ASSIGNMENT 3 & 4:
THE GRAVITATIONAL N-BODY PROBLEM

High Performance Programming 1TD062, 10 hp

Erik Bjerned & Mattias Persson

2023-03-03

UPPSALA
UNIVERSITET

Contents

1 The Problem

The gravitational N-body problem is based around solving how N particles affect the positions of all other particles with respect to their own gravity. Basically simulating a galaxy-type system where all particles affect every other particle. A particle in this case exists in a 2-dimensional domain, and consists of 6 data-points, x and y coordinates, velocity in x and y, mass and brightness. The simulation consists of updating each particle and its variables each timestep according to some governing equations, these equations resembles the normal gravitational equations but with a few tweaks to increase stability and some changes to constants to fit inside the set domain ($[0,1] \times [0,1]$). The following equations were the starting point for the simulation.

The force equation

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

$$\mathbf{a}_i = \frac{\mathbf{F}_i}{m_i} \tag{2}$$

The symplectic Euler method

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n + \Delta t \mathbf{a}_i \tag{3}$$

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1} \tag{4}$$

2 The Solution

2.1 Theory

At first a simple algorithm was implemented, looping through N^2 calculations to calculate the force and further the acceleration all particles. When starting to optimize the program, this became the part that was the most expensive. So as a step in optimizing this task, a smarter approach to calculating the acceleration had to be taken. The following theory gives the background for this implementation and it still grows as a $\mathcal{O}(N^2)$ algorithm but more specifically the acceleration calculations are reduced down to $\mathcal{O}(N^2/2)$

$$\mathbf{F}_i = -Gm_i \sum_{j=0,\ j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij},\ \mathbf{a}_i = \frac{\mathbf{F}_i}{m_i} \tag{5}$$

$$\implies \mathbf{a}_i = -G \sum_{j=0,\ j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij} \tag{6}$$

We can use the fact that

$$\frac{\mathbf{r}_{ij}}{(r_{ij}+\epsilon_0)^3} = -\frac{\mathbf{r}_{ji}}{(r_{ji}+\epsilon_0)^3} \tag{7}$$

to reduce the number of calculations. This is easily proven but left out for since we are being brief. Because of this symmetry it is possible to reduce the iterations of the acceleration calculations by half. This is especially important as r_{ij} is expensive to compute due to the square root involved in its computation.

2.2 Implementation

The program can be divided into a few parts which will be explained in the coming sections. Timing of the implementation was done with the function `get_wall_seconds()` from a previous lab [1].

1. Initialization
2. Solve loop
    2.1. Calculation of acceleration
    2.2. Solve for new positions
3. Write output

2.2.1 Initialization

A part from reading input variables and assigning them, this part is mainly reading the input data into memory and allocating memory for the acceleration calculation. The input data is read into an array of size $6N$ as there are six attributes per particle and in the programme each attribute will be accessed by offsetting the index of the particle with a integer between zero and five. This could also be done with an array of a defined struct, each containing data for a particle. But in this case where the particles have few attributes which all are doubles it is not necessary to use a struct for code clarity. Whether it is a array of structs or an array of doubles, it probably will not affect performance noticeably at run time. The compiler most likely reduces both alternatives to offsets of memory addresses, compilation time might be affected it will not be a great difference and is not to be taken in to consideration. There may be some difference in runtime when reading input and writing output as it can be written straight from an array or read straight into an array, however this is a quite small part of the total time, especially when calculating for large N.

2.2.2 Solve loop

The solve loop is divided into two sections because of the modification to the algorithm in section 2.1 causes the need to calculate the acceleration of all particles before starting to solve for the positions. The performance of this is is discussed in a later section.

2.2.3 Calculation of acceleration

For the actual implementation of calculating the acceleration this is translated into this pseudo code

```
for i less than N:
    for j less than i:
        acceleration[i] = -G*m_j*vec_r_ij/(r_ij+eps)^3
        acceleration[j] = G*m_i*vec_r_ij/(r_ij+eps)^3
```

Instead of having only two lines for assigning values of the acceleration, it is in reality four as the acceleration has two components, x and y. In this algorithm we see that since we calculate forces between two objects, we do not need to check every individual pair of particles. The contribution of gravitational force is equal, and the acceleration change is proportional to the mass of the two particles.

2.2.4 Solve for new positions

For solving the new positions, the given *Symplectic Euler method* is implemented in a separate `for`-loop. Because of the given method, it is possible to simply update the relevant values in the data array sequentially without the use of temporary variables. Firstly the velocity of each particle is updated using said sympletic euler method, and then the actual positions are updated accordingly.

2.2.5 Write output

Since the data in this implementation is stored in a single, continuous array it can be written with a single `fwrite`-call.

3 Performance & Discussion

The program was performance tested on the university's Linux machine `vitsippa`. Every combination of settings are tested three times and the test with shortest runtime is recorded in the tables below

3.1 Code optimization

Within the code some standard serial optimization techniques are used. For instance loop invariants. Some of the values used in the inner loop (j-loop), namely the mass and position of the outer loop (i-loop) variables. Setting these parameters reduces instructions within the j-loop by three. Some expensive algorithms, for instance calculating $(r_{ij}+\epsilon)^3$ using the function `pow` is avoided, since just multiplying it three times with itself is faster. Usage of division is also done sparingly, since it is more computationally expensive than multiplying, since $G/(r_{ij}+\epsilon_0)^3$ is used calculating the acceleration in x and y directions for both the i and j variable, computing this beforehand and multiplying is quicker than dividing with $(r_{ij}+\epsilon_0)^3$ 4 times. The most important optimization is probably the one discussed in the theory section, using the symmetry of force between particles, this because of the reduction in `sqrt` calls.

Some other techniques using keywords were tried to see if they would give any improvement. Usage of the keyword `const` on every input and non-changing variable resulted in no improvement, this probably stems from the small amount of calls to the main computing function. Using `inline` for the functions also has no impact, most probably because that the compiler already does this optimization.

Table 1: Code optimization.Tested on `ellipse_N_03000.gal` for 100 timesteps with $\Delta t = 10^{-5}$ with flag `-O2 -ffast-math -march=native`. Time measured over entire program execution.

Keyword	Time
<none>	5.8400
`const`	5.8403
`restrict`	5.8417
`inline`	5.9740

Half inner loop, reduces iterations and expensive square roots

Table 2: Code optimization. Tested on `ellipse_N_03000.gal` for 100 timesteps with $\Delta t = 10^{-5}$ with flag `-O0`. Time measured over entire program execution.

Keyword	Time
<none>	32.1721
`const`	32.1685
`restrict`	32.2298
`inline`	32.1710

Another optimization that was done was to improve how the array containing the acceleration data was reset at each iteration. This is necessary to do since the calculation is only

additive and it also done to reset any residual values in the allocated memory when performing the initially calculation. The naive approach is looping through the array and setting it to zero. This is often not considered as slow, but as it is needed to be done each time step there is a need to find a better solution. As seen previously in a lab `memset` was used as a faster alternative [2]. It does in fact result in a performance increase in the order of 1 ms per iteration with 3000 particles. There is however an even faster alternative, which is including this in the calculation loop, which is the alternative used and this further improved performance a reduction of 0.7 ms per iteration with 3000 particles.

3.2 Optimization flags

Using different optimization flags has large impact on performance. In the table below a comparison of different standard optimization flags are made using a data set of 3000 particles and 100 time steps. Using `-O0` is the default, and using the most basic `-O1` makes a huge difference in run time. The flags that gave the best result was `-O2 -ffast-math -march=native` which can be seen in bold in the table below. It is interesting that there is a time increase when switching `-O2` for `-O3` but it is difficult to say what is causing this as the compiler and the flags are complex.

Table 3: Optimization flags. Tested on `ellipse_N_03000.gal` for 100 timesteps with $\Delta t = 10^{-5}$. No special keywords. Time measured over entire program execution.

Flags	**Time**
`-O0`	32.1900
`-O1`	8.3986
`-O2`	7.1928
`-O3`	7.2655
`-Ofast`	7.2233
`-Os`	11.0113
`-O2 -ffast-math`	6.7691
`-O2 -ffast-math -march=native`	**5.8400**
`-O3 -ffast-math`	7.2250
`-O3 -ffast-math -march=native`	5.9034
`-Ofast -march=native`	5.8980

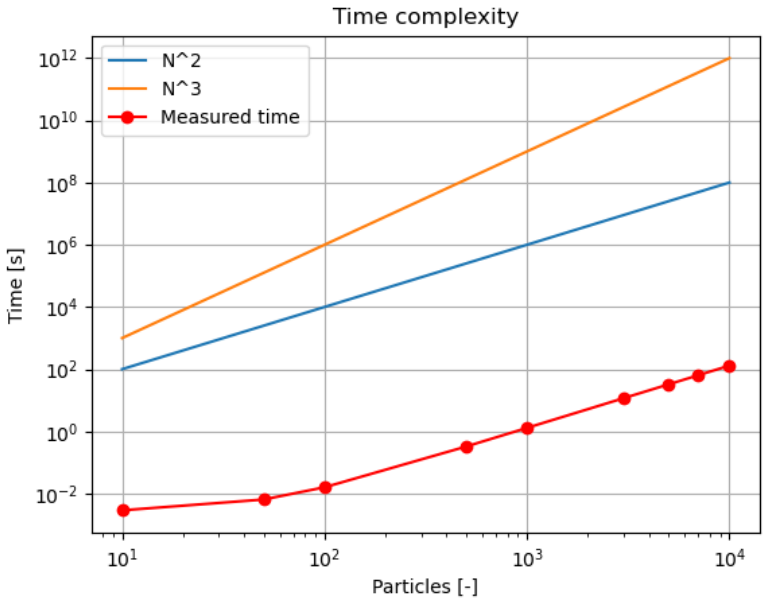

Figure 1: Time complexity of solution. Done with 200 timesteps with $\Delta t = 10^{-5}$

The complexity of the implementation is $\mathcal{O}(N^2)$ as expected. This can be seen in figure 1. For small $N < 100$ the complexity is a bit less than $\mathcal{O}(N^2)$, as the computation time is above to what is expected. In this case, it is more likely that the other parts of the program, such as reading input data or writing the output is the biggest contributor to computation time. The two mentioned depend on N but most certainly has some overhead time cost when opening and closing files which are independent of N causing this differing result. The computation time for the acceleration and solving is in this case so small that other parts of the program becomes the bottleneck.

4 Parallelization

After solving the problem using a serial solution, an implementation using some parallel methods were tried in efforts to optimize even further. The serial program was thoroughly timed to profile where the most expensive part of the program was located. As expected, this was the acceleration calculation which took at least in the order of 100 times longer to execute than any other section. Because of this, implementing the parallelization was only done for this calculation as it would not have made a significant difference on the other part of the code.

Using several cores can increase performance many times over. Two different parallelization-packages (`pthread` and `OpenMP`) are used and two different parallel solutions are implemented. These two packages provide different utility and difficulties while implementing, and comparing these two will hopefully give one very optimized version. The performance gain will be compared with the serial solution and the amount of cores used. The solver can be parallelized in the way that forces between particles are calculated simultaneously. Each time step is incremented serially.

4.1 OpenMP

Implementing a parallelized version of the serial solution above using the `OpenMP` package is quite trivial. Since no single thread is explicitly made, and no joining of threads are needed.

The serial code is simply updated within the acceleration function to include a parallelization section. This section divides the heavy work made between the user set amount of threads. This is made easy by changing the outer (i-loop) from

```
for i less than N, (increment = 1):
```

to

```
for i = (threadnumber) less than N (increment = number of threads):
```

Note that this is pseudo-code. Since the solver updating the position for each particle is not that computationally heavy, parallelising it results in no performance gain.

As of the current implementation each thread sums to a private array to then, after the parallel section, sum up to the complete array of acceleration data. This is fast since there are no synchronizations during the calculations, only after, but it requires a specific allocated array for each thread. Because of the simplified algorithm implemented in Assignment 3 causing the acceleration calculation contributing to every particle, independent of how the data is distributed between the threads. Thus it is not possible to split the original $2N$ sized acceleration array in to $2N/n_{threads}$ sized array, one for each thread. Instead each thread has to be allocated a $2N$ sized array, causing the memory usage to grow linearly with the number of threads.

An alternative that was experimented with, was to set the insertion of the acceleration data into the array to `#pragma omp critical` to avoid this. However, whilst keeping the memory usage constant, the performance increase of using several threads is significantly decreased. This is not surprising as the critical section is computed in the order of $N^2/2$ per time step, causing many interruptions between the threads.

4.2 Pthreads

Implementing with `Pthreads` is a bit more complicated than `OpenMP` since specifying each thread separately and then joining them is necessary. This involves creating a few separate functions and a structure containing the data that is used. The main function calculating the acceleration is the same as in the `OpenMP` case (at least algorithmically, some syntax changes are made). However a function to create each thread and the join them is made separately.

As with OpenMP, an alternative to using private arrays per each thread was tried. This was done with a mutex lock around the insertion into the array of acceleration data. This was tested as a way to reduce memory usage, but not surprisingly most of the parallelization was lost as the threads had to wait many times per time step.

4.3 Performance gain from parallelization

The performance gain is compared against the serial version deemed most optimal, e.g the version using the flags `-O2 -ffast-math -march=native`. However the parallized version is tested with 5000 particles as requested in the assignment. The tests were run on `tussilago.it.uu.se` to avoid other student using cores as it is less popular than `vitsippa.it.uu.se` although they have the same CPUs.

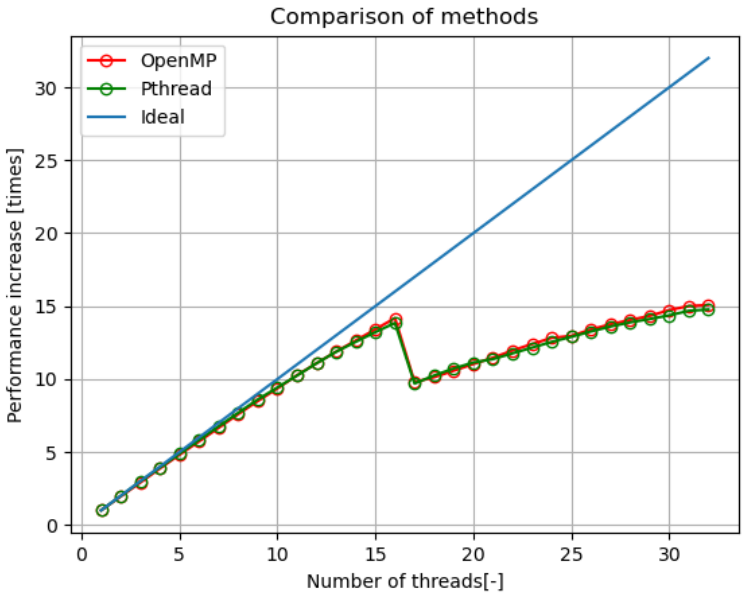

Figure 2: Parallellization test. Performed with `-O2 -ffast-math -march=native` and on 5000 particles for 100 timesteps

4.4 Discussion parallelization

The results of the parallelisation performed as expected growing quite linearly with the number of threads, as can be seen in figure 2. There are two significant observations that can be seen in this figure. The most significant is the drop in performance increase at 16 threads, however, this is expected. The `tussilago` machine consists of two 16 core CPUs with 16 threads each, making a total of 32 threads split over two sockets. Needing to use threads of different CPUs causes a drop in performance because the cache is not as local memory as before and causes threads to access cache memory from two different CPUs thus not being as fast as the local cache. A part from the drop at 16 threads the performance drops somewhat compared to the ideal line which is as expected, as with more threads the amount of overhead increases.

Using the second CPU on the machine is beneficial to the performance, as the best result is achieved by using 32 threads. But the increase between 32 and 16 threads is about equivalent to having a 17 core CPU instead. This could probably be reduced as the current serial optimization is not optimal and as the data is stored in a single large array and possibly slows down the connection between the two CPUs.

References

[1] Course material. "`get_wall_seconds()`". In: *Lab 3* Task 9 (2023), p. 7.

[2] Course material. "SERIAL OPTIMIZATION PART 1, REDUCING INSTRUCTIONS". In: *Lab 5* Task 4 (2023), p. 7.

Appendix

A A3: Serially optimized galsim.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

double* read_input(char* path, int n);
void acceleration(double* data_array, double* acc_array, int n);
void solver(double* data_array, double* acc_array, double dt, int n);
void write_output(double* data_array, double* acc_array, int n, char* path);
double get_wall_seconds();



int main(int argc, char* argv[]){
  double time = get_wall_seconds();
  double start = time;

  if(argc != 6){
    printf("Not enough arguments. Input arguments: 'N', 'filename', 'nsteps','delta_t', 'graphics'\n");
    return -1;
  }


  int n_particles = atoi(argv[1]);
  char* input_path = argv[2];
  int n_steps = atoi(argv[3]);
  double dt = atof(argv[4]);
//  char graphics = *argv[5];
  double* acc_arr = (double*)malloc(n_particles*2*sizeof(double));

  printf("Initialized in %lf s\n", get_wall_seconds()-time);
  time = get_wall_seconds();

  double* data_arr = read_input(input_path, n_particles);

  if (data_arr == NULL) return -1;

  printf("Input read in %lf s\n", get_wall_seconds()-time);
  time = get_wall_seconds();

  for(int i = 0; i < n_steps; i++){
    acceleration(data_arr, acc_arr, n_particles);
    solver(data_arr, acc_arr, dt, n_particles);
  }

  printf("\n");
  printf("Problem solved in %lf s, %lf per timestep with %i particles\n", get_wall_seconds()-time, (get_wall_seconds()-time)/n_steps, n_particles);
  time = get_wall_seconds();

  write_output(data_arr, acc_arr, n_particles, "result.gal");
  printf("Output written in %lf s\n", get_wall_seconds()-time);
```

```
52
53    free(data_arr);
54    free(acc_arr);
55    printf("Program finished in %lf s. Exiting...\n", get_wall_seconds()-start)...
      ;
56
57    return 0;
58  }
59
60  void acceleration(double* data_array, double* acc_array, int n){
61    double x_i, y_i, x_j, y_j,m_i, m_j, r_ij;
62    double G = 100/(double)n;
63    double eps = 0.001;
64    int i, j;
65    double denom;
66
67    for(i = 0; i < 6*n; i +=6){
68      // Assign current values to mitigate carry over from previous timestep. ...
      Faster than memset
69      acc_array[2*(i/6)] = 0;
70      acc_array[2*(i/6)+1] = 0;
71
72      // Fetch j-loop invariant values
73      x_i = data_array[i];
74      y_i = data_array[i+1];
75      m_i = data_array[i+2];
76
77      for(j = 0; j < i; j+=6){
78        x_j = data_array[j];
79        y_j = data_array[j+1];
80        m_j = data_array[j+2];
81
82        r_ij = sqrt((x_i-x_j)*(x_i-x_j)+(y_i-y_j)*(y_i-y_j));
83        denom = G/((r_ij+eps)*(r_ij+eps)*(r_ij+eps));
84
85        // Update x and y components of the acceleration
86        acc_array[2*(i/6)] += -m_j*(x_i-x_j)*denom;
87        acc_array[2*(i/6)+1] += -m_j*(y_i-y_j)*denom;
88        acc_array[2*(j/6)] += m_i*(x_i-x_j)*denom;
89        acc_array[2*(j/6)+1] += m_i*(y_i-y_j)*denom;
90      }
91    }
92  }
93
94
95  void solver(double* data_array, double* acc_array, double dt, int n){
96    int i;
97    // Symplectic Euler for the n particles
98    for(i = 0; i < 6*n; i+=6){
99      // Velocity update
100     data_array[i+3] = data_array[i+3] + dt*acc_array[2*(i/6)];
101     data_array[i+4] = data_array[i+4] + dt*acc_array[2*(i/6)+1];
102
103     // Position update
104     data_array[i] = data_array[i] + dt*data_array[i+3];
105     data_array[i+1] = data_array[i+1] + dt*data_array[i+4];
106   }
107 }
108
109
```

```
110
111 double* read_input(char* path, int n){
112   FILE* file = fopen(path, "rb");
113   if (file == NULL){
114     printf("ERROR: File path not valid\n");
115     return NULL;
116   }
117   // Each particle has 6 attributes --> [pos_x, pos_y, m, v_x, v_y, b, ...]
118
119   double* input_data = (double*)malloc(6*n*sizeof(double));
120   if(fread(input_data, sizeof(double), 6*n, file) != 6*n){
121     printf("ERROR: Mismatch between specified number of particles and read ...
      number of particles\n");
122     return NULL;
123   }
124   fclose(file);
125   return input_data;
126
127 }
128
129 void write_output(double* data_array, double* acc_array, int n, char* path){
130   FILE* file = fopen(path, "wb");
131
132   if(fwrite(data_array, sizeof(double) ,6*n, file) != 6*n){
133     printf("ERROR: Mismatch between specified number of particles and written...
      number of particles\n");
134     return;
135   }
136   fclose(file);
137 }
138
139 double get_wall_seconds() {
140   struct timeval tv;
141   gettimeofday(&tv, NULL);
142   double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
143   return seconds;
144 }
```

B A4: Parallelized with Pthread

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <sys/time.h>
5  #include <pthread.h>
6  #include <string.h>
7
8  double* read_input(char* path, int n);
9  void* acceleration(void* arg);
10 void acceleration_joiner(double* data_array, double* acc_array, int n, int ...
       n_threads);
11 void solver(double* data_array, double* acc_array, double dt, int n);
12 void write_output(double* data_array, double* acc_array, int n, char* path);
13 double get_wall_seconds();
14
15 typedef struct data_struct{
16   double* data_array;
17   double* acc_array;
```

```
18   int s;
19   int n;
20   int offset;
21 } data_t;
22
23
24 int main(int argc, char* argv[]){
25   double time = get_wall_seconds();
26   double start = time;
27
28   if(argc != 7){
29     printf("Not enough arguments. Input arguments: 'N', 'filename', 'nsteps',...
       'delta_t', 'graphics', 'n_threads'\n");
30     return -1;
31   }
32
33
34   int n_particles = atoi(argv[1]);
35   char* input_path = argv[2];
36   int n_steps = atoi(argv[3]);
37   double dt = atof(argv[4]);
38 //  char graphics = *argv[5];
39   int n_threads = atoi(argv[6]);
40   double* acc_arr = (double*)malloc(n_particles*2*sizeof(double));
41
42 //  printf("Initialized in %lf s\n", get_wall_seconds()-time);
43   time = get_wall_seconds();
44
45   double* data_arr = read_input(input_path, n_particles);
46
47   if (data_arr == NULL) return -1;
48
49 //  printf("Input read in %lf s\n", get_wall_seconds()-time);
50   time = get_wall_seconds();
51
52   for(int i = 0; i < n_steps; i++){
53     acceleration_joiner(data_arr, acc_arr, n_particles, n_threads);
54 //    printf("Acceleration computed in %lf s\n", get_wall_seconds()-time);
55 //    time = get_wall_seconds();
56     solver(data_arr, acc_arr, dt, n_particles);
57 //    printf("Solved in %lf s\n", get_wall_seconds()-time);
58   }
59
60 //  printf("\n");
61 //  printf("Problem solved in %lf s, %lf per timestep with %i particles\n", ...
     get_wall_seconds()-time, (get_wall_seconds()-time)/n_steps, n_particles);
62   time = get_wall_seconds();
63
64   write_output(data_arr, acc_arr, n_particles, "result.gal");
65 //  printf("Output written in %lf s\n", get_wall_seconds()-time);
66
67   free(data_arr);
68   free(acc_arr);
69 //  printf("Program finished in %lf s. Exiting...\n", get_wall_seconds()-...
     start);
70   printf("%i\t%lf\n", n_threads, get_wall_seconds()-start);
71   return 0;
72 }
73
74 void acceleration_joiner(double* data_array, double* acc_array, int n, int ...
```

```
    n_threads){
  int i,j;
  pthread_t threads[n_threads];
  double* sub_acc_array[n_threads];
  data_t* argument_data = (data_t*)malloc(sizeof(data_t)*n_threads);

  for(i = 0; i < n_threads; i++){
    sub_acc_array[i] = (double*) malloc(sizeof(double)*n*2);
    memset(sub_acc_array[i], 0, 2*n*8);

    argument_data[i].data_array = data_array;
    argument_data[i].acc_array = sub_acc_array[i];

    argument_data[i].s = n_threads;
    argument_data[i].n = n;
    argument_data[i].offset = i;
    pthread_create(&threads[i], NULL, acceleration, (void*) &argument_data[i...
    ]);
  }

  for(i=0; i < n_threads; i++){
    pthread_join(threads[i], NULL);
//    printf("Thread %i acceleretion:\n", i);

    for(j = 0; j < n*2; j++){
//      printf("%lf\n", argument_data[i].acc_array[j]);
      if (i == 0){
        acc_array[j] = argument_data[i].acc_array[j];
      }else{
        acc_array[j] += argument_data[i].acc_array[j];

      }

    }
    free(argument_data[i].acc_array);
  }
  free(argument_data);


}

void* acceleration(void* arg){
  data_t data_struct = *(data_t*) arg;
  double* data_array = (double*)data_struct.data_array;
  double* acc_array = (double*)data_struct.acc_array;
  int s = (int)data_struct.s;
  int n = (int)data_struct.n;
  int offset = (int)data_struct.offset;
  double x_i, y_i, x_j, y_j,m_i, m_j, r_ij;
  double G = 100/(double)n;
  double eps = 0.001;
  int i, j;
  double denom;

  for(i = 6*offset; i < 6*n; i +=6*s){

//    printf("Acceleration at particle %i by thread %i \n", i/6, offset);

    // Fetch j-loop invariant values
    x_i = data_array[i];
```

```
133     y_i = data_array[i+1];
134     m_i = data_array[i+2];
135 
136     for(j = 0; j < i; j+=6){
137 //      printf("\tAcceleration at particle %i by thread %i from particle %i\n...
    ", i/6,offset, j/6);
138       x_j = data_array[j];
139       y_j = data_array[j+1];
140       m_j = data_array[j+2];
141 
142       r_ij = sqrt((x_i-x_j)*(x_i-x_j)+(y_i-y_j)*(y_i-y_j));
143       denom = G/((r_ij+eps)*(r_ij+eps)*(r_ij+eps));
144 
145       // Update x and y components of the acceleration
146       acc_array[2*(i/6)] += -m_j*(x_i-x_j)*denom;
147       acc_array[2*(i/6)+1] += -m_j*(y_i-y_j)*denom;
148       acc_array[2*(j/6)] += m_i*(x_i-x_j)*denom;
149       acc_array[2*(j/6)+1] += m_i*(y_i-y_j)*denom;
150 //      printf("%lf\t%lf\n", acc_array[2*(i/6)],acc_array[2*(i/6)+1]);
151     }
152   }
153 }
154 
155 
156 void solver(double* data_array, double* acc_array, double dt, int n){
157   int i;
158   // Symplectic Euler for the n particles
159   for(i = 0; i < 6*n; i+=6){
160     // Velocity update
161     data_array[i+3] = data_array[i+3] + dt*acc_array[2*(i/6)];
162     data_array[i+4] = data_array[i+4] + dt*acc_array[2*(i/6)+1];
163 
164     // Position update
165     data_array[i] = data_array[i] + dt*data_array[i+3];
166     data_array[i+1] = data_array[i+1] + dt*data_array[i+4];
167   }
168 }
169 
170 
171 
172 double* read_input(char* path, int n){
173   FILE* file = fopen(path, "rb");
174   if (file == NULL){
175     printf("ERROR: File path not valid\n");
176     return NULL;
177   }
178   // Each particle has 6 attributes --> [pos_x, pos_y, m, v_x, v_y, b, ...]
179 
180   double* input_data = (double*)malloc(6*n*sizeof(double));
181   if(fread(input_data, sizeof(double), 6*n, file) != 6*n){
182     printf("ERROR: Mismatch between specified number of particles and read ...
     number of particles\n");
183     return NULL;
184   }
185   fclose(file);
186   return input_data;
187 
188 }
189 
190 void write_output(double* data_array, double* acc_array, int n, char* path){
```

```
  FILE* file = fopen(path, "wb");

  if(fwrite(data_array, sizeof(double) ,6*n, file) != 6*n){
    printf("ERROR: Mismatch between specified number of particles and written...
     number of particles\n");
    return;
  }
  fclose(file);
//  for(int i = 0; i < 2*n; i++)
//    printf("%lf\n", acc_array[i]);
}

double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}
```

C A4: Parallelized with OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <string.h>
#include <omp.h>

double* read_input(char* path, int n);
void acceleration(double* data_array, double* acc_array, int n, int n_threads...
    , double** sub_acc_arrays);
void solver(double* data_array, double* acc_array, double dt, int n);
void write_output(double* data_array, double* acc_array, int n, char* path);
double get_wall_seconds();



int main(int argc, char* argv[]){
  double time = get_wall_seconds();
  double start = time;

  if(argc != 7){
    printf("Not enough arguments. Input arguments: 'N', 'filename', 'nsteps',...
     'delta_t', 'graphics', 'n_threads'\n");
    return -1;
  }


  int n_particles = atoi(argv[1]);
  char* input_path = argv[2];
  int n_steps = atoi(argv[3]);
  double dt = atof(argv[4]);
//  char graphics = *argv[5];
  int n_threads = atoi(argv[6]);
  double* acc_arr = (double*)malloc(n_particles*2*sizeof(double));

//  printf("Initialized in %lf s\n", get_wall_seconds()-time);
  time = get_wall_seconds();
```

```
36
37    double* data_arr = read_input(input_path, n_particles);
38
39    if (data_arr == NULL) return -1;
40
41  //  printf("Input read in %lf s\n", get_wall_seconds()-time);
42    time = get_wall_seconds();
43    double** sub_acc_arrays = (double**)malloc(sizeof(double*)*n_threads);
44    for(int i = 0; i < n_threads; i++){
45      double* sub_acc = (double*) malloc(sizeof(double)*n_particles*2);
46      sub_acc_arrays[i] = sub_acc;
47    }
48  //  printf("Threads setup in %lf s\n", get_wall_seconds()-time);
49    time = get_wall_seconds();
50    for(int i = 0; i < n_steps; i++){
51  //    double atime = get_wall_seconds();
52      acceleration(data_arr, acc_arr, n_particles, n_threads, sub_acc_arrays);
53  //    printf("Acceleration calculated in %lf s\n", get_wall_seconds()-atime);
54  //    atime = get_wall_seconds();
55      solver(data_arr, acc_arr, dt, n_particles);
56  //    printf("Solution calculated in %lf s\n", get_wall_seconds()-atime);
57
58    }
59
60  //  printf("\n");
61  //  printf("Problem solved in %lf s, %lf per timestep with %i particles\n", ...
      get_wall_seconds()-time, (get_wall_seconds()-time)/n_steps, n_particles);
62    time = get_wall_seconds();
63
64    write_output(data_arr, acc_arr, n_particles, "result.gal");
65  //  printf("Output written in %lf s\n", get_wall_seconds()-time);
66
67    free(data_arr);
68    free(acc_arr);
69    for(int i = 0; i < n_threads; i++){
70      free(sub_acc_arrays[i]);
71    }
72    free(sub_acc_arrays);
73  //  printf("Program finished in %lf s. Exiting...\n", get_wall_seconds()-...
      start);
74    printf("%i\t%lf\n",n_threads, get_wall_seconds()-start);
75    return 0;
76  }
77
78  void acceleration(double* data_array, double* acc_array, int n, int n_threads...
      , double** sub_acc_arrays){
79
80    #pragma omp parallel num_threads(n_threads)
81    {
82    double x_i, y_i, x_j, y_j,m_i, m_j, r_ij;
83    double G = 100/(double)n;
84    double eps = 0.001;
85    int i, j;
86    double denom;
87
88    double* sub_acc = sub_acc_arrays[omp_get_thread_num()];
89    memset(sub_acc, 0, 2*8*n);
90
91    for(i = 6*omp_get_thread_num(); i < 6*n; i +=6*omp_get_num_threads()){
92      // Assign current values to mitigate carry over from previous timestep. ...
```

```
   Faster than memset
    acc_array[2*(i/6)] = 0;
    acc_array[2*(i/6)+1] = 0;

    // Fetch j-loop invariant values
    x_i = data_array[i];
    y_i = data_array[i+1];
    m_i = data_array[i+2];

    for(j = 0; j < i; j+=6){
      x_j = data_array[j];
      y_j = data_array[j+1];
      m_j = data_array[j+2];

      r_ij = sqrt((x_i-x_j)*(x_i-x_j)+(y_i-y_j)*(y_i-y_j));
      denom = G/((r_ij+eps)*(r_ij+eps)*(r_ij+eps));

      // Update x and y components of the acceleration


      sub_acc[2*(i/6)] += -m_j*(x_i-x_j)*denom;
      sub_acc[2*(i/6)+1] += -m_j*(y_i-y_j)*denom;
      sub_acc[2*(j/6)] += m_i*(x_i-x_j)*denom;
      sub_acc[2*(j/6)+1] += m_i*(y_i-y_j)*denom;

    }
  }
  }
  for(int i = 0; i < n_threads; i++){
    for(int j = 0; j<2*n; j++){
      if (i==0){
        acc_array[j] = sub_acc_arrays[i][j];
      } else{
        acc_array[j] += sub_acc_arrays[i][j];
      }
    }
  }
}


void solver(double* data_array, double* acc_array, double dt, int n){
  int i;
  // Symplectic Euler for the n particles
  for(i = 0; i < 6*n; i+=6){
    // Velocity update
    data_array[i+3] = data_array[i+3] + dt*acc_array[2*(i/6)];
    data_array[i+4] = data_array[i+4] + dt*acc_array[2*(i/6)+1];

    // Position update
    data_array[i] = data_array[i] + dt*data_array[i+3];
    data_array[i+1] = data_array[i+1] + dt*data_array[i+4];
  }
}



double* read_input(char* path, int n){
  FILE* file = fopen(path, "rb");
  if (file == NULL){
    printf("ERROR: File path not valid\n");
```

```
152     return NULL;
153   }
154   // Each particle has 6 attributes --> [pos_x, pos_y, m, v_x, v_y, b, ...]
155
156   double* input_data = (double*)malloc(6*n*sizeof(double));
157   if(fread(input_data, sizeof(double), 6*n, file) != 6*n){
158     printf("ERROR: Mismatch between specified number of particles and read ...
      number of particles\n");
159     return NULL;
160   }
161   fclose(file);
162   return input_data;
163
164 }
165
166 void write_output(double* data_array, double* acc_array, int n, char* path){
167   FILE* file = fopen(path, "wb");
168
169   if(fwrite(data_array, sizeof(double) ,6*n, file) != 6*n){
170     printf("ERROR: Mismatch between specified number of particles and written...
      number of particles\n");
171     return;
172   }
173   fclose(file);
174 }
175
176 double get_wall_seconds() {
177   struct timeval tv;
178   gettimeofday(&tv, NULL);
179   double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
180   return seconds;
181 }
```