IPP-Report

Max-Planck-Institut für Plasmaphysik

Tiago Tamissa Ribeiro

NEMOFFT project: Improved Fourier algorithms for global electromagnetic gyrokinetic simulations

IPP 19/2
August, 2013

NEMOFFT project: Improved Fourier algorithms for global electromagnetic gyrokinetic simulations.

Tiago Tamissa Ribeiro[1, *]

[1] *Max-Planck-Institut für Plasmaphysik, EURATOM Association, Boltzmannstrae 2, 85748 Garching, Germany*

(Dated: August 2013)

The NEMOFFT project addresses a well-known parallel scalability bottleneck of the global gyrokinetic ORB5 code, in its current electromagnetic version NEMORB. This code is very HPC-resource demanding, especially for large-sized simulations. It relies on efficient parallel algorithms that require filtering to refine the physical quantities and maintain appropriate numerical signal-to-noise levels. The filtering is done via bi-dimensional Fourier transforms (FTs) that involve parallel matrix transposes and hence require large data transfers across cores. Such communication naturally impairs the code's parallel scalability and renders, in practice, ITER-sized plasmas simulations unfeasible. To cope with this, NEMORB's filtering algorithm is modified. The Hermitian redundancy inherent to its purely real data is invoked to effectively reduce the number of floating point operations (FLOPs) involved in the FTs by roughly a factor of two. Further using a clever storage for the resulting data (half-complex packed format) reduces the message size being communicated by the same amount. Several parallel transpose methods are investigated within two main HPC architectures, namely, Intel CPU-based with InfiniBand interconnect (HPC-FF at JSC in Jülich and HELIOS at IFERC-CSC in Aamori) and IBM Power6 (VIP at RZG in Garching). The best overall performer is found to be based on a XOR/`MPI_Sendrecv` algorithm. Its flexibility allows to exploit NEMORB's low pass filter to further reduce the inter-core data exchange, yielding speedups higher than two.

CONTENTS

I. Introduction 2

II. NEMORB's 2D Fourier transform algorithm 2
A. Domain decomposition 2
B. Multi-dimensional Fourier transforms 3
C. XOR/`MPI_Sendrecv` distributed transpose algorithm 4
D. Basic algorithm profiling 4

III. Index order swapping: local transpose 5

IV. Fourier transform of real data: Hermitian redundancy 6

V. Half-complex packed format to avoid zero-padding 8

VI. Distributed transpose: XOR/`MPI_Sendrecv` vs. `MPI_Alltoall` 10

VII. Alternative methods 13
A. Transposes from FFTW and MKL libraries 13
B. Alternative to the transpose: `MPI_Gather/Scatter` directives 14

VIII. Optimization of the `XOR/MPI_Sendrecv` transpose 17
A. XOR exchange pattern sequence 17
B. Fourier filtering and the partial transpose 18

IX. Conclusions 20

Acknowledgments 21

References 21

* email: tiago.ribeiro@ipp.mpg.de; URL: http://www.rzg.mpg.de/~ttr/

I. INTRODUCTION

The NEMOFFT project aims at removing, or at least alleviating, a well-known parallel scalability bottleneck of the global gyrokinetic particle-in-cell (PIC) code ORB5, in its current electromagnetic version NEMORB. This code is very HPC-resource demanding, especially for large sized-simulations. At each iteration, the particle charges are deposited on a spatial three-dimensional (3D) grid, which represents the source term in the Poisson equation (the right-hand side). Due to the strong spatial anisotropy in a tokamak, only a restricted set of degrees of freedom, or modes, are allowed to exist, namely, the ones which are either aligned or close to being aligned with the background guiding magnetic field. Using this physical restriction on the Poisson solver, not only reduces the required FLOPs (solve only for the allowed modes), but also improves the numerical signal to noise ratio, which is of central importance in a PIC code. In practice this amounts to calculating bi-dimensional (2D) Fourier transforms to apply the corresponding filters, but because the mesh domain of NEMORB is distributed across several cores, it requires large amounts of grid data to be transposed across cores. Such inter-core communication naturally impairs the code's parallel scalability and renders, in practice, ITER-sized plasmas simulations unfeasible.

In principle, there are two possible ways to deal with this issue. Either to change the spatial grid to be globally aligned with the equilibrium magnetic field, which eliminates the need for field-alignment Fourier filtering, or to improve the way the parallel data transpose and the Fourier transforms are done. While the former would completely eliminate the aforementioned scalability bottleneck, it is not transparent how such an approach could be implemented in the finite element basis used in NEMORB. It would certainly imply fundamental changes to the original code. Conversely, the latter, even though potentially less effective, can be implemented as an external library, with minimal changes to the original code required. Furthermore, since there are several other codes of the EU fusion program that share the same parallel numerical kernel with NEMORB (e.g. EUTERPE), they would also directly benefit from it. Therefore, this project focuses exclusively on the second approach.

The bulk of the work on improving NEMORB's 2D Fourier transform algorithm can be divided into two main parts. Namely, the exploitation of the Hermitian redundancy inherent to its purely real input data and the optimization of the distributed transpose algorithm. Both together should yield speedup factors of the order of two. The corresponding steps made to achieve this improvement, as well as the performance measurements made on different HPC machines are detailed in the remaining sections of this report, which is organized as follows. Sec. II provides a basic profiling of the original NEMORB's 2D Fourier transform algorithm and introduces its main constituent blocks, as well as the domain decomposition used. Sec. III focuses on the optimization made on the interface between the input data (from the main code) and the library containing the aforementioned algorithm. The changes that were made based on the Hermitian redundancy are detailed in Secs. IV-V, where the first performance results for the whole algorithm are showed. The comparison between the original transpose algorithms based on the XOR/`MPI_Sendrecv` and `MPI_Alltoall` directives is provided in Sec. VI, where the effect of using MPI derived data types is also assessed. Sec. VII discusses other alternative transpose methods, like the ones provided by BLACS via the Intel MKL library and the latest versions ($\geq$ 3.3) of the FFTW library. It also explores the usage of the `MPI_Gather/Scatter` directives to collect and distribute the data as a replacement for the transpose. The concept of using the high count of zeros resulting from NEMORB's Fourier filtering to reduce the communication burden in the transpose algorithm is explained and tested in Sec. VIII. Finally, a summary is provided in Sec. IX.

II. NEMORB'S 2D FOURIER TRANSFORM ALGORITHM

This section describes the fundamentals behind the NEMORB's original 2D Fourier filtering algorithm. This analysis constitutes the first step needed before any changes to the code are made.

A. Domain decomposition

NEMORB's 3D spatial grid comprises the radial, poloidal and toroidal directions, discretized with N_s, N_χ and N_ϕ grid-nodes, respectively. Such domain is then decomposed into N_{cart} subdomains over the toroidal direction and distributed across the same number of cores (Fig. 1), typically as many as toroidal grid-nodes. This naturally limits the number of tasks over which

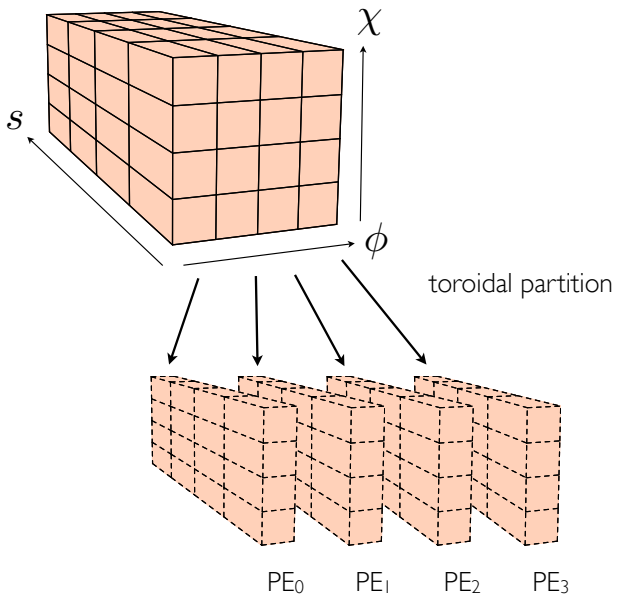

FIG. 1. Illustration of the spatial grid parallelization of NEMORB, with the sub-domains distributed in the toroidal direction for the case $N_{\mathrm{cart}} = 4$.

the problem can be parallelized to $N_{\mathrm{cart}} = N_\phi$. To overcome this limit, the domain is further decomposed into N_{C} clones, and each of them gets $1/N_{\mathrm{C}}$ of the total number of particles [1]. This implies that the total number of parallel tasks is given by $N_{\mathrm{cart}} \times N_{\mathrm{C}}$. For the purposes of this work, the filtering operations occur independently within each clone. Therefore, the following analysis focuses on a single clone only. Since all the clones are exactly equivalent, the results herein obtained are directly applicable to the remaining ones.

Before proceeding, a few remarks on the choice of the toroidal direction for the spatial decomposition direction instead of the poloidal one are in order. Since in a tokamak the toroidal direction is typically very close to the parallel direction, NEMORB approximates the gyro-radii, which occur in the plane perpendicular to the magnetic field, with their projection on the poloidal plane. Hence, all gyro-radii are local at a given toroidal location. If the domain decomposition was in the poloidal direction instead, this would no longer be the case, as there would necessarily be some gyro-radii shared by two adjacent domains, which would greatly increase the communication between them. Therefore, the reason for the choice to parallelize over the toroidal angle is the minimization of inter-core communication.

B. Multi-dimensional Fourier transforms

The basis functions for Fourier transforms are combinations of sines and cosines, which do not have a compact support (they are not spatially localized). Therefore, for numerical calculation of discrete Fourier transforms (DFTs), data locality is of key importance. The same obviously applies to a multi-dimensional DFT, which directly regulates its parallelization and data distribution concepts. In particular, an N-dimensional DFT can be obtained from N one-dimensional (1D) DFTs computed sequentially, one for each of the N dimensions. Since, as discussed before, each of these 1D transforms is a non-local operation, for the sake of communication efficiency, all the corresponding input data should be locally available on the core doing the computation. For the case of NEMORB, a 2D DFT on the toroidal and poloidal angles needs to be computed. The toroidal decomposition of its spatial domain (Fig. 1) sets the poloidal direction as the first one to be (1D) Fourier transformed, since the corresponding data is locally available to each core. Subsequently, the toroidal Fourier transform must be carried out, but since the corresponding data is now distributed across the cores, it must first be made local to each of them. This is done with a matrix transpose that swaps the toroidal and poloidal data, as is explained next, in Sec. II C.

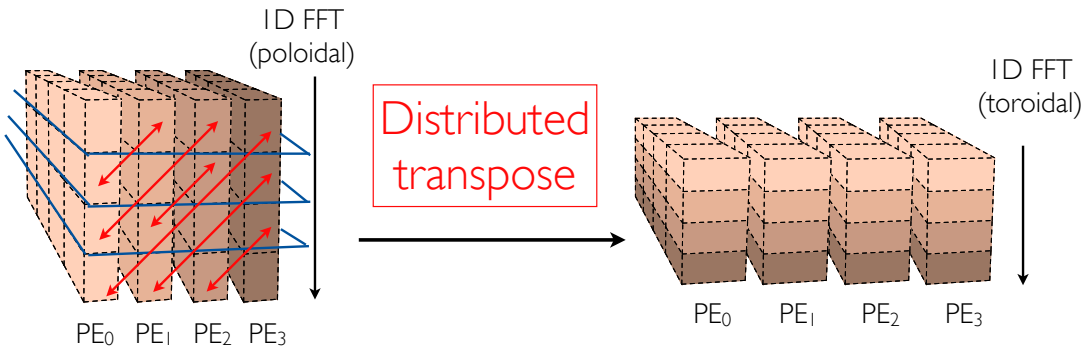

FIG. 2. Illustration of the distributed 2D FFT algorithm. It consists of a 1D FFT along the poloidal (local) direction, followed by a transposition between poloidal and toroidal (distributed) directions and lastly by a 1D FFT in the toroidal direction.

C. XOR/`MPI_Sendrecv` distributed transpose algorithm

The original NEMORB's distributed matrix transpose algorithm, written by Trach-Minh Tran (Centre de Recherches en Physique des Plasmas – CRPP), implements the necessary all-to-all communication pattern via a pairwise exchange algorithm. First, the $(N_\chi \times N_s \times N_\phi/N_{\text{cart}})$-sized data local to each core, represented by the slabs in Figs. 1 and 2, is sub-divided into as many sub-blocks or "pencils" as there are toroidal sub-domains (cores), i.e. N_{cart}. This corresponds to the blue cuts in Fig. 2, which yield sub-blocks of size $N_\chi/N_{\text{cart}} \times N_s \times N_\phi/N_{\text{cart}}$. Then, an "exclusive or" (XOR) condition establishes the exchange pattern of the sub-blocks across all cores. Excluding the diagonal sub-blocks for simplicity, since they stay local to the core they belong to initially, this pattern yields a set of $(N_{\text{cart}} - 1) \times N_{\text{cart}}/2$ pairs of sub-blocks to be swapped, as illustrated by the red arrows in Fig. 2. Furthermore, these pairs are organized into N_{cart} sub-groups in a non-overlapping manner. Therefore, the point-to-point data exchanges corresponding to all pairs within such a sub-group can proceed in parallel through `MPI_Sendrecv` calls, without race conditions. After sequentially going through all these N_{cart} sub-groups of pairs, the whole matrix transpose is achieved. This yields an extremely efficient all-to-all communication algorithm, at least for the problem sizes under consideration within this work, as shall be seen later in this report. As a limitation, stemming from the pair-wise organization of its communication pattern, this method implies the use of powers-of-two for the domain decomposition, and hence for the grid-count. In practice, this constitutes no drawback in NEMORB since these are the optimum array sizes for fast Fourier transforms. Moreover, it should be mentioned that there is in NEMORB an alternative transpose pattern based on the `modulo` function, which is meant for non-power-of-two cases, but which was not tested within this project.

The internals of the XOR/`MPI_Sendrecv` transpose method are revisited in detail in Sec. VII, when alternative MPI collective communication directives are discussed. Further details are also given in Sec. VIII B, within the framework of using NEMORB's Fourier filtering to exclude some of the communication-pairs from the full exchange pattern, such to yield a more efficient tailored distributed transpose.

D. Basic algorithm profiling

Before proceeding to the optimization of the algorithm in hand, it is necessary to measure the cost of each of its components. The major ones have already been presented in the previous two sections, but an overview of all the steps involved is still missing. This is the purpose of the following lines.

The initial step prepares the input data to be Fourier transformed. Since it corresponds to the right-hand side of the Poisson equation, it is therefore real-valued. Because the complex-to-complex (c2c) Fourier transform algorithm is used, the real-valued input data array is copied to a new complex array with the same size and imaginary part set to zero. Additionally, it has the first and second indexes exchanged, from the original order (s, χ, ϕ) to (χ, s, ϕ). The reason for this is simply that NEMORB's subroutines performing the 2D DTFs act on the 1st and 3rd indexes

of a 3D complex array. This operation of index-swapping amounts in practice to a local matrix transposition (all the data is locally available) and constitutes the first place where optimization can be applied. The relative cost of this compared to the whole algorithm can be checked in the profiling Table I, where it corresponds to the `Reord` entry. The next steps involve calculating the Fourier transforms. First in the poloidal (local) direction for all radial and toroidal grid-nodes. This is done via a standard c2c fast Fourier transform (FFT, a particularly efficient method to calculate DFTs for power-of-two sized arrays [2]) and called `FTcol1` in Table I. The result is stored in a 3D complex matrix with size $N_\chi \times N_s \times N_\phi$ that is distributed across N_{cart} cores over the last dimension. It further needs to be Fourier transformed in the toroidal direction, but since the corresponding data is now distributed, it must first be made local to each core. Such task is achieved using the transpose algorithm outlined in Sec. II C, which in Table I is denominated by `Transp`. The resulting transposed 3D complex matrix with size $N_\phi \times N_s \times N_\chi$, also distributed across N_{cart} cores over the last dimension (here the poloidal direction), can now be efficiently Fourier transformed in the toroidal direction using the same c2c FFT algorithm as before. Its cost is measured by `FTcol2` in Table I. To have a more pictorial idea about the bulk part of the algorithm, Fig. 2 schematizes the last three steps just described in this paragraph.

		Inclusive			Exclusive		
Subroutine	#calls	Time(s)	%	MFlops	Time(s)	%	MFlops
Main	1	72.392	100.0	614.469	0.014	0.0	4.990
:Reord	500	5.690	7.9	0.002	5.690	7.9	0.002
:FTcol1	500	8.303	11.5	2745.803	8.303	11.5	2745.803
:Transp	500	47.634	65.8	0.199	47.634	65.8	0.199
:FTcol2	500	7.357	10.2	2803.278	7.357	10.2	2803.278
:Normal	500	3.395	4.7	310.034	3.395	4.7	310.034

TABLE I. Performance measurement of NEMORBs original 2D FFT algorithm on a typical ITER-sized spatial grid $N_\chi = 2048$, $N_s = 512$ and $N_\phi = 1024$, distributed over 1024 cores on HPC-FF. The "hotspot" is the distributed transpose, highlighted in red.

Table I shows the performance measurement of the algorithm when executed 500 times on a typical ITER-sized spatial grid, $N_\chi = 2048$, $N_s = 512$ and $N_\phi = 1024$, distributed across 1024 cores on the HPC-FF machine, at the Jülich Supercomputer Centre (JSC). Clearly, the distributed matrix transpose represents the most significant part (65%) of the whole time cost of NEMORBs 2D FFTs. This percentage can even increase for smaller exchange block array sizes, as shall be discussed in more detail later. Since this part is mostly inter-core communication (note the negligible MFLOPs compared to the 1D FFTs), it implies that the whole algorithm is communication-bound. Most of the optimization effort has to be put there, either directly or indirectly. Nevertheless, since the index-swapping part (`Reord`) offers complementary room for improvement, that shall be the starting point addressed in the remaining text.

III. INDEX ORDER SWAPPING: LOCAL TRANSPOSE

The index order of the data in the main NEMORB code is (s, χ, ϕ), but its Fourier transform subroutines require (χ, s, ϕ). In practice, this amounts to performing a local transpose between indexes 1 and 3 on the code's 3D data array, which is done in the original code via two nested `DO`-loops. Such method is compared to both FORTRAN intrinsic and Intel MKL (`mkl_domatcopy`) local transpose counterparts for several matrix sizes. The former comparison yielded similar figures (not shown here) for both methods for matrices with 500×500 to 2000×2000 elements. The latter comparison is exemplified in Fig. 3, which shows the results obtained for the same matrix-size range on HPC-FF at the JSC. It shows the speedup achieved when using the MKL transpose over the nested `DO`-loops on real-valued data matrices.

For these matrix sizes, which are in the range of the ones used in NEMORB for an ITER-sized simulation, the MKL transpose is always faster. However, it should be noted that this result is not generalisable for all matrix sizes. The speedup pattern can be quite complex, with degradation or even slowdown occurring for specific matrix sizes. Two final remarks in this topic are in order. The first addresses the fact that a real-array is used here for the tests and not a complex one, as the material in Sec. II D might have suggested. The reason provides a hint to the material of the

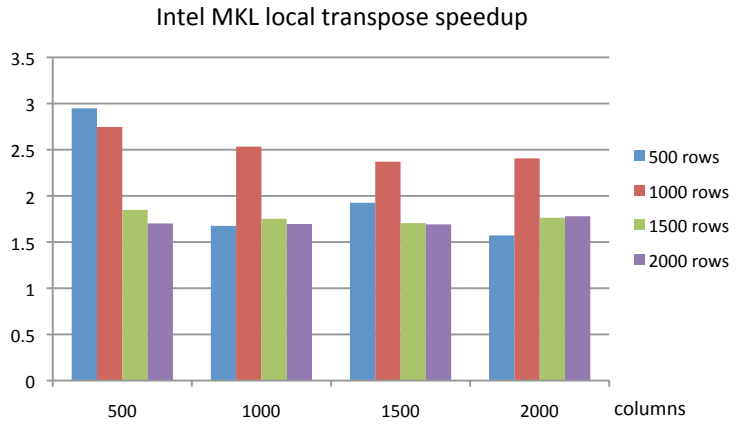

FIG. 3. Intel MKL (`v10.2.5`) local transpose (`mkl_domatcopy`) speedup relative to a `DO`-loop transpose (ratio of the time cost of the latter over the former).

next section, namely, the use of real-to-complex (r2c) FFTs to improve both CPU and memory efficiency. The second remark relates to eventual cache-line trashing due to usage of powers-of-two-sized arrays [3]. No tests were made regarding this subject, but the suggestion to do so in the future is left here.

IV. FOURIER TRANSFORM OF REAL DATA: HERMITIAN REDUNDANCY

As already hinted in Sec. III, the optimization plan for NEMORB's Fourier transforms invokes the Hermitian redundancy inherent to its real-valued input data (right-hand side or source term of Poisson's equation). Specifically, the idea is to use the following property. Given a N-sized real-valued discrete data series f_k with the integer domain-index $k \in [0, N-1]$, its discrete Fourier transform

$$F_n = \sum_{k=0}^{N-1} f_k e^{2\pi i k n/N} \tag{1}$$

exhibits the following symmetry

$$F^*_{N-n} = F_n \tag{2}$$

where the asterisk represents the complex conjugate and $n \in [0, N-1]$ is the frequency or mode index. Such symmetry implies that the DFT can be unequivocally represented with only $N/2+1$ independent complex values, as illustrated in Fig. 4. Taking this property into account allows to calculate DFTs of real data in a more efficient manner compared the full c2c transform, both memory- and CPU-wise.

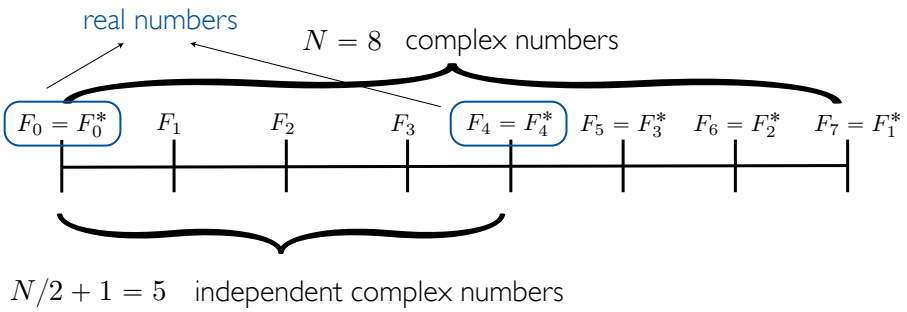

FIG. 4. Illustration of the Hermitian redundancy of the discrete Fourier transform of real valued data with size $N = 8$.

In NEMORB, the Hermitian redundancy can be used in the first array index to be Fourier transformed, namely, the poloidal direction (χ). This effectively reduces the number of needed FLOPs by roughly a factor of two. The same applies to the size of the transformed stored data-set, which for the N_χ-sized real-valued input array, falls from the full N_χ complex Fourier modes to a $N_\chi/2+1$ subset of them. The subsequent Fourier transform, in the toroidal direction (third array index), acts upon the already poloidally Fourier transformed data, which is complex. Therefore, it no longer exhibits the Hermitian redundancy and so, for each poloidal Fourier mode, the full c2c toroidal Fourier transform has to be carried out. However, since the number of poloidal Fourier modes was cut by roughly a half before, so is the number of c2c toroidal Fourier transforms that needs to be calculated. Together this leads to an expected speedup factor of two for the whole of the Fourier transform operations (excluding the transpose) in NEMORB's 2D filtering algorithm.

In practice, there are two possibilities to use the Hermitian redundancy in the poloidal direction. Either (i) store two poloidal (real) arrays as the real and imaginary parts of a same-sized complex array, compute the complex FFT and use the symmetry relations Eq. (2) to separate the two transforms, or (ii) store one poloidal array in an half-sized complex array in an appropriate manner (even indexes go to the real part, and odd indexes go to the imaginary part), compute the complex FFT and use the symmetry relations above to separate the two transforms. In terms of floating point operations, they are both equivalent, but they affect differently the distributed transpose. The former computes two poloidal arrays simultaneously, so it effectively reduces the matrix size to be transposed from the original $N_\chi \times N_s \times N_\phi$ to $N_\chi \times N_s \times (N_\phi/2+1)$. The latter stores the first 1D FFT in a half-sized complex array (half-complex format), so the matrix being transposed is now roughly of size $(N_\chi/2+1) \times N_s \times N_\phi$. While the former approach is more straightforward to code, it needs at least two poloidal arrays to be local at each core, preventing the cases with $N_{\rm cart} = N_\phi$, which is undesirable. The second method does not suffer from this issue, so it was chosen. Besides, since it is readily available in standard FFT libraries (e.g. FFTW and Intel MKL), there is no need to implement it explicitly. One simply calls the real-to-complex (r2c) transforms of such libraries.

As mentioned already, a direct consequence of using half-complex poloidal Fourier transforms is that, due to the effective reduction in the size of the data-set, less information needs to be exchanged across cores. Therefore, a similar two-fold performance gain is expected from the distributed matrix transposition part of the algorithm. Putting everything together, including the index swapping optimization described in Sec. III, leads to the performance results shown in Fig. 5 for the whole 2D FFT algorithm. This plot shows the scaling speedup factors achieved with the new method compared to the original one, on a grid count of $2048 \times 512 \times N_{\rm cart}$, in the poloidal, radial and toroidal directions, respectively, which keeps the amount of data per core constant. Therefore, these are pure weak scaling studies for the communication part of the algorithm (transpose), but not for the computing cost since the amount of work load per core varies as the size of the toroidal FTs increases with $N_{\rm cart}$. This defines the meaning of semi-weak scaling in the following. The measurements were made on the HPC-FF machine at JSC.

The several components involved in the algorithm are measured separately. The two main ones are the FFT calculations on both angles (green) and the parallel transpose (red). The blue line

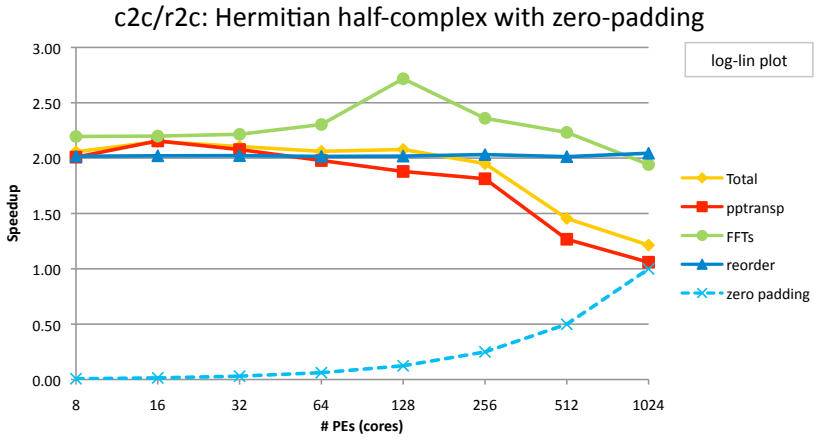

FIG. 5. Semi-weak scaling speedup (ratio between elapsed times) factors achieved on HPC-FF for each component of the 2D FFT algorithm when using the half-complex representation for the poloidal direction.

represents the local index-order swapping of the input data array that was discussed in detail in Sec. III. The yellow line represents the overall speedup factor. It follows closely the red line since, as shown in Table I, the transpose represents the main part of the overall computational costs. Up to 256 cores a factor of two is gained, but beyond that degradation is observed. The reason is that this method requires zero-padding in the poloidal direction to be compatible with the parallel transpose algorithm. The latter requires the poloidal direction of the matrix being transposed to be divided into $N_{\rm cart}$ blocks. Therefore, its number of rows must be a multiple of the number of cores ($N_{\rm cart}$), and when that is not the case, extra rows of zeros must be added. Knowing that the number of rows is given by $N_\chi/2+1$, due to the Hermitian redundancy, for small $N_{\rm cart}$ such condition is easy to fulfill with a small number of extra rows of zeros. As $N_{\rm cart}$ gets bigger, more and more zero-padding is necessary, as the light-blue curve shows. It measures the ratio between the number of extra zero rows added and the original $N_\chi/2+1$ size. In the limiting case of $N_\chi = N_{\rm cart} = 1024$, one needs to add 1023 extra rows of zeros to the initial $N_\chi/2+1 = 1025$ rows, for a total of 2048 rows. This is the same size as the original matrix that neglects the Hermitian redundancy. Hence, for this case there should be no speedup of the transpose algorithm, as the red curve confirms. However, it is important to realize that the zero-padding is, in most cases, simply due to the extra element on top of the remaining $N_\chi/2$ elements that make the Hermitian half-complex representation. For instance, without this extra array element, there would be no need for zero-padding up to $N_{\rm cart} = 1024$ for the same poloidal grid-count ($N_\chi = 2048$). This is the basis behind the work-around for this issue, which is described in the next section.

V. HALF-COMPLEX PACKED FORMAT TO AVOID ZERO-PADDING

As seen before, the Hermitian redundancy was used in NEMORB to improve the performance of its original 2D Fourier transforms by a factor of two. However, in the most general case, this did not hold due to the zero-padding needed by the transpose algorithm. This section explains how such can be circumvented invoking another property of the Hermitian redundancy, namely, that for even N, which is the case for NEMORB, the zero (F_0) and Nyquist ($F_{N/2+1}$) Fourier modes are purely real. Therefore, without any loss of information, the real part of $F_{N/2+1}$ can be stored in the imaginary part of F_0, as illustrated in Fig. 6. This is called half-complex packed format and allows to reduce the number of elements in the poloidal direction by one, from $N_\chi/2+1$ to $N_\chi/2$. Converting to this storage format before performing the transpose effectively avoids the need for zero-padding (provided that $N_\chi > N_{\rm cart}$) and should therefore eliminate the degradation seen in Fig. 5 for high numbers of MPI tasks.

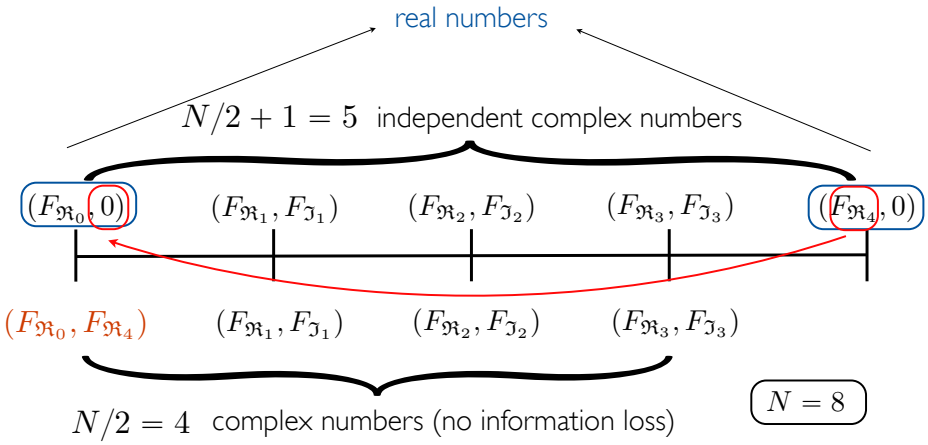

FIG. 6. Illustration of the half-complex packed format.

Obviously, after the data transposition, before the second (toroidal) Fourier transform can be computed, the data must be converted back to the "un-packed" half-complex format. Since the poloidal direction is at this stage distributed across different cores, this task requires an additional point-to-point communication between core rank 0, which needs to send the $F_{N/2+1}$ value to core $N_{\rm cart}-1$, where it should be stored. This is done with an additional call to the `MPI_Send` and `MPI_Recv` directives at no significant relative extra cost, as shall be seen next. As a final remark,

it should be mentioned that, naturally, all the cases with $N_\chi \leq N_{\text{cart}}$ need zero-padding when the Hermitian redundancy is used, even with packed format representation. The reason is simply that for those cases, $N_\chi/2$ is always smaller that N_{cart} and to use the parallel transpose algorithm, $N_{\text{cart}} - N_\chi/2$ rows of zeros need to be added. Hence, no speedup is expected for such spatial domain distribution settings. These are, however, uncommon in tokamaks, where the typical poloidal to toroidal field winding ratio is larger that one ($q > 1$), which implies $N_\chi > N_\phi$ and therefore also $N_\chi > N_{\text{cart}}$.

Using the half-complex packed format on the same example of Sec. IV ($N_\chi = 2048$, $N_s = 512$ and $N_\phi = N_{\text{cart}}$) requires no zero-padding up to $N_{\text{cart}} = 1024$ cores. The reason being simply that $N_\chi/2 = 1024$ is always a multiple of N_{cart}, provided that the latter is a power of two, as is always the case in NEMORB. The semi-weak scaling study of Fig. 5 was repeated under these circumstances. Since the changes affect only the transpose part of the algorithm, which moreover is its bulk piece cost-wise, the comparison of the corresponding total curves is sufficient, as shown in Fig. 7. As expected, the total speedup is always the same or higher (green curve) than before (yellow curve), even for the cases that originally required very small zero-padding percentages. This means that the overhead communication costs related to the packed format conversion are negligible. For higher numbers of cores (N_{cart}), which before required a substantial amount of zero-padding, the gain is clear and arises from having less amount of communication to do (smaller matrices to transpose). Still, for the two highest N_{cart} values, there is a degradation of the two-fold speedup. This can not be attributed to the matrix size increase (relative to the original full-complex matrix), as there is no zero-padding involved in these cases whatsoever.

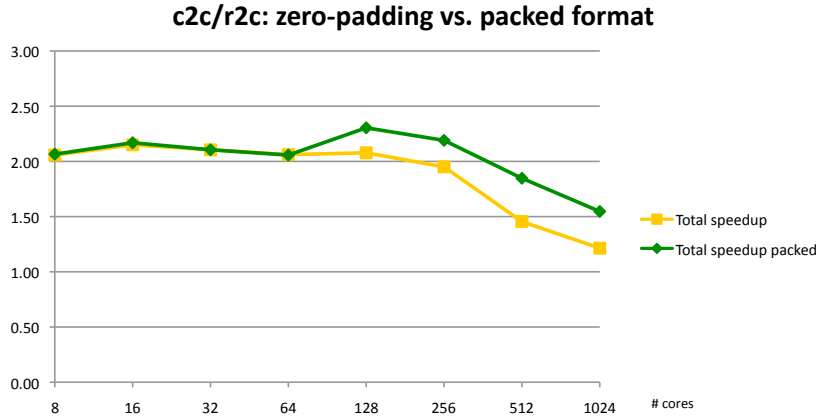

FIG. 7. Semi-weak scaling speedup factors achieved for the 2D FFT algorithm with (green) and without (yellow) the half-complex packed format representation for the poloidal direction on HPC-FF. The former is the same as the "Total" curve in Fig. 5.

The observed degradation is related network limitations of the underlying system. Two factors might contribute to this problem. To understand them lets first recall that, as explained in Sec. II C, both (i) the number of pairs of sub-blocks ("pencils") which are exchanged simultaneously in parallel and (ii) the number of times this exchange operation has to be sequentially performed to achieve the full distributed transpose increase linearly with N_{cart}. The former naturally implies an increase in the probability of network congestion (collisions) with N_{cart}, which effectively decreases the network bandwidth. The latter implies an accumulation of latency proportional to N_{cart}. Moreover, since the size of the sub-blocks being transposed decreases accordingly (they're given by $N_\chi/N_{\text{cart}} \times N_s \times N_\phi/N_{\text{cart}}$), the higher the core-count is, the higher the ratio between latency time to the time spent on the actual transfer becomes. On a given machine, when this ratio approaches unity, the algorithm reaches its scaling saturation.

To further clarify the mechanism is responsible for the results of Fig. 7, the same experiment is repeated with different matrix sub-block sizes. The angular grid-count is kept unchanged (same as in Figs. 5 and 7) but the radial grid count varied between 1 and 1024. The results are gathered in Fig. 8, where the corresponding speedup factors for the parallel transpose on semi-weak scaling tests are presented. It is clear that, for the smallest matrix sub-blocks (lighter curve), the speedup degradation occurs the soonest, indicating that the communication latency times dominate over the reduction of the number of matrix rows due to the Hermitian redundancy. Therefore, it can be concluded that the degradation observed for the smaller message sizes is mostly due to the latency

times, which are inherent to any inter-core communication.

Conversely, increasing the matrix sub-block sizes decreases the sensitivity of the results to the latency accumulation, which delays the speedup degradation. Indeed, the biggest matrix sub-block size case (darker curve) yields a transpose speedup larger than 1.8 (on HPC-FF), for $N_{\mathrm{cart}} = 1024$, which is much closer to the expected theoretical two-fold speedup. However, this is not conclusive with respect to the possible role of network congestion. For that the absolute execution time values need to be inspected for a wider set of the bigger message sizes, which increase the effect of collisions. This subject shall be revisited in Sec. VII.

At the time these tests where made, the HELIOS machine, at the Computational Simulation Centre of International Fusion Energy Research Centre (IFERC-CSC) in Aomori, was not yet available for production, so the same tests could only be repeated on that machine later on. This topic shall be revisited in the remaining text. For the time being, the results presented so far serve as motivation to experiment with alternative distributed transpose algorithms, which might be less prone to network latency issues. This is the material covered in the next few sections.

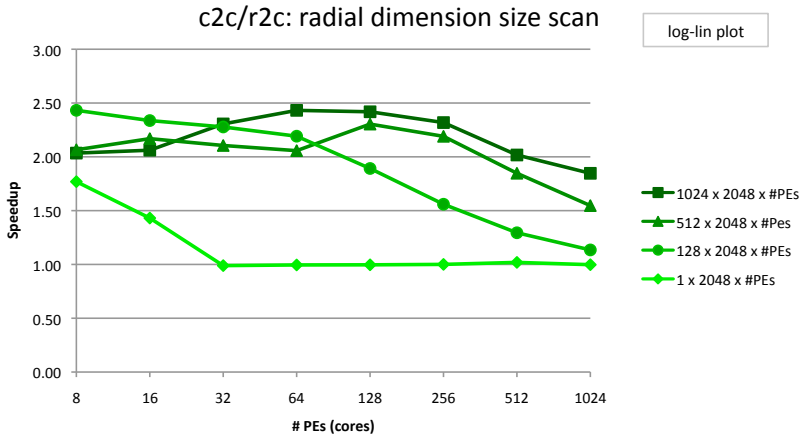

FIG. 8. Semi-weak scaling of the speedup dependence on the radial grid size (N_s), when using the FFT half-complex packed format for the poloidal direction. The values $N_s \in \{1, 128, 512, 1024\}$ are used.

VI. DISTRIBUTED TRANSPOSE: XOR/`MPI_SENDRECV` VS. `MPI_ALLTOALL`

NEMORB's original XOR distributed matrix transpose algorithm was already explained in some detail in Sec. II C and its scaling performance measured in Sec. V. There, it was found that, depending on the machine architecture being used, this algorithm can be affected by network latency and/or congestion when large numbers of cores (of the same order of the toroidal grid-count) are used. This motivates the use of alternative methods to perform the same task. The most obvious one invokes the `MPI_Alltoall` directive of the MPI standard, which establishes the all-to-all communication pattern and performs the message passing automatically. To be able to use it in the simplest way, the sub-blocks to be exchanged across cores need to be stored continuously in memory. This is obviously not the case for the problem in hand, where the corresponding sub-blocks are 3D "pencils". Therefore, two possibilities are available. Either to use temporary buffer arrays, where the data for each sub-block is stored continuously before being communicated, or to create a new data type which specifies the memory access pattern to the sub-blocks in a continuous manner. Both have been investigated and are described below.

The first solution is the simplest to implement. All that is needed is to create a temporary copy of the sub-blocks onto a 4D buffer array that stores them sequentially. Another 4D buffer array is necessary to receive the data after being acted upon by the `MPI_Alltoall` directive. The final step involves extracting the data from the second 4D buffer and putting it with the correct memory layout on the output 3D transposed matrix. Such a standard algorithm has been already implemented in NEMORB before. The source code originally written by Trach-Minh Tran (CRPP) was kindly provided by Paolo Angelino (CRPP) and it could, therefore, be used straightforwardly, with only minor adaptations required to comply with the new Hermitian reduced method.

The second solution aims at generalizing the previous method to avoid the need for explicit use of temporary storage buffers, leaving that task to the internals of the MPI library implementation.

This can be achieved via the most flexible MPI derived data type, namely, the structure type, accessible via the `MPI_Create_data_struct` directive. Passing it as an argument to the `MPI_Alltoall` directive allows it to read (write) directly the original 3D input (output) sub-blocks, with the eventual needed data buffering being handled "under the hood" by the MPI library. This is therefore expected to be more efficient, though it will be seen in the following paragraphs that this is not always necessarily the case. Once more, this method has been previously written by Trach-Minh Tran and the corresponding NEMORB's source code was also provided by Paolo Angelino (CRPP). Additional debugging was necessary to have it working according to the MPI2 standard. The corresponding sub-routines ended up being completely re-written, but the basic original structure was kept. Another change was introduced to create the MPI derived data types just the first time the distributed transpose is performed, to avoid unnecessary accumulation of overhead. Subsequent calls to the routine simply re-use the same derived data type, which does not change in time.

The very same principles can be applied to the XOR/`MPI_Sendrecv` method, which also uses temporary storage buffers to hold a single input (output) sub-block before (after) it is communicated with the `MPI_Sendrecv` call. Here the overhead involved in copying the temporary buffers is, in principle, less significant than within the `MPI_Alltoall` method, since they are much smaller and therefore have better chances to fit in the cache memory. Nevertheless, avoiding this explicit task could also be beneficiary in this case, so this method was implemented and tested along with the others.

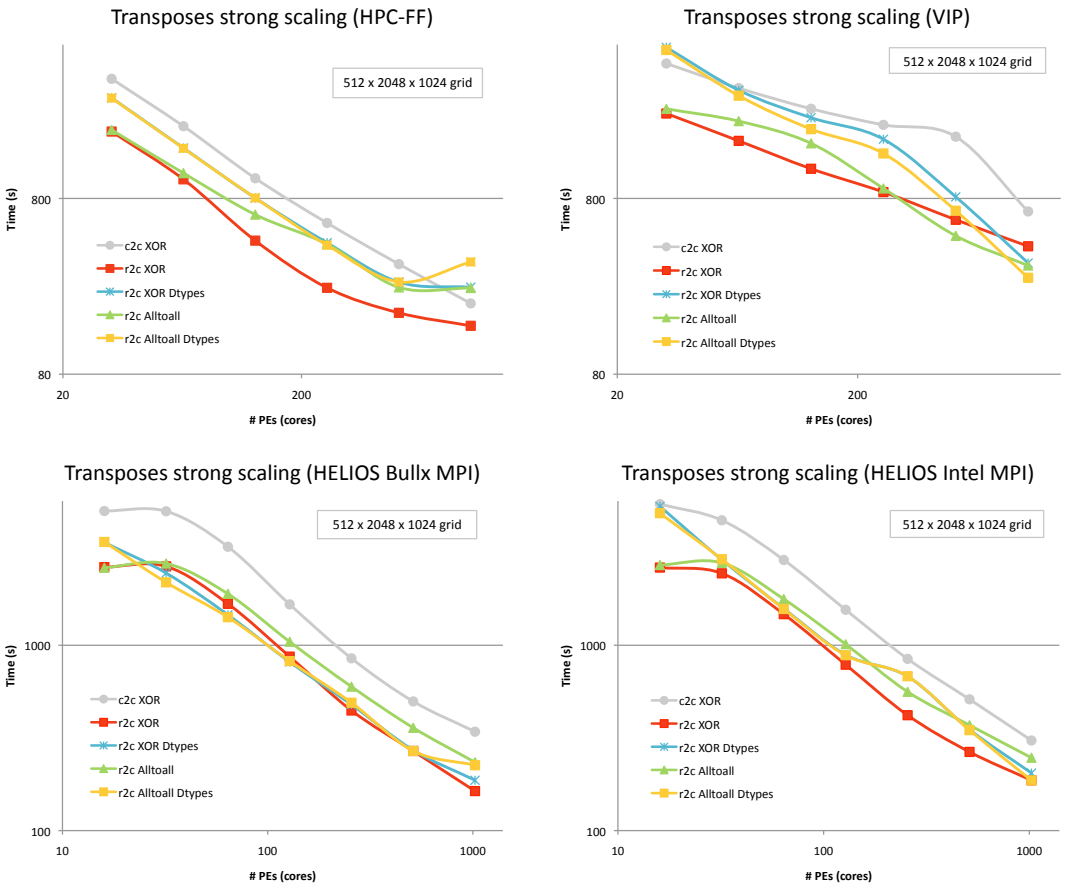

FIG. 9. Strong scaling of the distributed transpose on a grid-count of $N_\chi = 2048$, $N_s = 512$ and $N_\phi = 1024$ for the different transpose methods described in the main text. The results are shown for the HPC-FF (JSC), VIP (RZG) and HELIOS (IFERC-CSC) machines.

The transpose methods before are tested on a fixed grid-count of $N_\chi = 2048$, $N_s = 512$ and $N_\phi = 1024$, which is representative of NEMORB's spatial grid for an typical ITER simulation. A strong scaling study, obtained by distributing the same problem on different numbers of cores, is made on different HPC platforms, namely, the HPC-FF (JSC) with Intel Nehalem CPUs and InfiniBand interconnect, the IBM Power6 VIP (RZG) and the, meanwhile available, HELIOS (IFERC-CSC), which uses Intel Sandy-Bridge CPUs with InfiniBand interconnect. On the last, two

different MPI library implementations are used, namely, the default BullX MPI and the alternative Intel MPI. The results are gathered in Fig. 9, where each point on the graph represents the time it took to perform the transpose of the 3D grid (between the poloidal and toroidal directions, Fig. 2) followed by the corresponding back transpose, repeated a thousand times.

The XOR/`MPI_Sendrecv` method with temporary storage buffers is used in three variants. The first of them acts upon the original Hermitian redundant c2c Fourier transformed data. It yields the grey curves labeled "c2c XOR", which give the reference data, since they correspond to the original method. The second variant is given by the blue curves labelled "r2c XOR (unpacked)". They correspond to the Hermitian reduced data yielded by the r2c poloidal FFT, padded with zeros until the condition $N_\chi/2 + 1$ being a multiple of the number of cores N_{cart} is fulfilled, as explained in Sec. IV. Since $N_\chi = 2048$, having $N_{\text{cart}} = 1024$ cores requires padding with 1023 zeros, yielding a transpose as big as the one of the original Hermitian redundant data. This is the reason why both these methods cost the same for that number of cores on all machines. The third variant yields the red curves, labeled "XOR r2c", correspond to the same method remedied with the half-complex packed format of Sec. V, which avoids the zero-padding. The light-blue curves labeled "XOR Dtype" use an MPI derived data type to avoid the explicit temporary storage buffers. The remaining curves correspond to the two `MPI_Alltoall` variants applied to the Hermitian reduced data only. The green curves labeled "r2c Alltoall" use the temporary storage buffers, whereas the yellow curves labeled "r2c Alltoall Dtypes" use instead MPI derived data types.

All the curves displayed correspond to the best case out of ten runs made for each point in the plots. If all the data were to be plotted, quite some scatter would show, indicating that, as expected, the communication involved in this algorithm is indeed quite intensive. The overall performance depends on the machine load during the run, as well as on the physical location of the computer nodes involved in the transpose relative to each other. More distance implies naturally more time cost.

The first and most important conclusion to be drawn from the plots in Fig. 9 is that, barring the VIP case that uses a different architecture (IBM Power6), the most efficient method seems to be the half-complex packed XOR/`MPI_Sendrecv` already used in the previous section, that is, the red curves labelled "XOR r2c". Additionally, among the cases that use the `MPI_Alltoall` directive, it is the derived type variant (yellow curves) which performs best, as was expected. This method even yields the best overall results on VIP. Conversely, for reasons which are not transparent, the opposite happens on HPC-FF, where this method is even slower than the original full-complex transpose (grey curve). Together with the less performant derived data type variant of the XOR/`MPI_Sendrecv` algorithm on this machine, they seem to indicate that using derived data types within the ParTec MPI distribution available on HPC-FF is not the most efficient way to go. As such, whether or not using derived data types makes sense from the performance point of view depends strongly on the MPI implementation available on a given machine.

Another important point is that, loosely speaking, the general shape of the strong scaling curves on HPC-FF and HELIOS is rather similar, especially when compared to the VIP's counterparts. From the machine architecture point of view, this is the expected behavior. Nevertheless, in absolute terms, the fastest results were obtained on HPC-FF. Switching from the default BullX MPI (`v1.1.14.3`) implementation to the Intel MPI (`v4.0.3`) one on HELIOS did not change this picture. Specifically, taking the largest number of cores used in the study ($N_{\text{cart}} = 1024$), the same task took 203 s on Bullx MPI and 186 s on Intel MPI, but only 151 s on ParTec MPI (`v5.0.26-1`) on HPC-FF. These results were not initially expected, considering the similarity of the architectures on both HPC machines and that HELIOS has more recent hardware. In reality, a more detailed analysis of the issue revealed that this difference is related initialization costs for the all-to-all communication pattern. This renders the interpretation of such cross machine comparisons impaired, and for that reason, a detailed study of the issue was conducted in parallel elsewhere [4], [5, chapter 4 and 9]. Here it suffices to mention that the communication initialization costs are subtracted from what remains of the scaling studies shown in this report.

In sum, the answer to which is the best transpose method is not absolute. It is machine and/or MPI distribution dependent. Nevertheless, the XOR/`MPI_Sendrecv` transpose with temporary storage buffers on Hermitian reduced (packed) data (red curves) seems to be the algorithm with the most consistent performance across the machines used for this study, and is, as such, suggested as the default method. The speedup measurements made relative to the original full-complex algorithm yielded values of about 1.5, 1.7 and 2.0 for HPC-FF, VIP and HELIOS, respectively. For the VIP machine, the best method is the `MPI_Alltoall` with MPI derived types (a speedup of 2.4 was reached), contrary to HPC-FF, for which this is the worst one (even worse than the original Hermitian redundant method). As to the question of the latency effect seen before, which motivated the use of different transpose algorithms in the first place, it seems that it has a stronger

impact on HPC-FF (curves start to flat out for higher numbers of cores) than on the other two machines.

VII. ALTERNATIVE METHODS

This section starts with the general conclusion of the previous one, namely, that the `MPI_Alltoall` directive is not superior to the XOR/`MPI_Sendrecv` counterpart for the transpose problem-sizes under consideration. The follow-up investigations attempt to further find better alternatives to perform this task. Firstly, two additional transpose algorithms are considered, namely, from FFTW3.3 and BLACS/ScaLAPACK (via Intel MKL) libraries. Their performance measurements revealed that they are not competitive with the previous algorithms in hand. The corresponding results for both HELIOS and HPC-FF machines are shown in Sec. VII A. Another paradigm is attempted using the `MPI_Gather/Scatter` collective communication directives. They pass the messages via hierarchical trees, using neighboring cores to help propagate them in a 'divide and conquer' manner. Even though this is much more efficient than point-to-point communication patters, the blocking nature of this directive prevents good scaling properties when used to perform distributed matrix transpositions. This is the subject of Sec. VII B.

A. Transposes from FFTW and MKL libraries

Two additional parallel transpose algorithms are added to the set already tested before (Sec. VI). These come from the FFTW library and the Intel MKL implementation of the BLACS/ScaLAPACK library. For testing purposes, a simpler 2D version of the transposes is implemented, which in NEMORB's language, corresponds to considering a spatial domain with a single radial surface. They are compared to the 2D version of two of the methods implemented before, namely, the XOR/`MPI_Sendrecv` and the `MPI_Alltoall`, both without using derived data types (red and green curves in Fig. 9). This choice is based on the overall performance of the former and the close relation of the latter to the FFTW3.3 transpose algorithm.

The test cases correspond to a matrix size of $N_\chi \times N_\phi = 32768^2$, parallelized over the second dimension. So, on a given number of cores N_{cart}, each core holds $N_\chi \times (N_\phi/N_{\text{cart}})$ of the input data and $N_\phi \times (N_\chi/N_{\text{cart}})$ of the transposed data. This choice yields a similar effective data-size to the nominal 3D ITER-case of the previous sections. The transpose and back-transpose are performed a 101 times, but only the last 100 are measured. This effectively discards both `MPI_Init` and transpose related all-to-all communication initialization costs. The reason for this is the excessive values yielded by the HELIOS machine for this task at the time this report was written, which would masquerade the actual cost of the transpose, as already briefly discussed in Sec. VI. The figures obtained on HELIOS and HPC-FF on 1024 cores are listed in Table II. They correspond to the best results out of at least three simulations.

	XOR	Alltoall	FFTW3.3	MKL	MKL (impi)
HELIOS	14.4 s	25.2 s	46.1 s	839.6 s	1232.8 s
HPC-FF	12.9 s	41.2 s	25.6 s	1166.9 s	-

TABLE II. Time spent by the different transpose algorithms to perform 100 pairs of transposes and back-transposes of a 32768^2 matrix on 1024 cores, excluding communication initialization costs. On HELIOS, the Intel Fortran compiler (`v12.1.1`) and the libraries FFTW (`v3.3`) and MKL (`v10.3.7`) were used with BullxMPI (`v1.1.14`) for all results, except the last column one, for which the Intel MPI (`v4.0.3`) was used. On HPC-FF, the Intel Fortran compiler (`v12.1.1`) and the libraries the MKL (`v10.2.5`) and Partec MPI (`v5.0.26-1`) were used.

Clearly, there is no advantage in using either FFTW's or the MKL's distributed transpose algorithms for the cases under consideration. The latter even proved to be more than two orders of magnitude slower than its counterparts, which was somewhat surprising. A strong scaling study comparing this method with the `XOR/MPI_Sendrecv` revealed its poor scaling properties, which explains the corresponding figures in Table II. These results, represented in Fig. 10, were measured on HPC-FF. Similar figures were obtained on HELIOS.

It is also noteworthy that the input matrix data used is real-valued, not complex, as one would need for NEMORB, since the transpose takes place after the poloidal FFT is computed. The reason

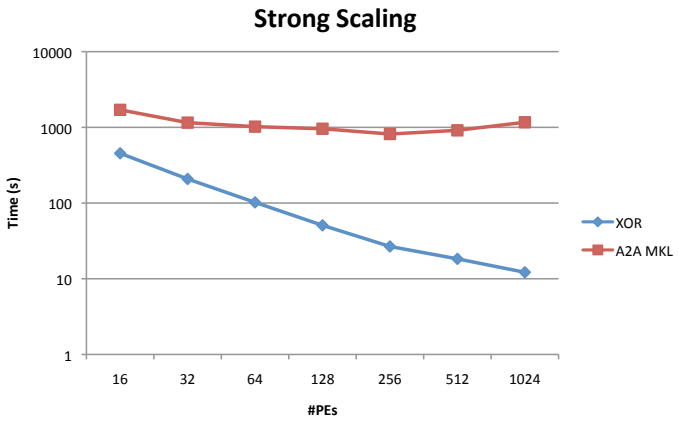

FIG. 10. Strong scaling of the `XOR/MPI_Sendrecv` and MKL's BLACS/ScaLAPACK transposes on HPC-FF. 100 pairs of transpose and back-transpose of a 32768^2-sized matrix are measured, excluding communication initialization times.

for this is that having FFTW's distributed transpose, which uses the Fortran ISO C bindings, working with complex numbers turned out to be harder than expected. Although stated in the manual that it can be done using "tuples", the attempts made have failed. This together with the not very encouraging performance results obtained implies that no further investment is made in this topic. For essentially the same reason, no big effort is put into developing the 3D counterparts of these transpose algorithms to mimic the actual NEMORB's grid-count. Moreover, the MKL case, which uses matrix descriptors within BLACS/ScaLAPACK that expect 2D matrices, makes it very hard (perhaps even impossible) to have an efficient 3D version of the transpose. In this case, one has to perform each 2D transpose separately, and repeat the process for each grid-point in the third dimension. This is obviously very inefficient compared to the remaining methods, which carry the whole third dimension when the 2D transpose is calculated. The MKL 3D tests made on HELIOS with NEMORB's nominal ITER grid-count ($N_\chi = 2048, N_s = 512, N_\phi = 1024$) reveal a slow down of more than three orders of magnitude compared to the other methods, as one would expect from the previous considerations.

B. Alternative to the transpose: `MPI_Gather/Scatter` directives

In the context of NEMORB's 2D FFT algorithm, the distributed matrix transposes are invoked to make the distributed angular data locally available, such that it can be efficiently Fourier transformed. The same goal can be achieved using instead a sequence of consecutive calls to `MPI_Gather` or `MPI_Scatter`, one for each task. These collective communication directives are done very efficiently by the MPI libraries. They pass the messages via hierarchical trees, using the neighbor cores to help propagate them. In the case of a gather (scatter), such communication tree is used to avoid doing a point-to-point communication between the core receiving (sending) the total message and all the remaining ones sending (receiving) their corresponding partial messages. Doing so increases on average the message size (which is good in terms of network latency) and decreases the number of steps needed for the full operation from $\mathcal{O}(N)$ to $\mathcal{O}(\ln_2 N)$, yielding bigger improvements the larger the array size N is. The gather/scatter algorithm shares the local data layout with its XOR counterpart. Namely, the local data is sub-divided into as many 3D sub-blocks ("pencils") as there are sub-domains (cores), as schematized in Fig. 2. The communication pattern is different, though. Taking the gather algorithm as the example (the scatter counterpart is completely equivalent), the different sub-blocks are gathered by the core whose MPI rank (plus 1) is equal to their row index (e.g. core zero gathers locally the first row of sub-blocks, core one gathers locally the second row of sub-blocks and so on). This has deep implications on the scalability of the algorithm, since when a core gathers its corresponding distributed sub-blocks, the message path involves all remaining cores. The next core can only start to gather its corresponding sub-blocks once the first one is finished, otherwise there would be race conditions (concurrent message paths). This can be regarded as a sort of a serially blocking communication pattern. On one hand, the efficiency of the hierarchical tree involved in a single gather communication can be inferred from

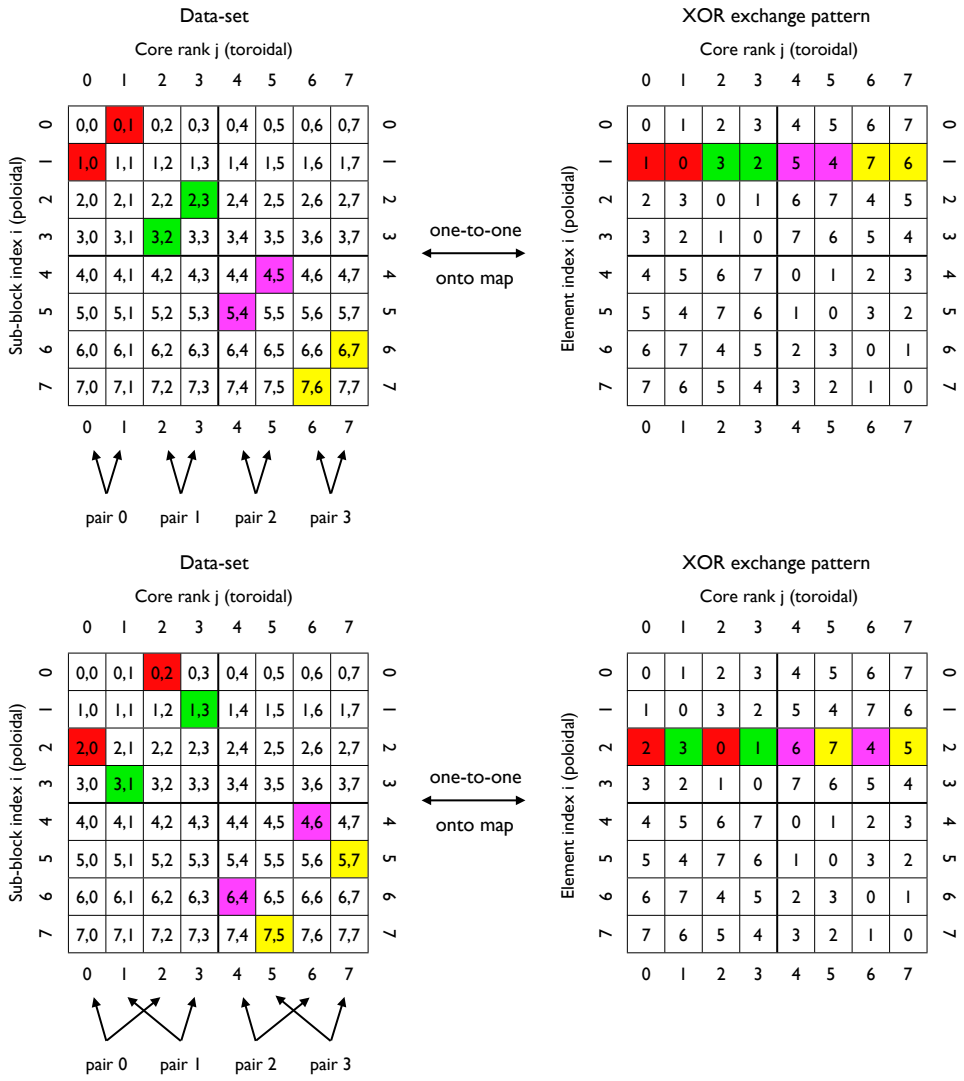

FIG. 11. Illustration of the map between the data-set (left) and the XOR communication pattern (right) for $N_{\rm cart} = 8$. The numbers in the latter specify simultaneously the i-index of the data sub-block to be exchanged and the core j-rank it goes to. The correspondence between the second (upper) and third (lower) rows of the exchange pattern (right) and the actual data-set exchange pairs (left) is highlighted in color. The non-concurrency between these pairs can be confirmed. The same applies to exchange pairs yield by the remaining rows of the exchange pattern (not shown here).

the performance achieved by this method within a small number of cores. Indeed, as shown in Fig. 12, this method is superior to the XOR counterpart within a single compute node ($N_{\rm cart} = 16$). On the other hand, for higher core-counts the blocking character of the algorithm takes over. So, even if one assumes that the time cost of a single core gathering all the sub-blocks corresponding to its attributed row would show perfect weak scaling (constant amount of data per core), the serial sequence of gathers done by the remaining cores makes the total cost increase linearly with the number of cores. This is exactly what is shown in Fig. 12.

In turn, the XOR/`MPI_Sendrecv` algorithm has a fundamentally different message path pattern For the (recall Sec. II C), the exchange of the sub-blocks proceeds according to a pre-established bi-dimensional $N_{\rm cart} \times N_{\rm cart}$ pattern of "core rank `.XOR.` sub-block (row) index". This is what is illustrated in Fig. 11, where the map between the data-set (left) and the XOR exchange pattern (right) is shown, on a case parallelized over eight cores. The values in the latter are integers that, together with their row and column indices form a one-to-one and onto map with the poloidal and toroidal indices of the sub-blocks of data ("pencils"), represented by the matrix elements on the left. Namely, within column (or core rank) j, element i stores an integer number $0 \leq l < N_{\rm cart}$. It specifies that sub-block l of the corresponding data array (left side) on core rank j is to be exchanged with core rank l. Conversely, the element i of the XOR column on core rank l (right

side) has the value j, which means that sub-block j of the data-set on core rank l (left side) is going to be exchanged with core rank j. This forms a consistent exchange pair between sub-block l on core j and sub-block j on core l. Each row of the XOR matrix forms $N_{\text{cart}}/2$ of such pairs, without any concurrent message paths. As a practical example of this rule, note that the elements in the first row of the XOR exchange pattern specify the diagonal data-set sub-blocks (left side) and therefore involve no actual inter-core exchange. The data exchange dictated by the second and third rows of the XOR pattern are depicted in Fig. 11. Sequentially realizing all rows in the pattern yields the full transpose.

Increasing the number of task while keeping the amount of data per task constant yields the weak scaling properties of this method. Even though (i) the number of messages to pass (sub-blocks) increases quadratically with the number of cores, (ii) their size decreases linearly and (iii) the amount of pairs available to exchange messages simultaneously increases also linearly. In the limit of large number of cores, the three factors balance and a flat curve representative of perfect weak scaling should be obtained, with a level set by the network bandwidth. This is shown by the red curve in left plot of Fig. 12, at least until 256 cores. For higher numbers of cores the scaling starts to degrade mostly due to network limitations, as we discussed already in Sec. V. In these cases, for HPC-FF, the accumulation of latency (at each row of the XOR pattern) was found to represent a significant part of the communication time, due to the smallness of the messages being passed, which limited the scaling properties of the algorithm. On HELIOS, a similar effect is seen. Additionally, for bigger message sizes, the effect of the network collisions seems to start playing a role for the highest core-count (2048). This can be seen from the plot on the right of Fig. 12, where the same scaling is repeated for higher radial grid-counts, which effectively increases the size of messages being passed. The larger the message size is, the lower the scaling sensitivity to network latency is, since the latency times become negligible compared to the actual message exchange time. The broken lines show tests made on messages four, eight and sixteen times bigger than the original case (solid line, same case of the left plot), respectively. For such message sizes the role of latency becomes sub-dominant and therefore it can not account for the increase observed in the execution time. The natural candidate to do so seems to be network congestion, caused by too many messages being exchanged simultaneously. The gather (or scatter) transpose method seem to suffer from the same problem, as is hinted by the last data point of the corresponding weak scaling curve Fig. 12 (left), which starts to deviate from the linear behaviour observed or smaller core-counts.

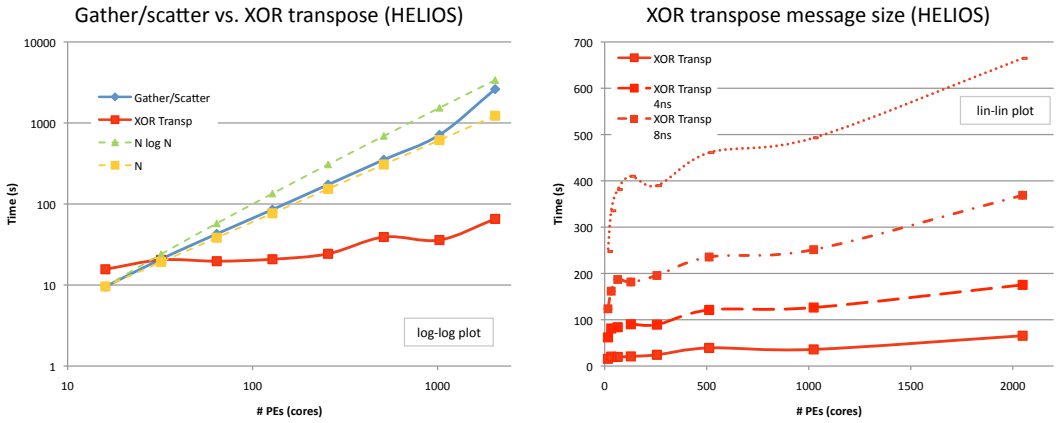

FIG. 12. On the left, weak scaling measurements for the XOR/`MPI_Sendrecv` transpose and `MPI_Gather/Scatter` methods on HELIOS with Bullx MPI (`v1.1.14`). The grid-count is given by $N_\chi = 2048$, $N_s = 512$ and $N_\phi = N_{\text{cart}}$. On the right same weak scaling measurements for the XOR/`MPI_Sendrecv` transpose using different message sizes by changing the radial grid-count: $N_s = \{512, 2048, 4096, 8192\}$. Note the logarithmic (left) and linear (right) plot scales.

In sum, one concludes that the gather/scatter algorithm can not be considered as an alternative to the XOR/`MPI_Sendrecv` transpose due to its much poorer scalability with the number of cores. However, it is worth to point out that the non-blocking version of these, as well as other collective directives, which are part of the new MPI-3 standard, might change this picture, by allowing to overlap the FFT computations with the distributed matrix transposition. Such tests are naturally left for future work. For the time being, the remaining sections focus on the XOR method, which still offers optimization potential.

VIII. OPTIMIZATION OF THE XOR/MPI_SENDRECV TRANSPOSE

Having tried several different alternative algorithms to the distributed transpose, it became clear that the best choice to perform such task is the XOR/`MPI_Sendrecv` method. Moreover, this choice further offers optimization possibilities related to its "hand-coded" algorithm. Unlike the methods relying on "canned" libraries (such as the `MPI_Alltoall` directive and its derivatives), this one allows changes to be made directly to the communication pattern. The following two sections explore this possibility. Firstly, by trying to realize the communication pattern matrix following different sorting orders in the sequential communication steps involved (Sec. VIII A). Secondly, by invoking the zeros yield by the known NEMORB's Fourier filters to perform only partial data transposition (Sec. VIII B).

A. XOR exchange pattern sequence

Secs. II C and VII discussed already the internals of the XOR/`MPI_Sendrecv` algorithm up to a degree that makes the material of this section appear as a natural follow-up. It was mentioned then that a matrix storing the communication pattern is followed, row-by-row, to perform the transpose in N_{cart} sequential steps within a `DO`-loop (recall Fig. 11). Each step (row) provides a set of pairs of cores that exchange specific sub-blocks of data in a non-overlapping fashion. Since such steps of communication are independent of each other, there is no restriction on the order of which they are executed.

Different sequences are followed here to cover the complete XOR exchange pattern. In practice, an extra array with dimension N_{cart} is created to store the order over which the rows of the XOR exchange pattern matrix are to be followed. Four different cases are considered, namely, the standard sequential increasing order (Reference), the reverse order sequence (Reverse), the Alternating sequence order given by $\{1, N_{\text{cart}}, 2, N_{\text{cart}} - 1, \ldots, N_{\text{cart}}/2, N_{\text{cart}}/2 + 1\}$ and finally the random sequence order, obtained using a Knuth shuffle algorithm. The complete 2D FFT algorithm (forward transform followed by inverse transform) using the optimized (half-complex packed format) XOR/`MPI_Sendrecv` is measured here on NEMORB's ITER-sized spatial grid count ($N_\chi = 2048$, $N_s = 512$ and $N_\phi = 1024$). What is shown in Table III are the elapsed times for executing it a thousand times, using the different exchange pattern sequences aforementioned. Following the same practice as before, the communication initialization costs have been removed from the results.

	Reference	Reverse	Alternating	Knuth shuffle
HELIOS	207.7 ± 5.9 s	205.5 ± 7.8 s	213.5 ± 3.3 s	245.1 ± 1.5 s

TABLE III. Elapsed times measured in HELIOS for the complete 2D FFT algorithm (half-complex packed format XOR/`MPI_Sendrecv`), followed by its inverse equivalent to revert back from Fourier to configuration space, executed one thousand times on NEMORB ITER-sized grid ($N_\chi = 2048$, $N_s = 512$, $N_\phi = 1024$). Each column corresponds to a different XOR exchange pattern sequence. Eight simulations for each sequence were performed and the average values are shown.

The best results are obtained for the case where the `DO`-loop proceeds in reverse order, although only marginally faster than the standard case figures, with the difference well inside the error bars, given by the spread in the measurements made. The alternating sequence yields slower execution times but it is the random order (Knuth shuffle) which gives the worst results, even subtracting the overhead related to the random shuffling of the sequence, which is done only once, before the first transpose is executed. These findings are consistent with the expectation that ordered memory access patterns pay-off compared to erratic ones, which are prone to cache misses. Similarly, the alternating sequence seems to have more cache misses than the standard and reverse cases that maximise the continuity of the sub-block memory access pattern inside the transpose `DO`-loop. Based on these results, the Reference sequence is kept as the default. The marginal gain obtained ($< 1\%$) with the reverse order sequence does not justify the necessary changes, especially considering how this would obscure the subject of the next section.

B. Fourier filtering and the partial transpose

This section outlines the exploitation of NEMORB's Fourier filtering, which forces a large fraction of the Fourier transformed matrix to vanish. Naturally, these zeros need not be transposed and inhibiting their communication should, in principle, alleviate the data exchange burden of the algorithm.

The basic filter used in NEMORB is a diagonal one, which retains only a relatively narrow band of poloidal modes around each toroidal mode in the system. Physically this corresponds to retaining only modes that are relatively well field-aligned. All the modes outside this band are set to zero, because they do not take part in the dynamics and moreover could contribute to increase the numerical noise. The problem is that this filter, being diagonal, can only be applied after the Fourier transforms in both angles are calculated, and not before. Indeed, after the poloidal (local) Fourier transform is performed, the resulting (poloidal) spectrum needs to be carried out as a whole into the toroidal Fourier transform, since the toroidal direction still contains configuration space data. This means that the full transpose must be carried out before the field aligned filter can be applied. The same holds for the back transpose that is done after the filter is applied. Such operation must occur only after the inverse toroidal Fourier transform is applied, which converts the narrowed toroidal spectra into a filled-up matrix corresponding to its toroidal configuration space representation. The problem was also investigated while trying to extend the convolution theorem treatment already used in NEMORB [7] to this part of the problem. The same conclusion was reached though, namely, that one needs to carry out the full matrix transposes simply because the filter is diagonal and therefore mixes both angular dimensions in a non-separable form.

On the other hand, besides the field aligned filter, a square filter is also applied to the data to ensure a lower Nyquist frequency cut (to yield at least four nodal points per wavelength, instead of two). Unlike before, it is possible to separate the projections of the square filter in the poloidal and toroidal directions. Hence, one can apply the filter's poloidal projection to the Fourier transformed poloidal data before it is exchanged across cores, therefore avoiding communicating the resulting zeros. Such a scheme would be easily feasible within the `MPI_Gather/Scatter` algorithm outlined in Sec. VII B (only a subset of cores would call the gather/scatter directive), but unfortunately the expected gain does not compensate for the worse scaling of this method compared to the XOR/`MPI_Sendrecv` transpose (recall Fig. 12). Implementing it on the latter is slightly more involved and is the subject discussed below. Since the algorithm to be modified is the fastest one, namely the one using the Hermitian redundancy, the half-complex representation has to be taken into account when both applying the poloidal filter and modifying the XOR communication pattern.

The "four-node per mode" filter in the poloidal direction discards the high frequency modes

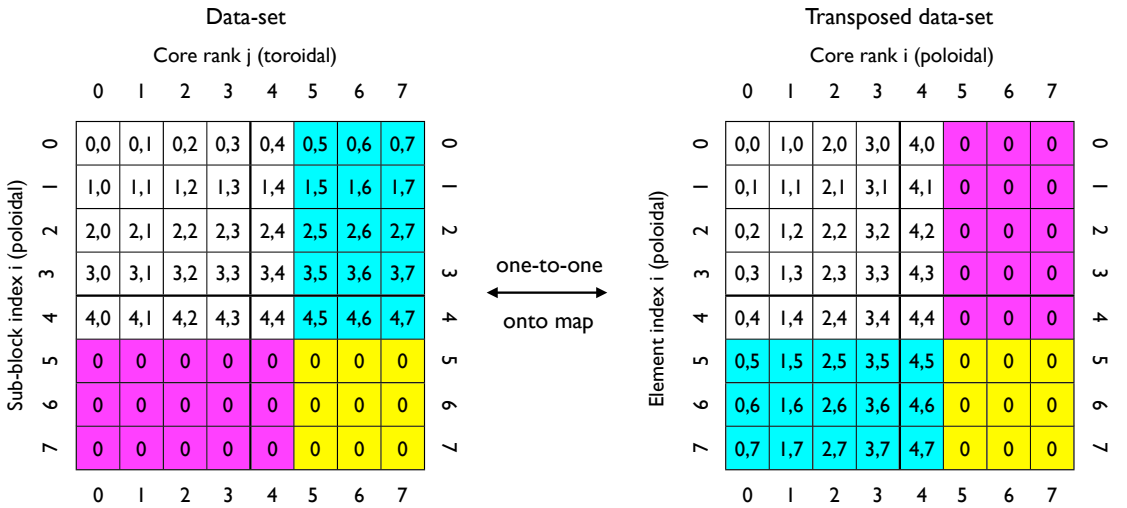

FIG. 13. Illustration of the partial data transposition required after the "four-node per mode" low-pass filter is applied to the poloidal Fourier transformed data. The white areas are transposed using original bi-directional message exchange (`MPI_Sendrecv`). The cyan area sends the data to the magenta area without receiving the corresponding zero-elements (uni-directional message passing). The zeros in the yellow area need not be communicated at all.

specified by the poloidal index interval $[N_\chi/4+1, 3N_\chi/4-1]$ (recall Fig. 4). Since only the non-redundant part of the spectrum is considered, this interval reduces to $[N_\chi/4+1, N_\chi/2]$. Moreover, because the Nyquist frequency mode ($N_\chi/2$) belongs to this interval, it is set to zero by the filter and there is no need to invoke the packed format of Sec. V. One simply retains the first $N_\chi/4+1$ of the $N_\chi/2+1$ rows that would constitute the full non-redundant unfiltered spectrum. Fig. 13 illustrates these ideas on a (Hermitian reduced) case with $N_\chi = 16$, distributed over $N_{\text{cart}} = 8$ cores. The filtered-out interval corresponds to the lower part of matrix on the left, coloured in magenta and yellow. The figure on the right side represents the corresponding distributed transpose matrix. The white regions maintain the XOR algorithm unchanged, with the messages being exchanged bi-directionally between the corresponding cores involved. The magenta and cyan regions need uni-directional message-passing only. The cores with non-zero elements (cyan) send them to their counterpart cores, which in turn have zero-elements (magenta). So, the former need not to receive any data from the latter. Finally, the lower right corner area (yellow) requires no communication whatsoever. Obviously, the very same principles directly apply to the toroidal projection of the square filter, although this is not done here.

If implemented, the uni-directional communication part of the filtered transpose algorithm (magenta/cyan areas) would reduce the total message size, but not the number of communication connections. Hence, such change is only expected to improve significantly the performance of the algorithm if it were network bandwidth-bound. Since however, the results obtained point to network latency and congestion as the main factors limiting its scalability for the grid-counts under consideration, most of the performance gain is expected to come from the yellow area of the matrix, where no communication is necessary. Implementing this part would effectively reduce the number of concurrent messages exchanged simultaneously. This is why the effort is put exclusively there.

Fig. 14 helps to understand in detail the modifications necessary to avoid communicating messages in the yellow area of Fig. 13. Similarly to Fig. 11, the left side shows the same (untransposed) data-set matrix of Fig. 13 and the right side provides the corresponding XOR exchange pattern, distributed in the same fashion over 8 cores. The colours provide a pictorial view of the map between communication pattern and data-set just described. In particular, it is the upper right corner of the XOR matrix (right side) that rules the lower right corner of the data-set matrix (left side). This is seen from the fact that those element values l specify data-set sub-block indexes (or core ranks) higher than $N_{\text{cart}}/2$. They have been coloured in yellow to highlight that the communication they specify is rendered unnecessary by the poloidal filter and, as such, is inhibited in the algorithm. Compare with the examples on the right side of Fig. 2 to see that, of the four exchange pairs per row, this causes the one coloured in yellow to be inhibited. Obviously, the bigger N_{cart} (and hence also the data-set), the more pairs are affected.

The reason for the "cyan diagonal" of elements within the yellow area has to do with the fact that

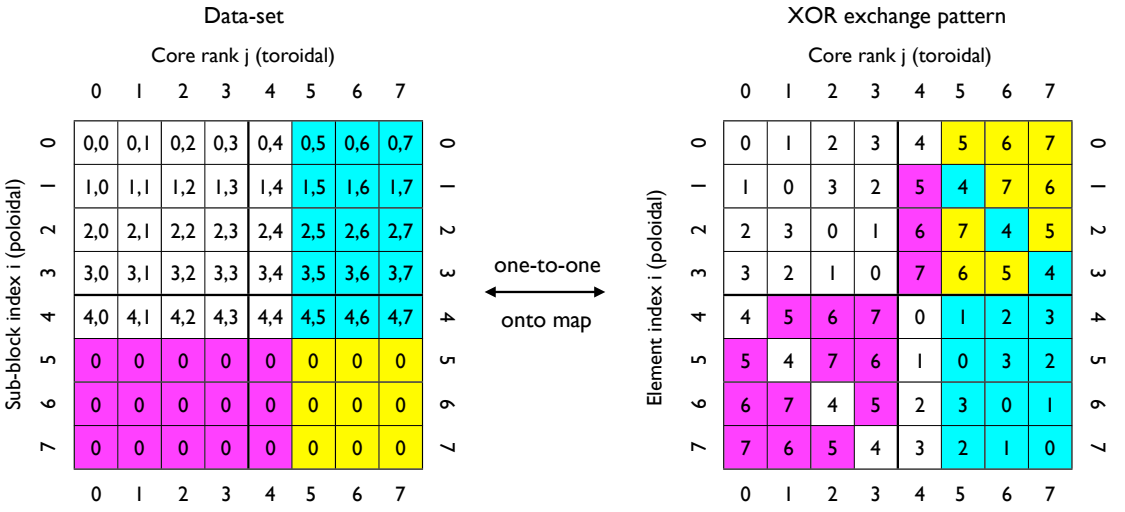

FIG. 14. Illustration of the effect of "four-node per mode" poloidal low-pass filter on the map between the data-set and the communication pattern. The same data-set matrix of Fig. 13 is shown on the left. The corresponding XOR exchange pattern on $N_{\text{cart}} = 8$ is represented on the right, using the same colours. The numbers in the latter specify simultaneously the i-index of the data sub-block to be exchanged and the core j-rank it goes to. The yellow area shows the part of the communication that is inhibited by filter.

the filter keeps the poloidal mode corresponding to the index $N_\chi/4$. They are given by the data sub-blocks with index $N_{\rm cart}/2$ belonging to the cores ranked higher than $N_{\rm cart}/2$. All remaining areas of the pattern (including the uni-directional communication areas in cyan and magenta) are left untouched in the algorithm and the original (bi-directional) XOR/`MPI_Sendrecv` communication is carried out. A clear conclusion that can be drawn from this figure before proceeding to the performance measurements is that, for a given problem size, the higher $N_{\rm cart}$ is, the higher the ratio between the yellow area and the rest becomes. Therefore, bigger grid-counts parallelised over high numbers of cores benefit the most from this method.

The Table IV compares the cost of the new NEMORB's 2D filter algorithm using the previous considerations (`Partial`) and the same algorithm without doing so and therefore carrying out the matrix transposition including all the filtered-out elements (`Reference`). An improvement in performance of about 6% is achieved for the ITER sized grid-count. This further enhances the speedup factors shown in Sec. VI for the new algorithm compared to the original one. On HELIOS the improved speedup factor reached about 2.2. On HPC-FF, the degradation observed previously was also reduced, with the speedup factor increasing from below 1.5 to slightly above 1.7. As expected initially, reducing the average number of messages that need to go through the network simultaneously alleviates congestion issues that the network might have and improves the scaling of the XOR/`MPI_Sendrecv` algorithm.

	Reference	Partial (filter)
HELIOS	$208.0 \pm 2.0\,$s	$197.3 \pm 3.5\,$s
HPC-FF	$209.3 \pm 16.6\,$s	$182.3 \pm 9.2\,$s

TABLE IV. Elapsed times measured for seven and three simulations made on HELIOS and HPC-FF, repectively, with full and (filtered) partial transposes on the ITER-sized grid-count. The speedups achieved are within the range $5 - 10\%$.

The final remarks relates to the filter choice presented here, which was motivated by the default poloidal component of the square filter used in NEMORB. In principle, the algorithm also allows different poloidal filter frequency values, although this part of the code was not tested extensively. Moreover, it should be noted that changing this values affects the performance. Obviously, narrower filters yield less zeros and therefore less communication is inhibited. Finally, because of its Hermitian redundancy, the algorithm imposes that the poloidal filter must be symmetric on the full (positive and negative) poloidal mode number domain. No different positive and negative frequency cuts are allowed. As already mentioned before, the same optimization can be done for the toroidal part of the square filter. This is nevertheless left as future work.

IX. CONCLUSIONS

The work reported here aimed at the optimization of NEMORB's bi-dimensional Fourier filtering algorithm, which constitutes a bottleneck of that very HPC-resource demanding code. The implementation of the Hermitian redundancy allowed to reduce the number of FLOPs involved in the Fourier transforms by roughly a factor of two. Further using a clever storage for the Hermitian-reduced data (half-complex-packed-format) decreased the data-set to be transposed across cores by the same amount. This led to speedup factors of the order of two on a nominal grid-count representative of a typical ITER simulation, measured on several HPC facilities. Nevertheless, the performance measurements further revealed that, for large numbers of tasks, a degradation of the speedup could occur due to network being unable to handle the all-to-all message passing with the same degree of efficiency.

The implementation of other transpose methods (`MPI_Alltoall` and its FFTW3.3 and BLACS counterparts), as well as an alternative algorithm based on the `MPI_Gather/Scatter` directives, was attempted, just to conclude that the original "hand-coded" XOR/`MPI_Sendrecv` algorithm gave, in general, the best results. An exception was observed on the VIP machine, at RZG, where the derived data type's version of `MPI_Alltoall` algorithm performed best. On the other hand, the same algorithm yielded the worst overall performance on HPC-FF, at JSC, showing how dependent this algorithm is on the available MPI library implementation. Another reason taken in to account for recommending the XOR/`MPI_Sendrecv` as the default method is related to its flexibility. This allowed to exploit NEMORB's low pass filtering, used to ensure enough resolution on all physically allowed modes in the system, to further reduce the data-set exchanged

across cores. Indeed, *a priori* knowledge about the matrix elements that are set to zero by such filter allowed to exclude parts of the matrix from the data exchange step, such that only a partial transposition was performed. This further increased the gain obtained compared to the original full transposition of the full-complex Fourier transformed data, with the speedup factor of about 2.2 being achieved on HELIOS, at IFERC-CSC. On HPC-FF, where network limitations degraded the expected theoretical two-fold speedup, a benefit was also achieved, with the speedup factor increasing from below 1.5 to slightly above 1.7.

Within the NEMORB context, the previous improvement results apply to each clone of the spatial domain. How will they translate in terms of the global NEMORB speedup depends on the problem size being considered, as well as on the HPC facility being used. Nevertheless, it noteworthy that it could make sense to consider running cases with half the usual number of cores N_{cart} for the toroidal communicator, i.e. with two toroidal grid-nodes per core instead of one, whenever the network limitations start to impair the all-to-all scaling. The same total number of MPI tasks could be achieved by increasing the number of clones (particle domain decomposition) accordingly. For instance, for an ITER-size grid of $512 \times 2048 \times 1024$, instead of using $N_{\mathrm{cart}} = N_\phi = 1024$ and four clones to run on 4096 MPI tasks, one could use $N_{\mathrm{cart}} = N_\phi/2 = 512$ and eight clones to get the same number of MPI tasks. In terms of the 2D filtering algorithm, the drawback of such setup is, of course, that only half of the resources are available to calculate the Fourier transforms and to perform the data exchange. The advantage stems from the effective reduction by a factor of four of the number of MPI connections needed for the all-to-all communication of the transpose (that goes with $\mathcal{O}(N_{\mathrm{cart}})^2$), which can be important when network latency and/or congestion start to take over. An extra benefit relates to a better grid-cells/guard-cells ratio, so the simulation becomes more memory efficient. This recommendation is consistent with the recent results reported in [8], which show it is advantageous to do the domain decomposition with the particle decomposition (clones) as the first communicator. Since the charge assignment operation involves global sums across the clones, if all the clones are distributed sequentially within each compute node (on HPC-FF one can have up to eight clones, on HELIOS the double is possible) this reduces significantly the amount of inter-node communication. Making the toroidal communicator the second one being distributed brings no disadvantages because the transposes that are involved in the 2D FFTs always require an all-to-all communication over large numbers of cores, beyond the node-core-count. Since the choice of using double the number of clones and reduce N_{cart} by the same factor affects the whole code, not just the 2D filter algorithm, to decide whether it makes sense or not implies running full NEMORB simulations using the new Fourier algorithm. This is left for future work.

ACKNOWLEDGMENTS

The author would like to thank Roman Hatkzy, Alberto Bottino, Mattieu Haefele, Trach-Minh Tran and Paolo Angelino for the stimulating discussions. A part of this work was carried out using the HELIOS supercomputer system at the Computational Simulation Centre of the International Fusion Energy Research Centre (IFERC-CSC), Aomori, Japan, under the Broader Approach collaboration between Euratom and Japan, implemented by Fusion for Energy and JAEA.

[1] R. Hatzky, Parallel Computing **32**, 325 (2006).
[2] M. T. Heideman, D. H. Johnson, and C. S. Burrus, Archive for History of Exact Sciences **34**, 265 (1985).
[3] G. Wellen and G. Hager, *PRACE PATC course: Node-level performance engineering* (2012).
[4] T. Ribeiro, *Intermadiate report on HLST Project NEMOFFT (April-June)* (2012).
[5] T. Fehér, M. Haefele, R. Hatzky, K. S. Kang, M. Martone, and T. Ribeiro, *HLST Core Team Report 2012* (2012).
[6] Note1, just a test.
[7] B. McMillan, S. Jolliet, A. Bottino, P. Angelino, T.-M. Tran, and L. Villard, Computer Physics Communications **181**, 715 (2010).
[8] P. Angelino, S. Brunner, S. Jolliet, B. Teaca, T.-M. Tran, T. Vernay, L. Villard, A. Bottino, and B. McMillam, *Parallel transpose in NEMORB, scalability tests* (2011).