Fast Multipole Method

Mitchell Smith, Jacob Thurman, Josh Belsterling

Introduction to Algorithms

November 27th, 2018

**Problem Description**

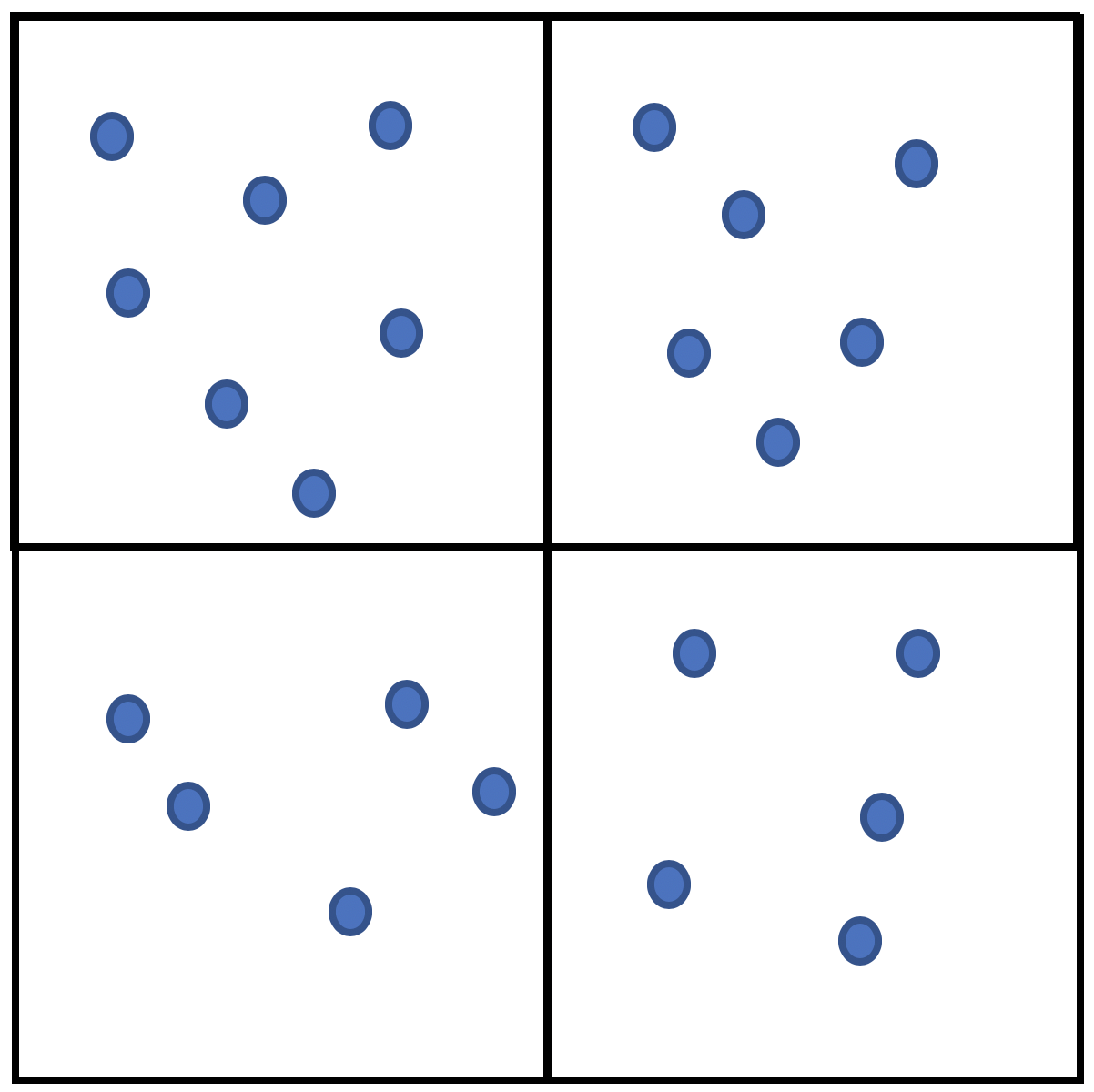
In physics simulations, mostly in the context of space, there can be several objects applying long-distance forces (like gravity) on each other. Calculating how the forces from each object affects the other objects iteratively will take Θ(n2) time, where n is the number of objects being considered (our straightforward solution). For some datasets the Fast Multipole Method can calculate the value on average in O(n) or O(n lg(n)) time [1]. In addition to space applications in physics there are applications of the Fast Multipole Method in electromagnetism problems, quantum chemistry through density functional theory, and many other matrix multiplication problems [2]. We will restrict our discussion to the Fast Multipole Method’s Barnes-Hut N body problem physics implementation [3].

Our team chose the Fast Multipole Method for our project because we are intrigued by outer space, and almost any problem with applications in space would pique our interest. So, we chose the Fast Multipole Method’s application to calculating forces in outer space. We were also impressed by its incredibly low margin of error for calculating matrix products. Additionally, while it would take O(n2) time to calculate an exact answer (such as by using our straightforward implementation), Fast Multipole Method’s provided impressive time savings.

Assume there are n items all within a space. Each item contains forces which interact with the other objects in that space. To calculate the exact resulting forces is the N-body problem, and standard methods would require all forces between each particle be calculated individually. This leads to an O(n2) operation, since n objects, which are each compared against n other objects, result in n\*n, or n2 total calculations. Suppose your computer can complete 106 numeric summations per minute, and you have a data size of 106. Then, an O(n) algorithm will finish in a few minutes while an O(n2) algorithm will take almost two years [1]. This runtime difference more than compensates for the slight inaccuracies of the Fast Multipole Method.

**Algorithm Descriptions**

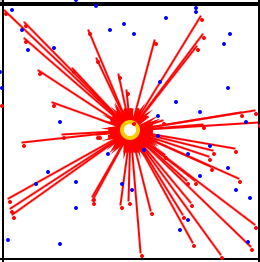
The Fast Multipole Method is used to make low-rank approximation of a matrix-vector product in such an efficient way that it is possible to bound the margin of error to that of machine error in floating-point approximation [4]. The N-body problem is one such example that we can represent as a matrix-vector product.

The method starts by looking at the whole set of *N* points within a given square box. This is considered to be the root of a tree, so this node and each of the other non-leaf nodes contain exactly 4 children. Each of these four children is a square the represents one quarter of its parent. This is performed until we have a tree of height O[log(n)], which we will call levels, when the whole surface is level 0. We can also say that for some level *k*, there are *4k* squares in that level and each of those squares contains O[N/(4k)]. Once we have the tree we can begin to process it.

*Figure 1: A parent node (the larger square) divided into 4 children nodes (the smaller squares).*

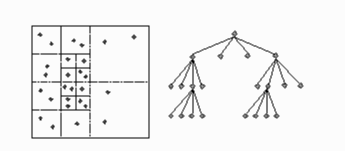
The higher rank approximation we use, the higher the accuracy for values close to the target value. As a dynamic programming solution to the problem, the algorithm can traverse through larger blocks toward smaller ones, yielding better approximations by calculating new values but taking memoized calculations from the upper levels to avoid needing to recalculate data. The first level of approximation (a rank 2 approximation), as partially visualized in Figure 2, is computed by calculating the total mass of all of the particles applying forces for the block. We use this as an approximation of the total force for each individual element in the block to calculate its effect on particles in other blocks. Then, the algorithm considers blocks at least 1 block away from the primary block and calculates their potential. Then, the multi-level Fast Multipole Method continues down to the next level and repeats the previous steps, but where larger blocks were already calculated we continue to use those values as we move down towards smaller and smaller blocks (the dynamic programming portion of this solution). Thus, when the algorithm arrives at the highest rank approximation, it considers all values by calculating those at least 1 block away, reusing as many calculations as possible.

Another option is to use a predetermined number of divisions. As an optimization to improve accuracy and efficiency we use a formula of 2(log N +1), where N is the number of particles, for the number of boxes we compute in each of these two levels rather traversing recursively through multiple levels. This approach allows the user to prioritize accuracy or speed. Our formula balances both speed and accuracy.



*Figure 2: The algorithm sums the mass for all of the particles in the block to the center [2].*

For real datasets, there are often clusters of data points that sit close together and are a substantial distance away from the object being considered. Therefore, if you use a specific version of the multipole method, the difference between the approximate angle of the force determined by the fast multipole method and the angle produced by a brute force calculation is so small that it is covered by the range of floating-point rounding error [4]. If we can bound these data clusters into a grid of approximately equal (quasi-random) distribution, we can use this grid to perform the above-mentioned process, yielding substantial improvements in execution time. Even if there are not clusters of data, we are still able to get results within an acceptable margin of error when the block of forces acting on the block being considered are at least the width of one block away from each other. Performing this operation on increasingly large ranks produces increasingly accurate answers.

*Figure 3: demonstrates breaking the overall set down into ranks to begin to isolate data points, and the tree that accompanies it as part of the multi-level Barnes-Hut algorithm [3] [5].*

Contrast this with the brute force algorithm, which considers each point’s influence on each other point directly. This takes O(n2) time, as will be discussed below.

**Pseudocode**

**Our Kernel Function**

This function will be used in both of the main algorithms as the calculation for the gravitational pull of one particle on another. This is the kernel function of the problem because you need to add all of the vectors together to produce the net force. This will return the force of particle b on a.

1 G(particle a, particle b)

2 force = <b.x, b.y> - <a.x, b.x>

3 force /= Sqrt((force.x^2) + (force.y^2)) //Normalize

4 Return force \* (G\_const \* a.mass \* b.mass) / ((a.x + b.x)^2 + (a.y + b.y)^2)

*Figure 5: Our kernel function G. G(a, b) is used to calculate the force of b on a.*

**Straightforward Version:**

Given:

An array A that contains N particles that have dimensional positions x and y, and a mass

x and y are values between 0.0 and 1.0. The gravitational constant G\_const. The kernel function G from Figure 4. This solution returns an array mapping the indices of A to the net force applied to each particle by every other particle.

1 ApplyFieldForces(A)

2 netForces := [1...A.size]

3 For v from 1 to A.size:  
4 netForces[v] := <0,0>

5 For i from 1 to A.size:

6 For j from i + 1 to A.size:

7 diff\_x := A[j].x - A[i].x

8 diff\_y := A[j].y - A[i].y

9 distanceSquared := diff\_x^2 + diff\_y^2

10 kernalResult := G(A[i], A[j])

11 netForces[i] += kernalResult

12 netForces[j] += -1 \* kernalResult

13 Return netForces

*Figure 6: The straightforward, brute force algorithm for finding the net forces on each object.*

**CLEVER VERSION:**

Given:

An array A that contains N particles that have a dimensional positions x and y, and a mass

x and y are values between 0.0 and 1.0

The gravitational constant G\_const

The kernel function G from Figure 3

1 ApplyFieldForces(A)

2 Int DIVISIONS= something user-defined

3 Float SQUARE\_LEN = 1/DIVISIONS

4 New Array of Particle CenterOfGravityPoints[DIVISIONS][DIVISIONS]

5 For i from 0 to DIVISIONS:

6 For j from 0 to DIVISIONS:

7 X = SQUARE\_LEN/2 + (SQUARE\_LEN \* i)

8 Y = SQUARE\_LEN/2 + (SQUARE\_LEN \* j)

9 CenterOfGravityPoints[i][j].add(new Particle(X, Y, 0))

10 New List of Particle Square[DIVISIONS][DIVISIONS]

11 For i from 1 to A.size:

12 X = (Int) Floor(A[i].x / SQUARE\_LEN)

13 Y = (Int) Floor(A[i].y / SQUARE\_LEN)

14 CenterOfGravityPoints[X][Y].mass += A[i].mass

15 Square[X][Y].add(A[i]) // Add by reference, unless we are returning new array

16 New Array of Float PotentialApprox[DIVISIONS][DIVISIONS]

// Pseudocode is not iterated for arrays

17 For Approx\_x from 1 to DIVISIONS:

18 For Approx\_y from 1 to DIVISIONS:

19 For Square\_x from 1 to DIVISIONS:

20 For Square\_y from 1 to DIVISIONS:

21 if(Square\_x - Approx\_x > 1 || Square\_x - Approx\_x < -1 || Square\_y - Approx\_y > 1 || Square\_y - Approx\_y < -1)

22 PotentialApprox[Approx\_x][Approx\_y] +=

23 G(CenterOfGravityPoints[Square\_x][Square\_y],

CenterOfGravityPoints[Approx\_x][Approx\_y])

24 For x from 1 to DIVISIONS:

25 For y from 1 to DIVISIONS:

26 For i from 1 to Square[x][y].size:

27 Square[x][y].get(i).netForce := PotentialApprox[x][y]

// Final summation - Begin with external squares

28 For x from 1 to DIVISIONS:

29 For y from 1 to DIVISIONS:

// Special iteration - See section below pseudocode for explanation

30 Int neighbor\_x, neighbor\_y = 1

31 for(neighbor\_x = -1; neighbor\_x < 2; neighbor\_x++)

32 If(x + neighbor\_x > 0 && y + neighbor\_y < DIVISIONS+1 && x + neighbor\_x < DIVISIONS+1):

33 For i from 1 to Square[x][y].size:

34 For j from 1 to Square[x + neighbor\_x][y + neighbor\_y].size:

35 sum := G(Square[x][y].get(i),Square[x + neighbor\_x][y + neighbor\_y].get(j))

36 Square[x][y].get(i).netForce += sum

37 Square[x + neighbor\_x][y + neighbor\_y].get(j).netForce -= sum

38 neighbor\_x = 1

39 neighbor\_y = 0

40 If(x + neighbor\_x > 0 && y + neighbor\_y < DIVISIONS+1 && x + neighbor\_x < DIVISIONS+1):

41 For i from 1 to Square[x][y].size:

42 For j from 1 to Square[x + neighbor\_x][y + neighbor\_y].size:

43 sum := G(Square[x][y].get(i),Square[x + neighbor\_x][y + neighbor\_y].get(j))

44 Square[x][y].get(i).netForce += sum

45 Square[x + neighbor\_x][y + neighbor\_y].get(j).netForce -= sum

// Final summation - finish with internal square forces

46 For x from 1 to DIVISIONS:

47 For y from 1 to DIVISIONS:

48 For i from 1 to Square[x][y].size:

49 For j from i + 1 to Square[x][y].size:

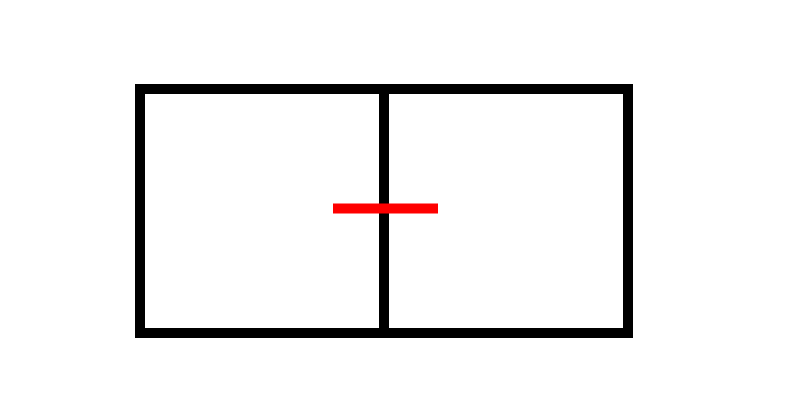
50 Sum := G(Square[x][y].get(i),Square[x][y].get(j))

51 Square[x][y].netForce += sum

*Figure 7: The much longer pseudocode for calculating, with a little error, the gravitational net forces on particles in a much shorter amount of time. This code allows the user to vary the order of the approximation in a Single-Level Barnes-Hut manner [3]. This pseudocode is not iterated.*

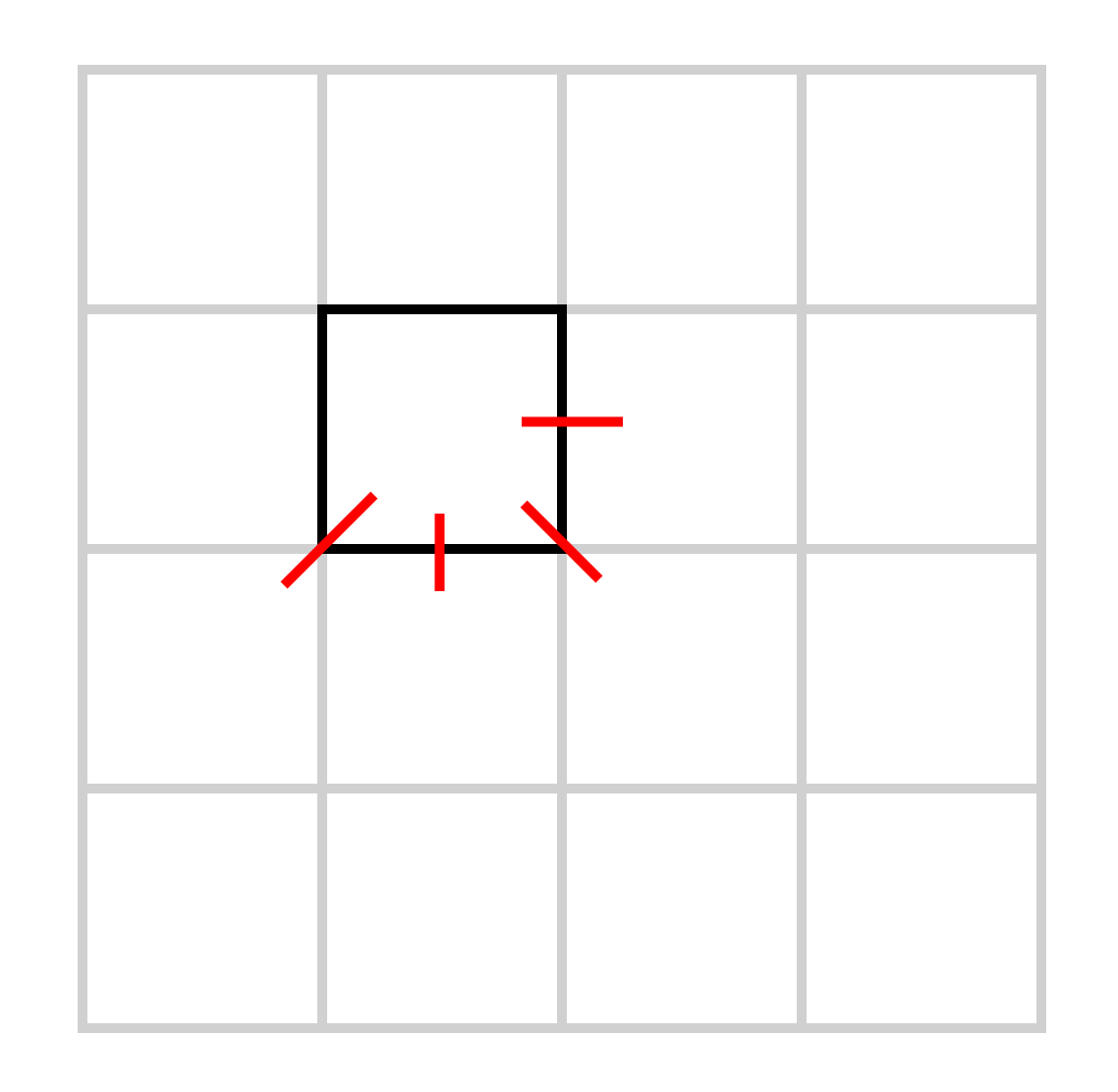
**“Special Iteration”**

When calculating the gravitational force between two particles, the force applied on each particle is equal, just in opposite directions. The special iteration in our code takes advantage of this so that there are no double calculations, therefore speeding up the algorithm. Not double-calculating within a square is easy, just have the second loop on line 48 start one particle ahead in the array relative to the previous loop and go to the end. When it comes to calculating the forces between particles in different squares, this method can no longer be used as every particle relative to another in the other neighboring square must be considered. The way to not double calculate is to calculate the individual forces between two neighboring squares, adding the force to both particles at the same time, and then not return to it. Let’s visually define a square that has had forces calculated as the following, a red line between two boxes:



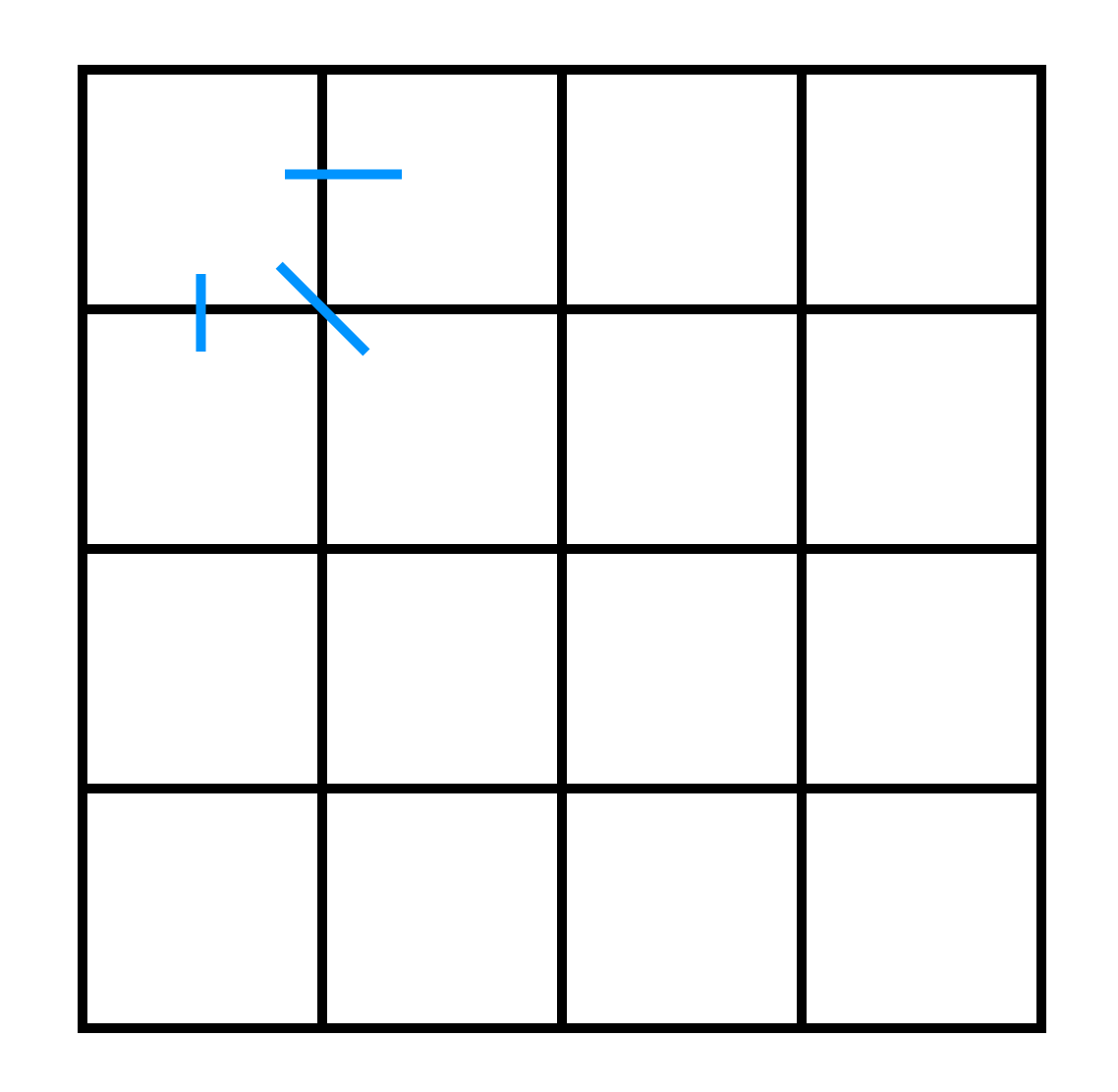
*Figure 8: A simple visual for when the forces of particles between the two squares have been calculated*

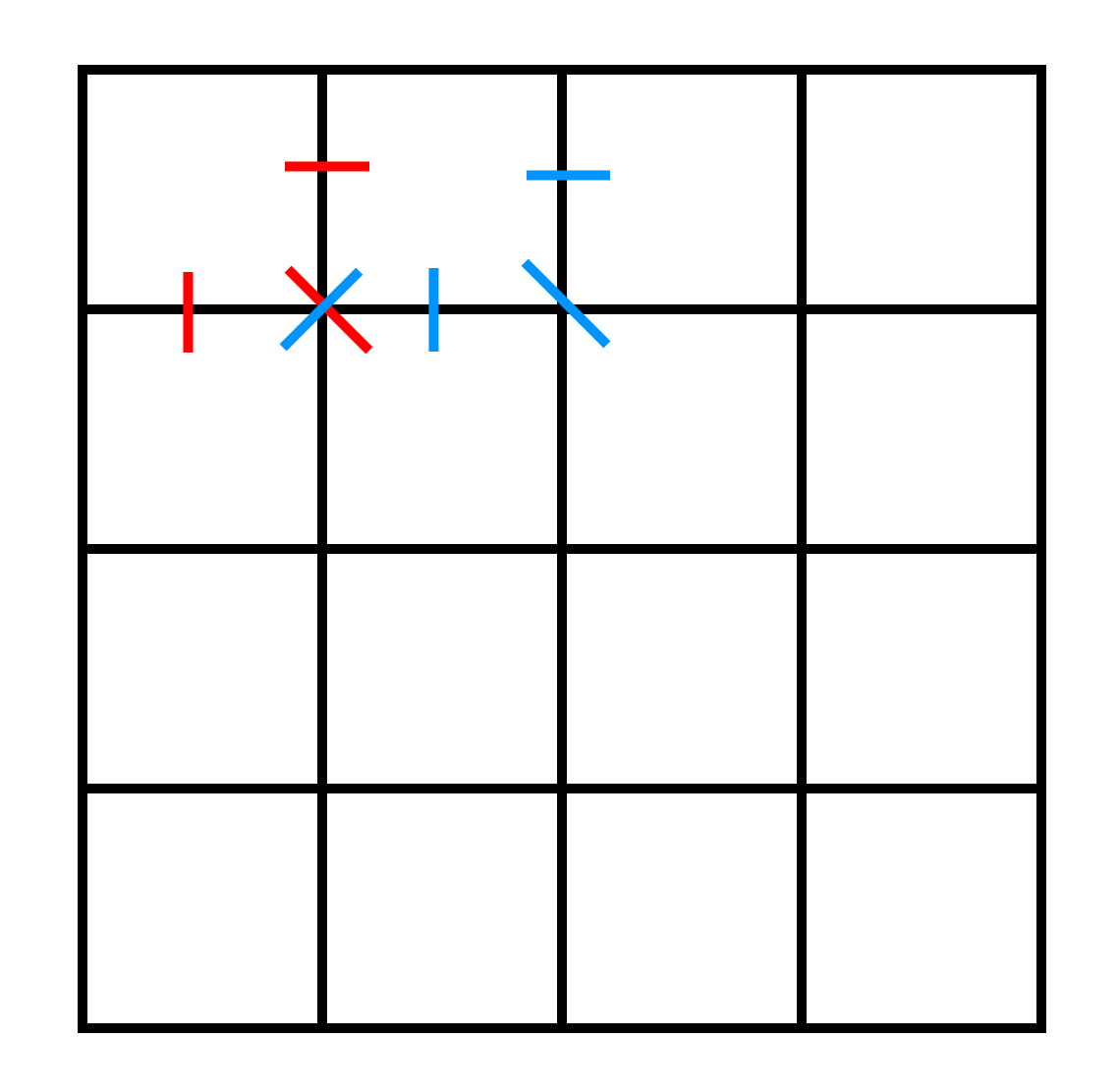
In order to not have any overlap while iterating through the squares, we need to choose a direction that cannot be overlapped. This is our current directions we calculate for:

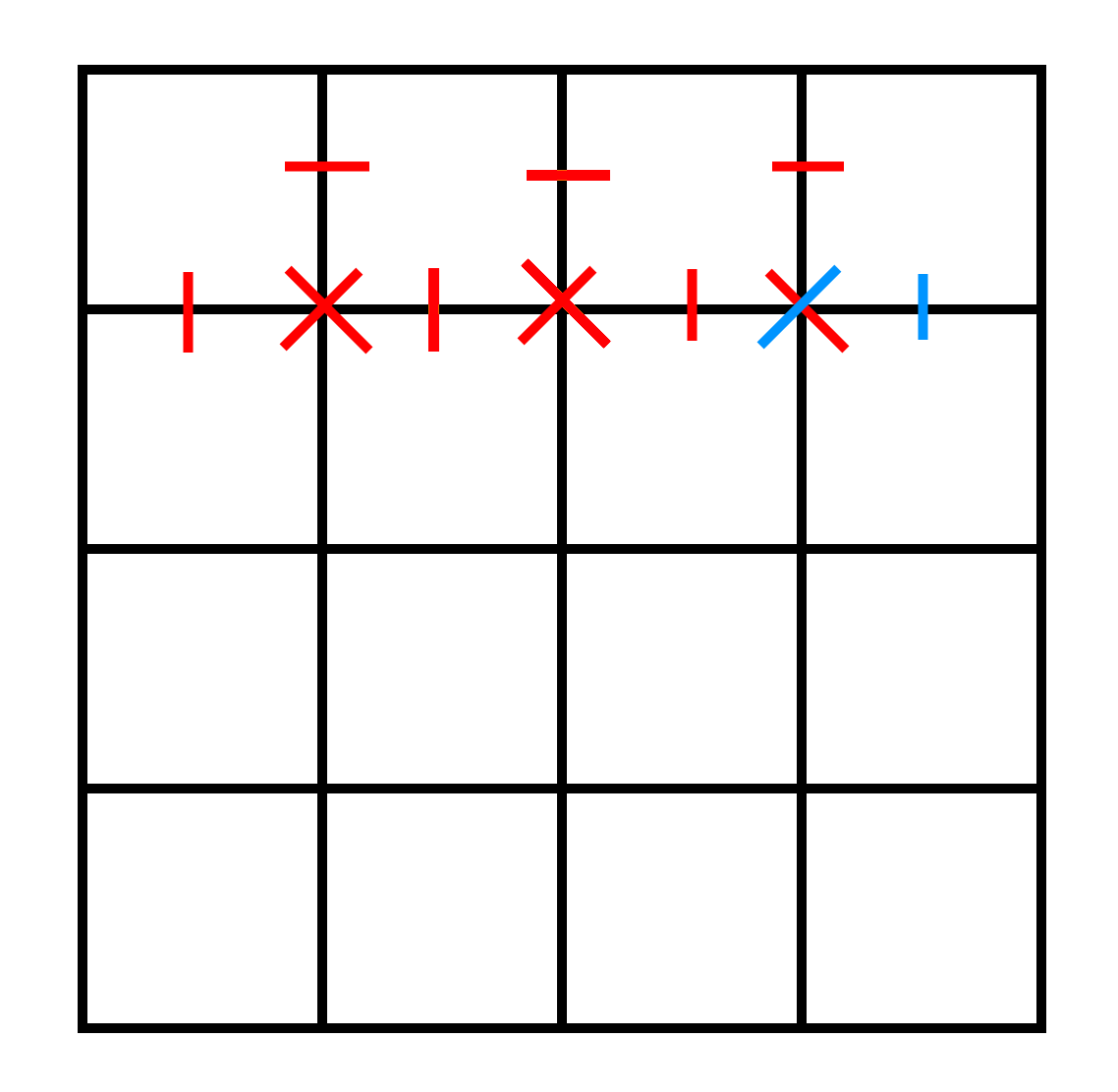


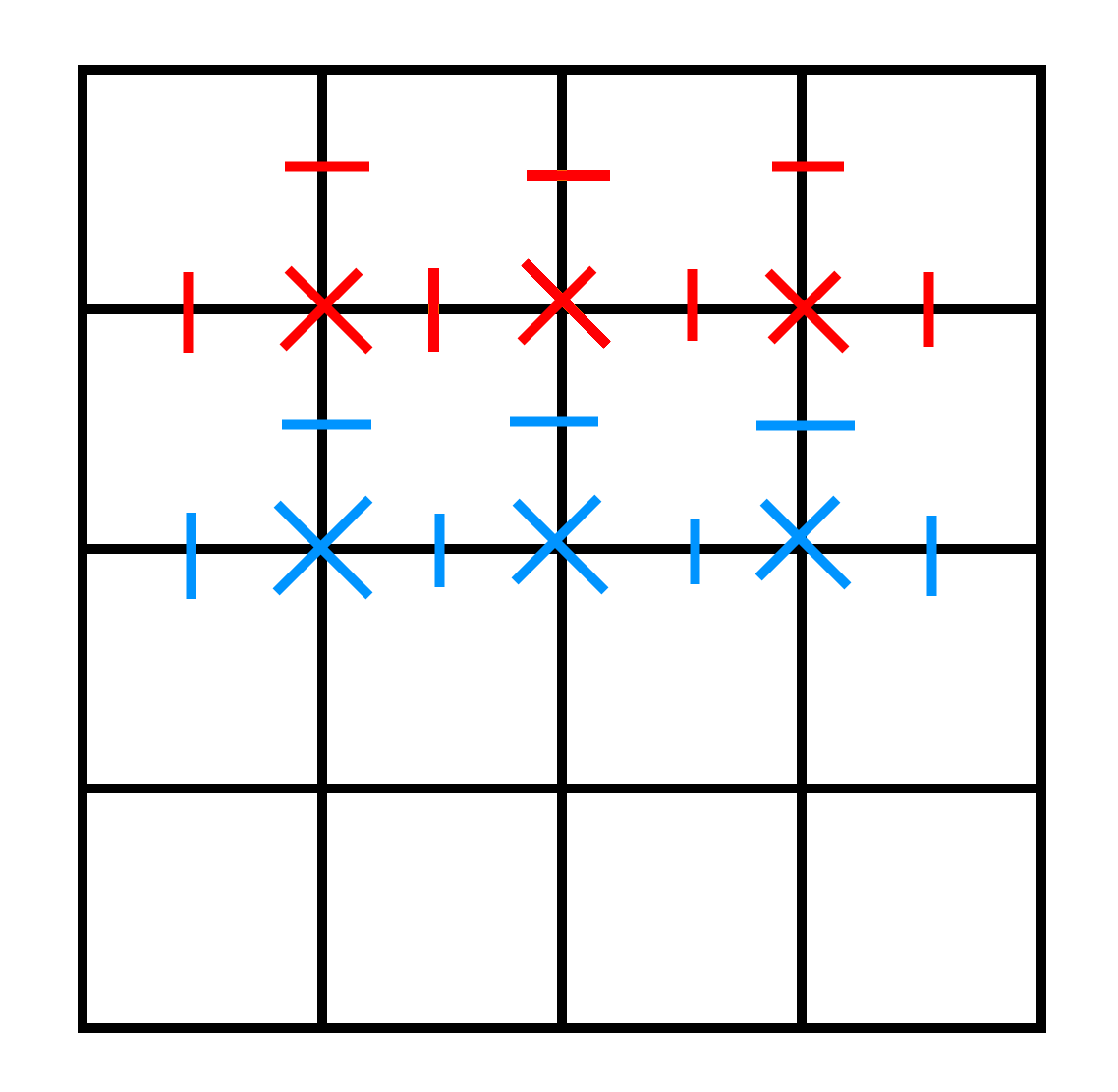
*Figure 9: A representation of where we calculate neighboring particle forces per square iteration.*

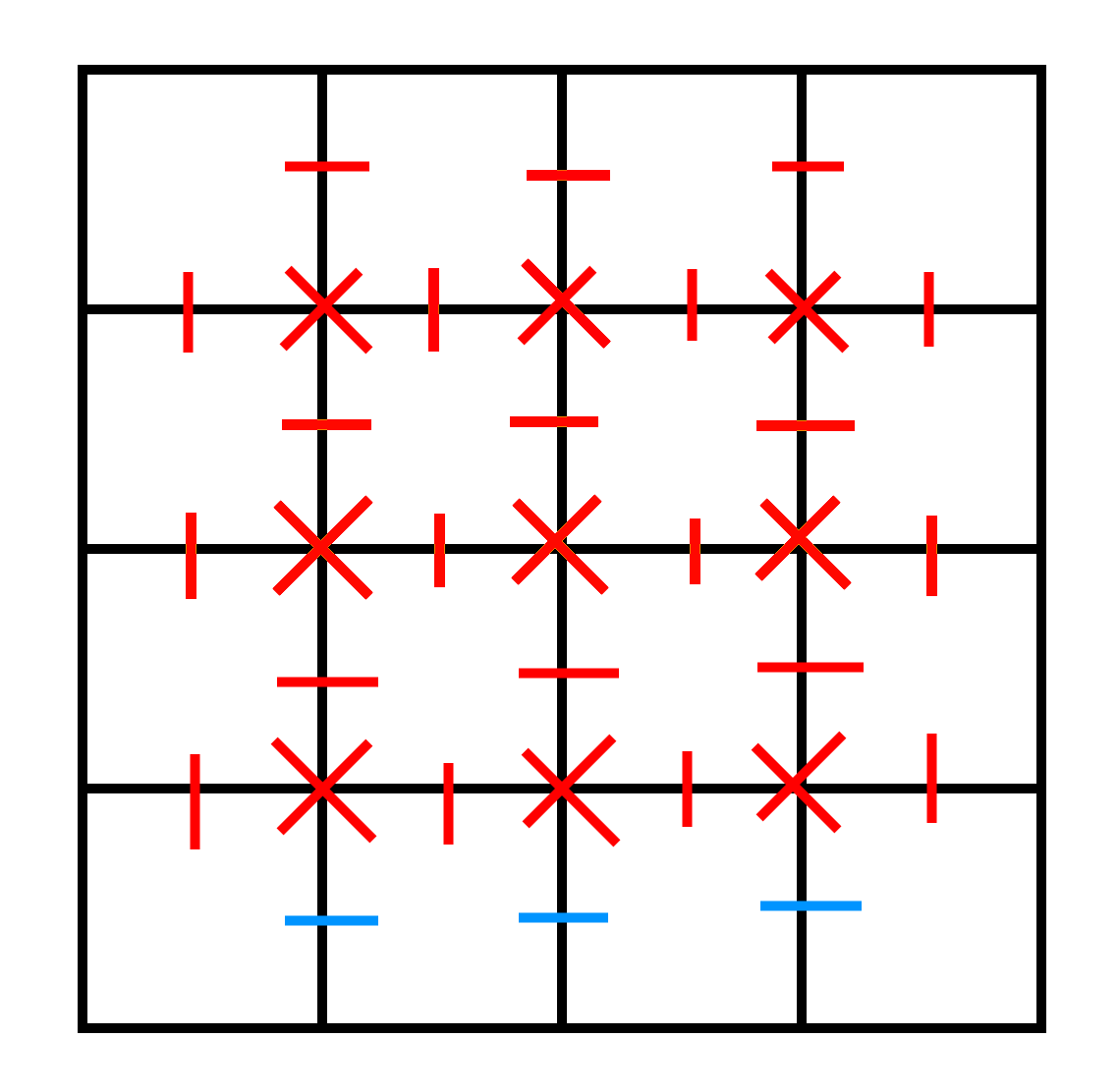
The three squares on the bottom are calculated on lines 31 through 37 with a simple for loop, then the west square is calculated on lines 40 to 45. Obviously there is a check needed to make sure a square exists where we are calculating first, and those are on lines 32 and 40. All of this said, a simple visual proof should show this works.











*Figure 10: A short, iterative visual representation of how our special iteration creates a grid that fully covers all neighborly interactions and allowing for one pass only through those interactions. Blue represents new iterations. Starting with the top grid with one iteration, the second grid being the second iteration, then the third being the last iteration for that grid. Using the third grid and knowing there is no checks to the north, we can iterate through the next row to get the fourth grid. And this shows there’s still no overlap, and that rows can keep being kept added. The only difference will be the last row, where only the westward neighbors are calculated, resulting in all of the needed interactions being computed.*

**Algorithm Analysis**

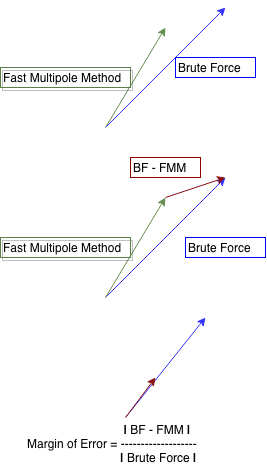
From the above pseudocode we can see the brute force algorithm (the “straightforward algorithm” is clearly a Θ (N2) operation, where n is the number of particles being considered. The double nested for loops on lines 5 and 6 both run through each particle, meaning each particle is considered N times, creating N2 total considerations. The above loop on line 3 only runs once per particle, meaning it performs an order of N operation, which is inconsequential due to the N2 term from the for loops on lines 5 and 6. Since there remaining operations not discussed here are constant time operations, we can safely bound the brute force algorithm at Θ (N2).

Determining the runtime of our “clever algorithm” is much more difficult. As lines 17-20 show, the bulk of this algorithm’s work is checking each block’s center of mass against that of the other blocks. Doing so requires an order of (M1/2)4 operation where M is the number of smaller blocks the total area is broken into, since each loop iterates over one side of the box (√M) and there are 4 loops. In our implementation, we set M = (2(log N +1))2, so runtime for our implementation is Θ (24 log(N))/log(10) + 4). The other lines all work on smaller datasets (like lines 5 and 6), so they are dominated by the above term.

*Table 1: An example of a single run of each data size collected. Both the margin of error and the runtimes improve with the scale of the data. The margin of error is calculated as a percentage of magnitudes between the true, Brute Force result and the difference vector (Brute Force minus Fast Multipole Method) as shown in Figure 9 below.*

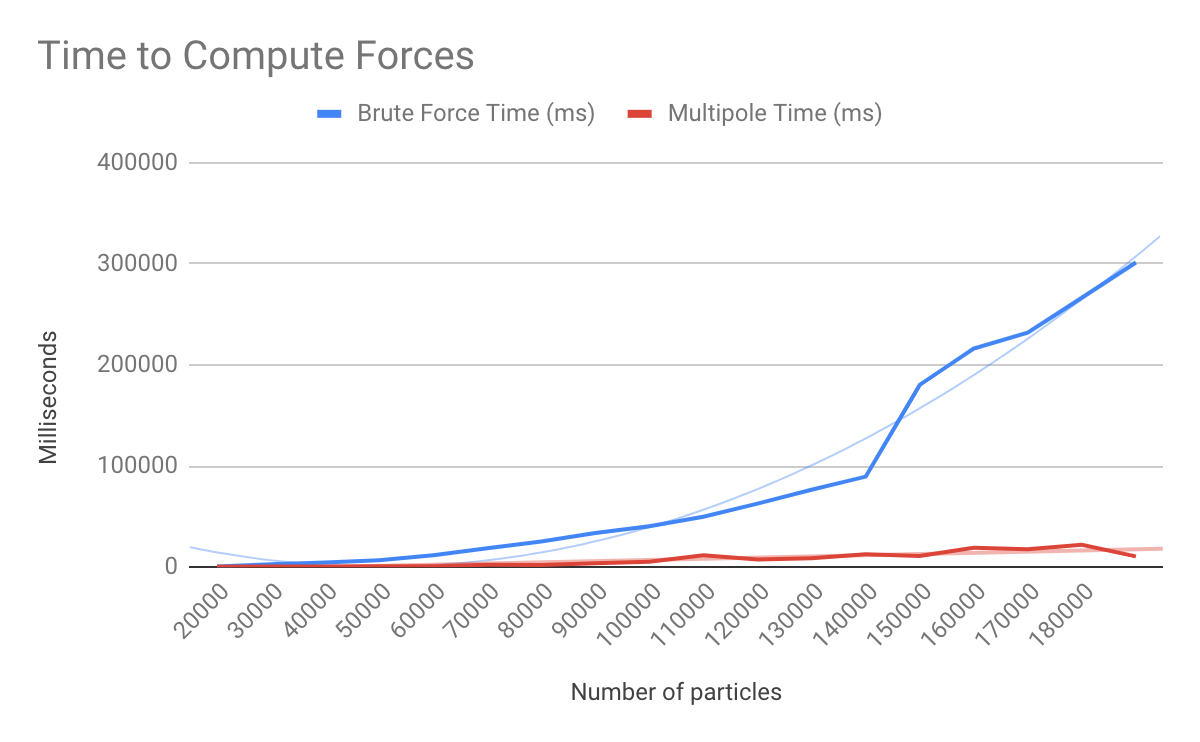
|  |  |  |  |
| --- | --- | --- | --- |
| Data Size | Brute Force Time (ms) | Multipole Time (ms) | Average Error (%) |
| 10000 | 678 | 399 | 3.651306883 |
| 20000 | 3044 | 728 | 3.648961269 |
| 30000 | 4723 | 781 | 3.650131895 |
| 40000 | 6985 | 896 | 3.65001422 |
| 50000 | 11888 | 1026 | 3.649961301 |
| 60000 | 18822 | 2538 | 3.650653231 |
| 70000 | 25544 | 2159 | 3.650433315 |
| 80000 | 33814 | 4029 | 3.650308999 |
| 90000 | 40465 | 5428 | 3.650572968 |
| 100000 | 49882 | 11777 | 1.935306392 |
| 110000 | 62918 | 7651 | 4.04589204 |
| 120000 | 76650 | 8924 | 4.033589583 |
| 130000 | 89608 | 12740 | 3.954383357 |
| 140000 | 180247 | 11212 | 4.638572761 |
| 150000 | 216093 | 19146 | 1.640164924 |
| 160000 | 231972 | 17821 | 3.76956412 |
| 170000 | 266376 | 22229 | 3.792755876 |
| 180000 | 301178 | 10744 | 1.576891041 |

As we expected, the Fast Multipole Method’s approximation of the values is significantly faster than brute force evaluation, particularly as the data size scales. We tested our data using Quasi-Random values from a Holton sequence, which we obtained from an open source apache project [6]. Because real data sources for this problem, such as star positioning, tend to replicate quasi random data, it is understandable that the Fast Multipole Method is designed to work best with quasi random data.

**

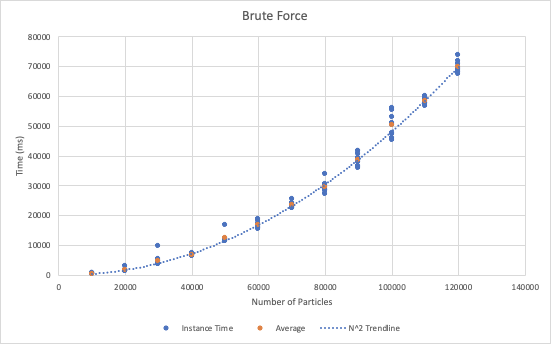
*Figure 11: This diagram explains the way we calculate margin of error. We first take the Brute Force (BF) and Fast Multipole Method vector results and then find the difference. We consider margin of error to be the magnitude of the difference over the magnitude of Brute Force.*

Our implementation uses a fixed number of divisions, rather than a multi-level approximation. In an attempt to gain machine level accuracy, the algorithm will often recursively estimate four quarters of the box it is looking at, and then use those quarters to improve the estimate. As stated earlier, our implementation uses a fixed number of levels for the approximation. As stated earlier, we use a formula of 2(log N +1), where N is the number of particles, to calculate the number of boxes we compute in each of these two levels rather than starting at a rank 2 approximation and working downward. This was empirically determined to have the best performance in a large number of cases. It optimizes for performance and improves margin of error. However since it scales with the number of particles and a larger number of boxes will produce more accurate results, the larger the N value, the better the accuracy.

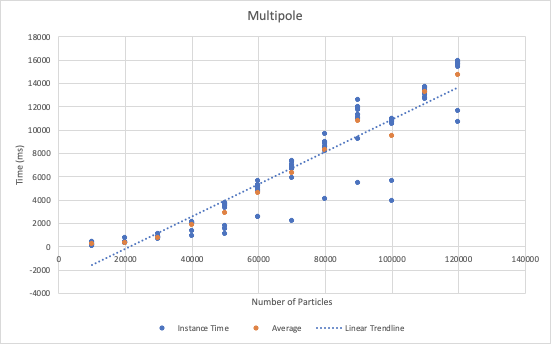


*Figure 12: This graph displays the runtimes to computer the forces using both Brute Force and the Fast Multipole Method. Multiple grows in linear time while brute force grows in N2 time.*

As Figure 12 displays, empirical analysis of the algorithms confirm that Brute Force runs around O(N2) time and some versions of the Fast Multipole Method run in about O(N) time. There are shadowed trend lines to help make this clearer. The Fast Multipole Method really shines for high N values.



*Figure 13: This graph displays the time to calculate forces using the Brute Force method alone across 10 runs for each of 12 quasi random data sets of sizes 1k, 2k, … up to 12k. The orange dots show the average of the points for the given data set. The dashed line is the expected trendline of N2 time.*

**

*Figure 14: This graph displays the time to calculate forces using the Multipole method alone across 10 runs for each of 12 quasi random data sets of sizes 1k, 2k, … up to 12k. The orange dots show the average of the points for the given data set. The dashed line is the expected trendline of linear time.*

**Sources**

[1] L. Chen. (2015, Oct. 7). *Introduction to Fast Multipole Methods* [Online]. Available:

<https://www.math.uci.edu/~chenlong/MathPKU/FMMsimple.pdf>

[2] Wikipedia. (2018, Sept. 14). *Fast multipole method* [Online]. Available:

<https://en.wikipedia.org/wiki/Fast_multipole_method>

[3] Gunnar Martinsson. (2014, June 23). *Lecture 2: The Fast Multipole Method* [Online].

Available: <http://amath.colorado.edu/faculty/martinss/2014_CBMS/Lectures/lecture02.pdf>

[4] Nicole Eikmeier. (2015, Mar. 19). *Intro to FMM - Nicole Eikmeier* [Online]. Available:

<https://www.youtube.com/watch?v=MeGc5VWJAec>

[5] University of Washington Faculty. (1995, July 26). *The Fast Multipole Method* [Online].

Available: <http://faculty.washington.edu/trq/hpcc/faculty/marios/papers/ppl/node7.html>

[6] The Apache Software Foundation. (2004, July 6). *Class HaltonSequenceGenerator*

[Online]. Available:

<http://home.apache.org/~evanward/commons-math-3.6.1-RC1-site/apidocs/index.html>