Parallel Computing: Homework 2

An Ju, Huazhe Xu

March 2, 2018

1 Introduction

Particle simulator is an important problem. We are trying to build our O(n) complexity linear code and MPI parallel computing. Our Makefile is slightly changed to output different executable files.

We developed a serial version that is better than the baseline provided by the source code.And uppon that, we further improve the optimization by using MPI.

2 Serial Implementation

In the serial version, we optimized the original simulation algorithm from $O(n^2)$ to $O(n)$. As a brief description of the algorithm, we divided the area into equal-size square bins (implemented as an array of `std::vector<particle_t*>`) with `cutoff` as side length. Then for each update, we

1. clear all the bins;
2. dispatch particles into bins;
3. for each particle p in bin bin_i, consider only particles in neighboring bins when calculating forces.

The time complexity is reduced from $O(n^2)$ to $O(9 * width * n)$, since there are 9 neighboring bins for each particle and each bin has on average a constant number of particles.

3 MPI Implementation

In the MPI implementation, it is developed based on message passing. To initialize them, equal rows of bins are assigned to processors by the master thread. Similar to the serial O(n) code, forces will be calculated using the neighboring bins at each timestep. Then all the particles will be moved with the rules: All the local particles will be checked with the location and there will be three possible conditions: up or down to the current row, or in the current row. If it is the first two conditions, the particles will be moved to the up or down row for the later communications. If it is on the same bin, we re-binning all the particles. Our implementation need the information of the neighboring 2 rows of the corresponding processors, which are the upside row and downside row.

We put our core code here:

```
MPI_Request req_1, req_2;
        std::vector<particle_t> cmessage_1, cmessage_2;
        // Send message upward
        if (rank != 0) {
            cmessage_1.reserve(message_1.size());
            cmessage_1.insert(cmessage_1.end(), message_1.begin(), message_1.end());
            message_1.clear();
            MPI_Isend(cmessage_1.data(), int(cmessage_1.size()), PARTICLE, rank-1, 0, MPI_COMM_WOR
        }
        if (rank != n_proc - 1) {
            cmessage_2.reserve(message_2.size());
```

```
    cmessage_2.insert(cmessage_2.end(), message_2.begin(), message_2.end());
    message_2.clear();
    MPI_Isend(cmessage_2.data(), int(cmessage_2.size()), PARTICLE, rank+1, 0, MPI_COMM_WOR
}

MPI_Status status_1, status_2;
int size_1, size_2;
std::vector<particle_t> msg_1, msg_2;
if (rank != 0) {
    MPI_Probe(rank-1, 0, MPI_COMM_WORLD, &status_1);
    MPI_Get_count(&status_1, PARTICLE, &size_1);
    msg_1.resize(size_1);
    MPI_Recv(msg_1.data(), size_1, PARTICLE, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
    for (int p = 0; p < msg_1.size(); p++) {
        bins[get_ind(msg_1[p], width)].push_back(msg_1[p]);
    }
}
if(rank != n_proc-1) {
    MPI_Probe(rank+1, 0, MPI_COMM_WORLD, &status_2);
    MPI_Get_count(&status_2, PARTICLE, &size_2);
    msg_2.resize(size_2);
    MPI_Recv(msg_2.data(), size_2, PARTICLE, rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
    for (int p = 0; p < msg_2.size(); p++) {
        bins[get_ind(msg_2[p], width)].push_back(msg_2[p]);
    }
}

MPI_Barrier( MPI_COMM_WORLD );
```

We compare the results of our MPI with the vanilla algorithm in the original code. We found that our method is significantly better than the original code.

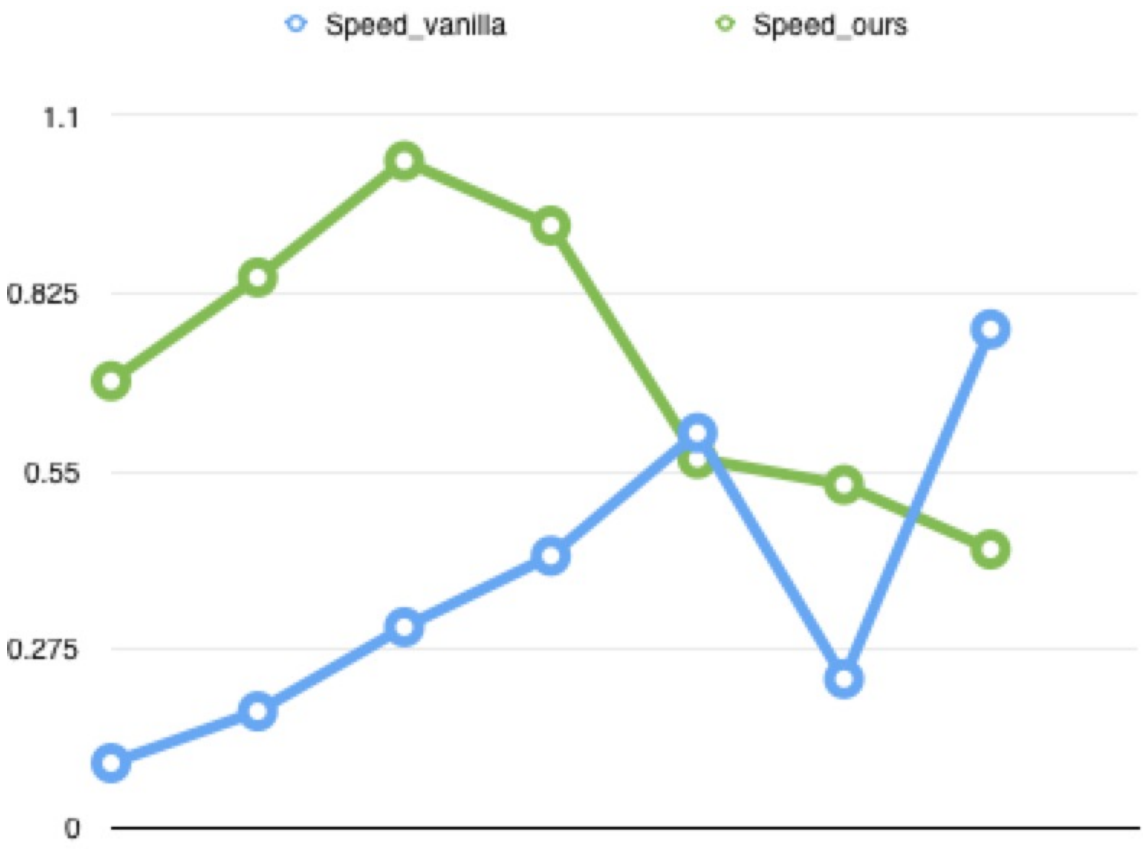

Figure 1: Speed performance. X axis is number of threads.

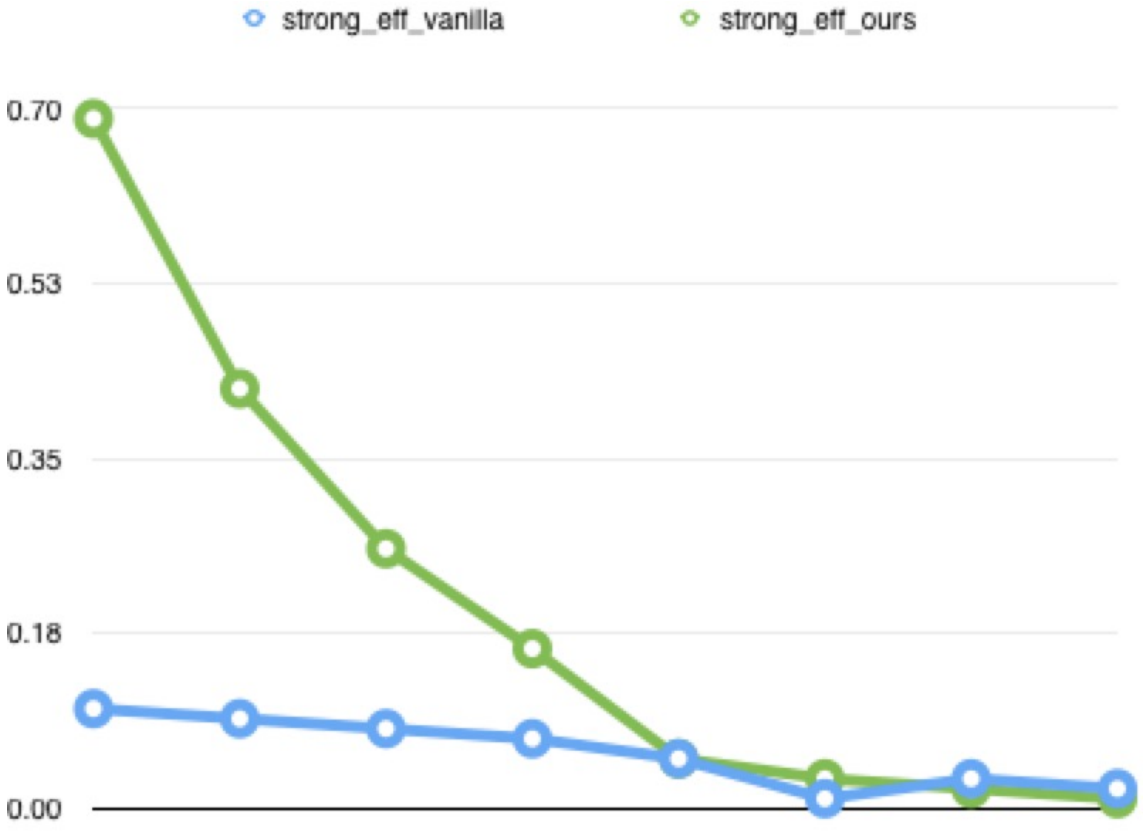

Figure 2: Efficiency with strong scaling. X axis is number of threads.

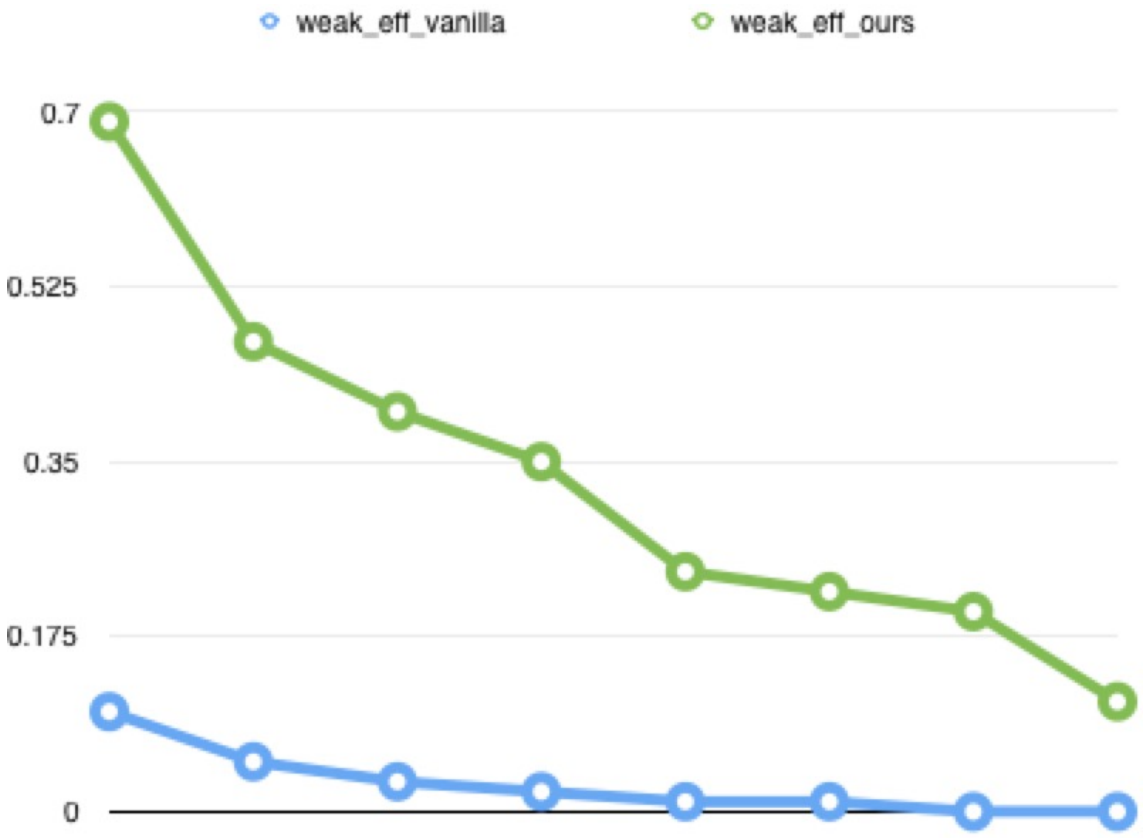

Figure 3: Efficiency with weak scaling. X axis is number of threads.