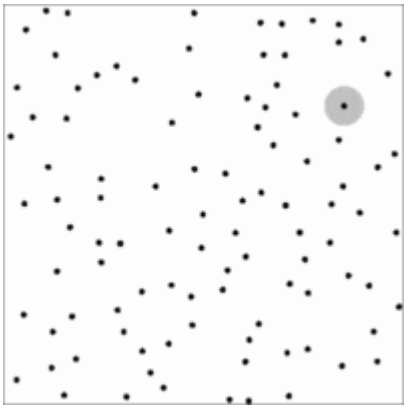
CSC581 Assignment 3: Optimizing and Parallelizing a Particle Simulation

Brandon Barker, Akash Kumar, Andrew Valenci

1. Problem Statement

The goal of this project is to optimize a toy particle simulator and parallelize it using OpenMP.



2. Modified Files

For this assignment, 4 files were modified:

- common.cpp
- common.h
- serial.cpp
- openmp.cpp

3. Optimizing the Serial Code

3.1 Identifying the Problem

In order to optimize the serial code, we had to identify the source of the inefficiency. Reading through common.cpp and common.h, we found nothing unusual, but going over it helped us to understand the code better. Looking at serial.cpp, at first we found nothing unusual there either. Then, our class professor, Dr. Linh Ngo, helped point us in the right direction.

The problem was with the nested for loop where we apply the forces between particles:

```
for( int i = 0; i < n; i++ )
{
    particles[i].ax = particles[i].ay = 0;
    for (int j = 0; j < n; j++ )
        apply_force( particles[i], particles[j],&dmin,&davg,&navg);
}
```

This nested for loop is very inefficient because it applies forces between each particle and every other particle! Since the range of interaction forces (*cutoff*) of each particle is 0.01, each particle only interacts with its neighboring particles. Considering this, it is highly unnecessary and a complete waste of computational resources to apply forces between particles that do not interact with each other. So how do we determine which particles are next to each other and only apply forces between those particles? This is where the spatial hash comes in handy.

3.2 Creating the Spatial Hash Data Structure

Spatial hashing, or locality-sensitive hashing, is used to determine the locality of an object on a 2D or 3D grid. It uses a "bucket" based system where the grid is divided into small buckets and objects are inserted into distinct buckets to later be queried against. Since spatial hashing is a flat data structure, it does not require much computational power, but it can provide a lot of false positive results if the object density is too big. For this project, we don't have to worry about false positives because the density is set sufficiently low so that given n particles, only O(n) interactions are expected.

When we first got started creating the spatial hash, we tried making a spatial hash class in a file named *spatial_hash.cpp* and importing that to the serial code so that we could create spatial hash objects from it. But we quickly encountered errors with the *Makefile* when running the `make` command. So, instead of a spatial hash class, we decided to create spatial hash functions within *common.cpp* and import them into *serial.cpp* using *common.h*.

At the bottom of *common.cpp*, we added a function to make a spatial hash:

```
double bin_size;
int bin_count;   // Number of bins per row/col in spatial hash.
// Represent a 2D particle grid as a 1D vector of particle bins.
std::vector<bin_t> particle_bins;

// A function that is used to create a spatial hash
// given particles of type particle_t.
void make_spatial_hash(int n, particle_t* particles)
{
    bin_size = cutoff * 2;
    bin_count = int(size / bin_size) + 1;

    particle_bins.resize(bin_count * bin_count);

    // Put each particle into its appropriate bin
    // in the 1D particle_bins vector.
    for (int i = 0; i < n; i++)
    {
        bin_particle(particles[i]);
    }
}
```

We set the `bin_size` to be the diameter of the interactions forces of a particle. To get the `bin_count`, which is the number of bins per row/column in the spatial hash, we simply divided the size of the grid (`size`) by the

size of each bin (`bin_size`). Then the total number of bins is `bin_count * bin_count` and this becomes the size of the `particle_bins` vector. Note the type of the `particle_bins` vector. It is a vector of `bin_t` objects. `bin_t` is a vector of `particle_t` objects defined as a new type in *common.h* as

```
typedef std::vector<particle_t> bin_t; // spatial hash bin.
```

This type is used to more easily represent bins in the spatial hash.

Once the `particle_bins` vector has been resized, we use a for loop to bin each particle into the spatial hash using the `bin_particle` function.

```
// A function that is used to add a particle to the spatial hash.
void bin_particle(particle_t& particle)
{
    int x = particle.x / bin_size;
    int y = particle.y / bin_size;
    particle_bins[x*bin_count + y].push_back(particle);
}
```

Now all we needed was a function to efficiently compute particle forces between bins. For this, we created a function called `compute_bin_forces`:

```
// A function that is used to compute all particle forces in a spatial hash bin.
void compute_bin_forces(int grid_row, int grid_col,
                        double& dmin, double& davg, int& navg)
{
    bin_t& bin = particle_bins[grid_row*bin_count + grid_col];
    // Reset acceleration of all particles in the bin.
    for (int k = 0; k < bin.size(); k++)
        bin[k].ax = bin[k].ay = 0;
    // Each non-edge particle bin is surround by 8 bins.
    // Apply forces between particles in the current bin
    // and particles in the surrounding 8 bins as well as
    // between particles in the same bin.
    for (int dx = -1; dx <= 1; dx++)
    {
        for (int dy = -1; dy <= 1; dy++)
        {
            if (grid_row + dx >= 0 && grid_row + dx < bin_count &&
                grid_col + dy >= 0 && grid_col + dy < bin_count)
            {   // bin2 represents one of the surrounding 8 bins.
                bin_t& bin2 = particle_bins[(grid_row+dx)*bin_count + grid_col +
dy];
                // Apply particle forces between the two bins.
                for (int i = 0; i < bin.size(); i++)
                    for (int j = 0; j < bin2.size(); j++)
                        apply_force(bin[i], bin2[j], &dmin, &davg, &navg);
            }
```

```
        }
    }
}
```

3.3 Optimizing the Serial Code

In our serial code, after we initialize the particles using `init_particles(n, particles)`, we call `make_spatial_hash(n, particles)` to create the spatial hash.

With our newly created spatial hash, we compute forces by spatial bins instead of individual particles.

```
//
// Compute forces
//
for (int grid_row = 0; grid_row < bin_count; grid_row++)
{
    for (int grid_col = 0; grid_col < bin_count; grid_col++)
    {
        compute_bin_forces(grid_row, grid_col, dmin, davg, navg);
    }
}
```

Then we move the particles. As we move them, we keep track of which particles move out of their original bins by storing them in a variable `changed` of type `bin_t`.

```
//
// Move particles
//
for (int grid_row = 0; grid_row < bin_count; grid_row++)
{
    for (int grid_col = 0; grid_col < bin_count; grid_col++)
    {
        bin_t& bin = particle_bins[grid_row*bin_count + grid_col];
        int tail = bin.size(), i = 0;
        while (i < tail)
        {
            move(bin[i]);                     // Move particle.
            int x = int(bin[i].x / bin_size); // Check the position.
            int y = int(bin[i].y / bin_size);
            if (x == grid_row && y == grid_col) // Still inside original bin.
                i++;
            else
            {
                changed.push_back(bin[i]); // Store particles that have changed
bins.
                bin[i] = bin[--tail]; // Remove the particle from the current bin.
            }
        }
        bin.resize(i); // Remove duplicates and shrink the bin.
```

```
    }
}
```

Finally, we rebin the particles in `changed` in their respective bins in the spatial hash.

```
for (int i = 0; i < changed.size(); i++)
{
    int x = int(changed[i].x / bin_size);
    int y = int(changed[i].y / bin_size);
    particle_bins[x*bin_count + y].push_back(changed[i]);
}
changed.clear();
```

This three-step process of applying bin forces, moving particles, and rebinning occurs for `NSTEPS` times or 1000 times in the main for loop.

3.4 serial.cpp Grade

After all that optimizing, we managed to get a serial grade of 100.00. Before, the code ran in O(n^2) time, which is unacceptably inefficient. Now, the code runs in about O(n^1.06) time, which is very close to linear time.

```
rm: cannot remove ‘serial_sum.txt’: No such file or directory
n = 500, simulation time = 0.07854 seconds
n = 1000, simulation time = 0.162432 seconds
n = 2000, simulation time = 0.338598 seconds
n = 4000, simulation time = 0.701901 seconds
n = 8000, simulation time = 1.50626 seconds

Serial code is O(N^slope)
Slope estimates are : 1.048336 1.059738 1.051694 1.101631
Slope estimate for line fit is: 1.063423
Serial Grade =  100.00
```

5. Parallelizing the Serial Code using OpenMP

Optimizing the OpenMP Code

Since the OpenMP code had the same problem of inefficient force applications as the serial code, much of the optimizations that were applied to the serial code was reused in the OpenMP code, with some slight modifications to support parallelization.

The key modification was in the use of multiple changed particle bins of type `bin_t` (one per thread) stored in the `thread_bins` vector. The `thread_bins` vector is initially resized by the master thread depending on the number of threads:

```
#pragma omp master // This code is only executed once by the master thread.
{
    numthreads = omp_get_num_threads();
    thread_bins.resize(numthreads);
}
```

Then, the bin forces are applied, with each thread performing partical reduction on addition operations on navg and davg to reduce computations.

```
//
//  compute all forces
//
// Each thread performs partial reduction on addition operations on navg and davg.
#pragma omp for reduction (+:navg) reduction(+:davg)
for (int grid_row = 0; grid_row < bin_count; grid_row++)
{
    for (int grid_col = 0; grid_col < bin_count; grid_col++)
    {
        compute_bin_forces(grid_row, grid_col, dmin, davg, navg);
    }
}
```

The particles are then moved, with the particles that move out of their original bins being stored in their respective thread's `changed` bin.

```
//
//  move particles
//
int id = omp_get_thread_num();
bin_t& changed = thread_bins[id];
changed.clear(); // No particles changed bins yet.
#pragma omp for
for (int grid_row = 0; grid_row < bin_count; grid_row++)
{
    for (int grid_col = 0; grid_col < bin_count; grid_col++)
    {
        bin_t& bin = particle_bins[grid_row*bin_count + grid_col];
        int tail = bin.size(), i = 0;
        while (i < tail)
        {
            move(bin[i]);                    // Move particle.
            int x = int(bin[i].x / bin_size); // Check the position.
            int y = int(bin[i].y / bin_size);
            if (x == grid_row && y == grid_col) // Still inside original bin.
                i++;
            else
            {
                changed.push_back(bin[i]); // Store particles that have changed
```

```
bins.
                bin[i] = bin[--tail]; // Remove the particle from the current bin.
            }
        }
        bin.resize(i); // Remove duplicates and shrink the bin.
    }
}
```

Then the rebinning of all the changed particles is done by the master bin.

```
#pragma omp master // This code is only executed once by the master thread.
{
    for (int i = 0; i < numthreads; i++)
    {
        changed = thread_bins[i];
        // Put the particles that have changed bins
        // into their respective bins.
        for (int j = 0; j < changed.size(); j++)
        {
            int x = int(changed[j].x / bin_size);
            int y = int(changed[j].y / bin_size);
            particle_bins[x*bin_count + y].push_back(changed[j]);
        }
    }
}
```

Finally, to synchronize all the threads at the end of the main for loop in preparation for the next iteration, we added `#pragma omp barrier`. Without this OpenMP section, we get a segmentation fault when we run the code with more than one thread.

openmp.cpp Grade

After applying the above-mentioned parallelizations and optimizations to the code, we managed to get an OpenMP grade of 82.42. Note the speedup as the number of threads increases.

```
n = 500, simulation time = 0.08331 seconds
n = 500,threads = 1, simulation time = 0.086536 seconds
n = 500,threads = 2, simulation time = 0.051989 seconds
n = 500,threads = 4, simulation time = 0.041003 seconds
n = 500,threads = 8, simulation time = 0.035748 seconds
n = 500,threads = 16, simulation time = 0.032117 seconds
n = 1000,threads = 2, simulation time = 0.094091 seconds
n = 2000,threads = 4, simulation time = 0.130803 seconds
n = 4000,threads = 8, simulation time = 0.177745 seconds
n = 8000,threads = 16, simulation time = 0.394284 seconds

Strong scaling estimates are :
    0.96    1.60    2.03    2.33    2.59 (speedup)
    0.96    0.80    0.51    0.29    0.16 (efficiency)    for
```

```
        1       2       4       8      16 threads/processors

Average strong scaling efficiency:    0.55

Weak scaling estimates are :
    0.96    0.89    0.64    0.47    0.21 (efficiency)    for
       1       2       4       8      16 threads/processors

Average weak scaling efficiency:    0.63


openmp Grade =   82.42
```

6. Conclusion

We found that by implementing a spatial hash data structure on the serial and OpenMP code, we were able to avoid unnecessary force applications between particles that were too far to interact with each other, thus greatly reducing computations and making the simulations run more efficiently. In the case of the OpenMP code, the use of one `changed` bin per thread as opposed to a single `changed` bin proved to be helpful in the parallelization of the spatial hash. However, considering that we did not get a full grade on the OpenMP code, there is still room for future improvement.

7. References

- *Quad-Tree vs. Spatial Hashing* by Brandon Barker
- *Optimization of Large-Scale, Real-Time Simulations by Spatial Hashing* by Erin J. Hastings, Jaruwan Mesit, Ratan K. Guha
- OpenMP API 4.5 C/C++
- Shared Memory Parallelism Video Lecture