P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2018

Set 12 – Particle Strength Exchange, Cell Lists

Issued: December 14, 2018
Hand in (optional): December 24, 2018 23:59

Question 1: 2D Diffusion using Particle Strength Exchange (PSE)

Consider a scalar two-dimensional field $\phi(x, y, t)$ defined on a periodic domain $(x, y) \in [0, 1)^2$. We want to utilize the PSE method to solve the diffusion problem:

$$\frac{\partial \phi}{\partial t} = \nu \left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right), \tag{1}$$

for some given initial condition $\phi(x, y, t = 0)$ and diffusion constant ν.

Instead of discretizing the field ϕ on a grid, we will use a collection of N *particles*. A particle represents a small "volume" of the field, and is defined by its position $\mathbf{x}_i$ and field value $\phi_i = \phi_i(t)$. In this exercise we assume the volume of each particle is equal ($V_i = V_{\text{total}}/N = 1/N$).

We rewrite Eq. (1) as a system of equations on particles:

$$\frac{\mathrm{d}\phi_i}{\mathrm{d}t} = \frac{\nu}{\varepsilon^2} \sum_j V_j \left(\phi_j - \phi_i\right) \eta_\varepsilon(\mathbf{x}_j - \mathbf{x}_i), \tag{2}$$

where $\eta_\varepsilon(\mathbf{r})$ is a kernel representing the laplacian operator, and ε a scale constant. In this exercise we consider the following kernel:

$$\eta_\varepsilon(\mathbf{r}) = \frac{1}{\varepsilon^2} \eta\left(\mathbf{r}/\varepsilon\right), \qquad \eta(\mathbf{r}) = \frac{4}{\pi} e^{-|\mathbf{r}|^2}. \tag{3}$$

a) You are given a skeleton code that initializes the particle positions and values. As a placeholder for the final timestep, the skeleton code simply decays the values ϕ_i in time. Get familiar with the skeleton code. Use `make run` and `make plot` to run the code and test the plotting script.

   Note: the plotting script requires numpy, matplotlib and ffmpeg.

b) Extend the provided skeleton code to compute the right-hand side of Eq. (2) using Eq. (3) for the kernel η_ε. Account for a periodic domain when computing the distance between two particles!

   The tricky part in the implementation are the ε factors in the Eqs. (2) and (3). There should be a factor of ε^{-2} in the exponential function of η, a factor of ε^{-2} due to the dimensionality of the system (2D), and a factor of ε^{-2} due to the order of the derivative we want to compute (2nd order).

To implement periodic boundaries, we assume that the kernel is small and that it is enough to consider the extended domain $[-1, 2)^2$ composed of 9 copies of the original domain. However, instead of actually making 9 copies of the domain (of the particles), we implement a "periodic distance function" $d(\mathbf{x}_i, \mathbf{x}_j)$, which considers 9 copies of the second argument and picks the closest one. It can be easily shown that x and y coordinate can be considered separately, such that $d(\mathbf{x}_i, \mathbf{x}_j) = d_{1D}(x_i, x_j) + d_{1D}(y_i, y_j)$. Then, the function $d_{1D}(x_i, x_j)$ can be implemented as

$$d_{1D}(x_1, x_2) = \min\{|x_1 - x_2|, 1 - |x_1 - x_2|\}.$$

The solution code provides a slightly different (and, in fact, potentially less efficient) formulation of the same function.

Tip: In the implementation of Eq. (2), it is not necessary to skip the case when $i = j$, because the expression inside the summation contains a factor $(\phi_i - \phi_j)$ and the kernel η_ε is finite for $i = j$.

Points:

- **5 for handling the periodicity of the domain (only 3 pts if you actually made copies of particles),**
- **10 for correctly implementing the summation and kernels.**

c) Implement the forward Euler scheme for the integration of the Eq. (2).

If RHS_i denotes the summation computed in the previous subquestions for the particle i, then the forward euler scheme is implemented as:

$$\phi_i \leftarrow \phi_i + \Delta t \cdot RHS_i.$$

Note that all RHS_i must be computed before updating any ϕ_i.

Points:

- **6 for implementing the forward Euler scheme (only 3 pts if you updated ϕ_i while computing RHS_i).**

d) Considering the domain size and the number of particles, what is qualitatively the distance h between neighboring particles? Parameter ε determines the spread of the kernel. Run the code for different values of ε. Experiment with $\varepsilon \ll h$, $\varepsilon \approx h$ and $\varepsilon \gg h$. What do you observe for different cases?

If $\varepsilon \ll h$, the kernel fades out too quickly and there is no interaction between the particles. If $\varepsilon \gg h$, the kernel is so wide that it does not even differentiate particles according to their distance. In practice it means that all particles initially set to $\phi_i = 0$ will increase at the same rate, independent on their position, and all particles initially set to $\phi_i = 1$ will decrease at the same rate. The case $\varepsilon \approx h$ shows the desired behavior of the diffusion kernel.

Note: in the case $\varepsilon \gg h$, strictly speaking, the 9 copies of the domain we consider are not enough to correctly simulate the domain periodicity. However, it does not qualitatively change the results. If you implemented a more complex treatment of the periodicity for this case, the solution is also acceptable.

Points:

- **9 for the analysis (3 pts for each case).**

Question 2: Cell lists (optional)

Cell lists are a data structure used to speed up the computation of short-range pairwise interactions between particles. In Question 1 the implementation of Eqs. (2) and (3) required $O(N^2)$ operations each timestep. Cell lists enable us to optimize the complexity to $O(N)$.

First, notice that the exponent in $\eta_\varepsilon(\mathbf{r})$ quickly vanishes as $|\mathbf{r}|$ is increased. For $\varepsilon = 0.05$ and $|\mathbf{r}| = 0.30$, we have $\eta_\varepsilon(\mathbf{r}) \approx 3 \times 10^{-16}$, which is close to the machine precision. Thus, if we set a distance threshold of $\Delta = 6\varepsilon$ and consider only interactions for pairs where $|\mathbf{x}_i - \mathbf{x}_j| < \Delta$, the result we get is virtually the same. We can even sacrifice a bit of accuracy and choose a threshold of $\Delta = 4\varepsilon$, such that $\eta_\varepsilon(\Delta) \approx 1.4 \times 10^{-7}$.

Now when we replace the long-range interactions with short-range interactions, in order to get the performance improvement, we need to be able to efficiently find all pairs of nearby particles. This is where cell lists come into play.

The idea is to split the domain into square cells whose side length is equal to Δ. For example, for $\varepsilon = 0.05$ and $\Delta = 4\varepsilon = 0.20$, the domain would be split into 25 cells. Because the cell size is exactly Δ, a particle i belonging to a cell k can only interact with particles of cell k and 8 neighboring cells (in 3D a cell would have 26 neighbors).

This way we reduced the number of (ordered) pairs we consider from N^2 to $9NK$, where K is the number of particles per cell. Thus, the complexity is now linear with the size of the problem[1]!

Your goal is to implement cell lists.

a) First, implement the data structure itself. Instead of storing all particles in one array, create a matrix of cells, and redistribute the particles into those cells.

   The cell-based data structure for particles can be implemented in many ways. We mention two variations here.

   One way, described in the question itself, is to store different arrays of particles for different cells. The benefit of this structure is that addition or removal of particles is fairly easy (as described in the last subquestion). However, we have to be careful with how we allocate the memory, as we could easily end up with high memory fragmentation and thus poor cache performance. Namely, to take advantage of automatic memory prefetching done by the CPU, we want our data to be stored continuously in memory.

   Another way to implement the cell data structure is to have one particle container for all cells, such that the particles are sorted with respect to the cell ID. In other words, particles of one cell are stored continuously as a chunk of the larger array. This way, the memory fragmentation is completely removed. On the other hand, the additional and removal of particles requires rebuilding the whole array.

   In either case, if we want to utilize vectorization, each cell should store its particle in structure-of-arrays (SoA) format (the mixed storage such as `soa<4>` in ISPC would also work). Moreover, all arrays should be allocated with proper alignment (32 or 64 bytes). Thus, it is probably easier to use pointer-based arrays with `posix_memalign` instead of `std::vector`.

[1] Value K is determined by the nature of the problem (interaction range) and numerics (ε and h), but it is independent of the size of the domain. We therefore say that cell lists optimize $O(N^2)$ to $O(N)$ complexity, even though the underlying constant is large.

b) Implement the traversal of each pair of nearby particles, i.e. all particles whose distance is less than Δ. In other words, implement a traversal over each particle i and each nearby particle j. Do not forget about periodic boundaries!

To compute RHS_i, instead of including all N particles in the summation, we use cell lists to sum the contribution of nearby particles only. First, to traverse over all particles i, we traverse over all cells, and then over all particles within each cell. Denote with C_X and C_Y the number of cells in x and y dimension, respectively. We iterate over all $c_y \in \{0, 1, \ldots, C_Y - 1\}$, then over all $c_x \in \{0, 1, \ldots, C_X - 1\}$ and then over all i inside the cell (c_x, c_y). To find all neighbors j, we iterate over the 9 neighboring cells (including itself) by using two for loops s_y and s_x, where $s_x, s_y \in \{-1, 0, 1\}$.

In the case of closed boundaries (non-periodic), we would skip pairs (s_x, s_y) that point outside of the domain. In the periodic case, we can for example define the neighboring cell (c'_x, c'_y) as $c'_x = (c_x + s_x + C_X)\%C_X$ and $c'_y = (c_y + s_y + C_Y)\%C_Y$. Moreover, to simplify the computations of the distance between between a particle i from (c_x, c_y) and a particle j from (c'_x, c'_y), we define an offset vector $\mathbf{O}$, which describes which copy of (c'_x, c'_y) we are currently considering (if any). With that, the displacement vector between particles i and j is given by $\Delta\mathbf{x} = \mathbf{x}_j - \mathbf{x}_i + \mathbf{O}$. Alternatively, a common solution is to create a layer of ghost cells outside the domain, but this way we avoid copying the data. The components of $\mathbf{O}$ are equal to 0, $+L$ or $-L$, depending on the values (c_x, c_y) and (s_x, s_y). For example, for $(c_x, c_y) = (0, 5)$ and $(s_x, s_y) = (-1, 0)$, vector $\mathbf{O}$ would be equal to $(+L, 0)$. Here, L denotes the domain size ($L = 1$ in our case).

c) Test your code and compare the performance with the old $O(N^2)$ implementation. Experiment with increased size of the domain (or effectively, larger N and smaller ε).

d) If the particles are allowed to move in time (as in the case of Molecular Dynamics), they will be occasionally crossing from one cell to another. How would you implement that operation? Concretely, how do you insert a particle to a cell, and how do you remove it? Can you guarantee $O(1)$ complexity for both? Keep in mind that at some point you might want to utilize vectorization, so you need to keep the storage simple.

We explain here only the approach when cells use separate arrays for particles. To insert a particle, we simply append it to the cell array. The complexity of this operation is amortized $\mathcal{O}(1)$.

A naive implementation of removing the particle k is to shift to the left all particles stored after it in the particle array. The complexity of such approach is $\mathcal{O}(K)$, where K is the number of particles in the cell. However, because the order of particles does not matter, we can swap the particle k with the last particle in the cell and then remove the last element. This approach has complexity $\mathcal{O}(1)$.

The comment about "simple storage" and vectorization is to avoid approaches such as linked lists. Although they allow elegant removal and insertion, they add large overhead (prevent filling the CPU pipeline) and complicate vectorization.

Question 3: Brain teasers (optional)

In this question we propose a few general problems and ask you to think of some good solutions.

a) We introduced cell lists as a data structure used to improve the performance of local interaction kernels. However, our appetite has grown and now we want to extend the capabilities of our simulations to large gravitational systems. Gravity is a long-range interaction, with force proportional to $1/r^2$, where r is the distance between two mass particles.

Think of a way to optimize this kind of kernel. Hint: think like an engineer, not as a mathematician.

Contrary to the short-range interaction, in the long-range case we cannot ignore the contribution from the distant cells. However, by using some approximations we can greatly reduce the execution time. One approach would be to use cell lists to cluster points into larger heavier particles. More precisely, for each particle i, we still traverse nearby 9 cells (or even 25 or 49), and manually iterate over each particle j within those cells to compute the contribution. For other more distant cells we compute the gravity force from the whole cell to the particle i, by considering only the total mass of the cell, and not each particle separately. The complexity of this approach is $\mathcal{O}(N^2/K + NK)$. It can be shown that the optimal K is of the order of $\sqrt{N}$, making the final complexity $\mathcal{O}(N^{3/2})$.

If we are dealing with extremely large systems, we can even have a hierarchy of cells, such that for even farther particles we have larger (and thus fewer) cells.

Finally, such hierarchy and effective masses are the basis of the Fast Multiple Method (FMM) algorithm, whose running time is $\mathcal{O}(\log(1/\epsilon)N)$, where N is the number of particles and ϵ the error tolerance. FMM is used in many real simulations, from vortex-based methods in fluid dynamics to gravitational evolution of galaxies in astrophysics.

b) Assume the pairwise interaction (the kernel) is expensive to compute, and that it is symmetric with respect to the order of particles. The exponential in Eq. (3) is a good example. The symmetry allows us to reduce the number of kernel computations by one half – when we compute the interaction effect of a particle A on the particle B, we get for free the effect of B on A.

How would you incorporate this symmetry into cell lists? How does the interaction calculation algorithm look like?

For each pair of particles i and j, instead of computing the interaction twice, we want to compute it once and update both RHS_i and RHS_j immediately.

To achieve that, when traversing the neighbors of some particle i, instead of considering all 9 neighbor cells, i.e. 9 combinations of (s_x, s_y) pairs, we consider only those that are "to the right or below". We skip the 4 combinations $(s_x, s_y) \in \{(-1,-1), (-1,0), (-1,1), (0,-1)\}$. For the 4 combinations $(s_x, s_y) \in \{(1,-1), (1,0), (1,1), (0,1)\}$, we compute the interaction of all i in (c_x, c_y) and all j in (c'_x, c'_y) as usual, and just update the RHS of both particles. The remaining case $(s_x, s_y) = (0,0)$ is special, where in order to avoid counting the interaction twice, we have to limit the (i,j) pairs only to those where e.g. $j > i$.

c) The machines we are running our codes on have multiple CPU cores, and it would be a shame not to take advantage of that.

How would you parallelize the computation of interactions based on cell lists? Of course, ensure you don't have a race condition. Ignore for the moment the symmetry trick from the previous subquestion.

Now, what if we want to take advantage of all three ideas: cell lists, symmetry and parallelization? What are the problems now? Atomics and locks are one solution. Can you think of a solution that avoids atomics? Consider first the 1D case and generalize to 2D/3D.

First, the idea of atomics would be to parallelize the loop in a normal fashion, inside of which RHS_i and RHS_j are updated atomically to avoid the race condition.

Depending on the architecture of the CPU (or GPU) and the problem details, that might or might not be the optimal way. Here we mention an alternative that avoids atomics and locks, at least on the particle level. Please note that this method has not been tested, and only benchmarking would show if it is beneficial over atomics (on your specific architecture!).

Consider first the 1D case. Denote the cell ID with c, where $c \in \{0, 1, \ldots, C-1\}$. In a serial case with the symmetry-based optimization, while traversing the cell c, the code will be updating not only the cell c, but also the cell $c+1$ (for simplicity, we don't talk about periodicity here). Thus, if the loop over c is naively parallelized, a race condition might occur.

However, race condition is unlikely if C is very large and if each thread is responsible for a continuous subsequence of cells (as in the static scheduling). Namely, the race condition may happen only on the boundary between two thread regions. Till the point that the thread t reaches its last cell, the thread $t+1$ will almost surely already be done with its first cell.

To ensure there is no race condition, we can use one lock for each boundary. Concretely, lock L_t will be locked by the thread $t+1$ while it is processing its first cell, and by the thread t while it is processing its last cell. Even though this is a lock, note that each thread will perform a lock operation only twice during the whole computation. Moreover, a lock with a simple and cheap mechanism can be used, because a) there are only two threads interfering, b) almost always there will be no waiting required.

The simplest way to generalize to 2D is to parallelize with respect to rows, such that there is one lock for each pair of consecutive rows that are assigned to different threads. In 3D, this generalizes to parallelization with respect to the z index (slowest moving index), such that each thread gets a continuous subset of xy slices.

There are few drawbacks of this algorithm. First, if each thread has only one cell (or one row/slice), the performance will be poor. Secondly, we implicitly assume the distribution of particles is uniform, i.e. that the workload per cell does not differ greatly between different cells. If this is not the case, our simple static parallelization of the c for loop will cause load imbalance. Finally, in 2D and 3D this may not be the most cache-efficient solution.

An alternative would be to use one lock for each cell, which would allow for potentially more cache-efficient scheduling. However, that approach could suffer from a deadlock, because each thread is locking two cells simulatenously.

In the end, whether cache is a problem or not depends on the specific case. In 3D Molecular Dynamics simulations, where each particle takes about 100B of storage, a 30MB L3 cache can store hundreds of thousands of particles, which is already a very large number.