Benefits of SMT and of Parallel Transpose Algorithm for the Large-Scale GYSELA Application

Guillaume Latu
CEA/IRFM
FR-13108
St-Paul-les-Durance
guillaume.latu@cea.fr

Julien Bigot
Maison de la Simulation
CEA, CNRS, Univ. Paris-Sud,
UVSQ, Université Paris-Saclay
FR-91191 Gif-sur-Yvette
julien.bigot@cea.fr

Nicolas Bouzat
INRIA (hosted by CEA/IRFM)
FR-13108
St-Paul-lez-Durance
nicolas.bouzat@inria.fr

Judit Gimenez
BSC-CNS / UPC
ES-08034 Barcelona
judit@bsc.es

Virginie Grandgirard
CEA/IRFM
FR-13108
St-Paul-lez-Durance
virginie.grandgirard@cea.fr

ABSTRACT

This article describes how we manage to increase performance and to extend features of a large parallel application through the use of simultaneous multithreading (SMT) and by designing a robust parallel transpose algorithm. The semi-Lagrangian code Gysela typically performs large physics simulations using a few thousands of cores, between 1k cores up to 16k on x86-based clusters. However, simulations with finer resolutions and with kinetic electrons increase those needs by a huge factor, providing a good example of applications requiring Exascale machines. To improve Gysela compute times, we take advantage of efficient SMT implementations available on recent INTEL architectures. We also analyze the cost of a transposition communication scheme that involves a large number of cores in our case. Adaptation of the code for balance load whenever using both SMT and good deployment strategy led to a significant reduction that can be up to 38% of the execution times.

1. INTRODUCTION

A key factor that determines the performance of magnetic plasma containment devices as potential fusion reactors is the transport of heat, particles, and momentum. To study turbulent transport and to model Tokamak fusion plasmas, several parallel codes have been designed over the years. Computational resources available nowadays have allowed the development of several petascale codes based on the well-established gyrokinetic framework. In this article, we focus on the GYSELA gyrokinetic code parallelized using a hybrid MPI/OpenMP paradigm[1, 14, 15, 18].

The main data in GYSELA is an ion distribution function in a 5D phase space. (r, θ, φ) are the space variables with r the radial direction, θ and φ respectively the poloidal and toroidal directions and $(v_\parallel, \mu)$ the velocity variables. $v_\parallel$ is the velocity parallel to the magnetic field lines while the magnetic momentum μ is proportional to the perpendicular velocity squared. The parallelization of the code take advantage of the fact that it is an adiabatic invariant therefore plays the role of a parameter. Uniform mesh partition is used, the size of the phase space grid is $N_r\, N_\theta\, N_\varphi\, N_{v_\parallel}\, N_\mu$. The computational domain is defined on $(r, \theta, \varphi, v_\parallel, \mu) \in [r_{\min}, r_{\max}] \times [0, 2\pi] \times [0, 2\pi] \times [v_{\min}, v_{\max}] \times [\mu_{\min}, \mu_{\max}]$. Lower and upper bounds in $r, v_\parallel$ and μ are fixed according to the physical case. One of the main computational cost is the advection operator (that performs a single time step integration of the Vlasov equation). This operator is split in order to avoid a costly 4D advection in $(r, \theta, \varphi, v_\parallel)$. The chosen Strang splitting consists in four 1D advections (along φ and $v_\parallel$ directions) and one 2D advection (both r and θ directions are treated simultaneously), that are applied at each time step [7, 8]. Producing physics results with this tool requires large computational resources and approaches to optimize the use of these resources are therefore required.

The first such contribution presented in this paper is a new parallelization of the GYSELA Vlasov solver. It relies on a transposition of the distribution function inside each μ value between the 1D and 2D advections. Unlike the previous parallelization [4, 14], it supports large displacements in the (r, θ) plane during one single time step. This enables to consider new physics sub-models and is especially important when the physics shows large poloidal velocities so as to support large time steps to shorten global execution time. Our evaluation show that somewhat counter-intuitively, even with a transposition, this algorithm shows good performance at scale.

The second contribution is an analysis of the best way to leverage new micro-processors architectures in codes such as GYSELA. While the clock-rate of processors has reached a maximum, vendors have to introduce new features so as to keep increasing the floating point operations per second (FLOPS) they can execute. These features include vector units, fused multiply-add, simultaneous multi-

threading (SMT) and an increased number of cores. This increasing complexity makes reaching a significant ratio of processors peak FLOPS more and more difficult. We identify specific problems that arise in GYSELA with the Haswell processor and solutions we have adopted. Amongst those is the use of SMT that now provides a noticeable gain whereas it was not so clear with previous processors.

The remaining of this paper is organized as follows. We first go through a short description of GYSELA. Then, we describe the transpose-based advection algorithm and analyze its performance. We study the impact of SMT and propose approaches to improve its use. Finally, we conclude the paper by summarizing the gains achieved.

2. GYSELA APPLICATION

The GYSELA code is a non-linear 5D global gyrokinetic full-f code which performs flux-driven simulations of ion temperature gradient driven turbulence (ITG) in the electrostatic limit with adiabatic electrons. It solves the standard gyrokinetic equation for the full-f distribution function, *i.e.* no assumption on scale separation between equilibrium and perturbations is done. This 5D equation is self-consistently coupled to a 3D quasineutrality equation. The code also includes other features not described here (ion-ion collisions, several kind of heat sources). The code has the originality to be based on a semi-Lagrangian scheme [20] and it is parallelized using an hybrid OpenMP/MPI paradigm [5, 14].

The presence of a strong magnetic field in the Tokamak induces a large anisotropy of the transport along and across the magnetic field lines. As a result, the particle density has small structures across the magnetic field lines and large smooth structures along the magnetic field lines. The heat diffusion coefficient is larger by several order of magnitude in the parallel direction than in the perpendicular direction. An anisotropy in the temperature distribution exists also, the perpendicular temperature gradient being much larger than the parallel temperature gradient. Using this strong anisotropy in the numerics enables a considerable reduction of mesh size and thus reduction of computing time. A common approach is to choose dimensions in such a way that the parallel and transverse motion are separated. One solution is to label each magnetic surface using a flux coordinate system. In our case, the coordinates θ and φ lie on the flux surface and represents angles. The r coordinate is normal to the flux surface and is a flux surface label.

Let $\vec{z} = (r, \theta, \varphi, v_\parallel, \mu)$ be a variable describing the 5D phase space. The time evolution of the ion guiding-center distribution function $\bar{f}(\vec{z})$ (main unknown) is governed by the gyrokinetic Vlasov equation:

$$\partial_t \bar{f} + \frac{1}{B^*_{\parallel s}} \nabla_{\vec{z}} \cdot \left(\frac{\mathrm{d}\vec{z}}{\mathrm{d}t} B^*_{\parallel s} \bar{f} \right) = 0 \qquad (1)$$

The guiding-center motion described by the previous Vlasov/transport equation is coupled to a field solver (3D quasi neutral solver which is a Poisson-like solver) that computes the electric potential $\phi(r, \theta, \varphi)$:

$$\frac{e}{T_e}(\phi - \langle\phi\rangle) = \frac{1}{n_0} \int J_0(\bar{f} - \bar{f}_{init})\, \mathrm{d}\mathbf{v} + \rho_i^2 \nabla_\perp^2 \frac{e\phi}{T_i} \qquad (2)$$

We will not describe this last equation (details can be found in [7, 8, 15]). This Poisson-like equation gives the electric potential ϕ at each time step t. One of the difficulty in the gyrokinetic approach is that the Vlasov equation (1) deals with guiding-centers while the quasi-neutrality equation (2) acts on particles. The link between particles and guiding-centers is ensured via a gyroaverage operator. It can be proved that it is equivalent to apply a Bessel function of first order J_0. This gyroaverage operator will not be addressed in this paper but numerical details can be found in [21] and parallelization optimizations for GYSELA in [19]. The derivatives of $J_0\,\phi$ along the torus dimensions are computed. Then, these quantities act as a feedback in the Vlasov equation, they appear into the term $\frac{\mathrm{d}\vec{z}}{\mathrm{d}t} B^*_{\parallel s} \bar{f}$ (not detailed here, see [8]). The Vlasov solver represents the critical CPU part, *i.e.* usually more than 90% of computation time.

3. TRANSPOSE-BASED ADVECTION

This section introduces the *transpose* algorithm and compare it to the original GYSELA algorithm through numerical experiments. The *transpose* algorithm is used by the 2D advection operator and removes a CFL-like condition at the expense of extra communications.

3.1 Algorithm Description

for *time step* $n \geq 0$ **do**
  Field solver, Derivative computation, Diagnostics
  Vlasov solver:
  - 1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
  - 1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
  - **Local 2D Adv.** in (r, θ) $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
  - 1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
  - 1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$

Algorithm 1: using local splines for 2D advection

for *time step* $n \geq 0$ **do**
  Field solver, Derivative computation, Diagnostics
  Vlasov solver:
  - 1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
  - 1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
  - **Transpose** of $\bar{f}$
  - **2D Advection** in (r, θ) $(\forall(\mu, \varphi, v_\parallel) = [local], \forall(r, \theta) = [*])$
  - **Transpose** of $\bar{f}$
  - 1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
  - 1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$

Algorithm 2: transpose of distrib. function for 2D advection

The original version of GYSELA, [7, 14] used a MPI domain decomposition along dimensions μ, r, θ. The μ dimension is at the highest level of parallelism and each μ-value constitutes a MPI communicator. Then, within each communicator dedicated to one μ, a domain decomposition in r and θ is used to distribute the distribution function among MPI processes. OpenMP is used to exploit the parallelism available in the inner loops (along variables $v_\parallel$ or φ typically). We use the following notation throughout the Algorithms of this paper: `local` indicates that in a given dimension each MPI process owns a parallel sub-domain, conversely `*` states that each MPI process possesses all points along a specified direction. In Algorithm 1, one can see that a single domain decomposition is valid for all advections, nevertheless the advection along (r, θ) has to be handled in a special way. Local

cubic splines have been designed years ago [5, 14] in order to perform interpolation on 2D sub-domains during the 2D advection in (r, θ). Typically, only a ghost zone of 3 points is needed at each border of a local sub-domain in r and θ with this approach. These local splines generate few communications between processors while preserving accuracy of the interpolation. However, it limits the displacement during one time step in (r, θ) to one grid cell at the maximum. This introduces a CFL-like condition that prevents use of large time steps [5].

An alternative to local splines is presented here where the MPI domain decomposition is switched between advections as shown in Algorithm 2. The 4D distribution function (for a given μ value) is transposed just before and after the 2D advection along (r, θ). Each processor exchange data to change its sub-domain from $(r = local, \theta = local, \varphi = *, v_\parallel = *, \mu = local)$ to a new sub-domain $(r = *, \theta = *, \varphi = local, v_\parallel = local, \mu = local)$. Even if it implies larger communication volumes, this solution enables one to make use of standard 2D cubic spline over the whole domain $(r = *, \theta = *)$. Hence, the CFL-like condition is removed. Furthermore, other advection schemes that need to consider the whole domain $(r = *, \theta = *)$ are now allowed. Communications of the transpose step have good locality properties, the message exchanges are done inside a μ-communicator that groups together adjacent processes. The numerical accuracy is close for the previous and new solution.

Some values as the J_0 *ϕ-derivatives*, are computed from ϕ. They should be provided as an input to each advection step with the appropriate parallel domain decomposition. Therefore, switching from Algo. 1 to Algo. 2 requires to adapt the communication scheme to send the appropriate J_0 *ϕ-derivatives* to each MPI process. In Algo 1, the domain decomposition of these derivatives matches the main code decomposition $(r = local, \theta = local, \varphi = *, v_\parallel = *, \mu = local)$, they are used equally into 1D and 2D advections. However, in Algo. 2, the new transposed 2D advection algorithm needs additional communications to get the J_0 *ϕ-derivatives* on the new domain decomposition $(r = *, \theta = *, \varphi = local, v_\parallel = local, \mu = local)$. The costs of these additional communications will be exhibited in the next subsection.

3.2 Experimental Evaluation

Timing measurements of this subsection have been realized on the Curie machine at the French GENCI-TGCC-CEA computing center. Each computing node is a dual socket Intel Xeon E5-2680 (Sandy Bridge): 2×(8 cores, 2.7 GHz, 20 MiB shared L3-cache, DDR3 1600 Mhz memory) with 64 GiB of RAM. Let us mention that the thread level for MPI is set to MPI_THREAD_FUNNELED in GYSELA. MPI_THREAD_MULTIPLE approach is avoided because of portability issues on production machines (notably with BullxMPI proposed on several supercomputers). Furthermore, it is not uncommon that codes relying on the level of MPI_THREAD_MULTIPLE run slower than the case where one of the other modes has been chosen [24] (management of locks and thread-safety can be costly).

For a strong scaling experiment, we choose a domain size of $N_r \times N_\theta \times N_\varphi \times N_{v_\parallel} \times N_\mu = 512 \times 512 \times 128 \times 128 \times 32$, representing 1 TiB of data for a single distribution function. Fig. 1, 2, 3 and 4 show a strong scaling from 2k cores up to 65k cores (16 threads per MPI process). On Fig. 1, one can observe a good global scaling behavior for the elapsed

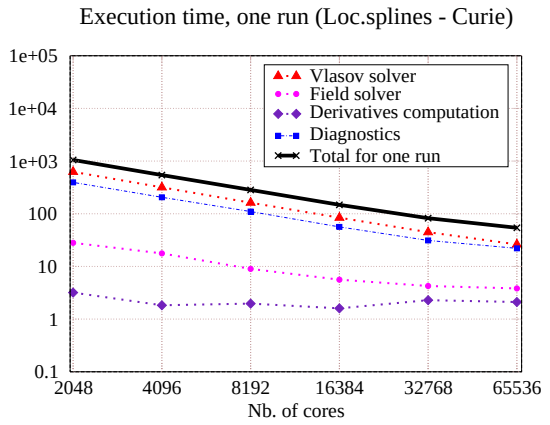

Figure 1: Strong scaling - Local spline - Timing

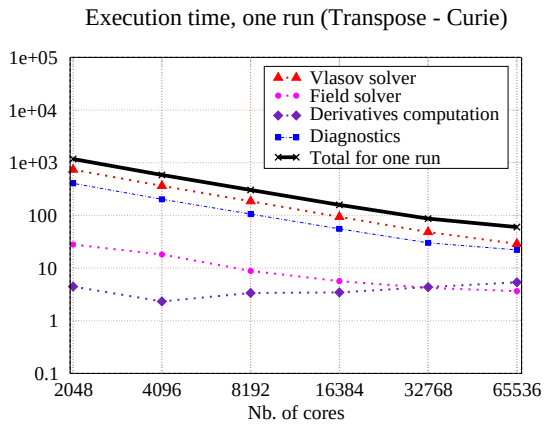

Figure 2: Strong scaling - Transpose - Timing

time of one entire run (thick black line). The Vlasov solver and *diagnostics* computations are processing 5D data and represent the biggest part of computation time; they are approximately divided by a factor two whenever the number of cores is doubling. On the other hand, the *field* solver and the part that gets *derivatives* of the electric potential are dealing with 3D data. These computation kernels are less computation intensive and do not provide as much parallelism as the previous ones. Their respective execution times are relatively low. This is due to dedicated algorithms [15] that: (i) reduce the amount of data transferred in large MPI collective calls within the field solver, (ii) favor communication schemes that send-receive data inside local communicators and (iii) split the field solver into two separate solving steps (treating a one-dimensional equation and then several two-dimensional equations) in order to decrease the overall computation cost.

Let us compare these execution times obtained by the local splines (Algo 1) shown in Fig. 1 with the transpose version (Algo 2) shown in Fig. 2. First, Algo. 2 introduces an overhead due to the transpose communications that represents from 1% up to 20% of the Vlasov solver in production runs compared to the local spline reference time (see Fig. 5 as well). As it can be observed, the overheads are not directly and linearly related to core counts. Another way to say this, transpose communication times are scaling

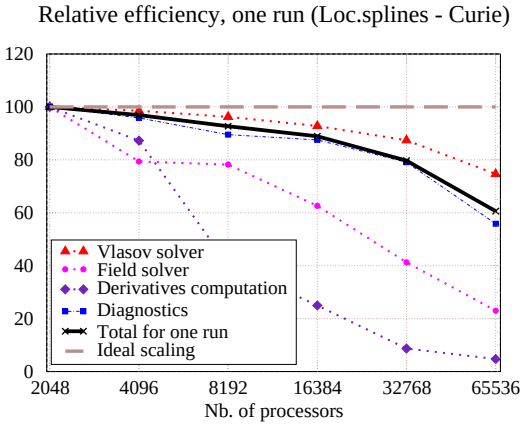

Figure 3: Strong scaling - Local spline - Efficiency

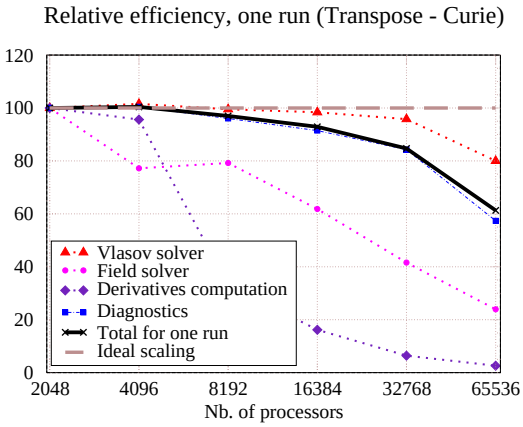

Figure 4: Strong scaling - Transpose - Efficiency

quite well on modern architectures (BlueGene/Q has also been checked [1]). Second, the *derivatives computation* step transmits a larger amount of data with Algo. 2, because the derivatives are distributed to each MPI process according to two different domain decompositions. This increase is significant, but it remains a low overhead relatively to the biggest computational parts (Vlasov solver, diagnostics). The relative efficiencies of Fig. 3 and 4 for the two algorithms are very similar. For the entire application this efficiency is competitive and larger than 60% at 65536 cores in this strong scaling benchmark. Practically, let us notice that physicists run the code between 1k up to 8k cores commonly for this type of domain size (16k cores is uncommon). Typically, we avoid the cases with 32k and 64k cores presented in Fig. 3 and 4. Thus, we mainly target high parallel efficiency for production runs in order to maximize the number of results one can get within the amount of CPU hours that we obtain every year on several supercomputers.

On Fig. 5, 6 and 7, a zoom on the time measurement of the new method compared to the original algorithm is given. Both computations and communications are accounted for in the curves (rationale is that separation of computation versus communications measurements is not easy for subroutines that finely mix both steps). Concerning the *2D advection* part, the difference of the new method compared to the original one tends to decrease considering a high number of cores. The *derivatives computation* does not scale well because the amount of communication ($J_0\,\phi$ derivatives on a 3D sub-domain) is increasing along with the number of cores, even if this growth is sub-linear. To oversimplify, the communication pattern is in-between a MPI scatter and a MPI broadcast of a 3D sub-domain. Finally, we are pleased with these timings for both algorithms as long as their costs stay well below those of main costly steps.

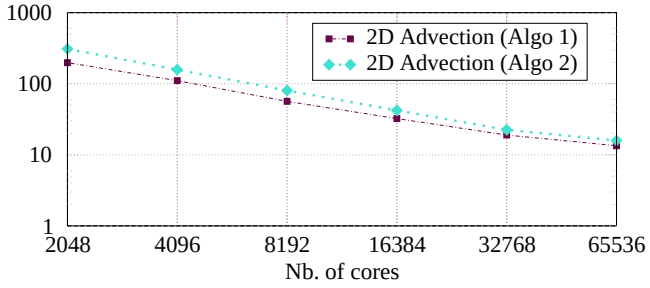

Figure 5: 2D Advection comparison - Timing

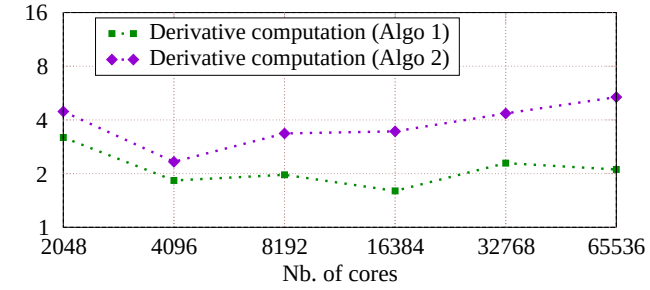

Figure 6: Derivatives comp. comparison - Timing

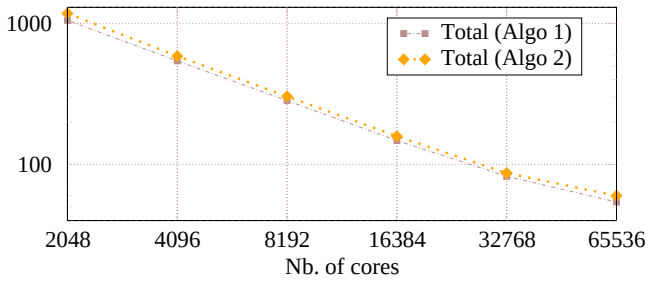

Figure 7: Total run-time comparison - Timing

The expense of this transpose method is an overhead per time step (Fig. 7) that can be up to 12% of the run time (due to the increase by 20% of the Vlasov solver). However, the transpose approach permits to gain more than a factor 10 on the time step in removing the CFL-like condition. Definitely, it seriously shorten global execution time for typical runs. Production runs of GYSELA use, now, exclusively the transpose algorithm.

3.3 Discussion

The transpose method is a great step forward for the Gysela application, we will discuss in the following paragraphs

whether it can bring a benefit to other codes also.

In scientific parallel computing, two communication patterns are useful among others: *Halo communication* pattern and *Redistribution of multi-dimensional arrays*. On the one hand, many *stencil-based* applications in HPC use domain decomposition to distribute the work among different processing elements. Decomposed sub-domains logically overlap at the boundaries and can, depending on the numerical schemes, be updated with neighbor values located on neighbor processing elements. The overlapping regions are called *halo* (or *ghost*) regions. They need to be updated with data from neighbor regions during a *halo communication* step. For example, *Halo communication* strategy together with domain decomposition is classicaly used by explicit methods to solve PDEs.

On the other hand, the operation of *remapping multi-dimensional array* (also called *transpose* method) on computing elements is a common tool. The goal of such a method is to reorganize the data distribution of a multi-dimensional array (of dimension n) across all processes. At start all the elements over m dimensions ($m < n$) of the multi-dimensional array are stored in the local process and the other $n–m$ dimensions are distributed over the processes typically using a domain decomposition. After the transpose step and several communications between processes, the domain decomposition has been switched, each process owns a very new subdomain of the multi-dimensional array. Often, this transpose is required because the numerical scheme expects that at a specific stage, all the the components over one or several dimensions are locally known in the process. A well designed redistribution communication schedule aims to minimize node contentions and maximize network bandwidth utilization. Data redistribution using message passing approach has been extensively studied in literature. Numerous fields use this communication pattern including: Climate and weather forecasting, Geophysics, Computational fluid dynamics, but also FFT libraries for 3D Fourier transform notably. Also, it is common that an explicit method requires impractical small time steps to keep the error low and an implicit method takes much less computational time due to larger time steps. From the parallel computation point of view, an implicit method is often more difficult to parallelize than an explicit method because the solution at a point is dependent on those in the entire domain (no spatial locality). Nevertheless, implicit method is able to reduce the total number of time steps and therefore possibly shorten the total time to solution.

The communication pattern of halo is sparse, whereas the transpose operation involves a dense one. Then, the overall cost of a single halo exchange of a few cells is expected to be a lot cheaper than a transpose step on a multi-dimensional array using many computing units of a supercomputer. Nevertheless, affording the cost of a transpose gives the opportunity to consider alternative efficient numerical schemes (*e.g.* Fourier transform). Examples exist in the literature where the transposition permitted to employ profitable schemes with good scaling on parallel machines [2, 3, 12, 13, 16, 17, 22]. In Gysela case, we have seen that transposition strategy managed to reduce time to solution. It allows us to take larger time steps, and at the same time the overhead in term of communication time remains limited. This conclusion should be also true for other applications that are suffering from stringent CFL condition, thus restricting the time step. Naturally, the ratio of communication time dedicated to the transpose over the useful computations time is a key factor. This ratio should be evaluated on a given target parallel system and target application in order to evaluate possible gains.

In order to reach Exascale, new hardware design approaches are expected to completely change many well accepted idioms for optimization and parallelization. We are told network and memory accesses will not follow the growth of computing power both in term of performance and energy consumption [6] and that, as a result, algorithms will have to be changed for example by recomputing some data so as to reduce stress on these parts. The transpose-based algorithm is one example of optimization that is favorable in terms of computation at the expanse of network bandwidth use. This statement should however be balanced by two observations. First, we can cope with a tripling of this relative cost without incurring a severe penalty on the total execution time. Second, the main problem expected regarding Exascale networks is related to latency, it is not so clear for bandwidth. In any case, we will have to keep comparing the halo vs. transpose based approaches with each new hardware architecture. If the situation was to become too critical, we could consider introducing a pipelined version of the transpose communication pattern in order to partially overlap communication costs with the upcoming computations. Using the current programming model to express such a pipeline might however make the code difficult to read and switching to a task-based model where computations are automatically scheduled when the data becomes available would likely help in this case.

4. SIMULTANEOUS MULTI-THREADING

4.1 Haswell Micro-Architecture

Haswell is based on a 22 nm production process and a new micro-architecture replacing that used in Sandy Bridge. Haswell introduces a huge number of new integer and floating-point vector instructions (AVX2 extension). Amongst those, the fused multiply-add (FMA) combines an addition and a multiplication and is especially important for the HPC market. Indeed it must be used to reach the announced peak FLOPS performance of the processor.

Figure 9 presents the pipeline of the Haswell architecture compared to that of Sandy Bridge in Figure 8. A noticeable change is the addition of two new units to the pipeline: a (vector) integer dispatch port (Port 6) and a memory port (Port 7). In addition, the Haswell floating-point units (Port 0 and Port 1) have been upgraded compared to Sandy, they now have the capability to perform both additions and multiplications. This enables to leverage both units even for applications not perfectly balanced in term of multiply/add. The addition of new execution units combined with the improved capabilities of the existing ones means that it becomes more realistic for the scheduler to submit close to its maximum of 4 μ-operations per cycle thus improving the parallelism at this level.

As with any kind of parallelism, the difficulty with this design is for the code to expose enough in-fly μ-operations from which the scheduler can choose. This is why simultaneous multi-threading[1] (SMT) is more and more important

[1]multiple hardware threads are handled by each core

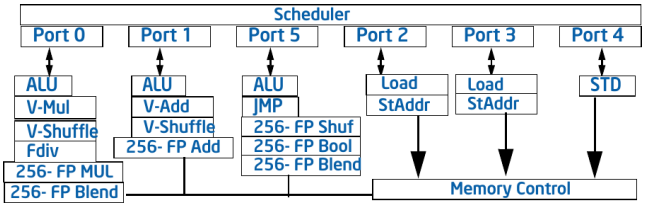

Figure 8: Sandy Bridge CPU Core Pipeline Functionality, extract from [10]

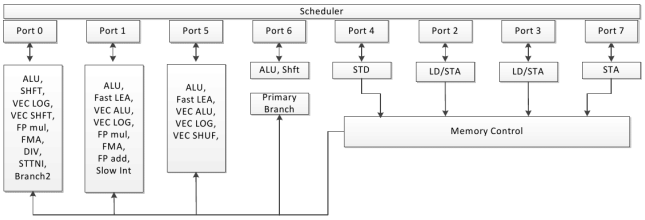

Figure 9: Haswell CPU Core Pipeline Functionality, extract from [10]

to feed all units (also known as a reduction of horizontal waste). In addition, using multiple hardware threads makes it possible to hide latencies related to data access by switching to another thread that is not waiting for data (reduction of vertical waste). All in all, this results in an outright doubling of peak FLOPS in Haswell vs. Sandy which requires for the memory interface to be improved similarly. The L1 bandwidth has been doubled compared to the previous generation, as well as the interface between the L1 and L2 cache.

In the following study, we use the Jureca machine from Jülich/Germany. This supercomputer hosts 1872 compute nodes. Each node contains two INTEL Xeon E5-2680 v3 Haswell CPUs (2×12 cores, 2.5 GHz).

4.2 Performance Metrics

From the user's point of view, the impact of SMT is typically measured by calculating the relative speed-up attained, *e.g.* the code is sped up by x% using SMT compared to one thread per core configuration. In order to measure the core-level effects of SMT, a useful quantity to analyze is the utilization level of the core's execution units. In particular, it is of utmost importance to access a significant fraction of micro-operation slots (executing units) that are available to execute an instruction. If the number of instructions performed per cycle (IPC) is high, then the execution units are being kept busy doing useful work. We will measure IPC in some important kernels of GYSELA. Let us notice that on Haswell processor, IPC can reach a maximal value of 4. This is due to instruction retirement and decode units that can treat up to 4 micro-operations per cycle [9].

4.3 Direct Benefits of SMT

To evaluate SMT, we choose a domain size of $N_r \times N_\theta \times N_\varphi \times N_{v_\parallel} \times N_\mu = 512 \times 256 \times 128 \times 60 \times 32$ in this section. Due to GYSELA internal implementation choices, we are constrained to choose, inside each MPI process, a number of threads as a power of two. Let us remark, that the application performance increases by avoiding very small power of two (*i.e.* 1, 2). Haswell node that we target are made of 24 cores. That is the reason why we choose to set 8 threads per MPI process for the runs shown hereafter. This configuration will allow us to compare easily an execution with or without SMT activated.

In the following, the deployment with 3 MPI processes per node (one compute node, 24 threads, 1 thread per core) is checked against a deployment with 6 MPI processes per node (one compute node, 48 threads, 2 threads per core, SMT used). Strong scaling experiments are conducted with or without SMT, timing measurements are shown in Table 1. Let us assume that processes inside each node is numbered with an index n going from 0 to 2 without SMT, and $n = 0$ to 5 whenever SMT is activated. For process n, threads are pinned to cores in this way: logical cores id from $8\,n$ up to $8\,n + 7$.

Number of nodes/cores	Exec. time (1 th/core)	Exec. time (2 th/core)	Benefit of SMT
22/ 512	1369s	1035s	**-24%**
43/1024	706s	528s	**-25%**
86/2048	365s	287s	**-21%**
172/4096	198s	143s	**-28%**

Table 1: Time measurements for a strong scaling experiment with SMT activated or deactivated, and gains due to SMT

The different lines show successive doubling of the number of cores used. The first column gives the CPU resources involved. The second and third columns highlight the execution time of mini runs comprising 8 time steps (excluding initialization and output writings): using 1 thread per core (without SMT), or using 2 threads per core (with SMT support). The last column points out the reduction of the run time due to SMT comparing the two previous columns. As a result, the simultaneous multi-threading with 2 threads per core gives a benefit of 21% up to 28% over the standard execution time (deployment with one thread per core). While an improvement is expected with SMT, as already reported for other applications (*e.g.* [11, 23]), this speedup is quite high for a HPC application.

We have investigated the most intensive computation parts of the code with Paraver tools (`www.bsc.es/paraver`). The tools are based on traces capturing the detailed behavior of the different MPI processes and threads along time. Calls to the MPI and OpenMP runtime can be enriched with hardware counters, so we were able to measure the instructions and cycles for each computation region. We observe that for each intensive computation kernel the number of instructions per cycle (IPC) cumulated over the 2 threads on one core with SMT is always higher than the IPC obtained with one thread per core without SMT. For these kernels, the cumulated IPC is comprised between 1.4 and 4 for two threads per core with SMT, whereas it is in the range of 0.9 up to 2.8 with one thread per core without SMT. These IPC numbers should be compared to the number of micro-operations achievable per cycle, 4 on Haswell. Thus, we use a quite large fraction of available micro-operation slots. Two factors explain the boost in performance with SMT. First, SMT hides some cycles wastes due to data dependencies and long latency operation (*e.g* memory accesses). Second, SMT enables to better fill available execution units. It provides a remedy against the fact that, within a cycle, some issue

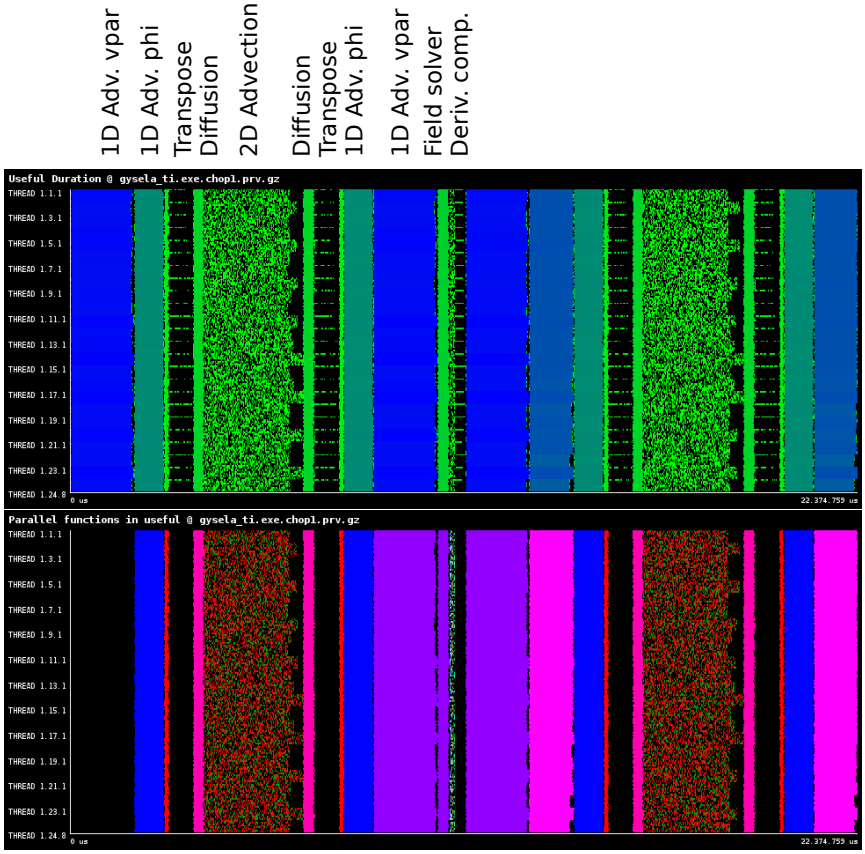

Figure 10: Snippet of a run with 2 threads per core (SMT), Top: Paraver useful duration plot, Bottom: Parallel functions plot

slots are often unused.

4.4 Optimizations to Increase SMT Gain

The Paraver tool gives us the opportunity to have a view of OpenMP and MPI behaviors at a very fine scale. The visual rendering informs rapidly the user of an unusual layout and therefore hints to look on some regions with unexpected patterns. On the Fig. 10 is plotted a snippet of the timeline of a small run with SMT (2 threads per core, 24 MPI processes, 8 threads per MPI process, meaning 4 nodes hosting 48 threads within each node). We can extract the following information:

1. The 2D advection kernel (first computationally intensive part of the code) is surprisingly full of small black holes.

2. There are several synchronizations during this timeline between MPI processes that are noticeable. As several moderate load imbalances are also visible, a performance penalty can be induced by these synchronizations. See for example 2D advection and Transpose steps (Useful duration plots), there is much black color at the end of these steps. This is due to final MPI barriers. Nevertheless the impact is relatively low in this reduced test case because the tool reported a parallel efficiency of 97% over the entire application indicating that only 3% of the iteration time is spend on the MPI and OpenMP parallel runtimes. The impact is stronger on larger cases, because load imbalance is bigger.

3. The transpose steps show a lot of black regions (threads remaining idle). Fig. 11 zooms into the transpose kernel execution for the MPI ranks showing larger communication times for the higher ranks despite they use the same communication pattern (in this plot, one colored bar represents one entire MPI process, no distinction by thread). At the end of the phase, all the ranks are synchronized by the MPI_Barrier. The useful duration plot shows this delay is caused by a larger duration of the initial computation phase on only few MPI processes. Checking the hardware counters indicate the problem is related with a different IPC where the fast processes are getting twice the IPC of the delayed ones. This behavior illustrates well that SMT introduces heterogeneity of the hardware that should be handled by the application even if the load is well balanced between threads.

These inputs from the Paraver visualization helped us to determine some code transformations to make better use of unoccupied computational resources. The key point was to point out the cause of the problem, the improvements were not so difficult to put into place. The upgrade are described in the following list. The Table 2 and Fig. 12 exhibits associated measurements.

1. The 2D advection kernel is composed of OpenMP re-

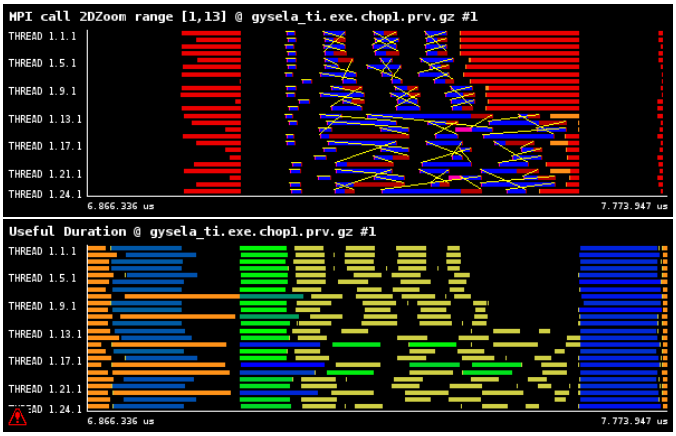

Figure 11: Zoom on the Transpose kernel (only MPI ranks are displayed), Top: Paraver MPI calls plot, Bottom: useful duration plot

gions. There is mainly an alternation of two distinct OpenMP kernels. The first one fills the input buffers to prepare the computation of 2D spline coefficients for a set of N poloidal planes (corresponding to different $\varphi, v_\parallel$ couples). The second kernel computes the spline coefficients for the same N poloidal planes and performs the advection itself that encompasses an interpolation operator. Yet, there is no reason for having two separate OpenMP regions encapsulated in two different routines, apart from historical ones. Thus, we decided to fuse these OpenMP regions in a single large one. This modification avoids the overheads due to entering and leaving the OpenMP regions multiple times. Also the implicit synchronization at the beginning and end of each parallel region are removed. Thus, avoiding synchronization leads to a better load balance by counteracting the imbalance originating mainly from the SMT effects.

2. Some years ago, with homogeneous computing units and resources, the workload in GYSELA was very balanced between MPI processes and inside them, between threads. Thus, even if some global MPI barriers were present within several routines, they induced negligible extra costs because every task was executed synchronously with the others. In latest hardware, there is heterogeneity coming from cache hierarchy, SMT, NUMA effects or even Turbo boost. The penalty due to MPI barriers is now a key issue, and thread idle time is visible on the plot. We removed several useless MPI barriers. As a result, we see for example in Fig. 12 that, now, diffusion is sandwiched between the transpose step and the 2D advection, without global synchronization.

3. The transpose step is compounded of three sub-steps: copy of data into send buffers, MPI non-blocking send/receive calls with a final wait on pending communications, copy of receive buffers into target distribution function. On the Fig. 10, it is worth noticing that only the first thread of each MPI process is working, *i.e.* only the master thread is performing a useful work. To improve this, we added OpenMP directives to parallelize all the buffers' copies. This modification increases the extracted memory bandwidth and the thread occupancy. On the Fig. 12, the bottom plot shows that the transpose step is now partly parallelized with OpenMP.

Thanks to these upgrades, there is much less black (idle time) in Fig. 12 compared to Fig. 10. Still, MPI communications induce idle time for some threads in the transpose step and in the field solver. This can not be avoided within the current assumptions done in GYSELA. Table 2 also illustrates the achieved gain in term of elapsed time. If one compares to Table 1, the timings are reduced with one or two threads per core. Comparing one against two threads per core, the SMT gain is still greater than 20% (almost the same statement as before optimization). Now, if we cumulate the gain resulting from SMT and from the optimizations, we end up with a net benefit on execution time of 32% up to 38% depending on the number of nodes.

Number of nodes/cores	Exec. time (1 th/core)	Exec. time (2 th/core)	Benefit of SMT	Benefit vs. Table 1
22/ 512	1266s	931s	**-26%**	**-32%**
43/1024	631s	474s	**-25%**	**-33%**
86/2048	320s	239s	**-25%**	**-34%**
172/4096	164s	124s	**-25%**	**-38%**

Table 2: Time measurements and gains achieved after optimizations that remove some synchronizations and some OpenMP overheads

5. CONCLUSION

GYSELA achieves a good parallel computation scalability up to 64k cores combining several levels of parallelism and a hybrid OpenMP/MPI approach. The transpose algorithm helps to reduce the CFL and represents a scalable and robust solution to handle different physics regimes. Global execution times are greatly reduced thanks to this upgrade.

Simultaneous multi-threading of Haswell architecture enables a program to better use each core efficiently with multiple threads. Therefore, by using multiple threads on a single core, we are able to increase the cumulated throughput of all available integer units and floating-point units. In GYSELA code, it leads to a significant reduction of the execution time, larger than 20% in typical cases. Some adaptations of MPI and OpenMP usage to avoid synchronization that conflicts with threads and SMT further improve performance. All in all, this leads to large gain in execution time whenever one uses several thousands of cores, compared to the initial version without SMT we notice an improvement from 32% up to 38% depending on the test case.

We observe that SMT technology introduces observable benefits, but also introduces heterogeneity of the hardware throughput. Well balanced applications such as GYSELA should revise their parallelization strategy in order to deal with this kind of imbalance due to hardware characteristics.

6. ACKNOWLEDGMENTS

This work was strongly supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon 2020 framework of the European Union. We gratefully acknowledge the POP project, that has also received funding from the European Union's Horizon 2020 research and innovation programme

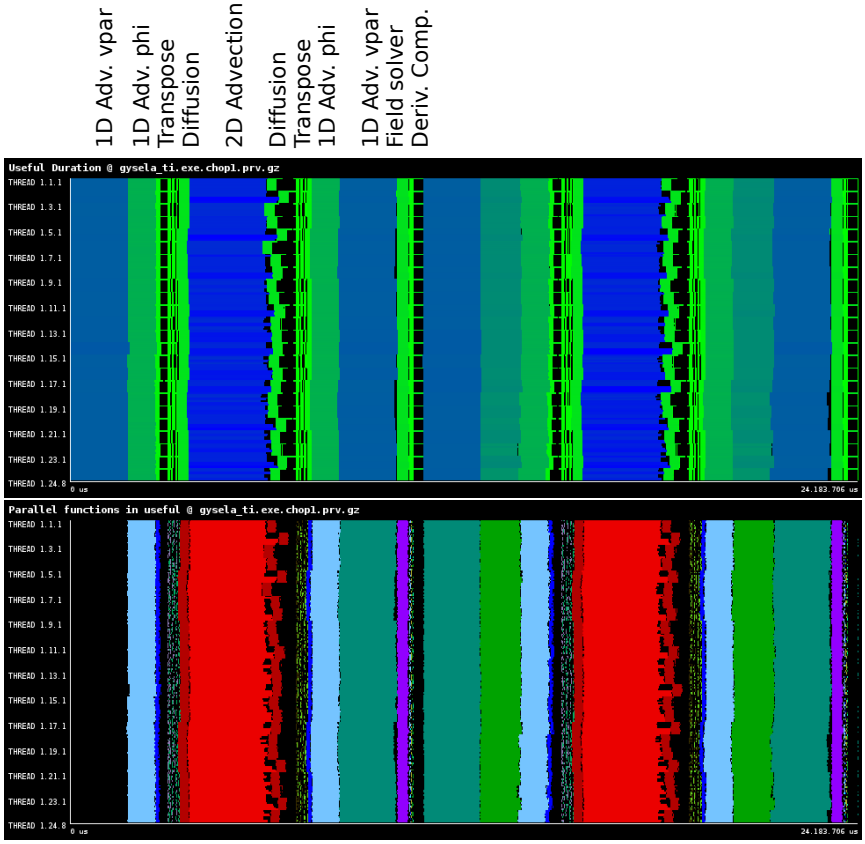

Figure 12: Snippet of a run with 2 threads per core (SMT), after optimizations are done, Top: Paraver useful duration plot, Bottom: Parallel functions plot

under grant agreement No 676553. This work was possible due to the generous computational and software engineering supports from FZJ Juelich (Germany) and CCRT Bruyères-le-Châtel (France). The authors would also like to thank Chantal Passeron for assistance and for her precious help.

7. REFERENCES

[1] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine. Scaling gysela code beyond 32K-cores on bluegene/Q. In *CEMRACS 2012*, volume 43 of *ESAIM: Proc.*, pages 117–135, Luminy, France, 2013.

[2] C. Christara, X. Ding, and K. Jackson. *High Performance Computing Systems and Applications*, chapter An Efficient Transposition Algorithm for Distributed Memory Computers, pages 349–370. Springer US, Boston, MA, 2000.

[3] A. W. Cook, W. H. Cabot, P. L. Williams, B. J. Miller, B. R. de Supinski, R. K. Yates, and M. L. Welcome. Tera-Scalable Algorithms for Variable-Density Elliptic Hydrodynamics with Spectral Accuracy. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*.

[4] N. Crouseilles, G. Latu, and E. Sonnendrücker. Hermite Spline Interpolation on Patches for Parallelly Solving the Vlasov-Poisson Equation. *Applied Mathematics and Computer Science*, 17(3):335–349, 2007.

[5] N. Crouseilles, G. Latu, and E. Sonnendrücker. A parallel Vlasov solver based on local cubic spline interpolation on patches. *Journal of Computational Physics*, 228:1429–1446, 2009.

[6] E. D'Hollander, J. Dongarra, I. Foster, L. Grandinetti and E. G. Joubert Eds. *Transition of HPC Towards Exascale Computing*. IOS Press, 2013.

[7] V. Grandgirard, M. Brunetti, P. Bertrand, N. Besse, X. Garbet, P. Ghendrih, G. Manfredi, Y. Sarazin, O. Sauter, E. Sonnendrucker, J. Vaclavik, and L. Villard. A drift-kinetic Semi-Lagrangian 4D code for ion turbulence simulation. *Journal of Computational Physics*, 217(2):395 – 423, 2006.

[8] V. Grandgirard et al. A 5D gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations. Submitted to CPC. `https://hal-cea.archives-ouvertes.fr/cea-01153011`, July 2015.

[9] D. Hackenberg, R. Schone, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 896–904, May 2015.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-031. September 2015.

[11] S. Jarp, A. Lazzaro, J. Leduc, and A. Nowak.

Evaluation of the Intel Sandy Bridge-EP server processor. Technical Report CERN-IT-Note-2012-005, CERN, Geneva, Mar 2012.

[12] M. Kuhn, G. Latu, N. Crouseilles, and S. Genaud. Parallelization of an advection-diffusion problem arising in edge plasma physics using hybrid MPI/OpenMP programming. In *Euro-Par 2015: Parallel Processing, Proceedings*, pages 545–557, 2015.

[13] S. Laizet and N. Li. Incompact3d: A powerful tool to tackle turbulence problems with up to $o(10^5)$ computational cores. *International Journal for Numerical Methods in Fluids*, 67(11):1735–1757, 2011.

[14] G. Latu, N. Crouseilles, V. Grandgirard, and E. Sonnendrucker. Gyrokinetic semi-Lagrangian parallel simulation using a hybrid OpenMP/MPI programming. In *Recent Advances in PVM and MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 356–364. Springer, 2007.

[15] G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable quasineutral solver for gyrokinetic simulation. In *PPAM (2)*, LNCS 7204, pages 221–231. Springer, 2011.

[16] D. Pekurovsky. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. *SIAM Journal on Scientific Computing*, 34(4):C192–C209, 2012.

[17] M. Pippig. PFFT: An Extension of FFTW to Massively Parallel Architectures. *SIAM Journal on Scientific Computing*, 35(3):C213–C236, 2013.

[18] F. Rozar, G. Latu, J. Roman, and V. Grandgirard. Toward memory scalability of Gysela code for extreme scale computers. *Concurrency and Computation: Practice and Experience*, 27(4):994–1009, 2015.

[19] F. Rozar, G. Steiner, Ch. Latu, M. Mehrenberger, V. Grandgirard, J. Bigot, T. Cartier-Michaud, and J. Roman. Optimization of the gyroaverage operator based on Hermite interpolation. In *CEMRACS 2014*, volume accepted of *ESAIM: Proceedings.*, Luminy, France, 2015.

[20] E. Sonnendrücker, J. Roche, P. Bertrand, and A. Ghizzo. The Semi-Lagrangian method for the numerical resolution of the Vlasov equation. *Journal of Computational Physics*, 149(2):201 – 220, 1999.

[21] C. Steiner, M. Mehrenberger, N. Crouseilles, V. Grandgirard, G. Latu, and F. Rozar. Gyroaverage operator for a polar mesh. *The European Physical Journal D*, 69(1):18, 2015.

[22] S. Stellmach and U. Hansen. An efficient spectral method for the simulation of dynamos in cartesian geometry and its implementation on massively parallel computers. *Geochemistry, Geophysics, Geosystems*, 2008.

[23] P. Szostek, A. Nowak, G. Bitzes, L. Valsan, S. Jarp, and A. Dotti. Beyond core count: a look at new mainstream computing platforms for HEP workloads. *Journal of Physics: Conference Series*, 513(6):062036, 2014.

[24] R. Thakur and W. Gropp. Test Suite for Evaluating Performance of Multithreaded MPI Communication. *Parallel Comput.*, 35(12):608–617, Dec. 2009.