Molecular simulation with three-body interactions - report

Mateusz Biegański

June 15, 2020

Abstract

I present my implementation of parallel Molecular simulation with three-body interactions, according to Penporn Koanantakool, Katherine Yelick paper **A Computation- and Communication-Optimal Parallel Direct 3-Body Algorithm** (2015).

Implementation

In my implementation I am using MPI 3.0 protocol, including asynchronous Isend()/Irecv() operations and collecive ones (like MPI_Gather()). I am using several C++ features like references, but whole code can be easily rewritten to pure C99 standard. In case of $NUM\_PROC > N$ only N nodes are used (new MPI group is created with only active nodes colored), rest finalizes and exits.

Quality tests

My implementation generates proper results against *part_4.txt* and *partg_5.txt* given examples.

Numerical intensity

All computations are performed using **double** datatype, $sizeof(double) == 8$ holds. Let's assume that we are computing one iteration of simulation: we need to compute a) new positions, b) forces, c) accelerations, d) velocities.

a) new positions

for each particle we compute new x,y,z coordinates using equation:

$x(t + \Delta t) := x + v_x t + 0.5 * a_x \Delta t$,

which adds 3 FLOPs per 3 reads and 1 write (I assume delta time in cache). per direction (summarly **9 FLOPS per 96 bytes** per each particle).

b) forces

For each triple we need to compute numerical difference quotient exactly 18 times to compute all potentials (3 triples * 3 directions * 2 (+/-)). Each time we also need to recompute distances in triple, because they differ. I assume *sqrt* function complexity equals 10 FLOPS (it's highly architecture-dependent). During whole computation I assume that all particles' positions are in cache, after being loaded from memory once (it's 3 * 3 * 8 for on read and same for one write = **144 bytes** in total) for each triple.

Single DISTANCE computation consumes 10 + 9 = 19 FLOPS a

```
   #define DISTANCE (std::sqrt((p2x - p0x) * (p2x - p0x) + \
                               (p2y - p0y) * (p2y - p0y) + \
                               (p2z - p0z) * (p2z - p0z)))
```

and 3*1 FLOPS for normalization. Thus we got 60 FLOPS for distances calculation (remark that positions are in cache).

Once we computed distances, we are able to calculate potential (V). Let's consider code snippet below:

```
double three = rij * rik * rkj;
double rij2 = rij * rij;
double rik2 = rik * rik;
double rkj2 = rkj * rkj;

res = 1.0 / std::pow(three, 3);
res += 3.0 * ((-rij2 + rik2 + rkj2) * (rij2 - rik2 + rkj2) * (rij2 + rik2
    - rkj2)) / (8.0 * std::pow(three, 5));
res *= E0;
return res;
```

Assuming FLOPS(std::pow(_, n) == n) summarly we got 27 FLOPS per single V computation.

Both distance and V computation for each triple consists of 18 calls (on same in-cache coordinates), thus we got 18 * (60 + 27) = **1566 FLOPS per 144 bytes.**

c) accelerations

Each acceleration value is computed using corresponding force like below, which implies 3 * 2 FLOPS per 3 * (1 read + 1 write) = **6 FLOPS per 48 bytes** for each particle (assuming MASS in cache).

```
ax = -FX(b1, i) / MASS;
ay = -FY(b1, i) / MASS;
az = -FZ(b1, i) / MASS;
```

d) velocities

Assuming 0.5 * DELTATIME in cache, computation for one particle takes 3 * 3 FLOPS per 3 * (2 reads + 1 write) (new values for acceleration are in cache (were computed just before, in `void compute_acc_maybe_vel()` function. Result is **9 FLOPS per 72 bytes** for each particle.

```
vx = oldvx + 0.5 * DELTA_TIME * (newax + oldax);
vy = oldvy + 0.5 * DELTA_TIME * (neway + olday);
vz = oldvz + 0.5 * DELTA_TIME * (newaz + oldaz);
```

Summary

As we can see above, **computing forces is dominant**. Furthermore, only forces calculation is per number of triples, not per number of particles like velocity or acceleration. It is 1566 FLOPS per 72 bytes for each triple. There are exactly $\binom{N}{3} = O(N^3)$ triples, thus total intensity is about $20 \cdot N^2$ FLOPS/byte, where N = number of particles.

Speedups measuring

In this section I present times that my solution obtained. Python sequential solution was too slow for speedups measuring, and parallel code works with at least 4 nodes, thus I assumed that fastest sequential code would have worked with $T_{seq} = T_4 \cdot 2$, which is even too safe and realistic assumption.

Strong scaling

Strong scaling was computed using **tests/test_100.txt** file and constant work (`stepcount = 40`)

INPUT NAME	PARTICLES	NODES	STEPS	TIME [s]	SPEEDUP
test_100.txt	100	1	40	19.36	-
test_100.txt	100	4	40	9.68	2.0
test_100.txt	100	8	40	5.23	3.72
test_100.txt	100	16	40	2.95	6.56
test_100.txt	100	32	40	2.16	8.96
test_100.txt	100	64	40	6.53	2.96

Weak scaling

I assumed time for `N = 1, stepcount = 2` == time for `N = 4, stepcount = 8`.

INPUT NAME	PARTICLES	NODES	STEPS	TIME [s]	SPEEDUP
test_100.txt	100	4	8	2.16	1.0
test_100.txt	100	8	16	2.00	1.08
test_100.txt	100	16	32	2.37	0.95
test_100.txt	100	32	64	2.89	0.75
test_100.txt	100	64	128	16.56	0.13

Results interpretation

We can see beautiful both weak and strong scaling over $\{4, 8, 16\}$ nodes, but what is interesting, there is a radical collapse in algorithm's efficiency when numerical intensity per node decreases. We can see communication against processor usage overload.