Internship Parallel Computer Evaluation

Parallelization of a Lagrangian Particle Diffusion Model

Cedrik Ansorge, Johann Weging

October 29, 2011

Contents

1 Preface

This paper is about the parallelization of a Lagrangian particle diffusion model.

1.1 Motivation

A Lagrangian particle diffusion model can be applied for different use cases. One possible scenario is to compute the air pollution of a city where factories and streets emits CO_2 and certain atmospheric conditions leads to a specific spreading of the particles in the air.

Another possible use case is a nuclear accident like a core meltdown at a atomic power plant, where atomic radiation emits in the air. The Lagrangian model can predict how the nuclear cloud spreads under different conditions. Than the spreading can be used to compute the concentration of the atomic radiation at a certain time at a specific place to evacuate the affected areas before the atomic cloud reaches that area.

1.2 Structure of this paper

First this paper will provide some information about the theoretical background of the computational model. Than the generic implementation is discussed and the different steps of the computation are explained. The next part is about the parallelization concepts that could possibly be implemented. Than some technical details are provided, like the used data structure and the load balancing. Further more the implementation of OpenMP and. MPI at least the implementation state is given the still existing problems with the software.

There are some terms that need to be clarified.
Model grid: This is the whole terrain that will be computed.
Particle: One single molecule floating in the wind field.
Compute unit: One unit that runs computation like a single core of a cpu.
Compute node: One computer consisting of multiple compute units.

2 Theoretical background

This section gives some of the theoretical background about the Lagrangian particle diffusion and the computational model. LaDis computation works with pre computed wind fields. So the terrain which is computed is predefined by the wind field that is loaded. Inside the wind field particle emitting sources can be defined. In a real world scenario this could be streets with cars or factories. Important to note is, that particles only interact with the wind field not with other particles. There is no collision detection, because the particles are so small that there is no significant effect.

The Lagrangian approach of turbulent particle diffusion computes the trajectories of particles to determine each individual displacement caused by the wind field. This enables the possibility to compute the particle concentration at any given time at any point of the model grid.

3 Compute model

There are some questions about how the general computation model that need to answered before talking about parallelization. The first question is how to load the data in to the compute model. This includes the parameters about the count of the particle emitting sources, the maximal count of particles in the area to compute and the rate of particles emitted by a source. Furthermore the terrain and the meteorological wind data.

The next problem is the arrangement of the computational steps. The model generally consist of of two loops, the time integration and the stochastic integration. The time integration is the outer loop so it is possible to collective read the data for the next time step.

To compute the spreading of the particles it is necessary to know the wind direction and speed in every point of the model. This is accomplished through 3D-linear interpolation. This brings the question up how to handle obstacles like buildings. If the wind field is precomputed or measured and get loaded into LaDis, the vertical wind in front of a obstacle is nearly 0 and the particles will maybe accumulate at this obstacle. The resolution of the grid model must be high enough, if one time step is computed, that the particles don't fly through this obstacle.

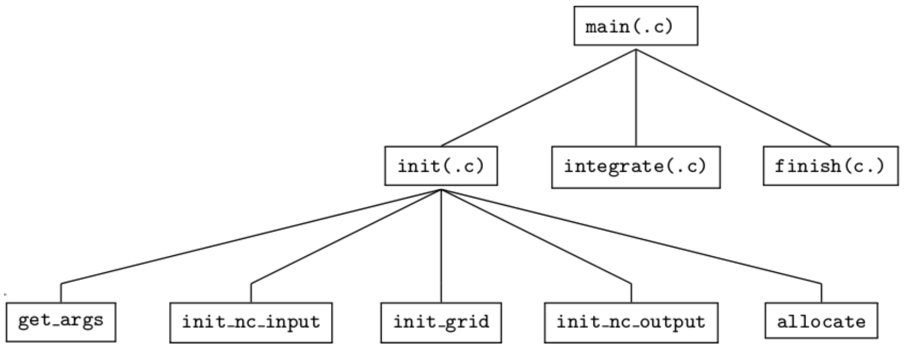

Figure 1: Compute model initialisation operation sequence order.

Figure 1 shows the general initialization of LaDis. First all the needed variables and parameters are read from a namelist file. The namelist file contains all necessary variables and values for the computation. Then the wind field is loaded from a provided NetCDF file. Is the wind data loaded, the compute grid model will be initialized. Now the output file of LaDis is initialized. At least the memory need for the computation is allocated.

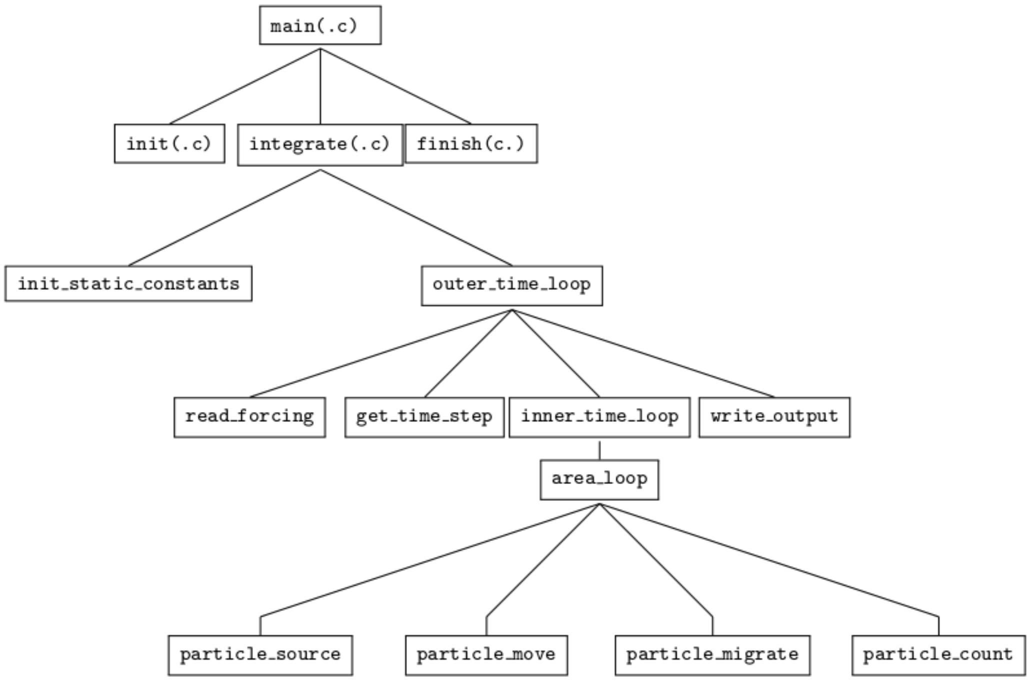

Figure 2: Model compute sequence order.

Figure 2 show the general operation order of the computation. The whole computation happens inside the outer time loop. First the forcing data for the next time steep is read. Than the next time step is read. Next the inner time loop, loops over a area containing multiple particles. The area concept is explained in more detail in the section **??**. The first step of the area loop is to emit particles to the area, if the area contains a source. Than the particles are moved, explained in 2. Next the particles are migrated to the area and or moved to a neighbour area. Al least the particles inside the area are count. This is necessary because the whole compute model is only allowed to consist of a maximum number of particles to avoid memory problems. After every area is computed the output is written for the time step.

Figure 3 shows sequential computation of one particle source. The computation shows the particle distribution after 300 seconds. The source is located at coordinates $30, 30, 90$ and emits 10 particles every second. The particles float to the upper right and accumulate at the upper right corner.

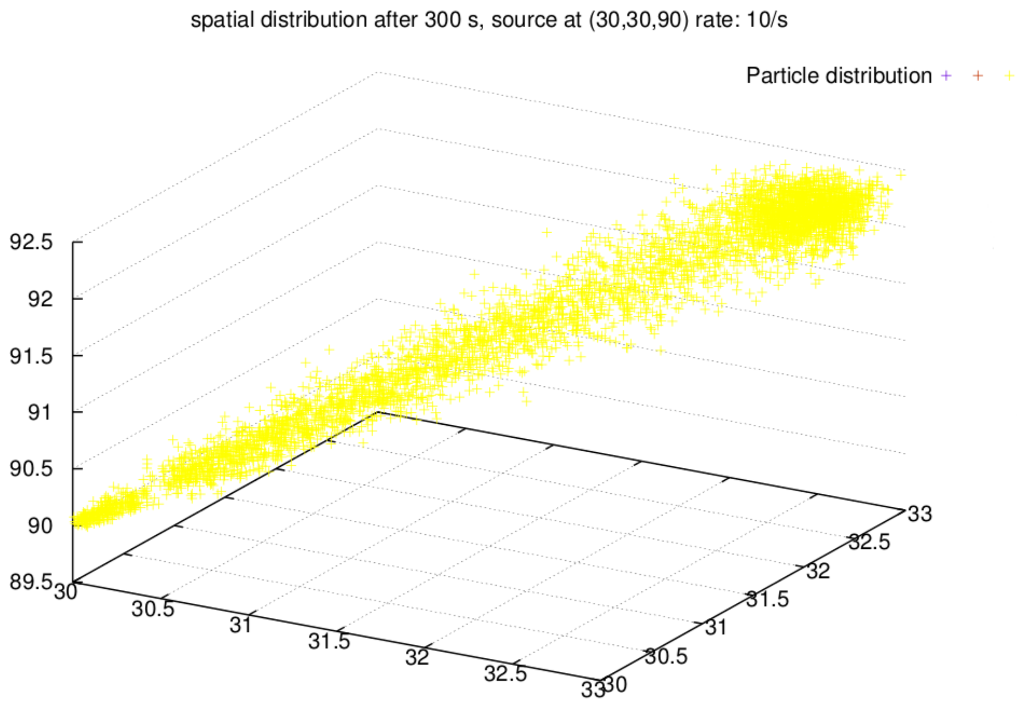

Figure 3: Plot of a sequential computation of one area with a single emitting source.

3.1 Parallelization concept

There a basically two approaches about how to parallelize the computation. The first one is a particle based approach where the particles where spread among the processing units.

The other parallelization plan is area based. The region that has to be compute is divided in several areas. These areas are than scattered among the processing units.

3.2 Particle based parallelization

The Particle based Parallelization, was not implemented in the program because a part of the initial task was to handle load balancing. This section will only hand an idea of how this approach could be implemented.

Figure 4 show the particle based approach. Every dot represents one particle and every compute unit handling particles is represented by one color. There is no mayor load balancing needed, because if every compute unit has the same amount of particles, particles that are emitted by a source could be scheduled in a round robin manner. If some of the particles leave the grid model through the outer boundaries, the processing unit which lost the particles will get assigned new particles from the source. After some time maybe the maximum number of particles is reached, if now some of them flood out of the model grid boundaries the sources will emit new particles and assigned them to the processing unit which lost the particles until the maximum count of particles is hit again. This guarantees that every processing unit will handle nearly the same amount of particles. By computing multiple particles at once it would be pssible to enable the use of the streaming SIMD extensions (SSE).

3.3 Area based parallelization

Figure 5 shows the implemented parallelization concept. The model grid is divided in multiple hash areas. Each compute unit gets multiple hash areas to compute, this will enable load balancing like explained in section 3.4. Every hash area is represented by one data structure containing the particles inside the hash area. If a particle migrates from one hash area to another they are transferred to a different data structure. This data struct can be located at the same processing unit, a different processing unit on the same compute node or a different compute node on the cluster. Furthermore it is possible to compute multiple hash areas inside of one area loop, this could enable the use of SSE.

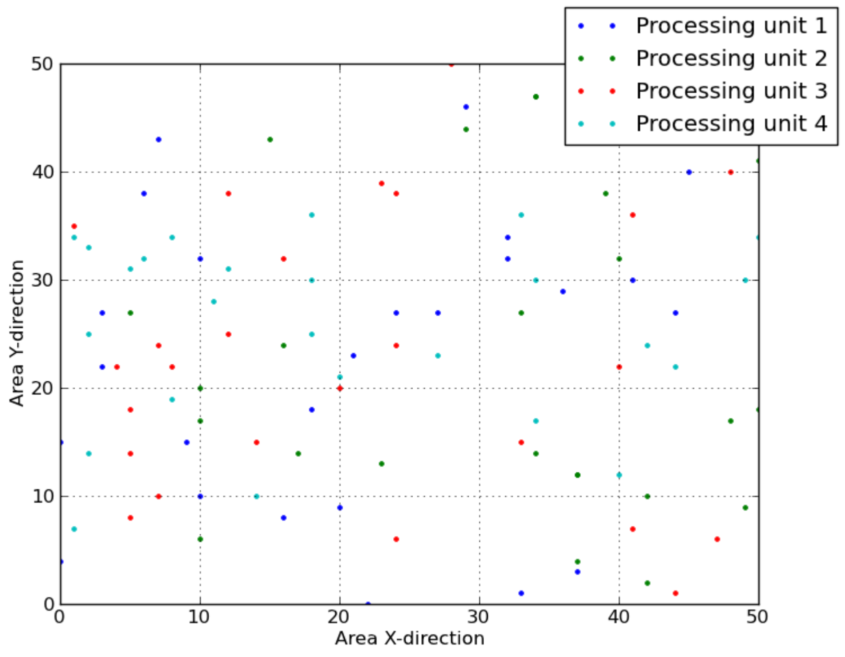

Figure 4: Particle based parallelization with a two dimensional grid

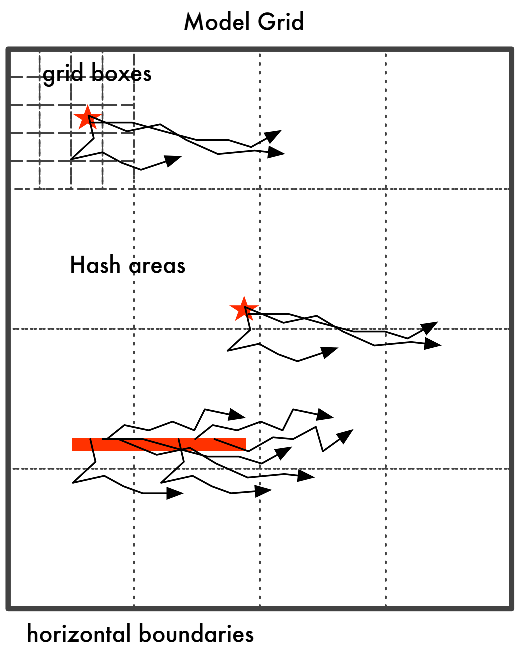

Figure 5: Area based parallelization with a two dimensional grid

3.4 Load balancing

The area based Parallelization needs the implementation of load balancing. If the model is running for some time the particles possible accumulate in one or more regions of the model grid. Because of the maximum count of particles possible in the grid model, there will be less particles in the rest of the grid model. If no particle leaves the grid model the maximum particle count will be reached at some point and the particles will be distributed along the hash areas. It is most likely that this distribution won't be uniform.

Figure 6 shows the model grid divided in eight areas, computed by two processing units, $P1$ and $P2$. Each processing unit has the same amount of areas containing the same amount of particles. At the moment the load is balanced between the both processing units. The amount of particles inside an area is indicated by the red tinted. Because of the uniform distribution of the particles both processing units will need the same amount of

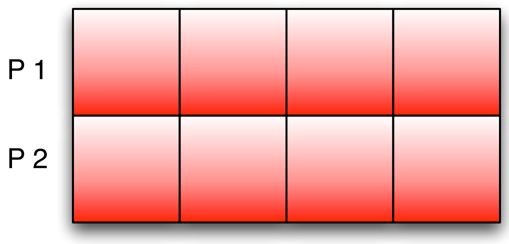

Figure 6: Two processing units sharing the same amount of particles.

time for the computation of the particle movement.

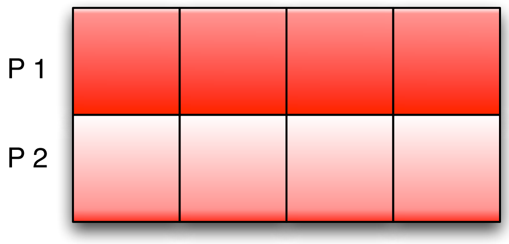

Figure 7: Two processing units sharing a different amount of particles.

Figure 7 shows an unbalanced model grid. The particles moved in the upper half of the model grid. Now $P1$ needs more time to compute all areas because $P1$ has to compute more particles. Now $P2$ has to wait each iteration for the communication of $P1$. This slows down the whole computation.

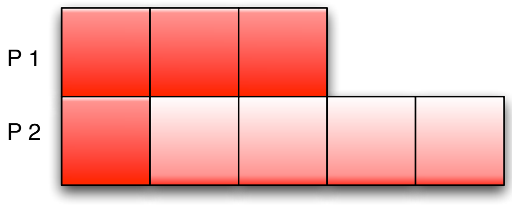

Figure 8: Rebalanced load by reassigning the areas to the processing units.

Figure 8 shows the area assignment after the load was rebalanced. The rebalancing works quit simple, a processing unit gets one area after another assigned until the particle inside the assigned areas is grater than the maximal particle count divided by the processing unit count. The costs and efficiency of this algorithms depends of the amount of areas used in relation of the model grid size. The more areas are used the better the load balancing work, but the more communication is needed for transferring the particles between the areas.

$$assigneParticleCount >= \frac{maxParticleCount}{processingUnitCount}$$

A possible check for an unbalanced load, is to compute this value every couple of iterations and check if the sum of the particles inside the areas varies to much of the desired value. If the variation is to high the load balancing is invoked. This shouldn't happen to often because it's a expensive operation.

3.5 Storing particles

The particles are simple structures, containing the position of the particle, stored inside a list. The list is a chuck of memory with a small header. The header contains the count of particles inside the list and the index of last used list place. This is needed because the list can fragmenting, this means the particle count can be lower than the index of the last used list place. The index of the last used list place is important so the loop over the particles don't run over the full allocated memory. There are some function defined to deal with the particle lists. Alongside the simple like add and remove particles to the list, there is a replace function. The replace function is used to minimize the fragmentation of the list. For example if two areas swapping particles the replace function can be used to place the particles of the other area in the old places of the particles which leaves the area.

Every function that operates on the particle list has a corresponding function that works with multiple contiguous particles at once. For example when the list is completely defragmentated, all particles that need to be added to the list can be appended at once.

3.6 OpenMP

The try to parallelize the model on a shared memory architecture with OpenMP was not successful. The performance increment was not measurable.

Listing 1: OpenMP parallelization with parallel area computation

```
FOR - outer time loop
        (update of the meteoroloical fields)

        #BEGINN OMP worksharing area
        WHILE - inner time loop
        (time integration)

                WHILE- area loop
                        emitting        particles

                        FOR- particle loop
                                move every particle

                        END particle loop
                END area loop

                WHILE- area loop
                        particle migration

                END area loop
        END innere-zeite-Schleife

        #END OMP worksharing area
END outer time loo
```

Listing 1 shows pseudo code of the OpenMP parallelization where every thread computes one area at a time. This solution doesn't scale because of unsynchronized memory access. A possible solution would be to divide the data structures further into smaller private chunks for every thread. This would require more communication between the threads when the particles are moved and the access to the input file would be a critical point.

Listing 2: OpenMP parallelization with parallel particle movement

```
FOR - outer time loop
        (update of the meteoroloical fields)

        WHILE - inner time loop
        (time integration)

                WHILE- area loop
                        emitting        particles

                        #OMP PARALLEL for
                        FOR- particle loop
                                move every particle

                        END particle loop
                        #END OMP PARALLEL FOR

                END area loop

                WHILE- area loop
                        particle migration

                END area loop
        END innere-zeite-Schleife

END outer time loo
```

Listing 2 shows pseudo code of OpenMP parallelization where every particle is moved in a single thread. The Problem with this approach is it's small granular parallelization. Because every thread is just doing a bit of work the whole program doesn't scale.

3.7 MPI

LaDis is parallelized only with MPI because of the poor performance improvement with OpenMP. The MPI parallelization generally works like illustrated in the section 3.3. At the beginning of the computation each MPI process gets the same amount of hash areas to compute. Every node only reads the data it needs to compute, the input data file gets opened multiple times. A further improvement would be to use a MPI implementation of the NetCDF data format to support a better I/O parallelization. Same counts for the output data of the model, every node file writes its own output file. The parallelization of the output data depends on the file system.

The MPI communication works in four mayor steps. First a process collect all the particles that needs to be send to another process after an iteration of particle movement. Than MPI_Isend is used to to send the particles to the process they are moved to. Next the process checks with MPI_Iprobe if it has to receive any particles of it's neighbour processes. If the process receive any particles MPI_Irecv is called to store the particles in the in the data structure of the area.

4 Test Runs

Here is the plotted output data two test runs. One run with the sequential implementation of LaDis and one of the parallel MPI implementation of LaDis executed on 4 processes.

4.1 Sequential Run

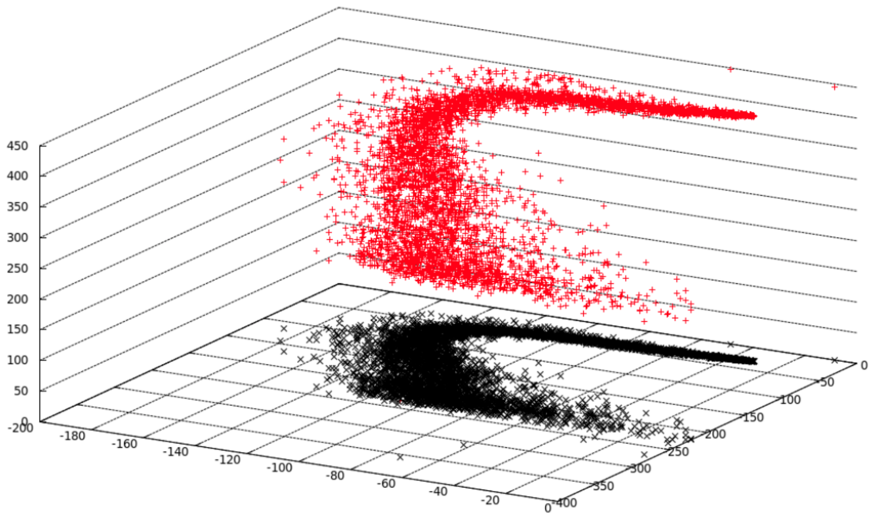

Figure 9: Particle diffusion computed with the sequential implementation of LaDis.

Figure 9 shows the plotted output data of a particle diffusion computed with the sequential version of LaDis. The red dots are the particles scatted in a three dimensional area. The black dots are the shadow the particles. The 3D-plot is projected into two dimensions on the X- and Z-axes to visualize the horizontal movement of the particles.

4.2 Parallel Run

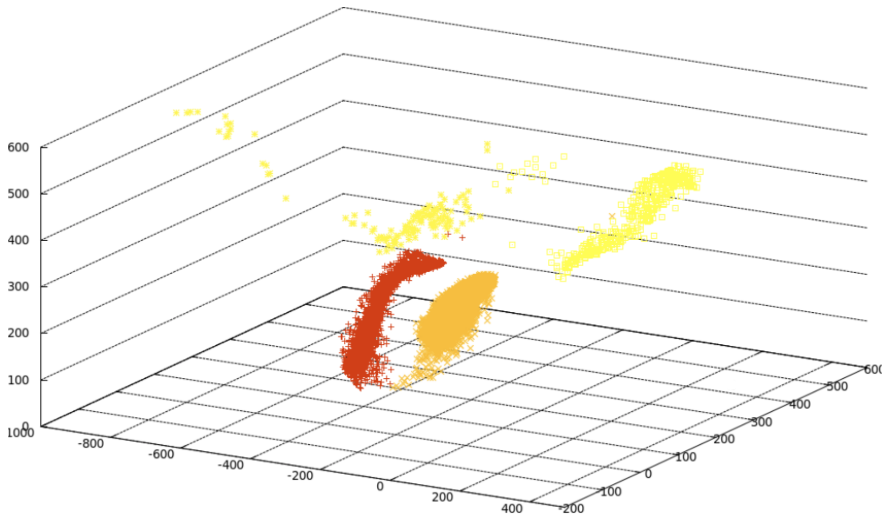

Figure 10: Particle diffusion computed with the parallel implementation of LaDis.

Figure 10 shows the same particle diffusion computed parallel with four processes. Again each data point is one particle in a three dimensional area. The different colours indicate which processes is computing which particle. This plot visualises some problems which the parallel model still has. One of the problems is the communication between the processes. There is a gap with basically no particles between the different processes. Further more some particles get ripped out of the stream, like in the upper left of the plot. One reason for the gap between the areas of the different processes could be a misplacement of the areas inside the model grid. If there is a gap between the processes that none process is handling the particles which float in these gaps get simply lost because no process is storing them.

In addition this test had a super linear speed up, most likely because of the small count of particles used. If bigger number of particles are used the program crashes because of the computational errors that still exists.

5 State of Development

The implementation of the software is lacking of some of the planed features and has some computational errors inside the parallel implementation.

5.1 What is working

The sequential computation of LaDis is working. To read the meteorological data LaDis supports NetCDF files. The data gets correctly distributed in the sequential implementation different areas . The test run which Figure 9 shows, uses multiple areas to divide the model grid. The data structure is implemented and provides an interface to handle the particle storing. A namelist reader is working to read the parameter and pass them to the program. Furthermore LaDis writes potable output data to visualise the finished computation of the particle diffusion.

5.2 What is not working

The parallel model with MPI is generally working. There are some problems with the communication between the processes which resolve in some computational errors. It looks like the processes are misplaced inside the model grid. If particles are migrated between two processes some of them are lost. Figure **??** shows a gap between the processes computing the model grid. At this state the load balancing is not implemented, as the particles move through the model grid the areas doesn't get reassigned to different processes if the load balance change. The data structure is missing the possibility to be defragmented, only the replace function can prevent a to strong fragmentation. At least one more improvement would be usage of MPI I/O to speed up the file read and writes.