A Fast Multipole Method for the Rotne-Proger-Yamakawa tensor for Polydisperse Particle Systems

Vipul Harsh

India

8thMay, 2014

Outline

Introduction

Why do we need better and faster algorithms?

- The naive algorithm is of quadratic complexity.
- The algorithm doesn't even scale for as less as 10^5 particles.
- Speed is achieved at the expense of accuracy. There are different algorithms to solve the problem approximately.

Contribution

- RPY tensor has already been done for same sized particles. However, most biological and physical systems comprise of different sized particles, for instance molecules within a cell are of different sizes.
- We have done multiple sized particles, which is new using state of the art methods. We also exploit threading and use multiple cores for the computation.
- The timings we are getting is very promising. We are able to do 10^6 particles within 2 minutes. This could help simulate large systems which has several biological and physical applications.

Barnes Hut Algorithm : Basic Idea

- A collection of particles far away from a single particle influence it in similar ways.
- The collection of particles can be approximated by a large point particle and the force(or potential) may be computed using that particle.
- We employ this technique very often without even realizing it. For instance, we don't see the sun as a huge collection of particles(which infact, it is).
- Instead, we treat the sun as a single large particle and use it's center to compute the gravitational force exerted by the sun on say, the Earth.

Fast Multipole Method: Basic Idea

We are interested in computing sums of the form.

$$f(x_k) = \sum_{n=1, n\neq k}^{N} K(x_k, x_n) q(x_n)$$

Suppose that, the Kernel function K is degenerate, *i.e.*

$$K(x, y) = \sum_{i=1}^{p} \phi_i(x)\ \psi_i(y)$$

Then, $f(x_k)$ becomes,

$$\begin{aligned} f(x_k) &= \sum_{n=1, n\neq k}^{N} \sum_{i=1}^{p} \phi_i(x_k)\psi_i(x_n)q(x_n) \\ &= \sum_{i=1}^{p} \left(\sum_{n=1, n\neq k}^{N} \psi_i(x_n)q(x_n) \right) \phi_i(x_k) \end{aligned} \tag{1}$$

Fast Multipole Method: Basic Idea, Continued

We can compute these p sums beforehand. So overall, the final algorithm can be performed in O(pN) time.

In practice, the Kernel function is rarely degenerate. Instead we expand the Kernel function into an infinite converging series, and then take the first p terms.

$$\begin{aligned} K(x,y) &= \sum_{i=1}^{\infty} \psi_i(x)\phi_i(y) \\ &\approx \sum_{i=1}^{p} \psi_i(x)\phi_i(y) \end{aligned} \tag{2}$$

RPY tensor for same sized particles

For equal sized particles, the RPY tensor D(x,y) is the following :

$$D(x,y) = \begin{cases} \frac{k_\beta T}{8\pi\eta}[(\frac{1}{r}\mathbf{I} + \frac{r\otimes r}{r^3}) + \frac{2a^2}{3r^3}(1 - 3\frac{r\otimes r}{r^2})] & r \geq 2a \\ \frac{k_\beta T}{6\pi\eta a}[(1 - \frac{9r}{32a})\mathbf{I} + \frac{3}{32a}\frac{r\otimes r}{r}] & r < 2a \end{cases}$$

RPY tensor for different sized particles

For different sized particles, the radius in the RPY tensor is replaced by effective radius.

$$D(x,y) = \begin{cases} \frac{k_\beta T}{8\pi\eta}\left[\left(\frac{1}{r}\mathbf{I} + \frac{r\otimes r}{r^3}\right) + \frac{(a_x^2+a_y^2)}{3r^3}\left(1 - 3\frac{r\otimes r}{r^2}\right)\right] & r \geq a_x + a_y \\ \frac{k_\beta T}{6\pi\eta a}\left[\left(1 - \frac{9r}{32a}\right)\mathbf{I} + \frac{3}{32a}\frac{r\otimes r}{r}\right] & r < a_x + a_y \end{cases}$$

Where a = effective radius $= \sqrt{\frac{(a_x^2+a_y^2)}{2}}$

Existing Work

We're interested in computing the following sum,

$$u_i^m = \sum_{i=1}^{n} \sum_{j=1}^{3} D_{ij}(x^m, x^n) v_j^n$$

The classical FMM for coulombic interactions is as follows.

$$P^m(q,p,d) = \sum_{n=1, n\neq m}^{N} \frac{q^n}{r_{mn}} + \sum_{n=1, n\neq m}^{N} \frac{(d^n . r_{mn}) p^n}{r_{mn}^3}$$

$$F_m^i(q,p,d) = \frac{\delta P^m(q,p,d)}{\delta x_i^m}$$

Existing Work

The RPY tensor has already been done for equal sized particles by decomposing the RPY tensor into 4 calls to the harmonic FMM, the one which computes electrostatic potential and it's gradient.

This sum is seperated as two sums, one for neighbouring particles and one for far away particles.

$$u_i^m = u_{i,loc}^m + u_{i,far}^m$$

After simplifications, the final expression is

$$u_{i,far}^m = C_1 P_{far}^m(v_i, 0, 0) \text{ - } C_1 \sum_{j=1}^{3} x_j^m F_{i,far}^m(v_j, 0, 0) + F_{i,far}^m(C_1(x.v), C_2, v)$$

RPY tensor for different sized particles

For different sized particles, the sum can be computed using 5 calls to the Harmonic FMM routine.

$$u^m_{i,far} = C_1 P^m_{far}(v_i, 0, 0) - C_1 \sum_{j=1}^{3} x^m_j F^m_{i,far}(v_j, 0, 0) + F^m_{i,far}(C_1(x.v), C_2, \frac{a_n^2}{2}.v) + \frac{a_m^2}{2} F^m_{i,far}(0, C_2, v) \quad (3)$$

Kernel Independent FMM

Recently there have been attempts to generalize the FMM for any Kernel function that smoothens for large r. These aim at computing sums of the form,

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_j) q(y_j)$$

Where, y_1,...y_N are the "source" points and x_1,...x_N are the "target" points at which the potential needs to be computed.
In our case, the target points are the same as source points.

Kernel Independent FMM

We implemented the RPY tensor for different sized particles using KIFMM with 2 calls to the KIFMM.

$$u_i^m = \sum_{n=1}^{N} \sum_{n=1}^{3} D_{ij}(x^m, x^n) v_j^n$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{3} \frac{k_\beta T}{8\pi\eta} \left[\left(\frac{1}{r} \mathbf{I} + \frac{r \otimes r}{r^3} \right) + \frac{(a_m^2 + a_n^2)}{3r^3} \left(1 - 3\frac{r \otimes r}{r^2}\right) \right]_{ij} v_j^n$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{3} \frac{k_\beta T}{8\pi\eta} \left[\left(\frac{1}{r} \mathbf{I} + \frac{r \otimes r}{r^3} \right) + \frac{a_n^2}{3r^3} \left(1 - 3\frac{r \otimes r}{r^2}\right) \right]_{ij} v_j^n$$

$$+ a_m^2 \sum_{i=1}^{n} \sum_{j=1}^{3} \frac{k_\beta T}{8\pi\eta} \left[\frac{1}{3r^3} \left(1 - 3\frac{r \otimes r}{r^2}\right) \right]_{ij} v_j^n$$

Adding Brownian dynamics

- Brownian forces for each particles is a (N $\times$ 3) multivariate gaussian distribution with the entries of the covariance matrix being the corresponding RPY tensor.
- To generate such a distribution, we generate a standard normal (N $\times$ 3) multivariate gaussian distribution.
- Multiplying this distribution with the square root of the covariance matrix gives a vector with the required distribution.
- However, it is expensive to compute the square root directly. Instead we use an approximation to the square root.

$$\sqrt{D}z \approx \sum_{i=1}^{T} a_i D^i z$$

Dz can be computed using FMM, and hence is not computationally expensive. This way, we avoid computing $\sqrt{D}$ explicitly.

Monodisperse vs Polydisperse

- There's just one call to the KIFMM routine for Monodisperse particles (as opposed to two calls for polydisperse particles). Thus, we expect the KIFMM method to take twice as much time for polydisperse particles.
- For the 5 call method, there are 5 calls instead of 4. The extra call has dipole moments and hence should take a little more than a call without dipole moments.

Timings

$n_{particles}$	KIFMM	5 call FMM	Naive
100	2170	5	2
300	2174	17	25
600	2220	55	130
1000	2289	142	374
3000	4287	744	4288
5000	5309	1734	13469
10000	8029	4270	53083

Table 1 : Comparison of computation times.Time units - ms

Timings

$n_{particles}$	KIFMM	5 call FMM	Naive
15000	10179	6216	-
20000	12667	8568	-
25000	15887	8745	-
30000	26993	11034	-
35000	34050	13915	-
40000	41125	24857	-
45000	44161	37953	-
50000	45639	33962	-
100000	68151	41671	-

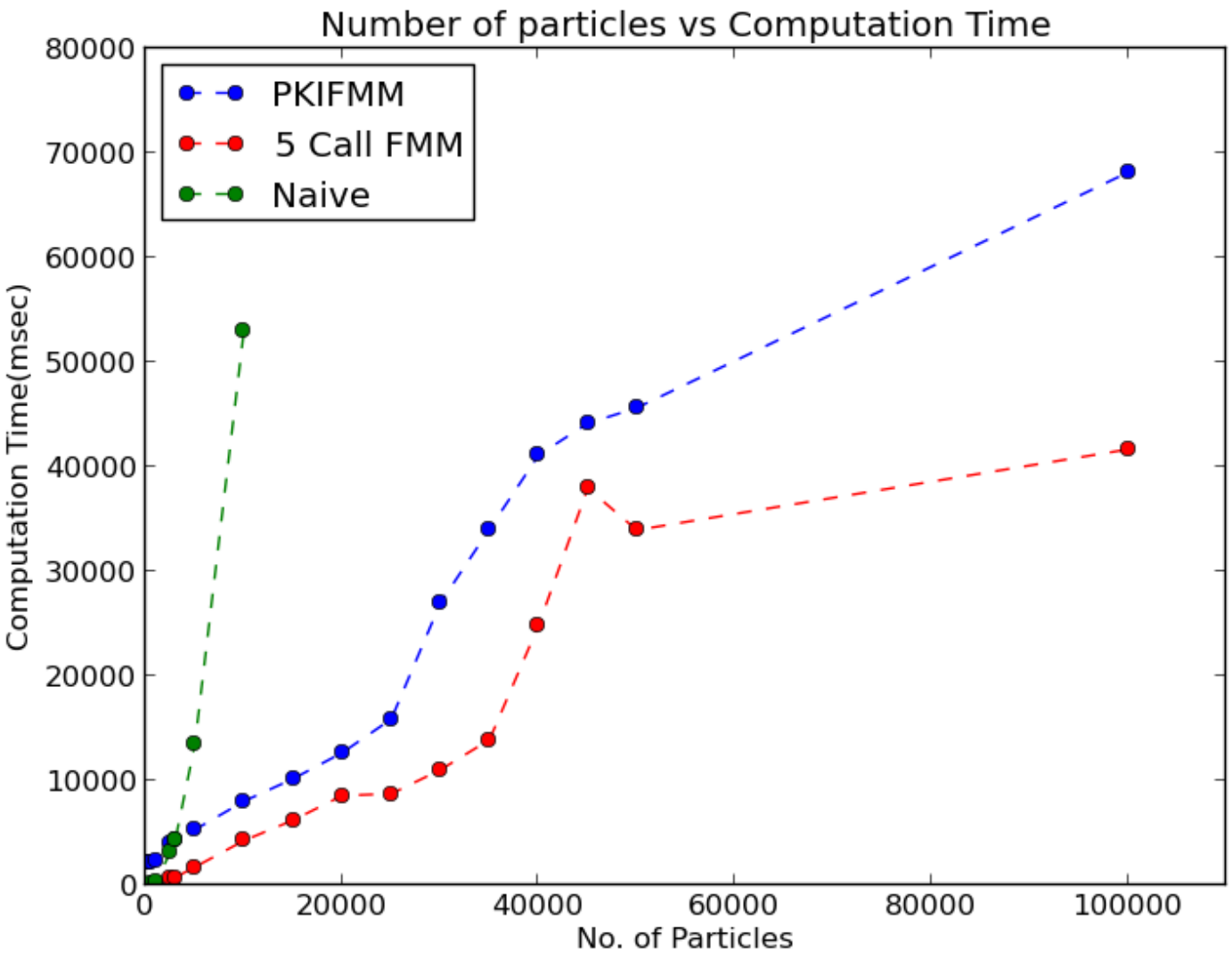
Table 2 : Comparison of computation times. Time units - ms

Parallel KIFMM Timings

Cores	PKIFMM(sec)	SpeedUp
1	642	1
2	354	1.81
4	191	3.36
6	145	4.43
8	124	5.18
10	106	6.06
12	111	5.78
16	113	5.68
20	110	5.84
24	127	5.06

Table 3 : Computation times with multiple cores for 10^6 particles

Timings Comparison



Conclusion

- We have presented two fast methods for doing large scale simulations for polydisperse particle systems involving hydrodynamic interactions and RPY tensor.
- We have extended the 4 call method for polydisperse particle systems, which involves 5 calls to the harmonic FMM.
- We have used the parallel version of the Kernel Independent FMM to run the simulations. We ran our experiments with multiple cores and achieved a decent speedup.

Thank you!