# Homework 2.1

CS 267

2/22/2024

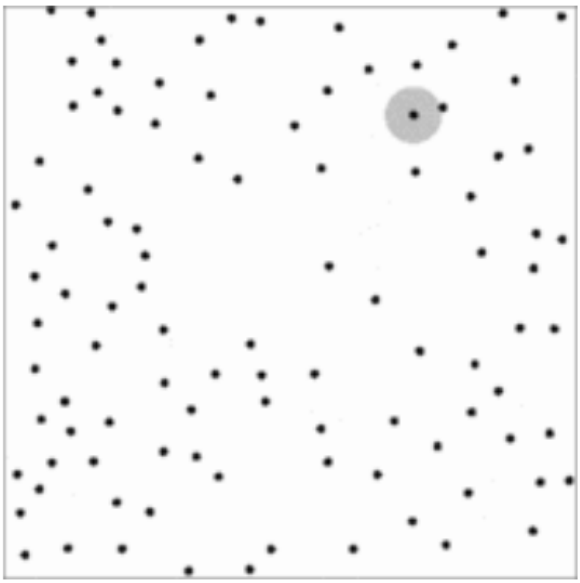
Giovanni Battista Alteri, Jeffy Jeffy, Elizabeth Gilson

Group 36

**Introduction**

The goal of this assignment is to parallelize a particle simulation, starting from a naive implementation algorithm. In our simulation, we have a closed system where the particles interact by repelling one another, but only when they are closer than a cutoff distance. The number of particles is constant, and so is the volume of the system (defined by the variable "size", given by the square root of the total number of particles). We have chosen a density sufficiently low so that we will have $O(n)$ interactions with $n$ particles. This means that we could achieve the same level of accuracy of the naive code even if we consider only the interactions among particles that are sufficiently close to each other.

This is a snapshot of the simulated system:



The starting naive implementation computes the forces on all the particles, by iterating through every pair, with a final asymptotic computational complexity of $O(n^2)$. This algorithm calls a nested for-loop that computes unnecessary interactions between pairs too far to interact, making this approach inefficient and slow.

Here you can see the naive code implementation:

```
void simulate_one_step(particle_t* parts, int num_parts, double size) {
    // Compute Forces
    for (int i = 0; i < num_parts; ++i) {
        parts[i].ax = parts[i].ay = 0;
        for (int j = 0; j < num_parts; ++j) {
            apply_force(parts[i], parts[j]);
        }
    }

    // Move Particles
    for (int i = 0; i < num_parts; ++i) {
        move(parts[i], size);
    }
}
```

The first phase of our work was the implementation of a serial code that could achieve a complexity of *O(n)*, by dividing the system into “bins”, squares that include a certain amount of particles each. By computing only the forces of the pairs of particles that are in near bins, we can reduce the computational complexity of the naive code.

**Serial Optimization**

For our first attempt of the serial implementation we used a vector of vectors of integers to store the bins and used indices to place the particles in bins. This increased performance a bit, down from 1.47 (from the naive implementation) to 0.92 for 1000 particles, however there was overhead from the dynamic memory allocation/deallocation and the vector operations.

To address these issues we used a linked list as our data structure to store the bins. This is because throughout the simulation particles move around and change bins frequently. Having a linked list allows for dynamic particle insertion and deletion, so particles can switch bins without the need to resize of shift elements. The dynamic nature of linked lists allows us to avoid unnecessary array/vector operations that are very costly.

```
constexpr int MAX_PARTICLES = 1000000;

// Define a struct for linked list node
struct ListNode {
    int index;
    ListNode* next;
    ListNode(int idx) : index(idx), next(nullptr) {}
};

// Global declaration of bins
std::array<std::array<ListNode*, MAX_PARTICLES>, MAX_PARTICLES> bins; // Use fixed-size array instead of vector
int binCountX, binCountY; // Number of bins in each dimension
int binSize;

// Apply the force from neighbor to particle
void apply_force(particle_t& particle, particle_t& neighbor) {
    // Calculate Distance
    double dx = neighbor.x - particle.x;
    double dy = neighbor.y - particle.y;
    double r2 = dx * dx + dy * dy;

    // Check if the two particles should interact
    if (r2 > cutoff * cutoff)
        return;

    r2 = fmax(r2, min_r * min_r);
    double r = sqrt(r2);

    // Very simple short-range repulsive force
    double coef = (1 - cutoff / r) / r2 / mass;
    particle.ax += coef * dx;
    particle.ay += coef * dy;
}
```

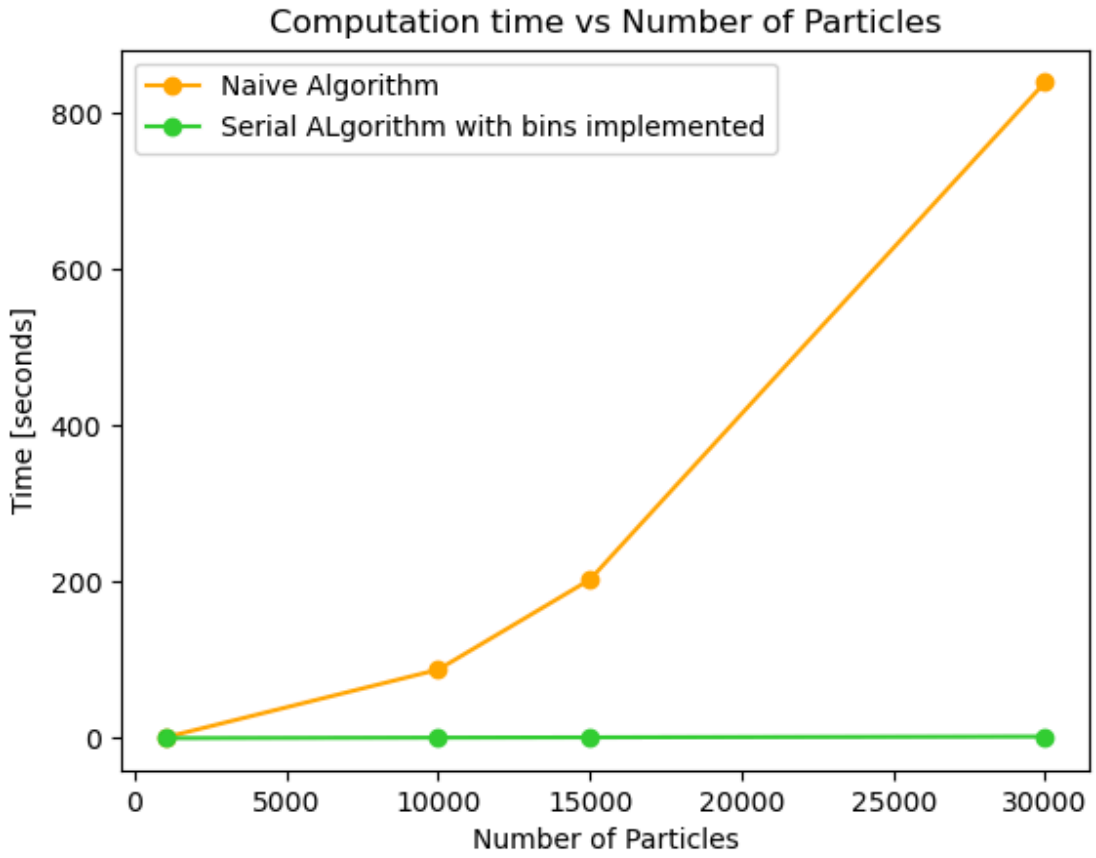
We use a fixed-size array instead of a dynamic vector for storing the bins. To do this we need to include a max number of particles. We have set this value to one million, however it can easily be changed by editing the MAX_PARTICLES value. This increases performance as it reduces memory overhead and improves cache locality compared to dynamic memory allocation.

```
void init_simulation(particle_t* parts, int num_parts, double size) {
    binSize = std::max(static_cast<int>(std::ceil(cutoff * 0.5)), 1);
    binCountX = std::ceil(size / binSize);
    binCountY = std::ceil(size / binSize);
}

// Assign particles to bins
void assign_particles_to_bins(particle_t* parts, int num_parts, double size) {
    // Clear bins
    for (int i = 0; i < binCountX; ++i) {
        for (int j = 0; j < binCountY; ++j) {
            bins[i][j] = nullptr; // Initialize all bins to nullptr
        }
    }

    // Assign particles to bins
    for (int i = 0; i < num_parts; ++i) {
        int binX = parts[i].x / binSize;
        int binY = parts[i].y / binSize;
        bins[binX][binY] = new ListNode(i); // Assign a new node to the bin
    }
}
```

Additionally the **assign_particles_to_bins** function initializes all bins to nullptr before assigning particles to them. This preallocation/initialization avoids the overhead of dynamic memory allocation during simulation steps.

```
void simulate_one_step(particle_t* parts, int num_parts, double size) {
    // Re-assign particles to bins to account for movement
    assign_particles_to_bins(parts, num_parts, size);

    // Compute forces with binning
    for (int i = 0; i < num_parts; ++i) {
        parts[i].ax = parts[i].ay = 0;
        int binX = parts[i].x / binSize;
        int binY = parts[i].y / binSize;

        // Iterate through neighboring bins
        for (int dx = -1; dx <= 1; ++dx) {
            for (int dy = -1; dy <= 1; ++dy) {
                int newX = binX + dx, newY = binY + dy;
                if (newX >= 0 && newX < binCountX && newY >= 0 && newY < binCountY) {
                    ListNode* node = bins[newX][newY]; // Get the head of the linked list
                    while (node != nullptr) {
                        apply_force(parts[i], parts[node->index]);
                        node = node->next;
                    }
                }
            }
        }
    }

    // Move particles
    for (int i = 0; i < num_parts; ++i) {
        move(parts[i], size);
    }
     // Clean up bins after each simulation step --> nvm makes it slower
    //cleanup_bins();
}
```

Here we can see the performances of the two algorithms, expressed as functions of the time vs the number of particles present in each simulation:



**OpenMP Optimization**

In order to parallelize the code, we limited all openMP functionality to the **simulate_one_step** function. Here, OpenMP parallelism is introduced to compute forces and move particles simultaneously. We use the **#pragma omp for** directive to enable parallel processing of particles, speeding the simulation by distributing the work across multiple threads.

```
void simulate_one_step(particle_t* parts, int num_parts, double size) {
    // Re-assign particles to bins to account for movement
    // assign_particles_to_bins(parts, num_parts, size);

    // Compute forces with binning
    #pragma omp for
    for (int i = 0; i < num_parts; ++i) {
        parts[i].ax = parts[i].ay = 0;
        int binX = parts[i].x / binSize;
        int binY = parts[i].y / binSize;

        // Iterate through neighboring bins
        for (int dx = -1; dx <= 1; ++dx) {
            for (int dy = -1; dy <= 1; ++dy) {
                int newX = binX + dx, newY = binY + dy;
                if (newX >= 0 && newX < binCountX && newY >= 0 && newY < binCountY) {
                    ListNode* node = bins[newX][newY]; // Get the head of the linked list
                    while (node != nullptr) {
                        apply_force(parts[i], parts[node->index]);
                        node = node->next;
                    }
                }
            }
        }
    }
```

Next, we used the **#pragma omp for** directive to enable the parallel processing of the loop which moves each particle based on its current velocity and the given size.

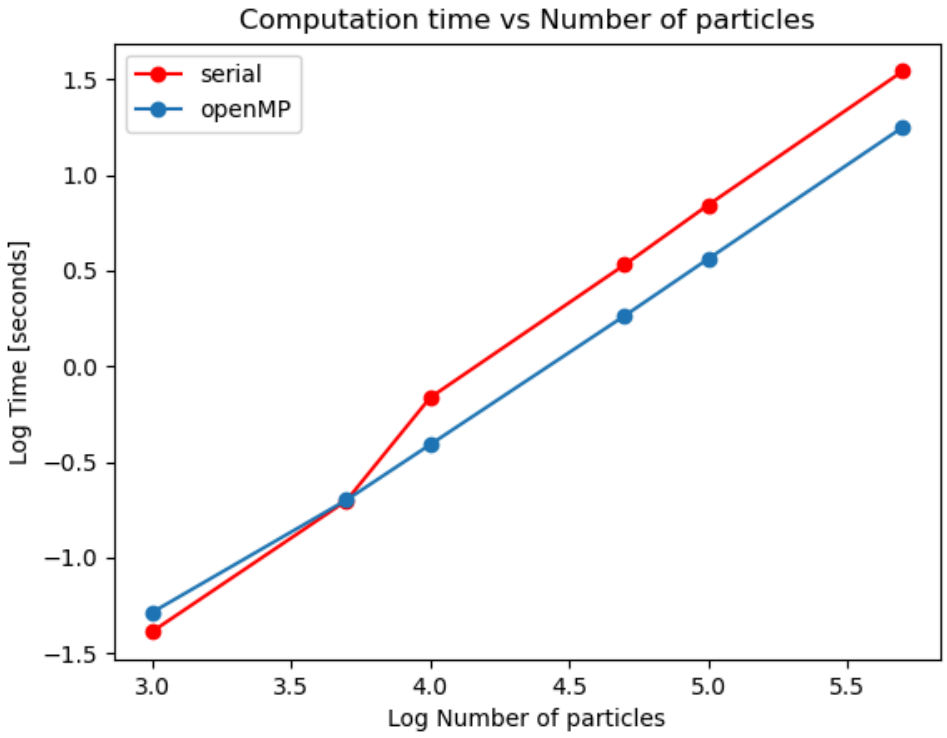
```
    // Move particles
    #pragma omp for
    for (int i = 0; i < num_parts; ++i) {
        move(parts[i], size);
    }
```

Lastly, we utilized **#pragma omp master** to recalculate the bins. Here, the bin calculation is only done by the master thread, preventing any redundant calculations. In sync with the **#pragma omp master** directive, we used the **#pragma omp barrier** directive to ensure that all threads wait for the master thread to finish bin calculations before proceeding to the next step.

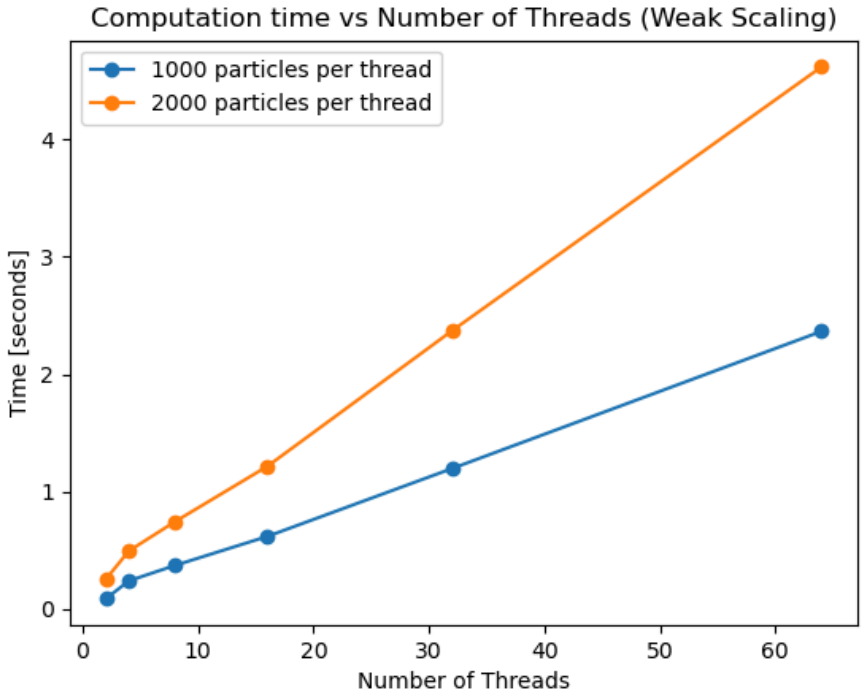
```
    // Recalculate bins for particles that moved in the previous time step
    #pragma omp master
    {
        assign_particles_to_bins(parts, num_parts, size);
    }
    // Synchronize threads

    #pragma omp barrier
    // Clean up bins after each simulation step --> nvm makes it slower
    //cleanup_bins();
```

**Results**

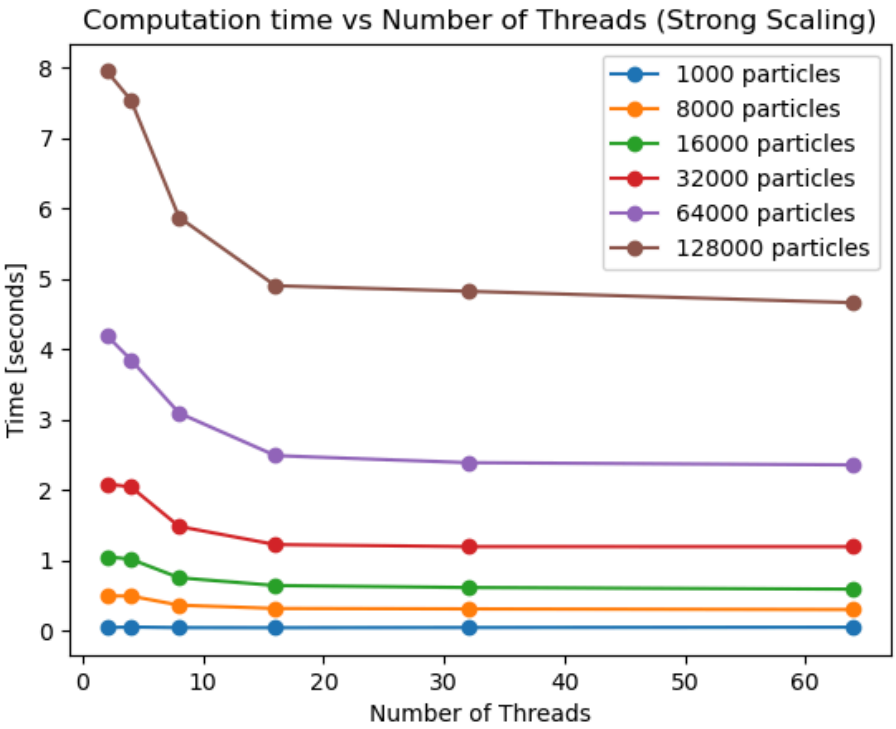
We compared our two implementations for various numbers of particles and plotted the results on a log time vs log number of particle scale. This analysis revealed that for very small numbers of particles the serial code outperformed the openMP code. But, as the number of particles increased the openMP implementation became faster. We believe this is due to the fact that the parallelization has some overhead, and for small particle counts the parallelization is less useful so therefore the serial implementation is faster.




Then to analyze how our openMP implementation performs with weak scaling we ran the code with 1000 and 2000 particles per thread for varying numbers of threads. Since we are doubling the number of particles, if our implementation scales well then we would assume double the processors should yield similar results. From our graph, we found that this is generally true, so therefore our model does scale well in terms of weak scaling.




For our test of strong scaling, we measured the computational time for varying numbers of particles and varying numbers of threads. The results are shown below.



From the graph, it is evident that there is a bottleneck in the initial openMP code prohibiting the code from strong scaling. We suspected that this might be due to the single thread control that we implemented before the **assign_particle_in_bins** function. We believe that utilizing locks within this function might allow us to clear this bottleneck, as it would allow multiple threads to run the code concurrently without the fear of threads overwriting each other.

**OpenMP Code Optimization:**

To test our theory regarding the bottleneck in utilizing a single thread to update the bin assignment, we generated an **omp_lock_t** pointer array that would store the locks for all the bins. We initialized this variable globally, and then used the **init_simulation** function to dynamically allocate space for the array. This time, we removed the barrier and the master thread restriction from the **assign_bins** function and parallelized the for loop that assigned the locks and the particles into bins.

The code did not show a significant improvement from the previous parallelization that we had already performed. The running times were the same as what has already been described.

Next, we tried to initialize a **max_bin** global variable to prevent the array from dynamically allocating memory, instead allocating a specific space and clearing this space in each simulation step, this slowed down the code even more.

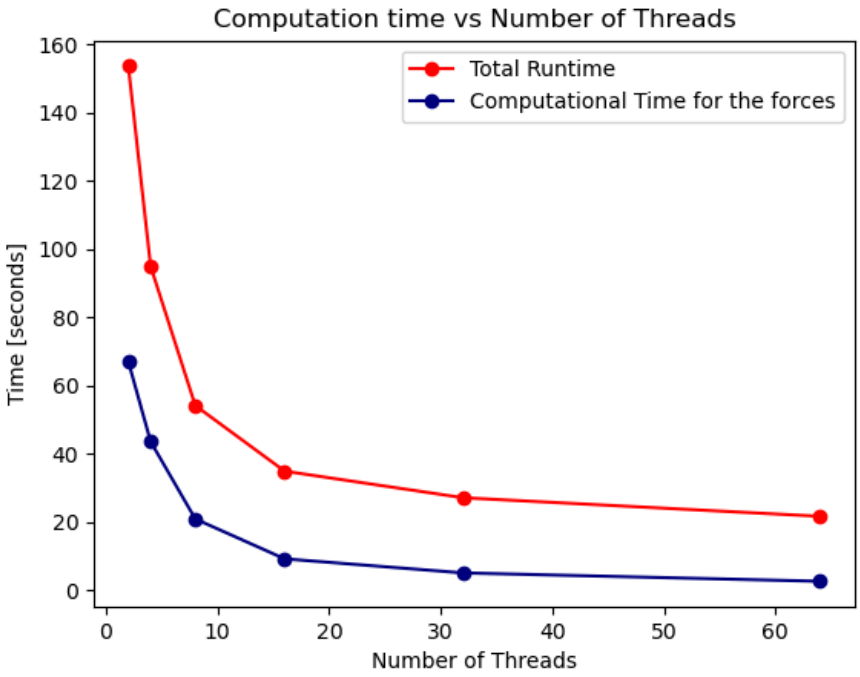
We tried some additional ways of assigning locks, tried critical and atomics pragma derivatives, and these either resulted in very slow running time, or segmentation. We believe that because our bins are stored in a linked list but our locks are in an array data structure, we might be running into deadlock. Additionally, it is also possible that we are running into data races while allocating memory, causing segmentation fault.

**Runtime analysis**

We studied the total runtime of our simulations introducing three new variables (*start*, *end* and *diff*, all doubles), and we calculated the time at each moment of interest using the built-in function of OpenMP **omp_get_wtime**, launched before and after each iteration of the function **apply_force** inside the while loop; the total time was stored in the variable *diff*, and returned by the function **simulate_one_step** itself.

```
double simulate_one_step(particle_t* parts, int num_parts, double size) {
    // Re-assign particles to bins to account for movement
    // assign_particles_to_bins(parts, num_parts, size);
    double start;
    double end;
    double diff;

    // Compute forces with binning
    #pragma omp for
    for (int i = 0; i < num_parts; ++i)
    {
        parts[i].ax = parts[i].ay = 0;
        int binX = parts[i].x / binSize;
        int binY = parts[i].y / binSize;

        // Iterate through neighboring bins
        for (int dx = -1; dx <= 1; ++dx) {
            for (int dy = -1; dy <= 1; ++dy) {
                int newX = binX + dx, newY = binY + dy;
                if (newX >= 0 && newX < binCountX && newY >= 0 && newY < binCountY) {
                    ListNode* node = bins[newX][newY]; // Get the head of the linked list
                    while (node != nullptr) {
                        start = omp_get_wtime();
                        apply_force(parts[i], parts[node->index]);
                        end = omp_get_wtime();
                        double tmp;
                        tmp = end - start;
                        diff += tmp;
                        node = node->next;
                    }
                }
            }
        }
    }

    // Move particles
    #pragma omp for
    for (int i = 0; i < num_parts; ++i) {
        move(parts[i], size);
    }

    // Recalculate bins for particles that moved in the previous time step
    #pragma omp master
    {
        assign_particles_to_bins(parts, num_parts, size);
    }

    // Synchronize threads
    #pragma omp barrier

    return diff;
}
```

Confronting the total computational time already calculated by the program, we could get an esteem of the computational time spend by our software to calculate the forces; the difference between this two values gives us the summation of all the times for the rest of the computation (the time spend to assign particles to bins, the time spent to synchronization and communication). Here you can see how this quantities scales with the number of threads:




The plot shows the total runtime, and the total computational time necessary to evaluate the forces at each step, as a function of the numbers of threads (2, 4, 8, 16, 32 and 64) used in simulations with 500000 particles each. Our results show that the parallelization of the evaluation of the forces in each step of the simulation is, roughly speaking, ~40% of the total runtime of the process with a low number of threads (10 <), and this percentage is reduced to ~10% of the total time with the maximum number of threads (64). The shape of the function indicates that, as we should expect, a higher number of threads is associated with a lower total runtime and a lower computational time for the forces; in both cases we reach the maximum parallelization (i.e. we observe that both functions reach a plateau in their values) with the maximum number of threads.

In a perfect scenario the speedup (the ideal p-time speedup) should be, for instance, four times using four threads (25% of the same process execution with only one thread). Our code shows a speedup that is not ideal, but still substantial anyway. We have an execution time of ~153 seconds with 2 threads and of ~20 seconds with the maximum amount of threads. So we managed to reach a final decrease of a factor ⅛ , or 12.5%.

**Can this performance be better?**

It is certainly possible to improve our performance in this code. We believe if we could have successfully implemented the locks, the code could have been better and would have strongly scaled. Although, it is entirely possible to have a situation where having too many locks could

drag down the code, and the speed. Additionally, we believe the time it takes to wait for master thread to finish assigning bins, and synchronizing the barrier as discussed in the previous section also adds to the performance of the code. Finding better ways to assign bins, beyond waiting for locks or master thread could be helpful in improving the code.

**Contributions**

**Giovanni Battista** - Created the graphs, added to the final report, helped review the code
**Jeffy** - Created first implementation of serial code, created both implementations openMP code, added to final report
**Elizabeth** - Created second implementation of serial code, added to the final report, helped review the code