HPP_A3

Group20

Group Member: Linjing Shen, Houwan Liu

galsim.c

1. The Problem

This task require us to implement a program that simulates the gravitational interaction between N particles in a galaxy using Newton's law of gravitation. The simulation involves updating the positions and velocities of the particles over a certain number of timesteps. Based on the given mathematical function and instruction, we wrote the initial version code, and use optimization techniques to do further improvement.

2. The Solution

2.1 Initial Version Code

The following is the initial version of the code, without any use both compiler optimisation flags and our own code changes.

galsim_initial.c

2.2 Time Consuming

```
"Initial version time consuming"
time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1

"PC1"
real    0m4.760s
user    0m4.760s
sys     0m0.000s

"PC2"
real      0m22,711s
user      0m22,703s
sys     0m0,000s
```

2.3 Accuracy

```
"Initial version accuracy"

N = 10
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.000000000000

N = 3000
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```

3. Performance

3.1 Summary

The performance of the code improved significantly after optimizations. By avoiding repeated computations, eliminating the use of `pow()`, loop skipping, merging functions, and using compiler optimizations, we saw a decrease in computation time.

Among the different optimization levels on each PC, the lower real time values indicate better performance.

On PC1, the -O1 and -O4 optimizations have the lowest real time values both at 2.114s. On PC2, the -O3 optimization level has the lowest real time at 10.146s.

3.2 CPU Model

```
"PC1 CPU model"
lish6557@LAPTOP-B927FN7A:~/HPP/Assignment3$ lscpu
Architecture:            x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:         39 bits physical, 48 bits virtual
Byte Order:            Little Endian
CPU(s):                  20
On-line CPU(s) list:   0-19
Vendor ID:               GenuineIntel
Model name:            12th Gen Intel(R) Core(TM) i7-12700H
CPU family:          6
Model:               154
Thread(s) per core:  2
Core(s) per socket:  10
Socket(s):           1

"PC1 compiler version"
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
"PC2 CPU model"
holi6891@arrhenius:~/Assignment3$ lscpu
Architecture:            x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:         40 bits physical, 48 bits virtual
Byte Order:            Little Endian
CPU(s):                  16
On-line CPU(s) list:   0-15
Vendor ID:               GenuineIntel
Model name:            Intel(R) Xeon(R) CPU           E5520  @ 2.27GHz
CPU family:          6
Model:               26
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           2

"PC2 compiler version"
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

3.3 Optimization

3.3.1 Optimization within Code

When performing in-code optimizations, start with the initial version of the code and don't use any compiler optimization flags.

```
"makefile used in 3.3.1"

galsim: galsim.c
gcc -o galsim galsim.c -lm
```

```
clean:
rm -f galsim
```

1. Avoiding repeat computations

In the `compute_forces` function, the positions and masses of each particle are calculated in advance to avoid repeated retrieval in the inner loop.

In the `update_particle` function, the parts of repeated calculations `delta_t_mass` are calculated in advance to avoid repeated calculations.

galsim_opt1.c

```
"Time consuming avoiding repeat computation"
time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1

        before                    after
"PC1"
real    0m4.760s          real    0m4.682s
user    0m4.760s          user    0m4.681s
sys     0m0.000s          sys     0m0.000s

"PC2"
real      0m22,830s         real      0m22,113s
user      0m22,818s         user      0m21,969s
sys     0m0,000s          sys       0m0,132s

"Accuracy"
N = 10
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.000000000000

N = 3000
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```

2.Avoiding use pow()

Avoiding use `pow()` in the `compute_forces` function because it's slower and less efficient than basic arithmetic operations like multiplication. `pow()` does a lot of work under the hood to handle different cases, which can slow things down. Plus, every function call adds extra overhead, making codes run slower. Also, `pow()` is all about precision, which might be overkill for what we need, slowing things down even more.

galsim_opt2.c

```
"Time consuming avoiding use pow()"
time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1

        before                    after
"PC1"
real    0m4.682s                    real    0m1.262s
user    0m4.681s                    user    0m1.261s
sys     0m0.000s                    sys     0m0.000s

"PC2"
real      0m22,113s         real      0m8,121s
user      0m21,969s             user      0m8,116s
```

```
sys     0m0,132s          sys       0m0,000s

"Accuracy"
N = 10
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.000000000000

N = 3000
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```

3. Cache optimisation and Merge functions

In order to ensure contiguous memory access, instead of using struct to contain the parameters of all the particles, we use six arrays to accommodate them separately.

Merging `compute_forces()`, `update_particles()` and `simulate()` into one function can improve performance by reducing the overhead of function calls. Every function call involves a certain amount of overhead, including the time it takes to jump to the function's code, push arguments onto the stack, and then pop the results off the stack.

galsim_opt3.c

```
"Time consuming after using arrays instead of struct"
time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1


        before                    after
"PC1"
real     0m1.262s                real   0m1.185s
user     0m1.261s                user   0m1.185s
sys      0m0.000s                sys    0m0.000s

"PC1"
real     0m8,121s                real   0m7,746s
user     0m8,116s                user   0m7,742s
sys      0m0,000s                sys    0m0,001s

"Accuracy"
N = 10
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.000000000000

N = 3000
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```

4. Loop avoidance

In the `compute_forces` function, optimizes this by starting the inner loop (j) from i+1, effectively avoiding redundant force calculations for pairs of particles, which makes changes based on the code of galsim_opt3.c.

galsim_opt4.c

```
"Time consuming with loop unrolling"
time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1
```

```
        before                    after
"PC1"
real    0m1.185s         real    0m0.674s
user    0m1.185s                 user    0m0.674s
sys     0m0.000s                 sys     0m0.000s

"PC1"
real    0m7,746s         real    0m4,421s
user    0m7,742s                 user    0m4,413s
sys     0m0.001s                 sys     0m0,005s

"Accuracy"
N = 10
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.000000000000

N = 3000
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```

5. Loop Vectorization

To make further optimisations, firstly we try to do vectorization for the main operation loop in `simulate()` function, but it seems like using compiler to vectorize the loop is infeasible due to the structure of our code.

In this optimisation case, we try to separate out as many loop-independent variables as possible, removing as many obstacles as possible in the loop of the simulate() function, in order to facilitate vectorised optimisation using the compiler.

Obstacles to Vectorization in loops:
• Uncountable loop (e.g. while - uncountable stop)
• Non-contiguous memory access
• Data dependencies (vectorization changes the order of operations)
• External functions inside a loop (unless these are inlined)
• Conditionals sentences (if/else, break, continue) which cannot be removed by the compiler (cannot be rewritten using bitwise operations)

galsim_vec.c

However, we found that in the compute forces loop in `simulate` function, `sqrt` will cause memory pollution, which is an unavoidable obstacle to auto-vectorization. So we seek for other methods to calculate the square root, such as Newton's iterative method. If we use this function, it will bring obvious `while loops`, which is an another obstacle of vectorization. Then we tried finite number of iterations to achieve approximate square root calculations, the result is much longer execution time and larger error.

```
"Time consuming after using arrays instead of struct"
time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1


 Loop avoidance         Loop Vectorization
"PC1"
real    0m0.674s                 real    0m1.180s
user    0m0.674s                 user    0m1.170s
sys     0m0.000s                 sys     0m0.010s

"PC1"
real    0m4,421s                 real    0m9,075s
```

```
user    0m4,413s                 user    0m9,070s
sys     0m0,005s                 sys     0m0.001s


"Accuracy"
N = 10
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.033221329630

N = 3000
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.036517519472
```

6. Cache blocking

We tried block sizes of 32, 64, and 128, and found that application cache blocking did not improve in that case, and in most cases slightly degraded performance.

galsim_block.c

```
"Time consuming with loop unrolling"
time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1




 before blocking          after blocking
"PC1"
real    0m1.185s         real    0m1.231s
user    0m1.185s                 user    0m1.231s
sys     0m0.000s                 sys     0m0.000s

"PC1"
real    0m7,746s         real    0m7,769s
user    0m7,742s                 user    0m7,759s
sys     0m0,001s                 sys     0m0,005s


"Accuracy"
N = 10
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.000000000000

N = 3000
fileName1 = '../result.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```

3.3.2 Compiler Optimisation

The following is the final version of the code, which has undergone our own code changes.

```
/*Final Version*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include <string.h>

#define EPSILON 0.001

void read_initial_configuration(const char *filename, double *position_x, double *position_y, do
uble *mass, double *velocity_x, double *velocity_y, double *brightness, int N) {
    // Implement function to read initial configuration from file

    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        fprintf(stderr, "Error: Could not open file %s\n", filename);
        exit(1);
    }

    for(int i = 0; i < N; i++){
        size_t particles_read = fread(&position_x[i], sizeof(double), 1, file);
        particles_read += fread(&position_y[i], sizeof(double), 1, file);
        particles_read += fread(&mass[i], sizeof(double), 1, file);
        particles_read += fread(&velocity_x[i], sizeof(double), 1, file);
        particles_read += fread(&velocity_y[i], sizeof(double), 1, file);
        particles_read += fread(&brightness[i], sizeof(double), 1, file);
    }

    fclose(file);
}


void simulate(double *position_x, double *position_y, double *mass, double *velocity_x, double *
velocity_y, double *brightness, double *force_x, double *force_y, int N, int nsteps, double delt
a_t) {
    double G = 100.0/N;
    for (int step = 0; step < nsteps; step++) {
        for (int i = 0; i < N; i++){
            force_x[i] = 0.0;
            force_y[i] = 0.0;
        }
        for (int i = 0; i < N; i++) {
            double position_x_i = position_x[i];
            double position_y_i = position_y[i];
            double mass_i = mass[i];

            double forcex = force_x[i];
            double forcey = force_y[i];

            for (int j = i + 1; j < N; j++) {
                double dx = position_x_i - position_x[j];
                double dy = position_y_i - position_y[j];
                double distance_squared = sqrt(dx * dx + dy * dy) + EPSILON;
                double distance_cubed = distance_squared * distance_squared * distance_squared;
                double force_magnitude = -G * mass_i * mass[j] / distance_cubed;
                forcex += force_magnitude * dx;
                forcey += force_magnitude * dy;

                force_x[j] -= force_magnitude * dx;
                force_y[j] -= force_magnitude * dy;
            }
            force_x[i] = forcex;
            force_y[i] = forcey;
        }
        for(int i = 0; i < N; i++){
            double delta_t_mass = delta_t / mass[i];
            velocity_x[i] += delta_t_mass * force_x[i];
            velocity_y[i] += delta_t_mass * force_y[i];
            position_x[i] += delta_t * velocity_x[i];
            position_y[i] += delta_t * velocity_y[i];
        }
    }
}

void write_results(const char *filename, double *position_x, double *position_y, double *mass, d
ouble *velocity_x, double *velocity_y, double *brightness, int N) {
    // Implement function to write results to file
    FILE *file = fopen(filename, "wb");

    for(int i = 0; i < N; i++){
        size_t particles_read = fwrite(&position_x[i], sizeof(double), 1, file);
        particles_read += fwrite(&position_y[i], sizeof(double), 1, file);
        particles_read += fwrite(&mass[i], sizeof(double), 1, file);
        particles_read += fwrite(&velocity_x[i], sizeof(double), 1, file);
        particles_read += fwrite(&velocity_y[i], sizeof(double), 1, file);
        particles_read += fwrite(&brightness[i], sizeof(double), 1, file);
    }

    fclose(file);
}

int main(int argc, char *argv[]) {
    // Parse command line arguments
    if (argc != 6) {
        printf("Usage: %s N filename nsteps delta_t graphics\n", argv[0]);
        return 1;
    }
    int N = atoi(argv[1]);
    char *filename = argv[2];
    int nsteps = atoi(argv[3]);
    double delta_t = atof(argv[4]);
    int graphics = atoi(argv[5]);

    double position_x[N], position_y[N], mass[N], velocity_x[N], velocity_y[N], brightness[N], f
orce_x[N], force_y[N];

    read_initial_configuration(filename, position_x, position_y, mass, velocity_x, velocity_y, b
rightness, N);
    simulate(position_x, position_y, mass, velocity_x, velocity_y, brightness, force_x, force_y,
N, nsteps, delta_t);
    write_results("result.gal", position_x, position_y, mass, velocity_x, velocity_y, brightnes
s, N);

    return 0;
}
```

Then trying the ability of different compiler optimisation methods to optimise the final code.

```
"Using final version code"
time ./galsim 3000 ellipse_N_03000.gal 100 0.00001 1

"Without using compiler optmisation"
gcc -o galsim galsim.c -lm

"Using -O1"
gcc -O1 -o galsim galsim.c -lm
```

```
"Using -O2"
gcc -O2 -o galsim galsim.c -lm

"Using -O3"
gcc -O3 -o galsim galsim.c -lm

"Using -O4"
gcc -O4 -o galsim galsim.c -lm

"Using -Os"
gcc -Os -o galsim galsim.c -lm

"Using -Ofast"
gcc -Ofast -o galsim galsim.c -lm
```

- `O1`: Basic optimization, attempting to reduce code size and execution time, including constant folding and removal of useless code.
- `O2`: Further performance improvement, including dead code elimination, function inlining, common subexpression elimination, etc.
- `O3`: Higher-level optimizations, including all `O2` level optimizations, as well as loop unrolling, vectorization, more function inlining, etc. It may increase code size and compilation time.
- `O4`: This is not a standard option for the GCC compiler, generally, it is considered equivalent to `O3`.
- `Os`: Optimize code size without significantly affecting performance, suitable for embedded systems, etc.
- `Ofast`: Enable all `O3` level optimizations and further enable some optimizations that may change program behavior. Allow the compiler to assume no floating-point exceptions, which may enable more optimizations.

"PC1"	Initial	-O1	-O2	-O3	-O4	-Os	-Ofast
real	0m3.246s	0m1.107s	0m1.145s	0m1.129s	0m1.123s	0m1.586s	0m1.161s
user	0m3.245s	0m1.107s	0m1.144s	0m1.129s	0m1.123s	0m1.586s	0m1.161s
sys	0m0.000s	0m0.000s	0m0.001s	0m0.000s	0m0.000s	0m0.000s	0m0.000s

"PC2"	Initial	-O1	-O2	-O3	-O4	-Os	-Ofast
real	0m20,806s	0m10,207s	0m10,538s	0m10,146s	0m10,156s	0m10,541s	0m10,157s
user	0m20,795s	0m10,196s	0m10,528s	0m10,140s	0m10,149s	0m10,530s	0m10,150s
sys	0m0,004s	0m0,004s	0m0,004s	0m0,000s	0m0,001s	0m0,004s	0m0,001s

The difference between the vectorized `O3` and the non-vectorized `O1` and `O2` is not significant, which may be influenced by the following factors:

1. **External functions inside the loop**: In the `simulate` function, the part that updates the particle position and speed includes a call to an external function, namely the `sqrt()` function. Although this is a standard mathematical library function, it will hinder vectorization unless it is inlined.
2. **Conditional statements**: There are conditional statements in the loop `if (j != i)`, which limit the possibility of vectorization, because it is difficult to predict the result of conditional statements during vectorization. In addition, `break` and `continue` statements may also prevent the loop from being vectorized.
3. **Data dependency**: In the inner loop, there are data dependencies in the calculation of forces between particles, i.e., the results of subsequent calculations depend on the results of previous calculations. This dependency may complicate vectorization because vectorization will change the order of operations.
4. **Memory access pattern**: The calculation of forces between particles may lead to non-continuous memory access patterns, which may affect the effect of vectorization.

```
"Final makefile"
"-O3/O4 seems better compared with other optimisation methods"
"All the compiler optimisation methods keep the code accuracy 0.0000"

galsim: galsim.c
gcc -O4 -o galsim galsim.c -lm
```

```
clean:
rm -f galsim
```

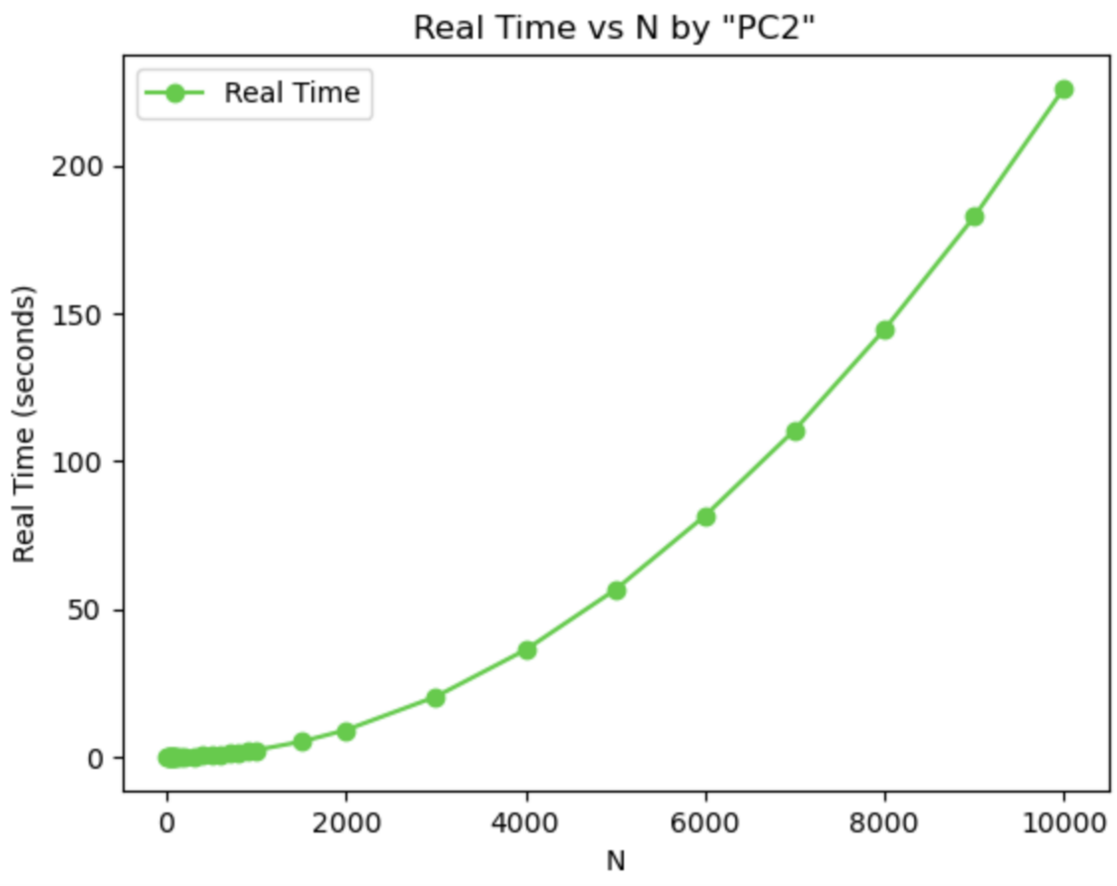
4. Discussion

4.1. Time complexity

How the computational time depends on N?

```
"Final version code with -O4 compiler optimisation"

"PC1"
time ./galsim 3000 ellipse_N_03000.gal 100 0.00001 1
real    0m2.158s
user    0m2.158s
sys     0m0.000s

time ./galsim 1500 ellipse_N_01500.gal 100 0.00001 1
real    0m0.539s
user    0m0.539s
sys     0m0.000s

"PC2"
time ./galsim 3000 ellipse_N_03000.gal 200 0.00001 1
real      0m20,299s
user      0m20,291s
sys     0m0,001s

time ./galsim 1500 ellipse_N_03000.gal 200 0.00001 1

real      0m5,089s
user      0m5,081s
sys     0m0,005s
```

The real time to run 3000 particles is always around 4 times that of running 1500 particles, which proves that the time complexity of the code is `O(N^2)`.



4.2 Accuracy

```
"Final version accuracy"

N = 10
fileName1 = '../input_data/ellipse_N_00010.gal'
fileName2 = '../ref_output_data/ellipse_N_00010_after200steps.gal'
pos_maxdiff =   0.000000000000

N = 3000
fileName1 = '../input_data/ellipse_N_03000.gal'
fileName2 = '../ref_output_data/ellipse_N_03000_after100steps.gal'
pos_maxdiff =   0.000000000000
```

We notice that the maximum difference in position (`pos_maxdiff`) is equal to 0 for both simulations with 10 and 3000 particles, indicating that the simulation is quite accurate.

4.3 Limitations and Further Improvements

Despite the significant performance improvements, there are still some limitations and potential areas for further optimization. For example, the current code does not take advantage of parallel computing, which could further speed up the calculations. Also, more sophisticated algorithms for handling particle interactions, such as the Barnes-Hut algorithm, could be used to improve performance for large particle numbers.