Master's thesis

Computational Science and Engineering
(Int. Master's Program)

Improved implementation of a Lagrangian microphysics module for the geophysical flow solver EULAG

Author: Benedikt Stegmaier

Submission date: Wednesday 2nd October, 2013

1st examiner: Prof. Dr. Robert Sausen (DLR)
2nd examiner: Prof. Dr. Hans-Joachim Bungartz (TUM)
Supervisor: Dr. Simon Unterstraßer (DLR)

Technische Universität München / Deutsches Zentrum für Luft- und Raumfahrt

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Wednesday 2nd October, 2013

Benedikt Stegmaier

Abstract

The model system EULAG-LCM allows cloud resolving simulations of pure ice crystal clouds. It is used to study dynamic and micro-physical processes in contrails and natural cirrus clouds.
The base model EULAG solves the dynamic equations, while the micro-physical module LCM treats the ice physics. LCM uses a Lagrangian approach, where large numbers of particles are tracked.
Past simulations have shown, that the original implementation has room for improvements in terms of storage of those particles. It uses a static memory scheme, which is not well suited for cirrus clouds, where particle concentrations are highly variable.
Within this thesis a new storage scheme based on singly linked lists is developed and implemented. The new scheme is more flexible and memory efficient. In a typical real world scenario, the required memory of the overall simulation could be reduced by a factor of three. In accordance to current trend in HPC architectures, this scheme prepares EULAG-LCM for the future generations of supercomputers with less memory per CPU core.

Acknowledgements

First and foremost I would like to thank my supervisor Simon who spared no effort and time to discuss all my questions and share his knowledge.
I am grateful to Prof. Bungartz for generously accepting to be an examiner. Special thanks to Prof. Sausen for making my thesis at Deutsches Zentrum für Luft- und Raumfahrt (DLR) possible and for examining this work.
Hendryk Bockelmann (DKRZ) made a great effort to help me find bugs and performance issues and answered my questions regarding the cluster.
I appreciate the help of Ingo Sölch with refactoring the code.
Thanks to all people within the department of Dynamics of the atmosphere. I enjoyed working in such a pleasant atmosphere.
A big thank you to my office co-workers Basti, Meli, Henrike und Vanessa and all other Ph.D./M.Sc./B.Sc. candidates who sweetened my summer through cake or afterwork excursions to nearby lakes.
I am particular grateful for the support by my family, Silvi and my friends. You carry me not only during this thesis but through all phases of life.

List of symbols and abbreviations

SIP virtual simulation particle, representing multiple ice crystals
DKRZ Deutsches Klimarechenzentrum
DLR Deutsches Zentrum für Luft- und Raumfahrt
MPI Message Passing Interface
HPC High performance computing

Contents

1 Introduction

The model system EULAG-LCM allows cloud resolving simulations of pure ice crystal clouds. It is used to study dynamic and micro-physical processes in contrails and natural cirrus clouds.
The base model EULAG solves the dynamic equations, while the micro-physical module LCM treats ice physics. LCM uses a Lagrangian approach, where large numbers of particles are tracked.
Past simulations have shown, that the current implementation has room for improvements in terms of storage of those particles.
Within this thesis a new more memory efficient and flexible storage scheme is developed and implemented.

2 EULAG

2.1 EULAG

EULAG is a numerical solver for geophysical flows. EULAG's popularity in geophysics and meteorology is founded in the wide range of physical scales over which it is applicable (Grabowski and Smolarkiewicz, 2002). The anelastic approximation of the Navier-Stokes equations are solved on a discrete, unstaggered grid in either Eulerian or Lagrangian form using the MPDATA advection algorithm (Smolarkiewicz and Margolin, 1998).

2.2 LCM

LCM is a Lagrangian micro-physics module for EULAG, developed and maintained at DLR (Sölch and Kärcher, 2010, 2011).
It is capable of cloud resolving simulations of pure ice crystal clouds and aims at the study of formation, development and persistence of natural cirrus clouds and contrails.
In the upper troposphere, where cirrus clouds form, conditions are such that the ice crystal concentration with $1\,\mathrm{cm}^{-3}$ is small compared to the concentration of air molecules $10^{19}\,\mathrm{cm}^{-3}$ to $10^{20}\,\mathrm{cm}^{-3}$. This dispersive dual-phase flow can be separated into air as continuous fluid phase and ice crystals as discrete particles, moving within the air. Sölch (2009, p. 9) explains and legitimizes this approach.
A mixed Eulerian-Lagrangian method is used: An Eulerian approach for the dynamics of the fluid flow and Lagrangian particle tracking for ice crystals.
Tracking every single one of those ice crystals is difficult on current computers, as there is not enough capacity in terms of memory and CPU available. Instead, a multitude of ice crystals with similar properties (size, location, ...) are represented by a virtual particle - called SIP - and are handled together.

2.2.1 Physical processes

This section is a short overview and explains some important keywords, needed during the subsequent chapters. For a complete and in-depth discussion of all pro-

cesses, implemented in LCM, please refer to Sölch (2009).

Homogeneous freezing, heterogeneous nucleation Ice crystals may form in several different ways. The most relevant origination processes at environmental conditions of $< 235\,\mathrm{K}$ are implemented in EULAG-LCM:

- **Homogeneous nucleation**: Freezing of fluid aerosol droplets, consisting of $H_2SO_4 + H_2O$ or $H_2SO_4 + H_2O + HNO_3$.
- Heterogeneous nucleation
  - **Immersion freezing**: Freezing of liquid aerosol droplets, which additionally contain a solid insoluble core.
  - **Deposition freezing**: Water vapour freezes on the surface of solid water-insoluble particles.

Deposition and Sublimation Individual water molecules of the gaseous phase attach to the surface of existing ice crystals. At the same time, water molecules from the ice crystal change into the gaseous phase.
At an ambient humidity of 100% both processes are in a dynamic equilibrium and the size of ice crystals remains constant. For small particles the Kelvin effect has to be considered. In supersaturated air a net grow of ice crystals can be observed, we call this deposition. A net reduction in subsaturated air is called sublimation.
Energy is released or absorbed in form of latent heat.

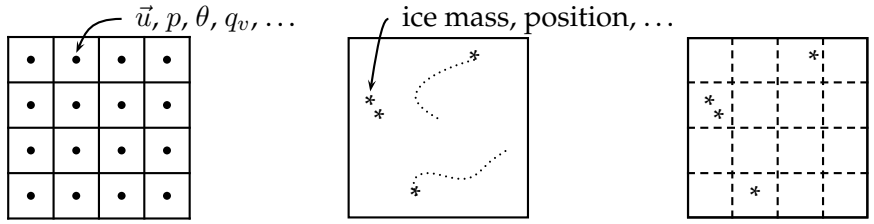
Sedimentation Ice crystals experience a downward force within the Earth's gravitational field. A thereby induced downward movement is called sedimentation. The sedimentation velocity of a single particle is influenced by its size and mass. Thus a separation of particles occurs, with heavier and bigger particles towards the bottom of an ice crystal cloud. This vertical redistribution of ice has for example an influence on water vapour content or the radiation budget.

Aggregation A process, during which ice crystals collide and stick together, forming a bigger structure, is called aggregation. Those collision are due to differential sedimentation. This means that particles with different mass and size have different vertical speeds and collect slower particles on their way down (Sölch, 2009, p. 40). The exact mechanism by which those crystals stick together is still under debate. The relevance of this process lies in a shift from smaller ice crystals to larger particles.

2.2.2 Coupling to EULAG

EULAG solves the dynamics of the air phase or another fluid using a finite differences approach on a structured grid. Mesh adaption is included in EULAG via time-dependent horizontal coordinate transformations (Prusa and Smolarkiewicz, 2003). In conjunction with LCM, EULAG is only run in simple Cartesian coordinates. Thus, in each grid point we know dynamic variables such as velocity

$(u, v, w)^\intercal$, pressure p and potential temperature θ. Additionally prognostic equations (including source terms and advective terms) for the mixing ratio of water vapour, trace gases and aerosol respectively are solved on these points.



(a) EULAG solves the dynamics of the air phase on a discrete, structured grid. Grid boxes are centered around their respective grid point.

(b) LCM: SIPs are discrete particles at arbitrary locations within the subdomain.

(c) Only for coupling of EULAG and LCM, particle positions have to be mapped to grid boxes. Grid point values are representative for the entire grid box.

Figure 1: A subdomain, as seen by EULAG and LCM.

Forward: In order to determine the movement of SIPs or interactions with the water vapour field, we need to know which grid box a SIP currently is in.
All SIPs with $|\vec{x}_{\mathrm{SIP}} - \vec{x}_{\mathrm{GP}}| < \frac{1}{2}(\mathrm{d}x, \mathrm{d}y, \mathrm{d}z)^\intercal$ i.e. less than half a grid width away from a grid point (i, j, k) are associated to the corresponding box.
The grid point values are assumed to be representative for the entire corresponding grid box. There is no interpolation towards particle positions within this box (Sölch, 2009, p. 27).

Back: Several LCM processes can influence the dynamics of the gaseous phase. Most relevant are the consumption and release of latent heat or interactions with the water vapour field, which leads to density variations. The net values are calculated by the LCM module, handed back to EULAG and are fed into the dynamic equations by special forcing terms.

2.2.3 Operator splitting & subcycling

Although all micro-physical processes happen simultaneously, in LCM each of them is treated and solved sequentially (Sölch, 2009, p. 12). The dynamic time step is usually subdivided into smaller time steps for the micro-physics module and may be divided even further for certain physical processes. This has been chosen since some processes respond very sensitive to tiny changes in water vapour concentration or temperature. For example the rate of nucleation has a strong non-linear dependence on the excess humidity. Thus a sufficiently small time step is required to accurately calculate the amount of ice crystals formed (Unterstrasser and Sölch, 2013).

2.3 Code quality and development

EULAG has a long history, its roots date back to the early 80^{ies} (Smolarkiewicz, 1983). The associated LCM module was developed during the PhD thesis of Sölch (2009). And applied in several cirrus and contrails studies (Sölch and Kärcher, 2011; Lainer, 2012).
This historically grown code base consists of one single file with $\sim 86,000$ lines of code. It is a mixture of *C shell* (*csh*) scripts and *FORTRAN 77* with enclosed *C preprocessor macros*.
There are no configuration files. Instead simulation scenarios have to be hard-coded at various places of the code. This sometimes leads to forgetting scenario-specific statements.
Development is done without tests, version control system or a consistent programming style in terms of indenting or naming.
The code itself contains duplicate code and functions with only minor differences in functionality and historic but now unnecessary dead code. There is no hierarchy or structure as *FORTRAN 77* did not support modules yet. Global methods operate on global variables, which makes estimating side-effects difficult. There is heavy use of once popular but nowadays frowned upon *FORTRAN* features like implicit type declarations.

For and during this thesis a major refactoring was performed: The *EULAG* core with its *csh* and *FORTRAN* code is now in its own file. All LCM related *FORTRAN* methods ($25,000$ lines of code) are grouped by purpose and distributed among 19 files. SIP and storage related code adds another 11 files.
All new storage related code is encapsulated in modules. It is extensively tested with unit tests; see section 5.1.
Additionally to the above mentioned restructuring, a lot of smaller refactorings such as correcting indenting, renaming variables and methods have been started.
To facilitate the development process, *git* is now used as version control system. This also has the advantage of different simulation scenarios, like benchmarks and tests, being easy to maintain.

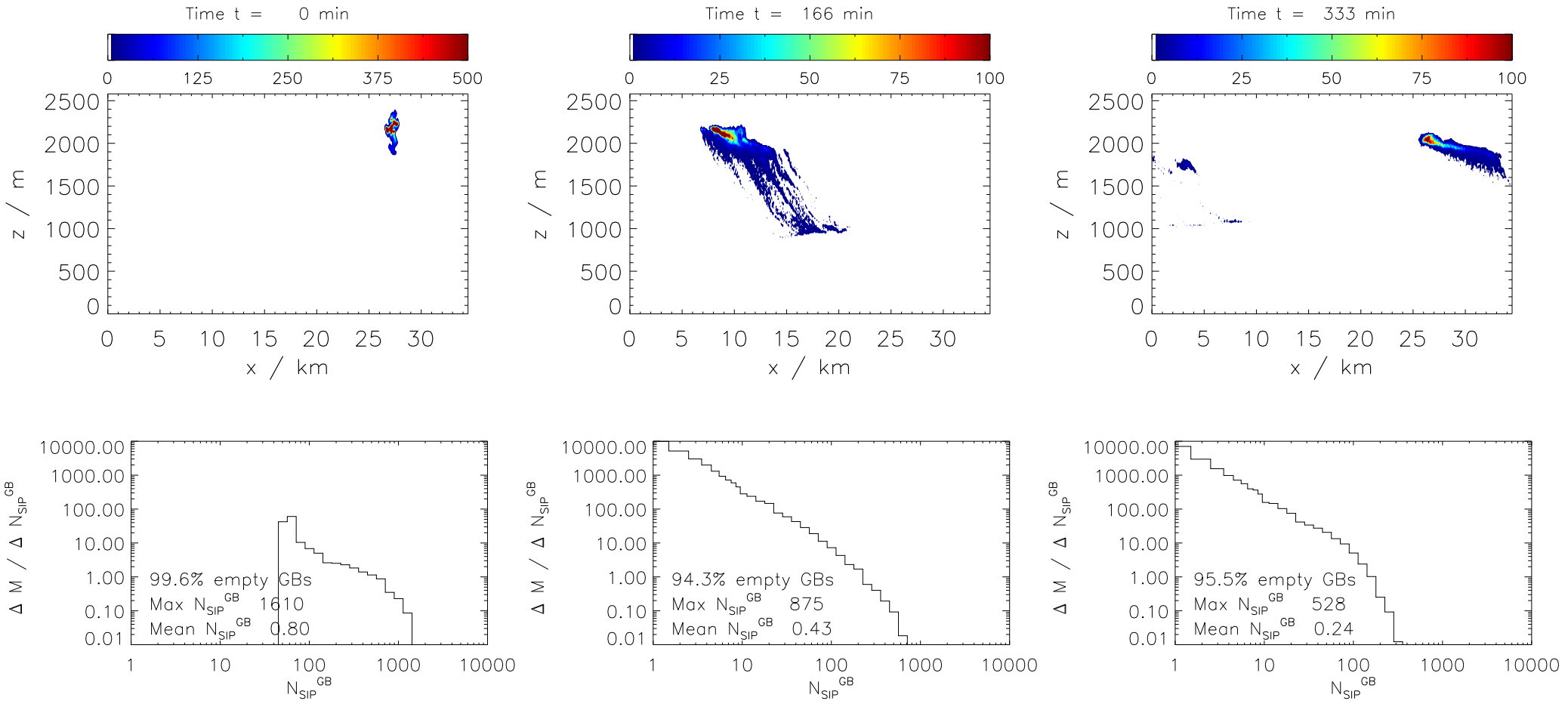


Figure 2: A simulated contrail cirrus cloud after 0 min, 166 min and 333 min. The upper panel shows the number of SIPs per grid box $N_{\mathrm{SIP}}^{\mathrm{GB}}$ over a 2D cross-section. Colorbars do not extend to the maximum $N_{\mathrm{SIP}}^{\mathrm{GB}}$ value. The frequency of occurence of a certain $N_{\mathrm{SIP}}^{\mathrm{GB}}$ value can be read from the histograms in the lower panel. Only non-empty grid boxes are taken into account, as $N_{\mathrm{SIP}}^{\mathrm{GB}}$ is zero for over 94% of all grid boxes.

3 Storage scheme

For each SIP, several attributes have to be remembered. The most important ones are:

- Number of ice crystals per SIP, ice crystal radius and shape
- Location within the subdomain and associated grid box
- Various flags, such as formation process (natural cirrus vs. contrail)

Overall it is 16 values per SIP, of which four are integers and 12 are floating points numbers. This adds up to 112 bytes.

3.1 Requirements

Requirements for storing SIPs are:

Low memory overhead: Current trends in microprocessor design and supercomputer hardware tend towards an increasing number of processor cores, whilst the amount of memory grows at a much slower pace. Thus the amount of memory available per core is decreasing.
At Deutsches Klimarechenzentrum (DKRZ) there are nodes with 750 MB and 1.5 GB of memory per core[1]. Experience has shown, that this is not enough for complex simulation scenarios.

Highly dynamic storage: A typical simulation scenario is to examine the temporal evolution of the 2D cross-section of a single contrail. Subdomain decomposition in this case is only along the horizontal direction. Figure 2 show exemplary plots for such a scenario at different points in time. A vertical cross-section with the number of SIPs per grid box is shown in the upper panel and the corresponding histogram of $C_{\mathrm{SIP}}^{\mathrm{GB}}$values is displayed in the lower panel.
These plots are quite revealing in several ways. First, a significant number of grid boxes is completely empty. This is since clouds are usually confined to a small region in space. In the plotted scenario only less than 6% of all grid boxes are within a cloud and contain SIPs. This property has to be reflected by the storage scheme: An optimal scheme requires no memory for empty grid boxes.
Second, among those grid boxes occupied, the number of SIPs per grid box stretches over several orders of magnitude. There are a lot of grid boxes with only one SIP while others have up to 1600.

Low complexity of operations: How fast data can be retrieved and saved to storage has a major impact on the overall runtime of a simulation. Often used operations regarding SIP storage are:

- Read and write access to SIPs properties

[1] Numbers are in terms of figures. As multiple cores share a certain amount of memory, imbalances might compensate.

- Insertion of SIPs
- Deletion of SIPs
- Counting the number of SIPs per grid box or subdomain
- Sorting according to a SIP property per grid box or subdomain

Read access is usually within an iteration over all SIPs belonging to a specific grid box. Within this grid box the exact order in which SIPs are processed is insignificant for microphysical processes. Random access to a specific SIP is never necessary. Read and write access are necessary in every part of the timestep routine (see Listing 1) and have a major impact on the overall performance.

```
subroutine LCM_timestep ! called by EULAG's timestep loop
    dissolution aerosol ! influences nucleation
    do subcycling with smaller timestep
        nucleation                                  r/w  insert
        dissolution aerosol
        deposition & sublimation                    r/w         delete
    end
    aggregation                                sort r/w         delete
    advection                                       r/w
    transfer SIPs between grid boxes                r/w  insert delete
    transfer SIPs between subdomains                r/w  insert delete
    delete sedimented SIPs outside domain           r/w         delete
    add up forcing from latent heat
    split / merge SIPs                              r/w  insert delete
    diagnose routines                               r/w
end subroutine
```

Listing 1: General structure of microphysics portion of a timestep. Operations on the SIP storage structure are noted on the right hand side.

Another often needed operation is to determine the number of SIPs per grid box or per subdomain.
Insertion and deletion might be performed numerous times during one timestep, as it is shown in Listing 1. Thus, the overall performance would benefit from a low complexity of those operations.
Sorting SIPs within a grid box, on the other hand, is only performed once per timestep, if aggregation is enabled. Additionally every 200^{th} to 500^{th} time step the output routine requires all SIPs within a subdomain to be sorted. Based on the rare usage of this operation, it plays only a minor role.

3.2 Old storage scheme

Before we search for improved ways of storing SIPs, we first have to analyze the previous storage scheme.

Previous versions of EULAG-LCM were entirely written in *FORTRAN 77*. However, dynamic memory allocation was not introduced until *FORTRAN 90*. As the number of SIPs is highly dynamic, this limitation required some detours: static arrays with a certain capacity are kept ready. Their dimensions have to be already specified before compilation.
The following description deals with storage of SIPs in one subdomain. It is replicated by each process.
Figure 3 contains a sketch, which visualizes the arrays and relationships of this storage scheme.

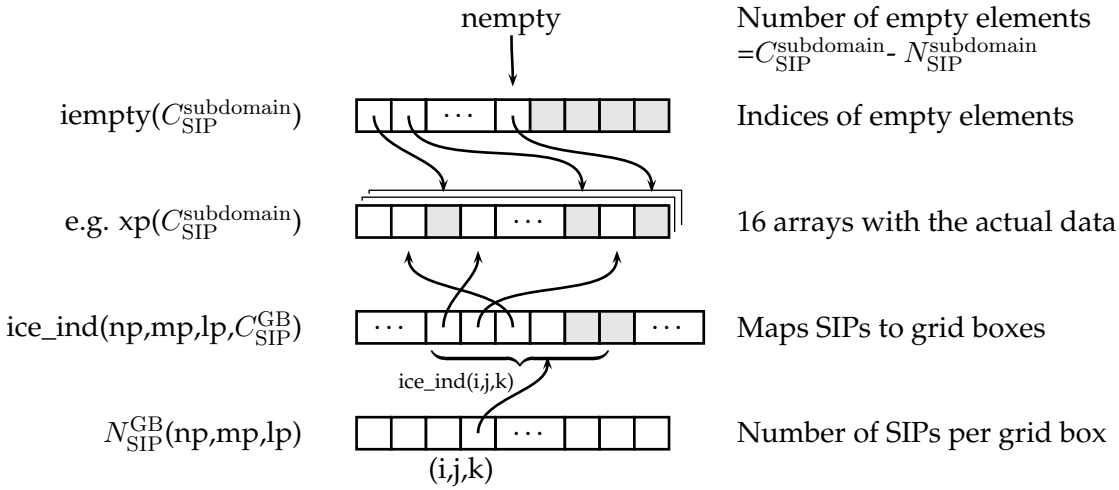


Figure 3: SIP data is stored in 16 static arrays of limited capacity. Additional arrays are necessary to keep record of empty entries (gray) or the association of SIPs to grid boxes.

There are 16 data arrays. One for each SIP parameter. Before compilation we have to make an educated guess about how many SIPs there will be in maximum per subdomain: $C_{\mathrm{SIP}}^{\mathrm{subdomain}}$. This value is used to statically set the length of all data arrays.
As the number of SIPs has to be smaller or equal to this maximum capacity $N_{\mathrm{SIP}}^{\mathrm{subdomain}} \leq C_{\mathrm{SIP}}^{\mathrm{subdomain}}$, there are unoccupied entries. Those are not necessarily all at the end of the data arrays. Due to deletion of single SIPs, they may be distributed along the entire index range. Common measures to prevent holes - e.g. copying the last entry to an empty slot - would have to be applied to all 16 data arrays and thus are avoided.
To keep track of what indices are empty, there is another array: iempty. Within $1 \ldots nempty$ it contains indices of entries within the data arrays, which are empty. To ensure, that all entries in iempty are within 1...nempty, inserting a SIP is always done at iempty(nempty) while deletions add an index at iempty(nempty+1).
Since potentially all entries are emtpy and thus should be listed in iempty, the length of iempty has to be $C_{\mathrm{SIP}}^{\mathrm{subdomain}}$as well.

In addition to saving SIP attributes, it is also important to know, which grid box a SIP is associated with.
It is not practical performance-wise to save the grid box as a SIP property into yet another data array, as it is often necessary to iterate over all SIPs within one grid box.
The existing approach is, to use a 4 dimensional array where 3 dimensions select a grid box and the 4th dimension is an array of data array indices.
This indices specify SIPs, which belong to the respective grid box.
Analogously to above, static allocation requires us to specify the size of this array before compilation. Therefore an estimate $C_{\mathrm{SIP}}^{\mathrm{GB}}$ of how many SIPs there will be in maximum per grid box is required.
For each grid box the number of associated SIPs $N_{\mathrm{SIP}}^{\mathrm{GB}}(i,j,k)$ is remembered and saved in another 3-dimensional array.
Empty slots in $ice\_ind(i,j,k)$ are always to the right. Elements from the tail are copied into gaps, if necessary.

Issues This storage scheme is able to cope with dynamic generation and movement of SIPs up to a certain level.
Though, it fails utterly as soon as the number of SIPs goes above the allocated capacity. In this case a simulation would have to abort, as there is no possibility to allocate further memory. Choosing higher capacities right from the start works around this issue but works only up to a certain point - until all the memory of the compute nodes is occupied.
In the scenario presented in Figure 2 over 95% of grid boxes have no SIPs. The maximum capacity per grid box nevertheless needs to be chosen conservative enough to possibly accommodate a maximum of 1600 SIPs. This leaves many elements of the 4D *ice_ind* array empty.
Considering a conservatively guess of $C_{\mathrm{SIP}}^{\mathrm{GB}} = 3000$ and an average occupation of only less than one SIP per grid box, this means that for each grid box in average 2999 entries in *ice_ind* are not needed. In other words, in this scenario the overhead for *ice_ind* is 299900%. Considering our scarce memory resources, this is quite wasteful.
The complexity of indexing or deleting SIPs is $\Theta(1)$ and thereby really low. Adding SIPs is also $\Theta(1)$, as long as there is enough space available. Sorting this structure is done with the Quicksort algorithm. It is analogue to standard Quicksort for arrays, hence, the usual complexities apply here.

3.3 New storage scheme: singly linked lists

First of all, all attributes of a SIP are composed into its own data type. As stated in Section 3, it is 4 integers and 12 floating point numbers. Hereinafter this data type is referred to as *struct* following the name of composition types in the *C* programming language. Variables of this type are going to be saved into a linked list.
A singly linked list is a set of ordered nodes, where each node contains a pointer to its successor node and a data pointer field. The list is terminated by a null pointer.
Here, an additional *struct* is used for the beginning of lists. It contains a pointer to the head node, as well as the length of the list.

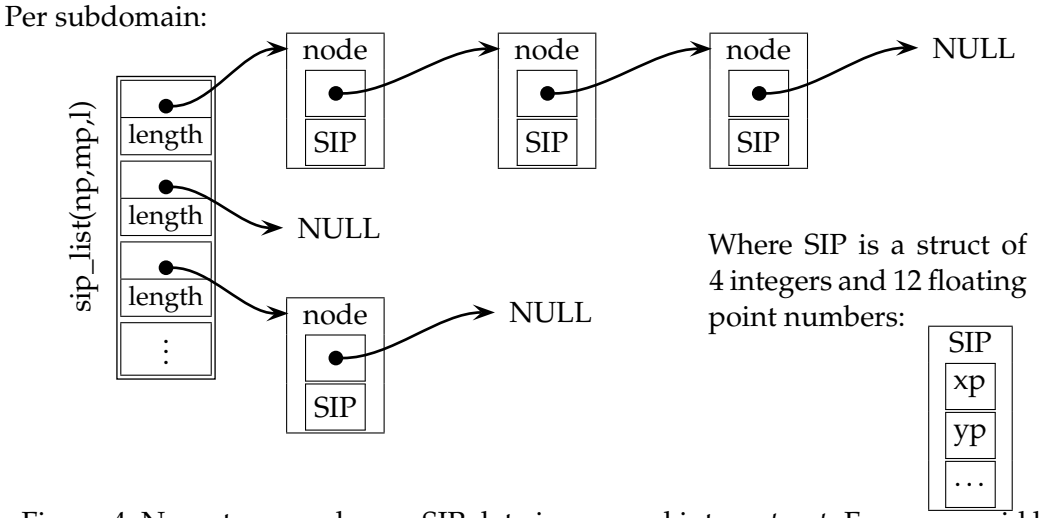


Figure 4: New storage scheme: SIP data is grouped into a *struct*. For every grid box there is a separate linked list which stores those SIP *structs*. The *sip_list* array contains a pointer to the head of each linked list.

There is one linked list for each grid box. After each timestep, the attribution of SIPs to grid boxes is checked. If a SIP moved to another grid box, its corresponding node is removed from the original list and inserted into the new one.
As the order of SIPs within a grid box is irrelevant, insertion is always performed at a list's head.
A doubly linked list, where each node has an additional pointer to its predecessor, is not necessary, as the microphysics modules always use forward sequential access. For deletion knowing the predecessor is required. However, during *foreach* loops we can simply remember the previous node, thus there is no need for an extra field inside the linked list itself.

Implementation details: Early versions of this storage scheme stored data as byte stream directly inside the node. Type casting was done via *FORTRAN*'s intrinsic *transfer* method. However, this is awfully slow and *FORTRAN* has no other means of type casting. Now the data field inside a node is a pointer of type *c_ptr*, comparable to a *void* pointer in *C*. It is pointing to a SIP structure elsewhere.
The implementation is divided into several modules, each with one specific task. They are ordered into a hierarchy of layers, where each layer is only allowed to call operations from any layer below. Access to higher layers is prohibited.
We allow for more than one modules on the same layer. However, in contrast to traditional open layer[2] architectures, method calls on the same layer are only allowed within a module. Access to foreign modules on the same layer is not allow.
The hierarchy is restricted to a depth of four layers. This is to prevent long chains of method calls, which would hurt performance.
Figure 5 visualizes the hierarchy. The singly linked list module, for example, has no knowledge about SIPs. Instead the SIP layer uses the data storage layer (in this

[2] a.k.a. transparent layering

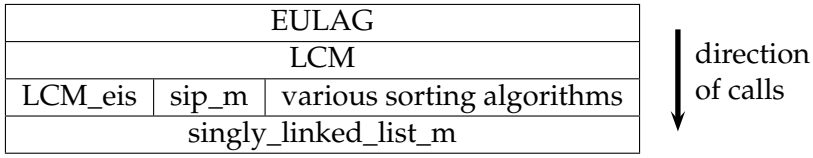


Figure 5: Functionality is encapsulated into modules, which are arranged into layers. For a given module, only methods from any layer below itself are permitted to be called.

case linked list), but is not allowed to access any LCM related functions.
This architectural scheme was chosen in order to decrease coupling between modules and increase testability and maintainability whilst at the same time maintaining adequate runtime efficiency.

3.4 Reasoning

Complexity Insertion at the beginning of a linked list has a complexity of $\Theta(1)$. Deletion is $\Theta(1)$, if the node prior to the one being deleted is already known. This is the case, as explained above.
The length of a list is remembered and updated on every insert or delete operation. Therefore counting is merely reading a variable and thus of complexity $\Theta(1)$.
During aggregation SIPs within the same grid box are sorted according to their vertical position. Section 4.2 provides a more detailed explanation. This is achieved with either an in-place Insertion sort, which as an $\mathcal{O}(n)$ best case and $\mathcal{O}(n^2)$ worst and average complexity. Alternatively an in-place Merge-sort algorithm with a complexity of $\mathcal{O}(n \log n)$ can be used. Both algorithm have a small $\mathcal{O}(1)$ auxiliary memory overhead.
Output of all SIPs within a subdomain in sorted order is rare and cannot be done by sorting in place. We use a modified Quicksort algorithm, as described in section 4.3. It has a best and average case complexity of $\mathcal{O}(n \log n)$, worst case complexity of $\mathcal{O}(n^2)$ and needs $\mathcal{O}(n)$ additional storage.

Highly dynamic storage As seen in figure 2, the average number of SIPs per grid box is less than one, while a few lists contain 1000 to 2000 SIPs. This is the optimal field of application for linked lists, as there is no need to determine a capacity or allocate space beforehand. Any necessary memory will be allocated dynamically at runtime, whenever a SIP is created. The only limiting factor for the maximum number of nodes is the size of the main memory.
The previous scheme had to allocate enough memory to accomodate the maximum number of SIPs for every grid box. Compared to this, linked lists save a great amount of memory.

Memory overhead Auxiliary memory overhead means that in addition to the data itself, some memory is required for managing those data. Using a linked list, this overhead consists of a fixed and a variable amount.

A linked list always needs memory for the head pointer and in our case an integer, which holds the list's length. Consequently even an empty linked list occupies 8 bytes, regardless of the number of actual SIPs stored in it. There is one linked lists for each grid box. The geometry of the simulated scenario, in particular the number of grid boxes, ghost layer cells and subdomains determine the amount of necessary memory. In case of the later discussed verification runs (see section 5.2) this adds up to about 1.34 MB. This is a typical value and many real scenarios will be in the same order of magnitude. In comparison to the total memory usage (see below), this is acceptable.
Furthermore there is a variable overhead, which varies with the number of SIPs. As there are usually a lot of SIPs, this is more critical than the fixed overhead. In the linked list scheme described above, there are two additional pointers required per node. This equates to 8 bytes per SIP and adds about 7% overhead compared to the pure data of one SIP.
We consider a hypothetical simulation, where 750 MB of memory are available per process. EULAG, LCM and the fixed overhead need approximately 130 MB of memory. This leaves us with enough memory for storing about 5.2×10^6 SIPs per process. The old storage scheme with commonly chosen capacities would have allowed far less than one million SIPs under the same conditions.

Spatial locality and cache misses Linked lists are stored non-continuously in memory. Thus, when traversing a list, the CPU cache cannot be used effectively and a lot of cache misses occur. Requests to the relatively slow main memory become necessary, which in turn has negative implications on the overall performance of a simulation.
The old storage scheme used indirect addressing and thus had the same issues. Therefore the runtime of the linked list scheme does not increase in comparison with the old scheme.
A better cache usage and consequently a speedup might be possible by a smart placement of data within the main memory. See Section 6 for more on this topic.

Comparison with other storage schemes Would an array-like storage structures be faster?
Arrays are continuous in memory and thus facilitate effective utilization of cache, when traversed linearly. To achieve a continuous layout in memory, upon each insertion or deletion the array has to be resized and some elements have to be shuffled and copied. This signifies an overhead. For small data types this overhead is neglectable and the overall performance of such a program is usually more efficient than one with linked lists thanks to better cache usage. However, the larger the underlying amount of data is, the higher the overhead of copying and reshuffling an array.
Currently SIPs are at 112 bytes each. As there is the option of chemical processes being introduced in the future, this could easily add another 400 bytes or more. For linked lists, the payload size is irrelevant on the performance of insertion or deletion.
Furthermore, insertion and deletion of SIPs is not confined to one point within the

micro-physics timestep routine, as shown in Listing 1. On the contrary, almost every process may create new SIPs or delete existing ones.
To conclude, the overhead of rearranging an array would add up significantly in our case, thus rendering linked lists more performant despite its cache misses.

4 Algorithms

4.1 Basic usage

Iterating over SIPs

```
use singly_linked_list_m
use sip_m

TYPE(llist_node_t), POINTER :: node
TYPE(sip_t), POINTER :: sip

node => llist_get_first(sip_list(i,j,k))
DO WHILE (associated(node)) ! i.e. unless node is NULL
    sip => sip_retrieve(node)

    ! do something with the sip structure. For example:
    position_x_in_subdomain = sip%xp

    node => llist_next(node)
END DO
```

Listing 2: Iterating over a singly linked list

As explained above, each process has a three-dimensional *sip_list* array. For each grid box (i, j, k) the corresponding element points to the head of a linked list. Hence iterating over all SIPs of a subdomain requires four nested loops: one for i, j and k respectively and one as shown in Listing 2.

Deleting a single SIP When deleting a node from the middle of a singly linked list, one needs to change the *next* pointer of the previous node. It has to point to the node, immediately following the node to be deleted. In general the previous node is unknown and has to be identified by traversing the list, starting from the head. This has $\mathcal{O}(n)$ complexity. If the previous node happens to be known, it is possible to delete a node with an order $\mathcal{O}(1)$ complexity instead.
In LCM we are indeed able to exploit this fact. Whenever there is the possibility of SIPs being deleted, linked lists are traversed in such a manner, that a pointer to the previous node is remembered.

Transfer between grid boxes At one point during a timestep, SIPs are advected. That is, their new position is calculated based on their sedimentation speed and the wind field velocity. This position is then saved within a SIP's data structure.

Depending on the direction and distance SIPs traveled, they might have moved into another grid box. Thus, the association between SIPs and grid boxes is potentially out of sync and has to be updated.
We iterate over all grid boxes and traverse the corresponding linked list of SIPs. For each SIP the correct grid box is calculated. See section 2.2.2 for what positions are associated with a grid box.
SIPs where old and new grid box are not identical have to be moved. Even though it would be possible to just delete a SIP from its old list and insert it into the correct one, this has the disadvantage that SIPs might be visited and checked more than once.
To avoid multiple visits, another approach is used. First, a temporary array of empty linked lists analogue to *sip_list(np,mp,l)* in Figure 4 is generated. After a SIP's grid box is calculated, it is inserted into the appropriate list of the temporary array. Finally the *sip_list* is overwritten by this temporary array. The order of SIPs within a grid box is reversed in this process.
Disadvantages of this approach are a $np \cdot np \cdot l \cdot 8$ bytes auxiliary memory overhead during this opperation and the fact, that all nodes are inserted into a new list and their *next* pointers are modified, even if SIPs did not move.
Which strategy would be more efficient depends on the scenario used, especially on the number and direction of SIPs moving between grid boxes.

Transfer between subdomains The computation domain is decomposed into several subdomains, on which EULAG solves the differential equations in parallel. In addition to its own values, each process has a layer of ghost cells around its subdomain. These cells contain a read-only copy of cells from a neighbouring subdomain and are updated several times during a timestep.
Considering SIPs the idea of a ghost layer is adopted, but it is used in a considerably different way. There is one or more[3] layers of ghost cells around each subdomain. They contain a linked list for saving SIPs. However, at the beginning of a timestep those lists are always empty, as there is no need to know anything about SIPs in neighbouring subdomains neither during the calculations in EULAG nor during EULAG-LCM. If an ice crystal is advected across the subdomain boundary, it is first saved in the linked list of an appropriate ghost layer cell. After each timestep those SIPs are communicated to the corresponding neighbouring process, where they are sorted into the appropriate grid box.
As pointers are only valid within the current process, transfering an entire linked list node would be nonsense. Instead only the SIP data structure has to be transfered. A custom MPI data type is defined for SIP data structures.
On the sending side, after counting SIPs in a certain ghost layer region, an array is allocated and SIP data is copied into it. After notifying the neighbour about the number of SIPs, this array is sent. The contents of all ghost layer lists can now be deleted entirely. Thanks to the use of non-blocking MPI communication, some of these tasks can be done while waiting for the receiver to get ready.

[3] In EULAG the width of a ghost layer can be configured. LCM uses the same width, although there is usually no need for more than one ghost layer cell.

After the receiver gets notified about the number of SIPs to be received, it allocates an array with sufficient size. Upon receipt, each SIP is encapsulated into a linked list node and then inserted into the linked list of the appropriate grid box.

4.2 Sorting SIPs for aggregation

Aggregation is explained in section 2.2.1. The algorithm for aggregation is described by Sölch (2009, p. 42) and is applied to each grid box separately. The idea is to scan all pairwise combinations of SIPs within one grid box for possible collisions and determine if they would stick together. However, these checks are of high complexity and performing n_{SIP}^2 of them is a waste of computation time. But it is possible to reduce the number of required checks by first sorting all SIPs by their z-coordinate and then probing only close neighbours.
Aggregation is the only physical process, where the order of SIPs plays a role. It is therefore possible to modify the order within the storage and sort a linked list in-place. This has the advantage of not needing any additional memory. Additionally it might lead to data being slightly presorted: The order of SIPs regarding their z-position might change between time steps due to differential sedimentation, aggregation or insertion/deletion of SIPs. However, this changes are often small and it can be assumed, that SIPs are slightly presorted most of the time.
Most of the time during a simulation, there are less than 1000 SIPs per grid box. See Figure 2 for some histograms.

In-place insertion sort has a constant auxiliary space overhead. It is very efficient for almost sorted lists or small lists. However, as soon as the presorting gets worse, the complexity goes from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$. In this case other algorithms are more efficient.
Mergesort, for example, has a best/average/worst case time complexity of only $\mathcal{O}(n \log n)$. It can be implemented as in-place algorithm with constant $\mathcal{O}(1)$ auxiliary space overhead.
A hybrid algorithm like Timsort would be able to combine the advantages of both algorithms.
As the impact on runtime of the above mentioned influences is difficult to estimate, both in-place merge sort and in-place insertion sort were implemented for linked lists. Tests in real world scenarios are yet to be done.

4.3 Saving SIPs to persistent storage

There are a handful of variants to write simulation results to a persistent storage. All of them have their specific field of application and purpose based on their further usage.

Restart files All relevant data to resume a previous computation at a later point in time have to be contained in a restart file. They require variables to be written and read bit-identical.
Technically it is not necessary to write the entire linked list of SIPs to disk, as the *next* pointers are only valid within the current process context anyways. Instead

we only need to write the *SIP structs*. Upon retrieval within a new simulation this data will be inserted into newly generated linked lists.

Sorted output for plotting: There is a *IDL* plotting script, which generates histograms of the ice crystal radius of each subdomain for each time step. It expects an array of SIPs as input and requires them to be sorted by the effective ice crystal radius.
For this reason, the output in LCM has to take place per subdomain, not per grid box. Furthermore it is necessary to sort all SIPs within a subdomain, but without modifying the original linked lists. In particular the assignment of SIPs to grid boxes must not be changed.
A simple approach would be, to first deep-copy the linked lists of all grid boxes within a subdomain. Afterwards, these copied lists could be concatenated to form a single huge list. In the end, an in-place sorting algorithm could be applied. Albeit this approach is simple, it would have a memory overhead of 100% and is therefore not favourable.

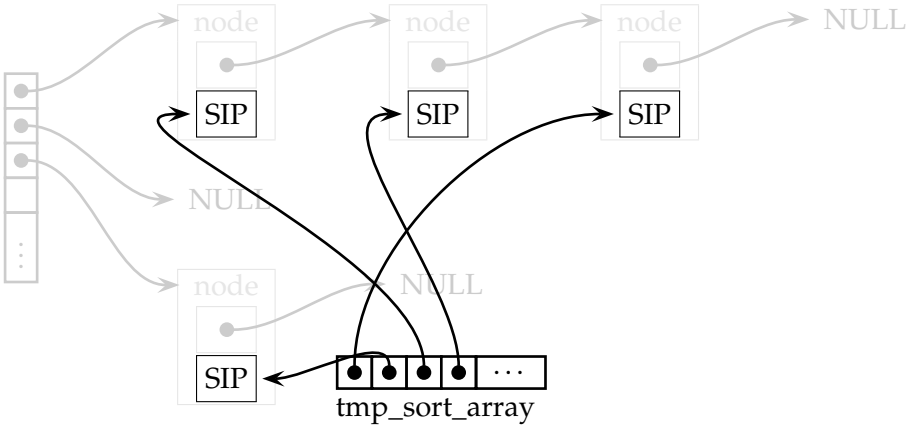


Figure 6: Approach for sorting all SIPs within one subdomain without modifying the linked lists of each grid box: A temporary array contains pointers to the original SIP structures. Then the array itself is sorted by comparing the values each element points to.

There is another possibility, which also has an $\mathcal{O}(n)$ auxiliary memory overhead, but which is more memory-efficient nevertheless: A temporary array of pointers to the original SIP structures is generated and sorted. The overhead is thus only $4\,\mathrm{B}$ per SIP.
An outline of this approach is shown in Figure 6. The total number of SIPs in a subdomain is known beforehand and does not change during output. A temporary array of appropriate size is allocated. Iterating over all grid boxes, we traverse each linked list and initialize the array elements with pointers to a SIP structure. Subsequently any sorting algorithm can be applied to this array, with one constraint: it is not the pointers' absolute values, which have to be compared, but rather a value of the SIP struct they point to.

In addition to the above mention low auxiliary space overhead, this method has another advantage: changing the order of SIPs is cheap, as it is simply swapping elements within the temporary array.

5 Testing and verification

In the following sections, several tests are presented, which demonstrate the validity of the new approach. This includes tests of basic functionality on one hand, but also comparisons of results with the original static version on the other hand.
It is important to note, that we do not expect bit-identical results in these comparisons. This is primarily due to the usage of random numbers at several points during the calculations, such as randomly placing SIPs within a grid cell upon initialization or adding turbulent fluctuations to the particle velocity.
Some physical processes like nucleation are non-linear and very sensitive to small changes in its input parameters. Therefore, these initially small fluctuations might add up and lead to some spread in the results.

5.1 Unit testing

Unit testing is employed to ensure that the basic storage and sorting modules work as expected and meet all requirements regarding their functionality. In particular this approach allows testing and debugging of specific components without the overhead of using the *EULAG/LCM* context and thus saves computation time and reduces complexity.
Unit testing has proven to be quite valuable during development, particularly as there are some incompatibilities between compilers: initial versions of the code, compiled with a local *GNU Fortran* compiler, were performing flawlessly, while the same code led to segmentation faults with the *IBM XL Fortran* compiler at *DKRZ*.
There are not many unit testing frameworks for *Fortran* available. Most of them are written not in *Fortran* itself, but rely on scripting languages like *ruby* or *python*.
We will be using *FUnit*. This framework was initially developed at *NASA*. Although not formally announced as unmaintained, it turned out to be not actively developed anymore. This required me to fix some bugs in the framework myself.
Tests are written in *Fortran*, mixed with "a small set of testing-specific keywords and functions" (NAS, 2009). *FUnit* handles the translation of those keywords into complex *Fortran* statements and afterwards compiles and runs the tests automatically.
Tests currently have to be initiated manually as there is no build automation system.
The following unit tests are available:

- For module singly_linked_list_m:
  - create node
  - insert node into a list
  - delete node from a list
  - delete an entire list

  - store and retrieve payload data

- For module singly_linked_list_merge_sort_m
  - sort lists of length 0, 1, and 42

- For module sllist_ptr_array_quicksort_m
  - sort lists of length 1 and 42

5.2 Test of individual mechanisms

The general setup is the same for all tests, see Table 1.

Domain dimensions	$n = 60$	$m = 40$	$l = 40$
Grid box dimensions	$\mathrm{d}x = 10\,\mathrm{m}$	$\mathrm{d}y = 20\,\mathrm{m}$	$\mathrm{d}z = 5\,\mathrm{m}$
Subdomain layout	$4 \times 4 \times 1$		
Subdomain size	$np = 15$	$mp = 10$	$l = 40$

Table 1: Domain and subdomain layout, used in all tests of individual mechanisms.

The domain consists of 4×4 column-shaped subdomains with $15 \times 10 \times 40$ grid boxes each. The dimension of a grid box is not symmetrical.
There is a ghost layer[4] of two cells around each subdomain, except for the upper and lower boundaries, where no neighbouring subdomains exist.
Timesteps are $1\,\mathrm{s}$ for dynamics, deposition and advection.
Aggregation is switched off.

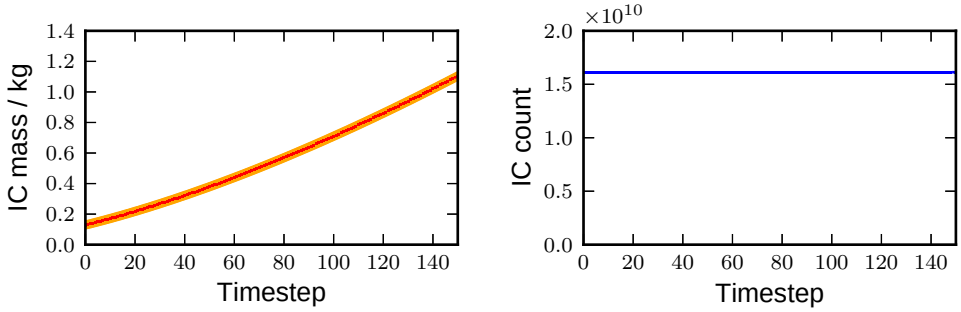
5.2.1 Deposition

The purpose of this test is to verify the physics of the deposition/sublimation routine.
Initially one SIP is placed per grid box. In order to test the growth of different-sized ice crystals, we prescribe the ice crystals' mass to be proportional to their position in x-direction: $i \cdot 10^{-12}$kg. Advection and nucleation are switched off. In this way, all ice crystals are stationary and their number remains constant.
The relative ambient humidity is set to 147%, which is above the threshold of 100%. Therefore water molecules of the gaseous phase freeze onto the surface of existing ice crystals, leading to a continuously growing ice mass.
Figure 7a shows that the total ice mass increases over time, while the number of ice crystals remains constant.
The two curves in figure 7a are with the old storage scheme and linked lists respectively. They agree and thus show, that the new implementation is right.

5.2.2 Sublimation

All settings are identical to deposition, with one exception: this time, the relative ambient humidity is 75%. Hence ice crystals shrink and eventually are so small,

[4] ghost layer cells are called HALO-cells in EULAG-LCM



(a) The accumulated ice crystal mass increases over time as the air is supersaturated and ice crystals grow. The two curves for the old and new storage scheme match.

(b) There is no nucleation, hence the number of ice crystals remains constant.

Figure 7: Verification of the physics behind *deposition*.

that they can be regarded as completely sublimated. In this case, the corresponding SIP is removed.
The physics behind sublimation is the same as with deposition. The only additional operation is deletion of SIPs, which is tested here.
The total ice crystal mass starts at a certain level and decreases over time. The rate of sublimation is nearly constant at first but will decrease as less and less ice crystals are available. A plot is omitted here, in favour of brevity.

5.2.3 Vertical transport and sedimentation

Vertical transport with constant speed: This test aims at verifying how SIPs, which cross vertical domain boundaries, are handled.
One SIP is placed in the middle of each grid box. Advection is switched on and ice crystals are forced to move with a fixed velocity $(u, v, w)^\mathsf{T}_{\text{local}} = (0, 0, \pm 1\,\text{m s}^{-1})$ either up- or downwards. The only difference between those scenarios is, that ice crystals which exit the domain through the lower boundary are counted as sedimented ice crystals, while this is not the case for the upper boundary.
Real world scenarios are usually set up in a way, such that the domain extends beyond the cloud top. Although ice crystals should not exit the upper boundary by this approach, it cannot be excluded and therefore has to be tested.
Figure 8 shows the temporal evolution of the ice crystals number.
Every $5 (= \mathrm{d}z/w)$ timesteps one layer of SIPs exists the computation domain.
Only SIPs leaving the domain through the lower boundary are counted as "sedimented".

Diagonal lines of same mass but different speed: The objective of this test is to validate the calculation of the sedimentation speed and its dependence on particle mass.
One SIP per grid box is placed in layer $l = 16$, i.e. a horizontal layer in the middle of the domain.

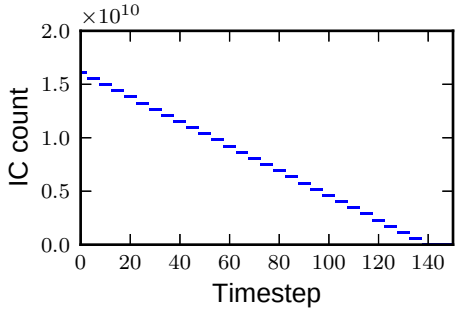


Figure 8: Discrete horizontal layers of ice crystals with prescribed downward speed exit the domain. The number of crystals shown here decreases step-wise.

What makes this test exceptional is that the mass of ice crystals was chosen to depend on the grid box they are placed in: $m(i,j) = (1+i-4+j-4)10^{-10}\text{kg}$.
Instead of prescribing a constant vertical velocity, we use the true sedimentation speed, which depends on the ice crystal mass.

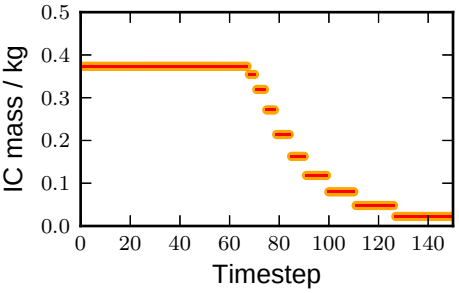
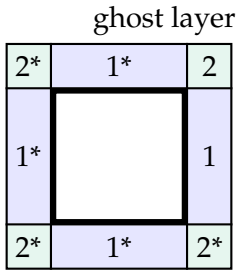


Figure 9: Sedimentation speed and thus the time for an ice crystal to exit the domain depends on its mass. The temporal evolution of the accumulated ice crystal mass agrees for both storage schemes.

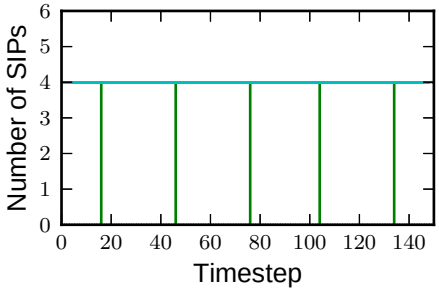
The results are shown in figure 9. Diagonal stripes of SIPs leave the domain at the same time.
Due to different sedimentation speeds, SIPs fan out. As mass and sedimentation speed have a non-linear relationship, the vertical distance between light particles leaving the domain is much higher than the distance between heavier particles. As a consequence, the time between steps in the graph increase over time.
The difference in ice crystal mass between steps is also not constant. While e.g. mid trough a lot of SIPs with medium mass add up to a big mass, at the end only a small amount of fairly light SIPs sediment.
The curves in figure 9 are for the old storage scheme and the new linked list approach. They are identical, suggesting that the new implementation is correct.

5.2.4 Lateral transport across subdomains

Test №1 This test aims at verifying communication via ghost layer cells which are marked with "1" in figure 10a. Edges marked with "1*" can be tested analogously by rotating the setup.



(a) Ghost layer around a subdomain. SIPs within different regions of this layer have to be send to different corresponding neighbouring subdomains.



(b) Four SIPs move through the domain. The transfer between subdomains is succesfull as the number of SIPs ist constant.

Figure 10: Test of the lateral transport of SIPs.

The leftmost subdomains are initialized with one single SIP each. It is placed in the center of the respective subdomain. All other grid boxes or subdomains remain empty. In this way, we have 4 SIPs in total.

SIPs are forced to move with a given velocity in positive x-direction.

Figure 10b shows the total number of SIPs as well as the number of SIPs transfered in each time step. The number of SIPs remains constant and the correct amount of SIPs is transfered between individual processes, whenever SIPs reach a subdomain boundary.

Test №2 Another special case is diagonal communication across the corner of a subdomain. The associated ghost layer cells are marked with "2" in figure 10a.

Now every grid box within the entire domain is initialized with one SIP. Again, SIPs are forced to move with a specified velocity, which this time, is diagonal.

Results are similar to figure 10b, but with more frequent transfer of SIPs.

5.3 Benchmarks

Unterstrasser and Sölch (2013) determine how many SIPs are required in order to reach statistical convergence for different micro-physical processes. They use several well defined scenarios for their sensitivity studies with a high variety in the number of initial SIPs: 2.7×10^5 to 5.4×10^6 for the runs A1 to A5.

This simulation serves as a comparison for benchmarking the modified version of EULAG-LCM in terms of memory usage and computation time.

5.3.1 Setup

All simulations start with a 30 min old contrail and calculate its temporal evolution over another six hours. During this time, turbulent dispersion leads to an expansion of the contrail's cross-section and it is further spread due to wind shear and sedimentation.
The initial number of SIPs is controlled, for example, by the maximum number $\nu_{\max}$ of ice crystals represented per SIP. Some parameters for different runs are listed in Table 2.
Nucleation and aggregation are switched off. The ambient relative humidity is 20% at the lower domain boundary and linearly increases up to 120% at the upper boundary. Further details can be found in Unterstrasser and Sölch (2013).

Run	$\nu_{\max}$	κ	$\rightarrow N_{\mathrm{SIP,init}}/10^6$	$C_{\mathrm{SIP}}^{\mathrm{GB}}/10^4$	$C_{\mathrm{SIP}}^{\mathrm{subdomain}}/10^6$
A1	$2 \cdot 10^6$	120	0.72	3	2
A2	$2 \cdot 10^6$	18	0.55	3	2
A3	$2 \cdot 10^6$	360	1.1	3	2
A4	$2 \cdot 10^7$	120	0.27	3	2
A5	$2 \cdot 10^5$	120	5.4	8	3.5

Table 2: Parameters for different benchmark runs. The maximum number $\nu_{\max}$ of ice crystals per SIP and κ, which is explained by Unterstrasser and Sölch (2013), influence the number of initial SIPs. Runs with the old storage scheme are performed with the specified capacities $C_{\mathrm{SIP}}^{\mathrm{subdomain}}$and $C_{\mathrm{SIP}}^{\mathrm{GB}}$.

To allow for better comparison, we further simplified the setup of Unterstrasser and Sölch (2013): Only influences on the water vapour are fed back to EULAG, latent heat is neglected. Vertical forces due to density variations of air with different water vapour content are ignored. No perturbations are added onto the sheared wind field.

5.3.2 Measurement

The objective is to measure memory usage and runtime.
The means of measurement are limited by what software is available at the DKRZ super computing center.

Memory usage The linked list-version of the simulation dynamically allocates a variable amount of heap memory instead of statically allocating memory on the stack. An optimal tool or library to compare both simulations and visualize these changes would measure the evolution of heap and stack usage over time.
Unfortunately, no such tool is available at DKRZ: While *valgrind-massif* did not compile on the AIX architecture, frequently polling the *C* system call *getrusage()* would be possible, but is quite complex from *Fortran*.
There is also the *rusage* command line tool. It provides the same values as *getrusage()* but only once per execution.

The average memory size used can be obtained by dividing the integral memory size by runtime. It has to be taken into account that computations with a large number of SIPs take longer and thus memory usage for those is weighted more than memory usage for small SIP numbers. Thus these values are not comparable between processes.
Peak memory size or *maximum resident set size (max_rss)* on the other hand is an absolute value and can be easily set into relation with the maximum number of SIPs. The number of SIPs gets written to a file every timestep and its maximum is simple to extract.
The y-intercept is determined by run where the creation of SIPs is suppressed.

Runtime The overall runtime can be determined with *rusage* too. It provides values for wall-clock time, user time and system time.
Incorporated within LCM, there are some methods for performance analysis. They provide similar measures like the *prof* family of performance analysis tools. However, there are some differences: code has to be instrumented manually, call chains cannot be analyzed and the output is for sections of marked code and can thus be much more detailed.

5.3.3 Results

Figure 11 shows the relationship of peak memory usage to peak SIP number. Both parameters are measured per process. Measured points for all runs A1 to A5 are shown for both the linked list scheme and the original storage scheme. Solid lines denote the respective theoretical memory usage.
Comparing the old version of the simulation with the *linked list*-version, it is apparent that the memory usage could be significantly decreased. This reduction depends on the choice of $C_{\mathrm{SIP}}^{\mathrm{GB}}$and $C_{\mathrm{SIP}}^{\mathrm{subdomain}}$. For runs A1 to A4 the reduction is about 62% and 75% for run A5.
The graph of the *linked list*-version indicates a linear increase of memory usage with the number of SIPs.
What is interesting in this data is that the slope of the measured curve is higher than the theoretical curve. This is most likely owed to reservation of memory on the stack whenever a called method has local arrays, which size depends on the number of SIPs. Unfortunately this hypothesis cannot be tested, as there are no appropriate tools available at DKRZ.
Even more striking are the results from the old simulation: The memory for potential SIPs is statically allocated right from the start and one would not expect the memory usage to go any higher than this. However, there is a significant increase with the number of SIPs. In simulations with 0 to 2×10^6 SIPs we observe memory usages of up to four times the amount of theoretically required memory.
Another eye-catching difference between both storage schemes is the size of fluctuations. For linked lists the relation between maximum $N_{\mathrm{SIP}}^{\mathrm{subdomain}}$and maximum memory usage is linear with almost no outliers. The old storage scheme, in contrast, has strong fluctuations in the range up to several hundred megabytes. An explanation could not be found.

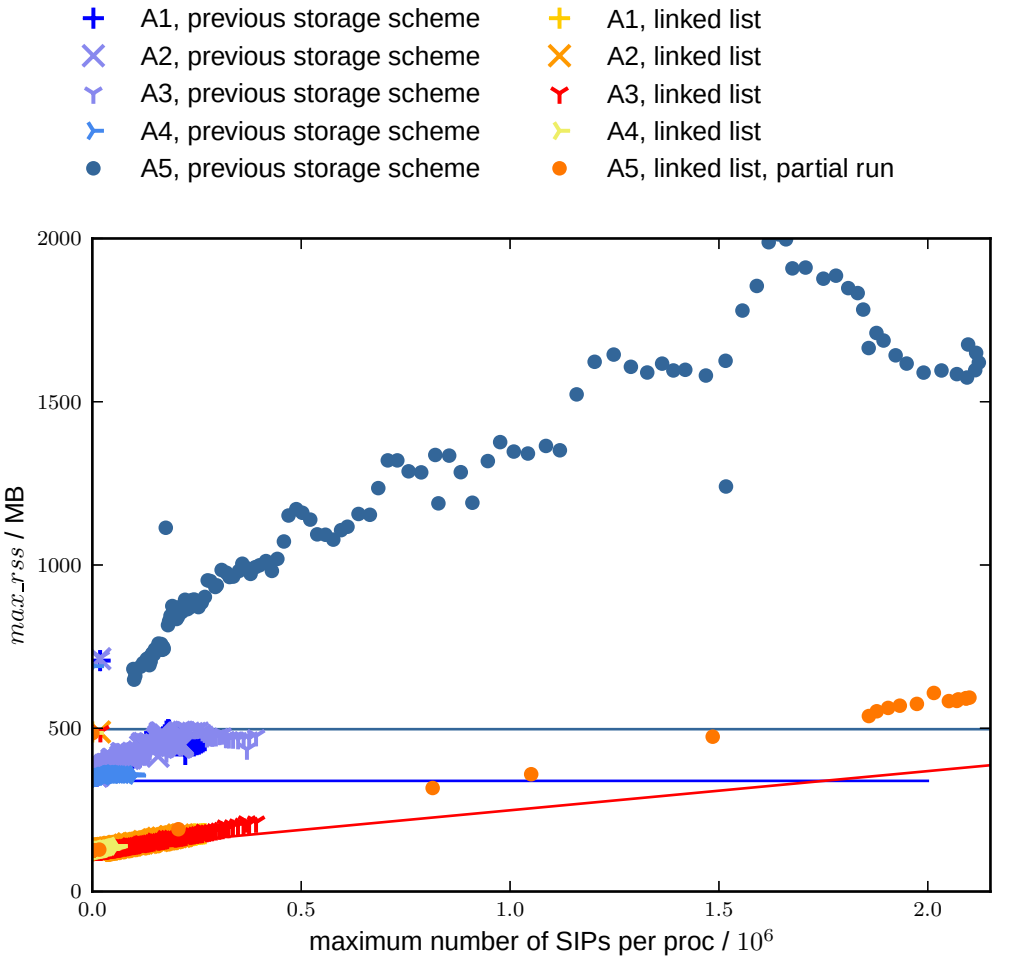


Figure 11: Peak memory usage (maximum resident set size from *rusage*) against peak SIP number. Both values are per process. Markers denote measurements with either the linked list scheme or the original storage scheme. Where no data with zero SIPs was available, the y-intercept as determined via a zero SIP run. Theoretical curves are drawn with solid lines. For the old scheme, they depend on $C_{\text{SIP}}^{\text{subdomain}}$ and $C_{\text{SIP}}^{\text{GB}}$, which are stated in Table 2.

6 Possible improvements

During this thesis an improved storage scheme for SIPs was developed. There has been a great effort to improve memory usage and choosing efficient algorithms. One point which still needs some work is to improve locality of reference, thus reduce cache misses and improve runtime.

In the current implementation, nodes have a *c_ptr* pointer to a SIPs data struct.

An easy possibility to improve spatial locality and cache usage is to put SIP data directly into the node. There are two possibilities for that:

1. The *c_ptr* field within a node could be replaced by a general byte array field into which SIP data is written directly. This requires type casting from the SIP struct type to a byte array type.
   The first version of the linked list storage scheme used exactly this approach and *FORTRAN*'s intrinsic *transfer* function for the type cast. However, the *transfer* function is so slow that the overall runtime of the simulation was by a factor of two higher than the original code. A combination of *c_ptr*, *c_f_pointer* and copying data might possibly give better results.

2. Sacrifice the universality of linked lists. This is to say, that instead of having a general data field in nodes it could be modified to only store SIP structs. The layout in memory is hence such that SIP data is stored within the node itself, which reduces cache misses. Additionally no type cast methods are necessary.

The exact influence on runtime of any of those improvements cannot be predicted.

Memory allocations are managed manually within the linked list module. This enables us to optimize the arrangement of data in memory, without changing the scheme itself. A possible approach is to place SIPs, which are close together in reality, close together in memory. However, the level of difficulty varies with the situation: When reading initial conditions from a file, the number of SIPs within a grid box is known and memory for nodes can be allocated as a block of appropriate length. The nucleation of ice crystals, on the other hand, cannot be predicted and copying or preallocation would become necessary to obtain contiguous blocks of nodes in memory.

7 Summary

During past simulations it became apparent that the original implementation has room for improvements in terms of storage of SIPs. It uses a static memory scheme, which is not well suited for cirrus clouds, where particle concentrations are highly variable.
Within this thesis a new storage scheme based on singly linked lists has been developed and implemented.
The new scheme is more flexible as it dynamically adapts to the simulated scenario. With this scheme time complexity for often used storage operations (forward-sequential access, insertion, deletion and counting of SIPs) is of order $\mathcal{O}(1)$. Several sorting algorithms have been implemented and satisfy a multitude of requirements. Memory overhead has been drastically reduced.
All changes to the code have been tested. Unit test were performed for each storage related module. Special scenarios were designed to test the correctness of individual processes like sublimation or sedimentation. Physical results were compared to the original implementation and found to be the same.

Typical real world scenarios, as published by Unterstrasser and Sölch (2013), were repeated and used as benchmarks. Physical results were identical to the reference. The required memory of the overall simulation was reduced by a factor of three. This is a drastic improvement over the original version.
The improved implementation facilitates the execution of future calculations as the required amount of memory is lower, thus extending possible target architectures. Furthermore no more guesses about the maximum storage capacities are necessary. In accordance to current trend in HPC architectures, this scheme prepares EULAG-LCM for future generations of supercomputers with less memory per CPU core.

References

W. Grabowski and P. Smolarkiewicz. A Multiscale Anelastic Model for Meteorological Research. *Mon. Weather Rev.*, 130(4):939–956, 2002.

M. Lainer. Numerische Simulationen von langlebigen Kondensstreifen mit Lagrange'scher Mikrophysik (in German). Master's thesis, LMU München, 2012.

Fortran unit testing framework documentation. NASA, Nov. 2009. URL `http://nasarb.rubyforge.org/funit/`.

J. Prusa and P. K. Smolarkiewicz. An all-scale anelastic model for geophysical flows: dynamic grid deformation. 190(2):601 – 622, 2003. doi: 10.1016/S0021-9991(03)00299-7.

P. Smolarkiewicz and L. Margolin. MPDATA: A Finite-Difference Solver for Geophysical Flows. 140(2):459–480, 1998.

P. K. Smolarkiewicz. A simple positive definite advection scheme with small implicit diffusion. *Mon. Weather Rev.*, 111:479–486, 1983.

I. Sölch and B. Kärcher. A large-eddy model for cirrus clouds with explicit aerosol and ice microphysics and Lagrangian ice particle tracking. *Q. J. R. Meteorolog. Soc.*, 136:2074–2093, 2010. doi: 10.1002/qj.689.

I. Sölch and B. Kärcher. Process-oriented large-eddy simulations of a midlatitude cirrus cloud system based on observations. *Q. J. R. Meteorolog. Soc.*, 137(655):374–393, 2011.

I. Sölch. *Ein Euler-Lagrange'sches Zirruswolken Modell mit expliziter Aerosol- und Eismikrophysik: Studien zur Aggregation von Eispartikeln*. PhD thesis, Ludwig-Maximilians-Universität München, 2009.

S. Unterstrasser and I. Sölch. Speeding up a Lagrangian ice microphysics scheme. *Geosci. Model Dev. Discuss.*, 6:3787–3817, 2013. doi: 10.5194/gmdd-6-3787-2013.