High Performance and Parallel Computing

Assignment 4 - Group 08

Dinindu Seneviratne (19920722T699)
Dhanushki Mapitigama (19950811T789)
Hemavathi Hanasoge Siddaiah (199305182827)

Introduction

In this mini-project, we have attempted to simulate the movement of stars in a two-dimensional space (galaxy) with respect to constant increments of time. We have been given many input data files which consist of data which represents the mass, position velocity and brightness of different celestial objects in a galaxy.

The objective of the project is to first come up with an implementation in C to calculate the state of each object after a given number of steps, and then attempt to optimize the code to reduce the processing time.

The approach to calculate the state of the objects after a given number of steps is based on Newton's Law of Gravitation in two dimensions. Given a distribution of N particles, a straight-forward calculation of the force exerted on particle i by the other $N - 1$ particles is given by (using C-style indexing),

$$F_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{r_{ij}^2} \cdot \hat{\boldsymbol{r}}_{\boldsymbol{ij}}$$

Where m_i and m_j are masses of two objects, r_{ij} is the distance between the objects and $\hat{r}_{ij}$ is the normalized distance vector. There is a built-in instability in the given formulation when $r_{ij} \ll 1$. To deal with this, we use a slightly modified force that corresponds to so-called Plummer spheres as follows:

$$\mathbf{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{\left(r_{ij} + \epsilon_0\right)^3} \mathbf{r}_{ij}.$$

where ε_0 is a small number (we will be using $10{-}3$). Then from the symplectic Euler time integration method, the velocity u_i and position x_i of particle i can then be updated with,

$$\begin{aligned} \mathbf{a}_i^n &= \frac{\mathbf{F}_i^n}{m_i}, \\ \mathbf{u}_i^{n+1} &= \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n, \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1} \end{aligned}$$

We have taken the input data as the 0^{th} step to calculate the state for the next step. Likewise, through iteration, we have come up with several versions of implementations to arrive at an n^{th} state.

Initial Implementation

For the initial implementation, we used a practically straightforward approach and we did not focus a lot on optimization at this step. The code we used for reading a file and simulation is given below. Throughout our implementation, we use a struct of arrays called Particles to store the data related to all the particles.

```
typedef struct
{
    double *posx;
    double *posy;
    double *mass;
    double *velx;
    double *vely;
    double *accx;
    double *accy;
    double *brightness;
} Particles;
```

When reading the input data, we first run some checks on the input arguments and then use `fread` to read the data in blocks of six into the Particle array as follows.

```
Particles *read_data_v1(int particle_count, char *filename)
{

    /* Open input file and determine its size. */
    FILE *input_file = fopen(filename, "rb");

    if (!input_file)
    {
        printf("read_doubles_from_file error: failed to open input file '%s'.\n", filename);
        return NULL;
    }

    /* Get filesize using fseek() and ftell(). */
    fseek(input_file, 0L, SEEK_END);
    size_t fileSize = ftell(input_file);

    /* Now use fseek() again to set file position back to beginning of the file. */
    fseek(input_file, 0L, SEEK_SET);

    if (fileSize != 6 * particle_count * sizeof(double))
    {
        printf("read_doubles_from_file error: size of input file '%s' does not match the
given n.\n", filename);
        printf("For n = %d the file size is expected to be (n * sizeof(double)) = %lu but
the actual file size is %lu.\n",
               particle_count, 6 * particle_count * sizeof(double), fileSize);
        return NULL;
```

```
    }

    double buffer[6 * particle_count];
    if (!fread(buffer, sizeof(char), fileSize, input_file))
    {
        printf("Failed to read.\n");
    }

    Particles *particles = malloc(sizeof(Particles));

    // Allocate memory for each array member
    particles->posx = malloc(particle_count * sizeof(double));
    particles->posy = malloc(particle_count * sizeof(double));
    particles->mass = malloc(particle_count * sizeof(double));
    particles->velx = malloc(particle_count * sizeof(double));
    particles->vely = malloc(particle_count * sizeof(double));
    particles->accx = malloc(particle_count * sizeof(double));
    // memset(particles->accx, 0, particle_count);
    particles->accy = malloc(particle_count * sizeof(double));
    // memset(particles->accy, 0, particle_count);
    particles->brightness = malloc(particle_count * sizeof(double));

    for (int i = 0; i < particle_count; i++)
    {
        particles->posx[i] = buffer[(6 * i) + 0];
        particles->posy[i] = buffer[(6 * i) + 1];
        particles->mass[i] = buffer[(6 * i) + 2];
        particles->velx[i] = buffer[(6 * i) + 3];
        particles->vely[i] = buffer[(6 * i) + 4];
        particles->brightness[i] = buffer[(6 * i) + 5];
        particles->accx[i] = 0.0;
        particles->accy[i] = 0.0;
        // we don't initiate accx and accy at this point - the values will be null
    }

    fclose(input_file);
    return particles;
}
```

In the initial simulation, we have three nested for-loops which iterates for the number of steps given by the user, and for each of the steps, we iterate through all the particles to calculate their velocity and position at the next time step using another for-loop to iterate over the rest of the particles.

```
void simulate_v1(Particle *particles, int particle_count, int G, int steps, double delta_t)
{
    double a_x, a_y; // X and Y components of the acceleration vector
    double r_x, r_y; // X and Y components of the relative position vector
    double r_xy; // Distance between two particles
    double r_xy_eps_3; // Distance between two particles plus epsilon to the power 3
```

```
    // Run simulations for given number of steps
    for (int iteration = 0; iteration < steps; iteration++)
    {
        // Loop over all particles
        for (int i = 0; i < particle_count; i++)
        {
            a_x = 0.0;
            a_y = 0.0;

            // Loop over all particles
            for (int j = 0; j < particle_count; j++)
            {
                if (i != j)
                {
                    r_x = particles[i].posx - particles[j].posx;
                    r_y = particles[i].posy - particles[j].posy;

                    r_xy = sqrt(pow((r_x), 2) + pow((r_y), 2));
                    r_xy_eps_3 = pow(r_xy + eps, 3);

                    a_x += (particles[j].mass * r_x) / r_xy_eps_3;
                    a_y += (particles[j].mass * r_y) / r_xy_eps_3;
                }
            }

            a_x = (-1) * G * a_x;
            a_y = (-1) * G * a_y;

            particles[i].velx += delta_t * a_x;
            particles[i].vely += delta_t * a_y;
        }

        for (int i = 0; i < particle_count; i++)
        {
            particles[i].posx += delta_t * particles[i].velx;
            particles[i].posy += delta_t * particles[i].vely;
        }
    }
}
```

Although the above code gives a pos_maxdiff = 0.00000000000, it takes more than 200 seconds to complete execution for `ellipse_N_03000.gal`. In the next section we discuss our approach to optimize the code to get better and quicker results.

Optimization Plan

The approach we took to optimize our implementation was based on a few optimization strategies.

1. Mathematical Simplification
2. Removal of function calls
3. Multiplication over Division

4. Use of constant variables
5. Compiler optimization
6. Using Newton's Third law to reduce loop iterations

Mathematical Simplification

We have used some mathematical simplification to make our code more efficient. Our goal is to calculate x_i^{n+1} using u_i^{n+1}, and u_i^{n+1} using a_i^n. Hence we use the following simplification to calculate the a_i^n.

$$
\begin{aligned}
a_i^n &= \frac{F_i^n}{m_i} \\
&= \frac{-G \cdot \cancel{m_i} \cdot \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3} \cdot \boldsymbol{r_{ij}}}{\cancel{m_i}} \\
&= -G \cdot \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3} \cdot \boldsymbol{r_{ij}}
\end{aligned}
$$

By doing the above simplification we could skip the calculation of force (F) and straightaway calculate the acceleration.

Removal of function calls

Another method we have used to optimize the code is by removing unnecessary function calls inside loops. In simulation version 2, we have removed the use of ***pow*** function used to calculate power and replaced it with multiplication. For example `pow(x, 3) = x * x * x;`

Multiplication over division

In simulation version 3, we have replaced the operations which involved division to be handled by multiplication, since the division operation is computationally much more expensive than the multiplication. An example code block is shown below.

```
div_r_xy_eps_3 = 1 / r_xy_eps_3;
a_x += (particles[j].mass * r_x) * div_r_xy_eps_3;
a_y += (particles[j].mass * r_y) * div_r_xy_eps_3;
```

Use of constant variables

Another method of optimization is to use constant variables whenever possible. When constant variables are used the compiler could make certain assumptions about the variable and make optimizations during compilations. In this case since G and epsilon are global constants. So we have used constant variables in simulation version 4 to initiate them.

Usage of compiler optimization

In addition to the optimizations done in the code we used compiler optimizations to achieve more speed.

Using Newton's Third law to reduce loop iterations

From the logic that each particle experiences an equal and opposite reaction for the force it applies on the other particles, we have done the following optimization.

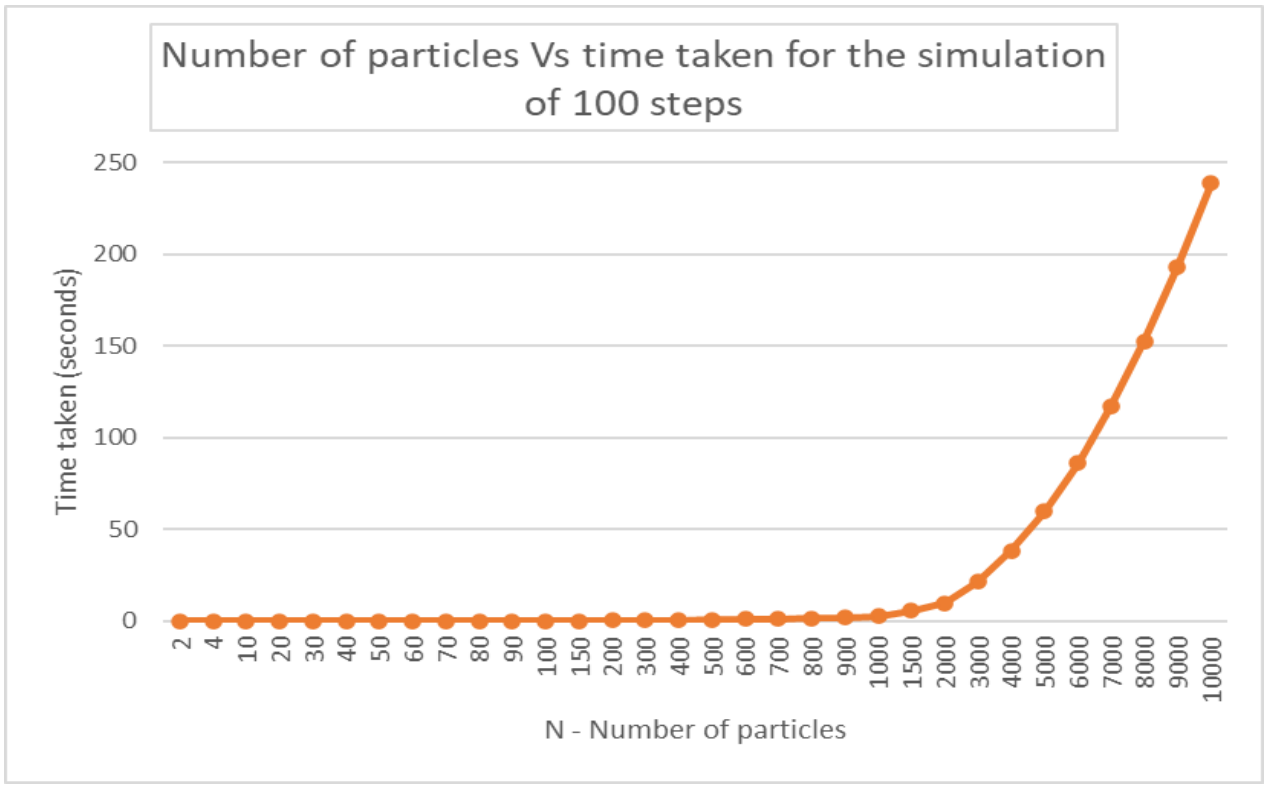
```
#if VERSION == 1
    // Start simulation - Optimized version 1
    for (int step = 0; step < nsteps; step++)
    {
        for (int i = 0; i < N; i++)
        {
            particles->accx[i] = 0.0;
            particles->accy[i] = 0.0;
            for (int j = 0; j < N; j++)
            {
                if (i != j)
                {
                    rx = particles->posx[i] - particles->posx[j];
                    ry = particles->posy[i] - particles->posy[j];

                    r = sqrt(rx * rx + ry * ry);
                    rr = r + epsilon;
                    div_1_rr = 1 / (rr * rr * rr);
                    particles->accx[i] += particles->mass[j] * rx *
div_1_rr;
                    particles->accy[i] += particles->mass[j] * ry *
div_1_rr;
                }
            }
            particles->velx[i] += dtG * particles->accx[i];
            particles->vely[i] += dtG * particles->accy[i];
        }

        for (int i = 0; i < N; i++)
        {
            particles->posx[i] += particles->velx[i] * delta_t;
            particles->posy[i] += particles->vely[i] * delta_t;
        }
    }
```

After implementing the above optimization, we could reduce the processing time further down to 1.584935 seconds in Apple M2.

Serial Optimization Results and Discussion

Variation of processing time with N

Given below is the movement of processing time with variable N (Number of particles). We can clearly see that the time taken for computations rapidly increases with the time taken (please note that the x axis labels are not equally distributed with the same intervals in the given figure.)



The expected $O(N^2)$ complexity is clearly visible from the above results.

Optimization Results

Given below are the time taken for different versions of the program on fredholm.it.uu.se server and the local machine. (We were able to achieve less time on the local machine - Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz). We can see that each optimization improved the processing in both environments.

Optimizations Used	Time taken on server (seconds)	Time taken on local machine (seconds)

V1 : Mathematical simplification only	259.899609	116.255100
V2 : V1 + removal of power function calls	43.100186	16.878140
V3 : V2 + multiplication over division	36.423995	15.974230
V4 : V3 + using constants	36.038836	14.641638
V5 : V4 + using compiler optimizations	21.465265	2.952106

Accuracy of results

The max_diff between the simulation results and the given sample outputs for 3000 particles (for 100 timesteps) was 0.0000000000. With this we concluded that the computations/simulations are accurate.

```
madh4406@fredholm:~/A3$ ./compare_gal/compare_gal_files 3000 result.gal op_data/ellipse_N_03000_after100steps.gal
N = 3000
fileName1 = 'result.gal'
fileName2 = 'op_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
madh4406@fredholm:~/A3$
```

We tested the program for memory leaks using valgrind and the results confirm that there are no memory leaks in the implementation.

```
madh4406@fredholm:~/A3$ gcc -O3 -o galsim galsim.c -lm
madh4406@fredholm:~/A3$ valgrind --leak-check=full --show-leak-kinds=all ./galsim 10 input_data/ellipse_N_00010.gal 100 0.00001 0
==593747== Memcheck, a memory error detector
==593747== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==593747== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==593747== Command: ./galsim 10 input_data/ellipse_N_00010.gal 100 0.00001 0
==593747==
Time taken for the simulation of 10 particals for 100 steps = 0.005089 seconds.
==593747==
==593747== HEAP SUMMARY:
==593747==     in use at exit: 0 bytes in 0 blocks
==593747==   total heap usage: 6 allocs, 6 frees, 18,832 bytes allocated
==593747==
==593747== All heap blocks were freed -- no leaks are possible
==593747==
==593747== For lists of detected and suppressed errors, rerun with: -s
==593747== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Parallelization Plan

In this section we have explained the approach we took to parallelize our implementation using both Pthreads and OpenMP. The general approach we have taken to parallelize the code is as follows.

Since the second loop is perfectly parallelizable, we first split the set of particles into sets of N / number of threads as follows.

```
/* Create multiple threads */
    pthread_t threads[thread_count];
    /* Create an array of indeces */
    int thread_index[thread_count];
    /* Create an array of ThreadInputs */
    ThreadInput thread_input[thread_count];

    // initialize the thread input array
    for (int i = 0; i < thread_count; i++)
    {
        ThreadInput temp_thread_input = {
            (N / thread_count) * i,
            (N / thread_count) * (i + 1),
            N,
            epsilon,
            dtG,
            delta_t,
            particles
        };
        thread_input[i] = temp_thread_input;
    }
```

Pthreads Implementation

Since there are dependencies when it comes to updating the variables within the second loop, we have broken the loop into two serially running loops to update acceleration and velocity of each particle first, and then the position.

```
pthread_mutex_init(&mutex, NULL);

    for (int step = 0; step < nsteps; step++)
    {
        // Start N number of threads for updating acceleration
        for (int i = 0; i < thread_count; i++)
        {
            thread_index[i] = i;
            pthread_create(&threads[i], NULL, update_acceleration_v2,
&thread_input[i]);
        }

        // Join N number of threads after updating acceleration
        for (int i = 0; i < thread_count; i++)
```

```
        {
            pthread_join(threads[i], NULL);
        }

        // Start N number of threads for updating position
        for (int i = 0; i < thread_count; i++)
        {
            thread_index[i] = i;
            pthread_create(&threads[i], NULL, update_position_v2,
&thread_input[i]);
        }

        // Join N number of threads after updating position
        for (int i = 0; i < thread_count; i++)
        {
            pthread_join(threads[i], NULL);
        }
    }
    pthread_mutex_destroy(&mutex);
```

We have used mutex variables to manage concurrent variable access when updating velocity.

```
for (int m = 0; m < thread_input->N; m++){
        pthread_mutex_lock(&mutex);
        thread_input->particles->velx[m] += tmp_velx[m];
        thread_input->particles->vely[m] += tmp_vely[m];
        pthread_mutex_unlock(&mutex);
    }
```

The functions used for implementation of the acceleration / velocity update and the position update is as follows:

```
void *update_acceleration_v2(void *arg)
{
    ThreadInput *thread_input = (ThreadInput *)arg;
    int start_n = thread_input->start_n;
    int end_n = thread_input->end_n;

    // Variables needed for calculations
    double rx, ry, r, rr, div_1_rr, rx_div, ry_div;
```

```
    double tmp_velx[thread_input->N];
    double tmp_vely[thread_input->N];

    memset(tmp_velx, 0, thread_input->N);
    memset(tmp_vely, 0, thread_input->N);

    for (int i = thread_input->start_n; i < thread_input->end_n; i++)
    {
        double tmp_accx = 0.0;
        double tmp_accy = 0.0;
        for (int j = i + 1; j < thread_input->N; j++)
        {
            rx = thread_input->particles->posx[i] -
thread_input->particles->posx[j];
            ry = thread_input->particles->posy[i] -
thread_input->particles->posy[j];
            r = sqrt(rx * rx + ry * ry);
            rr = r + thread_input->epsilon;
            div_1_rr = thread_input->dtG / (rr * rr * rr);
            rx_div = rx*div_1_rr;
            ry_div = ry*div_1_rr;

            // Calculating the acceleration of the i-th particle based on
the forces applied by N-i particles
            tmp_velx[i] += thread_input->particles->mass[j] * rx_div;
            tmp_vely[i] += thread_input->particles->mass[j] * ry_div;
              // Substracting the velocity change on the j-th particle due
to the equal and opposite reaction
            tmp_velx[j] -=  thread_input->particles->mass[i] * rx_div;
            tmp_vely[j] -=  thread_input->particles->mass[i] * ry_div;
        }
    }

    for (int m = 0; m < thread_input->N; m++){
        pthread_mutex_lock(&mutex);
        thread_input->particles->velx[m] += tmp_velx[m];
        thread_input->particles->vely[m] += tmp_vely[m];
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}
```

```
void *update_position_v2(void *arg)
{
    ThreadInput *thread_input = (ThreadInput *)arg;
    int start_n = thread_input->start_n;
    int end_n = thread_input->end_n;

    for (int i = thread_input->start_n; i < thread_input->end_n; i++)
    {
        thread_input->particles->posx[i] +=
thread_input->particles->velx[i] * thread_input->delta_t;
        thread_input->particles->posy[i] +=
thread_input->particles->vely[i] * thread_input->delta_t;
    }

    return NULL;
}
```

With the threading the program took a lot more time to finish the simulation. This happened due to the application of Newton's third law, where we update the velocity change of the equal and opposite force on the two particles as a result of the overhead of locking and unlocking. To avoid this we maintain a temporary array which is local to each thread for the velocity updates which happen within the 2 inner loops in update acceleration function. Once the two inner loops finish execution we lock and update the global velocities. With this, the execution time drastically reduced and we were able to achieve 1.584935 seconds on Apple M2

OpenMP Implementation

A similar implementation as above was carried out in the OpenMP version and we have used omp critical when to manage concurrent variable access when updating velocity. The code is mentioned below.

```
// Start simulation - Parallelized version 2 with OpenMP

    /* Create an array of indeces */
    int thread_index[thread_count];
    /* Create an array of ThreadInputs */
    ThreadInput thread_input[thread_count];

    // initialize the thread input array
    for (int i = 0; i < thread_count; i++)
    {
        ThreadInput temp_thread_input = {
            (N / thread_count) * i,
```

```
            (N / thread_count) * (i + 1),
            N,
            epsilon,
            dtG,
            delta_t,
            particles
        };
        thread_input[i] = temp_thread_input;
    }

    for (int step = 0; step < nsteps; step++)
    {
        // Start N number of threads for updating acceleration
        #pragma omp parallel for simd num_threads(thread_count)
        for (int i = 0; i < thread_count; i++)
        {
            update_acceleration_v2(&thread_input[omp_get_thread_num()]);
        }

        // Start N number of threads for updating position
        #pragma omp parallel for simd num_threads(thread_count)
        for (int i = 0; i < thread_count; i++)
        {
            update_position_v2(&thread_input[omp_get_thread_num()]);
        }
    }
```

The functions used for updating acceleration / velocity and position are given below.

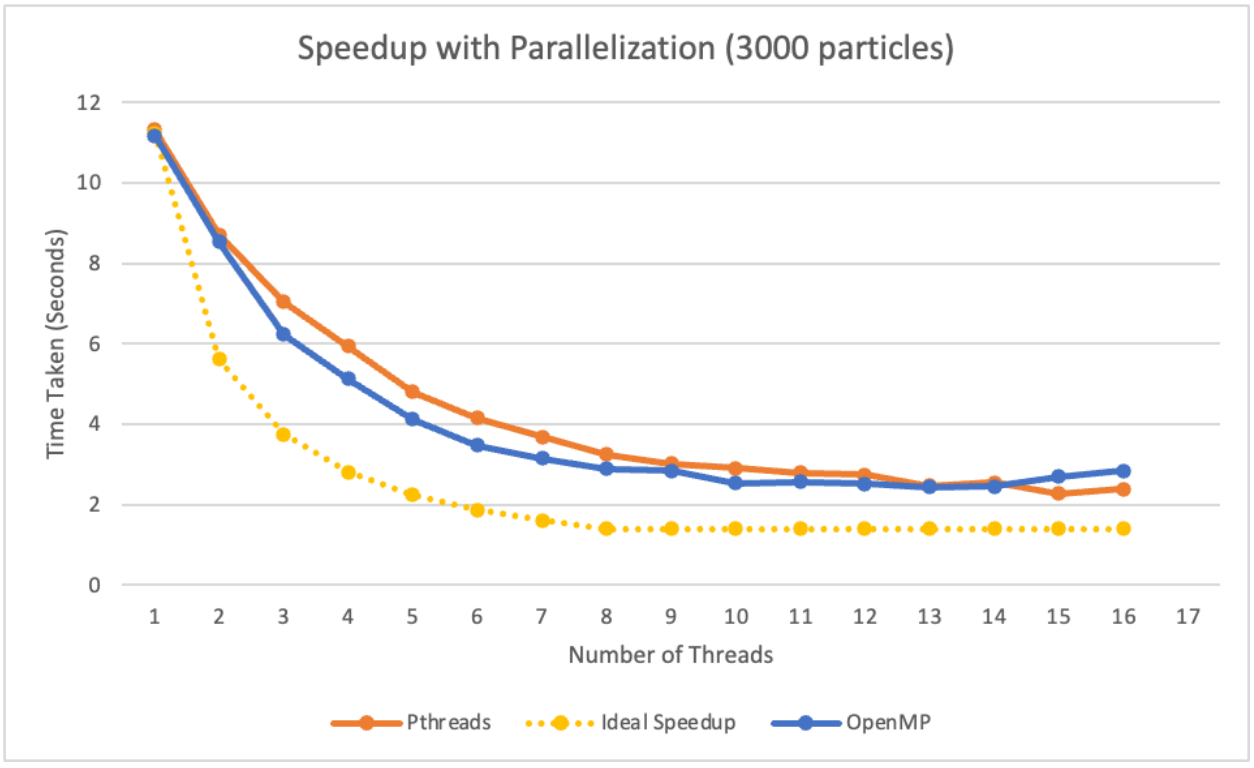
```
void *update_acceleration_v2(void *arg)
{
    ThreadInput *thread_input = (ThreadInput *)arg;
    int start_n = thread_input->start_n;
    int end_n = thread_input->end_n;

    // Variables needed for calcluations
    double rx, ry, r, rr, div_1_rr, rx_div, ry_div;

    double *tmp_velx = malloc(thread_input->N * sizeof(double));
    double *tmp_vely = malloc(thread_input->N * sizeof(double));

    memset(tmp_velx, 0, thread_input->N);
```

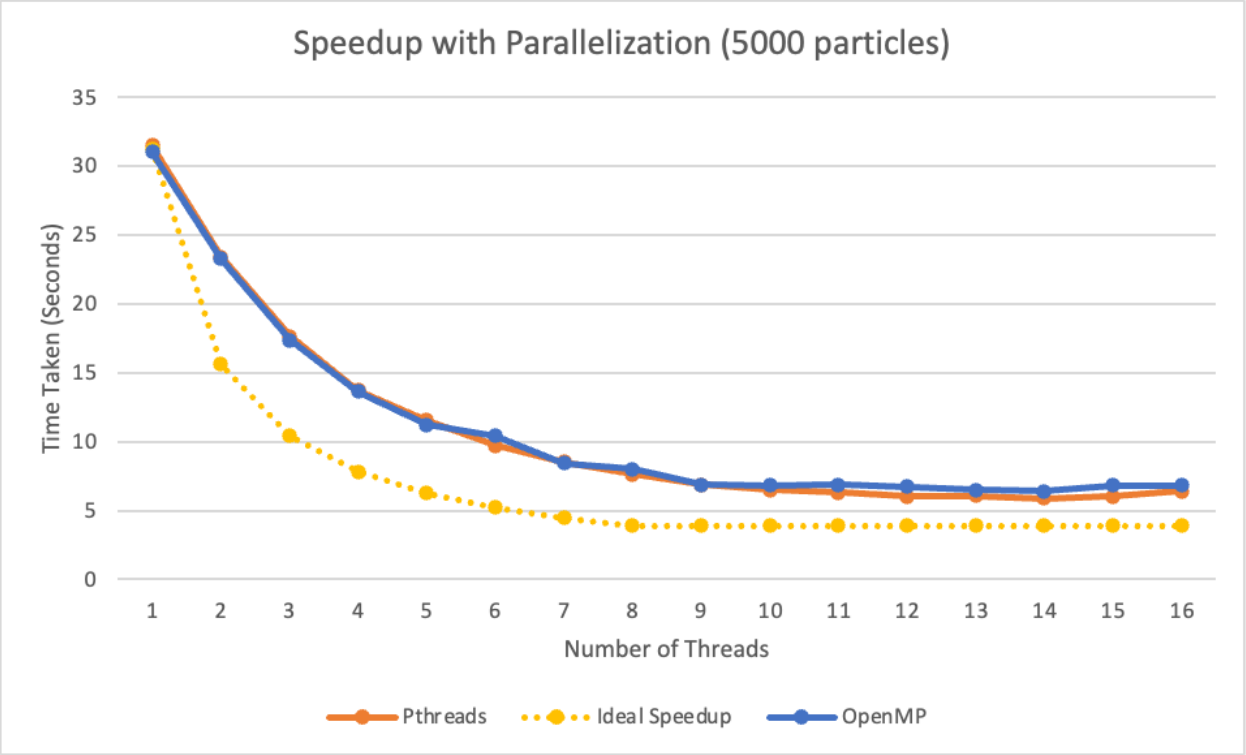
```
    memset(tmp_vely, 0, thread_input->N);

    //printf("Velocity-tmp-x: %lf,, %d\n", tmp_velx[3],
thread_input->start_n);

    for (int i = thread_input->start_n; i < thread_input->end_n; i++)
    {
        double tmp_accx = 0.0;
        double tmp_accy = 0.0;
        for (int j = i + 1; j < thread_input->N; j++)
        {
            rx = thread_input->particles->posx[i] -
thread_input->particles->posx[j];
            ry = thread_input->particles->posy[i] -
thread_input->particles->posy[j];
            r = sqrt(rx * rx + ry * ry);
            rr = r + thread_input->epsilon;
            div_1_rr = thread_input->dtG / (rr * rr * rr);
            rx_div = rx*div_1_rr;
            ry_div = ry*div_1_rr;

            // Calculating the acceleration of the i-th particle based on
the forces applied by N-i particles
            tmp_velx[i] += thread_input->particles->mass[j] * rx_div;
            tmp_vely[i] += thread_input->particles->mass[j] * ry_div;
              // Substracting the velocity change on the j-th particle due
to the equal and opposite reaction
            tmp_velx[j] -=  thread_input->particles->mass[i] * rx_div;
            tmp_vely[j] -=  thread_input->particles->mass[i] * ry_div;
        }
    }

    for (int m = 0; m < thread_input->N; m++){
        #pragma omp critical
        thread_input->particles->velx[m] += tmp_velx[m];
        #pragma omp critical
        thread_input->particles->vely[m] += tmp_vely[m];
    }

    return NULL;
}

void *update_position_v2(void *arg)
```

```
{
    ThreadInput *thread_input = (ThreadInput *)arg;
    int start_n = thread_input->start_n;
    int end_n = thread_input->end_n;

    for (int i = thread_input->start_n; i < thread_input->end_n; i++)
    {
        thread_input->particles->posx[i] +=
thread_input->particles->velx[i] * thread_input->delta_t;
        thread_input->particles->posy[i] +=
thread_input->particles->vely[i] * thread_input->delta_t;
    }

    return NULL;
}
```

Analysis of speed-up

We have run speed-up tests for the scenarios of 3000 and 5000 particles. Following chart depicts the speedup with respect to the threads used for parallelization. We have run our tests in fredholm.it.uu.se which has 8 cores. Hence the ideal speedup after 8 threads is not expected to be very high. From 1 thread to 8 threads the time reduction is about 74%. But the tests do not show trends which conclude that it is easier to get close to ideal speedup for larger problem sizes.





Accuracy of results

The max_diff between the simulation results and the given sample outputs for 3000 particles (for 100 timesteps) was 0.0000000000. With this we concluded that the computations/simulations are accurate.

```
hese7588@fredholm:~/galaxy-simulation-uu/compare_gal_files$ ./compare_gal_files
 3000 result.gal ../ref_output_data/ellipse_N_03000_after100steps.gal
N = 3000
fileName1 = 'result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```