EE4371 Data Structures and Algorithms Final Q2 2019

Rishhanth Maanav V
EE16B036

1. Part A Naive CPU Algorithm

Algorithm 1 Algorithm for Naive CPU fluid simulation

```
1: Input SET P of all particles, no of time steps T, time step duration dt
2: Output SET P of particles with updated positions, velocities and acceleration
3: function CalcAcc(PARTICLE a, PARTICLE b)  ▷ calculates force on particle a due to particle b and adds to acceleration of particle a
4: function MoveAllParticles(SET of particles P, time dt)  ▷ moves all particles and updates velocities using their current position, velocity and acceleration and time dt.
5: while T do
6:     for each particle i in set P do
7:         for each other particle j in P do
8:             if i $\neq$ j then
9:                 CalcAcc(i,j)
10:    MoveAllParticles(P,dt)
11:    T := T - 1
```

Total nested loop runs (for N = 10^8, T = 10^4)
= N * N * T
= $10^8 * 10 * 8 * 10 * 4 = 10^{20}$

Total times function MoveAllParticles runs = T = 10^4

Complexity of function MoveAllParticles = N = 10^8 (because every particle has to be moved)

Total order of operations used is $10^20 + 10^8 * 10^4 = O(10^{20})$.

If each operation takes 10 floating point operations steps (on an average), we get the total compute steps needed to run for the simulation to complete to be 10^{21}. Also each floating point operation takes 10 ns = 10^{-8} sec.

Total time taken for the simulation is: $10^{21}.10^{-8} = 10^{13} sec \approx 3,00,000\ yrs = O(10^5\ yrs)$!! As we can see this is computationally infeasible.

2. Part B 4-way Subtree Pseudocode

We present the pseudo-code to construct a 4-way tree by inserting particles at the leaves. This tree has depth 10 as given in the question. We follow a doubly linked-list based approach where every element whose parent is the same are doubly linked with other elements having the same parent. Essentially, using a doubly linked-list based approach, we avoid tree traversal because we can just traverse through the doubly linked list to model the interactions among particles having the same parent.

```
# function to insert particle to tree
void build_tree(NODE* root,PARTICLE* p)
{
    NODE* temp = root

    #BINARY(p.position)[i]
    #returns bin_x[i],bin_y[i]
    for i in 0 to 10
    {
        if(BINARY(p.position)[i]=='00'))
        {
            temp = temp->branch_1
        }

        else if(BINARY(p.position)[i]=='10'))
        {
            temp = temp->branch_2
        }

        else if(BINARY(p.position)[i]=='01'))
        {
            temp = temp->branch_3
        }

        else
        {
            temp = temp->branch_4
        }
    }
    #now insert the particles
    PARTICLE* ptemp = temp->particle_list;
    # no particles already
    if(ptemp is NULL)
    {
        temp->list = p;
    }

    # double link with other particles already
    else
    {
        while(temp->next != NULL)
        {
            ptemp = ptemp->next;
        }
        ptemp->next = p;
        p->prev = ptemp;
    }
}
```

Now we present the pseudocode for calculating the acceleration once the tree has been built and hence particles having the same parent are doubly linked.

```
# func to calculate acceleration of p
void calc_acc(PARTICLE* p)
{
    p->acc[0] = 0
    p->acc[1] = 0
    PARTICLE* temp = p->next
    # forward
    while(temp is not NULL)
    {
        # calculate force using repulsion()
        acc = repulsion(p,temp)
        p->acc[0] = p->acc[0] + acc[0] #x
        p->acc[1] = p->acc[1] + acc[1] #y
        temp = temp->next;
    }

    temp = p->prev;
    # backward
    while(temp is not NULL)
    {
        acc = repulsion(p,temp)
        p->acc[0] = p->acc[0] + acc[0] #x
        p->acc[1] = p->acc[1] + acc[1] #y
        temp = temp->prev;
    }

    p->acc[0] = p->acc[0]/mass;
    p->acc[1] = p->acc[1]/mass;
}

#func to simulate one step
void sim_one_step(PARTICLE** all_particles)
{
    for p in all_particles
    {
        # update pos using vel and acc
        update_pos(p->pos,p->vel,p->acc)
        update_vel(p->vel,p->acc)
        # reflect if particle goes out
        reflect_particle(p)
    }
    break_all_double_links(all_particles)
}
```

3. Time Complexity

Here we only compare for each particle, only particle associated with it in a leaf. So, if there are K leaves in the tree, then we can expect on an average we have n/K elements per leaf. Therefore we need to compare only n/K elements. If we take T timesteps, then the order of this algorithm is $O(n^2T/K)$ when compared to that of CPU method which is $O(n^2T)$.

3.1. Configuration of simulation

Particles = 10^5 particles
Number of time steps = 10^3 timesteps
Bit precision = 7 bits
Number of leaf nodes = $4^7 = 16384$ nodes

3.2. Execution time theory vs practical

For the given configuration,

Total time to run all steps = $10^5 * 10^5 * 10^3 * 10/4^7$
= 6103515625 nsec = 61.03 sec

Time taken to run in (in PC): 73.032 sec

The time taken calculated theoretically and obtained practically agree.

3.3. Algo a vs Algo b

Algorithm b is better than algorithm a looking at the time complexity. Rather than comparing each particle (n) with every other particle (n-1), we limit ourselves to the interactions of particles within a leaf of the tree constructed which is ($n/4^r$) where r is the binary bit resolution. However, if r is very large, we limit ourselves to the interaction between very few particles and the simulation with algorithm b is bound to give wrong results. But by setting the right value of r, we can get close to perfect results with algorithm b. Algorithm a, as we have seen is too much computationally intensive.

3.4. Speed Up in Algorithm b

On an average going by the order of complexity, algorithm b has a much lesser time complex than algo a. However, in practise we can observe good speed up by setting a higher value of the resolution of binary representation. We must also make sure that the resolution isn't too high as there might be too less particles interacting.

4. Sorting Technique for Binning

Size of L2 cache per core: 256KB

No. of cores: 2

Total L2 cache = 2x256KB = 512KB

For binning, the most suitable algorithm is **bucket sort**. In bucket sort, the input range of values are divided into several bins where each bin is a sub-array of the input whose values lie in the range of that bin. Recursively applying the same sorting technique on each bins, we will have each bin sorted and hence the entire array sorted.

For the task of binning, we just need to carry out the first step of the bucket sort algorithm. This is because we just need the count of values within a bin. We don't need to sort the entire array for this purpose.

5. Plots of Histograms obtained at Initialisation and after Final step

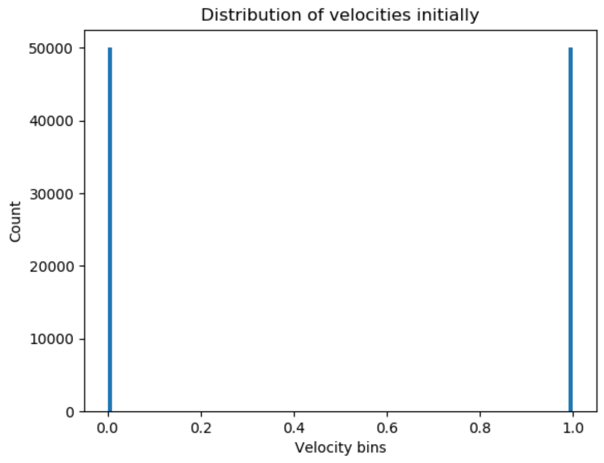

Figure 1. Histogram at initialisation

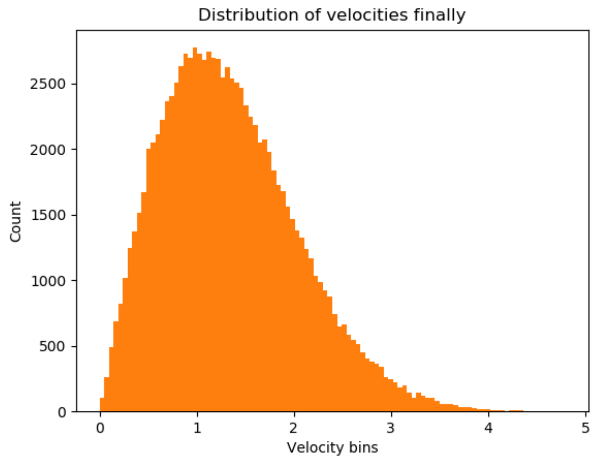

Figure 2. Histogram after final step