High Performance Programming – A4

Juan Rodriguez, Samuel Wallace, Alva Rensmo & Joel Andreasson

February 24, 2025

UPPSALA
UNIVERSITET

CONTENTS

1 INTRODUCTION

In this report we treat the N-body problem, that like the 3-body problem can not be solved explicitly using elementary analytical functions. We do this practically with a direct implementation of the time integration for a system of N coupled second-order-differential equations that are derived in cartesian coordinates under Newton's II, III and Gravitation laws. Since this system can not be solved analytically, computer simulations of the multi-body problem becomes a powerful tool to study the corresponding initial value problem (IVP). The IVP serves as a model of the dynamics of a galaxy given a finite set of prescribed initial conditions. Naturally, this problem can become quite computationally heavy due to the amount of degrees of freedom ($2N$), since we trace each particles position in the (x, y)-plane. But in order to trace the particle trajectories, we have to perform a significant amount of floating point operations in each time step. Thus, having a fixed time stepping method then imposes the challenge that there should be no redundant flops, the method should operate on an ideal data structure that stores data contiguously and exploit s spatial and temporal locality as much as possible. In this report we are looking to implement such a simulation in C, and further improve the performance by first identifying hotspots using profilers, and perform serial code optimizations accordingly together with choosing appropiate compiler flags.

2 THEORY

2.1 Newtonian Multi-particle system

The N-body problem can be summarized by the application of Newtons II:nd law on the i:th particle, relative to an inertial system centered at the origin O using the enumeration set $\mathcal{I} = (i \in \mathbb{N}^+ \mid i = 0, 1, \ldots, N-1)$, where each element is an unqiue labeling of a certain particle:

$$m_i \frac{d^2 \mathbf{r}_{i/O}}{dt^2} = m_i \frac{d^2}{dt^2} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \underbrace{\mathbf{F}_i^{(\text{ext})}}_{=0} + \sum_{i=0}^{N-1} \mathbf{F}_{i,j} \tag{1}$$

$$\mathbf{r}_{i/O}(t_0) = \mathbf{c}_i \tag{2}$$

$$\mathbf{v}_{i/O}(t_0) = \mathbf{w}_i \tag{3}$$

with specified initial conditions for the position and velocity of each particle. The first term $\mathbf{F}_i^{(\text{ext})}$ in the right hand side of (1) becomes zero since we assume no external forces (but also we impose no self interaction with $\mathbf{F}_{i,i} = 0, \ \forall i \in \mathcal{I}$), and the only forces that

affect the dynamics are the multi-particle system's internal forces:

$$\mathbf{F}_{i,j} = \underbrace{-\frac{Gm_i m_j}{\|\mathbf{r}_{i/O} - \mathbf{r}_{j/O}\|^3}}_{F_{i,j}} \mathbf{r}_{i,j} \tag{4}$$

$$\mathbf{r}_{i,j} = \underbrace{\begin{bmatrix} x_i \\ y_i \end{bmatrix}}_{\mathbf{r}_{i/O}} - \underbrace{\begin{bmatrix} x_j \\ y_j \end{bmatrix}}_{\mathbf{r}_{j/O}} \tag{5}$$

where $F_{i,j}$ is the magnitude of the force acting on particle i from particle j and $\mathbf{r}_{i,j}$ is the direction in which it acts, that is, the straight line path from j to i. Already here, we can see the first consideration in the implementation that we need to discuss. Since $\mathbf{F}_{i,j}$ is a vectorial quantity one may be inclined to use vectorial data structures, that have defined methods for binary operations on a linear space such as addition, subtraction and scalar multiplication. These come often in numerical linear algebra libraries, which we are not going to use since it is already feasible to work with 2 degrees of freedom per particle and their corresponding scalar quantities.

2.2 Time integration

Once the laws of the system has been determined and the initial conditions set we discretize time as $\mathcal{T} = \{t \mid t = t(n+1) = t(n) + \Delta t, \forall n \in \{0, 1, \ldots, n_{\text{steps}}\}\}$ and change the velocities and positions of the particles according to the *Symplectic-Euler Scheme*:

$$\begin{cases} v_x(n+1) = v_x(n) + \Delta t a_x(n+1) \\ v_y(n+1) = v_y(n) + \Delta t a_y(n+1) \\ r_x(n+1) = r_x(n) + \Delta t v_x(n+1) \\ r_y(n+1) = r_y(n) + \Delta t v_y(n+1) \end{cases} \tag{6}$$

where a_x is the x component of

$$\mathbf{a}_{i/O} = \frac{1}{m_i} \sum_{j=0}^{N-1} \mathbf{F}_{i,j} \tag{7}$$

$$= \frac{1}{m_i} \sum_{j=0}^{N-1} \frac{-Gm_i m_j}{\left[\sqrt{\Delta x_{ij}^2 + \Delta y_{ij}^2} + \epsilon\right]^3} \begin{bmatrix} \Delta x_{ij} \\ \Delta y_{ij} \end{bmatrix} \tag{8}$$

and similarly for a_y, where N is the number of particles in the system. This method combines the implicit and explicit Euler time integrations, with the addition of the Plummer numerical stabilisation constant ϵ.

2.3 Complexity analysis

Now we have to consider our second observation, that for every particle i we have to compute $N-1$ forces that act on i due to a different particle j (not N since we exclude $\mathbf{F}_{i,j}$ with $i=j$). This means that we would have to compute $[2(N-1)]^2$ scalar accelerations at each time step. We can also se that we require 1 subtraction each for $\Delta x_{ij}, \Delta y_{ij}$, 2 multiplications plus 1 addition and function call for the norm of the relative distance, $+1$ addition to add ϵ, $+3$ multiplications for the cubic term, $+1$ division for the inverse cube, and $+3$ multiplications twice for multiplication of $-G \cdot m_j \cdot R^{-3} \cdot \Delta x_{ij}$ and $-G \cdot m_j \cdot R^{-3} \cdot \Delta y_{ij}$, for all $N-1$ terms. This equates to $15 \cdot 2(N-1)+2(N-1) = 32(N-1)$ flops for a single particle since we also have to add all $2(N-1)$ acceleration in the x and y directions. We also have to do $+2$ constant look-ups x_i, y_i and $+3(N-1)$ look-ups for x_j, y_j, m_j, equating to a total of $3N-1$ look-ups. So in general we have $\mathcal{O}(N^2)$ complexity in terms of flops and look-ups, to only compute the forces as the equations are explicitly formulated. What we gain from this is that if we aim to keep the given time-stepping scheme and optimize this algorithm, then we should turn to how we can possibly decrease flops and establish an efficient storage of data in memory, and finally implement the algorithm so that we gain as much from the compiler as possible.

There are several ways to optimize the solution of a problem of this sort. One is to use knowledge about the laws governing the dynamics of the system to exploit symmetries to our advantage. Another possible route that involve computer science rather than physics is to utilize cache-blocking. To use cache-blocking is a way to optimize the usage of memory. The goal is to re-use data as much as possible before it is replaced in the cache. The reason this would make the calculations faster is the fact that memory stored in the CPU is accessed much faster than that from the RAM memory. This would be done by splitting data up into blocks that fit in cache. Then block-operations would be performed at this data localized in cache, henche cache-blocking.

3 METHOD

3.1 Data structure analysis

To solve this problem in a more efficient way, we have to analyse the equations at hand. If we recognize that each particle exerts an equal force in opposite directions ($\mathbf{F}_{i,j} = -\mathbf{F}_{j,i}$) then we can compare the different index structure representations:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,N-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,N-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N-1,0} & a_{N-1,1} & a_{N-1,2} & \cdots & a_{N-1,N-1} \end{bmatrix} \tag{9}$$

$$\begin{bmatrix} 0 & a_{0,1} & a_{0,2} & \cdots & a_{0,N-1} \\ a_{1,0} & 0 & a_{1,2} & \cdots & a_{1,N-1} \\ a_{2,0} & a_{2,1} & 0 & \cdots & a_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N-1,0} & a_{N-1,1} & a_{N-1,2} & \cdots & 0 \end{bmatrix} \tag{10}$$

For the first structure a naive method would be:

```
while (n < n_steps) {
    for (int i = 0; i < N; i++){
        double Fx = 0.0, Fy = 0.0;
        for (int j = 0; j < N; j++) {
            if (i != j) { // Conditional branching in the innermost loop is bad
                Fx += f(x[i], x[j], m[i], m[j]; G, ε); // f is an expensive function
                Fy += f(y[i], y[j], m[i], m[j]; G, ε);  // Division is expensive
            }
        }
        ax[i] = Fx/m_i; ay[i] = Fy/m_i;
    }
}
```

Listing 1 – Naive computation of the internal multi-particle system forces, without exploitation of the skew symmetry that is imposed by Newtons III:rd law.

Since ($\mathbf{F}_{i,j} = -\mathbf{F}_{j,i}$) is just a permutation of indices and adding a sign, the corresponding appropiate index structure becomes:

$$\begin{bmatrix} 0 & a_{0,1} & a_{0,2} & \cdots & a_{0,N-1} \\ \times & 0 & a_{1,2} & \cdots & a_{1,N-1} \\ \times & \times & 0 & \cdots & a_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \times & \times & \times & \cdots & 0 \end{bmatrix} \tag{11}$$

with the loop structure:

```
while (n < n_steps) {
    for (int i = 0; i < N; i++){
        double Fx = 0.0, Fy = 0.0;
        for (int j = i+1; j < N; j++) {
            Fx += f(x[i], x[j], m[i], m[j]; G, ε); // f is an expensive function
            Fy += f(y[i], y[j], m[i], m[j]; G, ε);
        }
        ax[i] = Fx/m_i; ay[i] = Fy/m_i; // Division is expensive
    }
}
```

The amount of iteration tuples above the main diagonal then becomes (by replacing the upper diagonal with 1:s and summing over all the rows)

$$\sum_{k=1}^{N-1} k = \frac{N(N-1)}{2} \tag{12}$$

so instead of iterating over the whole matrix $N(N-1)$ iterations for lower and upper diagonal, and N for the main diagonal $= N^2$, we reduce the amount of necessary iterations from N^2 to $N(N-1)/2$ which a little less than half the initial amount of iterations. Consequently, this also decreases the total amount of flops by the same factor.

3.2 A more efficient data structure & algorithm

Now at this point if further inspect:

$$\begin{bmatrix} 0 & F_{0,1} & F_{0,2} & \cdots & F_{0,N-1} \\ -F_{1,0} & 0 & F_{1,2} & \cdots & F_{1,N-1} \\ -F_{2,0} & -F_{2,1} & 0 & \cdots & F_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -F_{N-1,0} & -F_{N-1,1} & -F_{N-1,2} & \cdots & 0 \end{bmatrix} \tag{13}$$

one may think that storing forces in a matrix corresponding to different particles row by row, might be a good idea. But that would increase instead the amount of memory that is needed to be allocated for the forces ($2N^2$ elements of type **`double`**, if you include diagonal elements and $2N^2 - 2N$ if you do not). Even though it might come of as natural to allocate space in this way, we present the algorithm in (2) below as a rebuttal. The way (2) is written is so that the main data structures we are working with are the following arrays:

$$\begin{bmatrix} \mathbf{m} & \mathbf{x} \times \mathbf{y} & \mathbf{u} \times \mathbf{v} & \mathbf{L} \end{bmatrix} \tag{14}$$

$$\begin{bmatrix} \mathbf{a}_x \times \mathbf{a}_y \end{bmatrix} \tag{15}$$

where $\mathbf{x} \times \mathbf{y}$ denotes the ordered cartesian product (i.e. ordered tuples (x_0, y_0), ...,(x_{N-1}, y_{N-1}) where the order induces the correspondence to the dynamical properties of each particle) and $\mathbf{u} \times \mathbf{v}$ the same product but for the velocities in the x- and y-direction correspondingly.

```
while (n < n_steps) {
    for (int i = 0; i < N; i++){
        for (int j = i+1; j < N; j++) {
            Compute_ax_ay(i, j, m, x, a, G, eps);
        }
    }
}
[...]
// Use inline keyword to tell the compiler that it can put the function body where the functions was called.
static inline void Compute_ax_a_y(int i, int j,
                                  // Help compiler to verify that m and x = [x, y] are in the same memory region using __restrict (pointer aliasing).
                                  double *__restrict m, double *__restrict x,
                                  double *a, double G, double eps) {

    int I, J;
    double dx, dx2, dy, dy2, invR3, Gx, Gy, R, R2, R3, m_i = m[i], m_j = m[j];
    // f is in this case a pseudo code function, not an explicit function.
    Gx = f(x[i], x[j]);
    Gy = f(x[i+1], x[j+1]);
    // Since the forces are non-linear in m_i and m_j, we can not do a[J] -= a[I]
    a[I] += Gx * m_j; // We have very explicit loop-carried dependency chains here.
    a[J] -= Gx * m_i;
    a[I + 1] += Gy * m_j; // We store instead the accelerations in x and y directions contiguously
    a[J + 1] -= Gy * m_i; // and once the nested loops in i and j are finished we have accumulated only relevant quantities.
}
```

Listing 2 – Accumulation of particle accelerations with optimized computation. This method efficiently handles at most four accumulators per particle, aligning well with modern CPU capabilities. The algorithm is both readable and flexible, allowing explicit loop unrolling. Additional optimizations can lead to alternative algorithmic structures, as we will see in Section (4).

A more comprehensive program has been developed to solve the equations (1), (6), and (8) iteratively over (n_{steps}) time steps with step size Δt. The program was optimized using various techniques, including advanced compiler options, ensuring contiguous memory storage, enforcing out-of-order execution optimizations, loop fusion, and minimizing the number of floating-point operations (FLOPs), reads, and writes within the innermost function.

Several cache optimization strategies were explored to enhance the final program's performance. The first involved experimenting with alternative data storage formats in the buffer. Specifically, the data was stored as a single buffer containing all particle attributes, with pointers directing to the segments corresponding to specific attributes. Three buffer storage schemes were evaluated:

$$\begin{bmatrix}\mathbf{m} & \mathbf{x}\times\mathbf{y} & \mathbf{u}\times\mathbf{v} & \mathbf{L}\end{bmatrix}\quad\begin{bmatrix}\mathbf{a}_x\times\mathbf{a}_y\end{bmatrix} \tag{16}$$

$$\begin{bmatrix}\mathbf{m} & \mathbf{x} & \mathbf{y} & \mathbf{u} & \mathbf{v} & \mathbf{L}\end{bmatrix}\quad\begin{bmatrix}\mathbf{a}_x\times\mathbf{a}_y\end{bmatrix} \tag{17}$$

$$\begin{bmatrix}\mathbf{m} & \mathbf{x} & \mathbf{y} & \mathbf{u} & \mathbf{v} & \mathbf{L}\end{bmatrix}\quad\begin{bmatrix}\mathbf{a}_x & \mathbf{a}_y\end{bmatrix} \tag{18}$$

Additionally, several computational optimizations were implemented to reduce overhead and enhance efficiency (e.g. strength reduction). For instance, expensive function calls such as `pow(m,n)` were replaced with equivalent binary operations. Repeated calculations, such as those for flat array indexing, were avoided by saving intermediate results in variables and hence reduce FLOPs even further.

4 RESULTS

```
double *a = calloc(2 * N, sizeof(double));
double *m = DATA; // In DATA [m][x][y][u][v] are contiguously stored in the heap
double *x = DATA + N;
double *y = DATA + 2 * N;
double *u = DATA + 3 * N;
double *v = DATA + 4 * N;
double *ax = a;
double *ay = a + N;
double m_j, x_i, y_i;
int i, j, n;
register double F1, F2, dx, dy, dx2, dy2, R2, R, R3, invR3, Gx, Gy, m_i;
n = 0;
while (n < n_steps) {
for (i = 0; i < N; i++) {
    x_i = x[i]; y_i = y[i]; m_i = m[i];
    F1 = ax[i]; F2 = ay[i]; // Accumulator variables
    for (j = i + 1; j < N; j++) {
        dx = x[j] - x_i;
        dy = y[j] - y_i;
        dx2 = dx * dx;
        dy2 = dy * dy;
        R2 = dx2 + dy2;
        R = sqrt(R2) + eps;
        R3 = R * R;
        R3 *= R;
        invR3 = 1.0 / R3;
        Gx = Gy = invR3;
        Gx *= dx;
        Gy *= dy;
        ax[j] -= Gx * m_i;
        ay[j] -= Gy * m_i;
        m_j = m[j];
        F1 += Gx * m_j;
        F2 += Gy * m_j;
    }
    u[i] += G * (F1 * dt); // Since G is a constant we can move it out from the j loop
    x[i] += u[i] * dt;
    v[i] += G * (F2 * dt); // and reduce unecessary FLOPs
    y[i] += v[i] * dt;
}
n++;
memset(a, 0, 2 * N * sizeof(double));
}
```

Listing 3 – Final version of the best performing C implementation for `[m][x][y][u][v]`, `[ax][ay]`

```
double *data = readData(filename, N);
double *DATA = transform(data, N);
double *a = calloc(2 * N, sizeof(double));
double *m = DATA;
double *x = DATA + N;
double *v = DATA + 3 * N;
double m_i, m_j, x_i, y_i;
int i, j, I, J, n;
register double F1, F2, dx, dy, dx2, dy2, R2, R, R3, invR3, Gx, Gy;

n = 0;
while (n < n_steps) {
    for (i = 0; i < N; i++) {
        I = 2 * i;
        x_i = x[I]; y_i = x[I+1]; m_i = m[i];
        F1 = a[I]; F2 = a[I + 1];
        for (j = i + 1; j < N; j++) {
            J = 2 * j;
            dx = x[J] - x_i; dy = x[J + 1] - y_i;
            dx2 = dx * dx; dy2 = dy * dy;
            R2 = dx2 + dy2;
            R = sqrt(R2) + eps;
            R3 = R * R;
            R3 *= R;
            invR3 = 1.0 / R3;
            Gx = Gy = invR3;
            Gx *= dx; Gy *= dy;
            a[J] -= Gx * m_i; a[J + 1] -= Gy * m_i;
            m_j = m[j];
            F1 += Gx * m_j; F2 += Gy * m_j;
        }
        v[I] += G * (F1 * dt);
        x[I] += v[I] * dt;
        v[I+1] += G * (F2 * dt);
        x[I+1] += v[I+1] * dt;
    }
    n++;
    memset(a, 0, 2 * N * sizeof(double));
}
```

Listing 4 – Final version of the C implementation for `[m][xy][uv]`, `[axay]`

```
1  double *data = readData(filename, N);
2  double *DATA = transform(data, N);
3  double *a = calloc(2 * N, sizeof(double));
4  double *m = DATA;
5  double *x = DATA + N;
6  double *y = DATA + 2 * N;
7  double *u = DATA + 3 * N;
8  double *v = DATA + 4 * N;
9  double m_i, m_j, x_i, y_i;
10 int i, j, I, J, n;
11 register double F1, F2, dx, dy, dx2, dy2, R2, R, R3, invR3, Gx, Gy;
12 n = 0;
13 while (n < n_steps) {
14     for (i = 0; i < N; i++) {
15         I = 2*i;
16         x_i = x[i]; y_i = y[i]; m_i = m[i];
17         F1 = a[I];
18         F2 = a[I + 1];
19         for (j = i + 1; j < N; j++) {
20             J = 2*j;
21             dx = x[j] - x_i; dy = y[j] - y_i;
22             dx2 = dx * dx; dy2 = dy * dy;
23             R2 = dx2 + dy2;
24             R = sqrt(R2) + eps;
25             R3 = R * R;
26             R3 *= R;
27             invR3 = 1.0 / R3;
28             Gx = Gy = invR3;
29             Gx *= dx; Gy *= dy;
30             a[J] -= Gx * m_i; a[J+1] -= Gy * m_i;
31             m_j = m[j];
32             F1 += Gx * m_j; F2 += Gy * m_j;
33         }
34         u[i] += G * (F1*dt);
35         x[i] += u[i]*dt;
36         v[i] += G * (F2 *dt);
37         y[i] += v[i]*dt;
38
39     }
40     n++;
41     memset(a, 0, 2 * N * sizeof(double));
42 }
```

Listing 5 – Final version of the C implementation for `[m][x][y][u][v], [axay]`

4.1 Valgrind – cachegrind check

Data storage	Code line	DLmr %	D1mr %
A	`x_i = x[I]; y_i = x[I+1]; m_i = m[i];`	0.29	0.05
B	`x_i = x[i]; y_i = y[i]; m_i = m[i];`	0.35	0.05
A	`dx = x[J] - x_i; dy = x[J + 1] - y_i;`	39.55	21.55
B	`dx = x[j] - x_i; dy = y[j] - y_i;`	39.46	22.78
A	`a[J] -= Gx * m_i; a[J + 1] -= Gy * m_i;`	39.69	32.33
B	`ax[j] -= Gx * m_i; ay[j] -= Gy * m_i;`	39.38	31.33
A	`m_j = m[j]`	19.71	8.08
B	`m_j = m[j]`	19.58	8.54

Table 1 – DLmr: LL cache data read misses — D1mr: L1 data cache read misses – A: `[m][xy][uv]-[axay]` – B: `[m][x][y][u][v]-[ax][ay]`

Data storage	Code line	D1mw %
A	`J = 2 *j`	11.08
A	`dx2 = dx * dx; dy2 = dy * dy;`	22.16
B	`dx2 = dx * dx; dy2 = dy * dy;`	24.93
A	`R = sqrt(R2) + eps;`	11.08
B	`R = sqrt(R2) + eps;`	12.46
A	`a[J] -= Gx * m_i; a[J + 1] -= Gy * m_i;`	22.16
B	`ax[j] -= Gx * m_i; ay[j] -= Gy * m_i;`	24.93
A	`m_j = m[j]`	11.08
B	`m_j = m[j]`	12.46
A	`F1 += Gx * m_j; F2 += Gy * m_j;`	22.16
B	`F1 += Gx * m_j; F2 += Gy * m_j;`	24.93

Table 2 – D1mw: L1 data cache write misses – A: `[m][xy][uv]-[axay]` – B: `[m][x][y][u][v]-[ax][ay]`

4.2 Benchmark – Vitsippa / MacBook Pro

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	2.326	3.767
`[m][x][y][u][v]-[axay]`	2.230	3.603
`[m][x][y][u][v]-[ax][ay]`	2.231	3.554

Table 3 – Compiler flags: `-O0`

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	0.460	1.379
`[m][x][y][u][v]-[axay]`	0.609	1.313
`[m][x][y][u][v]-[ax][ay]`	0.631	1.362

Table 4 – Compiler flags: `-O2`

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	0.451	1.378
`[m][x][y][u][v]-[axay]`	0.607	1.309
`[m][x][y][u][v]-[ax][ay]`	0.633	1.362

Table 5 – Compiler flags: `-O3`

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	0.456	1.300
`[m][x][y][u][v]-[axay]`	0.606	1.311
`[m][x][y][u][v]-[ax][ay]`	0.630	1.257

Table 6 – Compiler flags: `-O3 -funroll-loops`

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	0.375	0.739
`[m][x][y][u][v]-[axay]`	0.356	0.704
`[m][x][y][u][v]-[ax][ay]`	0.325	0.660

Table 7 – Compiler flags: `-O3 -ffast-math -funroll-loops`

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	0.368	0.741
`[m][x][y][u][v]-[axay]`	0.356	0.703
`[m][x][y][u][v]-[ax][ay]`	0.325	0.658

Table 8 – Compiler flags: `-Ofast -funroll-loops`

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	0.370	0.739
`[m][x][y][u][v]-[axay]`	0.363	0.700
`[m][x][y][u][v]-[ax][ay]`	0.321	0.657

Table 9 – Compiler flags: `-Ofast -funroll-loops -ftree-vectorize`

Data storage	`Min User Time` Mac (s)	`Min User Time` Vitsippa (s)
`[m][xy][uv]-[axay]`	0.235	0.726
`[m][x][y][u][v]-[axay]`	0.197	0.635
`[m][x][y][u][v]-[ax][ay]`	0.166	0.599

Table 10 – Compiler flags: `-Ofast -funroll-loops -ftree-vectorize -march=native`. The compiler flag `-march=native` automatically enables all (AVX+) instruction sets supported by the machine CPU.

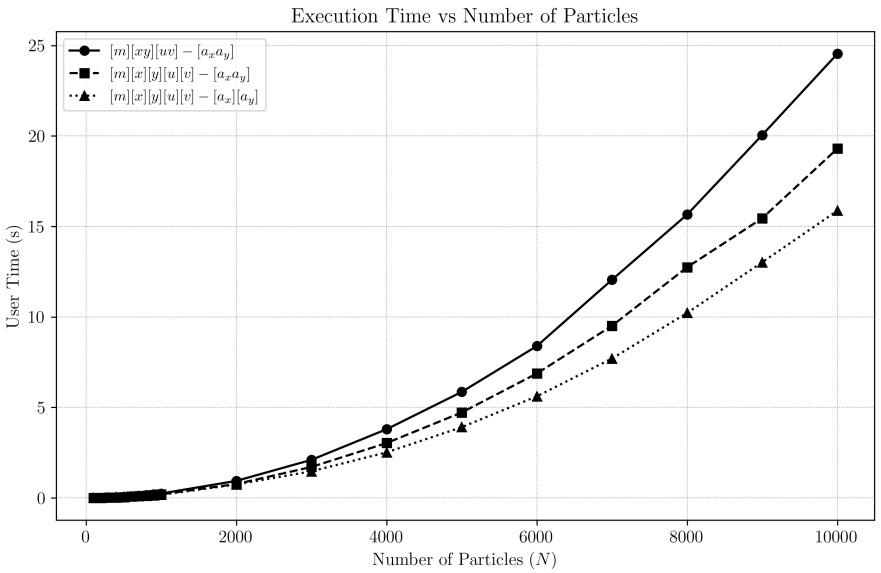


Figure 1 – User times for different amounts of particles for the corresponding data buffers.

5 PARALLELIZATION

5.1 Introduction

Above we utilize several different methods for improving the efficiency of our program in a serial sense. Below we follow implementations in a parallel sense, i.e. by distributing the work between several workers hence speeding up our program. We will first implement Pthreads and then OpenMP and compare the differences between each approach including the ease of use and the speedups. Due to the vastly superior performance below we will be evaluating mainly utilising the 3000 body problem with an extension up to 400 steps to highlight the differences between our programs speed.

5.2 Pthreads

Utilisng the Pthreads library our inital implementatin was naieve and failed to improve performance. We utilised a method of distributing work to a specified N_THREADS, number of threads and then gave each thread acess to a shared memory variable a for the acceleration. However due to the need for safe memory access each thread had to perform for each acceleration update a lock and unlock mutex call as shown below.

Acceleration

```
for (int i = start; i < end; i++) {
        for (int j = i + 1; j < N; j++) {
            // Lock the mutex before updating the shared acceleration array
            pthread_mutex_lock(&mutex);
            Compute_ax_ay(i, j, m, x, data->a, G, eps);
            // Unlock the mutex after updating the shared acceleration array
            pthread_mutex_unlock(&mutex);
        }
    }
```

This resulted in huge overheads in memory locking and unlock and hence we followed a different approach.

Instead for each thread we allocated it some local memory for storing its calculations and because we can at the end of each step add all of the indivudal results together to create a global accelearation only have to lock the global memory N_THREAD times.

Acceleration Imrpoved

```
// Must wait for all our threads to come back to us for this step iteration
        for(int t = 0; t < n_threads; t++){
          double *local_a = NULL;
          pthread_join(threads[t], (void**)&local_a); // Retrieve our calculations
          for(int i = 0; i < 2 * N; i++){
            a[i] += local_a[i];
          }
          free(local_a);
        }
```

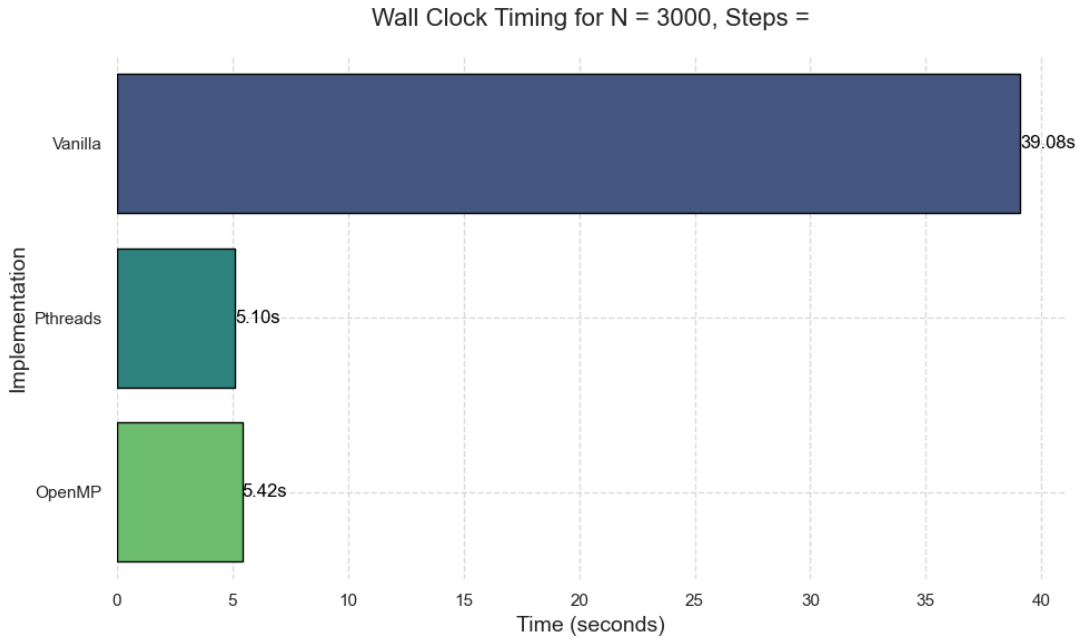
We found that working with POSIX threads was challenging at times because a lot of emphasis is placed on the user to determine the number of threads and work allocation as opposed to the tool of openMP which makes this process a lot less involved.

5.3 openMP

We utilised the same general architecture of local thread memoriues for acceleration and then adding them all at then end for openMP. We found that openMP supplied many out of the box solutions for parallelising code and lead to a much faster implementation time. We specifically found the "omp critical" command useful for performing acceleration updates as this restricts the following code to only be executed one at a time by a thread. Hence essentiually mirroring the mutex locking and unlocking of POSIX threads in one single line.

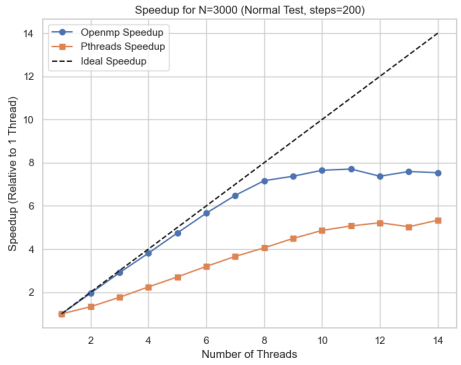
5.4 Performance

The majority of our tests were ran on a MacBook Pro 16-inch 2021, Apple M1 Pro chip with 16GB of RAM.

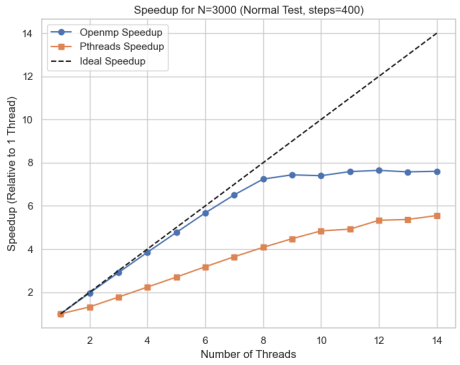


Above you can see that both Pthreads and openMP performed roughly equivalently, this was performed with 10 threads, which is the number of cores of the M1 CPU. We believe that the slight discrepancy for Pthreads is due to the use of a dynamic schedule allocaiton for the threads in the openMP implementation. This has a higher overhead as the workload is calculated at runtime instead of hardcoded, we utilised this as the workload from different problems can change and can be different for different architectures and we aimed to keep the most general form.

As you can see above our speedup is almost linear when using openMP and that once we reach 10 cores our performance levels off. This is because there are no more additional resources that can be assigned and our threads start competing for resources on each core which can actually lead to a decrease in performance.

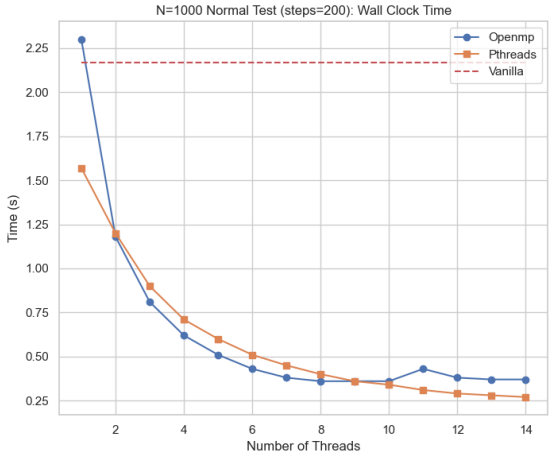


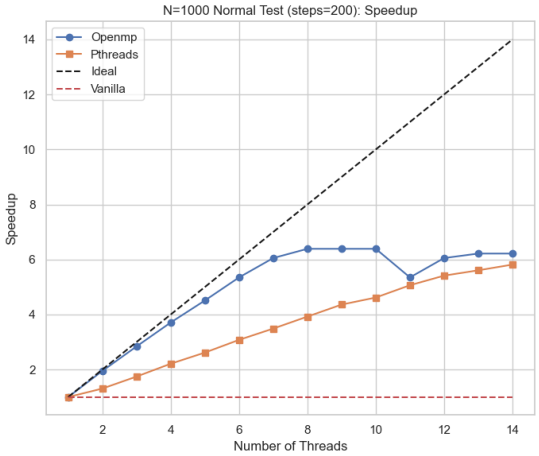
(a) Speed-up for 3000x200



(b) Speed-up for 3000x400

Figure 2 – Comparison of speed-up for different configurations





6 CONCLUSIONS

6.1 Performance improvements through compiler optimizations

As we can see from Tables (3) – (9) only with compiler optimizations we can reach an performance improvement with a factor of around 13 from no compiler optimizations to aggressive optimizations and enabling all instruction sets supported by the CPU. It is also clear that the compiler flags that improve performance the most are `-Ofast / -O3 -ffast-math` and `-march=native`. It is also evident that unless you have an appropriate allocation of memory, the autovectorization can not achieve its full potential (see Tables (7)-(9)). Once `-ffast-math` was applied excplicitly or implicitly in `-Ofast` the autovectorization, gave 41% ($0.235/0.166 \approx 1.41$, see Table (9)) improvement just based on data storage.

6.2 Effect of treating loop-carried dependency chains

The reader has to be informed that since we compute the sums

$$\mathbf{a}_{i/O} = \frac{1}{m_i} \sum_{j=0}^{N-1} \frac{-G m_i m_j}{\left[\sqrt{\Delta x_{ij}^2 + \Delta y_{ij}^2} + \epsilon\right]^3} \begin{bmatrix} \Delta x_{ij} \\ \Delta y_{ij} \end{bmatrix} \tag{19}$$

we establish a long loop-carried dependency for each particle, and the length is proportional to the amount of particles. This of course can cause congestion in the pipeline since the values of the accelerations depend on previously accumulated accelerations, and if not handled correctly a sacrifice in performance given by the auto-vectorization is inevitable. Listings (3) – (5) all show the declaration of register variables: It is known

```
register double F1, F2, dx, dy, dx2, dy2, R2, R, R3, invR3, Gx, Gy;
```

that the compiler can detect which variables to store in fast-access registers. So such declaration may be redundant, but it goes hand in hand with the treatment of loop-carried dependency chain. Since modern CPUs are pipelined, this allows for computations to commence before a previous one has been completed. Congestion in the pipeline can be quite expensive, so when we have loop-carried dependency chain we may resort to loop-unrolling; to feed more instructions to the pipeline and break the chain in smaller pieces, but is hard to achieve when then unroll factor is not a multiple of the amount of iterations. A second method is to define accumulators like `F1` and `F2`, preferably as loop invariants, and not access the memory once a partial sum of the total force sum is computed. Listing (6) highlights the explicit definition of loop invariants that lets us avoid to read/write the acceleration at position `i` for every iteration in `j`. Thus the pipeline does not get interrupted by needing to look up a value in memory before it then can add the next term of the sum. The definition of the accumulators above also decreases the amount of possible cache r/w-misses since we no longer would have the need to access the `i`:th element of the acceleration buffers for each iteration in `j`.

```
while (n < n_steps) {
    for (i = 0; i < N; i++) {
        x_i = x[i]; y_i = y[i]; m_i = m[i];
        F1 = ax[i]; F2 = ay[i]; // Accumulator variables
        for (j = i + 1; j < N; j++) {
            dx = x[j] - x_i; dy = y[j] - y_i;
            dx2 = dx * dx; dy2 = dy * dy;
            R2 = dx2 + dy2; R = sqrt(R2) + eps;
            R3 = R * R;
            R3 *= R;
            invR3 = 1.0 / R3;
            Gx = Gy = invR3;
            Gx *= dx;
            Gy *= dy;
            ax[j] -= Gx * m_i;
            ay[j] -= Gy * m_i;
            m_j = m[j];
            F1 += Gx * m_j; // Could have ax[i] += Gx * m_j
            F2 += Gy * m_j; // ay[i] += Gy * m_j
        }
        // Instead of saving the result of the partial sum,
        // we compute the full sum and store it explicitly in a fast access register
        ↪ variable
        u[i] += G * (F1 * dt);
        x[i] += u[i] * dt;
        v[i] += G * (F2 * dt);
        y[i] += v[i] * dt;
    }
}
```

Listing 6 – Accumulators as loop invariants.

6.3 Cache optimality

Looking at Tables (1) & (2) we observe the percentages for L1, LL cache read/write misses, are not significantly different inbetween data storages. We can therefore say that the cache optimality for the different storages are somewhat the same, and thus cache optimality is not the most significant performance attribute for the different data storages.

7 APPENDIX

```
double *data = readData(filename, N);
double *DATA = transform(data, N);
double *a = calloc(2 * N, sizeof(double));
double *m = DATA;
double *x = DATA + N;
double *v = DATA + 3 * N;
double m_i, m_j, x_i, y_i;
int i, j, I, J, n;
register double F1, F2, F3, F4, dx, dy, dx2, dy2, R2, R, R3, invR3, Gx, Gy;
n = 0;
while (n < n_steps) {
    for (i = 0; i < N; i++) {
        I = 2 * i;
        x_i = x[I], y_i = x[I+1], m_i = m[i];
        F1 = a[I];
        F2 = a[I + 1];
        for (j = i + 1; j < N; j++) {
            J = 2 * j;
            dx = x[J] - x_i;  dy = x[J + 1] - y_i;
            dx2 = dx * dx; dy2 = dy * dy;
            R2 = dx2 + dy2;
            R = sqrt(R2) + eps;
            R3 = R * R;
            R3 *= R;
            invR3 = 1.0 / R3;
            Gx = Gy = G * invR3;
            Gx *= dx; Gy *= dy;
            F3 = a[J]; F4 = a[J + 1]; // Redundant accumulators
            F3 -= Gx * m_i;
            F4 -= Gy * m_i;
            a[J] = F3; a[J + 1] = F4;
            m_j = m[j];
            F1 += Gx * m_j;
            F2 += Gy * m_j;
        }
        a[I] = F1;
        a[I + 1] = F2;
    }
    for (i = 0; i < 2 * N; i += 2) {
        v[i] += a[i] * dt;
        x[i] += v[i] * dt;
        v[i + 1] += a[i + 1] * dt;
        x[i + 1] += v[i + 1] * dt;
    }
    n++;
    memset(a, 0, 2 * N * sizeof(double));
}
```

Listing 7 – Version 1.

```
double *data = readData(filename, N);
double *DATA = transform(data, N);
double *a = calloc(2 * N, sizeof(double));
double *m = DATA;
double *x = DATA + N;
double *y = DATA + 2 * N;
double *u = DATA + 3 * N;
double *v = DATA + 4 * N;
double m_i, m_j, x_i, y_i;
int i, j, I, J, n = 0;
register double F1, F2, dx, dy, dx2, dy2, R2, R, R3, invR3, Gx, Gy;
while (n < n_steps) {
    for (i = 0; i < N; i++) {
        I = 2*i;
        x_i = x[i]; y_i = y[i]; m_i = m[i];
        F1 = a[I];
        F2 = a[I + 1];
        for (j = i + 1; j < N; j++) {
            J = 2*j;
            dx = x[j] - x_i; dy = y[j] - y_i;
            dx2 = dx * dx; dy2 = dy * dy;
            R2 = dx2 + dy2;
            R = sqrt(R2) + eps;
            R3 = R * R;
            R3 *= R;
            invR3 = 1.0 / R3;
            Gx = Gy = G * invR3; // Redundant multiplication
            Gx *= dx; Gy *= dy;

            a[J] -= Gx * m_i;
            a[J+1] -= Gy * m_i;

            m_j = m[j];
            F1 += Gx * m_j;
            F2 += Gy * m_j;
        }
        u[i] += F1*dt;
        x[i] += u[i]*dt;
        v[i] += F2 *dt;
        y[i] += v[i]*dt;
    }
    n++;
    memset(a, 0, 2 * N * sizeof(double));
}
```

Listing 8 – Version 2.