Vlasiator AMR Documentation

Tuomas Koskela - June 2019

Introduction

The investigation of a number of physics topics with Vlasiator requires capability for 6D simulations. In the past, 6D runs have been unfeasible due to memory and CPU time requirements even though the solvers in Vlasiator would theoretically be capable of them. In the PRESTISSIMO project, one of the deliverables is to include AMR capability to Vlasiator in order to enable large-scale 6D simulations.

The plan for 6D capability was prepared with Sebastian von Alfthan, Urs Ganse, Yann Pfau-Kempf and Markus Battarbee during April/May 2018. The main points were

- Implement static AMR in real space
    - DCCRG scales badly beyond O(100) MPI processes due to global cell data structure. Therefore adapting the refinement dynamically at runtime is not feasible. We can use a priori information of regions of interest to create a refined mesh during initialization and keep it static during runtime.
    - Velocity space AMR implementation is more complex.
- Update vlasov solver to update cells in 1d pencils.
    - Real space update is a translation, each dimension can be treated separately and summed at the end
    - Pencils can be updated in-place, relieving some memory requirements.

Pencil Building Algorithm

To update cells in a refined real space mesh, we need to find one-dimensional sets of local spatial cells, “pencils”, whose width is equal to the width of the cell with highest refinement in the set. An example is shown in Figure 1. The final set of pencils should fulfill the following criteria

1. Every cell in the local domain is included in at least one pencil
2. No cell is included more than 2^{R*R} times, where R is the maximum refinement level of the mesh.

The second criterion follows from whenever a refined cell is encountered, the pencil must be split to match the width of the refined cells. Because we only allow the refinement level to increase by one between two neighboring cells, the pencil will always be split into exactly four sub-pencils at each new refinement level. Since we do not know how many refinement levels will be included in a pencil beginning at an arbitrary cell, we use a recursive algorithm to build the pencils and split them up as necessary. The algorithm is described in Table 1 and the code can be found here. For an example illustration of how the algorithm proceeds, see this presentation.

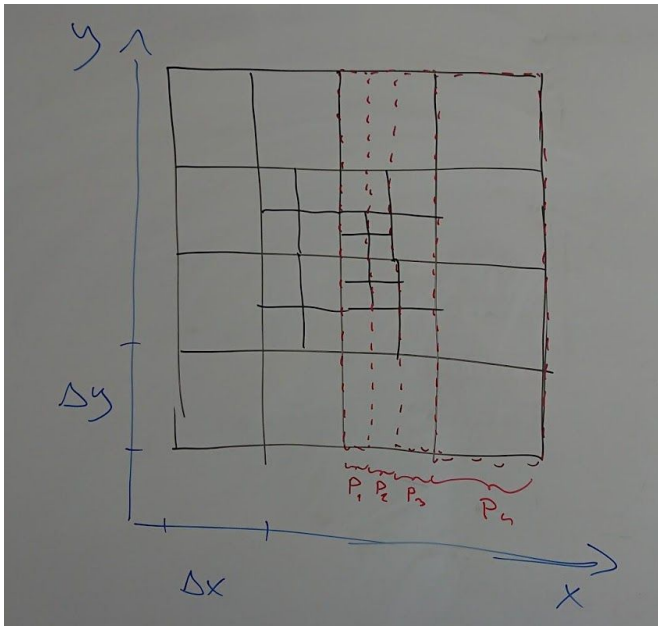

Figure 1. Pencil example. Pencils are shown in red dashed lines. The width of each pencil is equal to the smallest cell under the area of the pencil. $P_4 = \Delta x$, $P_3 = \Delta x/2$, $P_1 = P_2 = \Delta x/4$

Table 1. Pencil building algorithm

1. Start at an arbitrary seed cell Id with an initial list of cells (that can be empty)
2. Get the nearest neighbor cell in the positive direction
3. If the neighbor is an invalid cell, or a cell that has been specified as an end condition, return the list of cells and exit.
4. If the neighbor has a higher refinement level than what is stored in the list of cells, select one of the refined neighbors and add it to the list. For the remaining neighbors, spawn new builders by calling the function recursively.
5. If the neighbor does not have a higher refinement level, add it to the list and loop back to step 2.

To find the seed ids in the local domain we use a separate function. It gets the nearest neighbor of each local cell in the negative direction and marks any cell whose neighbor fulfills any one of the following three conditions as a seed cell for pencils

1. The neighbor is non-local
2. The neighbor is across a periodic boundary
3. The neighbor is a boundary cell in system boundary layer 1.

In the original implementation, that tries to maximise the size of the pencils and minimize the number of pencils, the seed ids are passed as pencil end conditions, ie. pencils are not allowed to overlap. However, since numerical errors were found in this approach, the current code passes all local cells as end conditions, minimizing the size of pencils and maximising the number. This was found to alleviate the numerical issues, see Outstanding Issues for more detail.

After the pencils for the local cells have been built, a check is made on the remote ghost cells within the vlasov stencil width of the cells of each pencil. If any ghost cells exist with a smaller size than the width of the pencil, the pencil is split further into narrower copies of itself until their width matches the size of the ghost cells. This is done because the vlasov translation has to read data from the remote cells and must know which refined ghost cell to read.
The pencils are stored in a data structure with the following fields

- uint N: Number of pencils in the set
- uint sumOfLengths: Sum of lengths of all pencils
- vector< uint > lengthOfPencils: Lengths of pencils
- vector< CellID > ids: List of cells in pencils
- vector< Realv > x,y: x,y - position of the pencil (dimensions perpendicular to the translation direction)
- vector< bool > periodic: Periodicity of the pencil (if periodic boundaries are used)
- vector< std::vector<uint> > path: Path taken in refined cells, each entry has a value [0,...,3] representing which refined neighbor was selected for this pencil for each refinement level in the pencil.

Vlasov Translation with Pencils

The routine to map velocity blocks in all local cells forward by one time step in one spatial dimension is an adaptation of the non-AMR version. The AMR version uses the pencil approach, which introduces a few differences. The algorithm is described in Table 2. The version of the solver is chosen at run time based on the maximum allowed refinement level of the grid. If the maximum refinement level is 0, the non-AMR version is used, otherwise the AMR version with pencils is used.

Table 2 Pencil translation algorithm

1. Fiddle indices x,y,z.
2. Compute pencils.
3. Compute unionOfBlocks. (same as non-AMR)
4. Loop over unionOfBlocks: (guided OpenMP loop)
    a. loop over pencil sets:
        i. Allocate target data for all pencils.
        ii. Loop over pencils:
            1. Allocate vectorized source data buffer for pencil.
            2. Load data from spatial cells to source data.
            3. Propagate all cells in pencil.
            4. Write data from source data to target data.
            5. Deallocate source data.
        iii. Reset blocks in all spatial cells.
        iv. Loop over pencils:
            1. Aggregate target data for blockid to original spatial cell.
        v. Deallocate target data.

In the first loop over pencils in step ii of Table 2, the velocity blocks in spatial cells are propagated in step 3. The propagation function loops over all spatial cells in the pencil and propagates all velocity cells in the velocity block ID, given by the iteration of loop 4 in Table 2. The function mainly follows what is done in the non-AMR version, with the difference of using a non-uniform grid calculation for the PPM coefficients. At the moment the AMR version is restricted to PPM coefficients only, as they were readily available in the paper by White et al. (2008) Non-uniform PQM coefficients can also be derived. The PPM coefficients are calculated from cell face estimates given by

$$u_{j-\frac{1}{2}} = \frac{1}{h_0+h_1+h_2+h_3} \times \left\{ \frac{(h_0+h_1)(h_2+h_3)}{(h_1+h_2)} (\bar{u}_1 h_2 + \bar{u}_2 h_1) \left(\frac{1}{h_0+h_1+h_2} + \frac{1}{h_1+h_2+h_3} \right) \right.$$
$$\left. + \frac{h_2(h_2+h_3)}{(h_0+h_1+h_2)(h_0+h_1)} [\bar{u}_1(h_0+2h_1) - \bar{u}_0 h_1] + \frac{h_1(h_0+h_1)}{(h_1+h_2+h_3)(h_2+h_3)} [\bar{u}_2(2h_2+h_3) - \bar{u}_3 h_2] \right\}$$

where $h_{0,\ldots3}$ are cell widths and $u_{0,\ldots3}$ are cell center values.
In the second loop over pencils in step iv, of Table 2, the data written into the cell from the pencil is scaled down by the ratio of the cross-section of the spatial cell to the cross-section of the pencil. This compensates for multiple pencils writing data into the same cell when pencils have been split to match the width of refined cells.

Ghost Cell Update

After each translation step, the ghost cells are updated with the data that has been mapped to remote neighbors by the local cells. The update is done in two steps, with the positive and negative neighborhoods updated separately.
In the non-AMR version the spatialCell data structure contains a single field of neighbor_block_data, a copy of the vspace cells that were mapped to a neighbor cell during the vlasov translation step. Since we don't allow the mapping to exceed one cell per time step, and neighbor updates are done separately for each dimension-direction pair, only one neighbor cell was ever needed.
The neighbor block data has been extended in the AMR version to hold an array of 8 neighbor vspaces. The number is defined in the variable MAX_NEIGHBORS_PER_DIM. The number is 8 instead of 4 because DCCRG defines neighborhoods relative to the size of a cell, which means a cell of coarser refinement will have 8 refined neighbors, even in a neighborhood of distance 1. This is a known issue and it has been discussed in the dccrg repository, with an update to address it planned during 2019. To circumvent issues arising from non-face neighbors getting included in size 1 neighborhoods, the ranks holding the true face neighbors of each cell is stored separately in the spatialCell data structure. The face neighbor ranks should be updated after each load balance, and are used during get_mpi_datatype() to determine which ranks neighbor block data is sent to.
In update_remote_mapping_contribution_amr the neighbor data arrays are allocated in such a way that the sending and receiving ranks allocate the same amount of memory. Any discrepancy will lead to MPI errors. Since the communication is handled by DCCRG and the same data has to be sent to every neighbor (except for the modifications described in the

previous paragraph), the basic idea is that on a coarse-refined boundary, each rank holding refined cells on the process boundary must receive the data of all the refined neighbors, and figure out which element(s) in the neighbor block data array to read. In the code, this is done by calculating sendIndex and receiveIndex, using the order of siblings in the refined cells. The sending rank will only write to sendIndex elements of the neighbor block array, and likewise the receiving rank will only read from receiveIndex elements.

Outstanding Issues

The major outstanding issue is numerical errors that appear in the pencil solver. The errors become visible when the pencil configuration is adjusted by any mechanism and the result is compared with a different pencil configuration. The following is known:

- When there are no refined cells in the mesh, the errors are 0
- When we force all pencils to be of size 1 cell, the errors become small (in the order of floating point accuracy)
    - Errors are smaller if we disable field solver and acceleration
    - Field solver alone does not have an effect
- No significant difference between 1 and 2 levels of refinement
- If we remove testpackage flags, the errors become larger
    - But still exactly 0 if no cells are refined and pencils have size 1
- If we let pencils have size greater than 1, the errors also become larger.

My hypothesis is that there are (at least) two sources of error: One that causes small errors with size 1 pencils, possibly order of summation or such, and another that causes larger errors with size greater than 1 pencils.

List of other outstanding issues/development ideas includes

- Interpolation when mapping between fsgrid and coarse dccrg cells. At the moment fsgrid cells will copy the value of the field solver source terms from the dccrg cell they map to. When dccrg cells are coarse, multiple fsgrid cells will copy the same value, and there will be a sudden jump at the coarse cell interface. This can cause problems for the field solver and should be replaced with some sort of interpolation, possibly the same PPM fits that are used for the velocity cell data.
- Investigate deeper refinement levels and fsgrid refinement. At the moment we have limited ourselves to two levels of refinement (three different cell sizes). In theory there is no reason not to use more refinement levels, however, the size of fsgrid grows as it always has the highest refinement everywhere. That will have to be changed eventually if we want to go to highly refined meshes.
- Once DCCRG supports nearest only neighborhoods, tidy up the ghost cell update.
- Derive and implement PLM and PQM coefficients for non-uniform mesh.
- Optimize the PPM calculations by reducing the number of divides
- Interpolate between pencils in x,y - dimensions, possibly using the same polynomial fitting functions that are used for the z - dimension.
- Investigate temporal sub cycling

- Replace DCCRG grid with better AMR-supported library, move towards dynamic criteria based refinement.
- Velocity space AMR
- TBA