Université de Liège

MATH0471-A-a : Multiphysics integrated computational project

Deadline 1 : linked list method and kernel

KOLLASCH Killian s202793
SANTORO Luca s201807

Professor: C. Geuzaine
Professor: R. BOMAN

Academic year : 2023-2024

I Implementation of the linked list algorithm

The primary concept of this approach, as outlined in L. Goffin's thesis, involves enclosing all particles within cells (subdomains) and exclusively examining the neighbours of a given particle in adjacent cells. These cells have dimensions equal to or greater than the neighbouring radius κh. Currently, κ is set to 1 and h is set to 1.2s. The choice of κ will be either 1, 2, or 3 depending on the kernel selected for computation.

One first needs to determine the good number of cells in each direction x,y,z of the domain (assuming a 3D rectangular domain). Given $L[0], L[1], L[2]$ (denoted L_0, L_1, L_2 lately) being respectively the lengths of the domain in the x,y and z direction, the number of cells in a prescribed direction is given by:

$$N_{x,y,z} = \left\lfloor \frac{L_{0,1,2}}{\kappa h} \right\rfloor. \tag{I.1}$$

For instance, if $L_1 = 1.0$ and $s = 0.25$:

$$N_x = \left\lfloor \frac{1}{1 * 1.2 * 0.25} \right\rfloor = 3. \tag{I.2}$$

In this example, each cell will be referenced by three indexes $(i, j, k) \quad \forall i, j, k \in \{0, 1, 2\}$.

While the correct number of cells in each direction is set, one first stores for each particle their corresponding cell using three lists *particle_i, particle_j, particle_k*. For instance, the last generated particle located at $(x, y, z) = (L_0, L_1, L_2)$ is in the last cell. This is written as:

$$(\textit{particle\_i[last], particle\_j[last], particle\_k[last]}) = (2, 2, 2). \tag{I.3}$$

One first iterates on each specific particle (labelled *particle_x*) and seeks its corresponding cell. Once the specific cell is located, one seeks over the neighbouring cells. But one needs to pay attention if an edging cell is encountered. This is done by the following coding lines:

```
            // Define neighbouring cell indices
            unsigned i_inf = (i_cell == 0) ? 0 : i_cell - 1;
            unsigned i_supp = (i_cell == Nx - 1) ? i_cell : i_cell + 1;

            unsigned j_inf = (j_cell == 0) ? 0 : j_cell - 1;
            unsigned j_supp = (j_cell == Ny - 1) ? j_cell : j_cell + 1;

            unsigned k_inf = (k_cell == 0) ? 0 : k_cell - 1;
            unsigned k_supp = (k_cell == Nz - 1) ? k_cell : k_cell + 1;

```

Code Listing 1: Edging cell check

Next, one can accurately iterate through neighbouring cells and then iterate through all other particles to identify the corresponding neighbours. The three lists stored earlier are utilised for this purpose. Subsequently, when a particle is encountered in a neighbouring cell (or in the same cell as *particle_x*), their relative distance is computed. If this value r_{ab} is less than κh, the particle is added to the neighbour list of *particle_x* (in a matrix labelled *neighbours_matrix*). This is done by the following coding lines:

```
            // Iterate over all 26 adjacents cells to find neighbours
        for (unsigned i = i_inf; i <= i_supp; i++){
            for (unsigned j = j_inf; j <= j_supp; j++){
                for (unsigned k = k_inf; k <= k_supp; k++){

                    // Iterate over all particles to find the corresponding
    neighbours
```

```
                    for (unsigned l = x+1; l < particle_i.size(); l++){
                        if (particle_i[l] == i){
                            if (particle_j[l] == j){
                                if (particle_k[l] == k){
                                    double rx, ry, rz, r2;
                                    rx = (particle_x[x] - particle_x[l] )*(particle_x[x] - particle_x[l] );
                                    ry = (particle_y[x] - particle_y[l] )*(particle_y[x] - particle_y[l] );
                                    rz = (particle_z[x] - particle_z[l] )*(particle_z[x] - particle_z[l] );
                                    r2 = rx + ry + rz;

                                    if(r2<= kappa*kappa*h*h){
                                        neighbours_matrix[x].push_back(l);
                                        neighbours_matrix[l].push_back(x);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }

```

Remark: To enhance the process, one have considered the reciprocity between two neighbouring particles. If *particle_1* is a neighbour of *particle_2*, the reverse statement holds true, and thus, the associated neighbour should be added for both particles. To incorporate this enhancement and prevent iterating over particles that have already been identified as neighbours, the loop (over particles) starts at **unsigned l = x + 1**. In other words, one does not iterate over all previous values of x.

II Validation and performance testing

One aims to assess the efficiency of the specified linked list algorithm. To achieve this, a basic algorithm has been implemented to identify the neighbours for all particles. This algorithm involves a straightforward pairwise comparison of each particle with every other particle. This process is carried out through the following lines of code:

```
            // Find neighbours for each particle
            for (unsigned i = 0; i < particle_x.size(); i++){
                for (unsigned j = i+1; j < particle_x.size(); j++){

                    double rx, ry, rz, r2;
                    rx = (particle_x[i] - particle_x[j] )*(particle_x[i] - particle_x[j] );
                    ry = (particle_y[i] - particle_y[j] )*(particle_y[i] - particle_y[j] );
                    rz = (particle_z[i] - particle_z[j] )*(particle_z[i] - particle_z[j] );
                    r2 = rx + ry + rz;
                    if(r2<= kappa*kappa*h*h){
                        neighbours_matrix[i].push_back(j);
                        neighbours_matrix[j].push_back(i);
                    }
                }
            }

```

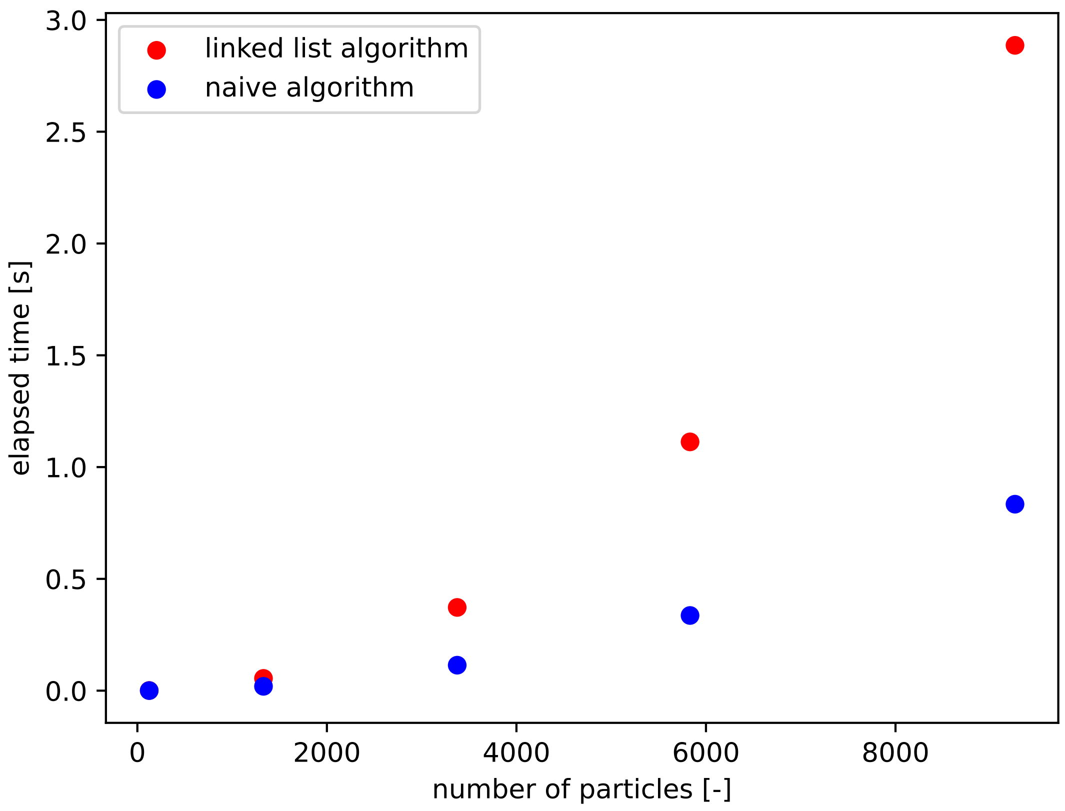

Figure 1: Time performance comparison of the linked list and naive algorithm.

The performance testing employed is rather straightforward and may not be the most appropriate method. The approach involves simply measuring the time before and after the algorithms are invoked, and then calculating the difference between these values to determine the time elapsed in each function.

As can be seen in Fig. 1, the poor performances of the linked list method are not expected. One needs to investigate further to determine the reason of these results. Either our implementation is not optimal at all, either higher number of particles have to be used to take advantage of this specific method.

III Kernel functions

The various kernel functions are implemented in the Kernel_function.cpp code. They will be utilised in the coming months for calculating the Smoothed Particle Hydrodynamics (SPH) method. These functions will be compared to each other in different simulations to observe how the solution is influenced by the choice of kernel.

IV What to do next

Several improvements need to be made to the linked-list algorithm with the aim of enhancing time performance and optimizing the code to reduce the surplus of memory currently being utilized.

Utilizing better tools to compare both the linked-list and naive algorithms will help in understanding the benefits of the former.