An Efficient Data Structure for an Adaptive Vlasov Solver

Olivier Hoenen and Éric Violard
ICPS - LSIIT (CNRS UMR-7005)
Université Louis Pasteur, Strasbourg
Email: {hoenen,violard}@icps.u-strasbg.fr

***Abstract*— Solving the Vlasov equation represents a great challenge due to the huge size of the problem. A specific numerical adaptive method is used to reduce the amount of computations. This method uses a structured dyadic mesh. In this paper, we focus on the design of an appropriate data-structure for minimizing memory usage and time access. After having modeled the data accesses, we propose two data-structures and several optimizations. One data structure is based on hash tables and the other one on 2-level arrays. Experimental results show the performance of these structures for serial and distributed memory parallel machines.**

I. Introduction

The Vlasov equation (see [2] for its mathematical expression) is a partial differential equation (PDE) that can describe the evolution in time of charged particles under the effects of electro-magnetic fields. It is used to model and simulate some important phenomena in plasma physics such as controlled thermonuclear fusion. This equation is defined in the *phase space*, i.e., the position and velocity space $(x,v) \in \mathbb{R}^d, d = 2,4,6$. Its solution $f((x,v),t)$ represents the distribution of particles in phase space.

The numerical resolution of this equation is usually performed by particle-in-cell (PIC) methods [3] which approximate the plasma by a finite number of particles. These methods yields satisfying results with a relatively small number of particles and low computational cost. But, it is well known that the numerical noise inherent to particular methods becomes, in some cases, too important to get an accurate description of the solution. To remedy this problem, methods discretizing the Vlasov equation on a phase space mesh have been proposed [4], [5]. In this paper, we are interested in designing an efficient solver that use such non-particular methods.

Due to the high number of dimensions of the equation domain (6 for real cases), solving it with a non-particular method yields a very large computational problem. Indeed, considering a uniform discretizing grid, 512 points in each dimension are generally needed to achieve an accurate approximation. If we restrict ourselves to the study of particles in the 4D phase space ($d = 4$), then for each time step more than 68 billions unknowns have to be computed and a minimum of 512GB of memory are needed to store them. Moreover, the particle study is usually performed on a long period (several hundred of time steps).

This work is a part of the french INRIA project CALVI [1] devoted to the numerical simulation of problems in Plasma Physics and beams propagation.

Using an adaptive mesh is a well known issue for improving the efficiency of PDE solvers. Adaptive mesh refinement (AMR) [6] is a generic technique for transforming a uniform solver into an adaptive one. It divides the adaptive mesh into a collection of independent uniform ones, called *patches*. Besides the mathematical foundations of this technique, it yields very complex systems. Writing and maintaining code that implements these complex system is a difficult task. A lot of work has been done in this domain to simplify the task of programmer by introducing a hierarchy of abstractions [7]. A central issue in this context is designing an efficient data-structure to represent and handle the adaptive mesh [8]. Particularly in the case of the Vlasov resolution, given the huge amount of data and operations on these data, the choice of the data-structure is of great impact on performance. It is crucial that the used data-structure minimize memory usage and time access. In AMR methods, *patches* overlap each other and the time is refined differently for each patch. This can result in memory and computational overheads especially for high dimensional problems.

On the contrary to the classical AMR approach, our work is based on a specific adaptive numerical scheme for solving the Vlasov equation [9]. This scheme takes advantage of a property of this equation stating that the solution $f((x,v),t)$ is constant along the characteristics of the equation. Numerical methods which use this property are referred to as semi-Lagrangian methods [10]. With this scheme, time step is constant whatever the part of the phase space. Therefore, in comparison with AMR technique, time refinement overheads are avoided and several optimizations can be applied especially to the data-structure. Based on this semi-Lagrangian numerical scheme, we developed an adaptive 4D Vlasov solver [11] called YODA (for Yet anOther aDaptive Algorithm). In this paper, we discuss the design of an underlying data-structure for YODA. We have developed two data-structures: one is based on hash table and the other is based on multilevel arrays [12]. They are well suited to distributed memory parallel machines.

The paper is organized as follows: section II briefly recalls the numerical method. Section III gives an illuminating analysis of data accesses. Our two data-structures are detailed in section IV. Section V presents some optimizations of data accesses. Section VI outlines the use of our data-structures for obtaining good performance onto parallel machines. Experimental results are given in section VII and they show the performance of our structures.

II. YODA ALGORITHM

The solver YODA [9], [13] is based on a *semi-Lagrangian* numerical method whose mathematical definition can be found in [2]. We give the algorithm below, but first we introduce a few notions. For sake of simplicity, the notions relative to the adaptive mesh are given for the case $d = 2$. These definitions easily generalize to higher dimensions.

This method principle lies on an *advection operator*, denoted $\mathcal{A}_E$, which depends on the electric field, denoted as E. This operator identifies a characteristics curve of the equation and describes the move of particles in phase space going forward through one time step. The method uses the property stating that solution f is conserved along characteristics ,i.e., $f(\mathcal{A}_E(x, v), t^{n+1}) = f((x, v), t^n)$, for all position (x, v) in phase space and any time step n. The operator $\mathcal{A}_E$ is one-to-one. It is so called *forward advection operator* whereas its converse $\mathcal{A}_E^{-1}$ describes the move of particles going back through one time step and is called *backward advection operator* (see [10] for more details about characteristics and advection operators).

The solver YODA uses a *dyadic* structured adaptive mesh. The dyadic property can be expressed as follows. The mesh forms a partition of the computational domain and considering the unit square $[0, 1]\times[0, 1]$ as the computational domain, 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 $(i_1, i_2, j) \in \mathbb{N}$. Figure 1 shows a dyadic mesh.

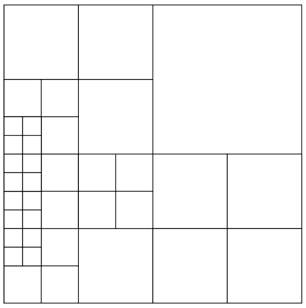

Fig. 1. A dyadic mesh (for $d = 2$)

This definition induces an implicit hierarchy of cells. For example, if 4 cells form a square, then they are considered as *daughter cells* of the cell they form which itself is called their *mother cell*. We sometimes called *sister cells*, some cells having the same mother. The integer j defines the cell size and identifies its *level* in the hierarchy. Higher levels in the hierarchy have smaller mesh cells than levels lower. In the rest, we consider only mesh with a fixed highest level, denoted J. We have thus $0\leq j\leq J$. Considering the levels of detail of the adaptive mesh, level 0 is the *coarsest level* and level J is the *finest level*.

Nodes of the mesh are located at the center, at the edge of each cell, and at the middle of each side. Therefore each cell has 9 equally spaced nodes as shown on Fig. 2.

More generally, for any dimension d, a mother cell has exactly 2^d daughters and each cell has 3^d nodes. Now, let us present the resolution algorithm.

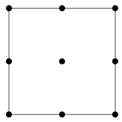

Fig. 2. Nodes of one cell (for $d = 2$)

The algorithm falls into four successive steps for each time step ($t^n \to t^{n+1}$). Fig. shows this algorithm. We will denote $\mathcal{M}^n$ and $\mathcal{M}^{n+1}$, the mesh at time t^n and t^{n+1} respectively.

The electric field computation step (1.) computes the values of charge density, denoted as ρ, at each position x of position space by summing the contribution of every cell of mesh $\mathcal{M}^n$ (by definition, $\rho(x) = \int f(x, v)dv$). Then the electric field E, is computed from ρ using Poisson equation ($\partial_x E = \rho$) and the advection operator $\mathcal{A}_E$ is then determined.

The prediction step (2.) builds a superset of $\mathcal{M}^{n+1}$ from scratch. Let us note $\widetilde{\mathcal{M}}^{n+1}$, this intermediate mesh. It is build as follows. For every cell, say α, of $\mathcal{M}^n$, its center point c_α is computed and advected forward using advection operator $\mathcal{A}_E$. The reached point, say b, ($b = \mathcal{A}_E(c_\alpha)$) determines an unique cell, say β having the same level as α and containing c_α. Then all daughters of β are inserted to $\widetilde{\mathcal{M}}^{n+1}$. This insertion implies several operations to maintain the global *mesh consistency*. The insertion of one cell in a dyadic mesh is illustrated on Fig.3.

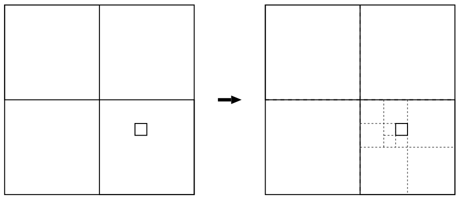

Fig. 3. Cell insertion in a dyadic mesh (for $d = 2$)

The evaluation step (3.) computes the approximate solution at every node of mesh $\widetilde{\mathcal{M}}^{n+1}$. For every node, say a, of $\widetilde{\mathcal{M}}^{n+1}$, the node point is advected backward using advection operator $\mathcal{A}_E^{-1}$. The reached point, say b, (by definition, $b = \mathcal{A}_E^{-1}(a)$) determines an unique cell, say α, of $\mathcal{M}^n$ containing b. Then the approximate solution at point b is computed from the approximate solution at nodes of cell α by using Lagrange interpolating polynomials. By the property of conservation of the unknown along characteristics, the approximate solution at node a of $\widetilde{\mathcal{M}}^{n+1}$ is the approximate solution at point b of $\mathcal{M}^n$.

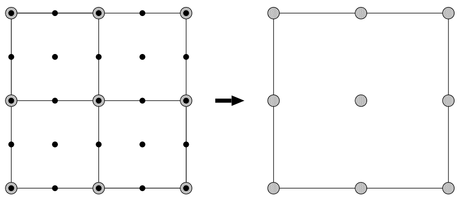

Fig. 4. Cell compression (for $d = 2$)

The compression step (4.) deletes useless elements from mesh $\widetilde{\mathcal{M}}^{n+1}$. It builds $\mathcal{M}^{n+1}$ from $\widetilde{\mathcal{M}}^{n+1}$. The procedure

1. Electric field computation
   for each cell $\alpha \in \mathcal{M}^n$:
      add contribution of α to ρ
   compute E from ρ
2. Prediction ($\mathcal{M}^n \to \widetilde{\mathcal{M}}^{n+1}$)
   for each cell $\alpha \in \mathcal{M}^n$:
      compute center of α ($\to c_\alpha$)
      advect c_α forward ($\to b$)
      find cell containing b ($\to \beta \in \mathcal{M}^{n+1}$)
      insert daughters of β to $\mathcal{M}^{n+1}$
3. Evaluation
   for each node $a \in \widetilde{\mathcal{M}}^{n+1}$
      advect a backward ($\to b$)
      find cell containing b ($\to \beta \in \mathcal{M}^n$)
      interpolate value at b from values of β
      store value at node a of $\widetilde{\mathcal{M}}^{n+1}$
4. Compression ($\widetilde{\mathcal{M}}^{n+1} \to \mathcal{M}^{n+1}$)
   for each group of sister cell
      for each node a of the group
         interpolate value at a from values of β
         compute difference between stored and interpolated values
      if (norm of the differences $< \epsilon$)
         replace daughter cells with mother

Fig. 5. Time marching resolution algorithm

is the following. While there exists a group of sisters in $\widetilde{\mathcal{M}}^{n+1}$ that has not yet been considered, compute the norm of the differences between values at each node and values interpolated from mother nodes. If this error norm is lower than a certain threshold ϵ, then replace the group of sisters with their mother in $\widetilde{\mathcal{M}}^{n+1}$ and delete daughter nodes and values. Cell compression is illustrated on Fig.4.

III. Data accesses

Here we report a study on how the algorithm access data. We wish to determine how much and in what way the algorithm access to the data structure. Such a preliminary study is of great importance to determine what structure will store data the most efficiently.

A. Analysis of data accesses

Our analysis lies on a classification of data accesses according to three criteria: read or write (Re/Wr) access, access to a cell or a node (C/N), and random or serial access (R/S). Let us precise our meaning for random and serial access. Serial access in our sense is an access which only depends on a reference to data previously accessed and random access is the opposite of serial access. Random accesses arise when the advection operator is used (steps (2.) and (3.)).

For each step of the algorithm, the approximate number of data accesses is reported and characterized with these criteria. We note $||\mathcal{C}^n||$ and $||\mathcal{N}^n||$ the number of cells and nodes of $\mathcal{M}^n$, respectively. We also note $||\widetilde{\mathcal{C}}^n||$ and $||\widetilde{\mathcal{N}}^n||$, the number of cells and nodes of $\widetilde{\mathcal{M}}^n$, respectively.

Step (1.) (electric field computation), the cells of mesh $\mathcal{M}^n$ are scanned in an arbitrary order. Therefore $||\mathcal{C}^n||$ serial accesses are performed. For each cell, all nodes are read once in an arbitrary order. We formulated this as:

$$\left(||\mathcal{C}^n||\right)_{ReCS} + \left(||\mathcal{C}^n||\times 3^d\right)_{ReNS} \tag{1}$$

Step (2.) (prediction), the cells of mesh $\mathcal{M}^n$ are scanned in an arbitrary order again. For each cell, all daughters of the unique cell containing the advected center, are inserted in mesh $\widetilde{\mathcal{M}}^{n+1}$. These insertions entails some read and write accesses for maintaining mesh consistency. For sake of simplicity, we neglect these accesses in our formula.

$$\left(||\mathcal{C}^n||\right)_{ReCS} + \left(||\mathcal{C}^n||\times 2^d\right)_{WrCR} \tag{2}$$

Step (3.) (evaluation), the nodes of mesh $\widetilde{\mathcal{M}}^{n+1}$ are scanned in an arbitrary order. To find the unique cell containing the backward advected node, the presence in the mesh of at most J cells is tested. Then the 3^d node values are read for the previously determined cell in order to perform the interpolation. Last, the interpolated value is added to mesh $\widetilde{\mathcal{M}}^{n+1}$.

$$\begin{aligned} &\left(||\widetilde{\mathcal{N}}^{n+1}||\right)_{ReNS} + \left(||\widetilde{\mathcal{N}}^{n+1}||\times J\right)_{ReCR} + \\ &\left(||\widetilde{\mathcal{N}}^{n+1}|| \times 3^d\right)_{ReNR} + \left(||\widetilde{\mathcal{N}}^{n+1}||\right)_{WrNS} \end{aligned} \tag{3}$$

Step (4.) (compression), the cells of mesh $\widetilde{\mathcal{M}}^{n+1}$ are scanned in an arbitrary order. Then for each groups of sister cells, a compression test is performed. The number of groups of sister cells depends on the form of the mesh. A limit superior of this number is $\frac{||\widetilde{\mathcal{C}}^{n+1}||}{2^d-1} = ||\widetilde{\mathcal{C}}^{n+1}||/2^d + (||\widetilde{\mathcal{C}}^{n+1}||/2^d)/2^d + \ldots$. In order to perform the compression test, the 5^d node of each of these groups are scanned with an arbitrary order. The compression test succeeds for a proportion of these groups. This proportion depends on threshold ϵ. Let us note $r_\epsilon \in [0, 1]$, this ratio. Every group of sister cells for which the compression test succeeds, is replaced with their mother in mesh $\widetilde{\mathcal{M}}^{n+1}$. This implies approximatively $\frac{||\widetilde{\mathcal{C}}^{n+1}||}{2^d-1} \times r_\epsilon$ write accesses using the same order as for scanning cell groups. Moreover, we assume that $||\mathcal{C}^{n+1}||$ is approximatively equal to $||\mathcal{C}^n||$, i.e., the number of mesh cells is conserved through time steps. Under this assumption, we find an expression for r_ϵ. This yields the following formula:

$$\begin{aligned} &\left(||\widetilde{\mathcal{C}}^{n+1}||\right)_{ReCS} + \left(\frac{||\widetilde{\mathcal{C}}^{n+1}||}{2^d-1} \times 5^d\right)_{ReNS} + \\ &\left(\frac{||\widetilde{\mathcal{C}}^{n+1}||}{2^d-1} \times (1 - \frac{||\mathcal{C}^n||}{||\widetilde{\mathcal{C}}^{n+1}||}\right)_{WrCS} \end{aligned} \tag{4}$$

$\diamond$

These formulae show that the number of random accesses is far greater than the number of sequential accesses for any dimension d: assuming that the number of mesh cells is conserved through time steps, i.e., $||\mathcal{C}^{n+1}|| = ||\mathcal{C}^n||$ and $||\widetilde{\mathcal{C}}^{n+1}|| = ||\widetilde{\mathcal{C}}^n||$, and considering that for any given mesh, the ratio of the number of nodes to the number of cells is at least 2^d (in case of an uniform mesh), i.e., $\|\widetilde{\mathcal{N}}^{n+1}\| = \|\widetilde{\mathcal{C}}^{n+1}\| \times 2^d$, formulae express that the number of random accesses is greater than twice the number of serial ones for $d = 2$, and more than ten times greater for $d = 4$.

Thus although a tree structure appears to be a natural representation of a dyadic mesh, our algorithm exhibits random accesses and we have to design a more appropriate data-structure.

In YODA algorithm only elements at leaves of the tree are examined and the traversal of all the levels of the cell hierarchy is too slow for achieving good performances. On the contrary, ND-tree structure [6] are commonly used in AMR algorithms because these methods put interest in elements at all levels of hierarchy.

Therefore our data-structure should be based on tables in which elements are accessed by an index. We first define the indexing of elements.

B. Indexing

Elements of our data-structure are nodes and cells.

A node refers to a position in phase space (and a value of the solution). Therefore we define a *node index* from the position. For sake of optimization, the index of a node is an integer. The binary representation of this integer is obtained by concatenating d strings of $J + 1$ bits. Each string is the binary representation on $J + 1$ bits of a coordinate of the corresponding point in the finest uniform grid as illustrated on Fig.6.

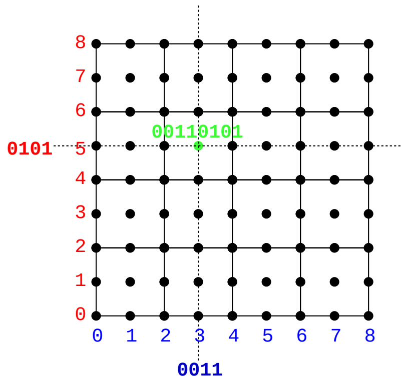

Fig. 6. *node indexing for $d = 2$ and $J = 2$*

Identifying the mother cell given one of its daughter or finding one sister of a given cell are some elementary operations in our algorithm. The indexing of cells must thus be defined so that these operations are performed very quickly. Therefore a *cell index* reflects the hierarchy of cells. As for node index, the index of a cell is an integer. The binary representation of the index of a cell is obtained recursively from the binary representation of its mother by adding d bits to the right, as illustrated on Fig.7.

IV. Data structure

Section III has shown that the underlying data-structure of our solver should provide fast random accesses, so the structure should use some kind of table. Moreover, adjacent cells

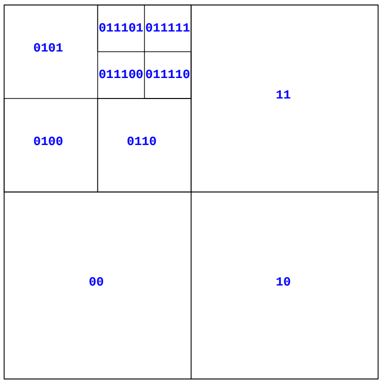

Fig. 7. *cell indexing for $d = 2$ and $J = 3$*

have nodes in common. It is very important that these nodes do not be duplicated in the data-structure for avoiding memory and computational overheads. This node sharing implies that nodes and cells are stored in their own structure. Therefore the dyadic mesh should be represented by two tables: the *cell table* and the *node table*. We decided to use hash table and multilevel array to implements these tables.

A. Hash table

A hash table is a very classical structure [14].

1) Cell table implementation: A cell is represented in the hash table by an element whose key is the pair level-index and whose value is not significant. In order to avoid collisions, we define a *perfect hash function*. Moreover, in order to improve locality, our hash function tends to associate similar hashes with cells that are close in the computational domain. This hash function is defined as follows:

$$hash(\alpha) = \alpha.index \ll \big((J - \alpha.level) \times d\big) \qquad (5)$$

where $\ll$ is the left shift operator. Different keys may have the same hash, but it is always for cells that cannot be in mesh simultaneously, one being systematically an ancestor of the other.

2) Node table implementation: A node is represented in the hash table by an element whose key is the node index and whose value is the value of the solution f at the corresponding point in phase space. Again, we define a perfect hash function as follows:

$$hash(a) = a.index \qquad (6)$$

B. Multilevel arrays

A multilevel array is an array where each element is a value or a multilevel array itself. In order to access to an element at a given level in such an array, we need to traverse all previous levels. Therefore, in order to enhance access time, we restrict ourselves to 2-level arrays. Moreover, we only consider 1D-arrays.

1) Cell table implementation: A cell is represented in the 2-level array by a value, which is a pair level-index. The implementation of the cell table is defined for a given $j_0 \in \{1, ..., J-1\}$. Cells of level from 0 to j_0 are stored in the array at first level. We then call this array the *coarse array*. Cells of level from j_0+1 to J are stored in arrays at second level. We then call these arrays the *fine arrays*. The position of a cell in the 2-level array is computed from the cell index.

2) Node table implementation: A node is represented in the 2-level array by a value, which is the value of the solution f at the corresponding point in phase space. The implementation of the node table is defined by the same j_0 as for cell table. The position of a node in the 2-level array is computed from the node index. The coarse array stores all the nodes of cells at level j_0. All other nodes are stored in fine arrays.

V. Optimizations

The main lines of the data structures are established, we are now presenting some optimizations that relies on data access performances.

A. Direct access

Section III shown that evaluation step (3) is the most data access demanding part of the program. This is due to interpolation for which all the values at nodes of a cell must be accessed. In order to improve accesses, we add 3^d pointers to each element of the cell table. Each element represents a cell and the pointers points to the values of the solution at nodes of the cell. So, the values at nodes of a cell are not accessed via the node table but directly by address. However, this optimization entails an overhead due to pointers assignment.

B. Storage remapping

This optimization aims to make cache effective by achieving good data locality. Amongst the different approaches that had been developed to improve data locality, let us cite *loop restructuring* [15], [16], where execution order of iterations is changed, and *array restructuring* [17], [18], where the storage order of an array is modified. In our case, loops on elements of the adaptive mesh are too complex to allow loop restructuring. Our optimization is based on array restructuring principle.

We focus again on step (3.) of our algorithm. Our optimization aims to enhance locality of the values at nodes of cells for avoiding cache miss during interpolations. These values must be contiguous in memory as much as possible. To achieve this goal we use a kind of *storage remapping* [19], [18], [20]. It is illustrated on Fig.8: values are stored in dynamically allocated arrays called *remapped arrays* and the node table do not stores values but pointers to values in a remapped array.

A simple and efficient heuristics consists in keeping the same order for the stored values than the one obtained from the scan of nodes during the evaluation step. As said previously nodes are shared between different cells, therefore it is not possible to achieve exact locality. However locality can be greatly improved by using this optimization.

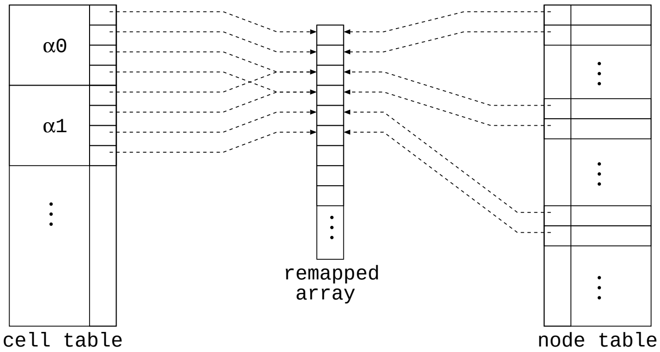

Fig. 8. *Direct access and storage remapping*

VI. Parallelization overview

Here we show that our data structures are well-suited for memory distributed parallel architectures. Our algorithm is implicitly data-parallel since the adaptive mesh can be viewed as a data-parallel structure. Therefore the parallelization of our solver mainly relies on the distribution of the data-structure among processors.

A. Data distribution

The computational domain is virtually subdivided into *regions*. A *region* is a surface of the computational domain which is defined by an union of cells of the mesh. Regions are allocated to processors so that a processor owns and computes the mesh cells and nodes which are included in its region.

Each processor owns a local view of the distributed mesh. This local view is the union of the cells of the local region plus a minimum of other cells for covering the whole domain. Each processor thus owns in its local memory one cell table representing the local view and one node table storing all values at local nodes. Moreover, one field is added to each element of the cell table. This field is used to store the owner processor rank. Therefore any processor can obtain the rank of the processor which owns any required data and can communicate data with it. This field is also used for implementing load balancing by switching two owner processor ranks in the local cell table.

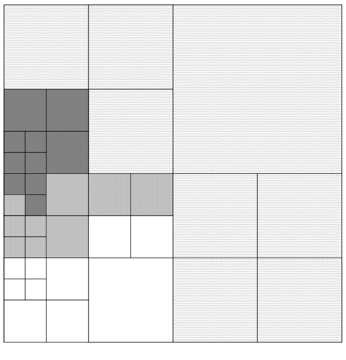

Fig. 9. *Mesh distribution among 4 processors*

B. Load balancing

As the mesh adapts to the evolution in time of the physics, the number of cells within a region changes. It is therefore necessary to integrate a load balancing mechanism into our solver. This mechanism then consists in redefining regions for each processor when a load imbalance is detected.

Newly updated regions must represent the same amount of workload and be compact in order to reduce communication volume. Assuming that the cell is the unit of load, each region must approximately contains the same number of cells, and these cells must be connex. Since connexity is hard to maintain as the dimensionality increases, we choose to restrict the neighborhood by by using the Hilbert's *space filling curve* (SFC) [21]. This curve is extensible in dimension and can fill a d-dimensional dyadic mesh as shown on Fig. 10.

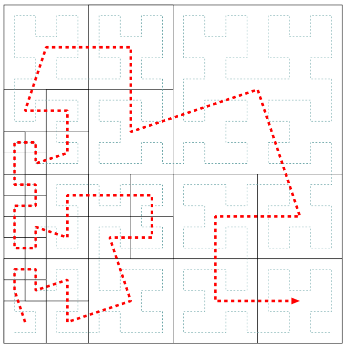

Fig. 10. *Hilbert's space filling curve for $d = 2$*

Each cell possess two neighbors: the previous one and the next one along the Hilbert's curve and a connex region corresponds to a portion of the Hilbert's curve.

Region updating consists then in determining the number of cells to receive from or send to neighbors at each ending of the local portion of curve. We define this number as the sum of differences between the local (region) load and the ideal load.

Load balancing is performed between step (2.) and (3.) so that the evaluation step is well balanced. Moreover, at this stage of the algorithm, the values at nodes have not been computed yet and communication overheads are thus reduced to the minimum.

Last, a processor performs compression step only within the limits of its region. Doing this, step (4.) of the algorithm do not require any communication. This is an approximation of the method since less cells are deleted, but it does not hazard convergence.

VII. Experimental results

The code YODA was implemented in C++/MPI and We developed 2 versions: one with hash tables (code 1) by using standard template library (STL), and one with 2-level arrays (code 2) with remapped arrays of 1024 elements each.

Experiments have been performed on a HP cluster with 30 Itanium bi-processors nodes running at $1.3Ghz$ and interconnected through a $2Gbits/s$ network.

The test case is a rotating Gaussian in 4D phase space. The finest level of the mesh is $J = 5$ which is equivalent to a uniform grid of 64^4 discretizing points. The simulation is 100 time steps long.

We have measured the execution time and memory usage of:

1) code 1 without optimization and with both direct access and storage remapping.
2) code 2 without optimization, with direct access, with both direct access and storage remapping, and for several values of j_0.

The results are reported on Table I and II. In table I, we notice that the optimized code is three times faster than the code without optimization. But the optimized code uses two times more memory. This overhead can be partly explained by the big amount of pointers, which has to be allocated. The number of pointers in the node table is at least equal to the number of nodes, whereas in the cell table, each element stores $3^4 = 81$ pointers to remapped arrays.

TABLE I

Runtime and memory usage of code 1

hash table	without optimization	direct access and storage remapping
runtime (s)	1506.3	453.79
memory use (KB)	1002240	2033600

Table I shows that the data structure based on 2-level arrays is more efficient than with hash table. Even with the same optimizations, it appears that code 1 is slower than code 2. This can have different causes: the cost of accessing an element (cell or node) or allocating new elements, the number of elementary operations to scan all elements of the structure.

We notice that the results obtained with code 2 and direct access only are the best. This shows that the overhead of the remapping optimization is bigger than its gain. As the remapping optimization aims to improve data locality, it shows that the 2-level array structure present a good data locality.

For our test case, we obtain the best performance for $j_0 = 3$. For lower values of j_0, fine arrays are larger which badly affects data locality. On the opposite, for higher values of j_0, fine arrays are smaller and this implies much more allocation and deallocation.

TABLE II

Runtime and memory usage of code 2

2-level array	j_0	without optimization	direct access and storage remapping	direct access
runtime (s)	2	411.15	372.44	328.9
	3	403.15	314.18	262.97
	4	426.76	334.74	290.05
memory use (KB)	2	334560	502320	362624
	3	248000	1060784	921088
	4	580688	1474784	1335088

Fig.11 shows the performance of each code for different number of processors. We notice that the optimizations do not affect the speed-up.

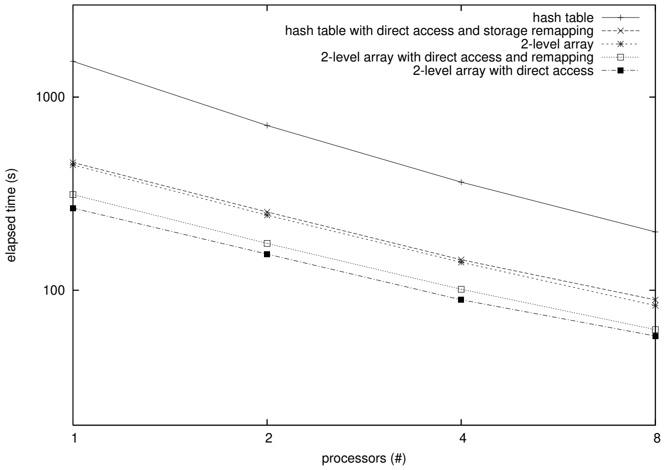

Fig. 11. *Performance of our parallel codes*

VIII. Conclusion

We proposed two data-structures for an adaptive numerical solver of the Vlasov equation. We based our design on a quantitative and qualitative analysis of data accesses. Experiments show the performance of our structures and the advantages of the proposed optimizations. The proposed data-structures are well-suited to load balancing on a distributed memory parallel architecture.

The structure with 2-level arrays and direct access exhibits the best performances for the considered test case. Performance of this structure depends on the value of parameter j_0. Further experiments have to be performed to determine how to find the best value of j_0.

Experiments show that the proposed optimizations significantly reduce the execution time but also increase memory usage. This work then gives some clues as to which structure is the best for real case ($d = 6$) for which a tradeoff has to be made between memory usage and access time.

References

[1] CALVI : french INRIA project, http://www.inria.fr/recherche/equipes/calvi.fr.html.

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

[3] C. K. Birdshall and A. Langdon, *Plasmaphysics via computer simulation*. McGraw-Hill, 1985.

[4] F. Filbet, E. Sonnendrücker, and P. Bertrand, "Conservative numerical schemes for the Vlasov equation," *J. Comput. Phys.*, vol. 172, pp. 166–187, 2000.

[5] F. Filbet, E. Sonnendrücker, and J. Lemaire, "Direct axisymmetric vlasov simulations of space charged dominated beams," in *International Conference on Computational Science, ICCS 2002*, ser. Lecture Notes in Computer Science, 2002, pp. 305–314.

[6] M. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *J. Comput. Phys.*, vol. 53, pp. 484–512, 1984.

[7] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large scale parallel structured amr calculations using the samrai framework," in *Supercomputing*, 2001.

[8] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski, "A common data management infrastructure for adaptive algorithms for PDE solutions," in *Supercomputing*, 1997.

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

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

[11] 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*, ser. Lecture Notes in Computer Science, vol. 3241. Springer-Verlag, September 2004, pp. 430–435.

[12] J. W. T. A. L. Rosenberg, "What is a multilevel array?" *IBM J. Res. Develop.*, vol. 19, no. 2, pp. 163–169, 1975.

[13] M. Mehrenberger, E. Violard, O. Hoenen, M. C. Pinto, and E. Sonnendrücker, "A parallel adaptive vlasov solver based on hierarchical finite element interpolation," in *ICAP'04*, 2004.

[14] D. Knuth, *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1973, vol. 3.

[15] S. Carr, K. S. McKinley, and C. W. Tseng, "Compiler optimizations for improving data locality," in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1994, pp. 252–262.

[16] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, 1992.

[17] S. A. Leung, "Array restructuring for cache locality," Department of computer science, University of Washington, Tech. Rep. TR-96-08-01, 1996.

[18] M. T. Kandemir, A. N. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee, "Enhancing spatial locality via data layout optimizations," in *European Conference on Parallel Processing (EuroPar '98)*, 1998, pp. 422–434.

[19] M. Cierniak and W. Li, "Interprocedural array remapping," in *International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, November 1997.

[20] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *IEEE PACT*, 1999, pp. 192–202.

[21] H. Sagan, *Space-Filling Curves*. Springer, 1994.