A New Parallelization Scheme for the Hermite Interpolation Based Gyroaverage Operator

Nicolas Bouzat[†]*, Fabien Rozar[‡§], Guillaume Latu*, Jean Roman[¶]
*CEA/IRFM, FR-13108 Saint-Paul-lez-Durance
[†]Inria, FR-54600 Villers-lès-Nancy
[‡]Maison de la Simulation, FR-91191 Gif-sur-Yvette
[§]Laboratoire de Mécanique et Génie Civil (LMGC),
Université de Montpellier, CNRS, FR-34000 Montpellier
[¶]Inria, Bordeaux INP, CNRS (LaBRI),
FR-33405 Talence

***Abstract*—Gyrokinetic modeling is appropriate for describing plasma turbulence in the core of Tokamaks, and the gyroaverage operator is a cornerstone of this approach. In a gyrokinetic code the gyroaveraging scheme needs to be accurate enough, but also requires a low computational cost because it is often applied on the main unknown, namely the 5D guiding-center distribution function, as well as on several 3D fields. The current gyroaverage implementation used in the GYSELA code has recently been improved [6], enhancing the precision of the operator thanks to Hermite interpolation. In the present paper, we describe a new parallelization scheme for the gyroaverage operator. It mainly avoids costly transpositions of the full 5D function using halo exchange instead. Though the computational cost remains the same, the communication one is much smaller. The overall algorithm is also improved by cleverly interleaving communications and computations, thus allowing for a reduction of communication costs and a more efficient thread parallelization. The execution time with this algorithm is up to twice as fast as the previous version. The benefit of an improved scheme providing the overlap of communications by computations is also shown, again improving execution times. The description of the algorithms is given, together with an analysis of the achieved performance.**

I. INTRODUCTION

Gyrokinetic theory is the framework chosen for the simulation code named GYSELA [4]. This parallel code is used to study turbulence dynamics in the core of a Tokamak plasma. The gyroaverage operator $\mathcal{J}$, a key element of the gyrokinetic model, transforms the so-called guiding-center distribution into the actual particle distribution [2]. It enables one to take into account the cyclotronic motion of the particles around the magnetic field lines at a distance called *Larmor radius*. These motions are faster than the turbulence we are looking at and the computational costs for simulating them explicitly would be too high. In the present paper, we improve the parallelization of the gyroaverage operator in the GYSELA code to shorten the execution time. In a gyrokinetic code, the gyroaveraging scheme needs to be accurate enough to avoid spoiling the data, but also requires low computational costs because it is applied several times per time step on the main unknown, the 5D guiding-center distribution function. The gyroaverage is used to compute the right-hand side of the Poisson equation, to compute the gyroaveraged electric potential that is used to get the advection field in the Vlasov solver and in several diagnostics that export physical quantities on mass storage. Our aim is to reduce the cost of this operator without compromising the overall numerical accuracy. In this context, the optimization of numerical methods, algorithms and implementations is a major issue. Physicists perform large GYSELA simulations using from 1k to 16k cores on supercomputers. This paper presents the improvements made on the algorithm based on Hermite interpolation which significantly speeds up the gyroaverage operator. Technical points are further described in the extended version of this paper [1].

The GYSELA code uses a five dimension mesh $r \times \theta \times \varphi \times v_{\parallel} \times \mu$ to simulate plasma charged particles evolving in the Tokamak. These particles are confined in a strong magnetic field. r, θ and φ are the spatial dimensions, $v_{\parallel}$ the speed along the magnetic field lines and μ the magnetic momentum. The gyroaverage operator mimics the cyclotronic motion projected in the $r \times \theta$ plane. Indeed the component of the motion along the field line is negligible towards its rotation around the field line. Thus, only data from a same plane are required to compute the gyroaverage. This paper essentially focuses on the integration of the gyroaverage operator in one costly diagnostic using the gyroaverage on the whole distribution function. Integration in other sections of the code will be part of future works. In this diagnostic, the data distribution is such that the r and θ dimensions are split among the MPI processes and the φ and $v_{\parallel}$ dimensions are entirely contained on each process. The μ dimension is always split between groups of MPI processes (*i.e.* a MPI process has only one value of μ). In the previous implementation of the gyroaverage operator, a Padé approximation method was used. It requires the whole $r \times \theta$ plane to be stored locally in memory. Therefore, it was necessary to transpose the whole distribution function between MPI processes before and after the computation of the gyroaverage. The method based on Hermite interpolation does not have this requirement which permits us to avoid these costly transpositions. Moreover it does not damp the small variations of the function as the Padé approximation did. This new method has previously been implemented in

GYSELA [6], replacing the Padé approximation based operator and leading to a great increase in precision and possibilities for better parallelization schemes. However, the transpositions were kept, as it was easier for a first integration in GYSELA to keep the same parallel strategy.

The work we have done on the gyroaverage operator is presented in this article as follows. The current method for gyroaveraging using Hermite interpolation is explained in Section II. Section III shows how the algorithm has been redesigned to fit a new data distribution. Section IV details the optimization done by overlapping computations and communications, shortening the execution time by a factor two in the best cases. Both of these two sections detail the performance results obtained in the GYSELA code. Section V concludes and gives some hints to optimize further the gyroaverage operator.

II. GYROAVERAGE ALGORITHM AND CURRENT IMPLEMENTATION

This section gives the details of how the gyroaverage is used in GYSELA.

A. Gyroaverage operator

We will now describe the numerical framework and approximations that are made for the gyroaverage operator. Let us consider a Larmor radius ρ, a grid in polar coordinates $r \times \theta$ (poloidal plane) and a function f defined over this grid. The gyroaverage operator $\mathcal{J}_\rho$ consists, for each point P of the plane, in a weighted integral of the value of f over the circle of radius ρ and center P. In a discrete space, this translates as a mean of $N_\mathcal{L}$ points uniformly distributed on the circle of Larmor and interpolated with the Hermite method. The precision of the operator directly depends on $N_\mathcal{L}$. An example is shown in Figure 1 for $N_\mathcal{L} = 5$. To compute the gyroaverage at the point ●, $N_\mathcal{L}$ points ▲ are placed on the circle of radius ρ. As these points are unlikely to coincide with a mesh point, an Hermite interpolation is performed for each of them using the values at the four corners ■ of the cell in which they are contained. Thus the gyroaveraged value of f at a point (r_i, θ_j) of the grid writes

$$\mathcal{J}_\rho(f)(r_i,\theta_j) \simeq \frac{1}{N_\mathcal{L}} \sum_{k=1}^{N_\mathcal{L}} \mathcal{H}(f)(x_k, y_k) \qquad (1)$$

with $\mathcal{H}$ being the Hermite interpolation function, $x_k = r_i cos(\theta_j) + \rho cos(\theta_j + k\frac{2\pi}{N_\mathcal{L}})$ and $y_k = r_i sin(\theta_j) + \rho sin(\theta_j + k\frac{2\pi}{N_\mathcal{L}})$ being the coordinates of the points on the Larmor circle in Cartesian coordinates. When one of these points is outside the mesh, a radial projection is done on the inner or outer border.

Given one of the $N_\mathcal{L}$ points ▲ of coordinate $(\tilde{r}, \tilde{\theta})$, the computation of the interpolation $\mathcal{H}$ requires the value, the derivatives in r and θ and the cross-derivatives of each of the corners of the containing cell. The 2D interpolation behaves as if two 1D interpolations along r and θ were performed. First, an interpolation along r is performed from the two corners of each θ side of the cell, θ_1 and $\theta_1 + 1$, to the points of same respective θ coordinate and same r coordinate as the target point ($r = \tilde{r}$). Then a second interpolation along θ is performed from these two new points to reach the target point ($\tilde{r}$, $\tilde{\theta}$). The coefficients used for the Hermite interpolation are detailed in [6].

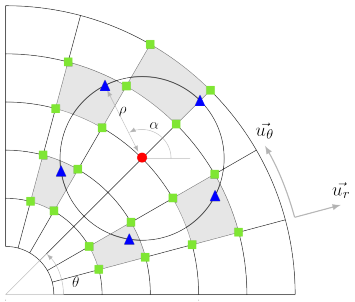

Figure 1: Computation of the gyroaverage.

B. Gysela implementation

This section presents how the numerical scheme explained in the previous paragraph is implemented in GYSELA according to the work conducted in [6]. We denote N_r and N_θ, the number of mesh points in the r and θ dimensions.

For a given point, the computation of the gyroaverage requires a certain set of points which coefficients in the Hermite interpolation are stored in the matrix M_{coef}. Each line i of the matrix corresponds to the coefficients of the value and three derivatives of all the points used in the interpolation of the point $(r{=}r_i, \theta{=}0)$. These coefficients are also valid for every θ-value thanks to radial symmetry. Furthermore, M_{coef} is sparse as for each r_i, only a few cells (grey cells in Fig. 1) along the Larmor circle center in $(r{=}r_i, \theta{=}0)$ are involved in the gyroaverage. Their number depends on $N_\mathcal{L}$, ρ and r.

In the actual implementation, M_{coef} is represented as an array of dimension N_r where each cell contains: an array of indexes indicating which point is involved in the interpolation for the given r and of an array of corresponding coefficients. Thus the matrix is reduced to the minimum[1] and is computed once for all during the initialization. The size of M_{coef} is then at most of $36 N_r N_\mathcal{L}$ elements (see [1]). As GYSELA simulations usually use $N_\mathcal{L} = 8$, this storage method is lighter than storing the whole mesh for each radius ($N_r^2 N_\theta$ elements). Once the initialization ended and M_{coef} computed, the gyroaverage operator is called several times during each time step on the whole poloidal plane. A matrix M_{fval} is built at the beginning of a call by computing the derivatives at each point of the plane registered in M_{coef}.

The cost of the core computation of the gyroaverage operator, where the matrices of coefficients and values are multiplied, is described in [1]. In short, the number of operations to compute the gyroaverage of one point depends, in most cases and for a given r value, on the number of different points involved in the interpolations of the $N_\mathcal{L}$ points on the Larmor circle. The cost of gyroaveraging one poloidal plane is then

[1] If two interpolation points are contained by neighboring cells, the different contributions of the overlapping corners are stored in a single coefficient in M_{coef}.

$\Theta(N_{\mathcal{L}} N_r N_\theta)$, so the total cost for the full 5D distribution function is $\Theta(N_{\mathcal{L}} N_r N_\theta N_\varphi N_{v_\parallel} N_\mu)$.

In GYSELA the distribution of data is twofold. A MPI process $P_{i,j}$, located on the i-th r row and j-th θ column of MPI processes, either has data $\mathbf{D}_1(r = *, \theta = *, \varphi = \varphi_i \rightarrow \varphi_{i+1}, v_\parallel = v_j \rightarrow v_{j+1}, \mu = \mu_{i,j})$ or $\mathbf{D}_2(r = r_i \rightarrow r_{i+1}, \theta = \theta_j \rightarrow \theta_{j+1}, \varphi = *, v_\parallel = *, \mu = \mu_{i,j})$ where $r_i = i \times (N_r / N_{\texttt{procr}})$ and similarly for θ_j, φ_i and v_j. $N_{\texttt{procr}}$ is the number of MPI processes in the radial direction. Several processes share the same $\mu_{i,j}$. The gyroaverage, as implemented in previous version, requires the full poloidal plane in local memory, *i.e.* distribution $\mathbf{D}_1$. However the data distribution when the operator is called is $\mathbf{D}_2$. It thus requires a costly transposition of the full distribution function before the gyroaverage and after as the computation performed subsequently requires the $\mathbf{D}_1$ distribution. This is the reason why a gyroaverage operator which can handle directly $\mathbf{D}_2$ distribution can drastically reduce the volume of communication.

Algorithm 1: Hermite gyroaverage in GYSELA

```
Data: Distribution function f, N_{lφ} and N_{lv_∥}
Result: Gyroaveraged distribution function J_0.f
begin
1   f_tmp = transpose_forward(f)
    OpenMP parallel zone
      for i : 0 → N_{lφ} N_{lv_∥} − 1 do
2       preprocess(f_tmp(i))
3       gyroaverage(f_tmp(i))
4       postprocess(f_tmp(i))
5   f = transpose_backward(f_tmp)
```

Algorithm 1 shows how the basic Hermite interpolation gyroaverage based on transpositions was integrated in GYSELA inside the diagnostic in which the new version will be implemented. $N_{l\varphi}$ and $N_{lv_\parallel}$ are the dimensions of the local subdomain in φ and $v_\parallel$. f_{tmp} exactly corresponds to f but using distribution $\mathbf{D}_1$. During the preprocessing step 2, the function to be gyroaveraged is built from the 5D distribution function. The same goes for the post-process step 4 where some macro-data are gathered on the gyroaveraged function (fluid momentum, velocity integrals over $v_\parallel$ and μ ...).

III. PARALLELIZATION WITH HALO EXCHANGE

This section details the new solution for the gyroaverage operator using Hermite interpolation. The algorithm has been changed to fit data distribution $\mathbf{D}_2$, and some optimizations have been done for the parallelization. The numerical analysis and core computations remain the same and communication schemes are improved. We only consider the cases were the Larmor radius ρ is small against the innermost radius of the poloidal plane r_{min}.

A. Halo and ghost points

In order to compute the gyroaverage with only a $r \times \theta$ patch of data in local memory, it is necessary to exchange a few data between processes.

Indeed, the computation of the gyroaverage for points on the border of a $r \times \theta$ patch requires a certain number of values from other MPI processes depending on the Larmor radius and on the discretization of the mesh. Considering Figure 2, the ● point of process 2 requires values and derivatives from points ■ located on the processes 1, 3 and 4. ■ point also requires neighboring points to compute their derivatives as seen in II-B. The number of exchanged points also depends on the r coordinate of the point to be gyroaveraged. The closer it is to the inner circle of the plane, the narrower the meshing in θ becomes and so the more cells the Larmor circle is likely to intercept. The more cells the Larmor circle intercepts, the larger the halo will be and thus the communication. The computation still mainly depends on $N_{\mathcal{L}}$ but now also loosely on the Larmor radius.

The halo consists of all the points located on neighbor processes (ghost points) needed by a process to be able to apply the gyroaverage on its local subdomain. Its size must be as small as possible so that its communication cost would be smaller than the cost of a full transposition. Otherwise the gain of Hermite interpolation gyroaverage would be lost. The computation of the size of the halo only requires knowing the points involved in the interpolation of the four corners of the patch. It can even be reduced to the knowledge of one corner as shown later.

For each specific subdomain, the number of ghost points in dimension r is N_{ghost_r}, and N_{ghost_θ} in dimension θ. N_{ghost_r} can be easily computed as the distance between two points with consecutive r coordinate and same θ coordinate is a constant value $((r_{max} - r_{min})/N_r)$. r_{max} and r_{min} are the radii of the r borders of the poloidal plane. Thus

$$N_{ghost\_r} = \lceil \frac{\rho N_r}{(r_{max} - r_{min})} \rceil + N_{deriv}$$

where N_{deriv} is the number of points used to compute the derivatives (here two). However, it is more complicated for N_{ghost_θ}. Considering the center of a Larmor circle (r, θ), the point with maximum r on the circle is $(r + \rho, \theta)$. The point with maximum θ on the circle though is not $(r, \theta + \rho/r)$, but the intersection of the tangent to the Larmor circle, originated from the center of the plane O, with the Larmor circle.

Thus the computation of the halo size follows these steps: given the corner of lowest r and highest θ of the patch, the indexes of the corners of the cells containing the points of its Larmor circle are calculated. Among those points, the largest θ index is kept and the difference with the θ index of the center of the Larmor circle gives number at which we need to add N_{deriv} to obtain N_{ghost_θ}. Finally, the size of the halo writes $N_{\mathcal{H}} = 4N_{ghost\_r}N_{ghost\_\theta} + 2(N_{ghost\_r}N_{l\theta} + N_{ghost\_\theta}N_{lr})$ with N_{lr} and $N_{l\theta}$ being the dimensions of the local subdomain. The Figure 3 represents these values on a subdomain.

Furthermore we chose to have a unique communication scheme for all processes sharing the same Larmor radius in order to simplify the implementation. It means, given a Larmor radius, that every process of the poloidal plane will have the same N_{ghost_r} and N_{ghost_θ} even though N_{ghost_θ} depends on r. Of course, the largest one is chosen as it is needed by the processes in charge of the inner radii. This simplification increase drastically the number of exchanged points if r_{min}

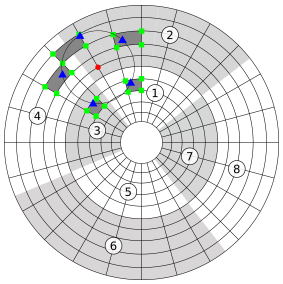

Figure 2: Data distribution over MPI processes and gyroaverage example.

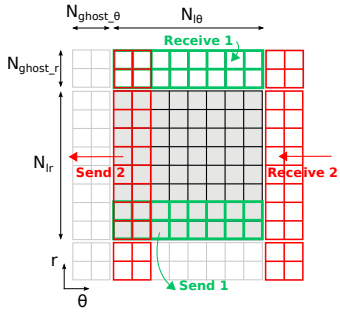

Figure 3: Subdomain representation (local points are in gray) and halo communication scheme.

is approximately equal to ρ which is not yet the case in GYSELA. The communication scheme for halo will need to be refined accordingly, keeping in mind that processes with smaller halo communications will anyway wait for processes with the highest communication costs. To conclude, the size of the halo is computed during the initialization steps as the mesh does not change during the simulation. For the processes in charge of the inner and outer r borders, boundary conditions are set up.

B. Block communication and OpenMP parallelization

Several optimizations for communications and for the parallelization of the computation can be done. Let us recall that for a process $P_{i,j}$, we have the data distribution $\mathbf{D}_2(r = r_i \rightarrow r_{i+1}, \theta = \theta_j \rightarrow \theta_{j+1}, \varphi = *, v_\parallel = *, \mu = \mu_{i,j})$; in our setting it means that every process has to compute the gyroaverage for $N_\varphi \times N_{v_\parallel}$ patches. Each process has only one μ, and the computation of the gyroaverage only depends on μ for the halo size as $\rho = \sqrt{2\mu}$. So μ will be considered fixed for the explanations that follow.

Until now, all the communications were performed during the transposition steps (see Algorithm 1), before and after the computation. The same could be done by exchanging the halos for every poloidal plane beforehand, but GYSELA is a highly memory consuming application and the gyroaverage operator is called within the part of the code where the memory peak is reached. Therefore, it is preferable to exchange the halos only when the corresponding planes are about to be processed. However, to reduce the initialization costs of each communication, it is also interesting to perform them by grouping the halos of several poloidal planes together. Thus, once the communication accomplished, several planes (a block) are ready to be gyroaveraged and they can be computed in parallel in a multi-threaded loop. The size of a block of halos can be tailored so that its memory footprint is not too important with regards to the memory limitation of the machine and so that the performance gains achieved by thread parallelization remains high. The steps of the computation of the gyroaverage are detailed in Algorithm 2.

During the initialization step 1, the local subdomains of the current block are copied in the temporary array f_{block} which is big enough to store the local subdomains plus their halos received during the communication step 2. Step 4 performs the backward operation, retrieving the gyroaveraged local subdomains and storing them back into the function f as well as gathering data *via* the post-process for the diagnostic. For a given i_{block}, the v_i and φ_i refers to the $v_\parallel$ and φ coordinates of the planes composing the block according to the formula

$$v_i = modulo(i_{block} \times bs + i,\ N_\varphi)$$
$$\varphi_i = (i_{block} \times bs + i) \div N_\varphi$$

where bs is the size of a block. For future reference, we will denote $N_{block} = \frac{N_\varphi N_{v_\parallel}}{bs}$.

In step 3, the gyroaverage operator used is the same as the one described in Section II-B, but it is applied to a function whose size corresponds to the local subdomain size plus the halo. In our implementation, we ensure that each plane of the block is initialized by the thread which gyroaverages it in order to maximize memory locality and affinity. The dimension sizes and the number of threads being both powers of two, there is no thread left with more work than another, leading to an ideal load balancing.

Algorithm 2: Halo based gyroaverage by block

Data: Distribution function f, block size bs, N_{ghost_r}, N_{ghost_θ}, N_{lr} and $N_{l\theta}$
Result: Gyroaveraged distribution function $\mathcal{J}_0.f$
begin
  f_{block} = `array` $(bs,\ N_{lr} + 2N_{ghost\_r},\ N_{l\theta} + 2N_{ghost\_\theta})$
  for $i_{block} : 0 \rightarrow \frac{N_\varphi N_{v_\parallel}}{bs} - 1$ **do**
    OpenMP parallel zone
      for $i : 0 \rightarrow bs - 1$ **do**
1       $f_{block}(i)$ = `preprocess` $(f(r_{min} \rightarrow r_{max}, \theta_{min} \rightarrow \theta_{max}, \varphi_i, v_i, \mu))$
2   `send_receive_halo` (f_{block})
    OpenMP parallel zone
      for $i : 1 \rightarrow bs$ **do**
3   `gyroaverage` $(f_{block}(i))$
4       $f(r_{min} \rightarrow r_{max}, \theta_{min} \rightarrow \theta_{max}, \varphi_i, v_i, \mu)$ = `postprocess` $(f_{block}(i))$

Concerning the communication step 2, one process exchanges data with the processes which are before and after it[2] in r (down and up), with those before and after him in θ (left and right) and finally with the four other neighboring processes "in the corners". The number of communications can be reduced from eight to four by avoiding the corners using the scheme pictured at Figure 3. For readability, only the r down and θ left phases have been pictured. First, each process sends and receives the requested data to its neighbors in r (in green), second it sends and receives its data in θ plus some of those received during the previous step (in red). Thus the communications with the processes located in the corners are avoided at the cost of a synchronization in the middle of the communication phase. The communications are carried out using the `MPI_Sendrecv()` routine to send the halos for all the planes of a block in one step. A communication scheme using non-blocking MPI routines has also been evaluated, but

[2]If the process is in charge of a r border, a boundary condition is applied for the corresponding part of the halo instead.

proves to be less performing in the benchmark we conducted. Only the solution using the blocking routines is presented here.

In the end, the new implementation is expected to be faster as the communication costs are reduced compared to the original version based on transposition though the computational cost is higher. The memory footprint is also reduced as the function distribution was previously fully duplicated whereas now, we only need one buffer $f_{block}(i)$ with relatively small size.

C. Performance results

In the following, the performance of the new solution, described in Section III-B, is compared to the one of the original approach of Section II-B.

The simulations presented in this section were performed on the Poincaré cluster located at Maison de la Simulation, France. Nodes are composed of two Intel(R) Xeon(R) E5-2670 with 8 cores and 32GB of shared memory each. In this study, we consider $N_{\mathcal{C}}$ constant and equal to 8 as it is the value which is used in usual production runs.

The free parameter of the new parallel algorithm is the block size, *i.e.* the number of poloidal planes for which halo exchange is performed in one communication and for which computation is parallelized at thread level. It is interesting to study the block size which allows to achieve the best performance. As shown in Figure 4, the optimal block size depends on the number of processes and on the value taken by μ. For instance, the optimal size for 4 processes in r and θ and $\mu = 4.0$ is 64, when for $\mu = 8.0$, it is 16. If now we switch to 2 processes in r and θ, the optimal block size is above 512 for $\mu = 8.0$. In fact, for a given mesh size, a greater number of processes means smaller subdomain sizes and thus less computation time with regards to communication time. Conversely the larger μ, the heavier the communication costs for the halo weighs in the gyroaverage operator; indeed μ is related to the Larmor radius *via* the relation $\rho = \sqrt{2\mu}$ and thus to the halo size. Hence, for production runs with multiple μ, there are as much optimal block sizes as different values of μ. It could be interesting to consider the possibility to have different block sizes depending on the value of μ in the code. Nevertheless, the MPI synchronization, located at the end of the diagnostic code we are focusing on, would make it useless as the final time will be dominated by the slowest MPI group which will be the one in charge of the largest value of μ. Ultimately, the main problem to solve is to determine the optimal block size for the group of largest μ.

Figure 5 shows the execution times of the gyroaverage operator for the previous version and for the new one. The mesh size is $(1024 \times 1024 \times 64 \times 32 \times 1)$ with $\mu = 4.0$ and a block size equal to 128. The number of cores is changed by increasing alternatively the number of MPI processes along r and θ. The number of threads per process is constant and equal to 8. With a small number of cores the halo based version is almost twice as fast. However it loses in scalability with the number of cores compared to the transposition version. This is mainly due to the decrease of the work load for each block

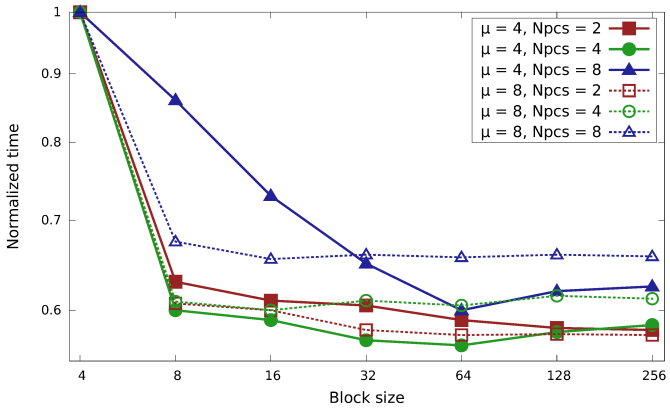

Figure 4: Normalized execution times for a mesh size of $(512 \times 512 \times 64 \times 31)$ according to the block size for $\mu = 8$ and $\mu = 4$. Each curve stands for a different number of MPI processes ($N_{\mathtt{pcs}} = N_{\mathtt{proc}\theta} = N_{\mathtt{proc}r}$).

whose cost becomes lower than its associated communication cost as explained below.

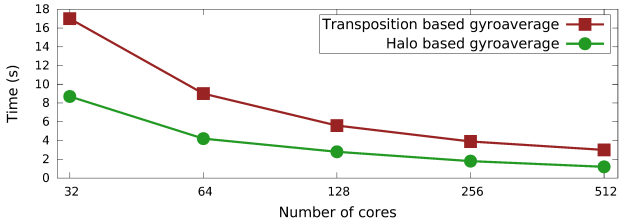

Figure 5: Strong scaling and comparison of execution time of the Hermite gyroaverage based on transposition and the gyroaverage based on halo exchange.

In Figure 6 the ratio between communication and computation time in the gyroaverage operator is given according to the size of blocks for a given mesh and number of threads. Though relatively big compared to the communication time for small blocks, the computation time becomes less predominant along with the size of the blocks. These results tend to show an equal amount of communication and computation time for large numbers of cores. Thus, it would be interesting to be able to absorb these communication costs by performing them concurrently with the computation.

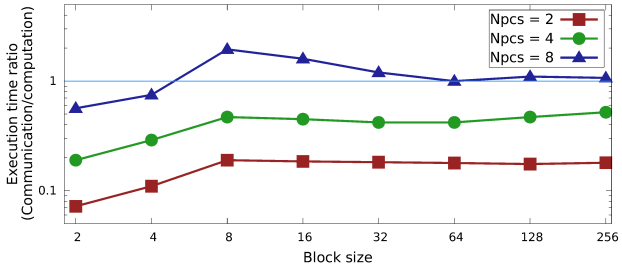

Figure 6: Comparison of the ration between communication and computation time for different number of MPI processes according to block size.

IV. Communication and computation overlapping

This section details how the performance of the gyroaverage is improved by overlapping communications and computations

in the operator. It is essentially performed through finer grain parallelization, one thread doing communication, while the other threads perform the computation.

A. Algorithm, complexity and expected speedup

As seen in Section III-C, large production simulations usually show communication times and computation times which are relatively close one to each other. It is then possible to further improve the algorithm and decrease execution time by performing communications and computations simultaneously. Similar and more complete analyses are performed in [3] and solutions for a good calibration of the parameters of the overlapping are suggested.

Using the improved blocked version of the Hermite gyroaverage (Section III-B), the idea is to start the initialization and communication of the next block while performing the computation of the current one. The different steps are detailed in Algorithm 3. Step 1 builds the different blocks to be processed. Step 2 consists in the initialization and communication of the current data block and step 3 is the computation and post-processing of the previous data block. There is one more iteration in the i_{block} loop than the total number of blocks so that the first iteration only performs the communication for the first block to initialize the macro-pipeline.

Algorithm 3: Patched gyroaverage with overlapping

Data: Distribution function f, block size bs
Result: Gyroaveraged distribution function $\mathcal{J}_0.f$
begin
  $L_{\varphi v_\parallel} = \{\varnothing\}$
1 **for** $i_{block} : 0 \rightarrow \frac{N_\varphi N_{v_\parallel}}{bs} - 1$ **do**
    $L_{tmp} = \{\varnothing\}$
    for $i : 0 \rightarrow bs - 1$ **do**
      $v_i = modulo(i_{block} \times bs + i,\ N_\varphi)$
      $\varphi_i = (i_{block} \times bs + i) \div N_\varphi$
      $L_{tmp} = L_{tmp} \oplus f(r_{min} \rightarrow r_{max}, \theta_{min} \rightarrow \theta_{max}, \varphi_i, v_i, \mu)$
    $L_{\varphi v_\parallel} = L_{\varphi v_\parallel} \oplus \{L_{tmp}\}$
  OpenMP parallel zone
    for $i_{block} : 0 \rightarrow \frac{N_\varphi N_{v_\parallel}}{bs}$ **do**
2   **Task 1**
        if $i_{block} \neq \frac{N_\varphi N_{v_\parallel}}{bs}$ **then**
          preprocess$(L_{\varphi v_\parallel}(i_{block}))$
          async_send_receive_halo$(L_{\varphi v_\parallel}(i_{block}))$
3   **Task 2**
        if $i_{block} \neq 0$ **then**
          wait_comm$(L_{\varphi v_\parallel}(i_{block} - 1))$
          gyroaverage$(L_{\varphi v_\parallel}(i_{block} - 1))$
          posprocess$(L_{\varphi v_\parallel}(i_{block} - 1))$

There are three ways algorithms with overlap can behave according to the relative size of the different execution times. Either the computation of a block and the communication of a block have the same duration (a), or communication is longer than computation (b), or computation is longer than communication (c). This is well depicted in [1]. We consider preprocess and postprocess times to be included in the communication and computation times, respectively, assuming they are negligible.

The ideal behavior is for cases where communication and computation have the same execution times (a). In this case the global execution time can be decreased by almost a factor 2. But if communication (b) (resp. computation (c)) times are too important compared to computation (resp. communication) times, the gain can be drastically reduced. This can be numerically assessed by estimating the complexity of the different stages of the diagnostic. Considering the costs of preprocess and postprocess negligible compared to the communication and computation ones, the cost of the gyroaverage operator with overlapping can be written

$$\mathcal{C}(\mathcal{J}_{overlap}) = N_{block}\max(\mathcal{C}_{comm}, \mathcal{C}_{comp}) + \min(\mathcal{C}_{comm}, \mathcal{C}_{comp})$$

where $N_{block} = \frac{N_\varphi N_{v_\parallel}}{bs}$ is the number of blocks and $\mathcal{C}_{comm}$ and $\mathcal{C}_{comp}$ are the costs of communicating and computing one block (which depend on the size of a block). Let the cost of communicating a message of size n be $\beta + n\tau$ (β start-up time, τ latency) and γ be the cost of gyroaveraging one plane. Then $\mathcal{C}_{comm} = \beta + bsN_{\mathcal{H}}\tau$ and $\mathcal{C}_{comp} = \gamma bs$. The optimal number of blocks can then be deduced analytically, but requires the knowledge of the network hardware typical times as well as the effective time needed to compute the gyroaverage on a block (also depending on the thread parallelization efficiency). Such a study is conducted in [3]. However, if we make the simplifying assumption that these two costs are linearly dependent on the block size ($\beta = 0$), the maximal expected speedup obtained with the overlap can be easily computed. According to III-B, the cost of the halo exchange version of the Hermite gyroaverage is $\mathcal{C}(\mathcal{J}_{halo}) = N_{block} \times (\mathcal{C}_{comm} + \mathcal{C}_{comp})$.

Let $\alpha = \frac{\mathcal{C}_{comm}}{\mathcal{C}_{comp}}$. Then the speedup between the halo and overlap version writes

$$\frac{\mathcal{C}(\mathcal{J}_{halo})}{\mathcal{C}(\mathcal{J}_{overlap})} = \frac{1+\alpha}{\frac{\alpha}{N_{block}} + \delta} \tag{2}$$

where $\delta = \alpha$ if $\alpha \leqslant 1$ and 1 otherwise. This equation shows first, that the speedup which can be expected from the overlap quickly decreases as communication time and computation time diverge, and second, that the speedup is maximal and equals to 2 for the ratio $\alpha = 1$. This result gives a coarse overview of the speedup which can be expected from the algorithm with overlapping.

B. Implementation

The implementation of the algorithm with overlapping is based on OpenMP thread parallelization. It requires MPI communications that can be performed in the background during computations. OpenMPI and IntelMPI, which are the main MPI implementations that GYSELA currently uses on most of the clusters, do offer non-blocking communication routines; however these are not really asynchronous. It means that the pending communications mainly progress whenever a MPI function is called [7]. An implementation with `MPI_Isend` and `MPI_Irecv` was first tried but quickly dropped as the communications mostly occurred during the `MPI_Wait`

though a lot of computation was done since the call to the send routine.

To get the expected overlapping behavior and design a portable approach, the "asynchronous" communications are performed by a dedicated thread (master). Moreover the loop scheduling of OpenMP has been set to dynamic, *i.e.* once a thread is done with its assigned loop iteration, it requests others to the scheduler. This way, no index of the parallelized loop is assigned beforehand to the master thread so the computation can be performed entirely even if the communications are longer. And in the case where the communications are shorter, this scheduling allows the master thread to join the computation loop once it has performed the communications. However, this could lead to a loss of cache locality as one plane is no longer pre-processed, computed and post-processed by the same thread. Indeed, OpenMP static scheduling ensures that a thread is assigned the same loop indexes in any loop which has the same first and last index, which is not the case for the dynamic scheduling. Figure 7 pictures the scheduling of the work load between the OpenMP threads in the case where the communication time for a block is shorter than its computation time (most common case).

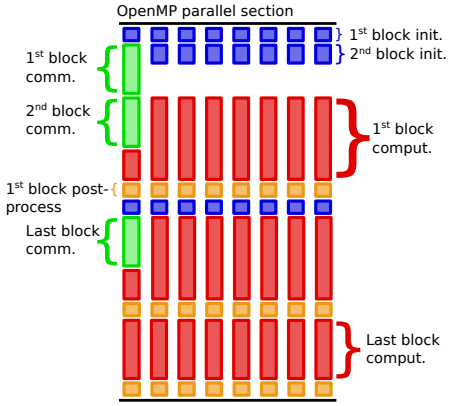

Figure 7: Behavior of the OpenMP threads in the main loop for the version of the algorithm with overlapping with $\alpha < 1$ considering three blocks.

C. Performance results

In the following, the performance of the solution with overlapping, is compared to that presented in Sections III-B and II-B. The dimensions of the mesh used are $(1024 \times 1024 \times 64 \times 32 \times 1)$. The performance detailed in this section were performed on the Helios cluster located at the IFERC center in Rokkasho, Japan which has an architecture very close to that of the Poincaré machine (Intel(R) Xeon(R) E5-2680 instead of E5-2680).

Tab. I shows how the version with overlapping of the gyroaverage scales with the number of cores and is compared to the halo version. The algorithm with overlapping is faster than the halo algorithm and scales in a similar way. The speedup results behave as expected in previous sections: the time gained over the halo version when the optimal block size of 64 (few blocks) is used is around 10%, and it reaches 100% with numerous small blocks of size 4. Moreover we see that the performance of the version with overlapping seems to be much less dependent on the block size. It allows us to use the same size for any value of μ (see Section III-C) without having to scan for each optimal block size beforehand.

Version and block size	Number of cores				
	64	128	256	512	1024
Halo version (4)	20.67s	10.16s	5.21s	2.83s	1.56s
Overlap version (4)	11.55s	5.73s	2.62s	1.39s	0.95s
Halo version (64)	12.23s	6.27s	3.02s	1.67s	1.01s
Overlap version (64)	11.39s	5.66s	2.65s	1.43s	0.98s

Table I: Scaling of the execution time of the gyroaverage operator for the halo and version with overlapping with different block sizes (between parenthesis).

Evaluating the efficiency of the improvements detailed in this paper on an overall GYSELA execution requires a test case with several μ values, similar to usual production runs. Table II shows the total time spent in the diagnostic in which the halo and overlap gyroaverage algorithms have been implemented. The execution covers 24 time steps, and the diagnostic is performed every 3 time step. The improvement achieved by the halo version is great but highly depending on the value of μ. Nonetheless, μ never takes a value above 16 in production runs and the halo version still demonstrates improvements at this point. One can also notice that when μ is small enough and is tending to zero, so that the Larmor circle only intercepts the four cells neighboring the point being gyroaveraged, the cost of the gyroaverage does not depend on μ anymore. Indeed, the number of cells used in the computation is at least four, leading to a constant amount of communication and computation. The 24 seconds of the transposition version for $\mu = 0$ are due to the transpositions which still needed to be performed for the diagnostic post-process though the gyroaverage function for $\mu = 0$ is the identity function. This is another benefit of the halo version.

Versions	μ values			
	0.	2.6667	5.3333	8.
Transp. version	24.53s	81.74s	80.71s	81.14s
Halo version	1.694s	36.91s	45.73s	51.75s
Overlap version	5.01s	32.66s	38.30s	44.40s

Table II: Total execution time of the 9 calls to the diagnostic on a typical production run with several μ values and for each version of the Hermite gyroaverage operator.

The performance gain achieved by the new implementations is effective, as the total execution time of diagnostic take 5.7% of the total time in the transposition version, down to 2.7% in the halo version and down to 2.4% in the version with overlapping.

The gain in terms of memory is also significant enough to be noticed. Indeed, in the transposition version, the transpositions were performed on a copy of the distribution function, thus making the memory footprint of same magnitude as the size of the function to be gyroaveraged. With the halo version, the memory cost of the operator is only the memory needed by one block of poloidal plane; given N_P MPI processes, this represents one N_P-th of the size of the function to be

gyroaveraged. It can even be smaller than the size of the matrices M_{coef} and M_{fval} (see Section II-B) used in the core operator depending on the size of the blocks and the number of processes in the poloidal plane. The version with overlapping uses twice as much memory because it stores two blocks of data in parallel for the communication-computation pipeline, but it has still a smaller footprint than the transposition version.

D. Discussion

In this paper we consider that the Larmor radius is way smaller than r_{min} which is a satisfactory condition in most cases, and the parallelization scheme detailed above gives good results in terms of speedup and memory usage. However, as an extension, we would like to consider the situation where r_{min} is so small that the numerical scheme becomes invalid, meaning that its value comes close or under the Larmor radius. Considering MPI processes on the inner border of the plane and the gyroaverage of their points located at (r_{min}, θ), the problem is that the Larmor circle will intercept cells further away than neighboring subdomains in θ direction. There are two ways for this to happen: either its radius ρ is larger than r_{min} (the circle then intercepts all the subdomains around the center and the computation requires points from all these processes), or the poloidal plane is divided between a large number of processes in θ (the circle intercepts two or more subdomains along θ direction). The current implementation does not take into account these specific cases. The problem does not occur in the r direction as there is a limit of at least 32 points in r per process in GYSELA and physically consistent conditions usually ensure that ρ does not exceed subdomain r width. A small r_{min} value is the most critical problem, as it is expected to be soon required in production runs. A convenient solution is, for processes in charge of the inner border of the plane, to share all their subdomains through an `MPI_AllGather` call. Thus, each process in charge of a subdomain starting at r_{min} knows all the data from the other processes of the r_{min} annulus and can compute all its own gyroaverages with the data received. However, this is a setback for performance as the distributed Hermite algorithm has been chosen to avoid large amounts of communication. Moreover, the special buffers required for this specific scheme add memory usage where it is the most critical and where we wanted to avoid it. A simple implementation of this algorithm (using `Allgather` on inner annulus of processes and halo on the rest of the plane) has been implemented on top of this paper's work and proves to be almost as costly as the initial full transposition algorithm. Indeed, communication costs are now proportionate to the size of the annulus ($N_{lr}N_{\theta}$) rather than to the halo size ($\alpha N_{lr}N_{l\theta}, \alpha < 1$). Same goes for the size of the right-hand side (computation of the derivatives) which computation cost is multiplied by $N_{\texttt{proc}\theta}$.

However, we can consider several more sophisticated options in order to adapt the behavior of the algorithm on the inner most processes without having to revamp the entire scheme. We could for instance have a local redistribution of data or we could reduce the number of points we use for interpolation, considering non-uniform grids. The problem will be fully addressed in future works.

V. CONCLUSION

A new parallel solution has been designed for the gyroaverage operator based on Hermite interpolation which is a key component of the semi-Lagrangian code GYSELA. The transpositions of the distribution function imply large communication costs, they have been replaced by an algorithm based on halo exchange. This leads to a significant reduction of the execution time spent into the gyroaverage operator, a gain of almost 40% for some settings, as well as a reduction of the memory footprint of the operator. Nevertheless, this approach requires to fix a free parameter, the block size, that impacts execution time.

On the other hand, an overlapping technique has been employed to further improve performance. The communication involved by halo exchange is overlapped with the gyroaverage computation of some previously received data. In this setting, execution times are better, down to 50% of the initial version. Execution times also depend more loosely on the block size which provides a large benefit over the previous solution.

These new parallel solutions bring an effective gain on production run execution times. They will also be integrated in other sections of the code, leading to further improvement of performance. In addition, it prepares for the future of GYSELA. Indeed, in [5], it has been shown that a 4D advection solver is able to perform valuable simulations. Combining the gyroaverage parallelization described in this paper with the 4D Vlasov solver would allow us to setup a version of the code that does not involve large transposition. In such a context, the new Hermite parallel gyroaverage solution paves the way for highly scalable gyrokinetic semi-Lagrangian solvers.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge use of IFERC center (Helios machine), Japan, as well as Maison de la Simulation facilities (Poincaré machine), France, and associated support services.

REFERENCES

[1] N. Bouzat, F. Rozar, G. Latu, and J. Roman. A new parallelization scheme for the Hermite interpolation based gyroaverage operator. Research Report RR-9054, Inria, Apr. 2017. https://hal.inria.fr/hal-01502513.

[2] N. Crouseilles, M. Mehrenberger, and H. Sellama. Numerical solution of the gyroaverage operator for the finite gyroradius guiding-center model. *Communications in Computational Physics*, 8(3):484, 2010.

[3] F. Desprez, P. Ramet, and J. Roman. Optimal grain size computation for pipelined algorithms. In *Euro-Par'96 Parallel Processing*, pages 165–172. Springer, 1996.

[4] V. Grandgirard et al. A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207:35 – 68, 2016.

[5] G. Latu, V. Grandgirard, J. Abiteboul, N. Crouseilles, G. Dif-Pradalier, X. Garbet, P. Ghendrih, M. Mehrenberger, Y. Sarazin, and E. Sonnendrücker. Improving conservation properties of a 5D gyrokinetic semi-lagrangian code. *The European Physical Journal D*, 68(11):1–16, 2014.

[6] F. Rozar et al. Optimization of the gyroaverage operator based on hermite interpolation. *ESAIM: Proc.*, 53:191–210, 2016.

[7] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous MPI for the Masses. *CoRR*, abs/1302.4280, 2013.