Parallelized FMM

Jiarui Li

ABSTRACT

Using MPI to parallelize the FMM method, and in each core, using OpenMP to parallelize also.

Keywords: FMM

INTRODUCTION

The fast multipole method (FMM) is a numerical technique that was developed to speed up the calculation of long-ranged forces in the n-body problem. In the project, we will try to parallelize a simple version of the 2D FMM method using both MPI and OpenMP.

MATHEMATICS

In this section, we will briefly introduce the four main operations of the FMM method just to give a simple taste of what is the FMM method.
First of all, the most important step that all FMM method is based on is the Multipole expansion. We have the following theorem, Suppose that m charges of strengths $\{q_i, i = 1, ..., m\}$ are located at points $\{z_i, i = 1, ..., m\}$, with $|z_i| < r$. Then for any z with $|z| > r$, the potential $\phi(z)$ induced by the charges is given by

$$\phi(z) = Q\log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k}$$

where

$$Q = \sum_{i=1}^{m} q_i, \quad a_k = \sum_{i=1}^{m} \frac{-q_i z_i^k}{k}$$

Futhermore, for any $p \geq 1$,

$$\left|\phi(z) - Q\log(z) - \sum_{k=1}^{p} \frac{a_k}{z^k}\right| \leq \frac{1}{p+1}\alpha|\frac{r}{z}|^{p+1}$$

$$\leq \left(\frac{A}{p+1}\right)\left(\frac{1}{c-1}\right)\left(\frac{1}{c}\right)^p$$

where

$$c = |\frac{z}{r}|, \quad A = \sum_{i=1}^{m} |q_i|, \quad \alpha = \frac{A}{1 - |\frac{z}{r}|}$$

In general, the multipole expansion represent the total potential introduced all particle inside certain region, the potential is exerted on some place well separated with these particles. Then, we the second part of the FMM method is the translation of a multipole expansion, which is the following,
Suppose that

$$\phi(z) = a_0\log(z) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k}$$

is a multipole expansion of the potential due to a set of m charges of strengths $q_1, q_2, ..., q_m$, all of which are located inside the circle D of radius R with center at z_0. Then for z outside the circle D_1 of radius

$(R+|z_0|)$ and center at the origin,

$$\phi(z) = a_0 \log(z) + \sum_{l=1}^{\infty} \frac{b_l}{z^l}$$

where,

$$b_l = -\frac{a_0 z_0^l}{l} + \sum_{k=1}^{l} a_k z_0^{l-k} \binom{l-1}{k-1}$$

Futhermore, for any $p \geq 1$,

$$\left| \phi(z) - a_0 \log(z) - \sum_{l=1}^{\infty} \frac{b_l}{z^l} \right| \leq \left(\frac{A}{1 - \frac{|z_0|+R}{z}} \right)^{p+1}$$

with same A defined as before. And then, the third part, which is the most work-intense, is the Conversion of a multipole expansion into a local expansion. So, the local expansion is kind of the opposite of the multipole expansion. The multipole expansion represent the potential due to the charges inside certain region D on some well seperated reigon. On the contray, the local expasion represent the potential in D that created by some well-separated particle that outside the region D. The thoerem is the following, Suppose that m charges of strengths $q_1, q_2, ..., q_m$ are located inside the circle D_1 with radius R and center at z_0, and that $|z_0| > (c+1)R$ with $c > 1$. Then the corresponding multipole expansion converges inside the circle D_2 of radius R centered about the origin. Inside D2, the potential due to the charges is described by a power series:

$$\phi(z) = \sum_{l=0}^{\infty} b_l z^l$$

where

$$b_0 = a_0 \log(-z_0) + \sum_{k=1}^{\infty} \frac{a_k}{z_0^k} (-1)^k$$

and

$$b_l = -\frac{a_0}{l z_0^l} + \frac{1}{z_0^l} \sum_{k=1}^{\infty} \frac{a_k}{z_0^k} \binom{l+k-1}{k-1} (-1)^k$$

for $l \geq 1$. Furthermore, an error bound for the truncated series is given by

$$\left| \phi(z) - \sum_{l=0}^{\infty} b_l z^l \right| \leq \frac{A(4e(p+c)(c+1)+c^2)}{c(c-1)} (\frac{1}{c})^{p+1}$$

where A is defined as before and e is the base of natural logarithms. Finally, is the Translation of a local expansion. The translation of a complex polynomial centered about z_0

$$\sum_{k=0}^{p} a_k (z - z_0)^k$$

into a complex polynomial centered about 0

$$\sum_{k=0}^{p} b_k z^k$$

can be achieved by the complete Horner scheme. So, above is the four main operations of the FMM method. We will discuss the implementation in next section.

A simple sketch of FMM

Initialization
Choose a number of levels so that there are, on average, s particles per box at the finest level. (The number of boxes is then approximately N/s.) Upward Pass
We begin at the finest level, and create multipole expansions from the source positions and strengths. The expansions for all boxes at all higher levels are then formed by the translation the children's' multipole expansions to the center of their parents and merging the translated multipole expansions. Downward Pass
We convert the multipole expansion into a local expansion about the centers of all boxes in b's interaction list. After these calculations are completed, we are left with a local expansion in each box at each level. Beginning at the coarsest level, these local expansions are shifted to the children's level and added to the children's local expansions. After this recursive process reaches the finest refinement level, a local expansion will have been created for each box which describes the field due to all particles outside the box's near neighbours. It is only this expansion which is evaluated. The near neighbour interactions, as before, are computed directly.

IMPLEMENTATION

At the beginning, I was trying to find others' code on the FMM method, and then I can parallelized it. However, it is hard to read someone else code. So I decide to write my own code. The following are my implementation in each stage of the FMM.

Initialization

Instead of writing a tree structure, I initialize a long array of boxes with length N_b, with each boxes is a C++ struct containing a p term multipole expansion and local expansion. First thing first, given N and s, where N is the total number of source points and s is the number of source points per boxes. Then, we have the number of level we want be n, which is the following,

$$n = \log_4(N/s)$$

Once we have the number of level, we know that total number of boxes following a geometric sequence, where

$$N_b = 1 + 4 + ... + 4^{n-1} = \frac{4^{n-1} - 1}{3}$$

The number of boxes on each level would be 4^{n-1}, the dimension of each level would be 2^{n-1}. Furthermore, the starting point for each level l in the long arrary would be,

$$\frac{4^{l-2} - 1}{3}$$

In the initialization, we go over the entire array of boxes to set their multipole and local expansion to zero. Then, we random generating fractions for the x-coordinate and y-coordinate and charges for each source point.

Upward Pass

In the Upward Pass, we start at the finest level, the starting point of the finest level would be,

$$\frac{4^{n-2} - 1}{3}$$

And, there are total number 4^{n-1} boxes in the finest level. Furthermore, the dimension of the finest level would be 2^{n-1}. Given the fact that the XY-coordinate we generate are uniformly distributed in interval $[0, 1]$. Then assure the side length of entire grid of boxes would be 1. So at finest level, each box has length

$$l_n = 1/2^{n-1}$$

When we divide the X-coordinate and the Y-coordinate of the each source points by l_n, the result number x, y would be the index of the box that this source points belongs to. To conculde, the position of the box index in the array would be given x, y

$$\frac{4^{n-2}-1}{3} + x + y \times 1/2^{n-1}$$

Once we located the box, we just add the contribution to the multipole expansion of corresponding box of each source points. This complete the computation of mutipole expansion at finest level. Then, we use the same method to compute the starting point and dimension of each level, and compute the translation of multipole expansion to its parents.

Parallelization and Downward Pass

During the downward pass, we need to exchange information between the MPI core at each level, because at the boudnary of each core, the interaction list contains boxes from adjacent cores. Suppose we are at level l Given that we only implement four cores, So the information exchange reqired would be the following, So, at every level, we first posting a non-blocking receive call for each core,

$$0 \leftarrow 1, 2$$

$$1 \leftarrow 0, 3$$

$$2 \leftarrow 0, 3$$

$$3 \leftarrow 1, 2$$

and then, we collect the information, i.e, the multipole expansion of the boundary boxes, i.e, the last two rows and last two columns. We store those multipole expansion in order into a large double array. Once the we finishing collecting all the data we need, we post a non-blocking send.

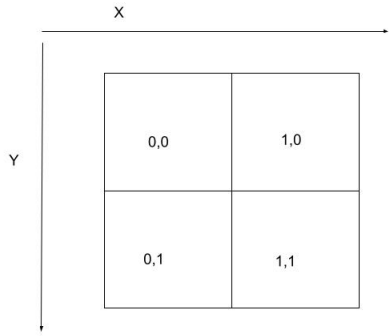
$$0 \rightarrow 1, 2$$

$$1 \rightarrow 0, 3$$

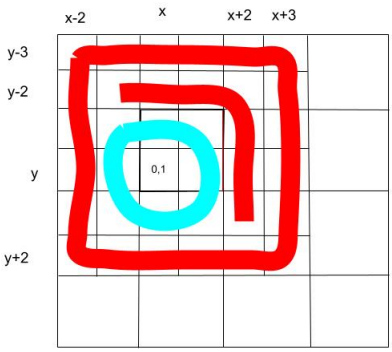
$$2 \rightarrow 0, 3$$

$$3 \rightarrow 1, 2$$

In the meantime, we would like to compute the translation of the multipole expansion to local inside each core. The max number of boxes that any interaction list can contain would be 27. So, for every single box B, I check all possible 27 boxes that are well-separated with B, exist and inside of given core. This step is very complicated, because given any box, I need to figure our the range that its interaction list contains. I should assign an interaction list to each box at the beginning, but it is same work here. So, the way I implemented is to add two addition parameters for the box B. The two parameters are the x-index of children and the y-index of children in B's parent. As we refine, each parent box has four children box, and I therefore have x-index of children and the y-index of children donate the index of children inside a parent box. For example, any



Once we have the children index, one example would be the following,



We can see if the children index is (1,0), the interaction list is contain in the large box from x-2 to x+3, y-3 to y+2 exclude box x-1 to x+1, y-1 to y+1. To conclude,

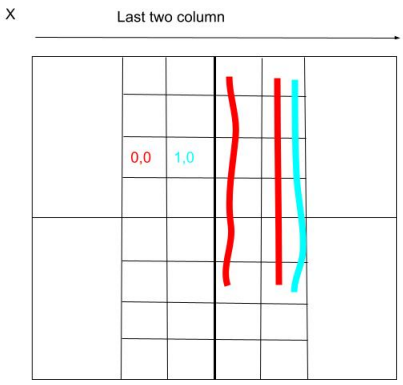
$$(0,0): \quad [x-2,x+3]\times[y-2,y+3]\setminus[x-1,x+1]\times[y-1,y+1]$$

$$(1,0): \quad [x-3,x+2]\times[y-2,y+3]\setminus[x-1,x+1]\times[y-1,y+1]$$

$$(0,1): \quad [x-2,x+3]\times[y-3,y+2]\setminus[x-1,x+1]\times[y-1,y+1]$$

$$(1,1): \quad [y-3,y+2]\times[y-3,y+2]\setminus[x-1,x+1]\times[y-1,y+1]$$

So, for each box B, once we get its x-coordinate, y-coordinate and the children index, we can compute its multipole expansion to local expansion for boxes in B's interaction list. We need to do this computation for every boxes in level l. Once the computation is done for every single box in level l, we would check whether the receive and send are complete. Then, we use the information received to compute the multipole to local operation. Again, we will utilize the children index. The following is an exmaple graph,



We can see if we try to update the last two column, x-children index matters, if x-children index is zero, we need update two column. If x-children index is 1, we only need to update 1 column. In the row case is pretty much the same, this time y-children index matters. If y-children index is zero, we need update two rows. If y-children index is 1, we only need to update 1 row.
To be more specific, core 0 needs update the last two columns and last two rows. Core 1 need to update the first two columns and the last two rows. Core 2 need to update the last two columns and the first two rows. And Core 3 need the update the first two columns and the last two rows. And for each box B contain in received array, the maximum number boxed it can effect would be 12, so we need to go over all possible 12 boxes that B can effect, check whether it exist and valid, then update the local expansion using B's multipole expansion. This complete the operation Multipole to Local.
Finally, from the coarest level, we translate the local expansion of each box to its childrem and add the translated local expansion to it children's.

result

The result was pretty good,

$$N = 10000$$

FMM
Time elapsed is 0.016410 seconds.
Standard
Evaluation time: 1.773828 seconds

$$N = 100000$$

FMM
Time elapsed is 0.220916 seconds.
Standard
Evaluation time: 211.596273 seconds
When I increase N more, the MPI given me some error, which I think it is too large to send and receive. The same problem happen in HW4, when I using MPI send and receive a very large array. MPI given me some errors that I do not understand.
However, we can see the result are still pretty good.

Conclusion

To be honest, the FMM is much more hard to implement than I think at the beginning. Although the parallel part is pretty straight forward. But the FMM itself is very complicated. There are a lot of part of my implementation I think is inefficient and need rework to speed up. It is hard to check whether

certain process is correct or not. I am still working on the accuracy part of my code, given the fact that all the formula is present in the complex world. I could not find some formula that only work with the reals. I just could not get the complex number worked my code. In my computation, I just ignore the complex term and work with the real part. This surely will result in a not very good accuracy. I will keep try implement my complex number to the FMM and work on the accuracy. The GitHub right now only contain the clean code. I will update the GitHub once I successful implement with complex number.
Furthermore, the point I generated would are uniform, the next step would be implement an adaptive version of FMM. We can utilize the material we discuss in the last lecture, the space filling curve to divide the entire grid into different operating cores. However, the information exchange may be a problem. Furthermore, I only utilize 4 cores, the next step would be extend to 16, 32, ... cores. However, more cores may not mean faster speed.

REFERENCES

https://math.nyu.edu/~greengar/shortcourse_fmm.pdf