Parallelization of 2D Fast Multipole Method
Sunli Tang, Yuwei Jiang

1 Introduction

N-body problem plays an important role in many real world problems, including astrophysical simulations, electrodynamics, fluid dynamics and data science. In particular, it is to compute the a sum of form

\[ f(x_i) = \sum_{j=1}^{n} K(x_i, y_j) s(y_j) \]  

for each \( i = 1, 2, ..., n \), where \( K \) is a given (usually smooth away from origin) kernel. For example, for gravitational interactions in 3D, \( K(x, y) = \frac{1}{|x - y|} \); for 2D Laplacian equation, \( K(x, y) = \log |x - y| \). The direct computation has a cost of \( O(n^2) \). In recent decades, many algorithm has been invented and developed to accelerate the quadratic time complexity. Fast multipole method (FMM), first introduced by V. Rokhlin and L. Greengard in 1985\[1\], is probably the most famous and prominent one – it achieves the linear complexity \( O(n) \), with constant factor depends on precision. This fact makes FMM one of the first choice in simulations of large-scale physical problems, and thus, an efficient parallelization of FMM is crucial in many applications. Many papers\[2\][3] have explored different methods/platforms to parallel different versions of FMM. In this project, we will explore the parallelization of 2D Laplacian FMM using OpenMP on a shared memory computing structure. We will first make a brief introduction to the FMM algorithm (section 2); after that we will discuss about the fork-join model (section 3.1) – a natural but clever way to parallel FMM, and then we will explore yet another parallelization of FMM – the interleaving model (section 3.2). Finally we will show some numerical result about the scaling of the parallel FMM (section 4).

The code we use in this project is based on FMM2D Libraries of Courant Mathematics and Computing Laboratory (CMCL). The original version of code is available at [http://www.cims.nyu.edu/cmcl/fmm2dlib/fmm2dlib.html](http://www.cims.nyu.edu/cmcl/fmm2dlib/fmm2dlib.html). Our parallel FMM code is available on the project github [https://github.com/tsl665/fmm2d_omp](https://github.com/tsl665/fmm2d_omp).

2 Fast Multipole Method

Since the main focus of this project is on parallelization of FMM, in this section, we will make a rather concise introduction to the idea of FMM. For a more detailed description of algorithm, one should read \[1\][4][5]. Here we will use some terms consistent with \[4\]: we will call \( y_j \) sources and \( x_i \) targets.

The main idea behind FMM is to deal with near-field effect and far-field effect separately. For near-field sources, we direct calculate its effect; for far-field sources, we represent its effect using low-rank approximation. One natural question here is, why is the far-field effect essentially low rank? A good way to explain this is by examples. Consider the problem of gravitational interactions: a cluster of far-field mass points are not very different from a representative mass points located at the center of the cluster; hence, instead of computing the effect of every mass points, we compute that of every clusters, which is an animal of much lower rank. Another more abstract way to explain the validity of such approximation is by Weiestrass approximation theorem\[7\], given the kernel is smooth away from the origin. In 2D FMM, we apply the knowledge from complex analysis and establish such approximation by Taylor expansion and Laurent expansion truncated at \( p \) terms. In the context of FMM, we call them multipole expansion and local expansion respectively.
Another important fact is, we are able to shift between expansions of different scales. This is equivalent to the fact that we can shift the center of Laurent or Taylor expansions under certain conditions.

A traditional non-parallelized FMM scheme is, first adaptively dissect the domain into boxes of different levels, and construct a quadtree (2D) or an orctree (3D); second, starting from leaf nodes, construct multipole expansions from sources, and construct multipole expansions of higher level by a upward pass from leaf nodes; third, construct local expansions from a downward pass, and at leaf nodes, evaluate at each targets; finally, add the effect of near-field points.

Notice that, in above algorithm, every expansion is generated from sources or other expansions by different operators. Indeed, in n-body problem, these operators are matrices. Therefore, we can also describe the non-parallelized FMM by these operators.

- **Particle-to-Multipole (P2M):** at the leaf level, source points in a box to any well-separated target point can be approximated by a multipole expansion at the center of the box up to order \( p \);
- **Multipole-to-Multipole (M2M):** the multipole expansion of a non-leaf box can be obtained by multipole expansions of its children boxes;
- **Multipole-to-Local (M2L):** at each level, the effect from a well-separated source box to a target box is obtained by converting the multipole expansion of the source box into a local expansion at the center of the target box;
- **Local-to-Local (L2L):** at each level, the effect of from a neighboring source box is obtained from the local expansion of the parent box of the source and the target;
- **Local-to-Particle (L2P):** after all the M2L and L2L operations, for each box at the leaf level, the overall effect of source points in the well-separated boxes is the local expansion. L2P is evaluating the value of the local expansion at a target point in the box;
- **Particle-to-particle (P2P):** for sources and targets in neighboring leaf boxes, interactions are evaluated directly.

In above operators, only P2P operator is near-field operator, and the other are far-field operators. The far-field operator has strong dependency with each other. All P2M must be done first. M2L must wait for M2M at this level to finish. L2L has to wait for M2L at the previous level to finish. At last, L2P has to wait for all M2L and L2L to finish. The near-field operator P2P is independent with all the far-field operators except that L2P and P2P write to the same memory. An operator version of non-parallelized FMM is described in Algorithm 1. It is worth to mention that, according to analysis in [3], the dominant cost of FMM computation lies in M2L at finest level (58.75%) and P2P (28.75%).

**Algorithm 1: Non-parallelized FMM**

```
1  Construct FMM Quadtree;
2  L ← tree.height;
3  P2P(tree.level[L-1])
4  P2M(tree.level[L-1])
5  for i ← L − 2 to 2 do
6      M2M(tree.level[i]);
7  for i ← 2 to L-2 do
8      M2L(tree.level[i]);
9      L2L(tree.level[i]);
10     M2L(tree.level[L-1]);
11     L2P(tree.level[L-1]);
```
3 Parallelized FMM

As we mentioned in last part, the whole FMM algorithm includes a preprocessing step (construction of FMM tree), and the actual computing. The preprocessing step is, of course, not neglectable and non-trivial. The history of research on parallel tree algorithm can be traced back to 1990's, for example [6] [7]. In this project, we assume the quadtree is given and focus on how to parallel the computing part of FMM using OpenMP.

3.1 Fork-join model

In the non-paralleled FMM scheme, each operator has a for loop traversing all cells at that level. The operations on these cells are independent. Therefore, the most obvious and simple way to parallelize FMM is just to rewrite each independent for loop into `omp parallel for`, and synchronize after each operator. However, this naive way of parallel is not efficient, since each `omp parallel for` has an implicit barrier in the end, this method has 3 barriers at each working level. In particular, a synchronization is necessary after each M2M, M2L, and L2L operator.

However, we observe that there are no dependency among M2L operators at different level. Therefore, instead of doing a level-wisely M2L, L2L operator, a better idea is to do M2L all together, synchronize and then do L2L all together. In this way, we do not need any synchronizing between levels in M2L calculations. Another important observation of dependency of operators is, the near-field operator (P2P) is independent to all far-field operators except L2P. Therefore, we can insert P2P between any two far-field operators, and we do not need synchronization after P2P, as long as we can guarantee it is completed before we start L2P.

In our algorithm, we put it between M2M and M2L to get a better workload balance. We call the algorithm applying the above two observations the Fork-join model. The pseudo-code of this algorithm is in Algorithm 2. In particular, Algorithm 2 only has two implicit barriers at each level.

Algorithm 2: Fork-join model for FMM

```plaintext
1  L ← tree.height;
2  # pragma omp parallel
3     P2M(tree.level[L-1]);
4  for i ← L − 2 to 2 do
5      M2M(tree.level[i]);
6     P2P_nowait(tree.level[L-1]);
7     for i ← 2 to L-1 do
8        M2L_nowait(tree.level[i]);
9     # pragma omp barrier
10    for i ← 2 to L-2 do
11       L2L(tree.level[i]);
12    L2P(tree.level[L-1]);
```

3.2 Interleaving model

The essence of workload balancing is to minimize the waiting time of idle threads. In the fork-join model, we do it by only keeping necessary synchronization. Another way to attack the same problem is that, whenever some threads are waiting, we want them to pick up some other independent work. Interleaving model minimizes the waiting time by the latter approach. We notice that in parallel FMM, most synchronization points are in far-field computation; because P2P is independent to P2M, M2M, M2L and L2L, when some threads are waiting for synchronization, we want them to pick up the near-field work. In OpenMP, we can do this by `omp parallel task`. The interleaving parallel FMM is described in Algorithm 3.
Algorithm 3: Interleaving model for FMM

```
Algorithm main(tree)
  L ← tree.height
  # pragma omp parallel
  # pragma omp sections
    P2P_task(tree.level[L-1])
  # pragma omp section
    P2M_task(tree.level[L-1])
    for i ← L - 2 to 2 do
      M2M_task_nowait(tree.level[i]);
    end
    for i ← 2 to L-1 do
      M2L_nowait(tree.level[i]);
    end
    # pragma taskwait
    for i ← 2 to L-2 do
      L2L_task(tree.level[i]);
    end
  # pragma omp single
    L2P_task(tree.level[L-1]);

Subroutine taskoperator(tree.level[i])
  for j ← box in level i do
    # pragma omp task
    P2P_task(tree.level[L-1])
  end
  # omp taskwait
  return

Subroutine taskoperator_nowait(tree.level[i])
  for j ← box in level i do
    # pragma omp task
    P2P_task(tree.level[L-1])
  end
  return
```
3.3 Chunk Size

![Time vs Chunk Size in Fork-join Model](image)

Figure 1: elapsed time for different chunk size in fork-join model

In the above algorithms, no matter we are using a dynamic *omp parrellel for* or *omp task*, we sacrifice a rather big overhead cost for flexibility. However, if each piece of work is too small, for example, if we set the chunk parameter in *omp parrellel for* to be 1 (or just by default) in Algorithm 2, or construct a *omp task* for each box in Algorithm 3, then the overhead cost will dominate due to huge amount of operations to dynamically assign jobs to different threads. One simple remedy of this is to send a group of several boxes, instead of a single box, to each thread. This modification turns out to be a very important improvement in reality. We can visualize this fact from Figure 1 and Figure 2.

Both figures are generating from an average of 10 tests, each of which runs a FMM computing of 1,000,000 sources/targets. We see that in both model, an optimal chunk size gives a distinguishable improvement in elapsed time.

4 Numerical Result

Here, we will show some numerical result of two versions (fork-join and interleaving) of parallel FMM, with some suitable chunk size. For fork-join model, we use chunk = 100, and for interleaving model, we use chunk = 800. The machine we run our program on has 60 cores.

First, Figure 3 shows the strong scaling of the parallel FMM. The dashed line represents the optimal
scaling; the blue and red plot show the elapsed time of 2D FMM with 1,000,000 sources/targets on different number of cores, using fork-join model and interleaving model respectively. We see that both model, especially the fork-join model, scales pretty good. The change in scalability at the last point might be the consequence of using up all cores in the machine.

Second, Figure 4 shows the weak scaling of our parallel FMM. Similarly, the dashed line represents the optimal scaling – an optimal weak scaled program should keep the same elapsed time for any number of cores, given that the work on each core is fixed. For interleaving model (red line), we see a moderate increase in running time when we increase the number of cores. Although it is not as good as our first imagination, the weak scalability of interleaving model is indeed reasonably good – every time we double the number of cores, it only has a 11% increase in time. A more exciting result is the weak scaling of fork-join model. It keeps optimal scale up to 10 cores, and only has very slightly increase in elapsed time as the number of cores increases. Every time we double the number of cores, it only increase the running time by about 4%.
Figure 3: strong scaling of parallel FMM
Figure 4: weak scaling of parallel FMM
References


