1 Serial Implementation

For \( n \) particles, the original code requires \( O(n^2) \) time because at each time step, the `apply_force` function is called for each particle (to be updated) with each other particle. Since interactions are local, one particle is influenced by a few nearby particles. Thus, most of the time, the `apply_force` function is computing a distance and returning after finding this distance to be greater than the local force cutoff. Since the density is constant (the size of the domain grows with \( n \)), the total number of local interactions is \( O(n) \). To run the simulation in \( O(n) \) time, the `apply_force` function should be called only \( Cn \) times, for some constant \( C \).

We created a data structure to index particles by their location within the domain. In this way, given a particle at location \((x,y)\), we can use the index to find which other particles are close to \((x,y)\) and only try to apply forces from the nearby particles (rather than all particles). We partitioned the domain into a 2-D array of square “bins” with size equal to the local interaction cutoff, so that a particle can only possibly interact with particles in one of the 8 neighboring bins (or in its own bin). We used an array of the STL `vector` data structures to store pointers to particles in each bin. Since the size \( s \) (one side) of the domain is proportional to \( \sqrt{n} \) and the number of bins \( B \) is proportional to \( s^2 \), \( B \) is proportional to \( n \) (the constant turns out to be about 5, depending on the fringe). Only pointers to particles need be stored (rather than copies) because the `apply_force` function reads the position of particles and writes the acceleration components. Thus, as long as no particles are moved before all particles’ accelerations are computed, no data copying is necessary.

Thus, for each time step, our serial algorithm does four things: clear the bins, assign particles (i.e. pointers to particles) to bins, compute forces (only those in neighboring bins), and move the particles. Clearing the bins requires \( B \) calls to the constant-time vector function `empty()`, assigning the particles to bins requires \( n \) calls to the constant-time vector function `push_back()`, computing forces requires \( \alpha n \) calls to `apply_force` (where \( \alpha \) is the average number of particles in neighboring bins), and moving the particles requires \( n \) calls to the `move` function. Thus, as long as \( \alpha \) is a constant (about 2.2 with the given density and cutoff constants), the algorithm runs in \( O(n) \) time. Figure 1 shows the original code runs in \( O(n^2) \) time and our serial (and parallel) code runs in \( O(n) \) time.

2 Shared Memory Implementation

In our shared memory implementation, we divide the work on the particle array among the threads (as in the un-optimized codes). Since we add the work of emptying the bins and assigning particles to bins at the beginning of every time step, we also divide the work on the bin array among the threads. In the pthread implementation, we computed a “first bin” and “last bin” value for each thread, but in the OpenMP implementation we were able to hide those calculations in a `#pragma omp for` statement. Again, the four steps taken at each time
Figure 1: Time in seconds required to simulate $n$ particles

step are: emptying bins, binning particles, applying forces, and moving particles. Emptying bins is embarrassingly parallel, and since applying forces does not change the positions of particles, it is also embarrassingly parallel. As long as no particles are moved before all accelerations are computed, moving particles is embarrassingly parallel. The only shared memory pitfall occurs during the particle binning step. In order to prevent two threads from adding particles to the same bin vector at the same time, we implemented a mutex/lock for each bin. Here is the section of code in our OpenMP implementation that handles particle binning:

```c
// place particles in bins
#pragma omp for
for (int i = 0; i < n; i++)
{
    int cbin = binNum(particles[i],bpr);
    particle_t* cparticle = particles + i;
    omp_set_lock( &binlock[cbin] );
    bins[cbin].push_back(cparticle);
    omp_unset_lock( &binlock[cbin] );
```
Figure 2 shows the performance of our OpenMP implementation for different numbers of threads.

![OpenMP Performance Graph]

Figure 2: OpenMP performance. Speedup is measured against the implementation run with 1 thread.

We first implemented our shared memory algorithm using pthreads. After a successful pthreads implementation, we wrote an OpenMP version. We found the original pthreads code to be more intuitive and easier to emulate ( pthreads doesn’t hide as much detail), but our final OpenMP code is much cleaner and differs only slightly from our serial version. We saw better performance with OpenMP (see Table 4). We believe this is due to a more efficient overhead management system in OpenMP.

### 3 Distributed Memory Implementation

Our first attempt at a distributed memory implementation was to mimic the shared memory version. After scattering the initial particle array, each processor copies its local particles to a buffer array. For each time step, in order to apply forces to local particles each processor bins
the particles in its buffer array, updates its local particles, passes its buffer to its neighbor (in a ring topology), and repeats until all collisions have been calculated. Once all forces have been applied, each processor moves its local particles. The main problem with this approach is that each processor must bin all particles during every time step, so the computational work remains $O(n)$ rather than $O(n/P)$.

In order to achieve a scalable algorithm we decided to decompose the global domain. That is, given four processors, we assign the particles in the top left corner of the domain to the first processor, the particles in the top right to the second, and so on. In this way, each processor can utilize the serial implementation to update its local particles at each time step. However, particles in neighboring domains can affect local particles, and local particles can move into neighboring domains. Because of the local restrictions on force and movement (by assumption), processors only need to communicate with nearest neighbors. Thus, both computation and communication are scalable.

Although we were not able to implement a complete version of this approach, we successfully decomposed the global domain in one dimension so that processors own vertical slices. With this decomposition, the number of messages sent to nearest neighbors at each time step is constant. Thus, the latency of the algorithm scales well. However, since the size of the boundaries between processor domains grows like $\sqrt{n}$, and the amount of particle information communicated between neighboring processors depends on the size of the boundaries, the size of messages will increase as the number of particles increase. Therefore, the bandwidth of our implementation does not scale well (we haven’t minimized the surface-to-volume ratio).

Here is the pseudocode for our implementation:

1: distribute particles to processors by position
2: for all timesteps do
3:   clear bins
4:   bin local particles
5:   exchange boundary bins with neighbors
6:   compute forces on local particles
7:   move local particles
8:   exchange escaped particles
9: end for

After scattering the particles initially, all of the communication in our implementation uses MPI_Isend and MPI_Irecv commands. During each time step, each processor must send particles in its local boundary bins to its neighbors (left/right in our implementation) and receive particles in the boundary bins of its neighbors before applying forces to local particles. After updating the positions of the particles for the next time step, each processor sends any particles that leave its domain to the appropriate neighbor (here we assume that a particle does not jump over a neighboring domain in a single time step) and receives other particles that enter its domain.

To avoid sending messages of size greater than necessary, we always send an initial integer message to announce the size of the following particle message. We also pre-post the
MPI_Irecv calls as high as possible. At the end of every timestep we use MPI_Barrier to ensure that all processors are synchronized at each timestep. In the case of generating an output file we use MPI_Gatherv to collect particle information onto the master processor.

Figure 3 shows the performance of our MPI implementation for different numbers of threads.

![Image of graph showing MPI performance]

Figure 3: MPI performance. Speedup is measured against the implementation run with 1 processor.

We note that the speedups we found exceed the ideal values (i.e. the 8 processor run required about a tenth of the time as the 1 processor run). We believe this is due to differences in memory access patterns. In the our distributed memory implementation, particles are spatially sorted and packed into smaller arrays, thereby achieving more spatial memory locality. Also, as the number of processors increases, the memory requirement of the local simulation is much smaller than the memory requirement of the serial implementation. Thus, the entire local problem may be able to fit into a faster memory.
4 Discussion

We ran all our codes on Bassi. Figure 4 shows the fastest performance of each of our codes (up to 8 processors/threads), and Table 4 shows all our collected results. We note that all of our MPI results were measured running on one node and varying numbers of tasks. We tested our MPI code on multiple nodes (with one task per node) and found slower performance than running on one node, but because of queue delays on Bassi we weren’t able to gather as much data in this case.

We’ve drawn a few general conclusions. First, it is very important to find an optimal serial algorithm before even considering parallel implementations. Second, we should not forget lessons learned from the first homework assignment. Although memory optimizations were not the goal of this assignment, we believe our lack of optimization in the serial code resulted in over-optimistic speedup results. We also found the shared memory implementation was much more intuitive (at least for this algorithm). Our MPI implementation was quite difficult to debug, an argument for other more intuitive distributed memory interfaces.

Figure 4: Final results (in seconds).
<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
<th>32000</th>
<th>64000</th>
<th>96000</th>
<th>128000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>8.79</td>
<td>34.75</td>
<td>140.40</td>
<td>554.78</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Serial</td>
<td>0.56</td>
<td>1.15</td>
<td>2.39</td>
<td>5.69</td>
<td>12.93</td>
<td>27.05</td>
<td>55.56</td>
<td>85.31</td>
<td>133.72</td>
</tr>
<tr>
<td>Pthread-8t</td>
<td>0.34</td>
<td>0.46</td>
<td>0.99</td>
<td>1.68</td>
<td>3.55</td>
<td>7.28</td>
<td>14.33</td>
<td>19.12</td>
<td>28.28</td>
</tr>
<tr>
<td>Openmp-1t</td>
<td>0.57</td>
<td>1.17</td>
<td>2.42</td>
<td>5.85</td>
<td>13.18</td>
<td>27.84</td>
<td>57.79</td>
<td>88.68</td>
<td>142.82</td>
</tr>
<tr>
<td>Openmp-2t</td>
<td>0.43</td>
<td>0.88</td>
<td>1.76</td>
<td>3.47</td>
<td>8.02</td>
<td>16.95</td>
<td>34.90</td>
<td>53.00</td>
<td>71.69</td>
</tr>
<tr>
<td>Openmp-4t</td>
<td>0.26</td>
<td>0.50</td>
<td>1.01</td>
<td>2.12</td>
<td>4.45</td>
<td>9.52</td>
<td>19.34</td>
<td>29.62</td>
<td>40.56</td>
</tr>
<tr>
<td>Openmp-6t</td>
<td>0.19</td>
<td>0.43</td>
<td>0.73</td>
<td>1.48</td>
<td>3.12</td>
<td>6.53</td>
<td>13.71</td>
<td>20.66</td>
<td>27.60</td>
</tr>
<tr>
<td>Openmp-8t</td>
<td>0.22</td>
<td>0.36</td>
<td>0.64</td>
<td>1.20</td>
<td>2.40</td>
<td>4.92</td>
<td>10.21</td>
<td>15.57</td>
<td>20.82</td>
</tr>
<tr>
<td>Mpi-V1-8p</td>
<td>0.40</td>
<td>0.74</td>
<td>1.46</td>
<td>3.54</td>
<td>7.61</td>
<td>14.87</td>
<td>29.48</td>
<td>44.89</td>
<td>60.32</td>
</tr>
<tr>
<td>Mpi-V2-1p</td>
<td>0.55</td>
<td>1.13</td>
<td>2.34</td>
<td>5.71</td>
<td>13.15</td>
<td>27.97</td>
<td>57.47</td>
<td>87.40</td>
<td>136.39</td>
</tr>
<tr>
<td>Mpi-V2-2p</td>
<td>0.31</td>
<td>0.59</td>
<td>1.20</td>
<td>2.43</td>
<td>5.89</td>
<td>13.45</td>
<td>28.46</td>
<td>43.54</td>
<td>58.24</td>
</tr>
<tr>
<td>Mpi-V2-4p</td>
<td>0.19</td>
<td>0.34</td>
<td>0.64</td>
<td>1.26</td>
<td>2.57</td>
<td>6.13</td>
<td>13.93</td>
<td>21.51</td>
<td>29.28</td>
</tr>
<tr>
<td>Mpi-V2-6p</td>
<td>0.16</td>
<td>0.26</td>
<td>0.47</td>
<td>0.89</td>
<td>1.73</td>
<td>3.69</td>
<td>8.86</td>
<td>14.18</td>
<td>19.43</td>
</tr>
<tr>
<td>Mpi-V2-8p</td>
<td>0.13</td>
<td>0.20</td>
<td>0.35</td>
<td>0.65</td>
<td>1.28</td>
<td>2.56</td>
<td>6.15</td>
<td>10.05</td>
<td>13.92</td>
</tr>
</tbody>
</table>

Table 1: Final results (in seconds). Mpi-V1 refers to our initial distributed memory implementation and MPI-V2 refers to our final implementation.