OPTIMAL FAST MULTIPOLE METHOD DATA STRUCTURES

S. Abeyratne[^], S. Manikonda[*], B. Erdelyi[*,^]
[^]Department of Physics, Northern Illinois University, DeKalb, IL 50115, USA
[*]Physics Division, Argonne National Laboratory, Argonne, IL 60439, USA

Abstract

The Fast Multipole Method (FMM) has been identified as one of the ten most significant numerical algorithms discovered in the 20th century. The FMM guarantees finding fast solutions to many problems in science, such as calculating Coulomb potentials among large number of particles by reducing memory footprint and runtime while attaining very high accuracy levels. One important practical issue that we have to solve in implementing a FMM algorithm is organizing large amounts of data, also called data structuring. The non-adaptive FMM is appropriate when the particles are uniformly distributed while the adaptive FMM is most efficient when the distribution is non-uniform. In practice, we typically encounter highly non-uniform 3D particle distributions. This paper summarizes our implementation of a 3D adaptive FMM algorithm data structure setup for non-uniform particle distributions.

INTRODUCTION

If the Coulomb potential of N particles is evaluated directly, it requires on the order of $O(N^2)$ operations since each pair of particles in the system needs to be taken into account. As the number of particles increases, the computational cost increases rapidly. The FMM overcomes this quadratic complexity [1]. The FMM has a broad range of applications in Beam Physics such as space charge, electron cooling and electron cloud effect. The computational cost of the FMM has two major parts: data structuring and potential calculation. This paper discusses only the data structure and seeks a method for an efficient implementation of the FMM. Consequently, the algorithm used to implement the data structure needs to be efficient. The efficiency of an algorithm is determined by its cost or complexity, which comes in two forms, space complexity and time complexity. The space complexity is a measure of memory usage and the time complexity is a measure of speed. Hence, the key challenges in the implementation of FMM are overall memory management and efficient acceleration of computation. In our analysis, we generated several data sets from arbitrary distributions, namely normal (or Gaussian) and uniform distribution, to measure peak runtime-memory usage and runtime.

Adaptive FMM and non-adaptive FMM, also known as regular FMM, are the two major types of FMM [1]. The notion of adaptation is inevitable in a setup of non-uniform distribution. In adaptive FMM, the subdivision of boxes with particles continues only if a particular box does not meet a precondition. On the other hand, in non-adaptive FMM, all boxes are subjected to recursive subdivision until the number of levels roughly becomes $\log_8 N$. There are some other adaptive FMM algorithms, where the precondition is parameterized by a user defined value, s, which is the average number of particles in the finest boxes [2, 3, 4]. The precondition in this paper is that the maximum number of source particles in the neighbourhood of a given target particle (also known as a receiver or evaluation point) should not exceed a user specified number, q. Therefore, the maximum level reached after hierarchical subdivision is governed by q. We measured the peak runtime-memory usage and runtime for different q, N, and distributions.

The primary data structure we used to implement the adaptive FMM algorithm was an octree (or quadtree in 2D). An octree is a tree data structure in which each internal node has eight children; it is typically used to partition a 3D space. Each node of the octree represents a box. Establishing an octree to save all necessary information makes the code run fast.

Since the boxes in the neighbour list of any given box are of the same size (or at the same level) they can be treated independently. Consequently, in this data structure, the error estimation becomes much easier. Also, this data structure works perfectly if the target particles are identical to the source particles or independent set of particles. Another crucial point of this algorithm is that it is fully adaptive in terms of both source and target particles.

By writing this code in C++ we exploit the portability property of C++. In addition to subsuming optimizers in C++, we optimized this code extensively. We used some non-basic operations such as sorting (used the fastest known sorting algorithm in practice, *quicksort*) and binary searching to make the code run fast. Hence, the algorithm presented in this paper is optimal since it is fast and uses dynamic memory allocation.

ALGORITHM

The first step in implementation of adaptive FMM is space partitioning. We start with a space, also known as the computational domain, large enough to contain all source and target points of the simulation.

In the 2D setup, this space is a square divided into four equal sub-regions (quadrants) and form quadtrees. In the 3D setup, this region is divided into eight equal sub-regions (octants) and form octrees. These sub-regions are known as children of the original region or the parent. In general, the cube (or the square in 2D) becomes the root of the tree, which is subsequently subdivided into 2^{dl} congruent boxes with equal volumes and side length $d/2^l$, where d and l are the side length of the root box and the

ISBN 978-3-95450-115-1

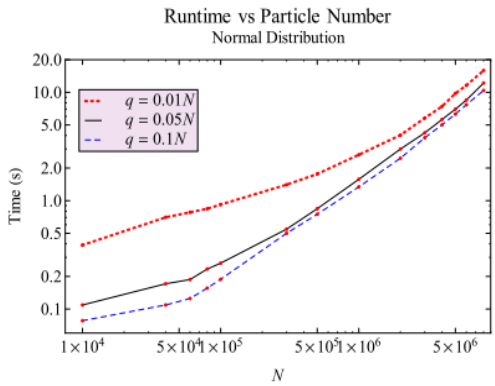

Figure 1: Variation of runtime with particle number, N.

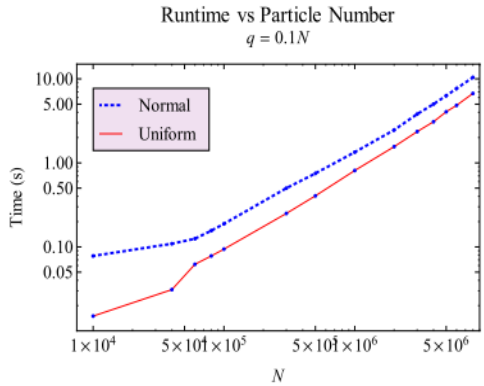

Figure 2: Runime measured for different distributions.

level, respectively. Therefore, the structure is a tree with a branching factor equal to 2^d, where d is the dimensionality.

Each particle is identified by its coordinates (x, y, z in 3D) and belongs to different boxes at different levels. In case of 3D, each box in the octree is uniquely identified by its universal index (n, l) where n and l are the Morton index and the level of the box, respectively [1]. The universal index of the root box is therefore (0, 0). The neighbourhood of a given box, B, is the set of boxes that shares at least one common boundary point and is at the same level as B (the set includes box B itself). In 2D, the neighbourhood may contain up to 9 boxes and up to 27 boxes in 3D. In addition, each box has an interaction list (up to 27 or 189 boxes in 2D or 3D, respectively), which consists of the children boxes of the neighbours of its parent box, but not its own neighbours. Using the technique of bit interleaving and deinterleaving of particle coordinates, one can precisely determine the index of the box at a given level [1]. The algorithm for space partitioning has four steps.

- Step1: Divide the root box into 8 cubes (or into 4 squares in 2D) as in non-adaptive FMM until $l = 2$.
- Step 2: Count the number of source points, N_n, in the neighborhood of each leaf node created in step 1.
- Step 3: If $N_n \leq q$, tag these boxes as (n, 2) boxes and stop further division. Otherwise, divide them one more time and create (n, 3) boxes.
- Step 4: Continue step 2 and 3 on newly created boxes in step 3 until all leaf nodes boxes have $N_n \leq q$.

The process of subdivision may result in empty boxes which do not contain any source particles or target particles: these boxes need to be discarded.

The above four steps result in the set of target boxes and parents of these target boxes build the D-tree, where each level l of the D-tree contains a set of boxes (n, l) whose $N_n \leq q$. In the D-tree, all target boxes represent leaves or ends for each branch of the tree. This D-tree is used in the downward pass of the adaptive FMM algorithm. The boxes in the interaction list of the each node in the D-tree containing source particles create leaves or nodes of a C-tree. The C-tree is built by traversing through these nodes bottom-up and finding their parent-child relationships. The collection of such trees is the C-forest [1]. This C-forest is used in the upward pass of the adaptive FMM algorithm [1]. It is worthwhile to notice that the number of nodes of the C-forest (at the finest level) is considerably smaller than the particle number, N. Since the level of the box of a given target is determined by both target and source sets, this algorithm is fully adaptive.

PERFORMANCE ANALYSIS

Time

The measured time or runtime consists of two parts: overhead and space partitioning. Even though both times increase with the particle number, the overhead associated with the management of its data, which is relatively small and needs to be applied once for a given set of data, dominates for fewer number of particles. As the particle number increases the time taken for space partitioning dominates and the runtime becomes directly proportional to N, if N is large enough.

For a given q, the runtime increases linearly with the particle number, N, (Fig. 1). Also, for a given N, the runtime linearly increases as q decreases. The runtime depends on the type of particle distribution as well. The runtime of a normal distribution, for a given q and N, is higher than that of a uniform distribution (Fig. 2). The difference among runtimes, however, for different q diminishes as N increases irrespective of the distribution. Normally distributed sources and targets contain highly clustered spaces, which require deeper subdivisions and create additional levels to achieve the user specified value, q. These additional levels contribute significantly to the computational costs (runtime and memory).

This code was tested for data ranging from ten thousand to 8 million particles (generated from normal and uniform distributions) with different q values. Furthermore, it was tested in 2D and 3D data sets. For simplicity, we have shown the results for 3D and normal distribution.

Memory

Peak runtime-memory usage also shows (Fig. 3) the same trend as time. Memory entails some overhead (storing the min/max value of the particle coordinates etc.

ISBN 978-3-95450-115-1

within the octree data structure) inherited to any size of particle number. As the particle number becomes larger,

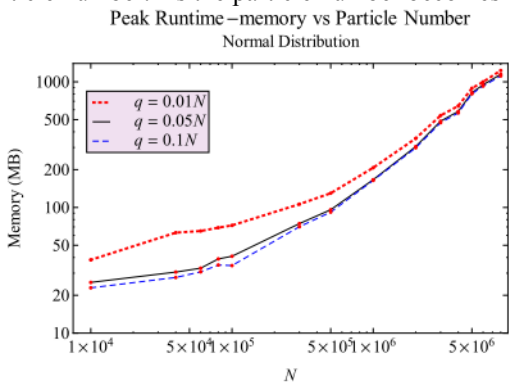

Figure 3: Variation of peak runtime-memory (measured using Valgrind 3.7.0) with particle number, *N*.

however, memory associated with overhead becomes less dominant.

As the number of particles increases, generated from normal distribution as well as uniform distribution, the recursive hierarchical subdivision creates more and more nodes, which requires more memory to store them.

Table 1: Runtime measured for normally distributed particles in 3D

Particle Number(*N*)	Time(s) $q = 0.01N$	$q = 0.05N$	$q = 0.1N$
$1x10^4$	0.389	0.109	0.078
$4x10^4$	0.702	0.171	0.109
$6x10^4$	0.781	0.187	0.125
$8x10^4$	0.842	0.234	0.156
$1x10^5$	0.921	0.265	0.188
$3x10^5$	1.404	0.546	0.500
$5x10^5$	1.763	0.843	0.749
$1x10^6$	2.653	1.575	1.341
$2x10^6$	4.025	2.995	2.465
$3x10^6$	5.787	4.227	3.807
$4x10^6$	7.410	5.616	5.007
$5x10^6$	9.765	6.988	6.302
$6x10^6$	11.591	8.487	7.676
$8x10^6$	15.912	12.183	10.404

Table 2: Peak runtime-memory measured for normally distributed particles in 3D

Particle Number(*N*)	Memory(MB) $q = 0.01N$	$q = 0.05N$	$q = 0.1N$
$1x10^4$	38.27	25.32	22.93
$4x10^4$	63.00	30.60	27.72
$6x10^4$	64.96	32.86	30.74
$8x10^4$	69.04	38.95	34.82
$1x10^5$	71.98	40.88	34.47
$3x10^5$	106.20	74.38	70.25
$5x10^5$	129.80	95.91	91.78
$1x10^6$	207.10	166.30	164.30
$2x10^6$	354.30	305.20	298.20
$3x10^6$	539.10	486.00	472.90
$4x10^6$	642.40	581.30	565.10
$5x10^6$	888.40	819.30	802.40
$6x10^6$	999.00	947.00	924.40
$8x10^6$	1230.00	1148.00	1113.00

Table 1 and 2 summarize the runtime and peak runtime-memory measured, respectively, for three different q values (identical sources and targets generated from normal distribution).

The runtimes are machine dependent. All the tests were run on a single core Intel® Core™ i5-2410M @ (2.30GHz) computer (no SSE). The machine had 8GB of RAM. The code is compiled under Cygwin on Wndows 7 and used the g++ compiler with optimization flags –O3 and –funroll-loops.

SUMMARY

In this paper, we proposed using the octree data structure to organize large amounts of data for an adaptive FMM algorithm with both uniform and non-uniform particle distributions. We chose the octree for its efficient representation of a 3D space and the straightforward error estimation it provides. It also works well for independent source and target sets, and is adaptive with respect to sources and targets which lead to minimal FMM computation. Our code was implemented in C++ for portability and used dynamic memory allocation to make it more adaptable. Analysis of peak runtime-memory usage and runtime was performed using arbitrarily generated source and target points (in 2D and 3D). If the particle number, *N*, is large enough, both memory and runtime showed linear dependence or O(*N*) scaling of computational effort. For a given q, normally distributed sources and targets sets or non-uniform particle distributions would require high refinement levels and take more time as well as memory compared to those of uniformly distributed data sets. For uniformly distributed identical sources and targets, the adaptive FMM reduces to regular FMM.

In future, we plan to use the octree data structure to implement the adaptive FMM and study electron cooling.

ACKNOWLEDGEMENT

This work was supported by the U.S. Department of Energy, Office of Nuclear Physics, under Contract No. DE-SC0005823.

REFERENCES

[1] Nail A. Gumerov and Ramani Duraiswami, *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*, (College Park: Elsevier, 2004).

[2] J. Carrier, L.Greengard and V. Rokhlin, A Fast Adaptive Multipole Algorithm for Particle Simulations. SIAM Stat. Comput., Vol. 9, p. 669-686(1988).

[3] H.Cheng, L. Greengard and V. Rokhlin, A Fast Adaptive Multipole Algorithm in Three Dimensions. J. Comput. Phys., Vol. 155, p. 448-498 (1999).

[4] H. Zhang and M. Berz, The Fast Multipole Method in Differential Algebra Framework. Nuclear Instruments and Methods, Vol. 645, p. 338-344 (2011).