# Parallel Particles

Patrick Williams and Gabriel Marcus

November 2019

## 1 Introduction

Standard molecular dynamics algorithms rely on calculations of interactions between particles or objects mediated by a force which simulates actual physical laws. At each time step, positions, velocities and accelerations are determined for each particle. These values are continuously updated so that particles can experience distance dependent forces and move accordingly. In a naive algorithm, forces are computed by the interactions of every particle with every other particle. Ultimately, with $n$ particles, this yields a time complexity of $O(n^2)$. The purpose of the following assignment was to redesign the MD algorithm in order to achieve a linear complexity of $O(n)$.

## 2 Revised Serial Algorithm

In order to reduce the required simulation time, each particle was allowed to interact with only a small subset of particles rather than every other particle. For this assignment, the particle box was subdivided into 2-dimensional cells with equal width and height. The length of a square cell for a given cutoff distance $c$ and total space dimensions $s$ by $s$ is $s/(\lfloor s/c \rfloor)$. At each timestep, a particle will be located in one cell and interact with only those particles in its own cell or in the immediate neighbor cells sharing a side or corner. The algorithm used to create the vectors for each cell is shown in Algorithm 1. Pseudocode for the algorithm used for computation is provided in Algorithm 2.

In the algorithm, particles are sorted according to the particular cell that they occupy during a given timestep. Vectors are updated each time step so that before any computations, each vector only contains indices of particles in their corresponding cell. Forces are computed between particles within a given vector as well as with the particles in the vectors representing neighboring cells. Before the forces are calculated, accelerations for all particles in a given vector are set to zero. After this is done, the neighboring particles are determined in a series of for loops. The set of loops runs over all neighbor cells and determines the forces between the particles in the vector and any nearby particles. Finally, velocities and positions are updated for the next timestep based on the interactions between particles in the present timestep.

# 3 Serial Results

Our optimized code successfully changed the complexity of the algorithm from $O(n^2)$ to $O(n)$. The results of our scaling experiments are shown in Figure 1 and Table 1. We confirmed that the optimized code worked correctly by visualizing the output. We were satisfied with the results of our optimization as the bash script confirmed that the estimated slope of the resulting data was 1.

**Algorithm 1** Prepare cell vectors for given timestep

```
for each cell in cell_vectors do
    clear cell
end for
for each particle p do
    cell_x = ⌊p.x/cell_edge_length⌋
    cell_y = ⌊p.y/cell_edge_length⌋
    cell_index = cell_x * num_cells_in_row + cell_y
    cell_vectors[cell_index].push(i)
end for
```

**Algorithm 2** Calculate forces of particles on particles in neighboring cells

```
for each cell do
    Set acceleration of all particles in cell to 0
    for each neighboring cell do
        for each particle i in cell do
            for each particle j in neighbor do
                apply_force(particles[i], particles[j])
            end for
        end for
    end for
end for
```

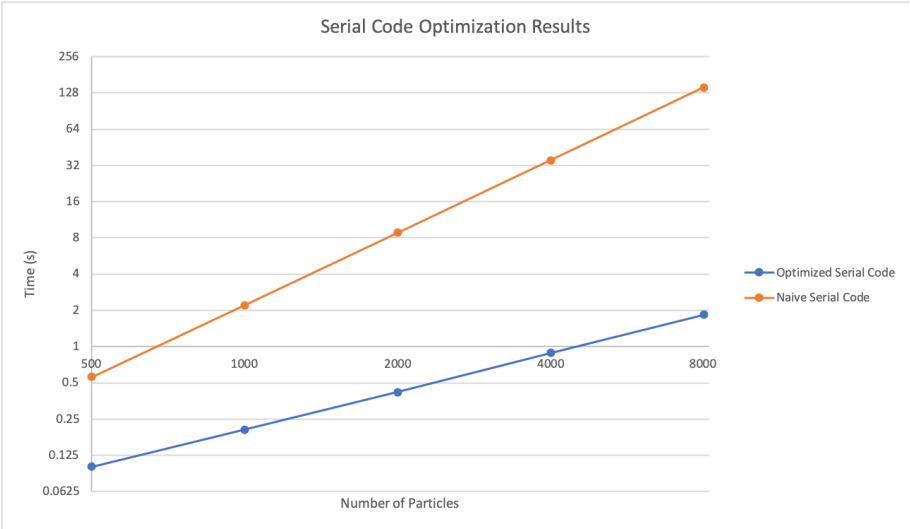


Figure 1: Results of linear time optimization of serial code

| Number of Particles | Optimized Serial Code (s) | Naïve Serial Code (s) |
|---|---|---|
| 500 | 0.100327 | 0.561042 |
| 1000 | 0.20444 | 2.21187 |
| 2000 | 0.417481 | 8.8017 |
| 4000 | 0.883508 | 35.277 |
| 8000 | 1.84302 | 141.392 |

Table 1: Results of linear time optimization of serial code

# 4 OpenMP Optimization

In order to optimize the algorithm further, we altered an existing OpenMP implementation of the original serial code to work with our optimized serial code. One of the major changes we made to allow the code to run in parallel was to lock the *cell_vector* variable when building it at the beginning of each timestep. We also made the outer two for loops that iterated through the x and y coordinates of the cells run in parallel with $pragma\ omp\ for\ collapse(2)$, making both loops run in parallel as if they were one loop. Because we parallelized the outermost loop, we could not make any of the inner portions of the loop parallel. Because of this, we were otherwise limited in the changes we could make.

# 5 OpenMP Results

After parallelizing the OpenMP code, we ran strong and weak scaling tests. The results of these tests are shown in Figures 2 and 3, and Tables 2 and 3. For strong scaling, we saw that for smaller numbers of threads, our optimized OpenMP code ran faster than the naive code, but as the number of processors increased, the speedup stalled, and the code ran slower than the naive code. This could be because of some overhead caused by generating the threads, or a performance bug in our code. However, for weak scaling we observed that the naive code ran much slower than the optimized code when the number of particles was doubled every time the number of processors was doubled.

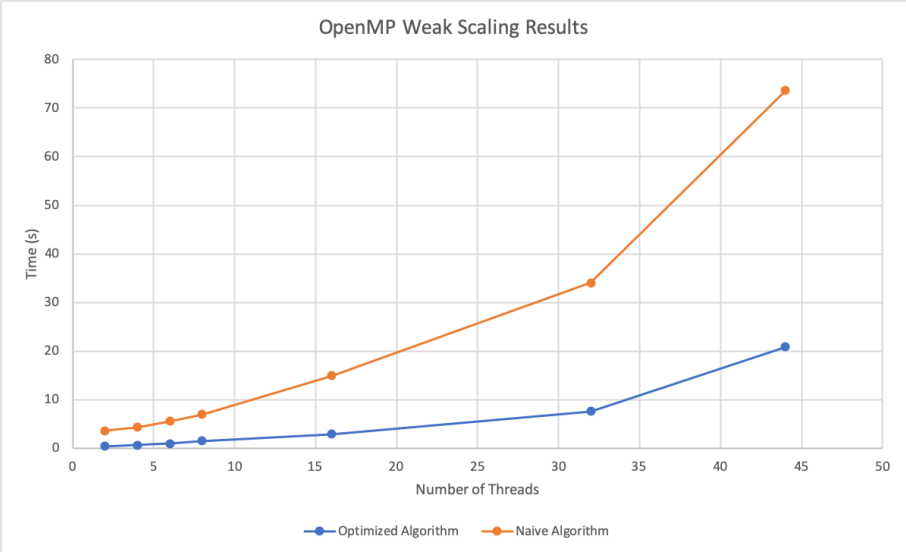


Figure 2: Weak scaling results of OpenMP optimization

| OpenMP Weak | | | |
|---|---|---|---|
| Number of Particles | Number of Threads | Optimized Algorithm (s) | Naïve Algorithm (s) |
| 1000 | 2 | 0.414039 | 3.6345 |
| 2000 | 4 | 0.627273 | 4.34984 |
| 3000 | 6 | 0.975642 | 5.54184 |
| 4000 | 8 | 1.49175 | 7.00919 |
| 8000 | 16 | 2.89867 | 14.94 |
| 16000 | 32 | 7.55588 | 34.0503 |
| 22000 | 44 | 20.8615 | 73.5848 |

Table 2: Weak scaling results of OpenMP optimization

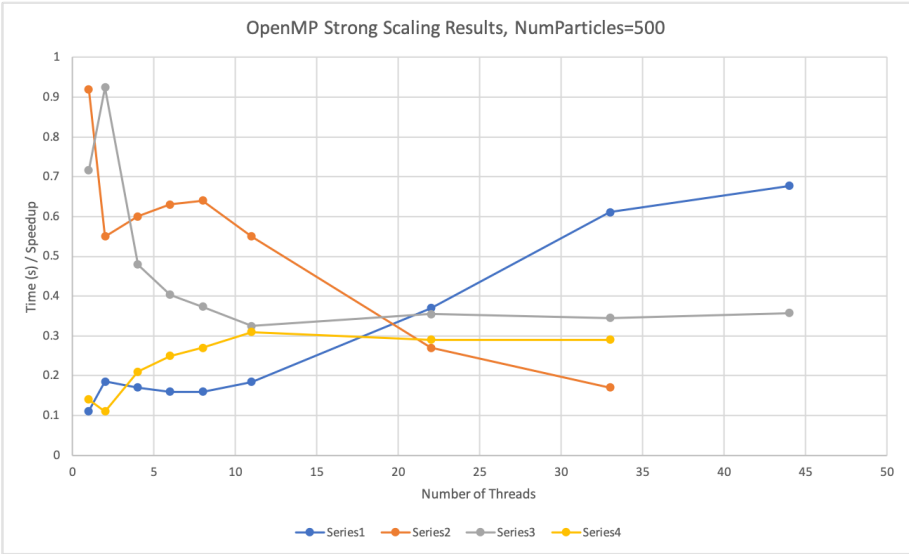


Figure 3: Strong scaling results of OpenMP optimization

| OpenMP Strong | | | | | |
|---|---|---|---|---|---|
| Number of Particles | Number of Threads | Optimized Algorithm (s) | Optimized Speedup | Naïve Algorithm (s) | Naïve Speedup |
| 500 | 1 | 0.110783 | 0.92 | 0.716103 | 0.14 |
| 500 | 2 | 0.185192 | 0.55 | 0.924768 | 0.11 |
| 500 | 4 | 0.170189 | 0.6 | 0.479296 | 0.21 |
| 500 | 6 | 0.160144 | 0.63 | 0.403416 | 0.25 |
| 500 | 8 | 0.159627 | 0.64 | 0.373625 | 0.27 |
| 500 | 11 | 0.184429 | 0.55 | 0.325238 | 0.31 |
| 500 | 22 | 0.369796 | 0.27 | 0.354469 | 0.29 |
| 500 | 33 | 0.611344 | 0.17 | 0.345148 | 0.29 |
| 500 | 44 | 0.676739 | | 0.357554 | |

Table 3: Strong scaling results of OpenMP optimization

# 6 MPI Parallelization

For the second attempt at parallelization of the optimized serial code, MPI was used. MPI methods present a greater challenge in parallelization because there is no global memory which is available for reading and writing by all threads. Instead, each thread has its own local memory and any data sharing must be accomplished through various types of message passing. This may include one-to-one, one-to-all, all-to-one, or all-to-all communication.

In order to parallelize the serial code using MPI, a simple approach using only two threads was attempted first as outlined in Algorithm 3. Initially, particles were scattered to each thread using the Scatterv command. Those particles scattered to thread 0 but located in the right side region of the simulation box were collected in a local vector of thread 0 while particles in the left side region but scattered to thread 1 were collected in a local vector of thread 1. Also,

particles located within a certain cutoff distance from the center of the simulation box were entered into a local boundary vector in each thread. After this was done, particles were exchanged using MPI's Send and Receive commands so that thread 0 contained all particles on the left side of the simulation box and thread 1 contained all particles on the right side of the simulation box. Boundary vectors were also exchanged between threads so that particles could properly communicate if they were close enough to each other but owned by different threads. Finally, forces were computed within each thread for interactions between all particles local to that thread. This procedure was repeated for each timestep as particle locations were updated.

**Algorithm 3** MPI Algorithm for Two Threads

```
Scatter particles evenly among threads
for each timestep do
  for each local particle do
    if particle is located inside range of other thread then
      push particle to send_vector
    else
      if particle is within cutoff distance of other thread's boundary then
        push particle to boundary_vector
      end if
      push particle to local_vector
    end if
  end for
  send and receive send_vector and boundary_vector
  append received send_vector to local_vector
  for each particle i in local_vector do
    for each particle j in local_vector do
      apply_force(i, j)
    end for
    for each particle j in boundary_vector do
      apply_force(i, j)
    end for
  end for
end for
```

# 7 MPI Results

Using MPI with two threads and a range of particle sizes up to 300 yielded a very small speedup as compared to the serial code. As seen in Figure 4, the deviation increased as particle size grew. Unfortunately, the simulation failed to run with particle sizes exceeding 300. Also, the code was not optimized to run with more than two threads. Scaling the parallelization to utilize more threads could potentially be achieved by dividing the simulation box further

and applying a similar method as described here. In this case, horizontal and vertical boundaries would be required and the particle exchange would need to accommodate the larger number of threads. For example, four threads would divide the simulation box into quadrants. Particles would need to be sorted and sent to the proper thread based on their location as in the algorithm described above. Boundary vectors would need to be created for every border between neighboring cells that each correspond to a given thread. For future work, we would fix the bugs preventing our MPI code from running with larger numbers of particles and would begin working to allow the code to run with more than two threads.

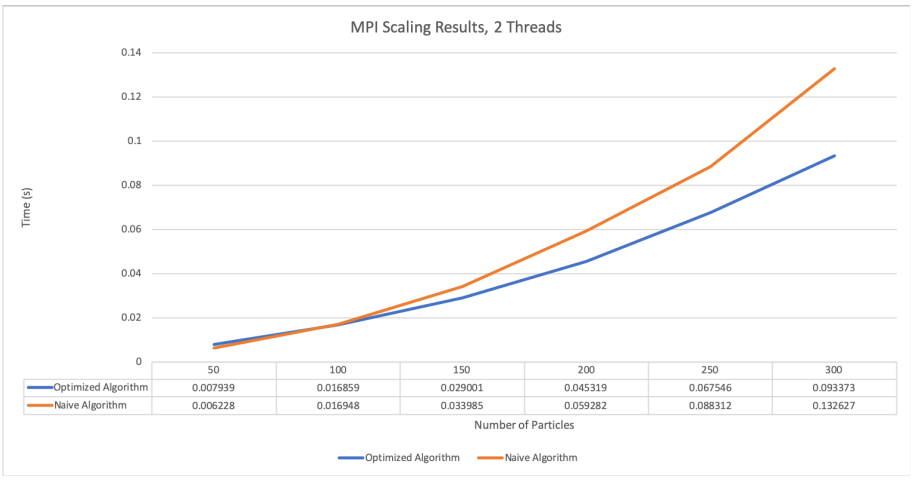


Figure 4: Results of using MPI parallelization with two threads for a range of particle sizes.