

AMSC664 Final Report

Computing the dynamics of large multi-particle systems using

Fast Multipole Method with multi-scale time stepping

Fei Xue

petrinet@math.umd.edu

Supervisor: Ramani Duraiswami Nail Gumerov

This final report is a summary of my AMSC663/664 project, primarily including the work in spring 2006 semester, with a brief review of my efforts in the previous semester.

1. Motivation

The motivation of this project is to develop an efficient parallelized simulation of dynamics of large multi-particle systems on a parallel CPU/GPU high performance computing architecture. The work includes three parts: parallelization of the fast multipole method (FMM), multi-scale time stepping integrators and mapping the computation of nearby particles forces to GPUs.

Possible applications of the target software include, but not limited to, efficient simulation of classic N-body problems with fairly uniform distribution and finding minimum Coulomb potential in the modern sphere equidistribution [6].

The report is organized as follows: section 2 gives a short introduction to the scientific background; section 3 presents the basic ideas and theories of the FMM related to the parallelization and GPU enhancement; section 4 is about the strategy and details of parallelization; section 5 shows the implementations of the multi-scale time stepping Beeman integrators; section 6 introduces the implementation of GPU computation of neighboring particles interplays; section 7 is the validation and verification of the code, followed by the conclusions and future work in section 8.

2. Scientific Background

Newton's second law of motion is the starting point of multi-particle systems, e.g. the N-body problem in astrophysics [2] and molecular dynamics in biochemistry and biophysics. These systems can be described as a set of mass points with initial positions and velocities, among which pairwise gravitational or Columbic forces act as the only driving impetus. The dynamics of the systems whose pairwise forces obey the inverse square law can be described by the following set of ODEs

$$\ddot{\vec{r}}_i(t) = -G \sum_{j=1, j \neq i}^N m_j \frac{\vec{r}_i(t) - \vec{r}_j(t)}{|\vec{r}_i(t) - \vec{r}_j(t)|^3} \quad (1)$$

subject to initial conditions $\vec{r}_i(t_0) = \vec{r}_{i0}$ and $\dot{\vec{r}}_i(t_0) = \vec{v}_{i0}$.

where $r_i(t)$ and m_i are the position at time t and the mass of i -th particle,

respectively. H. Poincaré was the first to prove that analytical solutions do not exist for general cases when N is larger than 2.

Large multi-particle systems are challenging for simulation as their cost rises with the problem size. The bottleneck is often the prohibitive computation of pairwise forces, which costs $O(N^2)$. The fast multipole method (FMM), first proposed by Greengard [3], has become one of the most powerful methods since late 1980's to speed up the computation by reducing the cost to $O(N \log N)$.

It is worthwhile to emphasize that the FMM can solve much more kinds of different problems, which share a common general mathematical rather than physical nature (matrix-vector multiplication in d dimensions) [4]. The application of the FMM is expanding from acoustics and electromagnetics to elasticity, statistics and computer visions, etc, all of which call for fast computation of matrix-vector multiplications.

The motivation of multi-scale time stepping is to get better energy conservation in simulation, or increase the time steps to decrease computational cost while keeping the energy drift bounded. Unfortunately, none of the integrators we use in simulation can preserve total energy rigorously, though some preserve the momentum. The key to achieve the goal of multi-scale time stepping is to discriminate the particles with fast changing dynamics from those with gently dynamics, and use finer time steps to get high resolution of the fast dynamics. We will show that there is tradeoff among the energy and momentum conservation and accuracy of positions and velocities.

The difference between “collisional” and “collisionless” systems is critical for N -body simulation. “Collisional” systems refer to those where two-body encounters (relaxations) play important roles, while such processes have much less effects on “collisionless” systems. Examples of the two systems can be found in [8]. Physical collisions are out of the scope of discussion. The relaxation in collisional systems imposes difficulties on the particle-particle method since one need very fine time steps to simulate the relaxation process accurately. Practically, N -body simulation refers mainly to the simulation of collisionless systems [8]. We also restrict ourselves to simulating *collisionless* systems in this project.

3. Introduction to FMM

3.1 Key ideas of FMM

The introduction gives both key ideas and some details about fundamentals of the FMM [4]. This part is essential to understand the parallelization of the FMM.

In principle, FMM can be used whenever there is a summation of a function of the distance of two point sets

$$f(x_i) = \sum_{j=1}^N C_j \Phi(x_i - x_j) \quad i = 1, \dots, M \quad (2)$$

where x_i and C_i are the position and the intensity of i -th point, respectively. Φ is some function (e.g. fundamental solution of Laplace equation) centered at x_j to be evaluated at x_i . Straightforward evaluation of all $f(x_i)$, which is in fact a multiplication of $M \times N$ matrix by an N vector, requires MN operations.

The key idea of FMM is easy to follow in the following example. Consider

$$S(x_i) = \sum_{j=1}^N C_j (x_i - y_j)^2 \quad i = 1, \dots, M \quad (3)$$

We can rewrite the right hand side as

$$S(x_i) = \left(\sum_{j=1}^N C_j \right) x_i^2 + \left(\sum_{j=1}^N C_j y_j^2 \right) - 2 \left(\sum_{j=1}^N C_j y_j \right) x_i \quad (4)$$

The virtue of this expression is the factorization of evaluation points x_i and source points y_j . Given the source points, we can evaluate the three summations in brackets without knowing x_i . These summations are evaluated only *once*, but can be used to evaluate $S(x_i)$ for *any* x_i at the cost of evaluating a quadratic function

$$S(x_i) = \alpha x_i^2 + \gamma - 2\beta x_i \quad (5)$$

where α , β and γ have obvious meanings. The computational cost therefore is reduced to $O(M+N)$.

The key idea of FMM, is to use analytical manipulation of series to factorize the summation as a separation of source and evaluation points, to achieve fast summation consequently. We need to understand that this kind of factorization may *not* always be possible for arbitrary matrix-vector product, but when possible, researchers are trying every effort to find such chances because the benefits of FMM are really far reaching.

3.2 Laplace equations and basis functions

It is a classical theory that the gravitational potential ϕ satisfies the Poisson's equation

$$\nabla^2 \phi = 4\pi G \rho \quad (6)$$

where G is the gravitational constant, ρ is the mass density. In the special case of unit mass point located at r_0 , the density is actually a delta function, which leads to

$$\nabla^2 \phi = 4\pi G \delta_{r_0} \quad (7)$$

By the basic theory of partial differential equations, the potential generated by a particle with unit mass is the fundamental solution of Laplace equation, as the Laplacian of this potential is the delta function. When the system consists of a group of mass points with mass m_i located at r_i , the equation formally becomes

$$\nabla^2 \phi = 4\pi G \sum_{i=1}^N m_i \delta_{r_i} \quad (8)$$

That is the gravitational potential satisfies Laplace equation $\nabla^2 \phi = 0$ except for the positions of those mass points.

The FMM usually consider the Laplace, biharmonic and Helmholtz equations in spherical coordinates. With some analytical efforts on separation of variables and the fundamentals of special functions, one can easily find that the following two kinds of spherical basis functions are solutions to the Laplace equation [9]

$$\begin{aligned} R_n^m(\vec{r}) &= r^n Y_n^m(\theta, \phi), \quad n = 0, 1, 2, \dots, \quad m = -n, \dots, n \\ S_n^m(\vec{r}) &= r^{-n-1} Y_n^m(\theta, \phi), \quad n = 0, 1, 2, \dots, \quad m = -n, \dots, n \end{aligned} \quad (9)$$

where R_n^m and S_n^m are “regular” and “singular” basis functions, respectively; r , θ and ϕ are well-defined in spherical coordinates; $\vec{r} = r(\sin \theta \cos \phi, \sin \theta \sin \phi, \cos \theta)$; Y_n^m is the spherical harmonics. The fundamental solution is nothing but $S_0^0(\vec{r})/2\sqrt{\pi}$. These basis functions can be used to generate more complicated solutions of Laplace equation due to its linearity and homogeneity.

With more knowledge of generating functions and the relations between the spherical harmonics and Legendre polynomials [11], one can write the fundamental solution of Laplace equation as a sum of series in which each term is a *factorization* of the source and evaluation points [9].

$$\begin{aligned} \frac{1}{4\pi |\vec{r} - \vec{r}_0|} &= \frac{1}{r_0} \sum_{n=0}^{\infty} \frac{1}{2n+1} \left(\frac{r}{r_0}\right)^n \sum_{m=-n}^n Y_n^{-m}(\theta, \phi) Y_n^m(\theta', \phi') \\ &= \sum_{n=0}^{\infty} \sum_{m=-n}^n \frac{1}{2n+1} S_n^{-m}(\vec{r}_0) R_n^m(\vec{r}) \quad r < r_0 \\ \frac{1}{4\pi |\vec{r} - \vec{r}_0|} &= \frac{1}{r_0} \sum_{n=0}^{\infty} \frac{1}{2n+1} \left(\frac{r_0}{r}\right)^{n+1} \sum_{m=-n}^n Y_n^{-m}(\theta, \phi) Y_n^m(\theta', \phi') \quad (10) \\ &= \sum_{n=0}^{\infty} \sum_{m=-n}^n \frac{1}{2n+1} R_n^{-m}(\vec{r}_0) S_n^m(\vec{r}) \quad r > r_0 \end{aligned}$$

3.3 Translation of basis functions

Section 3.3 and 3.4 are a short review of the basic theories and structure of FMM introduced in [4].

One of the key components of FMM is the translation of basis functions, which is analogous to shifting the center of a Taylor series. For example, we can shift the center of Taylor series of $1/(1+x)$ from the origin to 1 as follows

$$\frac{1}{x - (-1)} = \sum_{n=0}^{\infty} (-1)^n x^n = \frac{1}{2} \sum_{n=0}^{\infty} \left(\frac{-1}{2}\right)^n (x-1)^n \quad |x| < 1 \quad (11)$$

The same idea applies to FMM: suppose some solution of Laplace equation is a summation of basis functions, we can then shift the center of these basis functions from the center \mathbf{O} to a new center \mathbf{O}' to express the same solution under assumptions on the location of evaluation points. In figure 1, for instance, a multipole (or singular basis function) centered at \mathbf{O} can be expressed as a summation of multipoles or local (regular) basis centered at \mathbf{O}' , depending on the distance between the evaluation point and \mathbf{O}' . Almost the same expression exists for translating local basis, but there is no constraint on the evaluation point location.

The virtue of such a translation is again the factorization which splits the centers from the evaluation points, as one can see from figure 1. Given the difference vector \vec{s} between the old and new centers (their absolute coordinates are not important), this factorization gives us the chance to compute the translation coefficients $(S|S)_{kn}^{sm}$ or

$(S | R)_{kn}^{sm}$ once and evaluate $S_n^m(\vec{t} + \vec{s})$ for any \vec{t} at the cost of $O(p^2)$, where p^2 is the number of truncated terms.

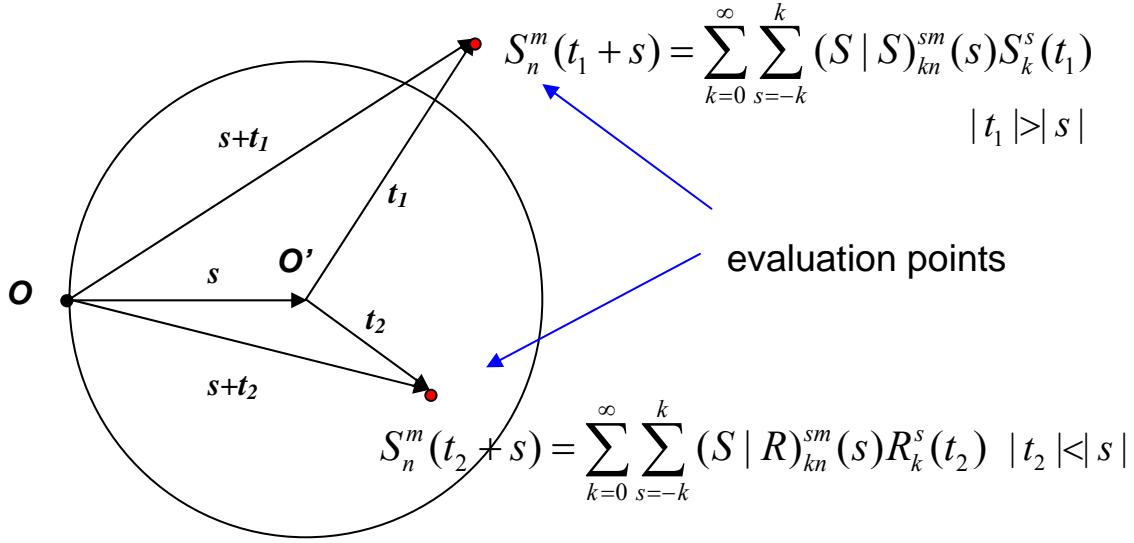


Figure 1 Translation of multipoles (singular basis) from O to O'

In figure 1, the upper translation is called *multipole-to-multipole* (or singular-to-singular), and the lower one is called *multipole-to-local* (or singular-to-regular).

The complexity of translation is essential for the efficiency of FMM. Suppose we use the first p^2 terms of the infinite series of basis functions (S_n^m or R_n^m with $n = 0, 1, \dots, p-1$, $m = -n, \dots, n$) to represent the potential in some domain. This p is called the truncation number. Using the formula in figure 1, we can derive the relation between the coefficients of the basis functions centered at O and O' as follows

$$\begin{aligned} \phi &= \sum_{n=0}^p \sum_{m=-n}^n C_n^m S_n^m(t+s) = \sum_{n=0}^p \sum_{m=-n}^n C_n^m \sum_{k=0}^p \sum_{s=-k}^k (S | R)_{kn}^{sm}(s) R_k^s(t) \\ &= \sum_{k=0}^p \sum_{s=-k}^k \left(\sum_{n=0}^p \sum_{m=-n}^n C_n^m (S | R)_{kn}^{sm}(s) \right) R_k^s(t) = \sum_{k=0}^p \sum_{s=-k}^k \tilde{C}_k^s R_k^s(t) \end{aligned} \quad (12)$$

or in matrix-vector product form

$$\begin{bmatrix} \tilde{C}_0^0 \\ \tilde{C}_1^{-1} \\ \tilde{C}_1^0 \\ \tilde{C}_1^1 \\ \dots \\ \tilde{C}_p^p \end{bmatrix} = \begin{bmatrix} (S | R)_{00}^{00} & (S | R)_{01}^{0-1} & (S | R)_{01}^{00} & (S | R)_{01}^{01} & \dots & (S | R)_{0p}^{0p} \\ (S | R)_{10}^{-10} & (S | R)_{11}^{-1-1} & (S | R)_{11}^{-10} & (S | R)_{11}^{-11} & \dots & (S | R)_{1p}^{-1p} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ (S | R)_{p0}^{p0} & (S | R)_{p1}^{p-1} & (S | R)_{p1}^{p0} & (S | R)_{p1}^{p1} & \dots & (S | R)_{pp}^{pp} \end{bmatrix} \begin{bmatrix} C_0^0 \\ C_1^{-1} \\ C_1^0 \\ C_1^1 \\ \dots \\ C_p^p \end{bmatrix} \quad (13)$$

Given the basis functions, the translation is just evaluation of \tilde{C}_m^n in terms of C_m^n .

The straightforward computation above needs p^4 operations, with all the entries of the matrix computed only once for the given distribution of particles. There exist some fast translation methods, such as the rotational-coaxial translation [3] whose complexity is $O(p^3)$. My supervisors made significant work to find $O(p^2)$ translations,

which is the theoretical lower bound because this implies a constant translation cost for each of the p^2 coefficients. The software I used and further developed is based on a $O(p^3)$ translation scheme.

Note that in almost every step of FMM except for the final summation, we are translating basis functions. Translation cost is critical for efficiency because as we need larger p to get more accurate results, expensive translations might make FMM slower than direct method even for reasonably large N , though FMM will eventually become faster for even larger N . [3] has more details on translation of basis functions.

3.4 Procedures of Multi-level FMM

In low dimensional Euclidean spaces, it is natural and efficient to use boxes to partition the computational domain. At the very beginning, the positions of particles are normalized to be put in a unit cube, which is divided into octants recursively. The unit cube is also called level 0 boxes; any of the 8 octants of level 0 box is called level 1 boxes, and so on. The number of finest boxes, therefore, is $2^{d/\max}$, where l/\max is the maximum number of levels used. We only consider 3D space in this report.

The FMM procedure consists of an upward pass, which is run for each finest box up to the level 2 box and uses the multipole-to-multipole translation for these source boxes, the two-step downward pass, which goes from level 2 boxes down to the finest boxes and uses multipole-to-local and local-to-local translations for evaluation boxes, and the final summation in which the total potential on each particle is evaluated [3].

3.4.1 Upward pass

In the first step of upward pass, we need to translate the potential of each particle in the finest box to the box center O and sum up all the multipoles centered at O .

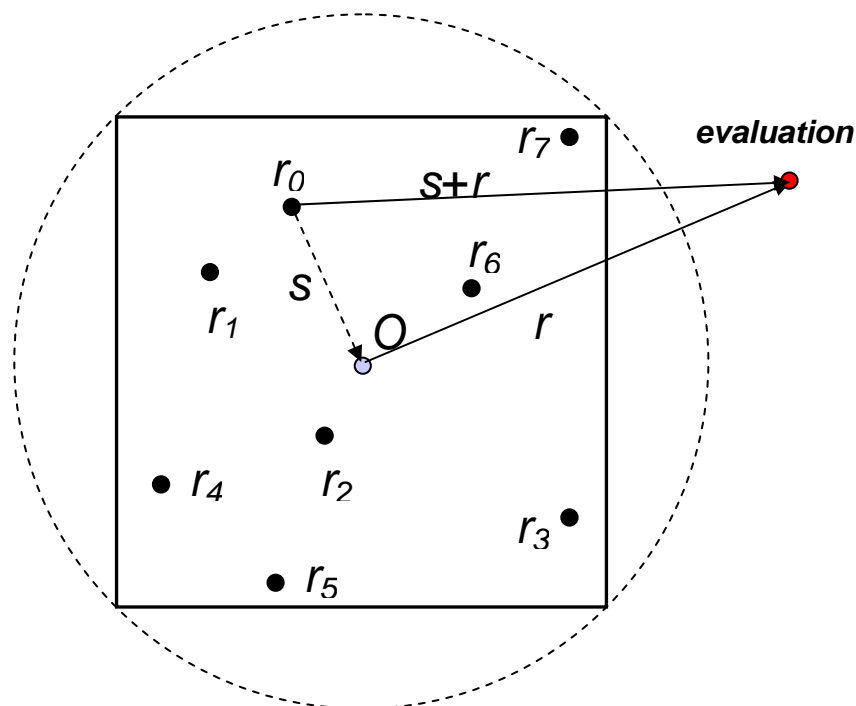


Figure 2 Translate the potential by each particle to the finest box center

This procedure shown in figure 2 is described by the following derivations

$$\begin{aligned}
 m_0 \Phi(\vec{r} - \vec{r}_0) &= \sum_k C_{0k} S_k(\vec{r}) \\
 m_1 \Phi(\vec{r} - \vec{r}_1) &= \sum_k C_{1k} S_k(\vec{r}) \\
 &\dots \\
 m_n \Phi(\vec{r} - \vec{r}_n) &= \sum_k C_{nk} S_k(\vec{r}) \\
 \sum_i m_i \Phi(\vec{r} - \vec{r}_i) &= \sum_i \sum_k C_{ik} S_k(\vec{r}) = \sum_k \left(\sum_i C_{ik} \right) S_k(\vec{r})
 \end{aligned} \tag{14}$$

where the subscript k is a simple representation of the sub-sup script in R_n^m and S_n^m . In fact, $k = n^2 + n + m + 1$, with $-n \leq m \leq n$, the mapping between k and (n, m) is one-to-one. The last summation is valid for any \vec{r} outside the neighboring boxes of the finest box under consideration.

The second step of upward pass is to translate the multipoles centered at the finest boxes to the center of the parent boxes, sum up the 8 sets of multipoles from the finest boxes, and then translate the multipoles centered at the parent box to the center of the grandparent box, and so on through level 2 boxes, as shown in figure 3. In this step, whenever we get a summation of multipoles centered at a box, the potential generated by all particles in this box can be evaluated for any point outside the neighboring boxes of the box.

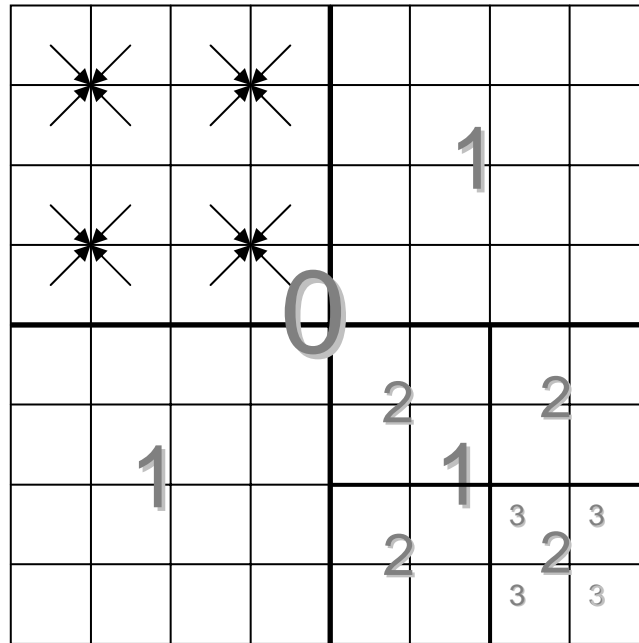


Figure 3 Recursive translation of multipoles from child boxes to parent boxes

The upward pass makes sense because the cost of summing up the multipoles centered at all finest boxes outside the neighboring boxes is quite prohibitive. With the multipoles in parent and grandparent boxes, the potential generated by particles in

summation of local basis centered at the child box. If we could then get the total potential coming from the purple boxes in (b) and also represent this potential as a summation of local basis centered at the child box, we can sum up the two parts and get the total potential coming from the colored boxes. Note that the colored boxes in (a) and (c) are both the whole domain subtracted by the neighborhood of the circled boxes, it is natural that the procedure in figure 4 can be done recursively downward.

As we stop at level 2 boxes in the upward pass, one should consequently start from level 2 boxes in the downward pass. Note that for a *level 2* box, the gray domain shown in figure 4(a) becomes simpler as shown in figure 5(a), which is actually the purple domain. To evaluate the total potential coming from the purple (gray) boxes in the azury box, we need to perform some multipole-to-local translations (because the distance t between the evaluation point and the center of the box is smaller than the distance s between the old and new centers of translation). The same idea applies to level 3 boxes in figure 5(b).

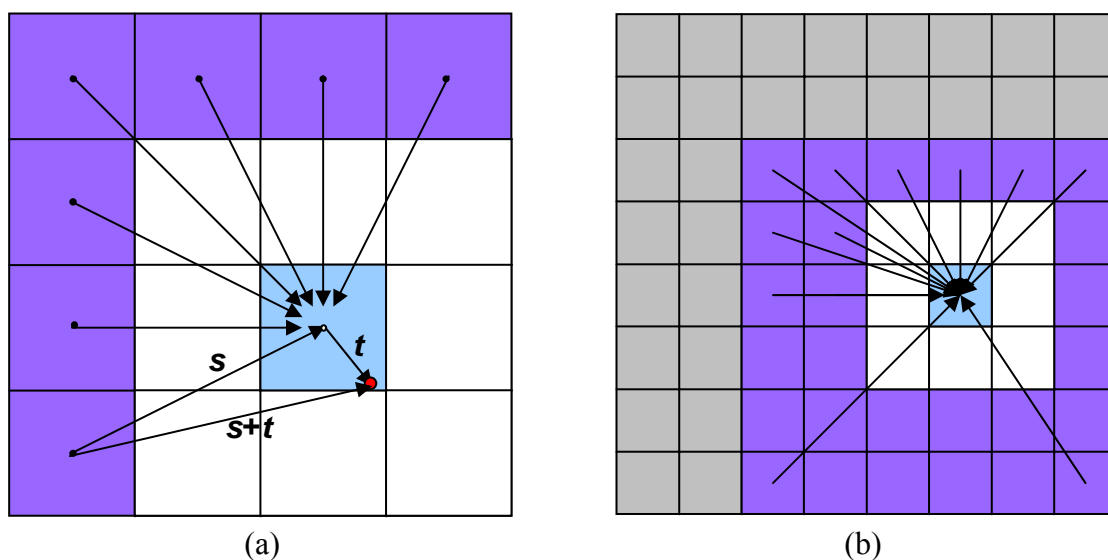


Figure 5 Multipole-to-local translations in the downward pass

3.4.3 Final summation

When we have obtained the summation of local basis functions centered at a finest box which represents the total potential coming from the whole domain except for the neighboring boxes, we can evaluate the total potential on each particle in the finest box by summing up the local basis function value and the potential coming from the neighboring boxes.

The potential from neighboring boxes cannot be evaluated by translation, due to the fact that the particles in the center box may be located between the circle and the box in figure 2 and thus are not in the valid evaluation domain of the multipole-to-multipole translation. Straightforward computation of this potential must be done for each particle in the finest box.

As mentioned before, the introduction to FMM is essential for understanding the parallelization.

4. Parallelization

4.1 Platform

The platform is one of the critical issues for parallelization, as it greatly affects the strategy of implementations such as task decomposition, data structure, balance between communication and computation, what to be communicated and so on.

The platform for my project is the vnode cluster in UMIACS. There are twelve nodes in total. One node is a submit node for users to develop, compile, and submit their jobs to a job scheduler. The other eleven nodes are the compute nodes, where your jobs will actually run. Every node is configured as follows:

- Two 3.0 GHz Xeon EM64 processors (24 CPUs in total)
- 8GB RAM
- 2GB of transitory storage in /tmp
- 60GB of transitory storage in /scratch1
- 160GB of transitory storage in /scratch2

Every node has a connection to a commodity gigabit Ethernet network for IP communication. This in general is only used as a control connection. For message passing, the compute nodes have a connection to an [Infiniband](#)® network.

4.2 Data and task decomposition

The serial FMM for 3D Laplace equations recursively divides each level k box into 8 level $k+1$ boxes. It is natural to use 8 processors, with each taking care of one octant of the whole computational domain—to perform upward pass, downward pass and final summation for each box in its own octant. If there were 64 processors, we could also consider giving each CPU one level 2 box.

This strategy is efficient for systems with fairly uniform distribution of particles. Some stellar systems, on the contrary, have highly nonuniform distribution, and thus are difficult for parallelized simulation with above decomposition strategy as they impose serious problems of load balance. One solution is to use adaptive FMM and manage the load balance dynamically as the simulation evolves [7]. In this project, we restrict ourselves only to fairly uniform distributions.

4.3 Parallelization of upward pass

The upward pass is easy to be parallelized with our task decomposition strategy. Recall that this procedure goes from the finest boxes upward through level 2 boxes, and each processor takes care of an octant consisting of 8 level 2 boxes (figure 3). The code can be completely parallelized as no communication is needed in this procedure.

This procedure is still easy for parallelization on 64 processors, but it becomes not trivial if there were more than 64, say 512, CPUs because in this case each level 2 box is partitioned to different processors and communication becomes necessary when the upward pass translate multipoles to level 2 boxes.

We can evaluate the complexity of upward pass as follows: the translation of

potential of a single particle to the center of the finest box is p^2 (recall equation (*), with all C_n^m equal to 0 except C_0^0 , the translation is only a scalar multiple of a vector); the cost of translating multipoles from a child box to the parent box is p^4 . It is obvious that the total cost of upward pass on all processors is

$$\begin{aligned}
 Np^2 + p^4 (2^{3l_{\max}} + 2^{3(l_{\max}-1)} + \dots + 2^{3(2+1)}) &\leq Np^2 + p^4 2^{3l_{\max}} (1 + \frac{1}{8} + \frac{1}{8^2} + \dots) \\
 &= Np^2 + p^4 2^{3l_{\max}} \left(\frac{8}{7}\right) \quad (15)
 \end{aligned}$$

4.4 Parallelization of downward pass

Communication is necessary for the downward pass because the multipole-to-local translation sometimes needs the information of multipoles and particle positions from other processors, as shown in figure 6.

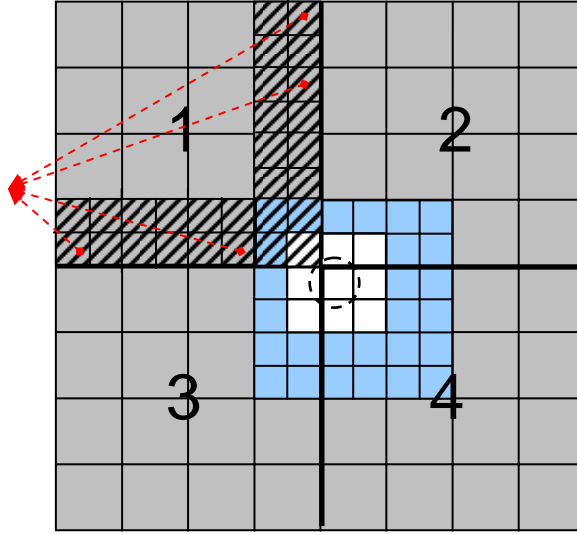


Figure 6 Necessary communication for downward pass

Take the circled box on processor 4 as example. In the downward pass, we need to translate multipoles in the azure boxes to local basis centered at the circled box. The other 3 processors need to send the coefficients of multipoles in azure boxes in their octants to processor 4. Besides, the final summation for the circled box needs the positions and mass of particles in the white boxes on other processors.

In this simulation, to determine exactly what other processors need in downward pass needs a long piece of code and careful ordering of MPI send/receive to avoid deadlocks. Our implementation takes a simple solution: broadcast the union of what is needed by all other processors. It is easy to understand from figure 6 that it is sufficient for processor 1 to broadcast the coefficients of multipoles centered at those striped boxes to other processors for their multipole-to-local translation to boxes at the same level of the circled box. Also, processor 1 broadcasts the positions and mass of particles in the striped boxes just on the boundary (boxes pointed by red arrows) to other processors for their direct summation of potentials from neighboring boxes.

The local-to-local translation, on the contrary, doesn't need any communication

because it simply translate local basis from parent box to the child boxes, and all information needed is local to each processor.

Besides the need for most communication in the whole FMM procedure, the multipole-to-local translation is also one of the most expensive operations in FMM, because this translation must be performed from all purple boxes to the circled box in figure 4(b). In 3D space, the number of purple boxes is $27 \cdot 8 - 27 = 189$, as long as the circled box is not close to the boundary of the whole domain. We therefore need 189 translations for each box, compared to only 1 multipole-to-multipole translation in upward pass, and 1 local-to-local translation in downward pass. This kind of expensive translation may take more than 95% of all translation time.

The complexity of downward pass can be evaluated as follows: each translation takes p^4 (or p^3 if we use rotational coaxial translation) operations, each box may need as many as $189 + 1 = 190$ translations. The upper bound of total cost on all processors is

$$190 p^4 (2^{3/l_{\max}} + 2^{3(l_{\max}-1)} + \dots + 2^{3(2)}) \leq 190 p^4 8^{l_{\max}} (1 + \frac{1}{8} + \frac{1}{64} + \dots) = 190 (\frac{8}{7}) 8^{l_{\max}} p^4 \quad (16)$$

4.5 Parallelization of final summation

After the communication and computation in downward pass, it is then easy to parallelize final summation because each processor already has what is needed for this procedure at the beginning of downward pass. Each processor simply does the work for all finest boxes in its octant.

When evaluating the potential from neighboring boxes, we can get a significant speed up by using the symmetry $F_{ij} = -F_{ji}$. While this is easy for serial code, it is much harder in parallelized code due to the boxes on the boundary of the octants: one processor needs to do the direct computation first and pass the F_{ij} to the processor to prevent the later from computing F_{ji} , but the MPI code are single-threaded and each processor needs to do send/receive explicitly, therefore the later doesn't have any clue whether and when the former is going to send F_{ij} , and cannot prepare to receive the information. This problem is actually due to the fact that the downward pass is totally asynchronous after the communication at the beginning. Each processor doesn't have idea how the progress is on the other processors is proceeding and cannot receive the partial results on F_{ij} from other processors. If there were some support of multi-thread as the case for Java, then each processor can launch an independent thread responsible for receiving data and putting in some buffer for the main thread to read, we can then make full use of symmetric property for any boxes.

After the final summation, the potential and acceleration of each particle have been evaluated. The integrator is launched to compute the new position and velocity of all particles. At this point, a communication is necessary for each processor to pass the information of particles that move into some other octant to the corresponding processor. Each processor then sets up hierarchical boxes and does the upward pass again for the next time step.

One can see from the complexity of upward and downward pass that the cost of translation increases exponentially with the number of box levels l_{\max} . On the other

hand, the cost of directly evaluating potential from neighboring boxes decreases with l_{\max} because the number of particles in neighboring boxes becomes smaller for larger l_{\max} . The total cost of FMM is the sum of the two parts, and can be minimized by finding the best l_{\max} . If we could somehow speed up the direct evaluation cost, we can possibly decrease l_{\max} . This is the motivation of using the GPU for direct evaluation, as the GPU is supposed to do this simple but laborious work very fast.

5. Multi-scale time stepping integrator

5.1 Commonly used integrators for dynamics

The integrators in simulation of multi-particle systems are generally not the most computationally expensive part, but they play a significant role in determining the accuracy of positions, velocities and the drift of linear/angular momentum and energy. The integrator should have some fundamental nice properties, such as minimal need to compute forces, good accuracy in position and velocity, conservation of linear, angular momentum and energy in long time, and hopefully time-reversible.

In molecular dynamics, the Verlet family integrators are the most widely used ones due to their simplicity and good long-time preservation of energy (linear and angular momentum are strictly conserved). The family includes three versions listed below. Though the Velocity Verlet and Leapfrog are almost dominant in the literature, Beeman integrator [5] is more competent in terms of position and velocity accuracy. Rodger [10] discussed the accuracy of common integrators and showed that the Beeman algorithm is the most accurate among the ‘‘Verlet-equivalent’’ algorithms.

$$\begin{aligned} \text{Position (basic) Verlet} \quad x_{n+1} &= 2x_n - x_{n-1} + a_n (\Delta t)^2 \\ v_n &= (x_{n+1} - x_{n-1}) / (2\Delta t) \end{aligned} \quad (17)$$

$$\begin{aligned} \text{Velocity Verlet} \quad v_{n+1/2} &= v_n + \frac{1}{2} a_n \Delta t \\ x_{n+1} &= x_n + v_{n+1/2} \Delta t \\ v_{n+1} &= v_{n+1/2} + \frac{1}{2} a_{n+1} \Delta t \end{aligned} \quad (18)$$

$$\begin{aligned} \text{Leapfrog} \quad v_{n+1/2} &= v_{n-1/2} + a_n \Delta t \\ x_{n+1} &= x_n + v_{n+1/2} \Delta t \\ v_n &= \frac{1}{2} (v_{n-1/2} + v_{n+1/2}) \end{aligned} \quad (19)$$

$$\begin{aligned} \text{Beeman} \quad x_{n+1} &= x_n + v_n \Delta t + \left(\frac{2}{3} a_n - \frac{1}{6} a_{n-1} \right) (\Delta t)^2 \\ v_{n+1} &= v_n + \left(-\frac{1}{6} a_{n-1} + \frac{5}{6} a_n + \frac{1}{3} a_{n+1} \right) \Delta t \end{aligned} \quad (20)$$

Integrator	order of global error in position	order of global error in velocity
Position Verlet	3	1
Velocity Verlet	2	2
Leapfrog	2	2 (for n+1/2 steps)
Beeman	3	3

Table 1 Order of global error of the four integrators

As mentioned in the progress report, we use the Beeman integrator in our codes for its high accuracy in positions and velocities. The cost for the gain is more memory, namely, 5 arrays of size N compared to 3 such arrays in Velocity Verlet, and slight drift in angular momentum. But as we get more accurate positions and velocities, the potential and kinetic energy are computed more accurately. A further justification of using Beeman integrator is that the memory cost of FMM is generally $O(N \log N)$ [4], while the cost of any integrators is $O(N)$, which is in principle not the bottleneck.

Note that the velocity Verlet is the only self-starting integrator. Our strategy is to use velocity Verlet in the first step, followed by Beeman in all the following steps.

5.2 Multi-scale time stepping

The motivation of multi-scale time stepping comes from the following fact: the integrator is supposed to simulate both fast changing and slow changing dynamics, but a single uniform tiny time step can resolve fast dynamics at the cost of getting unnecessary high accuracy for the slow changing dynamics, while a uniform large time step may fail to track the change of fast dynamics and therefore result in considerable energy drift. The natural idea is then discriminate particles with fast dynamics from those with slowly changing dynamics, and use small steps for the former.

The first issue to be considered is the criteria of “fast”. A particle with high speed is supposed to be a “fast” particle for sure, because integrators with large time step for these particles may not find how the particles move within a potential field with high gradient (large variation of potential in very small spatial domain). Besides, a particle with “very” high acceleration should also be a “fast” particle, since they might gain high speed in only a few steps, and the low resolution of the movement in these steps may incur considerable energy drift. One detailed issue is that we use inf-norm of velocity and acceleration for the criteria due to the lower cost than 2-norm.

The procedure of my multi-scale time stepping integrator is as follows

Multi-scale time stepping integrator

1. Given the positions and velocities of all particles at the current step, use FMM to get the instant acceleration, and find “fast” particles
2. find some constant accelerations for the “slow” particles such that when they move with the constant accelerations during all the K sub-steps of the current step, they would reach the same positions as if they moved according to the integrator without multi-scale time stepping (sub-steps). Move the “slow” particles in this way.
3. Use velocity Verlet in the first sub-step, and Beeman from 2nd through K -th sub- step. In each sub-step, compute the accelerations of the “fast” particles directly.
4. At the end of K -th sub-step (also the end of current step), update the velocities of “slow” particles as if they moved according to the integrator without sub-steps.
5. go to 1 if we need more steps

End

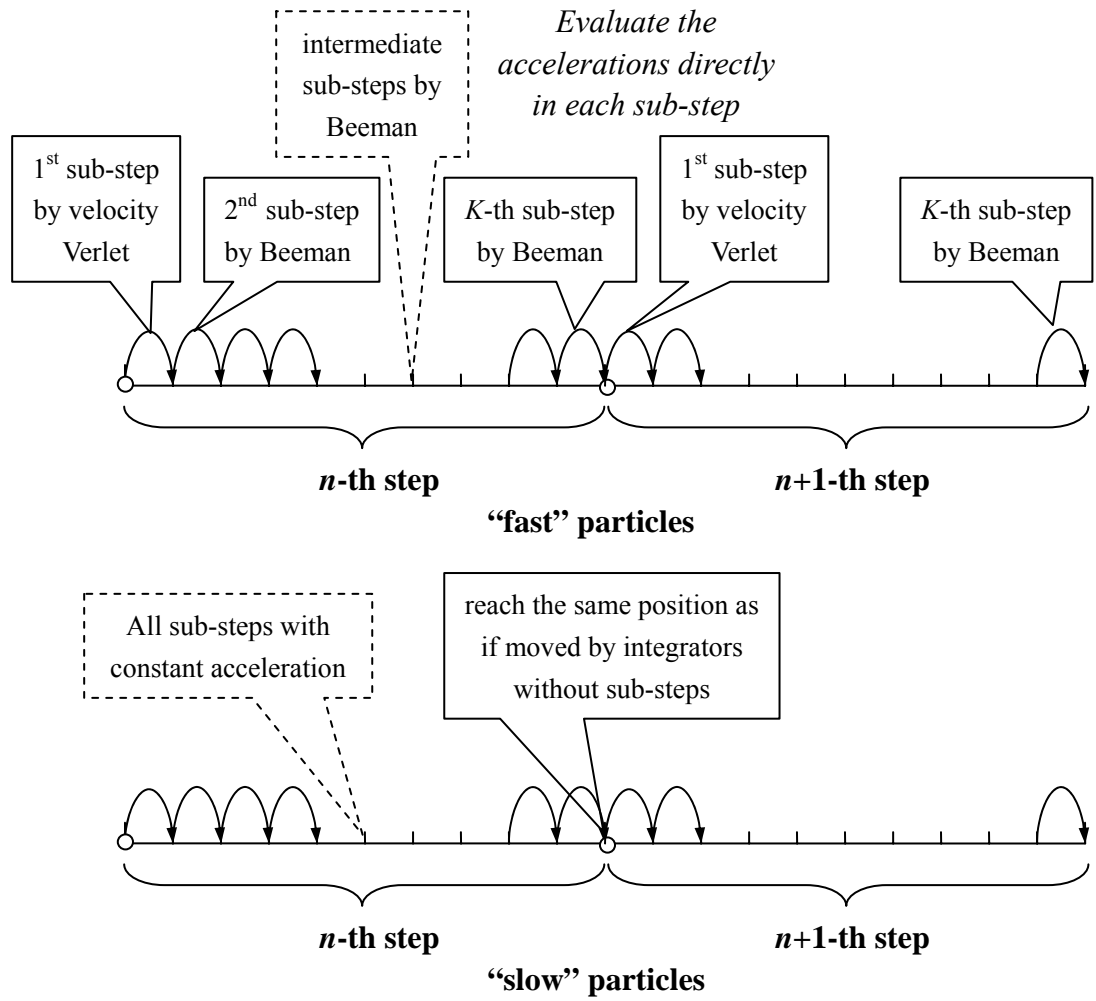


Figure 7 Illustration of multi-scale time stepping

The above procedure uses only two scales of time stepping. One can easily apply three or more scales in the integrator for "slow", "medium" and "fast" particles.

In the above procedure, "slow" particles are moved with constant accelerations in all the K sub-steps, and therefore cannot become "fast" particles until the end of the current step. There are some possible improvements to detect "fast" particles more instantly: take a ball centered at a current "fast" particle, and compute the acceleration of all particles in the ball in each sub-step and see if there are some new "fast" particles, and exclude the particles that become "slow" at some sub-step. This idea is based on the intuitive observation that particles close to "fast" particles have larger chance to be also "fast".

One disadvantage of this algorithm is the loss of rigorous conservation of linear momentum, which is common for multi-scale time stepping integrators. The reason is that we move the slow particles in sub-steps in such a simple way that the dynamics of these particles do *not* obey Newton's second law. One possible solution is to somehow update the velocity of some "slow" particles close to the "fast" ones to get better linear momentum conservation.

The above strategy would not incur considerable additional computation cost as

long as the number of “fast” particles is bounded by $O(\log N)$, because as we compute the accelerations of these particles directly, the additional cost is below $O(N \log N)$, which is the complexity of FMM.

More complicated integrators in molecular dynamics, such as r-RESPA can be found in [13].

6. Mapping the FMM to the CPU/GPU cluster

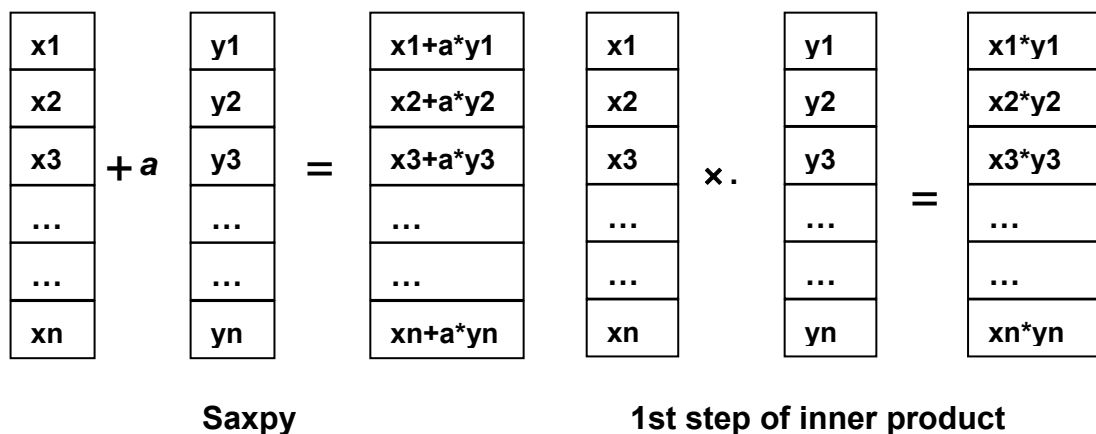
The major motivation of mapping the FMM to the CPU/GPU cluster is to get better efficiency. When a validated piece of CPU code is correctly ported to GPU, it is supposed to run much faster and give the same results as the original code. The CPU/GPU cluster, when appropriately used, is a powerful performance computing platform.

6.1 General purpose GPU programming

The GPU was originally designed for hardware acceleration of graphic rendering, with the ability to process many pixels simultaneously by several parallel processing units (e.g. the GeForce 7800GTX graphic card has up to 24 such units). The use of graphics hardware for general-purpose computation has been active research area for many years. The wide deployment of GPUs in the last several years has resulted in an increase in experimental research with graphics hardware [16]. [12] gives a detailed summary of the types of computation available on modern GPUs. The modern GPU programming is evolving fast, with more features being added and enhanced in every generation of new graphic cards.

Modern GPU 32-bit floating point arithmetic is supported on NVIDIA GeForce FX, ATI RADEON 9500 or higher graphics card. These cards are becoming available on more PCs and laptops.

GPU is vector processor competent for single instruction multiple data (SIMD) problems, where the same single instruction is executed on multiple data entries and there are no dependencies between any output entries. Some typical examples of SIMD include Saxpy and the first step of inner product of vectors.



Saxpy

1st step of inner product

Figure 8 Two examples of SIMD computations

The general purpose GPU programming needs the following support: one of the popular graphic programming languages called the C for graphics (Cg for short) [15]; OpenGL libraries such as OpenGL Utility Toolkit (GLUT) and OpenGL Extension Wrangler Library (GLEW); C/C++ compiler, and the graphic card mentioned above.

6.2 Mapping FMM to CPU/GPU

The relation between the work to be done by GPU and that by CPU is shown as follows: recall that the computation of forces for each particle is nothing but a matrix-vector product, with FMM taking care of some of the work by upward and downward translation, what is left is a sparse matrix-vector product. This is exactly what we need to in final summation: to evaluate potentials coming from neighboring boxes directly. That is, the whole piece of work is

$$\begin{bmatrix} \vec{a}_1 \\ \dots \\ \vec{a}_N \end{bmatrix} = G \begin{bmatrix} 0 & -\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3} & \dots & -\frac{\vec{r}_1 - \vec{r}_N}{|\vec{r}_1 - \vec{r}_N|^3} \\ -\frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|^3} & 0 & \dots & -\frac{\vec{r}_2 - \vec{r}_N}{|\vec{r}_2 - \vec{r}_N|^3} \\ \dots & \dots & 0 & \dots \\ -\frac{\vec{r}_N - \vec{r}_1}{|\vec{r}_N - \vec{r}_1|^3} & -\frac{\vec{r}_N - \vec{r}_2}{|\vec{r}_N - \vec{r}_2|^3} & \dots & 0 \end{bmatrix} \begin{bmatrix} m_1 \\ \dots \\ m_N \end{bmatrix} \quad (21)$$

or $a = GRm$, where a , R and m have obvious meanings from the above equation. CPU FMM code evaluates $a_{FMM} = GR_{FMM}m$, and GPU would do $a_{GPU} = GR_{GPU}m$ so that $a = a_{FMM} + a_{GPU} = GR_{FMM}m + GR_{GPU}m = G(R_{FMM} + R_{GPU})m = GRm$.

The sparse matrix-vector product is one of the classical problems for GPGPU programming [1]. However, this point of view might not be practical for our simulation because the graphic memory might be a serious constraint if the number of particles is moderately large (say, $N=100,000$ or larger). Our strategy is to do it block by block, with one invocation of GPU procedure for each finest box.

Suppose for simplicity that each finest box contains s particles. The center box being considered contains particle $1,2,\dots,s$. We need to evaluate the potential from the neighboring 27 boxes on each particle of the center box as follows:

$$\begin{bmatrix} \vec{a}_1 \\ \dots \\ \vec{a}_s \end{bmatrix} = G \begin{bmatrix} 0 & -\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3} & -\frac{\vec{r}_1 - \vec{r}_3}{|\vec{r}_1 - \vec{r}_3|^3} & \dots & \dots & \dots & -\frac{\vec{r}_1 - \vec{r}_{27s}}{|\vec{r}_1 - \vec{r}_{27s}|^3} \\ -\frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|^3} & 0 & -\frac{\vec{r}_2 - \vec{r}_3}{|\vec{r}_2 - \vec{r}_3|^3} & \dots & \dots & \dots & -\frac{\vec{r}_2 - \vec{r}_{27s}}{|\vec{r}_2 - \vec{r}_{27s}|^3} \\ -\frac{\vec{r}_3 - \vec{r}_1}{|\vec{r}_3 - \vec{r}_1|^3} & -\frac{\vec{r}_3 - \vec{r}_2}{|\vec{r}_3 - \vec{r}_2|^3} & 0 & \dots & \dots & \dots & -\frac{\vec{r}_3 - \vec{r}_{27s}}{|\vec{r}_3 - \vec{r}_{27s}|^3} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ -\frac{\vec{r}_s - \vec{r}_1}{|\vec{r}_s - \vec{r}_1|^3} & -\frac{\vec{r}_s - \vec{r}_2}{|\vec{r}_s - \vec{r}_2|^3} & \dots & \dots & 0 & \dots & -\frac{\vec{r}_s - \vec{r}_{27s}}{|\vec{r}_s - \vec{r}_{27s}|^3} \end{bmatrix} \begin{bmatrix} m_1 \\ \dots \\ m_{27s} \end{bmatrix} \quad (22)$$

which is a dense matrix-vector product. The matrix size is s by $27s$.

6.3 Dense matrix-vector product on GPU

There are several ways to implement dense matrix-vector product on GPU. We pick up the most natural one: for each matrix row, compute the inner product of the row with the column vector. The inner product takes two steps: an element-wise multiplication and a summation (reduction in GPU terminology).

The first step is a typical SIMD operation whose input parameters include G , the matrix row and the mass column vector. While programmers use 1-d array in CPU programming for a vector, they convert to 2-d array (or texture) in GPU programming because textures are easier to be rendered than 1-d array by graphic cards. One useful mechanism here is the intrinsic RGBA texture supported on all GPUs: such texture contains 4 primitive data (say, single precision floating point) values in a single entry of the 2-d array. We can then put the x, y and z components of the R matrix entry and the corresponding mass value in one RGBA texture entry, making full use of memory and computation power. The layout of this input parameter is shown below (suppose we are computing the i -th row)

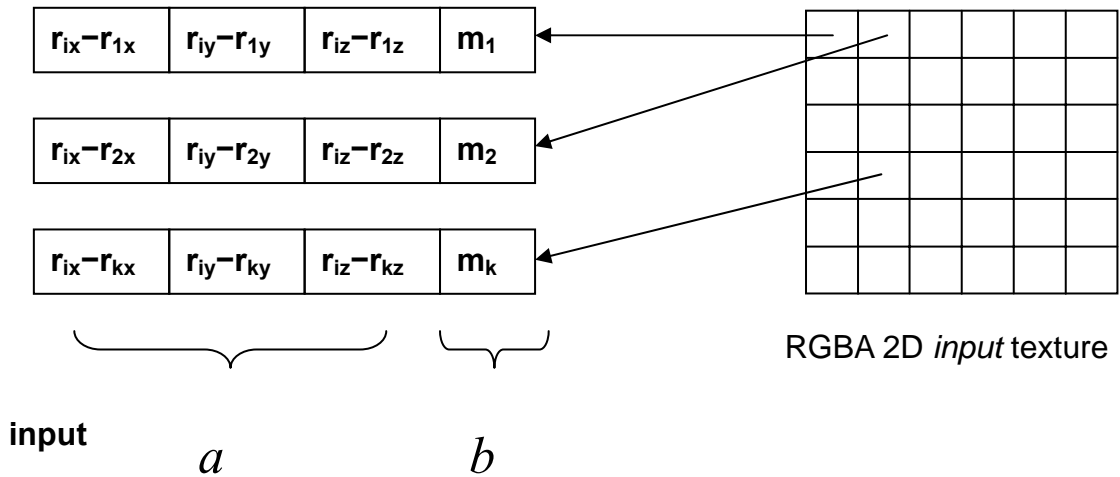


Figure 9 Layout of input parameters of element-wise multiplication

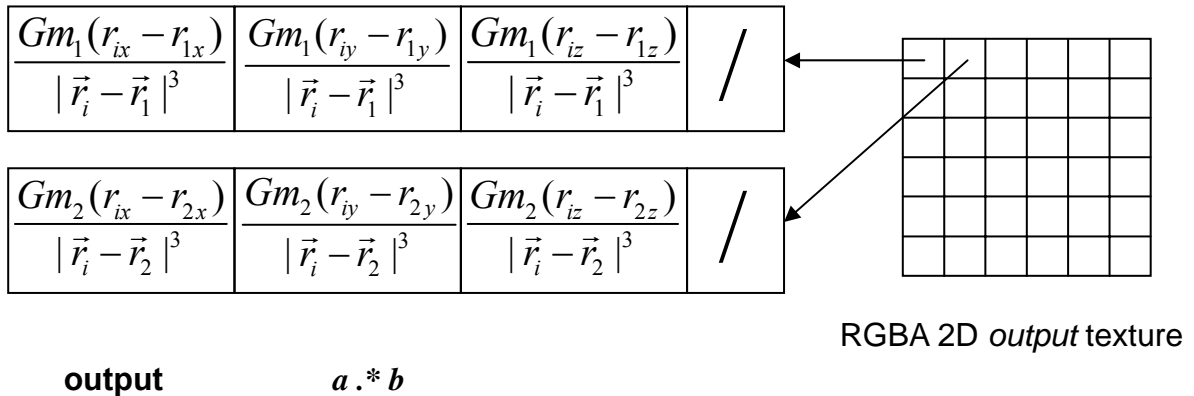


Figure 10 Layout of output parameter of element-wise multiplication

The computation of the element-wise multiplication is simply done by drawing a rectangle with the same size as the input texture. Every entry of the texture is sent to GPU and the Cg program is invoked for each output element in a parallelized way.

The second step, namely the reduction, is a little bit more complicated because the input is a vector but the output is a scalar. This can be done by a recursive partial reduction shown in figure 11. In each pass (execution of GPU program), the texture is reduced to 1/4 of the original size as four entries in the old texture are reduced to one entry in the new texture.

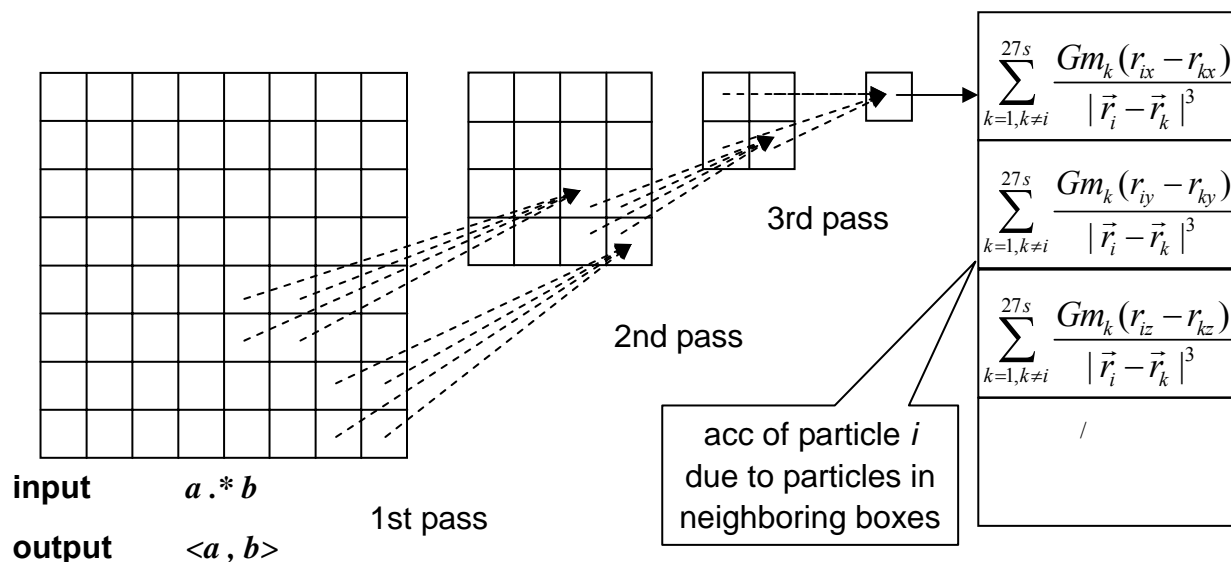


Figure 11 Recursive reduction of texture to a scalar

In i -th loop in GPU procedure, the above two steps are carried out to get the acceleration of i -th particle in the center box due to all particles in the 27 neighboring boxes. At the end of s -th loop, the current GPU procedure ends and the acceleration of all particles in the center box caused by all the 27s nearby particles are evaluated.

At the moment of writing this report, I got the same results from GPU programs as from CPU code, but the bottleneck of downloading the input textures to GPU remains a problem, which takes more than 60% of GPU time. (UMIACS staff has received my results, and thought what I did is correct. But the ideal case is that data input should take little time. We are going to figure out the reason later)

We should also to realize that we can no longer take advantage of the symmetric property $F_{ij} = -F_{ji}$, because this advantage is purely for serial code. Only after we get F_{ij} can we know that F_{ji} doesn't need to be evaluated. GPU codes are intrinsically parallelized, without any assumption or possible use of the order of computation for each element. Therefore, the GPU code must be fast enough to make up for the loss.

In my implementation the Cg procedure is called in a C function, which is called by the FORTRAN FMM routine. Some FORTRAN OpenGL libraries and APIs are also available. We can think about easier mixture of languages in future work and hopefully this could give some performance improvement.

7. Verification and Validation

We use the Plummer's model [2][16] to test our codes since people understand every aspect of this model thoroughly. Plummer's model is a N-body system where the particles are distributed spherically symmetric (in statistical sense) with isotropic velocities. The probability distribution functions of initial mass, positions and speeds are constructed in such a way that the system is in a *dynamical equilibrium* state—the *sample distributions* of positions and velocities are static, so we can not tell how long the system has evolved by analyzing these properties.

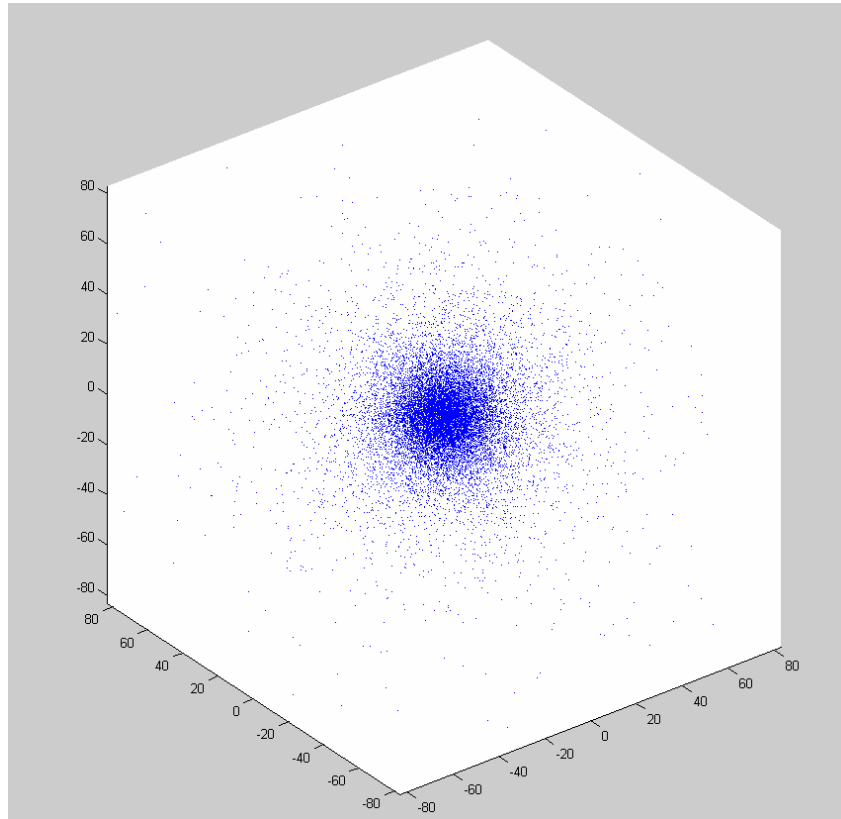


Figure 12 Plummer's model with 16384 particles

Verification

The verification of the parallelized FMM with the Beeman integrator consists of two parts: 1) to check if the straightforward computation with Beeman integrator can get the same sample distribution of positions and velocities before and after simulation, and 2) to check if the dynamics computed by FMM are close enough to those by the straightforward method. Note that the first part is actually validating the applicability of Beeman integrator on Plummer's model, e.g. if the linear/angular momentum and energy drift are small enough; if the positions and velocities are accurate enough so that we would not undermine the dynamical equilibrium. The second part is to verify the FMM code is “correct”— the dynamics computed by FMM can be arbitrarily close to that by the straightforward method.

The first part of verification is already done in last semester in a more or less

rigorous way [14]. The conclusion is that straightforward method with Beeman integrator does keep the dynamical equilibrium of Plummer’s model, including the sample distribution of positions and velocities of particles.

The second step verification is shown in table 2. We use the Plummer’s model with 8192 particles, the uniform (no multi-scale) time step $\delta t = 0.005$ with 100 steps.

p (truncation number)	time cost	average 2-norm of error in position	average 2-norm of error in velocity
4	20.387	5.8981087e-06	1.82811455e-04
5	25.632	1.6118045e-06	4.07934352e-05
6	32.438	6.3953034e-07	1.84028286e-05
7	40.996	2.1366334e-07	5.08061990e-06
8	51.453	9.4818021e-08	1.65653603e-06
9	63.898	3.7467507e-08	6.30517249e-07
10	78.293	1.7745579e-08	4.72877475e-07
11	95.123	8.2478729e-09	1.81276400e-07
12	114.127	4.1043049e-09	6.63084210e-08

Table 2 Errors of dynamics between FMM and straightforward method

Suppose the positions of particle i computed by FMM and the straightforward method are r_i and \tilde{r}_i , respectively, the error in 3rd column is $\frac{1}{N} \sum_{i=1}^N |r_i - \tilde{r}_i|$. The error in velocity in 4th column is defined in the similar way.

This verification shows that the FMM with Beeman integrator is “correct”. One cannot go too many steps in this verification because tiny perturbation at one step may cause huge difference after long evolution of these kinds of dynamical systems.

Speed up of the parallelized FMM

To test the performance of parallelized FMM, we use a sequence of Plummer’s model, in which the later doubles the size of the former.

N	l_{\max}	Straightforward	Serial FMM	Parallelized FMM	Speed up ratio
3654	4	1.365	1.53	0.2899	5.27
7292	4	5.26078	2.71	0.4779	5.68
14714	4	21.3672	5.15	0.8057	6.39
29397	4	84.924	11.33	1.8399	6.16
58860	5	340.782	26.55	4.0863	6.50
117843	5	-	50.17	7.6320	6.57
235308	5	-	140.2	21.0832	6.65
470982	6	-	290.5	58.2*	4.99*

Table 3 Performance of straightforward method, serial and parallelized FMM

Table 3 clearly shows that the straightforward method scales like $O(N^2)$ while the complexity of FMM is $O(N \log N)$. The parallelized FMM can achieve good speed up

ratio for reasonably large N . The speed up ratio keeps increasing except for the $N = 470982$, where a significant drop occurs.

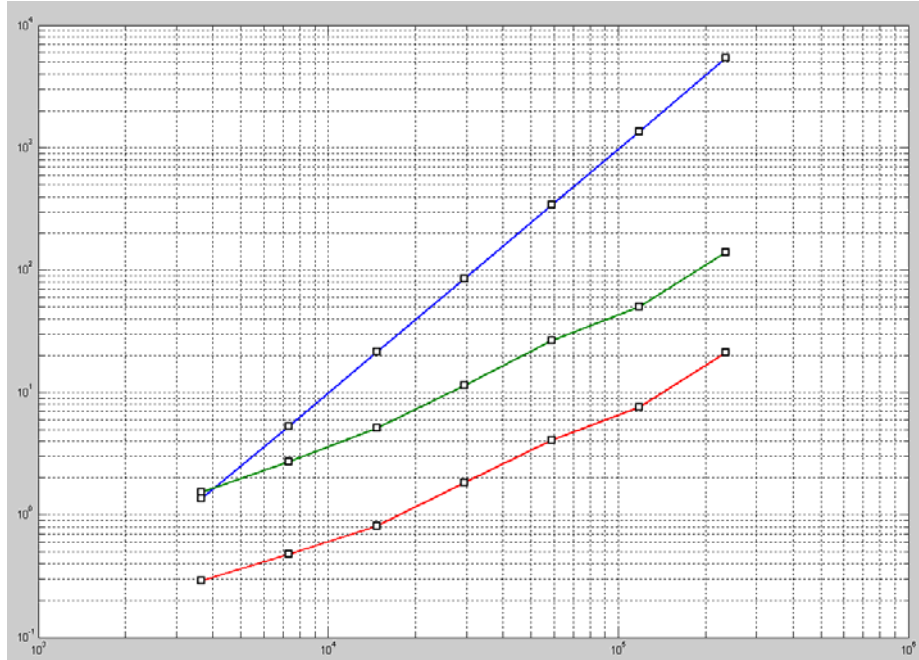


Figure 13 Illustration of time cost of the three methods (N from 3654 up to 235308)

Multi-scale time stepping

We use a Plummer’s model with 4096 particles to test the energy, linear and angular momentum drift by integrators with and without multi-scale time stepping. At the beginning, the system linear momentum is 0, the magnitude of angular momentum is $5.6e-03$, and the total energy is -0.25 . Each time step δt is divided into 10 sub-steps. We compare the system state at $T = 5$ with the initial state to evaluate the drift. “Fast” particles are those whose speed is 2.5^+ times of the average or whose acceleration is 20^+ times of the average.

Time stepping	Energy drift	Linear momentum drift	Angular momentum drift
No multi-scale stepping $\delta t = 0.05$, 100 steps	0.080987	$1.13e-06$	$1.8379e-06$
Multi-scale stepping $\delta t = 0.05$, 100 steps	0.016432	$3.39e-04$	$1.3533e-04$
No multi-scale stepping $\delta t = 0.025$, 200 steps	0.041546	$1.39e-06$	$2.4147e-006$
Multi-scale stepping $\delta t = 0.025$, 200 steps	0.012096	$2.47e-04$	$1.0532e-004$
No multi-scale stepping $\delta t = 0.0125$, 400 steps	0.022822	$1.67e-06$	$2.6629e-006$
Multi-scale stepping $\delta t = 0.0125$, 400 steps	0.003555	$2.52e-04$	$3.7449e-005$

Table 4 The energy, linear and angular momentum drift by non-multi-scale and multi-scale time stepping integrators.

We can see from table 4 that multi-scale time stepping does give much better

energy preservation without noticeable additional time cost. The side effect is also obvious: the linear and angular momentum drift are larger. This effect is common for multi-scale time stepping integrators since the “slow” particles do not obey Newton’s second law in the sub-steps. As mentioned before, we can improve the algorithm by detecting “fast” particles in a more timely manner, or update the velocities of “slow” particles close to “fast” ones to get precise linear momentum conservation. Compared to linear momentum, angular momentum and energy are harder to be conserved due to their nonlinearity. There should be some trade-off between the energy and the angular momentum conservation when using multi-scale time stepping.

8. Conclusions and future work

Project conclusions

We have finished a parallelized FMM with multi-scale time stepping Beeman integrator for N-body problems with fairly uniform distribution. The code has been verified and validated by some basic testing cases to be “correct” and “applicable” to Plummer’s model.

We can achieve ideal speed up ratio by using the parallelized code for fairly large N . In principle, for larger N we need larger l_{\max} for which the proportion of boundary boxes decreases so that the symmetric property can be applied to more boxes and better efficiency can be achieved.

Testing shows that multi-scale time stepping can better preserve the energy without noticeable additional time cost. But the side effect is larger momentum drift. Some easy improvement can be made easily.

We have proposed a mapping of FMM to CPU/GPU cluster. The work to be done by GPU is the direct evaluation of accelerations given by particles in the neighboring boxes, which is a sparse matrix-vector product and has to be processed block by block as a sequence of dense matrix-vector product. If GPU can do this computation very fast, we can reduce l_{\max} and thereby reduce the communication cost. (see figure 6, the number of striped boxes whose multipole coefficients are communicated increases exponentially with l_{\max}).

Scientific Computing conclusions

The system constraints are especially obvious in this project: GPU memory limit and data transfer speed have significant effect on our strategy and optimization. CPU memory doesn’t seem to be bottleneck for our testing problems, but it will definitely show up at some point.

Whether a serial code can be parallelized easily mainly depends on the problem. If the evaluation points are more or less uniformly distributed, the parallelization is straightforward, but it can be quite difficult for highly nonuniform distribution.

Mixed languages can be done easily. My project uses a mixture of FORTRAN, C and Cg without special difficulties. Programmers do not need to be disturbed by the potential inconvenience, as long as the array parameters in the interface between two languages and linking problems are taken care of.

Some platform dependent issues should be noticed, e.g. the same function may return single precision values on one platform and double precision value on another. Besides, it is wise to turn off the optimization flag when debugging since sometimes floating point arithmetic may encounter problems with high-level optimization flag or in the MPI environment on some platforms.

Future work

Future work may include the following

- ⌚ More and thorough testing, including the sphere equidistribution problem, and comparison with other N-body codes
- ⌚ Make the code more scalable and robust for larger simulations
- ⌚ Improvement of multi-scale time stepping including timely detection of “fast” particles and adjustment for better linear momentum conservation
- ⌚ Solve the GPU bottleneck and get good speed up
- ⌚ Porting the parallelized code to 2-CPU or 4-CPU PCs so that more people can run the code conveniently
- ⌚ Develop parallelized adaptive FMM with better load balance strategies to handle systems with highly nonuniform distributions
- ⌚ Some paper or technical reports based on this project

References

- [1] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," SIGGRAPH, pp. 917--924, 2003.
- [2] D. Heggie and P. Hut. The gravitational million-body problem: a multidisciplinary approach to star cluster dynamics. Cambridge University Press. 2003
- [3] Leslie Greengard. The Rapid Evaluation of Potential Fields in Particle Systems. ACM Press, 1987
- [4] N. Gumerov and R. Duraiswami. Fast Multipole Methods for the Helmholtz Equation in Three Dimensions. Elsevier Ltd., Oxford, UK, 2004.
- [5] B. J. Leimkuhler et al. Integration methods for molecular dynamics. IMA Volumes in Math. Appls 82, 1997
- [6] E.B. Saff and A.B.J.Kuijlaars, Distributing many points on a sphere. Trans. Amer. Math. Soc. 350(2) (1998) 523-538
- [7] J. P. Singh *et al.* A Parallel Adaptive Fast Multipole Method. IEEE Supercomputing 93, pp. 54-65, 1993
- [8] Paul Ricker. Astrophysical Hydrodynamics
<http://www.astro.uiuc.edu/~pmricker/research/numhydro/>
- [9] R. Duraiswami and N. Gumerov. Lecture notes on Fast Multipole Methods: Fundamentals and Applications
<http://www.umiacs.umd.edu/~ramani/>
- [10] P. M. Rodger. On the accuracy of some common molecular dynamics algorithms, Mol. Simul. 3 (1989), 263-269.
- [11] Sansone, G. "Harmonic Polynomials and Spherical Harmonics," "Integral Properties of Spherical Harmonics and the Addition Theorem for Legendre Polynomials," and "Completeness of Spherical Harmonics with Respect to Square Integrable Functions." 3.18-3.20 in Orthogonal Functions, rev. English ed. New York: Dover, pp. 253-272, 1991.
- [12] Trendall, C. and Steward, A.J. General Calculations using Graphics Hardware, with Applications to Interactive Caustics. In Proceedings of Eurographics Workshop on Rendering 2000, Springer, 287- 298. 2000.
- [13] M. Tuckerman and B. Berne. Reversible multiple time scale molecular dynamics. Jour. Chem. Phys. 97(3), August 1992
- [14] F. Xue. AMSC 663-664 Progress Report. Fall 2005
<http://www.math.umd.edu/~petrinet/ProgressReport.pdf>
- [15] NVIDIA FAQ http://www.nvidia.com/page/gelato_faq.html
- [16] GPGPU Forum <http://www.gpgpu.org/>
- [17] The Art of Computational Science—the Kali Code for Dense Stellar Systems
<http://www.artcompsci.org/kali/vol/plummer/volume9.pdf>