A Local Adaptive Algorithm for Solving the Vlasov Equation*

Olivier Hoenen Éric Violard

ICPS - LSIIT (CNRS UMR-7005)
Université Louis Pasteur, Strasbourg
{hoenen,violard}@icps.u-strasbg.fr

Abstract

Solving the Vlasov equation with a high accuracy and a minimum computational cost represents a great challenge. Adaptive methods based on a sparse mesh are a classical answer. In this paper, we propose a new adaptive method with a minimum overhead due to sparse mesh management.

Keywords. Adaptive Numerical Method, Vlasov Equation, Algorithmic Transformation

1 Introduction

The Vlasov equation models the evolution in time of charged particles under the effect of an electric field. Its solving is of great importance for the simulation of phenomena in plasma physics, such as nuclear fusion. Recent and interesting methods for solving the Vlasov equation are based on the *semi-Lagrangian* scheme [8, 1, 2]. This scheme uses a useful property of the equation, saying that the solution $f((x, v), t)$, which is called the distribution function, is constant along characteristics. The characteristics of the Vlasov equation model the particles movement within the physical domain (x, v) (position and velocity space) called *phase space*. This property is used to compute the value of f on a grid of the phase space. The total amount of values to compute increase drastically along with the dimension and the domain on such a grid. So it becomes interesting to use adaptive algorithm based on a sparse grid that evolves during simulation.

We propose here a full evolution of the YODA solver [6] we have developped. The principle for computing values is the same as YODA, but

*This research was supported by a grant from Région Alsace

mesh adaptation algorithm is totally new, allowing to reduce drastically the number of elements to insert or remove in the data structure. In the first part of this report, we present our numerical resolution shceme based on a classical semi-Lagrangian principle and a new adaptation algorithm.

2 Numerical scheme

By discretizing time, the conservation property says that $f((x,v),t^{n+1}) = f(\mathcal{A}^n(x,v),t^n)$ where $\mathcal{A}^n$ denotes the discretized characteristics called *backward advection operator* which depends on the electric field. Thus, given any phase space mesh, say $\mathcal{M}^n$, denoting f^n an approximate of the solution at time t^n and assuming $f^n(a)$ is known at every node a of $\mathcal{M}^n$, the *semi-Lagrangian* scheme to compute f^{n+1} at any node a of $\mathcal{M}^{n+1}$ is the following : compute the backward advected point $\tilde{a} = \mathcal{A}^n(a)$, compute an approximate of $f^n(\tilde{a})$ by interpolation using the values of f^n at nodes of $\mathcal{M}^n$ and last identify $f^n(\tilde{a})$ as $f^{n+1}(a)$ by the conservation property.

This scheme defines an uniform method by taking $\mathcal{M}^n$ for a regular grid. However, the computational cost which mainly depends on the amount of interpolations can be drastically reduce by taking $\mathcal{M}^n$ for an *adaptive mesh*. This is particularly true when the phase space has a high dimensionality and for real cases, i.e., when $(x,v) \in \mathbb{R}^3 \times \mathbb{R}^3$. Notice that, for sake of simplicity, the method proposed in this paper is given for $(x,v) \in \mathbb{R} \times \mathbb{R}$ but can be generalised to higher dimensions. Moreover, we chose to use a local interpolator in order to lower the computational cost by inherently exploiting data locality.

In order to obtain an adaptive method, a mesh topology and a mesh adaptation procedure have to be defined.

2.1 Mesh topology

Our method is based on a *dyadic* structured adaptive mesh. A dyadic mesh is a partition of the unit square $[0,1] \times [0,1]$ where each cell identifies a square $[i_1 2^{-j}, (i_1+1)2^{-j}] \times [i_2 2^{-j}, (i_2+1)2^{-j}]$, where $(i_1, i_2, j) \in \mathbb{N}$. The integer j in the previous definition defines the cell size and identifies its *level* in an implicit *hierarchy* of cells. As example, let us consider a cell, say α, of level j, then it covers the area of four cells of level $j+1$. The four cells are called the *daughters* of cell α and cell α is called the *mother* of the four cells. Moreover, we sometimes called *sister cells* some cells having the same mother. These relationships identify a quad-tree hierarchy. The cells at higher levels in the hierarchy have smaller size. In the rest, we consider only mesh with a fixed highest level, denoted J, and a fixed lowest level, denoted j_0. We have thus $j_0 \leq j \leq J$. Considering the levels of detail of the adaptive mesh, level j_0 is the *coarsest level* and level J is the *finest level*. Nodes of the mesh are

located at the centre, at the edge of each cell, and at the middle of each side. Therefore each cell has nine equally spaced nodes.

The solution f is represented locally by the values at nodes of each cell. More precisely, the approximate value of $f((x,v),t^n)$ is the value obtained by interpolation using the values at every nodes of the cell of mesh $\mathcal{M}^n$ where point (x,v) is located.

2.2 Mesh adaptation

Our adaptation procedure falls into two parts: mesh prediction and compression. Mesh prediction builds a fresh mesh from an old one by using the backward advection operator and mesh compression deletes cells considered useless as the values at their nodes could be deleted without loss of accuracy. These two parts are detailed here below.

Mesh prediction builds $\mathcal{M}^{n+1}$ from $\mathcal{M}^n$ putting the computational effort into zones of interest (high gradient areas) and neglecting the other ones. The procedure is the following: starting with a coarse uniform mesh (made of cells of level j_0), each cell verifying the *refinement criterium* is replaced with its four daughters repeatedly until no such cell exists. A cell, say α, verifies the *refinement criterium* if and only if its level is lower or equal than the level of β, where β is the unique cell of $\mathcal{M}^n$ which contains the backward advected of c, i.e., $\mathcal{A}^n(c)$ and c is the centre of α.

Mesh compression is intended to obtain a coarser version of $\mathcal{M}^{n+1}$ reducing the amount of cells within low gradient areas. It proceeds as follows: each group of four daughters verifying the *coarsening criterium* is replaced with their mother, repeatedly until no such group exists. A group verifies the *coarsening criterium* if and only if the difference between the local representation of f with the mother values and the representation with the values of the cells in the group is under a given threshold. If both representations are similar, only the mother needs to be conserved.

2.3 Related works

Previous works [3, 6] used the same semi-Lagrangian scheme and mesh topology but another adaptation procedure. Instead of using the backward advection operator, prediction of $\mathcal{M}^{n+1}$ from $\mathcal{M}^n$ used its converse, i.e., the forward advection operator. A great drawback to using this, so called forward, adaptive scheme, is that *mesh consistency* have to be maintained each times a new cell is added to $\mathcal{M}^{n+1}$ which entails an important computational overhead.

In other works [4, 5] adaptivity is performed by using a wavelet framework. Prediction step aimed to build a list of points from the backward adevction of the previously computed wavelet coefficients. It is a straight

forward method but the wavelet coefficients needed to interpolate the values are less local than with our method.

In the algorithm presented here, we have kept the interpolation operator of the first works to preserve data locality. We have developped a predicted mesh construction where mesh consistency is implicitly verified at no cost.

3 The recursive algorithms

Here we give a first algorithm directly derived from the numerical scheme previously presented. In this algorithm, the representation of the solution at time t^n is identified by a pair $(\mathcal{M}^n, F^n)$, where $\mathcal{M}^n$ is a set of cells and F^n is a set of pairs (node,value).

This algorithm falls into three successive phases for each time step: mesh prediction, node evaluation, i.e., computation of the values at nodes and mesh compression.

Our algorithm for predicting the cells of mesh $\mathcal{M}^{n+1}$ (algorithm 1) starts with an empty set and applies to every coarse cell (of level j_0). It checks whether the cell verifies the refinement criterium. In that case, the algorithm repeats itself for every daughter. Else, the cell is added to the predicted mesh and the algorithm terminates. The algorithm is thus naturally recursive and performs a depth-first scan of the cell hierarchy. Its complexity is $\mathcal{O}(\mathcal{T})$, where $\mathcal{T}$ is the number of nodes of the cell hierarchy tree.

Algorithm 1: Mesh prediction

Input: $\mathcal{M}^n$
Output: $\mathcal{M}^{n+1}$

```
1  predict(α) ::
2  begin
3      c := centre(α)
4      set β to the unique cell of M^n such as A^n(c) ∈ β
5      if min(J−1, |β|) ≥ |α| then
6          for each daughter α_{k∈0..3} of α do
7              predict(α_k)
8      else
9          add α to M^{n+1}
10 end
```

Node evaluation applies to every node of $\mathcal{M}^{n+1}$ (algorithm 2). Our algorithm scans these nodes by using a nested loop. The outer loop scans the cells and the inner one scans the nodes of a cell. Each node is treated only once thanks to the test $a \notin F^{n+1}$. It is assumed that this test is not costly. This is an important assumption because the ratio of nodes

belonging to several cells is high. The complexity of this algorithm is thus $\mathcal{O}(\mathcal{N})$, where $\mathcal{N}$ is the number of nodes of the mesh.

Algorithm 2: Node evaluation

Input: $\mathcal{M}^n, \mathcal{M}^{n+1}, F^n$
Output: $\mathcal{M}^{n+1}, F^{n+1}$

```
1 begin
2     for each cell α of M^{n+1} do
3         for each node a of α (a ∉ F^{n+1}) do
4             set β to the unique cell of M^n such as A^n(a) ∈ β
5             v := interpolate(A^n(a), β)
6             add the pair (a, v) to F^{n+1}
7 end
```

Our algorithm for compressing the cells of mesh $\mathcal{M}^{n+1}$ (algorithm 3) applies to every coarse cell (of level j_0). If the cell belongs to the mesh, the algorithm terminates. Else the algorithm is first applied to each daughter and then it checks whether every daughter belong to the mesh and this group of daughters verify the coarsening criterium. In this case, this group is replaced with the mother in $\mathcal{M}^{n+1}$ and the algorithm terminates. Like mesh prediction, the algorithm is naturally recursive and performs a depth-first scan of the cell hierarchy. Its complexity is also $\mathcal{O}(\mathcal{T})$.

Algorithm 3: Mesh compression

Input: $\mathcal{M}^{n+1}, F^{n+1}$
Output: $\mathcal{M}^{n+1}$

```
1  compress(α) ::
2  begin
3      if α ∉ M^{n+1} then
4          b := true
5          for each daughter α_{k∈0..3} of α do
6              compress(α_k)
7              b := b ∧ (α_k ∈ M^{n+1})
8          if b ∧ compression-test(α) then
9              add α and remove its 4 daughters from M^{n+1}
10 end
```

4 Recursion removal

Recursive algorithms are often more elegant than iterative algorithms, but most of the time their implementation lacks of efficiency. To obtain good performances, we have thus to eliminate recursion from our algorithms, but preserve the depth-first traversal order that enables to obtain the best computational complexity.

For the algorithm 1, elimination of recursion is achieved in a very classical way, by using a stack of cells saving the recursion context. This is shown by algorithm 4.

Algorithm 4: Mesh prediction (iterative version)

```
   Input: M^n
   Output: M^{n+1}
 1 begin
 2     set P to a new stack
 3     push α to P
 4     while ¬ empty(P) do
 5         pop α from P
 6         c := centre(α)
 7         set β to the unique cell of M^n such as A^n(c) ∈ β
 8         if min(J−1, |β|) ≥ |α| then
 9             push each α's daughter to P
10         else
11             add α to M^{n+1}
12 end
```

In algorithm 3, the recursion is only used to perform a depth-first scan of existing cells. It can thus be removed provided the cells of $\mathcal{M}^{n+1}$ can be accessed in depth-first order in the data structure into which they are stored. For example, if we use a stack of cells, say $\mathcal{P}$, as for prediction, and assume the cells are arranged in depth-first order in this stack, then we can eliminate recursion from algorithm 3 and obtain algorithm 5.

Algorithm 5: Mesh compression (iterative version)

Input: $\mathcal{M}^{n+1}, F^{n+1}, \mathcal{P}$
Output: $\mathcal{M}^{n+1}$

```
1  begin
2      k[j_0..J] := 0
3      b[j_0..J] := true
4      while ¬ empty(P) do
5          pop α from P
6          j := |α|
7          k[j] := k[j] + 1
8          while k[j] = 4 ∧ j > j_0 do
9              k[j] := 0
10             α := mother(α)
11             if b[j] ∧ compression-test(α) then
12                 add α and remove its 4 daughters from M^{n+1}
13             else
14                 b[j − 1] := false
15             b[j] := true
16             j := j − 1
17             k[j] := k[j] + 1
18 end
```

In order to restore the context of each recursive call, the successive values of each variable b and k in algorithm 3 are stored in an array. The array stores the values at each level of recursion. As the recursive algorithm scans cells in depth-first order, each level of recursion identifies a level of cell between j_0 and J. Therefore, the array range of indices is $[j_0..J]$. The values are updated according to algorithm 3. The values of k is incremented each times a new cell of level j is read. When $k[j]$ reaches value 4, it means that a group of 4 cells of level j is to be considered for compression and $k[j]$ is set again to zero for the next group at the same level. The value of $b[j]$ is updated (set to $false$) as soon as one compression test fails at level $j+1$ (it corresponds to the result of the logical and in algorithm 3 and set again to $true$ before a new group of cells of level j is to be considered for compression (at the beginning of the recursive algorithm).

As said previously, our algorithm for going forward one time step consists in performing three phases successively. It entails three traversal of the cell hierarchy. An interesting optimization is thus to perform only one traversal of the cell hierarchy by applying a part of the three phases on each cell. This can be achieved since cells are traversed in the same order in prediction and in compression phase and the node evaluation is not driven by a particular

order. The algorithm 6 described this last transformation.

Algorithm 6: One time step (only one traversal of the mesh hierarchy)

Input: $\mathcal{M}^n, F^n$
Output: $\mathcal{M}^{n+1}, F^{n+1}$

```
1  begin
2      k[j_0..J] := 0
3      b[j_0..J] := true
4      for each cell α of level j_0 do
5          push α to P
6          while ¬empty(P) do
7              pop α from P
8              j := |α|
9              c := centre(α)
10             set β to the unique cell of M^n such as A^n(c) ∈ β
11             if min(J − 1, |β|) ≥ j then
12                 push each of α's daughter to P
13             else
14                 add α to M^{n+1}
15                 for each node a of α (a ∉ F^{n+1}) do
16                     set β to the unique cell of M^n such as A^n(a) ∈ β
17                     v := interpolate(A^n(a), β)
18                     add the pair (a, v) to F^{n+1}
19                 k[j] := k[j] + 1
20                 while k[j] = 4 ∧ j > j_0 do
21                     k[j] := 0
22                     α := mother(α)
23                     if b[j] ∧ compression-test(α) then
24                         add α and remove its 4 daughters from M^{n+1}
25                     else
26                         b[j − 1] := false
27                     b[j] := true
28                     j := j − 1
29                     k[j] := k[j] + 1
30 end
```

5 Data structure

The data structure used to represent the adaptive mesh and store values (i.e., to represent $\mathcal{M}^n$ and $\mathcal{F}^n$) is of great importance in the efficient implementation of an adaptive method. Data should be as local as possible which leads to a treelike representation of the information. Moreover, the actual

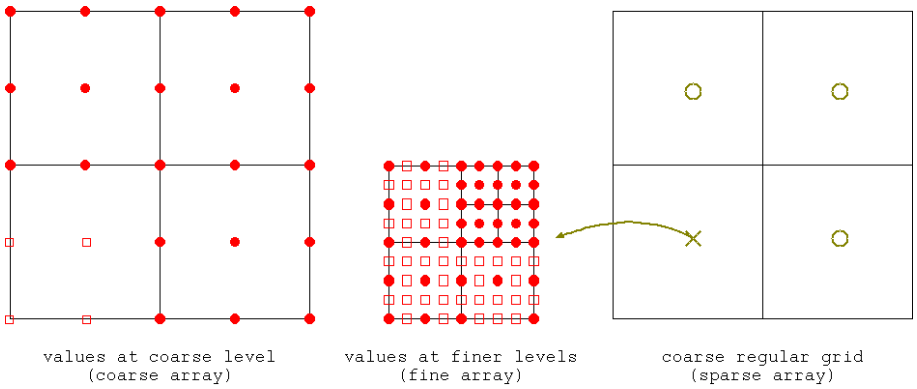

Figure 1: Data organization

numerical method derived from the semi-Lagrangian scheme exhibits random accesses determined by the backward advection operator (that depends itself from the electric field): the value of f^{n+1} at (x, v) is the value of f^n interpolated at $\mathcal{A}^n(x, v)$. Thus the values should be organized by location in phase space to provide fast random access given a point of the space. Therefore, as in [7], we use arrays to store values.

The values at nodes of a coarse cell are stored in one single 2D array, so called *coarse array*, and for any coarse cell α which does not belong to the mesh, all the values at nodes of descendant cells (whose ancestor is α) are stored in their own 2D array, so called *fine array*. Each element of these 2D arrays either stores an approximate of the solution, or stores a special constant which means that no value is present at this node. Therefore the test $a \notin F^n$ is efficiently implemented by just comparing the element with the constant. Notice that this is an important reduction of memory usage compared with using an unique 2D array. However, since some nodes are shared between coarse and finer cells, it may happen that some values are stored and computed several times in our data structure.

In addition to these arrays of values, another array, so called *sparse array*, represents an approximate of the treelike structure of the adaptive mesh. Elements of this array are in one-to-one corespondance with cells of the uniform grid of level j_0. Each element either is the null pointer identifying the presence of a coarse cell, or a pointer to an array of values identifying the presence of some finer cells. Hence, getting access to a value at an existing node requires only two memory address reads: one for reading the right pointer within the coarse array, and one other for accessing to the right element in the right 2D array (coarse or fine as the pointer is null or not). Fig.1 illustrates our data organization.

Among the operations whose complexity depends on the data structure, one occurs a number of times about the number of nodes and therefore

its complexity has to be inspected carefully. This operation is searching the unique cell which contains a given point of the phase space. It can be performed in various ways:

- Linear search. Checks all the cells which contains the point by scanning levels from J to j_0 until the center node has a value or else from j_0 to J until the center node has no value. The linear search thus runs in $\mathcal{O}(J - j_0 + 1)$.
- Binary search. Find the median level and check the cell of this level. If no value exists at center node (it means no finer cells exist), then search strictly coarser cells in same manner, else (it means no coaser cells exist in the mesh) search finer cells until only one level remains. The binary search thus runs in $\mathcal{O}(\log_2(J - j_0 + 1) + 1)$.
- Storage of level. This process uses an additional 2D array for each fine array. Its elements are in one-to-one correspondance with the cells of the fine uniform grid. This array stores, for each fine cells, say α, the level of the mesh cell which contains α. Hence, getting the level of the searched cell requires only one read in this array. This process thus runs in $\mathcal{O}(1)$. But drawback is that any time a cell is added to the mesh, the additional array has to be updated. The update complexity is of order of the cell surface, i.e., $\mathcal{O}((J - j + 1)^2)$, where j is the cell level.

Therefore some performance measurements have to be performed in order to determine the best process in any case. The experimental results and conclusions are reported in section 6.

6 Experimental results

The algorithm presented in this paper had been implemented in C with special care on structures for recursion removal. For example, the stack of states ($\mathcal{P}$), used in our iterative algorithm (see section 4), can be advantageously represented by an array whose indices range from j_0 to J and identify cell levels. Each element of this array is an array itself, storing at most four cells of the same level. This small structure is sufficient to describe all states of the recursive traversal because we have to consider at most 4 sister cells for each level at a same time.

For the data structure presented in section 5, we implemented all three searching algorithms. Then we made some measurements in order to compare their performances in a real simulation of a semi-gaussian beam under the effect of a uniform electric field. The simulation last for 500 iterations of the algorithm, with at most 2048 points per dimension ($J = 10$). The

program runs on a linux PC with an Athlon64 processor at 2.4GHz with 1024KB of cache and a 4GB RAM. The table 6 gives execution time for this simulation. It shows that the linear search algorithm, which is the most simple of the three algorithms, is also the fastest whatever the size of fine arrays is used.

coarse level (j_0)	3	4	5	6	7	8
linear	407	353	335	342	376	495
binary	438	387	369	374	402	529
level storage	409	356	344	346	387	510

Table 1: Execution time (s) with different searching algorithms

coarse level (j_0)	3	4	5	6	7	8
linear	1.50	1.54	1.50	1.49	1.40	1.21
binary	3	3	2.90	2.85	2	2

Table 2: Number of access to the data structure

Results exposed in table 6 are rather unexpected. The level storage algorithm is slower than the linear search, that can be explained by the writing overhead introduced which is more important than the reduction of the number of accesses to one. But we could expect that binary search will also reduce the number of accesses for a very low cost overhead of the median level computation. But table 6 shows that for this realistic simulation, binary search accesses more times to data structure than the simple linear search. First of all, the binary search has a serious drawback because it always need an extra access to confirm the result. It implies that the minimum number of access with the binary search is 2 whereas linear search needs only 1 access in the best case. And as there is more fine level elements than coarse one (simply because of their size), there is more favorable cases for the linear search than the binary search. If a case is more advantageous for binary than linear search, we could almost say that the chosen finest level J should be coarsened.

7 Conclusion and futur works

We have presented a new adaptation framework for solving Vlasov equation. Our method is based on a recursive algorithm that allows a by-block processing of a time step. This locality reduces the accesses to adaptive mesh which are the main cause of overhead in this adaptive method. This study has

been defined for a 2 dimensional domain. It has to be extended for higher dimensions.

Moreover, for a future parallelization, the block processing is advantageous because it will minimize synchronization barriers that restrict speedup of code. Recursion removal and a study on how to search for mesh elements in the sparse structure make the sequential code more efficient. Now we have to parallelize this solver in order to run simulations in higher dimensions. We have to give a particular attention to load balancing and mesh distribution. The inherent recursive construction of the mesh and distant access to elements make this load balancing a main convern to achieve a good scalability.

References

[1] N. Besse. Convergence of a semi-lagrangian scheme for the one-dimensional vlasov-poisson system. *SIAM Journal on Numerical Analysis*, 42(1):350–382, 2004.

[2] N. Besse and E. Sonnendrücker. Semi-lagrangian schemes for the Vlasov equation on an unstructured mesh of phase space. *J. Comput. Phys.*, 191:341–376, 2003.

[3] M. Campos-Pinto and M. Mehrenberger. Adaptive numerical resolution of the Vlasov equation. In *Numerical Methods for Hyperbolic and Kinetic Problems, CEMRACS'03*, 2003.

[4] M. Gutnic, M. Haefele, I. Paun, and E. Sonnendrücker. Vlasov simulations on an adaptive phase-space grid. *Computer Physics Communications*, 164:214–219, dec 2004.

[5] M. Haefele, G. Latu, and M. Gutnic. A parallel vlasov solver using a wavelet based adaptive mesh refinement. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 181–188, Washington, DC, USA, 2005. IEEE Computer Society.

[6] O. Hoenen, M. Mehrenberger, and E. Violard. Parallelization of an adaptive vlasov solver. In *11th European PVM/MPI Users' Group Conference (EuroPVM/MPI '04), ParSim Session*, volume 3241 of *LNCS*, pages 430–435. Springer-Verlag, September 2004.

[7] O. Hoenen and E. Violard. An efficient data structure for an adaptive vlasov solver. Technical Report 06-02, University Louis Pasteur, LSIIT–ICPS, February 2006.

[8] E. Sonnendrücker, J. Roche, P. Bertand, and A. Ghizzo. The semi-lagrangian method for the numerical resolution of Vlasov equations. *J. Comput. Phys.*, 149:201–220, 1999.