On the design of Monte-Carlo particle coagulation solver interface: a CPU/GPU Super-Droplet Method case study with PySDM*

Piotr Bartman[1][0000−0003−0265−6428] and Sylwester Arabas[1][0000−0003−2361−0082]

Jagiellonian University, Kraków, Poland

piotr.bartman@doctoral.uj.edu.pl, sylwester.arabas@uj.edu.pl

Abstract. Super-Droplet Method (SDM) is a probabilistic Monte-Carlo-type model of particle coagulation process, an alternative to the mean-field formulation of Smoluchowski. SDM as an algorithm has linear computational complexity with respect to the state vector length, the state vector length is constant throughout simulation, and most of the algorithm steps are readily parallelizable. This paper discusses the design and implementation of two number-crunching backends for SDM implemented in PySDM, a new free and open-source Python package for simulating the dynamics of atmospheric aerosol, cloud and rain particles. The two backends share their application programming interface (API) but leverage distinct parallelism paradigms, target different hardware, and are built on top of different lower-level routine sets. First offers multi-threaded CPU computations and is based on Numba (using Numpy arrays). Second offers GPU computations and is built on top of ThrustRTC and CURandRTC (and does not use Numpy arrays). In the paper, the API is discussed focusing on: data dependencies across steps, parallelisation opportunities, CPU and GPU implementation nuances, and algorithm workflow. Example simulations suitable for validating implementations of the API are presented.

Keywords: Monte-Carlo · coagulation · Super-Droplet Method · GPU.

1 Introduction

The Super-Droplet Method (SDM) introduced in [20] is a computationally efficient Monte-Carlo type algorithm for modelling the process of collisional growth (coagulation) of particles. SDM was introduced in the context of atmospheric modelling, in particular for simulating the formation of cloud and rain through particle-based simulations. Such simulations couple a grid-based computational fluid-dynamics (CFD) core with a probabilistic, so-called super-particle (hence the algorithm name), representation of the particulate phase, and constitute a tool for comprehensive modelling of aerosol-cloud-precipitation interactions

---

(e.g., [1,3,10,15]; see [13] for a review). The probabilistic character of SDM is embodied, among other aspects, in the assumption of each super-particle representing a multiple number of modelled droplets with the same attributes (including particle physicochemical properties and position in space). The super-droplets are thus a coarse-grained view of droplets both in physical and attribute space.

The probabilistic description of collisional growth has a wide range of applications across different domains of computational sciences (e.g., astrophysics, aerosol/hydrosol technology including combustion). While focused on SDM and depicted with atmospheric phenomena examples, the material presented herein is generally applicable in development of software implementing other Monte-Carlo type schemes for coagulation (e.g., Weighted Flow Algorithms [8] and other, see Sect. 1 in [21]), particularly when sharing the concept of super-particles.

The original algorithm description [20], the relevant patent applications (e.g., [22]) and several subsequent works scrutinising SDM (e.g., [2,9,18,25] expounded upon the algorithm characteristics from users' (physicists') point of view. There are several CFD packages implementing SDM including: SCALE-SDM [19] and UWLCM [10]; however the implementation aspects were not within the scope of the works describing these development. The aim of this work is to discuss the algorithm from software developer's perspective. To this end, the scope of the discussion covers: data dependencies, parallelisation opportunities, state vector structure and helper variable requirements, minimal set of computational kernels needed to be implemented and the overall algorithm workflow. These concepts are depicted herein with pseudo-code-mimicking Python snippets (syntactically correct and runnable), but the solutions introduced are not bound to a particular language. In contrast, the gist of the paper is the language-agnostic API (i.e., application programming interface) proposal put forward with the very aim of capturing the implementation-relevant nuances of the algorithm which are, arguably, tricky to discern from existing literature on SDM, yet which have an impact on the simulation performance. The API provides an indirection layer separating higher-level physically-relevant concepts from lower-level computational kernels.

Validity of the proposed API design has been demonstrated with two distinct backend implementations included in a newly developed simulation package PySDM [5]. The two backends share the programming interface, while differing substantially in the underlying software components and the targeted hardware (CPU vs. GPU). PySDM is free/libre and open source software, presented results are readily reproducible with examples shipped with PySDM (https://github.com/atmos-cloud-sim-uj/PySDM).

The remainder of this paper is structured as follows. Section 2 briefly introduces the SDM algorithm through a juxtaposition against the alternative classic Smoluchowski's coagulation equation (SCE). Section 3 covers the backend API. Section 4 presents simulations performed with PySDM based on benchmark setups from literature documenting CPU and GPU performance of the implementation. Section 5 concludes the work enumerating the key points brought out in the paper.

2   SDM as compared to SCE

2.1   Mean-field approach: Smoluchowski's coagulation equation

Population balance equation which describes collisional growth is historically known as the Smoluchowski's coagulation equation (SCE) and was introduced in [23,24] (for a classic and recent overviews, see e.g. [7] and [14], respectively). It is formulated under the mean-field assumptions of sufficiently large well-mixed system and of neglected correlations between numbers of droplets of different sizes (for discussion in the context of SDM, see also [9]).

Let function $c(x,t) : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$ correspond to particle size spectrum and describe the average concentration of particles with size defined by x at time t in a volume V. Smoluchowski's coagulation equation describes evolution of the spectrum in time due to collisions. For convenience, t is skipped in notation: $c(x) = c(x,t)$, while $\dot{c}$ denotes partial derivative with respect to time.

$$\dot{c}(x) = \frac{1}{2} \int_0^x a(y, x - y)c(y)c(x - y)dy - \int_0^\infty a(y, x)c(y)c(x)dy \qquad (1)$$

where $a(x_1, x_2)$ is the so-called kernel which defines the rate of collisions (and coagulation) between particles of sizes x_1 and x_2 and $a(x_1, x_2) = a(x_2, x_1)$. The first term on the right-hand side is the production of particles of size x by coalescence of two smaller particles and the factor $1/2$ is for avoiding double counting. The second term represents the reduction in number of colliding particles due to coagulation.

The Smoluchowski's equation has an alternative form that is discrete in size space. Let x_0 be the smallest considered difference of size between particles, $x_i = ix_0$, $i \in \mathbb{N}$ and $c_i = c(x_i)$ then:

$$\dot{c}_i = \frac{1}{2} \sum_{k=1}^{i-1} a(x_k, x_{i-k})c_k c_{i-k} - \sum_{k=1}^\infty a(x_k, x_i)c_k c_i \qquad (2)$$

Analytic solutions to the equation are known only for simple kernels [14], such as: constant $a(x_1, x_2) = 1$, additive $a(x_1, x_2) = x_1 + x_2$ (Golovin's kernel [12]) or multiplicative $a(x_1, x_2) = x_1 x_2$. Taking atmospheric physics as an example, collisions of droplets within cloud occur by differentiated movements of particles caused by combination of gravitational, electrical, or aerodynamic forces, where gravitational effects dominate. As such, sophisticated kernels are needed to describe these phenomena, and hence numerical methods are required for solving the coagulation problem. However, when multiple properties of particles (volume, chemical composition, etc.) need to be taken into account, the numerical methods for SCE suffer from the curse of dimensionality due to the need to distinguish particles of same size x but different properties.

Additionally, it is worth to highlight that, in practice, the assumptions of the Smoluchowski equation may be difficult to meet. First, the particle size changes at the same time due to processes other than coalescence (e.g., condensation/evaporation). Second, it is assumed that the system is large enough and the
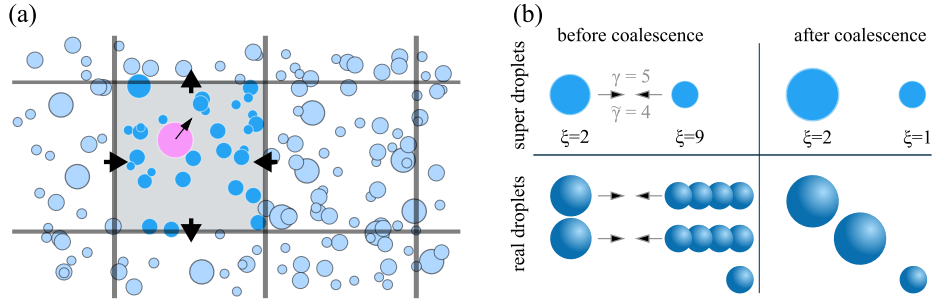
Fig. 1. (a): Schematic of how a CFD grid is populated with immersed super-particles; (b) schematic of super-particle collision (based on Fig.1 in [20]), the parameters γ and $\tilde{\gamma}$ are pertinent to representation of multiple collisions within a single coalescence event.

droplets inside are uniformly distributed, which in turn is only true for a small volume in the atmosphere. Moreover, using the Smoluchowski's equation that describes evolution of the mean state of the system, leads to deterministic simulations. The alternative to Smoluchowski equation are Monte-Carlo methods based on stochastic model of the collision-coalescence process. Stochastic approach enables simulating ensembles of realisations of the process aiding studies of rare (far-from-the-mean) phenomena like rapid precipitation onset (see [13]).

2.2   Monte-Carlo approach: Super-Droplet Method (SDM)

The stochastic coalescence model for cloud droplet growth was already analysed by Gillespie in [11]. There remained however still challenges (elaborated on below) related to computational and memory complexity which are addressed by SDM. The original formulation of SDM given in [20] was presented in the context of liquid-phase atmospheric clouds, and extended to cover clouds featuring ice phase of water in [21]).

Tracking all particles in large simulations has immeasurable cost, which is why the notion of super-particles is introduced. For convenience, let Ω be an indexed set of super-droplets for which collisions are considered (e.g., particles within one CFD grid cell, see Fig. 1a with the blue droplets in a shaded cell depicting set Ω). Each element of this set has unique index $i \in [0, 1, ..., n(\Omega) - 1]$, where $n(\Omega)$ is the number of considered super-droplets. Attributes related to a specific super-droplet i are denoted by $attr_{[i]}$. Note that only extensive attributes (volume, mass, number of moles) are considered within the coalescence scheme.

In SDM, each super-droplet represents a set of droplets with the same attributes (including position) and is assigned with a *multiplicity* denoted by $\xi \in \mathbb{N}$ and which can be different in each super-droplet. The multiplicity $\xi_{[i]}$ of super-droplet of size $x_{[i]}$ is conceptually related to $c_i = c(x_i)$ from equation (2). The number of super-droplets must be sufficient to cover the phase space of particle properties. The higher the number, the smaller the multiplicities and the higher fidelity of discretisation (see [20,21] for discussion).

Table 1 summarises the main differences between the SCE and SDM. The SDM algorithm steps can be summarised as follows.

Step 1: cell-wise shuffling

In SCE, all considered droplets (e.g., those within a grid cell in a given time step) are compared with each other and a fraction of droplets of size $x_{[i]}$ collides with a fraction droplets of size $x_{[j]}$ in each time step. In contrast, in SDM a random set of $\lfloor n(\Omega)/2 \rfloor$ non-overlapping pairs is considered in each time step.

To ensure linear computational complexity, the mechanism for choosing random non-overlapping pairs must also have at most linear computational complexity. It can be obtained by permuting an indexed array of super-droplets where each pair consist of super-droplets with indices $(2j)$ and $(2j + 1)$ where $j \in [0, 1, 2, \ldots, \lfloor n(\Omega)/2 \rfloor)$, see appendix A in [20].

Step 2: collision probability evaluation

If a fraction of droplets is allowed to collide, a collision of two super-droplets can produce third super-droplet which has different size. To represent such collision event, a new super-droplet would need to be added to the system what leads to: increasing the number of super-droplets in the system, smaller and smaller multiplicities and increasing memory demands during simulation. Moreover, it is hard to predict at the beginning of a simulation how much memory is needed (see discussion in [16]). Adding new super-droplets can be avoided by mapping colliding super-droplets to a multi-dimensional grid (e.g., 2D for particle volume and its dry size), simulating coalescence on this grid using SCE-based approach and discretising the results back into Lagrangian particles as proposed in [1]. Such approach however, not only entails additional computational cost, but most importantly is characterised by the very curse of dimensionality and numerical diffusion that were meant to be avoided by using particle-based technique.

SDM avoids the issue of unpredictable memory complexity stemming from the need of allocating space for freshly collided super-particle of size that differs from the size of the colliding ones. To this end, in SDM only collisions of all of $\min\{\xi_{[i]}, \xi_{[j]}\}$ droplets are considered, and thus collision of two super-droplets always produces one or two super-droplets (see Fig. 1b). This means there is no need for adding extra super-droplets to the system during simulation.

For each pair, probability of collision p is up-scaled by the number of real droplets represented by the super-droplet with higher multiplicity ($\max\{\xi_{[i]}, \xi_{[j]}\}$). As a result, computational complexity is $\mathcal{O}(n(\Omega))$ instead of $\mathcal{O}(n(\Omega)^2)$.

The evaluated probability of collision of super-droplets requires further up-scaling due to the reduced amount of sampled candidate pairs as outlined in Step 1 above. To this end, the probability is multiplied by the ratio of the number of all non-overlapping pairs $(n(\Omega)^2 - n(\Omega))/2$ to the number of considered candidate pairs $\lfloor n(\Omega)/2 \rfloor$.

For each selected pair of j-th and k-th super-droplets, the probability p of collision of the super-droplets within time interval Δt is thus evaluated as:

$$p = a(v_{[j]}, v_{[k]}) \max\{\xi_{[j]}, \xi_{[k]}\} \frac{(n(\Omega)^2 - n(\Omega))/2}{\lfloor n(\Omega)/2 \rfloor} \frac{\Delta t}{\Delta V} \tag{3}$$

Table 1. Comparison of SCE and SDM approaches to modelling collisional growth.

	SCE (mean-field)	SDM (probabilistic)
considered pairs	all (i,j) pairs	random set of $n(\Omega)/2$ non-overlapping pairs, probability up-scaled by $(n(\Omega)^2 - n(\Omega))/2$ to $n(\Omega)/2$ ratio
comp. complexity	$\mathcal{O}(n(\Omega)^2)$	$\mathcal{O}(n(\Omega))$
collisions	colliding a fraction of $\xi_{[i]}, \xi_{[j]}$	collide all of $\min\{\xi_{[i]}, \xi_{[j]}\}$ (all or nothing)
collisions triggered	every time step	by comparing probability with a random number

where a is a coalescence kernel and ΔV is the considered volume assumed to be mixed well enough to neglect the positions of particles when evaluating the probability of collision.

Step 3: collision triggering and attribute updates
In the spirit of Monte-Carlo methods, the collision events are triggered with a random number $\phi_\gamma \sim Uniform[0,1]$, and the rate of collisions (per timestep) $\gamma = \lceil p - \phi_\gamma \rceil$ with $\gamma \in \mathbb{N}$. Noting that the rate γ can be greater than 1, further adjustment is introduced to represent multiple collision of super-particles:

$$\tilde{\gamma} = \min\{\gamma, \lfloor \xi_{[j]}/\xi_{[k]} \rfloor\} \tag{4}$$

where $\xi_{[j]} \geq \xi_{[k]}$ was assumed (without losing generality). The conceptual view of collision of two super-droplets is intuitively depicted in Fig. 1b, presented example corresponds to the case of $\gamma = 5$ and $\tilde{\gamma} = \min\{5, \lfloor 9/2 \rfloor\} = 4$.

As pointed out in Step 2 above, SDM is formulated in a way assuring that each collision event produces one or two super-droplets. A collision event results in an update of super-droplet attributes denoted with $A \in \mathbb{R}^{n_{attr}}$ where n_{attr} is the number of considered extensive particle properties. During coalescence, particle positions remain unchanged, values of extensive attributes of collided droplets add up, while the multiplicities are either reduced or remain unchanged. This corresponds to the two considered scenarios defined in points (5) (a) and (b) in Sec. 5.1.3 in [20] and when expressed using the symbols used herein, with attribute values after collision denoted by hat, gives:

1. $\xi_{[j]} - \tilde{\gamma}\xi_{[k]} > 0$

$$
\begin{aligned}
\hat{\xi}_{[j]} = \xi_{[j]} - \tilde{\gamma}\xi_{[k]} \qquad &\hat{\xi}_{[k]} = \xi_{[k]} \\
\hat{A}_{[j]} = A_{[j]} \qquad &\hat{A}_{[k]} = A_{[k]} + \tilde{\gamma}A_{[j]}
\end{aligned} \tag{5}
$$

2. $\xi_{[j]} - \tilde{\gamma}\xi_{[k]} = 0$

$$
\begin{aligned}
\hat{\xi}_{[j]} = \lfloor \xi_{[k]}/2 \rfloor \qquad &\hat{\xi}_{[k]} = \xi_{[k]} - \lfloor \xi_{[k]}/2 \rfloor \\
\hat{A}_{[j]} = \hat{A}_{[k]} \qquad &\hat{A}_{[k]} = A_{[k]} + \tilde{\gamma}A_{[j]}
\end{aligned} \tag{6}
$$

Case 1 corresponds to the scenario depicted in Fig. 1b. In case 2, all droplets following a coalescence event have the same values of extensive attributes (e.g., volume), thus in principle could be represented with a single super droplet. However, in order not to reduce the number of super-droplets in the system, the resultant

super droplet is split in two. Since integer values are used to represent multiplicities, in the case of $\xi_{[k]} = 1$ in eq. (6), splitting is not possible and the j-th super-droplet is removed from the system.

3   Backend API

The proposed API is composed of four data structures (classes) and a set of library routines (computational kernels). Description below outlines both the general, implementation-independent, structure of the API, as well as selected aspects pertaining to the experience from implementing CPU and GPU backends in the PySDM package. These were built on top of the Numba [17] and ThrustRTC Python packages, respectively. Numba is an just-in-time (JIT) compiler for Python code, it features extensive support for Numpy and features multi-threading constructs akin to the OpenMP infrastructure. ThrustRTC uses the NVIDIA CUDA real-time compilation infrastructure offering high-level Python interface for execution of both built-in and custom computational kernels on GPU.

In multi-dimensional simulations coupled with a grid-based CFD fluid flow solver, the positions of droplets within the physical space are used to split the super-particle population among grid cells (CFD solver mesh). Since positions of droplets change based on the fluid flow and droplet mass/shape characteristics, the particle-cell mapping changes throughout the simulation. In PySDM, as it is common in cloud physics applications, collisions are considered only among particles belonging to the same grid cell, and the varying number of particles within a cell is needed to be tracked at each timestep to evaluate the super-particle collision rates. Figure 1a outlines the setting.

3.1   Data structures and simulation state

The `Storage` class is a base container which is intended to adapt the interfaces of the underlying implementation-dependent array containers (in PySDM: Numpy or ThrustRTC containers for CPU and GPU backends, respectively). This make the API independent of the underlying storage layer.

The `Storage` class has 3 attributes: data (in PySDM: an instance of Numpy `ndarray` or an instance of ThrustRTC `DVVector`), shape (which specifies size and dimension) and dtype (data type: `float`, `int` or `bool`). The proposed API employs one- and two-dimensional arrays, implementations of the `Storage` class feature an indirection layer handling the multi-dimensionality in case the underlying library supports one-dimensional indexing only (as in ThrustRTC). The two-dimensional arrays are used for representing multiple extensive attributes (with row-major memory layout). In general, structure-of-arrays layout is used within PySDM.

`Storage` handles memory allocation and optionally the host-device (CPU-accessible and GPU-accessible memory) data transfers. Equipping `Storage` with an override of the `[ ]` operator as done in PySDM can be helpful for debugging

```
1  idx = Index(N_SD, int)
2  multiplicities = IndexedStorage(idx, N_SD, int)
3  attributes = IndexedStorage(idx, (N_ATTR, N_SD), float)
4  volume_view = attributes[0:1, :]
5
6  cell_id = IndexedStorage(idx, N_SD, int)
7  cell_idx = Index(N_CELL, int)
8  cell_start = Storage(N_CELL + 1, int)
9
10 pair_prob = Storage(N_SD//2, float)
11 pair_flag = PairIndicator(N_SD, bool)
12
13 u01 = Storage(N_SD, float)
```

Fig. 2. Simulation state example with N_SD super-particles, N_CELL grid cells and N_ATTR attributes.

and unit testing (in Python, `Storage` instances may then be directly used with `max()`, `min()`, etc.), care needs to be taken to ensure memory-view semantics for non-debug usage, though. Explicit allocation is used only (once per simulation),

The `IndexedStorage` subclass of `Storage` is intended as container for super-particle attributes. In SDM, at each step of simulation a different order of particles needs to be considered. To avoid repeated permutations of the attribute values, the `Index` subclass of `Storage` is introduced. One instance of `Index` is shared between `IndexedStorage` instances and is referenced by the `idx` field.

The `Index` class features permutation and array-shrinking logic (allowing for removal of super-droplets from the system). To permute particle order, it is enough to shuffle `Index`. To support simulations in multiple physical dimensions, `Index` features sort-by-key logic where a cell id attribute is used as the key.

The `PairIndicator` class provides an abstraction layer facilitating pairwise operations inherent to several steps of the SDM workflow. In principle, it represents a Boolean flag per each super-particle indicating weather in the current state (affected by random shuffling and physical displacement of particles), a given particle is considered as first-in-a-pair. Updating `PairIndicator`, it must be ensured that the next particle according to a given `Index` is the second one in a pair – i.e., resides in the same grid cell. The `PairIndicator` is also used to handle odd and even counts of super-particles within a cell (see also Fig. 4). The rationale to store the pair-indicator flags, besides potential speedup, is to separate the cell segmentation logic pertinent to the internals of SDM implementation from potentially user-supplied kernel code.

Figure 2 lists a minimal set of instances of the data structures constituting an SDM-based simulation state.

3.2 Algorithm workflow and API routines

The algorithm workflow, coded in Python using the proposed data structures, and divided into the same algorithm steps as outlined in Sec. 2 is given in Fig. 3. An additional Step 0 is introduced to account for handling the removal of zero-multiplicity super-particles at the beginning of each timestep.

The `cell_id` attribute represents particle-cell mapping. Furthermore, the `cell_idx` helper `Index` instance can be used to specify the order in which grid

```
1  # step 0: removal of super-droplets with zero multiplicity
2  remove_if_equal_0(idx,                                    # in/out
3                    multiplicities)                          # in
4
5  # step 1: cell-wise shuffling, pair flagging
6  urand(u01)
7  shuffle_per_cell(cell_start,                             # out
8                   idx,                                     # in/out
9                   cell_idx, cell_id, u01)                  # in
10
11 flag_pairs(pair_flag,                                     # out
12            cell_id, cell_idx, cell_start)                 # in
13
14 # step 2: collision probability evaluation
15 coalescence_kernel(pair_prob,                            # out
16                    pair_flag, volume_view)                # in
17
18 times_max(pair_prob,                                     # in/out
19           multiplicities, pair_flag)                     # in
20
21 normalize(pair_prob,                                     # in/out
22           dt, dv, cell_id, cell_idx, cell_start)         # in
23
24 # step 3: collision triggering and attribute updates
25 urand(u01[:N_SD//2])
26 compute_gamma(pair_prob,                                 # in/out
27               u01[:N_SD//2])                             # in
28
29 update_attributes(multiplicities, attributes,           # in/out
30                   pair_prob)                             # in
```

Fig. 3. Algorithm workflow within a timestep, in/out comments mark argument intent.

cells are traversed – for parallel execution scheduling purposes. Both `cell_id` and `cell_idx` are updated before entering into the presented block of code.

Step 1 logic begins with generation of random numbers used for random permutation of the particles (in PySDM, the CURandRTC package is used). It serves as a mechanism for random selection of super-particle pairs in each grid cell. First, a shuffle-per-cell step is done in which the `cell_start` array is updated to indicate the location within cell-sorted `idx` where sets of super-particles belonging to a given grid cell start. Second, the `pair_flag` indicator is updated using the particle-cell mapping embodied in `cell_start`.

In PySDM, the shuffle-per-cell operation is implemented with two alternative strategies depending on the choice of the backend. A parallel sort with a random key is used on GPU, while the CPU backend uses a serial $\mathcal{O}(n)$ permutation algorithm (as in Appendix A in [20]). This choice was found to offer shortest respective execution times for the considered setup. In general, the number of available threads and the number of droplets and cells considered will determine optimal choice. Furthermore, solutions such as MergeShuffle [4] can be considered for enabling parallelism within the permutation step.

Instructions in Step 2 and Step 3 blocks correspond to the evaluation of subsequent terms of equation (3) and the collision event handling. To exemplify the way the GPU and CPU backends are engineered in PySDM, the implementation of the pair flag routine with ThrustRTC, and the update-attributes routine with Numba are given in Fig. 4 and Fig. 5, respectively.

```
1  _kernel = ThrustRTC.For(
2      ['perm_cell_start', 'perm_cell_id', 'pair_flag', 'length'], "i", '''
3      pair_flag[i] = (
4          i < length - 1 &&
5          perm_cell_id[i] == perm_cell_id[i+1] &&
6          (i - perm_cell_start[perm_cell_id[i]]) % 2 == 0
7      );
8      ''')
9
10  def flag_pairs(pair_flag, cell_start, cell_id, cell_idx):
11      perm_cell_id = ThrustRTC.DVPermutation(cell_id.data, cell_id.idx.data)
12      perm_cell_start = ThrustRTC.DVPermutation(cell_start.data, cell_idx.data)
13      d_length = ThrustRTC.DVInt64(len(cell_id))
14      _kernel.launch_n(len(cell_id),
15          [perm_cell_start, perm_cell_id, cell_idx.data,
16           pair_flag.indicator.data, d_length])
```

Fig. 4. GPU backend implementation of the pair-flagging routine.

```
1  @numba.njit(parallel=True, error_model='numpy')
2  def _update_attributes(length, n, attributes, idx, gamma):
3      for i in prange(length//2):
4          j = idx[2*i]
5          k = idx[2*i + 1]
6          if n[j] < n[k]:
7              j, k = k, j
8          g = min(int(gamma[i]), int(n[j] / n[k]))
9          if g == 0:
10             continue
11         new_n = n[j] - g * n[k]
12         if new_n > 0:
13             n[j] = new_n
14             for attr in range(0, len(attributes)):
15                 attributes[attr, k] += g * attributes[attr, j]
16         else:  # new_n == 0
17             n[j] = n[k] // 2
18             n[k] = n[k] - n[j]
19             for attr in range(0, len(attributes)):
20                 attributes[attr, j] = attributes[attr, j] * g \
21                                     + attributes[attr, k]
22                 attributes[attr, k] = attributes[attr, j]
23
24  def update_attributes(n, intensive, attributes, gamma):
25      _update_attributes(len(n.idx),
26          n.data, intensive.data, attributes.data,      # in/out
27          n.idx.data, gamma.data)                       # in
```

Fig. 5. CPU backend implementation of the attribute-update routine.a

4   Example simulations

This section outlines a set of test cases useful in validating implementation of the proposed API (and included in the set of examples shipped with PySDM). First, a simulation constructed analogously as that reported in Fig. 2 in the original SDM paper [20] is proposed. The considered volume of $V = 10^6$ m^3 is populated with $N_0 \cdot V = 2^{23} \cdot 10^6$ particles. Sizes of the particles follow an exponential particle volume distribution $p(v_{[i]}) = \overline{v}^{-1} \exp(-v_{[i]}/\overline{v})$ where $\overline{v} = (30.531\mu\mathrm{m})^3 4\pi/3$. The simulation uses 2^{17} super-droplets initialised with equal multiplicities so that at $t = 0$ s each one represents a quantile of the total number of modeled particles (within an arbitrarily chosen percentile range from 0.001% to 99.999%). The timestep is 1 s. The Golovin [12] additive $a(v_1, v_2) = b(v_1 + v_2)$ coalescence kernel with $b = 1500\mathrm{s}^{-1}$ is used for which an analytical solution to
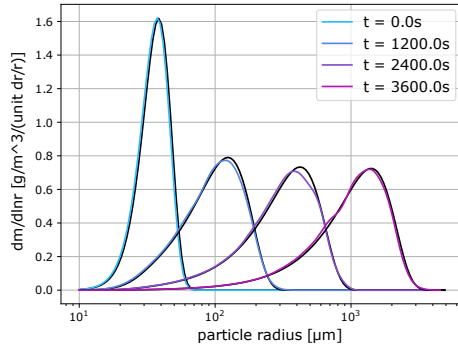
Fig. 6. Particle mass spectrum: SDM results (colour) vs. analytic solution (black).
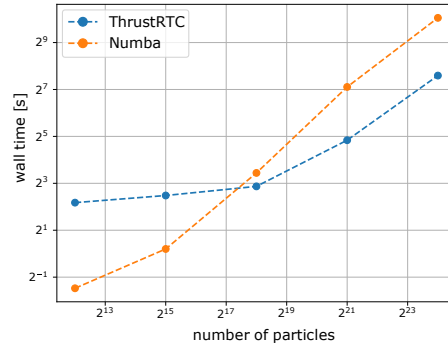
Fig. 7. Simulation wall time as a function of the number of super particles used.

the Smoluchowski's equation is known (cf. equation 36 in [12]):

$$\phi(v,t) = \frac{1 - \tau(t)}{v\sqrt{\tau(t)}} \mathrm{I}^1_{\mathrm{ve}}\left(\frac{2v\sqrt{\tau(t)}}{\overline{v}}\right) \exp\left(\frac{-v(1 + \tau(t) - 2\sqrt{\tau(t)})}{\overline{v}}\right) \qquad (7)$$

where $\tau = 1 - \exp(-N_0 b\overline{v}t)$ and $\mathrm{I}^1_{\mathrm{ve}}$ stands for exponentially scaled modified Bessel function of the first kind (`scipy.special.ive`).

Plot 6 shows the relationship between the mass of droplets per unit of ln$r$ and the droplet radius r (assuming particle density of $1000\,\mathrm{kg/m^3}$ as for liquid water). Results obtained with PySDM are plotted by aggregating super-particle multiplicities onto a grid of ca. 100 bins, and smoothing the result twice with a running average with a centered window spanning five bins.

Figure 7 documents simulation wall times for the above test case measured as a function of number of super-particles employed. Wall times for CPU (Numba) and GPU (ThrustRTC) backends are compared depicting a five-fold GPU-to-CPU speedup for large state vectors (tested on commodity hardware: Intel Core i7-10750H CPU and NVIDIA GeForce RTX 2070 Super Max-Q GPU).

Figure 8 presents results visualised in analogous plots but from simulations with more physically-relevant coalescence kernels. Simulation setup follows the work of [6] and features so-called gravitational kernel involving a parametric form of particle terminal velocity dependence on its size, and a parametric kernel modeling the the electric field effects on collision probability. Since the analytic solution is not known in such cases, the results are juxtaposed with figures reproduced from [6].

5   Summary and discussion

This paper has discussed a set of data structures and computational kernels constituting a number-crunching backend API for the Super-Droplet Method Monte-Carlo algorithm for representing collisional growth of particles. Employment of the API assures separation of concerns, in particular separation of parallelisation logic embedded in the backend-level routines from domain logic per-
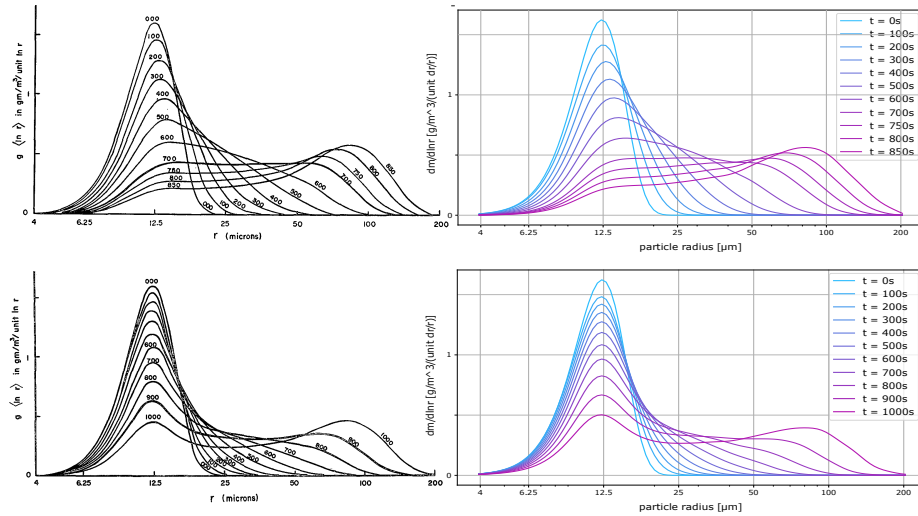
Fig. 8. Left panels show Figs. 5 and 8 from [6] (copyright: American Meteorological Society, used with permission). Right panels depict solutions obtained with PySDM. Top: gravitational kernel; bottom: kernel modelling electric field effect.

tinent to the algorithm workflow. This improves code readability, paves the way for modular design and testability, which all contribute to code maintainability.

The presented SDM algorithm and API descriptions discern data dependencies across the steps of the algorithm (including in/out parameter "intent") and highlight parallelisation opportunities in different steps. Most of the discerned steps of the SDM algorithm are characterised by some degree of freedom in terms of their implementation. Embracing the API shall help in introducing a dependency injection mechanism allowing unit testing of selected steps and profiling performance with different variants of implementation.

The design of the API has been proved, in the sense of reused abstraction principle, within the PySDM project [5] where two backends sharing the API offer contrasting implementations for CPU and GPU computations. Both backends are implemented in Python, however: (i) they are targeting different hardware (CPU vs. GPU), (ii) they are based on different underlying technology (Numba: LLVM-based JIT compiler, and ThrustRTC: NVRTC-based runtime compilation mechanism for CUDA), and (iii) they even do not share the de-facto-standard Python Numpy arrays as the storage layer. This highlights that the introduced API is not bound to particular implementation choices, and in principle its design is applicable to other languages than Python.

It is worth pointing out that, in the super-particle CFD-coupled simulations context SDM was introduced and gained attention in, the scheme is particularly well suited for leveraging modern hybrid CPU-GPU hardware. First, the algorithm is (almost) embarrassingly parallel. Second, the CPU-GPU transfer overhead is not a bottleneck when GPU-resident dispersed phase representation (super-particles) is coupled with CPU-computed CFD for the continuous phase (fluid flow) as only statistical moments of the size spectrum of the particles

are needed for the CFD coupling. Third, the CPU and GPU resources on hybrid hardware can be leveraged effectively if fluid advection (CPU) and particle collisions (GPU) are computed simultaneously (see [10]).

Overall, the discussed API prioritises simplicity and was intentionally presented in a paradigm-agnostic pseudo-code-mimicking way, leaving such aspects as object orientation up to the implementation. Moreover, while the presented API includes data structures and algorithm steps pertinent to multi-dimensional CFD grid coupling, presented examples featured zero-dimensional setups, for brevity. In PySDM [5], the backend API is extended to handle general form of coalescence kernels, representation of particle displacement, condensational growth of particles, aqueous chemical reactions, and the examples shipped with the package include simulations in multiple physical dimensions.

Acknowledgements

References

1. Andrejczuk, M., Grabowski, W.W., Reisner, J., Gadian, A.: Cloud-aerosol interactions for boundary layer stratocumulus in the Lagrangian Cloud Model. J. Geophys. Res. Atmos. **115** (2010), https://doi.org/10.1029/2010JD014248
2. Arabas, S., Jaruga, A., Pawlowska, H., Grabowski, W.W.: libcloudph++ 1.0: a single-moment bulk, double-moment bulk, and particle-based warm-rain microphysics library in C++. Geosci. Model Dev. **8** (2015), https://doi.org/10.5194/gmd-8-1677-2015
3. Arabas, S., Shima, S.: Large-eddy simulations of trade wind cumuli using particle-based microphysics with Monte Carlo coalescence. J. Atmos. Sci. **70** (2013), https://doi.org/10.1175/JAS-D-12-0295.1
4. Bacher, A., Bodini, O., Hollender, A., Lumbroso, J.: Mergeshuffle: A very fast, parallel random permutation algorithm (2005), https://arxiv.org/abs/1508.03167
5. Bartman, P., Arabas, S., Górski, K., Jaruga, A., Łazarski, G., Olesik, M., Piasecki, B., Talar, A.: PySDM v1: particle-based cloud modelling package for warm-rain microphysics and aqueous chemistry (2021), https://arxiv.org/abs/2103.17238
6. Berry, E.: Cloud droplet growth by collection. J. Atmos. Sci. **24** (1966), https://doi.org/10.1175/1520-0469(1967)024%3C0688:CDGBC%3E2.0.CO;2
7. Chandrasekhar, S.: Stochastic problems in physics and astronomy. Rev. Mod. Phys. **15** (1943), https://doi.org/10.1103/RevModPhys.15.1
8. DeVille, R.E., Riemer, N., West, M.: Weighted Flow Algorithms (WFA) for stochastic particle coagulation. J. Comput. Phys. **230** (2011), https://doi.org/10.1016/j.jcp.2011.07.027
9. Dziekan, P., Pawlowska, H.: Stochastic coalescence in Lagrangian cloud microphysics. Atmos. Chem. Phys. **17** (2017), https://doi.org/10.5194/acp-17-13509-2017
10. Dziekan, P., Waruszewski, M., Pawlowska, H.: University of Warsaw Lagrangian Cloud Model (UWLCM) 1.0: a modern large-eddy simulation tool for warm cloud modeling with Lagrangian microphysics. Geosc. Model Dev. **12** (2019), https://doi.org/10.5194/gmd-12-2587-2019

11. Gillespie, D.: The stochastic coalescence model for cloud droplet growth. J. Atmos. Sci. **29** (1972), https://doi.org/10.1175/1520-0469(1972)029%3C1496:TSCMFC%3E2.0.CO;2

12. Golovin, A.: The solution of the coagulation equation for raindrops. Taking condensation into account. Bull. Acad. Sci. SSSR Geophys. Ser. **148** (1963), http://mi.mathnet.ru/dan27630, in Russian

13. Grabowski, W., Morrison, H., Shima, S., Abade, G., Dziekan, P., Pawlowska, H.: Modeling of cloud microphysics: Can we do better? Bull. Am. Meteorol. Soc. **100** (2019), https://doi.org/10.1175/BAMS-D-18-0005.1

14. Hansen, K.: Abundance distributions; large scale features. In: Statistical Physics of Nanoparticles in the Gas Phase, vol. 73. Springer, Cham (2018), http://doi.org/10.1007/978-3-319-90062-9_8

15. Hoffmann, F., Noh, Y., Raasch, S.: The route to raindrop formation in a shallow cumulus cloud simulated by a Lagrangian cloud model. J. Atmos. Sci. **74** (2017), https://doi.org/10.1175/JAS-D-16-0220.1

16. Jensen, J.B., Lee, S.: Giant sea-salt aerosols and warm rain formation in marine stratocumulus. J. Atmos. Sci. **65** (2008), https://doi.org/10.1175/2008JAS2617.1

17. Lam, S., Pitrou, A., Seibert, S.: Numba: a LLVM-based Python JIT compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. LLVM '15, ACM (2015), https://doi.org/10.1145/2833157.2833162

18. Li, X.Y., Brandenburg, A., Haugen, N.E., Svensson, G.: Eulerian and Lagrangian approaches to multidimensional condensation and collection. J. Adv. Model. Earth Sy. **9** (2017), https://doi.org/10.1002/2017MS000930

19. Sato, Y., Shima, S., Tomita, H.: Numerical convergence of shallow convection cloud field simulations: Comparison between double-moment Eulerian and particle-based Lagrangian microphysics coupled to the same dynamical core. J. Adv. Model. Earth Syst. **10** (2018), https://doi.org/10.1029/2018MS001285

20. Shima, S., Kusano, K., Kawano, A., Sugiyama, T., Kawahara, S.: The super-droplet method for the numerical simulation of clouds and precipitation: a particle-based and probabilistic microphysics model coupled with a non-hydrostatic model. Q. J. Royal Meteorol. Soc. **135** (2009), https://doi.org/10.1002/qj.441

21. Shima, S., Sato, Y., Hashimoto, A., Misumi, R.: Predicting the morphology of ice particles in deep convection using the super-droplet method: development and evaluation of SCALE-SDM 0.2.5-2.2.0, -2.2.1, and -2.2.2. Geosci. Model Dev. **13** (2020), https://doi.org/10.5194/gmd-13-4107-2020

22. Shima, S., Sugiyama, T., Kusano, K., Kawano, A., Hirose, S.: Simulation method, simulation program, and simulator (2007), https://data.epo.org/gpi/EP1847939A3

23. Smoluchowski, M.: Drei Vorträge über Diffusion, Brownsche Molekularbewegung und Koagulation von Kolloidteilchen I. Phys. Z. **22** (1916), https://jbc.bj.uj.edu.pl/dlibra/publication/411755/edition/387533/content

24. Smoluchowski, M.: Drei Vorträge über Diffusion, Brownsche Molekularbewegung und Koagulation von Kolloidteilchen II. Phys. Z. **23** (1916), https://jbc.bj.uj.edu.pl/dlibra/publication/411756/edition/387534/content

25. Unterstrasser, S., Hoffmann, F., Lerch, M.: Collisional growth in a particle-based cloud microphysical model: insights from column model simulations using LCM1D (v1.0). Geosci. Model Devel. **13** (2020), https://doi.org/10.5194/gmd-13-5119-2020