Assignment 1

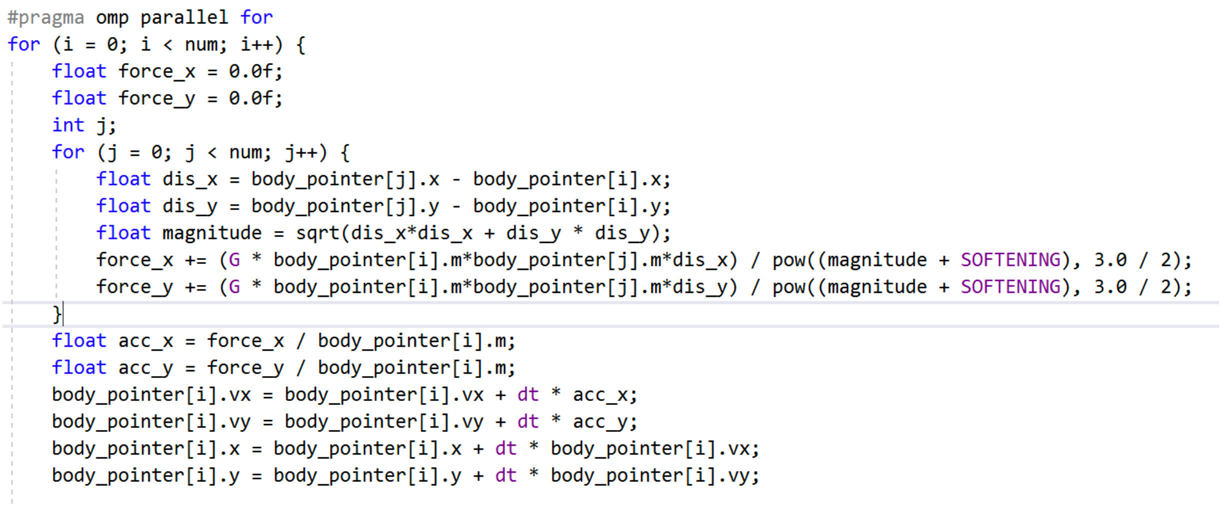
There are two function that was created in N-body simulation project. Those are step() and acti_map_func(). The first function could calculate the latest situation of bodies in the system. The second function is able to count the densities of grid and store them into one dimensional array.

1. Step function:

There are three loop including iteration, outer and inner loop in step function. To be specific, iteration is the number of simulation. Outer loop is the traversal of every body in N-body system. Inner loop is for calculating the sum of force from other bodies. Three parallel strategies including outer loop parallel, inner loop parallel + critical section, parallel nesting, will be discussed. After comparing those strategies, the conclusion is that outer loop parallel with dynamic schedule will be used in step function.

1.1 Outer loop:

Firstly, let's compare running time between serial and for loop parallel. There are variant for loop parallel strategies based on different schedule including static, dynamic and guided.



```
#pragma omp parallel for
for (i = 0; i < num; i++) {
    float force_x = 0.0f;
    float force_y = 0.0f;
    int j;
    for (j = 0; j < num; j++) {
        float dis_x = body_pointer[j].x - body_pointer[i].x;
        float dis_y = body_pointer[j].y - body_pointer[i].y;
        float magnitude = sqrt(dis_x*dis_x + dis_y * dis_y);
        force_x += (G * body_pointer[i].m*body_pointer[j].m*dis_x) / pow((magnitude + SOFTENING), 3.0 / 2);
        force_y += (G * body_pointer[i].m*body_pointer[j].m*dis_y) / pow((magnitude + SOFTENING), 3.0 / 2);
    }
    float acc_x = force_x / body_pointer[i].m;
    float acc_y = force_y / body_pointer[i].m;
    body_pointer[i].vx = body_pointer[i].vx + dt * acc_x;
    body_pointer[i].vy = body_pointer[i].vy + dt * acc_y;
    body_pointer[i].x = body_pointer[i].x + dt * body_pointer[i].vx;
    body_pointer[i].y = body_pointer[i].y + dt * body_pointer[i].vy;
```

Argument: num = 1000 grid = 5 iteration = 100			
Serial	9.668	9.744	9.711
Parallel for	2.466	2.387	2.429
schedule(static, 1)	2.478	2.445	2.497
schedule(static, 2)	2.441	2.426	2.511
schedule(static, 4)	2.517	2.489	2.345
schedule(dynamic)	**1.917**	**2.18**	**2.12**
schedule(guided)	**2.192**	**2.244**	**2.184**

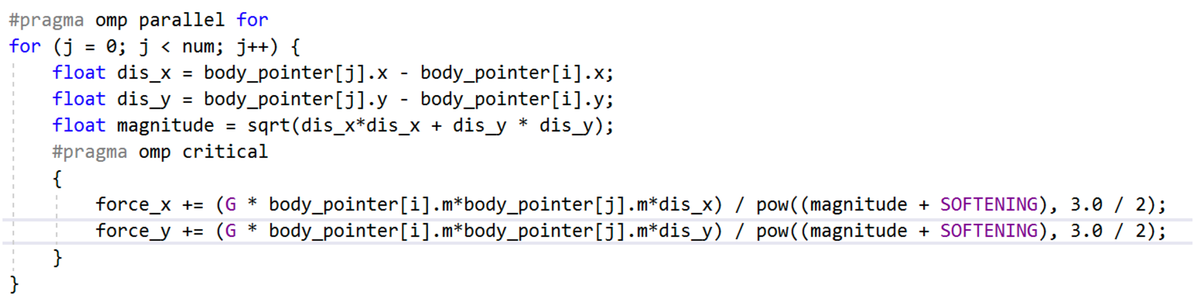
According to the result above, the program could achieve better performance when schedule are guided, dynamic. Undoubtedly, dynamic and guided could solve unbalance of overload of threads, which could perform better. Let's increase the number of bodies and decrease gird and iteration because this part is the calculation of latest situation of bodies in system.

Argument: grid = 3 iteration = 1		
Schedule	**Guided**	**Dynamic**
Num = 5000	0.433	0.426
	0.463	0.411
	0.427	0.423
Num = 10,000	1.731	1.731
	1.762	1.788
	1.741	1.747
num = 50,000	44.345	44.83
	44.837	43.918
	43.721	43.720

As a consequence, dynamic and guided have similar performance. In this case, dynamic schedule will be used in outer loop.

1.2 Inner loop:

Secondly, let's compare running time between serial and inner loop parallel with critical section.



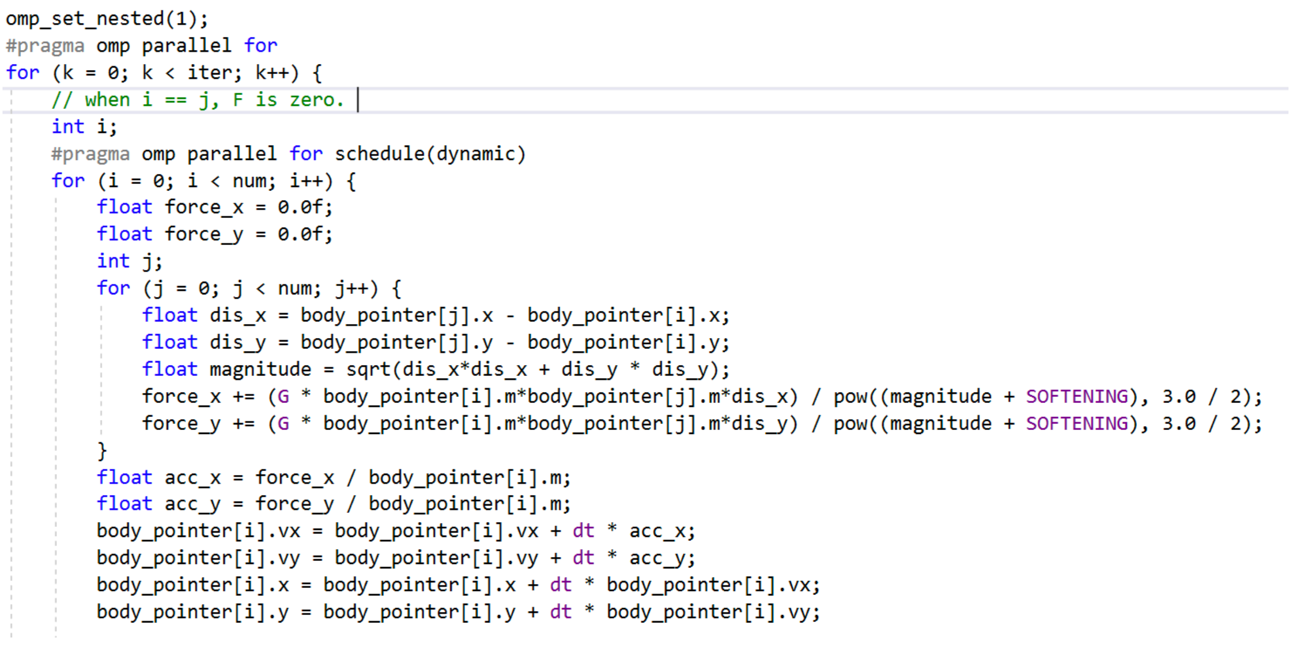
```
#pragma omp parallel for
for (j = 0; j < num; j++) {
    float dis_x = body_pointer[j].x - body_pointer[i].x;
    float dis_y = body_pointer[j].y - body_pointer[i].y;
    float magnitude = sqrt(dis_x*dis_x + dis_y * dis_y);
    #pragma omp critical
    {
        force_x += (G * body_pointer[i].m*body_pointer[j].m*dis_x) / pow((magnitude + SOFTENING), 3.0 / 2);
        force_y += (G * body_pointer[i].m*body_pointer[j].m*dis_y) / pow((magnitude + SOFTENING), 3.0 / 2);
    }
}
```

Argument: num = 1000 grid = 5 iteration = 100	
Serial	**OPENMP**
9.636	37.496
9.821	33.994
9.643	36.191

According to the result, the program based on inner loop parallel is more time-consuming than serial. The reason is because of critical section. It should be used to control shared variable force_x and force_y because multiple threads could write them at the same time if critical section is not used. In this case, threads cannot be allocated to other iteration because it is waiting to write shared variables. As a result, ‘inner loop parallel’ strategy will be discarded because critical section cannot be avoided.

1.3 simulation loop:

Thirdly, the parallel nesting will be used in iteration loop and outer loop. The next is the code.



```
omp_set_nested(1);
#pragma omp parallel for
for (k = 0; k < iter; k++) {
    // when i == j, F is zero.
    int i;
    #pragma omp parallel for schedule(dynamic)
    for (i = 0; i < num; i++) {
        float force_x = 0.0f;
        float force_y = 0.0f;
        int j;
        for (j = 0; j < num; j++) {
            float dis_x = body_pointer[j].x - body_pointer[i].x;
            float dis_y = body_pointer[j].y - body_pointer[i].y;
            float magnitude = sqrt(dis_x*dis_x + dis_y * dis_y);
            force_x += (G * body_pointer[i].m*body_pointer[j].m*dis_x) / pow((magnitude + SOFTENING), 3.0 / 2);
            force_y += (G * body_pointer[i].m*body_pointer[j].m*dis_y) / pow((magnitude + SOFTENING), 3.0 / 2);
        }
        float acc_x = force_x / body_pointer[i].m;
        float acc_y = force_y / body_pointer[i].m;
        body_pointer[i].vx = body_pointer[i].vx + dt * acc_x;
        body_pointer[i].vy = body_pointer[i].vy + dt * acc_y;
        body_pointer[i].x = body_pointer[i].x + dt * body_pointer[i].vx;
        body_pointer[i].y = body_pointer[i].y + dt * body_pointer[i].vy;
```

The next table demonstrates the difference between nesting and outer loop parallel.

Argument: num = 1000 gird = 3		
Schedule	**Nest**	**Outer loop parallel**
Iteration = 100	1.807	2.10
Iteration = 500	8.880	10.135
Iteration = 1000	17.854	20.284
Iteration = 3000	53.938	59.847
Iteration = 5000	93.419	102.116

Obviously, parallel nesting is less time consuming. Let’s compare the running time with variant num.

Argument: iteration = 500 grid = 3		
	Nest	**Outer loop parallel**
Num = 300	0.816	2.53
Num = 500	2.565	3.498
Num = 1000	9.236	9.977
Num = 3000	82.52	81.346

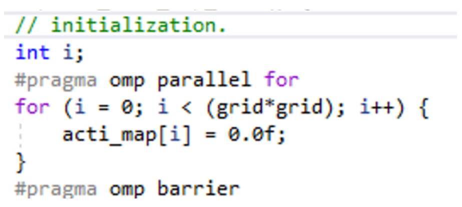
According to the result, nesting is a little lower than no nesting parallel when num is bigger than 3000. The reason might be that the creation and destroy of thread is time consuming. This phenomenon is more obvious when we use nested loop parallel. Therefore, parallel nesting will not be used in step function.

2. Activity map function:

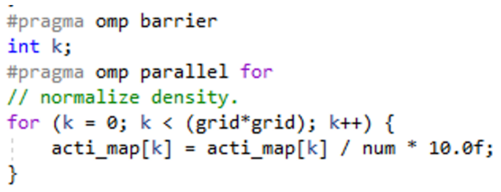
There are three steps in this function. Firstly, it initialises density array to zero. Secondly, calculate the index of one dimensional array from two dimensional. Thirdly, normalise the array to the range between 0 and 1. According to the result, those three parts will use for loop parallel respectively, and use barrier to avoid error happen.

2.1 Initialisation and Normalisation:

As for initialisation and normalisation of one dimensional array, it could use for loop parallel and barrier in the middle of several parallel blocks. The code is displayed in the next figure.



```
// initialization.
int i;
#pragma omp parallel for
for (i = 0; i < (grid*grid); i++) {
    acti_map[i] = 0.0f;
}
#pragma omp barrier
```



```
#pragma omp barrier
int k;
#pragma omp parallel for
// normalize density.
for (k = 0; k < (grid*grid); k++) {
    acti_map[k] = acti_map[k] / num * 10.0f;
}
```

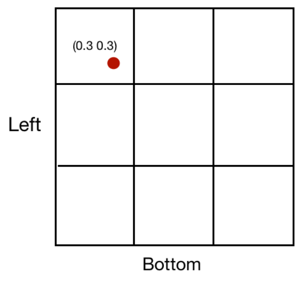
The next table is the comparison of performance between them.

Argument: num = 100, iteration = 100		
	Serial	**For loop parallel**
Gird = 1000	1.106	0.292
Grid = 5000	19.585	6.240
Grid = 10,000	67.756	24.526

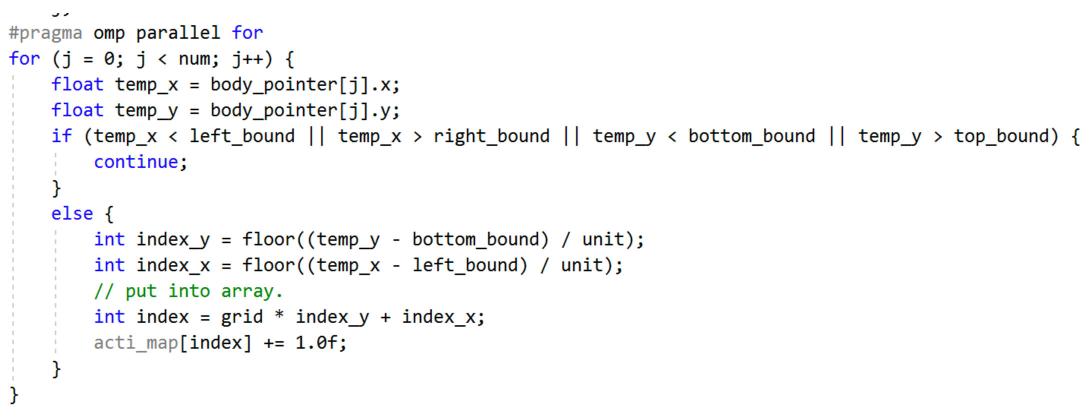
According to the result, for loop parallel and barrier strategy could decrease running time. Therefore, it will be used in this function.

2.2 Density Calculation:

The next code is the calculation of density of bodies in every grid. It calculate the index of one dimensional directly, rather than using two dimensional array. For example, there is a body whose position is 0.3 and 0.3. the grid is 3. The left and bottom boundary is 0 respectively.



Firstly, the unit is 1/grid. In this case is 0.333. Secondly, the index_y and index_x is floor((y - bottom) / unit) and floor((x - left) / unit) respectively. In this case, index_y and index_x are both 0. Therefore, the index of one dimensional array, grid * index_y + index_x, is 0. In this way, the limitation of memory band width could be ignored and it could improve the performance because it avoid using two dimensional array. The next is the code.



```
#pragma omp parallel for
for (j = 0; j < num; j++) {
    float temp_x = body_pointer[j].x;
    float temp_y = body_pointer[j].y;
    if (temp_x < left_bound || temp_x > right_bound || temp_y < bottom_bound || temp_y > top_bound) {
        continue;
    }
    else {
        int index_y = floor((temp_y - bottom_bound) / unit);
        int index_x = floor((temp_x - left_bound) / unit);
        // put into array.
        int index = grid * index_y + index_x;
        acti_map[index] += 1.0f;
    }
}
```

3. Conclusion

In this assignment, there are two functions. Those are step() and acti_map_func(). In step function, outer for loop parallel based on dynamic schedule was used. In acti_map_func function, for loop parallel and barrier were used together.