Uppsala University

High Performance Programming

1TD062

Assignment 3 Report: The 2D Gravitational N-Body Problem

Students:
Anders Köhler
Benjamin Bucknall
Julián Ramón Marrades Furquet
Omar Ghulam Ahmed Malik

Teachers:
Jarmo Rantakokko
Benjamin Weber

May 6, 2021

Contents

Introduction

At the heart of classical mechanics lies the following question: given the net force acting on a particular body, how would that body's position change over time? The answer to this question is given by Newton's 2nd law:

$$\mathbf{F}_{net} = m\mathbf{a} = m\frac{d^2\mathbf{r}}{dt^2}, \tag{1}$$

where $\mathbf{F}_{net}$ is the net force acting on the body, m is its mass, $\mathbf{a}$ is its acceleration, and $\mathbf{r}$ is its position as a function of time. In other words, the body's trajectory in space can be determined once the net force acting on it is known. This trajectory can easily be found for a system that contains one body, and 2-body systems can be treated as "pseudo 1-body" systems by determining the two bodies' *relative* trajectories to each other. However, for 3-body systems and beyond, such as in a galaxy of of numerous stars and/or planets, it is impossible to solve Eq. (1) exactly. It should then be solved numerically, and there are many standard algorithms available that can do so. However, these algorithms must be implemented on a computer program, and these implementations should be as efficient as possible so that they can scale well with the *size* of the system — that is, the number of bodies in the system itself.

The objective of this assignment was to solve Eq. (1) for a two-dimensional N-body galaxy where the only force present is the gravitational force between every two bodies; this includes planet-planet, planet-star, or star-star pairs. The force that body i feels due to the influence of body j is denoted as $\mathbf{f}_{ij}$, and it takes the following form:

$$\mathbf{f}_{ij} = -G\frac{m_i m_j}{r_{ij}^3}\mathbf{r}_{ij}, \tag{2}$$

where G is the gravitational constant, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the separation vector between body i and body j, and $r_{ij} = \sqrt{\mathbf{r}_{ij} \cdot \mathbf{r}_{ij}}$ is the distance between them. In 2D, $\mathbf{r}_i$ has two components: x and y. The net force on body i is simply the sum of the forces due to all other j bodies:

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

where the *net* subscript has been dropped from $\mathbf{F}$ for the sake of convenience in the rest of this report. For numerical stability, a small positive number

ϵ_0 was added to the distances r_{ij} to prevent divisions by near-zero values in Eq. (2). (This effectively caps the maximum gravitational force acting on the bodies.) The total force acting on body i then becomes:

$$\mathbf{F}_i = -Gm_i \sum_{j=0,\ j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij}. \tag{4}$$

Once the net force on body i, $\mathbf{F}_i$, has been found, the next step was to numerically solve Eq. (1) for its position r_i at a given point in time. This was done by using the *symplectic* Euler method to calculate its velocity first, followed by its position:

$$\begin{aligned} \mathbf{a}_i^n = \frac{\mathbf{F}_i^n}{m_i} &= -G \sum_{j=0,\ j\neq i}^{N-1} \frac{m_j}{(r_{ij}^n+\epsilon_0)^3}\mathbf{r}_{ij}^n \\ \mathbf{u}_i^{n+1} &= \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n \\ \mathbf{r}_i^{n+1} &= \mathbf{r}_i^n + \Delta t \mathbf{u}_i^{n+1}, \end{aligned} \tag{5}$$

where $\mathbf{u}_i$ is the velocity of body i, Δt is the size of each timestep from one point in time to the next, and the n and $n+1$ superscripts denote the given and next timesteps, respectively (they are not exponents). Eq. (5) can be repeated for as many timesteps as needed, thereby simulating the galaxy of interest.

Methodology

The simulation was implemented using the C programming language and was done so in three ways: the straightforward way which has no optimizations, henceforth referred to as the base implementation, an optimized implementation, and a vectorized implementation. This was done to see the importance of optimizing code so that its execution time scales feasibly with larger inputs. Some optimizations could be automatically handled by the GNU Compiler Collection (GCC), but even further optimizations were done manually.

For every simulation, a .gal file is provided accounting for initial attribute values of N bodies. The bodies are defined by the following attributes: x-axis position, y-axis position, mass, x-axis velocity, y-axis velocity and brightness. For the purpose of this project, the latter is not used at all, but read and stored with the others nonetheless. This input data is stored in

a matrix, as are the acceleration values along the x- and y-axes computed during runtime. The matrices are assumed to represent bodies along the rows and attributes along the columns, and are operated accordingly.

Specifications

The simulation was done on the AMD Opteron^{TM} 16-core, 64-bit processor running at 2.6 GHz. The machine uses the Scientific Linux release 6.10 (Carbon) operating system. Regarding the C programming language, the C99 standard was used with the GCC compiler release 4.4.7. The optimized and vectorized C codes use the following additional flags to allow the compiler to make further optimizations:

- `-O3`: Tells the compiler to perform aggressive optimizations when building the assembly code from the C code.
- `-march=native`: The compiler should produce a code that matches the current CPU's processor.

The usage of the following flags was avoided:

- `-funroll-loops`: Automatically unrolling loops can be dangerous, especially if the workload within them is heavy. Since this was the case, loop unrolling was manually added at a later stage, when the code was vectorized.
- `-ffast-math`: Since the positions of the planets are spread close to the origin, it is not safe to risk a substantial impact on accuracy if values get flushed to 0.

The performance of the code was tested by calculating the difference between the wall times before and after the galaxy simulation part of the code. (The rest of the code simply allocates or deallocates the memory needed in the simulation and so was unimportant when considering its performance.)

Manual optimizations

The straightforward way to numerically solve Eq. (5) is given in Algorithm (1). The implementation of this algorithm — the base implementation — has much room for manual optimizations. As a result, several of these were introduced to produce the final, optimized version of the code. The optimizations that were used are listed below:

- **`restrict` and `const`:** Given that two function input pointers are guaranteed not to point to the same memory location (no pointer aliasing), the compiler may find a more efficient order of execution. The keyword `restrict` assures the compiler that this is safe to do. Likewise, knowing that the value of an input parameter will not be changed during the function call, the compiler may operate faster without having to check for such changes. The keyword `const` instructs the compiler to treat a parameter as a constant. Both keywords were added to the relevant arguments of the two functions `getNextState` and `eucDist` in `integrator.c`.

- **Temporary sums:** As specified by Eq. (5), the acceleration of body i at time n, $\mathbf{a}_i^n$, may be calculated directly from the sum of influences of other bodies by simply multiplying it with the negative gravitational constant. As the net force, $\mathbf{F}_i^n$, is of no interest other than computing the acceleration, the base implementation directly stores the sums in the `acc` matrix within the inner for loop in `getNextState`, which is then multiplied by $-G$ outside the loop.

  The base implementation writes each term of a sum to the `acc` matrix for every iteration of the inner for loop. The matrix index is dependent only on the outer loop, however, so this entails that the memory location of the same matrix element has to be reaccessed many times. In the optimized code, this is avoided by storing all (negative) terms of a sum in the temporary variables `sumX` and `sumY`. The full sums are then multiplied with G after the for loop and added to `acc`.

- **Distance lookup table:** Note that $r_{ij} = r_{ji}$, which implies that $(r_{ij} + \epsilon_0)^3 = (r_{ji} + \epsilon_0)^3$. So half of the computations can be saved by calculating half of the distances and storing these cubed terms in an $N \times N$ matrix to be used in a later stage. In particular, an iteration from $i = 0$ to $N - 1$ is implemented and the inner loop is separated into two sections. This has the added benefit of eliminating the if statement that checks whether $i \neq j$ for every iteration.

  Firstly, iterating from $j = i+1$ to $N-1$, the distance values are stored in the corresponding elements of the lookup table. Then, iterating from $j = 0$ to $i-1$, the cubed terms are taken from the upper-triangular part of the matrix by inverting the indexing; r_{ij} now becomes r_{ji}. This is depicted by the solid red arrow in Fig. 1, which takes the values from the dashed red arrow.

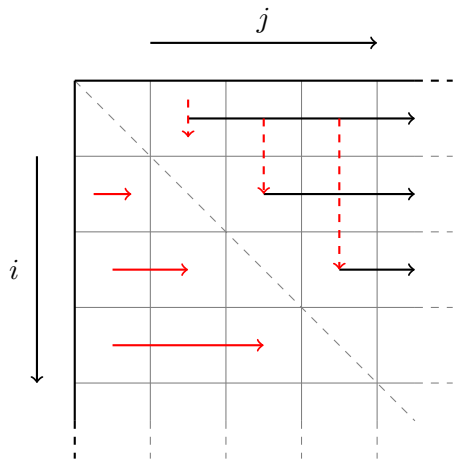

Figure 1: Diagram for distance lookup table. For every row i, the upper-triangular section (black arrows) is processed first, with distance values being computed and stored. The lower-triangular section (red arrows) then retrieves the values from the corresponding symmetric column (dashed red arrows).

- **Column-wise storage:** As the symplectic Euler scheme involves a sequence of calculations which only depend on a few variables contained in the state matrix, it is a waste of computational resources to access every attribute for a body simultaneously. If the matrix is stored row-wise in memory, like any regular implementation of a 2D array, the cache line would be loaded with all data pertaining to a specific body before moving on to the next one. In order to ensure maximally efficient cache line usage, the matrices were reworked to operate with a column-oriented storage scheme, thus allowing for the cache to only contain data explicitly pertinent to the current operations.

- **Matrix as 1D array:** Rather than separately allocating storage space for each of the n columns in a matrix, thus scattering their data across the main memory, the columns may be concatenated into one 1D array of length $n \cdot$ (length of column). With the column data stored completely contiguously in this manner, even less memory accesses will have to be performed as more data will be found in the cache as the result of a previous load.

- **Vector registers:** Further expanding upon the sequential data man-

agement of the two previous steps, many of the operations performed in the symplectic Euler method were able to be vectorized using Intel SSE intrinsics, effectively allowing the data for multiple bodies to be updated simultaneously. In order to maintain the standard of double precision used so far in the implementation, `__m128d` vector registers are used, allowing two double-precision values to be stored in each register.

In order to take full advantage of this capability, the required data structures `state`, `acc`, and `dists` storing the states of the bodies are declared and initialized as arrays aligned to 16 bytes. This allows memory access to be performed more efficiently than if assumed to be unaligned. Furthermore, the `getNextState()` function initializes `__m128d` vector registers which are used throughout the function to temporarily store values such as x- and y-coordinates while performing Euler iteration using SSE intrinsic vector operations. In practice, the vector registers are employed to compute the influence of two distinct bodies on the individual currently under consideration for each pass through lines 10-15 and 18-20 in Algorithm 2.

Finally, it is worth noting that special cases have to be made for instances in which the number of bodies to consider is odd. In this case, vector operations are performed for all possible pairs of bodies, with the influence of the remaining individual being computed through regular scalar operations.

In summary, all above-mentioned improvements to the base implementation yield Algorithm (2), where the input data, accelerations and cubed distances are stored column-wise in 1D arrays. The vectorization is implemented over lines 4-20, handling pairs of j at a time, since two doubles (64 bit) fill a 128-bit SSE3 register.

Experiments

We were asked to perform two experiments:

1. Check that the algorithm scales quadratically with input size $\mathcal{O}(N^2)$ by plotting N^2 against the execution time for a working version of the code.

2. Compare the performance of the three versions of the code for several N, in order to measure the effectiveness of the optimizations.

Algorithm 1 Unoptimized galaxy simulation algorithm

Input: $\{\mathbf{r}_i^t, m_i, \mathbf{u}_i^t : i = 1, 2...N\}$ - positions, masses, and velocities of all N bodies at the tth point in time.
T - number of timesteps in the simulation
Δt - size of each timestep

Output: $\{\mathbf{r}_i^T, \mathbf{u}_i^T : i = 1, 2...N\}$ - positions and velocities of all N bodies at the final, Tth point in time.

Initialization: read $\{\mathbf{r}_i^0, m_i, \mathbf{u}_i^0\}$, the initial positions; masses; and velocities, from the input file in a 2D array.
for $t = 0, 1, ..., T-1$ **do**

  Calculate the accelerations acting on each body at a given point in time. The accelerations are stored in a 2D array.
  for $i = 0, 1, ..., N$ **do**
    $\mathbf{a}_i^t := \mathbf{0}$
    for $j = 0, 1, ..., N$ **do**
      if $j \neq i$ **then**
        $\mathbf{r}_{ij}^t := \mathbf{r}_i^t - \mathbf{r}_j^t$
        $r_{ij}^t := \sqrt{\mathbf{r}_{ij}^t \cdot \mathbf{r}_{ij}^t}$
        $\mathbf{a}_i^t := \mathbf{a}_i^t + \frac{m_j}{(r_{ij}^t + \epsilon_0)^3} \mathbf{r}_{ij}^t$
      end if
    end for
    $\mathbf{a}_i^t := \mathbf{a}_i^t * -G$
  end for

  Update the positions of each body using the calculated accelerations.
  for $i = 0, 1, ..., N$ **do**
    $\mathbf{u}_i^{t+1} := \mathbf{u}_i^t + \Delta t \mathbf{a}_i^t$
    $\mathbf{r}_i^{t+1} := \mathbf{r}_i^t + \Delta t \mathbf{u}_i^{t+1}$
  end for

end for

Algorithm 2 Optimized galaxy simulation algorithm

Input: $\{\mathbf{r}_i^t,\, m_i, \mathbf{u}_i^t : i = 1, 2...N\}$ - positions, masses, and velocities of all N bodies at the tth point in time.
T - number of timesteps in the simulation
Δt - size of each timestep

Output: $\{\mathbf{r}_i^T, \mathbf{u}_i^T : i = 1, 2...N\}$ - positions and velocities of all N bodies at the final, Tth point in time.

Initialization: read $\{\mathbf{r}_i^0,\, m_i, \mathbf{u}_i^0\}$, the initial positions; masses; and velocities, from the input file and store them column-wise in a vector.
for $t = 0, 1, ..., T-1$ **do**

Calculate the accelerations acting on each body at a given point in time. The accelerations matrix is stored column-wise in a vector.
for $i = 0, 1, ..., N$ **do**
Initialize a temporary sum variable that is independent of i.
sum := **0**

Loop over the right of the diagonal. Calculate and store upper-right distances d_{ij}.
for $j = i+1, i+2, ..., N$ **do**
$\mathbf{r}_{ij}^t := \mathbf{r}_i^t - \mathbf{r}_j^t$
$r_{ij}^t := \sqrt{\mathbf{r}_{ij}^t \cdot \mathbf{r}_{ij}^t}$
$d_{ij}^t := (r_{ij}^t + \epsilon_0)^3$
sum := **sum** $- \frac{m_j}{d_{ij}^t}\mathbf{r}_{ij}^t$
end for

Loop over the left of the diagonal. Use upper-right distances (d_{ji}) instead of bottom-left ones (d_{ij}) since they are the same.
for $j = 0, 1, ..., i-1$ **do**
sum := **sum** $- \frac{m_j}{d_{ji}^t}\mathbf{r}_{ij}^t$
end for
$\mathbf{a}_i^t := G * \mathbf{sum}$

end for

Update the positions of each body using the calculated accelerations.
for $i = 0, 1, ..., N$ **do**
$\mathbf{u}_i^{t+1} := \mathbf{u}_i^t + \Delta t \mathbf{a}_i^t$
$\mathbf{r}_i^{t+1} := \mathbf{r}_i^t + \Delta t \mathbf{u}_i^{t+1}$
end for

end for

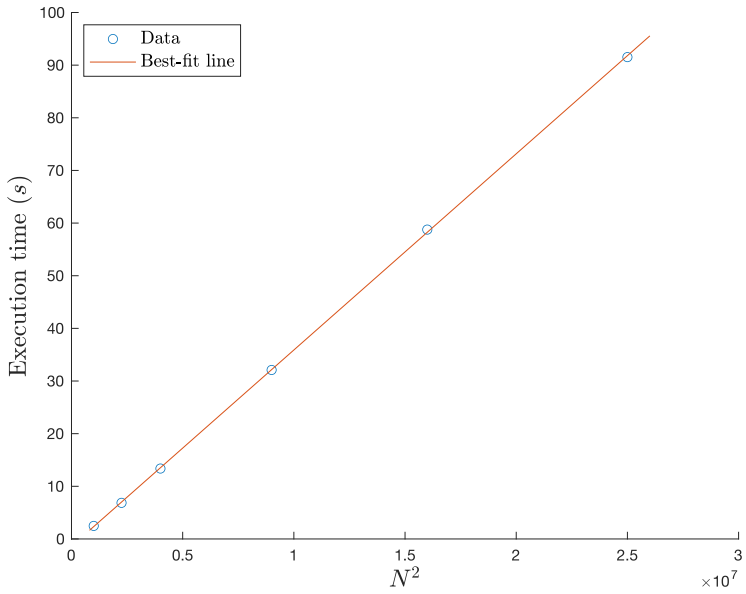

Figure 2: Average execution time for several N of the non-vectorized code without column-wise storage. For each N, 10 time measurements were taken after 200 symplectic Euler steps with $\Delta t = 10^{-5}$. Note that the execution time was plotted against N^2.

Results and Discussion

Quadratic growth $\mathcal{O}(N^2)$

As demonstrated in Fig. 2, the best-fit line passes through all the data-points, indicating an excellent fit and thus a quadratic time growth.

Performance comparison

Fig. 3 shows execution times for the three versions of the code. One can observe that measurements were not taken for $N > 2000$ in the base version. This is because they would blow up and the difference between the other two versions of the code would not be appreciated. Note that there is a major improvement in performance when the code is optimized (without vector registers yet). For example, an execution time of more than 50 seconds for $N = 2000$ becomes smaller than 10 seconds. On the other hand, the

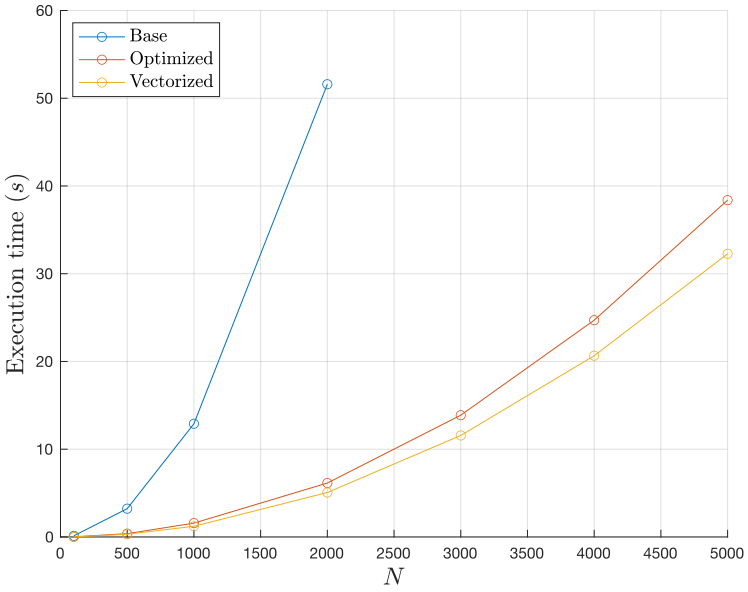

Figure 3: Average execution time for several N of the three versions of the code. For each N, 5 time measurements were taken after 200 timesteps with $\Delta t = 10^{-5}$.

improvements of vectorization only become noticeable for large N, e.g. 10 second difference for $N = 5000$.

Conclusion

In this paper, a symplectic Euler integration scheme to solve the 2D N-body problem was implemented, enhanced, and evaluated. Multiple optimization techniques were discussed and included in the algorithm, such as 1D matrix storage, avoiding conditionals within loops, and vector registers. At the end, three versions of the code remained: (1) a non-efficient algorithm, (2) an optimized routine including column-wise matrix storage in a 1D array, and (3) a vectorized implementation, which adds vector registers to the optimized variant. It was shown that the latter presented the best performance benchmarks. However, the larger jump in execution time is made with the optimized code, while the effects of vectorization are only noticeable when N is large.