TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG

Die Ressourcenuniversität. Seit 1765.

HPC Project WS18/19

Parallelism of N-Particles Interaction

Meghal Shah
Matriculation Nr. 62853

January 17, 2021

Prüfer:
Prof. Dr. rer. nat. Oliver Rheinbach

Institut für Numerische Mathematik und Optimierung
TU Bergakademie Freiberg

Contents

1 Introduction

In this project, interaction of n particles through a force i.e. Gravitational force is simulated. Resulting from gravitational forces of other particles, each particle moves in different directions.Newton's law of universal gravitation stats that every particle attracts every other particle in the universe with a force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between their centers (1). The equation for universal gravitation is given by equation (1) (1):

$$F = G\frac{m_1 m_2}{r^2} \tag{1}$$

The acceleration a is related to the to the force F through $F = ma$, where m is mass of the particle. Time by time particle advances by new position and velocity. To simulate position of all the particles over time, first we need to find interaction forces between all the particles. Through forces, new positions and velocities could be found out by above relation.

Generally, simulating n particles in space would require more computational power and time, if sequential code is used. But, it is possible to allocate chunk from whole size of particles to different processes to reduce computational time and effort. This could be done by Massage Passing Interface (MPI).

1.1 Tasks

- Write Parallel program for simulating n particles using MPI
- Perform strong and weak scalability tests on TUBAF cluster

1.2 Approach

- By choosing reasonable time step size (dt), enough particles n and total time steps run C parallel program
- Check for strong and weak scalability test
- Get output of positions of particles and visualize with Plotly Python

2 Implementation

The simulation of movement of particles starts with generation of *n* particles in 3-Dimension space. Each particles will have random positions in 3-Dimension simulation box. Similarly, masses and velocities of the particles are assumed with random function *rand*(). All particles interact with each other with gravitational force, which can be calculated by their masses and distance between the particles. Finally, to check movement of particles, new positions and velocities are calculated over time. This process can be parallelized to save computational time by using MPI commands.

2.1 Steps for MPI Implementation

1. Variables, arrays and pointers are initialized.
2. Enable MPI Environment by *MPI_Init(&argc,&argv)* function.
3. Get size and rank of all the processes in MPI environment by *MPI_Comm_size* and *MPI_Comm_rank* function.
4. Find chunk size of data by equally dividing same amount of data to all processes.
5. Assign dynamic memory to all arrays i.e. masses, position_x, velocity_x etc.
6. Generate random positions, masses and velocities of *n* particles by using **get_rand_num()* and **rand_positions()* functions in Process 0 (also known as Master Process).
7. Use *MPI_Barrier(MPI_COMM_WORLD)* to wait for process 0 to finish tasks assign to it.
8. Once Process 0 finishes all tasks, now it is time to broadcast all arrays of masses and positions to all other processes. To do so, *MPI_Gather* command is used with suitable pointers and buffers.
9. Divide chunk of velocities equally to all processes by *MPI_Scatter* command.
10. Calculate force interaction of chunk particles with all the particles over time period followed by change in velocities and positions.
11. Wait until all processes finish their tasks.
12. Gather all new positions to the root process and write output to output files.
13. Free up allocated memories to all pointers.
14. Finally, Turn off MPI environment by *MPI_Finalize*.

2.2 Visualization

Visualization of state of particles at the beginning and the end of simulation are shown in the figure 2.1 and 2.2 respectively. Figure 2.1 shows initial condition of the particles in simulation box. After $time = 10s$, final positions of the particles is shown in the figure 2.2. For this demonstration $n = 200$ particles, $timestep = 0.1$ and $totaltime = 10s$ have been chosen.

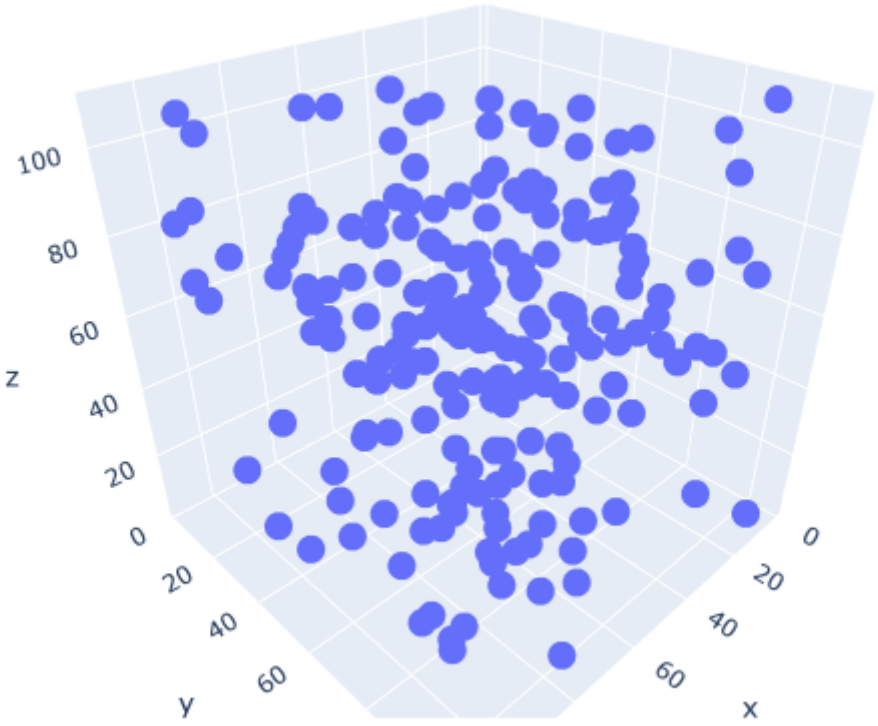

Figure 2.1: Position of Particles at time = 0 s

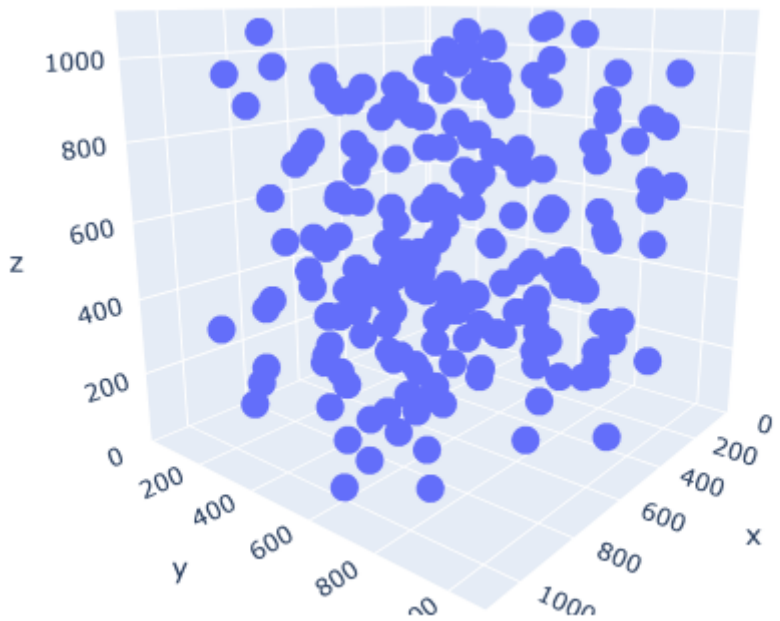

Figure 2.2: Position of Particles at time = 10 s

3 Scalability Tests

With the help of High performance computing (HPC) a large problem can be solved using different processors, which is known as parallel computing. Work is divided among different processors by different MPI commands and processors work simultaneously to reduce computation time. Scalability is widely used to indicate the ability of hardware and software to deliver greater computational power when the amount of resources is increased (2). To perform scalability tests on written MPI Program, we need to disable the portion, where Master process writes output to file.

3.1 Strong Scaling

Strong scaling is defined as variation of time over number of processor for fixed size problem (3). If the amount of time to complete a work unit with 1 processing element is t_1 and the time to complete the same unit of work with N proecssing elements is t_N, the strong scaling efficiency is given as (3):

$$t_1/(N * t_N) * 100 \tag{1}$$

The following table 3.1 shows the strong scaling for 100 particles, with step size of $0.1s$ for total time of $10s$. Figure 3.1 also justifies that with increasing number of processors, time required is decreasing.

No. of Processors	Time Taken (s)	Strong Scaling (%)
1	0.186	100
2	0.0889	104.61
4	0.04513	103.035
5	0.038134	97.55
8	0.0269	86.21
10	0.0828	22.45

Table 3.1: Strong Scaling

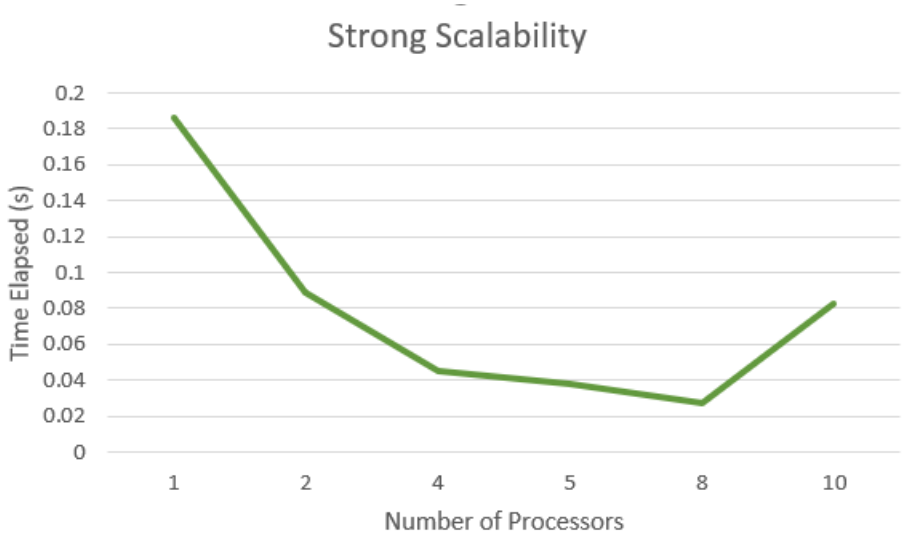

Figure 3.1: No.of Processors Vs Time elapsed Graph

3.2 Weak Scaling

Weak scaling is defined as how solution time varies with the number of processors for a fixed problem size per processor. In this case workload assigned to each processor will be the same. Linear scaling is achieved if the run time stays constant while the workload is increased in direct proportion to the number of processors (3). If the amount of time to complete a work unit with 1 processing element is t_1, and the amount of time to complete N of the same work units with N processing elements is t_N, the weak scaling efficiency (as a percentage of linear) is given as (3):

$$(t_1/t_N) * 100 \tag{2}$$

As shown in table 3.2 and figure 3.2 , it can be seen that up to 8 processors scaling follows linear path, but for 10 processors it is somewhat lacking to meet the estimate.

No. of Processors	Time taken	No. of Particles
1	0.00423	20
2	0.00676	40
4	0.01104	80
5	0.01382	100
8	0.01954	160
10	0.0828	200

Table 3.2: Weak Scaling

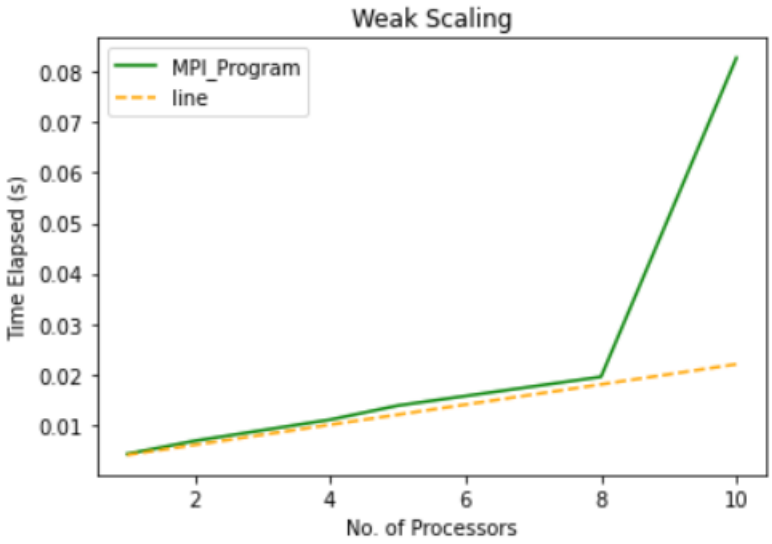

Figure 3.2: No.of Processors Vs Time elapsed (weak scaling)

Bibliography

[1] https://de.wikipedia.org/wiki/Newtonsches$_{G}$*ravitationsgesetz*.

[2] Xin Li. Scalability: strong and weak scaling. https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/: :text=Strong

[3] https://www.sharcnet.ca/help/index.php/Measuring$_{P}$*arallel*$_{S}$*caling*$_{P}$*erformance*.

A1 Appendix

A1.1 MPI Program

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "mpi.h"
#include<assert.h>

#define Nparticles 500
#define Dimension 3
#define delta_t 0.1
#define Nsteps 100

// Function Prototype
double *get_rand_num(int num_elements, double* any_array);

double *rand_positions(int num_elements, double* any_array);

int main(int argc, char **argv)
{
    int size;                       /* Size of communicator*/
    int rank;                       /* Rank of processes*/
    double start_time;              /* start time*/
    double end_time;                /* end time */
    int chunk_size;                 /* chunk size for each process*/

    double *masses = NULL;          /* Pointer for Masses*/

    double *position_x = NULL;      /* Pointer for Positions*/
    double *position_y = NULL;
    double *position_z = NULL;

    double *velocity_x = NULL;      /* Pointer for Velocities*/
    double *velocity_y = NULL;
    double *velocity_z = NULL;

    double *sub_velocity_x = NULL;  /* Pointer for Portion of
       Velocities for each processes*/
    double *sub_velocity_y = NULL;
    double *sub_velocity_z = NULL;

    double acceleration_x[chunk_size];  /* Array for accelerations*/
    double acceleration_y[chunk_size];
    double acceleration_z[chunk_size];
```

```
double *new_position_x = NULL;  /* Pointer for new positions*/
double *new_position_y = NULL;
double *new_position_z = NULL;

double *final_position_x = NULL;    /* Pointer for final position
   storage*/
double *final_position_y = NULL;
double *final_position_z = NULL;

/* Initialize MPI Environment */
MPI_Init(&argc, &argv);

/* Get size of Processes in MPI Environment*/
MPI_Comm_size(MPI_COMM_WORLD, &size);

/* Get rank of each processes */
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

chunk_size = Nparticles / size;
masses = (double *) malloc(sizeof(double) * Nparticles);
assert(masses != NULL);
position_x = (double *) malloc(sizeof(double) * Nparticles);
assert(position_x != NULL);
position_y = (double *) malloc(sizeof(double) * Nparticles);
assert(position_y != NULL);
position_z = (double *) malloc(sizeof(double) * Nparticles);
assert(position_z != NULL);
sub_velocity_x = (double*) malloc(sizeof(double)*chunk_size);
assert(sub_velocity_x != NULL);
sub_velocity_y = (double*) malloc(sizeof(double)*chunk_size);
assert(sub_velocity_y != NULL);
sub_velocity_z = (double*) malloc(sizeof(double)*chunk_size);
assert(sub_velocity_z != NULL);

new_position_x = (double*) malloc(sizeof(double)* chunk_size);
assert(new_position_x != NULL);
new_position_y = (double*) malloc(sizeof(double)* chunk_size);
assert(new_position_y != NULL);
new_position_z = (double*) malloc(sizeof(double)* chunk_size);
assert(new_position_z != NULL);


if (rank == 0)
{
    velocity_x = (double *) malloc(sizeof(double) * Nparticles);
    assert(velocity_x != NULL);
    velocity_y = (double *) malloc(sizeof(double) * Nparticles);
    assert(velocity_y != NULL);
    velocity_z = (double *) malloc(sizeof(double) * Nparticles);
    assert(velocity_z != NULL);

    // Get Random Masses of particles
    masses = get_rand_num(Nparticles, masses);
    position_x = rand_positions(Nparticles, position_x);
    position_y = rand_positions(Nparticles, position_y);
    position_z = rand_positions(Nparticles, position_z);
    velocity_x = rand_positions(Nparticles, velocity_x);
```

```
        velocity_y = rand_positions(Nparticles, velocity_y);
        velocity_z = rand_positions(Nparticles, velocity_z);

        final_position_x = (double *) malloc(sizeof(double) *
            Nparticles);
        assert(final_position_x != NULL);
        final_position_y = (double *) malloc(sizeof(double) *
            Nparticles);
        assert(final_position_y != NULL);
        final_position_z = (double *) malloc(sizeof(double) *
            Nparticles);
        assert(final_position_z != NULL);
    }
    start_time = MPI_Wtime();
    /* MPI_Bcast(void* data, int count, MPI_Datatype datatype,
                int root, MPI_Comm communicator)*/
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(masses, Nparticles, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(position_x, Nparticles, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(position_y, Nparticles, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(position_z, Nparticles, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* MPI Scatter to send all arrays to other processes*/
    /* MPI_Scatter(void* send_data, int send_count, MPI_Datatype
        send_datatype,
         void* recv_data, int recv_count, MPI_Datatype recv_datatype,
            int root, MPI_Comm communicator) */
    MPI_Scatter(velocity_x, chunk_size, MPI_DOUBLE, sub_velocity_x,
        chunk_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(velocity_y, chunk_size, MPI_DOUBLE, sub_velocity_y,
        chunk_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(velocity_z, chunk_size, MPI_DOUBLE, sub_velocity_z,
        chunk_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // MPI_Barrier(MPI_COMM_WORLD);
    for(int i=1; i<=Nsteps; i++)
    {
        for(int j=0; j<chunk_size; j++)
        {
            for(int m=0; m<chunk_size; m++)
            {
                acceleration_x[m] = 0.0;
                acceleration_y[m] = 0.0;
                acceleration_z[m] = 0.0;
            }
            int check_particle = rank*chunk_size + j;
            for(int k=0; k<Nparticles; k++)
            {
                double delta_x, delta_y, delta_z;
                double temp, invr, invr3, force;
                double eps;
                if (k != check_particle)
                {
                    delta_x = position_x[check_particle] - position_x[k
                        ];
                    delta_y = position_y[check_particle] - position_y[k
                        ];
                    delta_z = position_z[check_particle] - position_z[k
```

```
                    ];
                temp = sqrt(delta_x*delta_x + delta_y*delta_y +
                    delta_z*delta_z + eps);
                invr = 1.0 / temp;
                invr3 = invr*invr*invr;
                force = masses[k] * invr3;
                acceleration_x[k] += force*delta_x;
                acceleration_y[k] += force*delta_y;
                acceleration_z[k] += force*delta_z;
            }
        }
        new_position_x[j] = position_x[check_particle] + delta_t*
            sub_velocity_x[j] + 0.5*delta_t*delta_t*acceleration_x[j
            ];
        new_position_y[j] = position_y[check_particle] + delta_t*
            sub_velocity_y[j] + 0.5*delta_t*delta_t*acceleration_y[j
            ];
        new_position_z[j] = position_z[check_particle] + delta_t*
            sub_velocity_z[j] + 0.5*delta_t*delta_t*acceleration_z[j
            ];
        sub_velocity_x[j] += delta_t*acceleration_x[j];
        sub_velocity_y[j] += delta_t*acceleration_y[j];
        sub_velocity_z[j] += delta_t*acceleration_z[j];
        position_x[rank*chunk_size + j] = new_position_x[j];
        position_y[rank*chunk_size + j] = new_position_y[j];
        position_z[rank*chunk_size + j] = new_position_z[j];
    }
    MPI_Barrier(MPI_COMM_WORLD);
    /* MPI_Gather(void* send_data, int send_count, MPI_Datatype
        send_datatype,
            void* recv_data, int recv_count, MPI_Datatype
                recv_datatype,
            int root, MPI_Comm communicator)*/
    MPI_Gather(new_position_x, chunk_size, MPI_DOUBLE,
        final_position_x, chunk_size, MPI_DOUBLE, 0,MPI_COMM_WORLD);
    MPI_Gather(new_position_y, chunk_size, MPI_DOUBLE,
        final_position_y, chunk_size, MPI_DOUBLE, 0,MPI_COMM_WORLD);
    MPI_Gather(new_position_z, chunk_size, MPI_DOUBLE,
        final_position_z, chunk_size, MPI_DOUBLE, 0,MPI_COMM_WORLD);
    if (rank == 0)
    {
        if (i == 1)
        {
            FILE *output_file = fopen ( "initial_position.csv","w+"
                );
            fprintf(output_file, "X,Y,Z \n");
            for(int i=0; i<Nparticles; i++)
            {
                fprintf(output_file, "%lf, %lf, %lf \n",
                    final_position_x[i], final_position_y[i],
                    final_position_z[i]);
            }
            fclose(output_file);

        }

        if (i == Nsteps)
        {
```

```
                FILE *output_file = fopen ( "final_position.csv","w+");
                fprintf(output_file, "X,Y,Z \n");
                for(int i=0; i<Nparticles; i++)
                {
                    fprintf(output_file, "%lf, %lf, %lf \n",
                       final_position_x[i], final_position_y[i],
                       final_position_z[i]);
                }
                fclose(output_file);

            }
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    end_time = MPI_Wtime();
    if (rank == 0)
    {
        printf("********************* Stats of Simulation
           ********************\n");
        printf("\t Number of MPI Processees = %d \t \n", size);
        printf("\t Number of Particles = %d \t \n", Nparticles);
        printf("\t Total Time = %lf \t \n", Nsteps*delta_t);
        printf("\t Time step = %lf \t \n", delta_t);
        printf("\t Total Steps = %d\t \n", Nsteps);
        printf("\t Time Elapsed = %e \t \n", end_time-start_time);
        printf("
           ************************************************************\
           n");

    }
    free(masses);
    free(position_x);
    free(position_y);
    free(position_z);

    free(sub_velocity_x);
    free(sub_velocity_y);
    free(sub_velocity_z);

    free(new_position_x);
    free(new_position_y);
    free(new_position_z);

    if (rank == 0)
    {
        free(velocity_x);
        free(velocity_y);
        free(velocity_z);
        free(final_position_x);
        free(final_position_y);
        free(final_position_z);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

```
/* Generate Random number of Masses according to number of particles*/
double *get_rand_num(int num_elements, double* any_array)
{
    int i;
    for(i=0; i<num_elements; i++)
    {
        any_array[i] = rand() % 10 + 1;
    }
    return any_array;
}

/* Generate random number of Positions */
double *rand_positions(int num_elements, double* any_array)
{
    int i;
    for(i=0; i<num_elements; i++)
    {
        any_array[i] = rand() % 100;
    }
    return any_array;
}
```