parallelising the simulation iterations is not a good approach. This is because:

1. Each thread must have its own instance of `nbodies` and `activity_map` to prevent race conditions from occuring within the `step()` function.
2. Each step is dependent on the previous step (see **Section 3.1.2**). This implies that an `ordered` clause needs to added to the `parallel for` construct to execute the `step()` function call in serial, which is equivalent to executing the loop in serial.
3. The simulation iterations loop will only be executed in iteration mode. Therefore, it will not have any impact on the performance of visualisation mode.

4.2 `step()`: Outer N-Bodies Loop

```
1  static void step(void) {
2  #pragma omp parallel for default(none) shared(N, D, nbodies, activity_map) if (M == OPENMP)
3      for (i = 0; i < (int)N; i++) {
4        ...
5        if (cell >= 0 && cell < grid_size) {
6            // Race condition for `activity_map`
7            ++activity_map[cell];
8        }
9      }
10 }
```

Listing 13: Parallelising the outer N-bodies loop in `step()` function.

4.2.1 Race Condition Synchronisation

Value	Baseline (seconds)	Synchronisation (seconds)	
		critical	atomic
Number of Bodies (N)			
256	0.037	0.013	0.012
512	0.144	0.042	0.044
1024	0.569	0.152	0.152
2048	2.300	0.628	0.594
4096	9.353	2.457	2.303
Activity Grid Dimension (D)			
100	0.572	0.157	0.155
1000	0.599	0.182	0.178
2000	0.736	0.345	0.329
5000	1.651	1.388	1.321
10000	5.020	5.005	4.984

Table 12: Benchmark of synchronisation techniques for `activity_map` race condition.

The results in **Table 12** have shown that both `omp critical` and `omp atomic` perform better than the baseline performance, and `omp atomic` performs faster than `omp critical`. N.B. Hence, `omp atomic` is a better approach for avoiding the race condition.

Question: Can `omp master` and a local activity map prevent the race condition?

No. To prevent the race condition using `omp master` and a local activity map, the local activity map must be a 2D array. However, the dimension **D** is only known at runtime and it is not a constant. The compiler will generate compile error C2057 [1] for declaring the 2D array with unknown size at compile time, such as `float local_activity_map[D * D][N]`.

4.2.2 Static Scheduling

Value	Baseline (seconds)	Chunk Size (seconds)				
		default	1	2	4	8
Number of Bodies (N)						
256	0.037	0.011	0.013	0.012	0.014	0.014
512	0.144	0.040	0.041	0.043	0.071	0.046
1024	0.569	0.158	0.160	0.162	0.182	0.160
2048	2.300	0.611	0.622	0.607	0.607	0.608
4096	9.353	2.343	2.494	2.470	2.441	2.383
Activity Grid Dimension (D)						
100	0.572	0.156	0.186	0.167	0.158	0.159
1000	0.599	0.176	0.182	0.184	0.196	0.199
2000	0.736	0.343	0.378	0.377	0.337	0.335
5000	1.651	1.377	1.420	1.396	1.427	1.387
10000	5.020	4.972	5.017	4.933	4.934	4.942

Table 13: Benchmark of parallelising outer N-bodies loop with static scheduling.

The results in **Table 13** have shown that the default chunk size (determined by OpenMP) has the best overall performance improvements. This indicates that the workloads are equally shared between each thread.

4.2.3 Dynamic Scheduling

Value	Baseline (seconds)	Chunk Size (seconds)			
		1 (default)	2	4	8
Number of Bodies (N)					
256	0.037	0.015	0.013	0.012	0.013
512	0.144	0.045	0.042	0.045	0.042
1024	0.569	0.156	0.159	0.156	0.160
2048	2.300	0.592	0.590	0.588	0.594
4096	9.353	2.309	2.329	2.314	2.330
Activity Grid Dimension (D)					
100	0.572	0.157	0.173	0.158	0.159
1000	0.599	0.188	0.190	0.181	0.180
2000	0.736	0.346	0.409	0.339	0.338
5000	1.651	1.390	1.368	1.391	1.383
10000	5.020	4.991	4.959	4.909	4.936

Table 14: Benchmark of parallelising outer N-bodies loop with dynamic scheduling.

The best chunk size could not be deduced based on the results in **Table 14**. Dynamic scheduling produces inconsistent results, and thus is not suitable for this parallelisation strategy.

4.2.4 Performance Improvements

Based on the benchmark results and analysis in **Sections 4.2.2 and 4.2.3**, static scheduling with default chunk size is selected as the scheduling type for this parallelisation strategy.

Average performance speedup for **Number of Bodies (N)** with `schedule(static)`:

$$\frac{\frac{0.037}{0.011} + \frac{0.144}{0.040} + \frac{0.569}{0.158} + \frac{2.300}{0.611} + \frac{9.353}{2.343}}{5} = 3.45$$

Average performance speedup for **Activity Grid Dimension (D)** `schedule(static)`:

$$\frac{\frac{0.572}{0.156} + \frac{0.599}{0.176} + \frac{0.736}{0.343} + \frac{1.651}{1.377} + \frac{5.020}{4.972}}{5} = 2.28$$

4.3 `step()`: Inner Overall Force (F_i) Calculation Loop

```
1  static void step(void) {
2      ...
3      for (i = 0; i < (int)N; i++) {
4          float sum_x = 0, sum_y = 0;
5
6  #pragma omp parallel for default(none) shared(N) if (M == OPENMP)
7          for (j = 0; j < (int)N; j++) {
8              ...
9              // Race condition below
10             sum_x += m_div_soft * dist_x;
11             sum_y += m_div_soft * dist_y;
12         }
13     }
14     ...
15 }
```

Listing 14: Parallelising the inner loop in `step()` function.

4.3.1 Race Condition Synchronisation

Number of Bodies (N)	Baseline (seconds)	Synchronisation (seconds)		
		critical	atomic	reduction
256	0.037	0.822	0.722	0.049
512	0.144	3.275	3.271	0.114
1024	0.569	12.892	14.319	0.297
2048	2.300	51.49	54.137	0.879
4096	9.353	278.471	229.10	2.911

Table 15: Benchmark of synchronisation techniques for `sum_x`, `sum_y` race condition.

Table 15 has shown that `reduction` outperforms `omp critical` and `omp atomic`. Therefore, `reduction` will be used to avoid the race conditions for `sum_x` and `sum_y`.

4.3.2 Static Scheduling

Value	Baseline (seconds)	Chunk Size (seconds)				
		default	1	2	4	8
Number of Bodies (N)						
256	0.037	0.048	0.048	0.052	0.050	0.050
512	0.144	0.111	0.117	0.117	0.117	0.118
1024	0.569	0.296	0.319	0.349	0.334	0.322
2048	2.300	0.874	0.975	1.018	1.004	1.000
4096	9.353	2.942	3.174	3.604	3.303	3.195

Activity Grid Dimension (D)						
100	0.572	0.293	0.453	0.323	0.312	0.306
1000	0.599	0.328	0.365	0.390	0.385	0.372
2000	0.736	0.483	0.531	0.545	0.526	0.517
5000	1.651	1.466	1.558	1.559	1.544	1.548
10000	5.020	5.060	5.171	5.150	5.103	5.119

Table 16: Benchmark of parallelising inner force calculation loop with static scheduling.

The results in **Table 16** have shown that the default chunk size has the best overall performance improvements. This indicates that the workloads are equally shared between each thread.

The benchmark where $\mathbf{N} = 256$ shows that the baseline performance is faster than the parallelised performance. This is possible as there is some overhead in forking/joining of threads, and the overhead added is larger than the performance improvements achieved. Hence, if the value of **N** is small, it is better to execute the loop in serial.

4.3.3 Dynamic Scheduling

Value	Baseline (seconds)	Chunk Size (seconds)					
		1 (default)	2	4	8	32	64
Number of Bodies (N)							
256	0.037	0.718	0.453	0.310	0.307	0.292	0.282
512	0.144	1.886	1.498	1.250	0.699	0.602	0.587
1024	0.569	6.877	4.809	3.392	2.817	1.271	1.244
2048	2.300	25.224	15.580	12.094	7.881	4.287	3.087
4096	9.353	83.913	56.561	40.620	26.950	11.991	10.290
Activity Grid Dimension (D)							
100	0.572	6.543	4.571	3.254	2.861	1.283	1.243
1000	0.599	6.451	4.631	3.336	2.880	1.317	1.283
2000	0.736	6.610	4.767	3.505	3.029	1.488	1.450
5000	1.651	8.036	5.773	4.485	4.005	2.493	2.436
10000	5.020	11.364	9.273	7.983	7.659	6.030	5.998

Table 17: Benchmark of parallelising inner force calculation loop with dynamic scheduling.

As shown in **Table 17**, dynamic scheduling has caused the execution time to increase. Although the increase in execution time is lower with larger chunk sizes, it is still slower than the baseline performance. Hence, dynamic scheduling is not applicable for this parallelisation strategy.

4.3.4 Performance Improvements

Based on the benchmark results and analysis in **Sections 4.3.2 and 4.3.3**, static scheduling with default chunk size is selected as the scheduling type for this parallelisation strategy.

Average performance speedup for **Number of Bodies (N)** with `schedule(static)`:

$$\frac{\frac{0.037}{0.048} + \frac{0.144}{0.111} + \frac{0.569}{0.296} + \frac{2.300}{0.874} + \frac{9.353}{2.942}}{5} = 2.06$$

Average performance speedup for **Activity Grid Dimension (N)** with `schedule(static)`:

$$\frac{\frac{0.572}{0.293} + \frac{0.599}{0.328} + \frac{0.736}{0.483} + \frac{1.651}{1.466} + \frac{5.020}{5.060}}{5} = 1.48$$

4.4 `step()`: Activity Map Normalisation Loop

```
1  static void step(void) {
2      ...
3      const float normalise = (float)D / (float)N;
4  #pragma omp parallel for shared(activity_map) if (M == OPENMP)
5      for (i = 0; i < (int)grid_size; ++i) {
6          activity_map[i] *= normalise;
7      }
8  }
```

Listing 15: Parallelising the activity map normalisation loop in `step()` function.

4.4.1 Static Scheduling

Value	Baseline (seconds)	Chunk Size (seconds)				
		default	1	2	4	8
Number of Bodies (N)						
256	0.037	0.066	0.066	0.066	0.066	0.067
512	0.144	0.264	0.263	0.263	0.264	0.263
1024	0.569	1.054	1.055	1.056	1.062	1.053
2048	2.300	4.213	4.215	4.209	4.229	4.216
4096	9.353	14.052	14.102	14.083	14.119	14.142
Activity Grid Dimension (D)						
100	0.572	1.052	1.054	1.053	1.053	1.053
1000	0.599	1.090	1.114	1.103	1.105	1.096
2000	0.736	1.240	1.316	1.283	1.293	1.297
5000	1.651	2.312	2.875	2.524	2.589	2.642
10000	5.020	5.924	9.854	8.617	8.450	7.338

Table 18: Benchmark of parallelising activity map loop with static scheduling.

4.4.2 Dynamic Scheduling

Value	Baseline (seconds)	Chunk Size (seconds)			
		1 (default)	2	4	8
Number of Bodies (N)					
256	0.037	0.067	0.067	0.067	0.067
512	0.144	0.264	0.265	0.267	0.264
1024	0.569	1.054	1.053	1.059	1.055
2048	2.300	4.222	4.215	4.242	4.218
4096	9.353	14.128	14.128	14.72	14.158

Activity Grid Dimension (D)					
100	0.572	1.085	1.070	1.062	1.080
1000	0.599	3.871	2.485	1.808	1.467
2000	0.736	11.819	6.787	4.032	2.677
5000	1.651	70.271	37.69	19.927	11.326
10000	5.020	274.31	144.390	76.251	41.420

Table 19: Benchmark of parallelising activity map loop with dynamic scheduling.

4.4.3 Performance Improvements

No performance improvements have been observed in **Tables 18 and 19**. Conversely, parallelising the activity map normalisation loop has an adverse effect on performance. Since the loop only has a single multiplication operation, the performance improvements might not be sufficient to cover the forking/joining overhead of threads. N.B. Therefore, the activity map normalisation loop is already in optimal condition and does not require any further modification.

4.5 Parallel Nesting Analysis

This section aims to investigate the possibility to further improve the performance through nested parallelism.

4.5.1 Outer N-bodies Loop + Inner Overall Force Calculation Loop

Value	Baseline (seconds)	Outer N-bodies + Inner Force Calculation (seconds)	Speedup
Number of Bodies (N)			
256	0.037	0.321	0.12
512	0.144	0.388	0.37
1024	0.569	0.594	0.96
2048	2.300	1.172	1.96
4096	9.353	3.350	2.79
		Average	**1.24**
Activity Grid Dimension (D)			
100	0.572	0.640	0.89
1000	0.599	0.615	0.97
2000	0.736	0.817	0.90
5000	1.651	1.854	0.89
10000	5.020	5.511	0.91
		Average	**0.91**

Table 20: Parallelising both outer N-bodies and inner force calculation loop.

The performance speedup achieved by parallel nesting is poor in comparison with the parallelisation strategy in **Sections 4.2 and 4.3**. Possible reasons are listed as follow:

1. Parallel nesting may create too many threads and cause thread oversubscription [4], where the number of simultaneously working threads are more than the number of available logical cores on the system. This can cause performance issues as the CPU would spend more time switching between tasks (context switching) [7].
2. The extra overhead added for additional forking/joining of threads in the nested parallel region. If the performance improvements are not sufficient to cover the extra overhead, then the overall performance will decrease.

4.6 Summary

Summarising the performance improvements achieved in iteration mode by each parallelisation strategy:

1. `step()`: Outer N-bodies Loop, (static, default chunk).
   - Average speedup for **N**: 3.54
   - Average speedup for **D**: 2.28
2. `step()`: Inner Overall Force (F_i) Calculation Loop, (static, default chunk)
   - Average speedup for **N**: 2.06
   - Average speedup for **D**: 1.48
3. `step()`: Outer N-bodies + Inner Overall Force (F_i) Calculation Loop
   - Average speedup for **N**: 1.24
   - Average speedup for **D**: 0.91

The result above has shown that **parallelising the outer N-bodies loop** is the optimal strategy.

5 Visualisation Performance Improvements

All benchmarks in the previous sections are being carried out in iteration mode to measure and compare the performance of the program over large simulations. This sections aims to compare the visualisation performance improvements between the serial and parallel implementation.

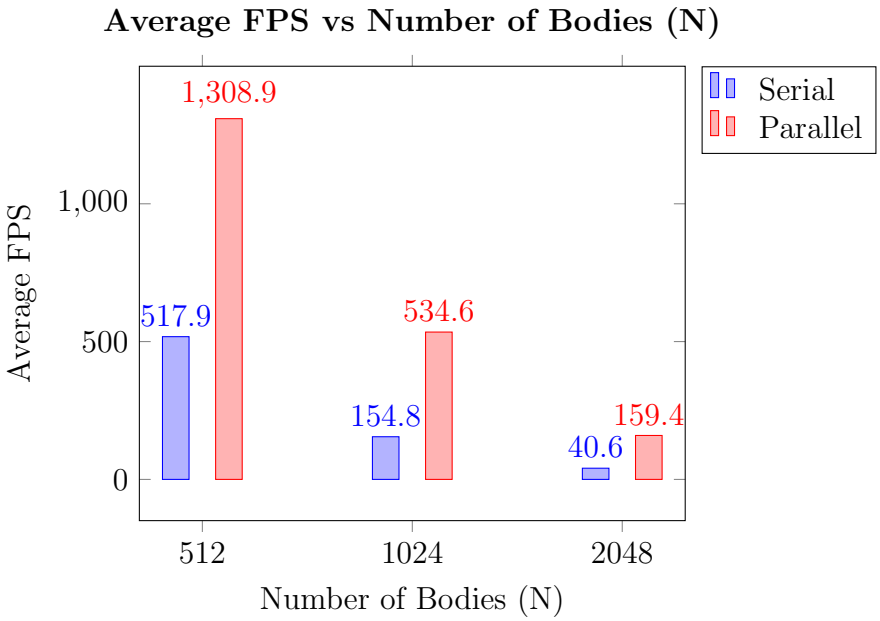

Figure 5: Comparison between the average FPS of serial and parallel implementation in visualisation mode.

As shown in **Figure 5**, parallelising the outer N-bodies loop will increase the performance in visualisation mode as well.

Figure 6: Interface of CapFrameX [3], a free software for FPS monitoring.

References

[1] *Compile error c2057*, Microsoft. [Online]. Available: `https://docs.microsoft.com/en-us/cpp/error-messages/compiler-errors-1/compiler-error-c2057` (visited on 16/03/2020).

[2] *Compile error c3016*, Microsoft. [Online]. Available: `https://docs.microsoft.com/en-us/cpp/error-messages/compiler-errors-2/compiler-error-c3016` (visited on 13/03/2020).

[3] DevTechProfile, *Devtechprofile/capframex*. [Online]. Available: `https://github.com/DevTechProfile/CapFrameX` (visited on 15/03/2020).

[4] *Some tips on using nested parallelism*, Oracle. [Online]. Available: `https://docs.oracle.com/cd/E19205-01/819-5270/aewbj/index.html` (visited on 16/03/2020).

[5] *Storage of basic types*, Microsoft. [Online]. Available: `https://docs.microsoft.com/en-us/cpp/c-language/storage-of-basic-types` (visited on 12/03/2020).

[6] *Strtoul*, Microsoft. [Online]. Available: `https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/strtoul-strtoul-l-wcstoul-wcstoul-l` (visited on 12/03/2020).

[7] *Thread oversubscription*, Intel. [Online]. Available: `https://software.intel.com/en-us/vtune-help-thread-oversubscription` (visited on 16/03/2020).