**UNIVERSITÉ DE STRASBOURG**

**HABILITATION À DIRIGER LES RECHERCHES**

Informatique et calcul scientifique

présentée par

**Guillaume LATU**

ÉCOLE DOCTORALE MATHÉMATIQUES, SCIENCES DE L'INFORMATION ET DE L'INGÉNIEUR - ED 269

---

# Contribution à la simulation haute-performance et aux méthodes de calcul très extensibles

---

# Contribution to high-performance simulation and highly scalable numerical schemes

---

**Soutenue le :** 18 Mai 2018, 13h30, à Strasbourg (laboratoire IRMA)

**Devant la commission d'examen composée de :**

| | | |
|---|---|---|
| Raymond Namyst ..... | Professeur, Université de Bordeaux | Rapporteur |
| Frédéric Desprez ..... | Directeur de Recherche, INRIA, Grenoble | Rapporteur |
| Rémi Abgrall ..... | Professeur, Université de Zurich | Rapporteur |
| Eric Sonnendrücker ..... | Professeur, TUM, Allemagne | Examinateur |
| Jean Roman ..... | Professeur, INP (Enseirb-Matmeca) & INRIA, Bordeaux | Examinateur |
| Stéphane Genaud ..... | Professeur, ENSIIE, Strasbourg | Examinateur |

---

*Occigen suppercomputer (CINES - Montpellier)*

# Remerciements

Je tiens d'abord à remercier Stéphane Genaud, Philippe Clauss et Eric Sonnendrücker pour leur accueil à mon arrivée à Strasbourg en 2003 et leur invitation à faire partie de leurs équipes ICPS et CALVI. Cette HDR n'aurait pas été possible sans leur soutien.

Je tiens également à remercier l'ensemble de mes collaborateurs avec qui j'ai eu la chance et le plaisir d'interagir depuis 2009 au CEA: physiciens, doctorants, postdocs et bien d'autres. Xavier Garbet, Philippe Ghendrih, Yanick Sarazin, Guillhem Dif-Pradalier m'ont accompagné dans mon apprentissage de la physique des plasmas de tokamaks ; mais pas seulement, ils se sont ausi impliqués dans les sciences et techniques du HPC à mes côtés. Julien Bigot, Virginie Grandgirard et Chantal Passeron ont partagé avec moi tellement de lignes de codes, mais aussi tant de grands challenges sur des super-calculateurs tout neufs, et encore des courses effrénées pour éliminer des bogues et autres anomalies dans Gysela. Merci à vous tous pour ces activités interdisciplinaires qui sont si riches en enseignements.

En toute humilité, le travail de recherche que j'ai mené depuis 2003 a été effectué en étroite association avec 7 thésards et 3 postdocs que je cite ici: Gaël Tessier, Matthieu Haefele, Rached Abdelkhalek, Matthieu Kuhn, Xavier Lacoste, Fabien Rozar, Nicolas Bouzat, Marc Sauget, Olivier Thomine, Yuuichi Asahi. Ces travaux que nous avons réalisés constituent avant tout une aventure humaine que les qualités de chacun font avancer et grandir.

Depuis mes débuts dans la recherche, mes activités se sont déroulées avant tout en collaboration et dans plusieurs équipes. Je remercie mes collègues de l'ICPS, du laboratoire iCube et du département Math-Info de Strasbourg, ainsi que ceux de l'IRMA et des équipes-projets INRIA CALVI, SCALAPPLIX, TONUS. Avec eux, j'ai partagé les problèmes et les plaisirs quotidiens durant bien des années. J'ai une pensée particulière pour Nicolas Crouseilles, Michel Mehrenberger, Alain Ketterlin, Pierre Gançarski, Romaric David, Pierre Ramet, Pascal Hénon, qu'ils sachent combien j'ai apprécié de travailler avec eux.

Je tiens aussi à adresser des remerciements aux personnels du centre de calcul du CINES que j'ai côtoyés ces dernières années, et tout spécialement à Gérard Gil, Bertrand Cirou et Gabriel Hautreux qui m'ont démêlés de nombreux sacs de nœuds sur les supercalculateurs.

Un grand merci à Rémi Abgrall, Frédéric Desprez, Raymond Namyst, qui ont répondu favorablement à ma requête pour être rapporteurs de cette HDR. Au-delà de la démarche scientifique du chercheur, il me semble que tout processus de recherche implique invariablement un fort engagement personnel. J'ai sollicité leur concours ayant reconnu en eux cet engagement. Au-delà des agendas surchargés, ils continuent d'initier avec d'autres, de nouvelles actions de recherche.

Je tiens notamment à exprimer ma gratitude à Jean Roman, qui après ma thèse, a continué à m'accompagner dans de nombreuses activités de recherche. Ses conseils, son soutien, ses encouragements permanents tout au long de ces 20 dernières années, ainsi que son dynamisme inoxydable, ont joué un rôle primordial dans mon travail.

Enfin, mes remerciements vont pour sûr à ma famille et à mes amis dont la présence m'apporte toujours du bien. La part d'honneur revient, bien sûr, à Isabelle et Adam qui me portent au jour le jour.

# Contents

# Chapter 1

# Introduction

*"The man who moves a mountain begins by carrying away small stones."*

Confucius

Numerous scientific domains express a need for high-performance computing (HPC), which has intensified in recent decades. While still evolving, the technologies of single-processor machines are insufficient. Parallelism is a natural solution to meet the needs of the largest simulation applications. At the same time, the size of supercomputers available to the academic community has grown steadily. The modern hierarchical computing infrastructures are hard to program and to use efficiently, particularly for extreme-scale computing. Consequently, looking for new solutions that are able to properly handle such environments is an active field of research. The increase in computational power and the development of numerous algorithmic and methodological tools have combined to make numerical simulation a discipline in its own right. Thus, the development of parallel software and hardware methods and solutions has become a major focus of academic and industrial research. Although many domains benefit from the spin-offs of numerical simulation innovations, this field is at the crossroads of many domains and therefore not very structured and insufficiently identified yet.

During the second half of the twentieth century, numerical simulation was mainly aimed at reducing costs and delays, supplementing experiments whose modeling was well controlled. At the end of the twentieth century, it has contributed to technical innovation, allowing to take into account increasingly complete physical models and to access information that is difficult to obtain by measurements. Today, it also helps to develop new physical models, for example by integrating the combined effects of simple phenomena by calculation. Many scientific and technical projects are developed within the framework of a collaborative approach between theory, simulation and experimentation (for example, CEA[1], implements this approach). The comparison of the simulation versus the experimentation can lead to a questioning of the physical model as well as the numerical methods. It is also possible to develop numerical devices or experiments numerically without carrying out full-scale real-world experiments whose costs are often prohibitive.

I realized much of my research work in such a context, in interaction with several scientific fields. My contributions concern the improvement of computational methods from the point of view of parallelization, the design of optimized algorithms and implementations for specific machines, but also on the upgrade of some numerical schemes. In the various simulation codes on which I have been involved, it has been required to adapt or renovate calculation schemes for efficient execution on machines with a large number of cores. My inputs contribute throughout the entire chain, from modeling, to efficient implementation on large supercomputers. The studies I carried out were done in close interaction with the designers and users of the simulation codes so that the results are effective and usable in production. This would not have been possible without mutual efforts of understanding and adaptation with my collaborators: physicists, mathematicians, and computer scientists.

The document is composed of three main chapters plus a final one enclosing conclusions and perspectives. Although my scientific work is not limited to Gysela, I chose here to focus a large part of the document on this application in order to simplify the contextual setting and to allow me to get into some of the details.

Chapter 2 concerns various studies carried out to make the best use of the today's supercomputers. After a short description of the physical problem in Gysela, the numerical methods are described,

[1]French Alternative Energies and Atomic Energy Commission (CEA) is a key player in research, development and innovation in four main areas: defence and security, nuclear and renewable energies, technological research for industry, fundamental research in the physical sciences and life sciences.

and then I discuss the improvements required to exploit the biggest academic calculators in Europe. In its beginnings, the GYSELA code treated the so-called *Gyrokinetic Vlasov* equation coupled to a *Poisson* solver without any additional operator. This was a 5D-Vlasov equation using cylindrical geometry with $(r, \theta, \varphi)$ the space variables and $(v_\parallel, \mu)$ the velocity variables. The semi-Lagrangian method was chosen to solve the Vlasov part. At that time, this choice was original because it differs from the Eulerian and Lagrangian approaches that were quite standard for this kind of code. In the early 2000s, the collaborations between several INRIA teams, the LPMIA at Nancy University and the CEA IRFM clearly spurred this choice. Since 2005, the major digital bottlenecks have been cleared step by step and the scalability on the largest accessible parallel machines has been consolidated. In 2006, the reduced 4D version of the code performed well on 128 processors. The definition of the local splines method [9, 14, 35] allowed to enhance the parallel scalability while preserving the numerical quality in a set of applications among which GYSELA. In 2007, thanks to an adapted MPI+OpenMP parallelism, I obtained a relative efficiency of 82% on 4096 cores for a strong scaling of a cylindrical GYSELA-5D case on a BULL/INTEL itanium2 machine. More recently, these splines have been challenged by high-order Lagrange interpolators [81] which are currently used in simulations. In 2010, a Grand Challenge[2] achieved 81 % of relative efficiency on 8192 cores on a SGI machine (18th position at top500 list[3]) at the CINES computing facility. On the other hand, a simulation close to the non-dimensional parameters of the ITER Tokamak [4] (International Thermonuclear Experimental Reactor) was carried out, using 272 billion grid points in 5D space. The code version, available then, incorporated many novelties in terms of physics to achieve realistic setting: heat source, collisionality, toroidal geometry. A 60MW power source forced the plasma out of thermodynamic equilibrium, generating turbulence and self-organization that we could follow during 1 ms. This simulation required 6.1 million CPU hours, which was performed during 31 days on 8192 cores. By introducing two new domain decompositions and additional parallel algorithms, it has been possible to globally reduce the volume of communications within and at the end of the Poisson solver, thus shortening restitution time. The major issue for achieving good scalability beyond 8k was then pushed back beyond 65k cores [26, 29, 91, 92]. Then, a different bottleneck appeared, the memory scalability was not excellent whenever performing big physical cases. When doubling the number of cores for a given simulation setting, the memory footprint was far from halved. Many very large physical cases were impossible to run because of memory exhaust. By introducing more complex algorithms, by adding communications, it has been possible to make the memory costs associated with the 3D structures scalable along with the number of cores. In 2013, the memory scalability was significantly improved [4, 26, 22, 29, 83]. The adaptation to the IBM BlueGene/Q machine has also led to extending the scalability limits. GYSELA is a member of the Hi-Q club (Highest Scaling Codes on JUQUEEN) with 91% relative efficiency on 458 752 cores (Weak scaling) on the whole super-calculator Blue Gene Juqueen (Juelich, Germany). Works to get better reproducibility and to improve the validation and robustness of the code have been conducted [19]. Thanks to its very good scalability and its portability (the code is deployed on ten computers permanently), GYSELA frequently uses 8k to 32k cores. In addition, a simulation often takes several weeks. The annual consumption of computing time is steadily increasing and is currently over 90 million mono-processor hours per year (figures for 2016/2017). Recently, in order to make the best use of the latest INTEL and IBM architectures, I was able to optimize several parts of the code so that several threads could be executed on each computing core. Although GYSELA is well balanced in terms of distributing computations between execution units, it seems that computing resources are less homogeneous than before, due to competition over resources: caches, sharing of computational units by the threads running on the same core, NUMA effects [18]. This implies that the structure of the code will have to be revised in the near future to match the rapid evolution. Task-based programming is a way that I am currently investigating. Also, synchronizations induced by the management of parallelism (BSP model) weigh more and more on large platforms and the task-based approach will partially remedy the problem. In any case, the gyrokinetic codes are good candidates to test, as soon as they appear, the exaflopic machines (able to perform $10^{18}$ floating point operation per second). Moreover, the gyroaverage operator is a cornerstone of the gyrokinetic theory and represents a significant cost in GYSELA. This operator transforms the so-called guiding-center distribution into the actual particle distribution. It is essential to adapt the code to the next generations of machines so that the gyroaverage becomes scalable. Several works with multiple collaborators have led to great progress on the accuracy and speed of calculation of this operator [5, 17, 80, 83], for which overlapping communications by calculations is a key component.

Along with the efforts for achieving good parallelization, I also contributed to the numerical methods in several applications to improve the precision or the realism of the simulations, but also to

[2] `https://www.cines.fr/wp-content/uploads/2014/02/GazetteGD2010.pdf`
[3] https://www.top500.org/system/176897
[4] `https://www.iter.org`

accelerate the calculations. While Chapter 2 of the document is focusing on parallel computing, Chapter 3 summarizes works in closer connection with the field of applied mathematics. The implementation of specific test cases within GYSELA and the adaptation of numerical methods in the Vlasov and Poisson solvers make it possible to better preserve certain invariants and improve the precision of the code [7, 89]. Incidentally, I designed and contributed to the implementation of a continuous integration platform to ensure systematic tests leading to better code robustness for users [19]. A series of theoretical studies have established that the alignment of physical structures around the magnetic field lines can be used to reduce the number of necessary mesh points in the direction which is parallel to the field lines. I implemented a new numerical method with aligned interpolation for GYSELA in close collaboration with the designers of the Selalib library (a group of mathematicians). This approach based on a fine understanding of the physical processes effectively saves a lot of meshing points and thus reduces the cost of simulations [1, 85]. An hypothesis was originally made in the GYSELA code concerning the geometry of the poloidal plane (plane which is transverse to the field lines): the polar coordinate system was chosen to represent a circular plasma. This was appropriate a few years ago to model the plasma of the Tore Supra Tokamak (CEA IRFM) which was circular. This is no longer the case today, as the current Tokamaks have a more sophisticated geometry: with X-point, double X-point, snowflake configuration. On the other hand, GYSELA has long taken as a simplifying hypothesis (for a better robustness of numerical methods) the existence of a central hole in the poloidal plane around the point $r = 0$ (at magnetic axis). Advances have led to a much better modeling of the poloidal plane and improves the realism of the simulations, the central hole has now disappeared from most simulations [27, 81]. In addition, methods for modeling non-circular plasma are being evaluated.

Chapter 4 focuses on work on the development of parallel algorithms and the implementation of optimization techniques dedicated to new architectures. Studies on various applications are highlighted. I draw some insights from lessons learned on these new computing devices. A parallel solution for petroleum exploitation was developed on a cluster of GPUs (RTM methods - *Reverse Time Migration*). The memory access patterns and the management of CPU-GPU and MPI communications play a major role, they were the main bottlenecks [8, 31, 87]. Nevertheless, speedups are substantial on GPUs compared to the conventional architectures for this application. But the adaptation of the initial code and the maintenance of several versions (CPU cluster + GPU cluster) remain a cost in human resources that can not be neglected. In addition, a Vlasov-Poisson model, not so far from GYSELA equations, was studied on a single GPU card. The organization of memory access and the development of very fine-grained algorithms are important to focus on from the performance point of view [30]. The overhaul of the original code was inevitable. From these experiments, we deduce that considering a solution using GPU in an application of the same size as GYSELA would require rewriting the code in depth. Indeed, it is hard to design algorithms with sufficiently adaptive graininess of the computations to match specific hardware requirements. I realized some optimization works on some GYSELA kernels on the Intel KNC coprocessor that appeared in 2012 (also called Xeon Phi). A major problem here is to adequately vectorize, because it is an essential condition to obtain reduced execution times. Some *memory-bound* and *compute-bound* kernels were accelerated by a factor of two on the coprocessor compared to the INTEL Sandy Bridge [2, 20], which was a good result. Again, the access patterns to the memory represent a real challenge, a lot more than for a standard CPU architecture, as well as the fine management of the data locality within the cache. Because of many difficulties, it is not easy to achieve good performance levels in a large number of routines for a large production application such as GYSELA. More recently, the appearance of production platforms using Intel KNL processors, the next generation after KNCs, have changed the landscape of the HPC. These computing devices are quite close to traditional architectures (they do not need a host device as KNC and GPU do), but with higher peak performance and noticeable energy efficiency improvements. Auto-tuning techniques have also helped to address some of the challenges that these machines offer for the GYSELA code [82].

The very last chapter 5 gives a conclusion of the previous chapters, and outlines some of the research projects I plan for the years to come. One of the constant problem facing the parallel application developer is to find a compromise between efficiency, portability and code readability. The complexities of hardware, of applications and the difficulty to choose a programming model remain major issues. My aim is to help GYSELA cross over the obstacles and to end up soon running on an Exascale machine.

A reader interested in applied mathematics in the first place could start with the introduction given at the beginning of Chapter 2 up to p.10- 17, and then jump directly to Chapter 3 p. 56-92. On the contrary, a reader most interested in high-performance computing can surely skip the reading of Chapter 3.

# Chapter 2

# Parallel solutions for numerical simulations

*" If knowledge can create problems, it is not through ignorance that we can solve them."*

Isaac Asimov
Asimov's guide to science

Compared to serial computing, parallel computing is much better suited for modeling, simulating and understanding complex, real world phenomena. First, throwing more resources (processing components) at a task is able to shorten its time to completion thus reducing from several months on a single processor to a few hours on a large supercomputer. Second, many problems are so large or so complex that it is impractical or even impossible to solve them on a single computer, especially given the limited computer memory. But in order to use parallel computers, mathematical methods and parallel algorithms have to be designed carefully, and one should also keep up with hardware upgrade along time. Four main lines of actions can improve our use of supercomputers evolution nowadays: adapt parallel software to the still increasing number of cores, exploit processors more efficiently (*e.g* optimizing the use of cache memory and instruction set), design architecture-friendly algorithms and try to target portability of performance, look for algorithms and numerical schemes delivering more results per arithmetic operation (mesh adaptivity, high-order methods, mixed-precision). Along this line, this chapter is devoted to the presentation of the main parallel solutions that I designed the last few years for the Gysela code. This exemplifies how research on parallel algorithms permits to go for larger and more realistic plasma physics simulation. The first part is dedicated to a brief description of the context: main trends in HPC over the past years, the main physics aims in Gysela, the magnetic configuration of the tokamak, the gyrokinetic model and major numerical schemes. Next, the improvements of the efficiency of the central Vlasov and Poisson solvers are detailed. These contributions have permitted the Gysela users to target larger simulations employing more cores, but have also allowed us to reduce restitution time. Algorithms and solutions are given concerning the memory scalability and the calculation of the gyroaverage operator. Parallelization and performance issues are explained. As a conclusion of this chapter, additional references are mentioned; they highlight some of my other works focusing on parallel algorithmic that are not related at all to Gysela.

## 2.1 High-Performance Computing

### 2.1.1 Introduction

High-Performance Computing is employed to solve complex issues in computational and data-intensive sciences. For problems where experiments are impossible, dangerous, or too costly, HPC permits predictive modeling and analysis of massive quantities of data. Throughout the past 20-years, high-tech industries such as transportation, aerospace, nuclear energy, and petroleum have adopted and used HPC to accurately represent multiscale phenomena, to simulate, and to gain insights to better predict and understand large or complex systems. Furthermore, in the field of simulation, HPC has the potential to suggest new experiments not foreseen before.

Although some excitement exists about the largest supercomputers and on specific benchmarks, such as TOP500 [189], there is a much deeper commitment from international scientific community in HPC. Many actors and organizations have spent much resource developing tools, methods, schemes, software and applications which are nowadays part of the foundation of HPC ecosystem. The past twenty years have shown a large increase in both the use and scale of parallel machines. In 1997,

the ASCI Red system at Sandia Labs broke the one Terascale[1] barrier on the TOP500 (LINPACK benchmark) with a bit more than 9,000 cores. It was the beginning of the *Terascale* era. After a while, in 2008, Roadrunner machine reached PFLOPS ($10^{15}$ FLOPS). Peta-scale systems usually have more than 100,000 computing cores and an interconnected multi-socket, multi-core architecture. Compared to Tera-scale situation, significantly higher parallelism is required for peta-scale algorithms to perform well without significant overheads.

### 2.1.2 Trends

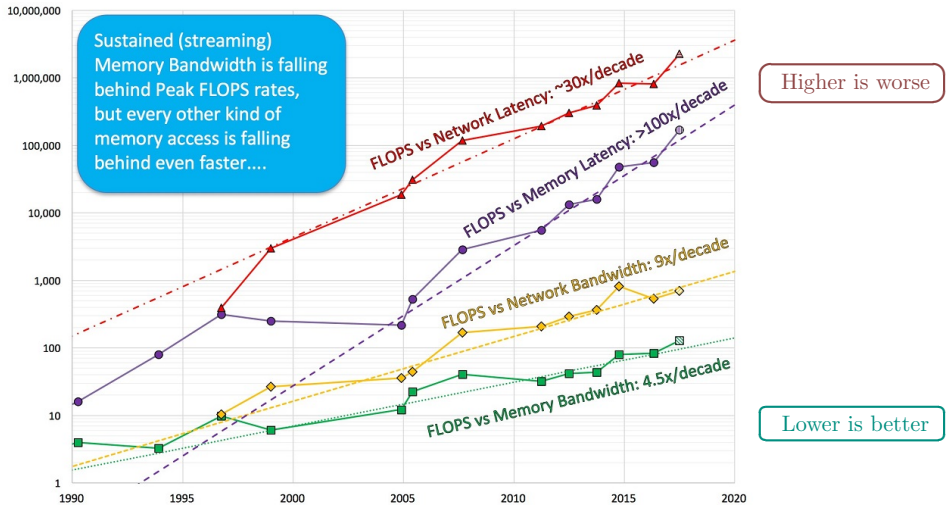



Figure 1: Trends of FLOPS vs memory/network access on HPC servers, SC'16, John McCalpin

During the last several decades, some trends have been at the forefront of the HPC systems evolution. First, the *memory wall* is figuring out the processor/memory performance gap that has grown steadily over the years. If memory latency and bandwidth become insufficient to provide processing elements with enough instructions and data to continue computation, they are stalled waiting on the data to be delivered. The trend of placing more and more cores on chip exacerbates this situation, since each processing units enjoys a relatively narrower channel to shared memory resources (e.g. memory, network). Fig. 1 illustrates how the number of FLOPS is increasing per byte received by memory access or through the network, these curves are disappointing for HPC users. In addition, the relative *latencies* of memory access and network are also raising compared to FLOPS. As a consequence, there are more and more memory-bound applications, and there are many applications that achieve less than 5% of computational peak performance, whereas it was commonplace back into the year 2000 to reach more than 50% of computational peak. This implies that application designers and developers should revise their algorithms to reduce memory moves, to increase *computational intensity* (FLOPS performed per byte), and possibly to establish *communication avoiding* methods. Such objectives are quite challenging, it is a research topic. It calls into question the current *programming models*, libraries and runtime systems used as well. We need much less synchronizations within applications, to introduce more overlapping opportunities to hide latencies, to exhibit finely dependencies between processing activities to improve the load balance because some computation kernels can stall occasionally. Equally, a great need for performance modeling tools is experienced, accurate description of byte moves is critical, while counting floating point operations becomes less important than it used to be.

A second major trend is the *power wall* issue. Since 2004, the increase of clock frequency in CPU has stalled. The power wall refers to a hard limit on frequency due to the handling of power dissipation of CPU heat with inexpensive techniques. To go beyond this physical limit, chip manufacturers have focused on bundling several processing units together into one chip (multicore approach), possibly with a lower clock frequency. Another solution is to build special-purpose processors such as general-purpose graphics processing units (GPGPU) or accelerators. These two solutions improve performance at the cost of increased programming and optimization complexities. On top of that, the costs of moving the data in and out of a central processor are becoming prohibitive. Then, developers are invited to write algorithms that save power, and to move forward on data centric programming to better exploit recent and future machines. To put it another way, performance optimization has

[1] terascale refers to methods for using supercomputers capable of performing at least 1 TFLOPS, i.e. $10^{12}$ FLOPS.

thus shifted from computing to data access. This shift of emphasis towards memory optimization also demands significant system support, from tools to compiler technologies, and from modeling to new programming paradigms. Programming models must evolve to encompass all aspects of the data management and computation; it requires upcoming programming abstractions where ideally compute, data and communication will play an equal role.

The constantly evolving HPC technologies is leading many actors of the community to periodically rethink how applications and software are written to take advantage of the last available parallel systems. Many contributions shown in this document typically go in this direction of adapting algorithms and numerical schemes to minimize overheads on the most up-to-date supercomputers.

### 2.1.3 Exascale target

The Terascale to Petascale transition has been quite smooth. For many applications this passage was incremental, the same global framework and tools remained valid. One had to remove scalability bottlenecks as the number of distributed processors and the partitioning of data increased. There was however no disruptive change as was the case of the transition from the vector processing era to distributed memory processing (around 1992). The path to Petascale included the introduction of small scale threading (*e.g.*, using OpenMP with a few cores), exposing vectorizable code to compilers, and going to GPU accelerators in some cases. But these features did not force a complete redesign for most codes. Instead, application developers could *incrementally* refactor the most important computational kernels to run well and leave much of less computation intensive code untouched.

After passing through the Petascale era[2], the upcoming exascale target ($10^{18}$ FLOPS) means big changes in supercomputers' architecture. These shifts are currently undergoing, mainly increasing the levels of parallelism to millions of processing units on a single supercomputer, large growth of on-node concurrency, increased penalty for having any sequential execution regions in the code. China, USA, Japan and Europe are investing billions of dollars each for demonstrating their ability to setup such exaflop machines around 2022. Exascale forces radical changes in how hardware is designed (power consumption is a major issue), in how we update the application codes or design new ones, and in how we glue application codes to hardware together (compilers, I/O, middleware, and related software tools). There are indications that we need to introduce new control layers and system software support (*e.g.* to support asynchronous tasking) and change significant parts of existing HPC application. Understanding the advantages to be gained by going to exascale, based on the past experiences is of utmost importance, because the costs associated to exascale computing are so large. Among all costs to design and build new processors, network and software stack, there is the question of power consumption for exascale machine. The Cray C90 used 0.5 MW of power in 1991 (16 GFLOPS), while by 2003 the ASCI/Q used 3 MW while being 2,000 times faster (30 TFLOPS), increasing the performance per watt 300 fold. However, first exascale systems are expected in 2022 to lie in the range of 20 MW up to 30 MW per machine for one EFLOPS. The improvement in term of raw performance per watt compared to ASCI/Q would be 4000 fold, which is a good point, nevertheless power feeding systems should be sized carefully. The power consumption is already so massive that some operators of large machines in the US are required to notify the electricity provider before they execute certain applications. A segmentation fault in a large application can cause a surge in power consumption and power-outages.

Already now, we are facing several challenges in making exascale computing a practical reality. These challenges arise both in the hardware realm and in the software, and will call for deep changes in the ways we build and use high-performance computers [101]. These are sufficiently complex, and interrelated, that a new methodological approach is called for in how the various research disciplines – from computer engineering to applied mathematics and computer science, and ultimately to the applications – interact as they pursue their own research agendas. One current trend is to develop and deploy a "co-design" methodology, in which the designs of hardware, numerical schemes, algorithms, programming models, and software tools are carried out in a tightly coupled and iterative fashion. To efficiently use HPC systems, it is not a wishful thinking to really invest time for getting closer to such co-design approach and to foster strong collaborations with nearby disciplines. It is common to see applications using less than 2% of peak performance of modern supercomputers (see HPCG benchmark results[3]), fruitful intertwined research and developments can really help to improve this status and to target the best usage of supercomputers. Much of my contributions described in this document typically goes in this direction, developing strong interdisciplinary works spanning mathematical modeling, high-performance computing, optimization and physics.

[2] one PFLOPS means $10^{15}$ FLOPS.
[3] http://www.hpcg-benchmark.org

## 2.2 Physics and numerical settings of Gysela

### 2.2.1 Physics

#### Main objectives

The tokamak is a magnetic confinement device being developed to contain the hot plasma needed for producing controlled thermonuclear fusion power. It is the leading candidate for a practical fusion reactor on earth. Magnetic fields are used for confinement because no solid material is able to accept the high temperature and fluxes of the hot plasma. The world's largest tokamak project is ITER[4] (International Thermonuclear Experimental Reactor) currently being constructed in Saint-Paul-lez-Durance, in southern France. First plasma is scheduled in 2025, it is expected to produce an output power of 500 megawatts. ITER will bridge the gap between today's smaller-scale experimental fusion devices and the demonstration fusion power plants of the future. Physicists will be able to study plasma under conditions similar to those expected in a future power plant and test technologies such as heating, control, diagnostics, cryogenics and remote maintenance.

Understanding and control of turbulent transport in thermonuclear plasma's in magnetic confinement devices is a major goal. This aspect of first principle physics plays a key role in achieving the level of performance expected in fusion reactors. In the ITER design, the latter was estimated by extrapolating an empirical law, which obviously is not enough. The simulation and understanding of the turbulent transport in fusion plasma within tokamak remains therefore an ambitious endeavor. Indeed, we do not have yet achieved a deuterium-tritium plasma in which the reaction is sustained through internal heating, which is an ITER goal.

The fusion energy community has been engaged in high-performance computing (HPC) for a long time. Among classical applications running on large facilities, the gyrokinetic simulations are one of the time hungriest (thousands up to millions of CPU-hours each simulation) and we then need large amounts of computational time that are typically provided by advanced computational facilities [132]. Computer simulation is and will remain a key tool for investigating several aspects of fusion energy technology to better learn about burning plasma experiments. Some of the key issues to address realistic simulations are: efficient and robust numerical schemes, realistic physics sub-models, accurate geometric description, good parallelization algorithms to use large supercomputers.

To better explain the physics inside tokamaks, a set of turbulent transport dynamics can be investigated with the so-called gyrokinetic global codes. In order to provide reliable physical results, the used schemes should be adapted to lower the noise induced by the numerical methods. Gysela is a global nonlinear electrostatic code which solves the gyrokinetic equations in a five dimension phase space with a semi-Lagrangian method. With the older versions of Gysela, one can model mainly the so-called *Ion Temperature Gradient* instability for one ion species with adiabatic electrons using flux-driven heat sources. Newer versions of Gysela include more physics with kinetic electrons or multi-species capabilities, but these novelties do not invalidate the various studies with the previous reduced setting presented in this document. Also, impurity sources in tokamak plasmas can have a deleterious impact on plasma performance, by diluting the fusion fuel and because they lead to radiative energy losses. A development has been engaged to integrate impurities in Gysela since 2012, the full ion and impurity species (two distribution functions) are evolved, coupled to quasi-neutrality with adiabatic electrons.

One important aim of gyrokinetic theory, and applications based on it, is to predict turbulent transport in fusion plasma. Some difficulties exist that are highlighted hereafter, some of them are quite common in the field of numerical HPC simulation. Hence validation and proper comparison between numerical results and experimental outputs originating from actual tokamak experiments are really an issue. This can be done mainly in two ways. The first one is based on a comparison of calculated and measured turbulent fluxes (or transport coefficients). The second approach consists in confronting other statistically averaged quantities such as the turbulence intensity, spectra or bi-coherence to experimental data, whenever available. However, if any mismatch exists between observed quantity and numerical output, there are many places (physics model, numerical scheme, code bug) where the error can be located. Another important physics ingredient is to properly model a fusion plasma defined as an open system. A tokamak plasma is essentially composed of an autonomous system with sources and sinks and complex boundary conditions that include some uncertainties and multiscale/multiphysics phenomena. A very fine balance sheet has to be established for conserving numerically invariant quantities. Checking the codes and improve robustness of numerical schemes is a real challenge. A trade-off should be made to both incorporate submodels with realistic physics and have ways to verify the numerical results. Also, the background geometry of the magnetic confinement configuration is not trivial and introduces a very strong anisotropy. Large gradients exist along some

[4] `https://www.iter.org`

directions that should be properly represented, magnetic topology and boundary conditions are serious constraints. In addition, multiscale dynamics develop in space and time. Thus, this setting is quite demanding in terms of robust and inexpensive numerical schemes, but also efficient parallel algorithms to operate large machines. The numerous issues and unknowns we encounter for tokamak modeling imply that research activities are really tightly coupled and need pragmatic solutions in Physics, in Applied Mathematics and in Computational Science.

### 2.2.2 Gyrokinetic setting

In the GYSELA code, one is interested in turbulence modeling with a *global code* that considers a large part of the plasma within the tokamak core. The *flux-driven* approach is considered with imposed localized source and sinks which induces the property that the distribution function and the profiles are left to freely evolve in time with the least possible artificial constraints (to avoid wrong assumptions). An important hypothesis is also that we consider no separation between equilibrium and fluctuations, *i.e.* a *full-f* code that differs from the so-called *delta-f* codes that assume a fake equilibrium underneath. Within this setting, complex self-organization turbulence can take place within the simulations. Our gyrokinetic model considers as main unknown a distribution function $f$ that classically represents the density of ions at a given phase space position. This function depends on time and on five other dimensions.

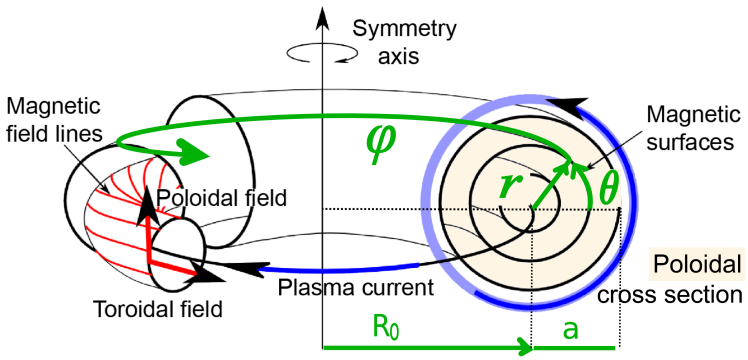



Figure 2: A tokamak magnetic configuration using a toroidal coordinate system $(r, \theta, \varphi)$. The geometry of the torus can be described by its minor radius $a$ and major radius $R_0$ (at the magnetic axis $r = 0$)

The coordinate system we consider is as follows (see Fig. 2 for an illustration). First, 3 dimensions in space $\mathbf{x}_G = (r, \theta, \varphi)$ with $r$ and $\theta$ the polar coordinates in the poloidal cross-section of the torus, while $\varphi$ refers to the toroidal angle. Second, velocity space has two dimensions: $v_\parallel$ being the velocity along the magnetic field lines and $\mu$ the magnetic moment corresponding to the action variable associated with the gyrophase. In a single species model, we are considering a collection of identical particles having charge $q \neq 0$ and mass $m > 0$, immersed in a static magnetic field $\mathbf{B}(\mathbf{x})$ The magnetic moment $\mu = mv_\perp^2/(2B)$ is an adiabatic invariant with $v_\perp$ the velocity in the plane orthogonal to the magnetic field. The computational domain is defined on $r \in [r_{\min}, r_{\max}], \theta \in [0, 2\pi], \varphi \in [0, 2\pi], v_\parallel \in [v_{\min}, v_{\max}], \mu \in [\mu_{\min}, \mu_{\max}]$. Let us consider the gyro-center coordinate system $(\mathbf{x}_G, v_\parallel, \mu)$ , then the non-linear time evolution of the 5D guiding-center distribution function $f(\mathbf{x}_G, v_\parallel, \mu, t)$ is governed by the so-called gyrokinetic equation which reads [139, 142] in its conservative form:

$$B_{||}^* \frac{\partial f}{\partial t} + \boldsymbol{\nabla} \cdot \left( B_{||}^* \frac{\mathrm{d}\mathbf{x}_G}{\mathrm{d}t} f \right) + \frac{\partial}{\partial v_\parallel} \left( B_{||}^* \frac{\mathrm{d}v_\parallel}{\mathrm{d}t} f \right) = B_{||}^* \left( \mathcal{D}_r(f) + \mathcal{K}_r(f) + \mathcal{C}(f) + \mathcal{S}(f) \right) \tag{2.1}$$

where $\mathcal{D}_r$ and $\mathcal{K}_r$ are respectively a diffusion term and a Krook operator [166] applied on a radial buffer region, $\mathcal{C}$ corresponds to a collision operator (see [64] for more details) and $\mathcal{S}$ refers to source terms (detailed in [69]). Hence, a heat source is mandatory in view of exploring the long time, typically on energy confinement times, behavior of turbulence and transport. The scalar $B_{||}^*$ corresponds to the volume element in guiding-center velocity space. The expressions of the gyro-center coordinates evolution $\mathrm{d}\mathbf{x}_G/\,\mathrm{d}t$ and $\mathrm{d}v_\parallel/\,\mathrm{d}t$ are not given here (details provided at p.57). The main information is that they depend on the 3D electrostatic potential $\phi(\mathbf{x}_G)$ and its derivatives. In this Vlasov/Boltzmann gyrokinetic equation, $\mu$ acts as a parameter because it is an adiabatic motion invariant. Let us denote by $N_\mu$ the number of $\mu$ values, we have $N_\mu$ independent equations of form (2.1) to solve at each time step. The function $f$ is periodic along $\theta$ and $\varphi$. Vanishing perturbations are imposed at the boundaries in the non-periodic directions $r$ and $v_\parallel$.

GYSELA is a global nonlinear electrostatic code which solves the gyrokinetic equations with a semi-Lagrangian method [10, 139]. We combine this with a second order in time Strang splitting method. Detailed explanations about the way we solve the Vlasov/Boltzmann Eq. (2.1) are presented in [3, 35, 139], some information are also shown in Section 3.1.

We now sketch the parallel domain decomposition used by this solver in the most standard configuration. Large data structures are used in GYSELA, the main ones are: the 5D data $f$, and the 3D data representing the electric potential $\phi$. Let $\mathrm{N}_r$, $\mathrm{N}_\theta$, $\mathrm{N}_\varphi$, $\mathrm{N}_{v_\parallel}$ be respectively the number of points in each dimension $r$, $\theta$, $\varphi$, $v_\parallel$. In the Vlasov/Boltzmann solver, we grant the responsibility of each value of $\mu$ to a given set of MPI processes [35] (a MPI communicator). We fix that there are always $\mathrm{N}_\mu$ sets, such as only one $\mu$ value is assigned to each communicator. Within each set, a 2D domain decomposition allows us to assign to each MPI process a subdomain in $(r, \theta)$ dimensions. Thus, a MPI process is then responsible for the storage of the subdomain defined by $f(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *, \mu = \mu_{value})$. The parallel decomposition is initially set up knowing local values $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$. They are derived from a classical block decomposition of the $r$ domain into $\mathrm{p_r}$ pieces, and of the $\theta$ domain into $\mathrm{p}_\theta$ subdomains. The numbers of MPI processes used during one run is equal to $\mathrm{p_r} \times \mathrm{p}_\theta \times \mathrm{N}_\mu$. The OpenMP paradigm is used in addition to MPI (#T threads in each MPI process) to bring another level of fine-grained parallelism.

**Quasi-Neutrality equation**

The quasi-neutrality equation and parallel Ampere's law close the self-consistent gyrokinetic Vlasov-Maxwell system. However, in an electrostatic code, the Maxwell field solver reduces to the numerical resolution of a Poisson-like equation (theoretical foundations are presented in [142]). In tokamak configurations, the plasma quasi-neutrality (denoted QN) approximation is currently assumed [139, 142]. Besides, in GYSELA code, electrons are assumed adiabatic (in most of the cases), *i.e.* electron inertia is ignored. Hence, the QN equation reads in dimensionless variables

$$-\frac{1}{n_0(r)}\nabla_\perp \cdot \left[\frac{n_0(r)}{B_0}\nabla_\perp \phi(r,\theta,\varphi)\right] + \frac{1}{T_e(r)}\left[\phi(r,\theta,\varphi) - \langle\, \phi \,\rangle_{\mathrm{FS}}(r)\right] = \tilde{\rho}(r,\theta,\varphi) \tag{2.2}$$

where $\tilde{\rho}$ is defined by

$$\tilde{\rho}(r,\theta,\varphi) = \frac{1}{n_0(r)} \int\int \mathcal{J}_{\mathbf{v}}\, J_0\, (f - f_{eq})(r,\theta,\varphi,v_\parallel,\mu)\, dv_\parallel\, d\mu\ . \tag{2.3}$$

with $f_{eq}$ representing local ion Maxwellian equilibrium. The perpendicular operator $\nabla_\perp$ is defined as $\nabla_\perp = (\partial_r, \partial_\theta/r)$. The radial profiles $n_0(r)$ and $T_e(r)$ correspond respectively to the equilibrium density and the electron temperature. $B(r,\theta)$ represents the magnetic field with $B_0$ being its value at the magnetic axis. $\mathcal{J}_{\mathbf{v}} = 2\pi\, B^*_{||}(r,\theta,v_{||})/m$ is the jacobian in velocity space. $J_0$ which denotes the Bessel function of first order is an approximation of the gyro-average operation[5]. $\langle\cdot\rangle_{\mathrm{FS}}$ denotes the flux surface average defined as $\langle\cdot\rangle_{\mathrm{FS}} = \int \cdot\, \mathcal{J}_{\mathrm{x}}\, \mathrm{d}\theta\, \mathrm{d}\varphi / \int \mathcal{J}_{\mathrm{x}}\, \mathrm{d}\theta\, \mathrm{d}\varphi$ with $\mathcal{J}_{\mathrm{x}}$ the jacobian in space of the system. The presence of this non-local term $\langle\, \phi \,\rangle_{\mathrm{FS}}(r)$ couples $(\theta, \varphi)$ dimensions and makes the parallelization yet more complex. We employ a solution based on FFT to overcome this problem; this method assumes polar coordinates and does not fit all geometries [163]. The QN solver includes two parts. First, the function $\tilde{\rho}$ is derived taking as input function $f$ that comes from the Vlasov/Boltzmann solver. In Eq. (2.3) specific methods are used to evaluate the gyroaverage operator $J_0$ on $(f - f_{eq})$ [91]. Second, I introduced new approaches to derive 3D electric potential $\phi$ using parallel algorithms to solve QN equation [93, 29]. Furthermore, the extension of these methods to toroidal setting instead of cylindrical one was described in [7, 3]. These studies will be summarized in Section 2.4.

In the following, we will refer to several data as *3D field data*. They are produced and distributed over the parallel machine shortly after the QN solver. These field data sets, namely: $\phi$, $\partial_r J_0\, \phi$, $\partial_\theta J_0\, \phi$, $\partial_\varphi J_0\, \phi$, are distributed on processes in a way that is caused by the parallel domain decomposition fixed for the Vlasov/Boltzmann solver. Indeed, they are inputs for the Eq. (2.1), and they play a major role in the terms $\mathrm{d}\mathbf{x}_G/\,\mathrm{d}t$ and $\mathrm{d}v_\parallel/\,\mathrm{d}t$. In some early versions of GYSELA (older than 2011) the whole 3D field data were known redundantly on each MPI process, whereas on the recent versions the 3D field data are well distributed with a specific parallel domain decomposition that will be detailed in the following.

### 2.2.3 Semi-Lagrangian and time integration methods

Two types of methods are most commonly used to solve the Vlasov-Poisson system. On the one hand, Lagrangian (or Particle-In-Cell) methods [107] discretize the distribution function into a finite number

[5] in Fourier space $J_0$ operator depends on two variables $k_\perp$ and $\mu$, with $k_\perp$ being the transverse component of the wave vector, see Section 2.4.2 for details.

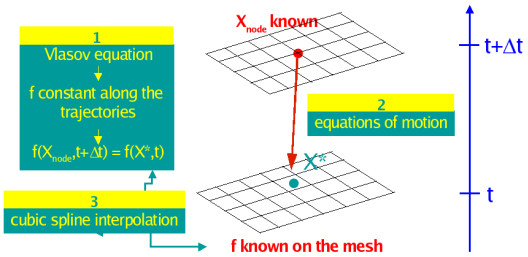



Figure 3: Basic algorithm for the Semi-Lagrangian method (Fig. from [139])

of macro-particles. The evolution of these macro-particles then follows the equations of motion derived from Vlasov[6], while a grid is necessary only in real space in order to solve the Poisson equation. Lagrangian methods allow for efficient computations with a numerical cost, parameterized by the number of particles, that can be tuned depending on the expected accuracy. The main drawback of such methods is the numerical noise due to the sampling of the distribution function, which requires advanced noise reduction methods (see for instance [156] for state-of-the-art noise reduction techniques in gyrokinetic simulations). On the other hand, Eulerian methods solve the Vlasov equation on a fixed grid in phase-space using finite differences, finite volumes or spectral methods to discretize the operators in the Vlasov equation. The key issues for Eulerian methods is that the discretization of the operators leads to a CFL[7] condition restricts the time step.

The semi-Lagrangian method is a mix between the Lagrangian and Eulerian methods, which tries to eliminate the main drawbacks of each method. This method was first developed for meteorological studies (see [183] for a review) and was more recently adapted to plasma simulations [182]. In order to avoid the statistical noise observed in Lagrangian codes, a fixed "Eulerian" grid in phase-space is used. On the other hand, to take advantage of the conservation of the distribution function along trajectories by the Vlasov equation, the characteristics of the equation are used to compute the time evolution of the distribution function. The basic algorithm for the backward semi-Lagrangian method is described in Fig. 3 [182]. For every grid point at a given time step, the characteristic curves are integrated backward to find the value of the distribution function at the foot of the characteristic. As this point $X^*$ is hardly ever on the grid, an interpolation must be performed to compute the value of the distribution function. It has been shown [136, 129, 105] that cubic spline interpolation provides a good compromise between accuracy (low diffusivity) and numerical cost. GYSELA combines semi-Lagrangian scheme and cubic spline interpolation as interpolation operator in the usual setting. The Vlasov equation is solved using the Strang splitting [186] combined with semi-Lagrangian scheme, insuring second order accuracy of the numerical scheme. Let $\hat{r\theta}$ denotes the shift operator associated to 2D advection equation solving in $(r, \theta)$ directions during one time step $\Delta t$. Similarly, $\hat{\varphi}$ and $\hat{v_\parallel}$ denote the shift operators respectively in $\varphi$ and $v_\parallel$ directions. Then, the splitting of Vlasov solver can be summarized by $(\hat{v_\parallel}/2, \hat{\varphi}/2, \hat{r\theta}, \hat{\varphi}/2, \hat{v_\parallel}/2)$.

The time integration scheme currently used is a predictor-corrector of order 2 in time. Let us denote time as $t_n = n\Delta t$. The *prediction step* advances the distribution function $f^n = f(t_n)$ to $f^{n+1/2}$ using electric potential $\phi^n$ (at time $t_n$). Knowing $f^{n+1/2}$, the corresponding potential $\phi^{n+1/2}$ is evaluated. Finally, we use both $f^n$ and $\phi^{n+1/2}$ to compute $f^{n+1}$. Thus, at each time step, we solve twice the set of Vlasov/Poisson equations. In 2010, I replaced the initial Leapfrog [139] integration scheme by this predictor-corrector scheme. With the Leapfrog approach (order 2 in time also), numerical instabilities may show up during simulations (sawtooth-like oscillation can be observed on mass and total energy diagnostics). Furthermore, the computational cost is similar for both methods.

### 2.2.4 Main parts of the code

The application can be seen as a collection of physical submodels and a set of solvers that are combined to model and mimic a set of physics phenomena taking place inside a plasma of tokamak. Here is a list of the major components of the GYSELA code :

- **Vlasov/Boltzmann solver** - Vlasov/Boltzmann equation is a transport equation posed in the phase space (see Eq. (2.1)). The solver uses a semi-Lagrangian approach and a Strang splitting as main ingredients. It is mixed with some of the components (related to Right-Hand Side computations) that follows, in that sense it should be called *Boltzmann solver*. Its role is to

[6] we will discuss here mainly the method for solving the left-hand side of Eq. (2.1) with no right-hand side; additional methods have to be considered to treat the right-hand side.

[7] the Courant-Friedrichs-Lewy (CFL) condition is a prerequisite for convergence while solving PDE; the time step must be less than a threshold in many explicit time-marching schemes, otherwise the simulation behaves incorrectly.

move the distribution function $f$ one time step forward. Within the Vlasov/Boltzmann solver, the following steps are embedded :

- **Advections** - At left-hand side, directional advections are solved (1D and 2D). Computation of the feet of characteristics and interpolation are the main components.
- **Sources** - Source terms are used to model the external heating of the plasma. These terms are injected as a right-hand side of Eq. (2.1).
- **Collisions** - Core plasmas are widely believed to be collisionless, but evidence from experimental side and from modeling have contributed to strongly alter this idea. Collisions are included through a reduced Fokker-Planck collision operator. The collisions appear in the right-hand side of Eq. (2.1).
- **Diffusions** - In order to model radial boundary conditions, diffusion operators are used on buffer regions (mainly close to internal/external radial boundaries). Buffer regions represent a very small radial fraction of the minor radius.

- **Field solver**
  - **Poisson solver** - The QN solver relies on an efficient 2D Poisson solver. It gives the 3D electric potential $\phi$ generated by the distribution of particles at a time step taking as input the distribution function $f$. Immediately following the Poisson solver, derivatives of $\phi$ are produced.
  - **Derivatives computation** - The derivatives of gyroaveraged electric potential (generated by the Field solver) are used to establish the particle trajectories inside the Vlasov solver. The derivatives of $J_0\,\phi$ are computed along the $r$, $\theta$ and $\varphi$ directions.

- **Diagnostics** - Physically relevant quantities are computed at a given time frequency in the code. This part, named *Diagnostics*, derives and outputs some files taking as input distribution function and electric potential.

The Vlasov/Boltzmann solver that includes the Sources/Collisions/Diffusions operators are the first computation intensive part of the code. For large physical cases, the solving of Eq. (2.1) represents usually more than 90% of the computational cost.

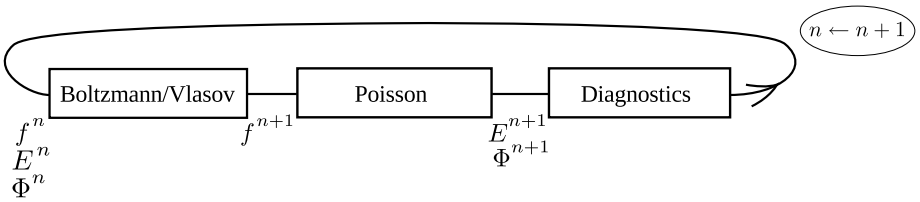



Figure 4: Numerical scheme for one time step of GYSELA

The execution of the application is decomposed in an initialization phase, iterations over time, and an exit phase. Fig. 4 illustrates the numerical scheme used during a time step: $f^n$ represents the distribution function, $\phi^n$ the electric potential and $E^n$ the electric field which corresponds to the derivatives of $\phi^n$. The Vlasov-solver step performs the evolution of $f_n$ over time and the Field-solver step computes $E^{n+1}$. Periodically, diagnostics are activated that export meaningful results extracted from $f^{n+1}$, $E^{n+1}$ and save the results in HDF5 files. This scheme is a simplified view, the time integration (predictor-corrector) scheme induces that the Vlasov and Field solvers are called two times per time step (one needs to compute intermediate time step $n+\frac{1}{2}$). To be more specific, predictor-corrector method is used within the Vlasov/Boltzmann step and implies that intermediate quantities $f^{n+1/2}, \phi^{n+1/2}, E^{n+1/2}$ are produced then discarded.

## 2.3 Vlasov solver

In this section, the main progress achieved over the last years concerning the parallel solutions, the performance and the scalability over large platforms of the Vlasov solver are described. First, we will go through *local splines* method that in the first place made it possible to scale well on several hundreds of processors with a good level of accuracy. Second, an alternative approach is presented that requires larger communication cost, due to a redistribution of the main 5D data over the processors, but permits to enlarge the time step size. Third, we highlight the gains achieved in the Vlasov solver thanks to simultenous multi-threading and fine-grained OpenMP parallelization.

### 2.3.1 Local cubic splines

**Context**

Initially, one of the first setup of GYSELA was using a cylindrical geometry[8] and a single $\mu$ value. The code was written in Fortran 90 and parallelized using MPI only. Mainly, there were simple parallelization strategies for the Vlasov and Poisson solvers. The scalability was poor for core counts larger than 64 [139].

If one takes the cubic splines over the whole $(r, \theta)$ plane, it does not provide a good locality property for the interpolations needed for advections along $(r, \theta)$. Indeed, all the values of the distribution function for a given poloidal cut $(r = *, \theta = *, \varphi = value, v_{\|} = value)$ are necessary to interpolate $f$ in this poloidal cut. The need comes from the sparse linear system to solve that involve all the $f$ values over the poloidal plane to derive the spline coefficients. To overcome this problem of strong dependencies, we proposed a solution that enables us to interpolate on quasi independent small 2D patches of the $(r, \theta)$ plane. Thus, we decompose global 2D cut into patches, each patch being devoted to a set of processes. One patch computes its own local cubic spline coefficients by solving reduced linear systems. Some adapted boundary conditions are imposed at the interface of the patches to obtain a $\mathcal{C}^1$ global solver which gives numerical results very close to the sequential solver that uses classical global splines. Moreover, thanks to a restrictive condition on the time step, the inter-process communications are only done between logically adjacent sets of processes, which enable us to improve communication speed. This approach has been successfully applied for two parallel applications: in the LOSS code [9,14] and the version adapted for GPU [30] (standard Vlasov-Poisson equations), and in GYSELA [35] (gyrokinetic setting).

**Description**

We introduce now an interpolation technique, based on a cubic spline method, in one dimension [9, 14, 35]. With a 2D tensor product of this spline method, interpolations on a 2D subdomain are achievable. In order to apply the $\hat{r\theta}$ operator of GYSELA, we used this 2D extension. Nevertheless, we explain here only the 1D case to simplify the explanations.

Let us consider a function $f$ which is defined on a global domain $[x_{\min}, x_{\text{Max}}] \subset \mathbb{R}$. This domain is decomposed into several subdomains called generically $[x_{m_p}, x_{M_p-1}]$; each subdomain will be devoted to one MPI process $p$. In the following, we will use the notation $x_i = x_{m_p} + ih$, where $h$ is the cell size: $h = (x_{M_p} - x_{m_p})/K$ and $K$ the number of cells on a subdomain ($K \in \mathbb{N}$). Let us now restrict the study of $f : x \mapsto f(x)$ on the interval $[x_{m_p}, x_{M_p}]$ (in order to decouple computations) with $M_p = m_p + K$. The projection $s$ of $f$ onto the cubic spline basis reads

$$f(x) \simeq s(x) = \textstyle\sum_{\nu=-1}^{K+1} \eta_\nu B_\nu(x),$$

where $B_\nu$ is the cubic B-spline. The interpolating spline $s$ is uniquely determined by $(K+1)$ interpolating conditions and the Hermite boundary conditions at both ends of the interval in order to obtain a $\mathcal{C}^1$ global approximation

$$f(x_i) = s(x_i), \ \ \forall i = m_p, ..., M_p, \quad f'(x_{m_p}) \simeq s'(x_{m_p}), \quad f'(x_{M_p}) \simeq s'(x_{M_p}). \tag{2.4}$$

The only cubic B-spline not vanishing at point $x_i$ are $B_{i\pm1}(x_i) = 1/6$ and $B_i(x_i) = 2/3$. Hence (2.4) yields

$$f(x_i) = 1/6\ \eta_{i-1} + 2/3\ \eta_i + 1/6\ \eta_{i+1}, \ \ i = m_p, \ldots, M_p. \tag{2.5}$$

On the other hand, we have $B'_{i\pm1}(x_i) = \pm 1/(2h)$, and $B'_i(x_i) = 0$. Thus the Hermite boundary conditions (2.4) become

$$f'(x_{m_p}) \simeq s'(x_{m_p}) = -\frac{1}{2h}\ \eta_{m_p-1} + \frac{1}{2h}\ \eta_{m_p+1}, \quad f'(x_{M_p}) \simeq s'(x_{M_p}) = -\frac{1}{2h}\ \eta_{M_p-1} + \frac{1}{2h}\ \eta_{M_p+1}.$$

Finally, $\eta = (\eta_{m_p-1}, ...\eta_{M_p+1})^T$ is the solution of the $(K+3) \times (K+3)$ system $A\eta = F$, where $F$ and $A$ are

$$F = \left[f'(x_{m_p}), f(x_{m_p}), ..., f(x_{M_p}), f'(x_{M_p})\right]^T, \quad A = \frac{1}{6}\begin{pmatrix} -3/h & 0 & 3/h & 0 & \cdots & 0 \\ 1 & 4 & 1 & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & -3/h & 0 & 3/h \end{pmatrix}.$$

[8] the magnetic-field curvature effects due to toroidal geometry is lacking in the cylindrical configuration.

A classical $LU$ algorithm is used to solve the tridiagonal linear system $A\eta = F$ (known as the Thomas algorithm).

*Approximation of the interface derivatives* In order to get accurate numerical simulations, one has to take care of the approximation of the derivatives at the interface of the subdomains. Various approximations were considered and evaluated. In order to recover the approximation of these interface derivatives obtained by a classical global cubic splines interpolation, a new formula can be derived to evaluate $f'(x_{m_p})$ and $f'(x_{M_p})$:

$$f'_{left}(x_i) = \sum_{j=1}^{j=10} \tilde{\gamma}_j^+ f_{i+j} \; ; \;\; f'_{right}(x_i) = \sum_{j=-10}^{j=-1} \tilde{\gamma}_j^- f_{i+j} \; ; \;\; f'(x_i) = f'_{left}(x_i) + f'_{right}(x_i).$$

We refer the reader to [9, 14] for the details obtaining this approximation and numerical values of coefficients $\gamma_j^-$ and $\gamma_j^+$.

*Properties.* Within the simulation we expect to interpolate on the interval $[x_{m_p-1}, x_{M_p}]$ instead of $[x_{m_p}, x_{M_p-1}]$ to avoid extra communication when interpolation is located a little bit out of the local domain. In order to extend the interpolation capability on one process, we compute an extra $\eta_{m_p-2}$ coefficient. We impose the property that $m_{p_i} = M_{p_j}$ for $p_i$ and $p_j$ two adjacent processes that share the grid point $x_{m_{p_i}}$. With these minor modifications, each process has the responsibility to modify the values of grid points in the interval $[x_{m_p}, x_{M_p-1}]$, and have the capability to interpolate onto the extended interval $[x_{m_p-1}, x_{M_p}]$. In the 2D splitting phase, where the local splines are used, it means that the displacement of one single grid point on the border of a subdomain must not exceed the elementary cell width. This constraint can be restrictive and constitutes the main drawback of the method, since it is not possible to consider big shift (in $r$ or $\theta$) during a single advection in the 2D splitting phase. The computation of the $\eta_{m_p-2}$ coefficient is deduced from (2.5) with $i = m_p-1$. The value $f(x_{m_p-1})$ is transmitted from a neighboring process.

*Communication pattern.* In a parallel implementation of the local spline method, communications are required between adjacent processes to build the right-hand side term $F$. On a local process, the known values are $f(x_i)_{i\in[m_p,M_p-1]}$. For processes located at the borders of the global domain ($x_{\min} = x_{m_p}$ or $x_{\text{Max}} = x_{M_p}$), boundary conditions (compact or periodic) are considered to retrieve the values of $f$ needed outside the domain. Hereafter, we enumerate the data lacking on the local process to get $F$ (excluding specific problems that arise at the global domain boundaries):

a) Values of $f(x_{m_p-1})$, $f'_{left}(x_{m_p})$ are received from a neighboring process.

b) Values of $f(x_{M_p})$, $f'_{right}(x_{M_p})$ are received from a neighboring process.

c) The quantities $f'_{right}(x_{m_p})$ and $f'_{left}(x_{M_p})$ are computed on the local process and send to processes that need them.

Concerning the third item, we choose practically a large enough $K$ to have only local calculations to compute $f'_{right}(x_{m_p})$ and $f'_{left}(x_{M_p})$. $K \geq K_{\min} = 32$, leads to a relatively small overhead and provides good numerical stability.
In the case of a 2D interpolation, the $F$ term is a matrix instead of a vector. The assembly of $F$ requires communications with the 8 neighboring processes. On one process and for a 2D patch of size $K_1 \times K_2$, the number of double precision real numbers to receive is $4\,(K_1 + K_2 + 4)$. This amount of communication could be compared to the interpolation cost of the $K_1 \times K_2$ points in a patch, which is in $\Theta(K_1\,K_2)$. For $K_1$ and $K_2$ greater to $K_{\min} = 32$, the ratio of communication cost over computation cost remains small enough on current machines.

*Limitation.* Numerical experiments with the local spline method for the 2D splitting on physical test cases have shown a bottleneck in Gysela. The shifts in direction $\theta$ are often too large and above the limit we fixed (the width of one cell). It was not feasible to keep this configuration, so we were compelled to remove completely the $\theta$ parallelization in the algorithm and just keep the parallelization along $r$ dimension. Another solution would have been to improve the interpolation capacity of each process to larger subdomains. But in such case, extra costly communications would be required.

**Benchmarks**

In addition to MPI, the OpenMP programming model can be combined to obtain a hybrid paradigm to exploit levels of parallelism at a finer grain, without heavy code manipulation. The hybrid approach is suitable for clusters of nodes where MPI provides communication capability across processes and OpenMP exploits loop level parallelism within a process. Each advection within the Vlasov solver has been parallelized with OpenMP, as well as the field solver. At that time (2007), the numerical experiments were performed on a cluster of IBM 16-core nodes located in Bordeaux, France. Each

| | Time | Effic. | Time | Effic. | Time | Effic. | Time | Effic. |
|---|---|---|---|---|---|---|---|---|
| **Nb. processes** | **1** ($nbt$=1) | | **8** ($nbt$=1) | | **64** ($nbt$=8) | | **128** ($nbt$=16) | |
| advections 1D ($\varphi$) | 334.7 | 100 | 40.87 | 102 | 5.38 | 99 | 2.66 | 98 |
| advections 1D ($v_\parallel$) | 305.6 | 100 | 40.22 | 95 | 5.04 | 95 | 2.52 | 95 |
| advection 2D ($r, \theta$) | 709.5 | 100 | 99.67 | 89 | 12.94 | 85 | 6.86 | 81 |
| Total advections | 1349.8 | 100 | 180.76 | 93 | 23.37 | 90 | 12.04 | 88 |
| Total field solver | 33.1 | 100 | 9.95 | 42 | 1.50 | 33 | 0.78 | 33 |

Table 1: Strong scaling - Efficiency and computation time in seconds for a single time step of a medium 4D test case $\mathrm{N_r} = 256, \mathrm{N}_\theta = 256, \mathrm{N}_\varphi = 128, \mathrm{N}_{\mathrm{v}\parallel} = 64, \mathrm{N}_\mu = 1$ ($nbt$ the number of threads within each MPI process, and [*Nb. procs*/$nbt$] the number of MPI processes)

node hosted a Power5 processor and offered 27GB of shared memory. In Table 1, we show performance for a 4D GYSELA case with $\mathrm{N}_\mu = 1$. The MPI parallelization on variable $\theta$ is not active; so in the given test case with $\mathrm{N_r} = 256$, the $(r, \theta)$ domain could be decomposed up to only $proc_r = 8$ subdomains. The maximum number of cores that we could use is then $proc_r\, N_\mu = 8$. The hybrid paradigm usage increases this maximum number to 128 (one could use up to 16 threads per node), thus improving our capability to use more computing resources.

In Table 1, the 1D advections are perfectly parallel and scalable, because no overhead in computation nor in communication is triggered. However, the 2D advections requires a communication step to transmit boundary coefficients and derivatives to use local splines. Furthermore, the 2D interpolation on patches induces a small computation overhead in comparison to a global spline sequential method. These two facts explain why the efficiency decays whenever $proc_r$ and number of processes increases from 1 to 8. The field solver, because of a too much simple parallelization has not a good speedup (this issue will be studied later). Nevertheless, computation time for this field solver remains small compared to others. The main advantages of the hybrid MPI+OpenMP approach is to allow one to use more cores for a given test case. These early results (2007) demonstrated the overall possibility for the GYSELA application to scale well up to 128 cores with only a single $\mu$ (*i.e.* a 4D test case, instead of multiple $\mu$ for a standard 5D test case). This would suggest that the scalability for a 5D setting should be nice up to a few thousands of cores.

### 2.3.2 Transpose algorithm

#### Context

The parallelization based on local splines [14, 35] does not support large displacements (several grid cells typically) in the $(r, \theta)$ plane during one single time step. Using a toroidal setting (classical since 2009 in GYSELA) instead of cylindrical one, drift velocities in the poloidal plane have quite high values along $(r, \theta)$ (some displacements of more than 10 cells during one advection). Then, to support large time steps and to shorten global execution time, a transposition of the distribution function inside each $\mu$ value is appropriate. Our evaluation shows that somewhat counter-intuitively, even with a transposition, this strategy shows good performance even with large number of cores [18]. Let us introduce the *transpose* algorithm and compare it to the local spline 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.

#### Algorithms description

**for** *time step* $n \geq 0$ **do**
  Field solver, Derivatives' computation, Diagnostics
  Vlasov solver
    1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    1D Advection in $\varphi$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    **Local splines 2D Advection** in $(r, \theta)$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    1D Advection in $\varphi$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$

**Algorithm 1:** using local splines for 2D advection

The local spline version of GYSELA [35, 139] used a MPI domain decomposition along dimensions $\mu$, $r$, $\theta$. The $\mu$ dimension is at the highest level of parallelism and each $\mu$-value constitutes a MPI communicator. Indeed, there is no advection along $\mu$ direction, thus Vlasov solving consists in solving

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

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

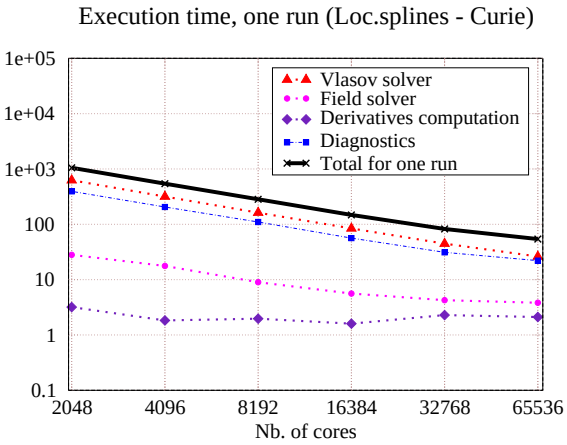



Figure 5: Strong scaling - Local spline

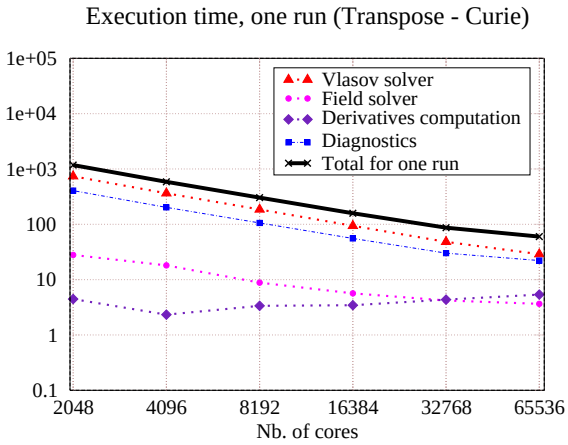



Figure 6: Strong scaling - Transpose

$N_\mu$ independant equations. Then, within each communicator dedicated to one $\mu$, a domain decomposition in $r$ and $\theta$ is used to distribute the distribution function among MPI processes. OpenMP is used to exploit the parallelism available in the inner loops. We use the following notation: `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 (typically needed for standard cubic spline interpolations). In Algorithm 1, one can see that a single domain decomposition is valid for all advections, nevertheless the advection along $(r,\theta)$ has to be handled in a special way. Local cubic splines have been designed in order to perform interpolation on 2D sub-domains during the 2D advection in $(r,\theta)$. Typically, only a ghost zone of 3 points is needed at each border of a local sub-domain in $r$ and $\theta$ with this approach. These local splines generate few communications between processes while preserving accuracy of the interpolation. However, it limits the displacement during one time step in $(r,\theta)$ to one grid cell at the maximum. This introduces a CFL-like condition that prevents use of large time steps [9].

An alternative to local splines is presented now where the MPI domain decomposition is switched between advections as shown in Algorithm 2. The 4D distribution function (for a given $\mu$ value) is transposed just before and after the 2D advection along $(r,\theta)$. Each process 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 $(r=*, \theta=*)$ for a given $(v_\parallel, \varphi, \mu)$ tuple. Hence, the CFL-like condition is removed. Furthermore, other advection schemes that need to consider the whole domain $(r=*, \theta=*)$ are now feasible. Communications of the transpose step have good locality properties, the message exchanges are done inside a $\mu$-communicator that groups together adjacent processes leading usually to improved network bandwidth. The numerical accuracy is close for the previous and the new solution.

Some values, as the $J_0\,\phi$-*derivatives*, are computed from $\phi$. 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 also requires to adapt the communication scheme to send the appropriate $J_0\,\phi$-*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 $J_0\,\phi$-*derivatives* on the domain decomposition $(r=*, \theta=*, \varphi=local, v_\parallel=local, \mu=local)$. The costs of these additional communications will be exhibited further below.

**Experimental evaluation**

Timing measurements of this subsection have been realized on the Curie machine at the French GENCI-TGCC-CEA computing center in 2013. 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, essentially because MPI_THREAD_MULTIPLE mode is not available or not benefical in our context on most of the platforms we use.

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. 5, 6, 7 and 8 show a strong scaling from 2k cores up to 65k cores (16 threads per MPI process). Let us notice that these curves are plotted with a logarithmic scale for abcissa. On Fig. 5, one can observe a good global scaling behavior for the elapsed 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, which is an excellent behavior. 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 others. However, field solver and derivatives computations have relatively low execution times (see Section 2.4).

Let us compare these execution times obtained by the local splines (Algo 1) shown in Fig. 5 with the transpose version (Algo 2) shown in Fig. 6. 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. 9 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 quite well on modern architectures (IBM BlueGene/Q architecture has also been checked [26]). 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. 7 and 8 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. Typically, we avoid the cases with 32k and 64k cores presented in Fig. 7 and 8. 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 users obtain every year on several supercomputers.

On Fig. 9, 10 and 11, 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 an hybridization between a MPI scatter and a set of MPI broadcasts 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 the main costly steps.

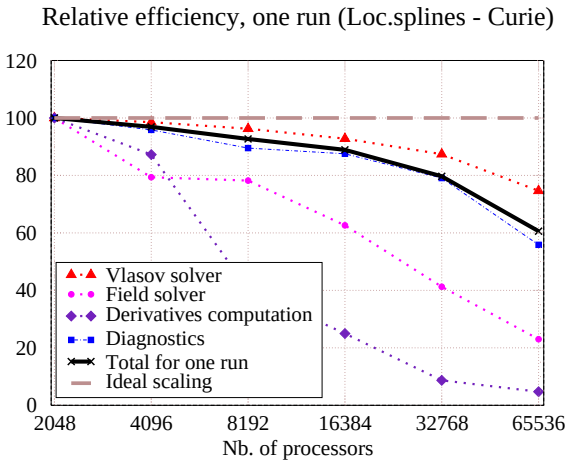



Figure 7: Strong scaling - Local spline

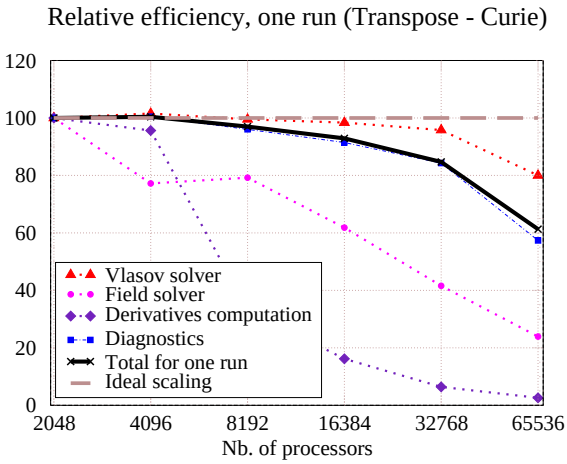



Figure 8: Strong scaling - Transpose

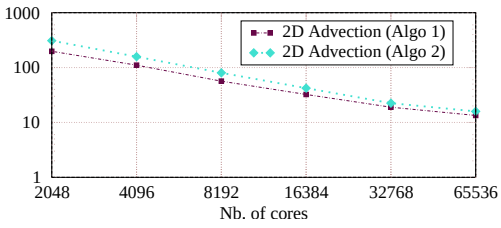



Figure 9: 2D Advection comparison - Timing

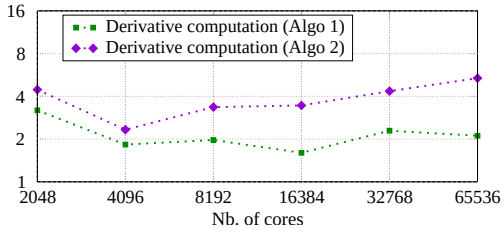



Figure 10: Derivatives comp. comparison - Timing

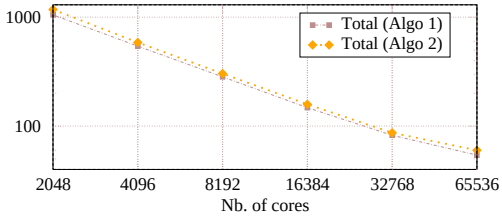



Figure 11: Total run-time comparison - Timing

The expense of this transpose method is an overhead per time step (Fig. 11) 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 shortens global execution time for typical runs. Production runs of Gysela now exclusively use the transpose algorithm.

### 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 classically 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 or a part of the 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. Examples exist in the literature where the transposition permits to employ profitable schemes with good scaling on parallel machines [21, 113, 114, 161, 174, 177, 184]. 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 the trade-off.

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 [120] 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 track of performance ratio of halo vs. transpose with the new hardware architectures. We are now considering a pipelined version of the transpose communication pattern in order to partially overlap communication costs with the 2D advection computations (unpublished work of postdoc Y. Asahi). 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 will help a lot.

### 2.3.3 Simultaneous multi-threading

#### Context

*Trends* 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 (arrival of this hardware in 2015 on supercomputers) 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. We now describe the adaptation of the code for balance load whenever using both SMT and good deployment strategy. It led us to a significant reduction that can be up to 38% of the execution times [18].

*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). For example, 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.

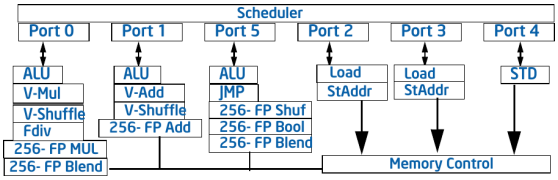



Figure 12: Sandy Bridge CPU Core Pipeline Functionality, extract from [154]

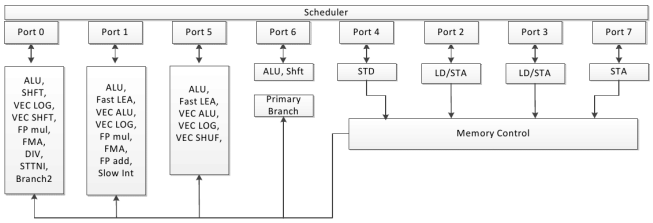



Figure 13: Haswell CPU Core Pipeline Functionality, extract from [154]

Fig. 13 presents the pipeline of the Haswell architecture compared to that of Sandy Bridge in Fig. 12. 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 $\mu$-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 $\mu$-operations from which the scheduler can choose. This is why simultaneous multi-threading[9] (SMT) is more and more important 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 experimental study, we use the Jureca machine from Jülich/Germany (2016). This supercomputer hosts 1872 compute nodes. Each node contains two INTEL Xeon E5-2680 v3 Haswell CPUs ($2 \times 12$ cores, 2.5 GHz).

*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 [141].

### 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 (this constraint has been removed during year 2016). Let us remark, that the application performance increases by avoiding very small power of two (*i.e.* 1, 2). Haswell node that we target is 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 2. Let us assume that processes inside each node is numbered with an index $n$ going from 0 to 2 without SMT, and 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 2: Time measurements for a strong scaling experiment with SMT activated or deactivated, and gains due to SMT. Minirun of 8 times steps.

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 by others (*e.g.* [155, 187]), 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`) thanks to a collaboration with BSC/Spain through the H2020 EoCoE project (2015-2018). 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) accumulated over the 2 threads on one core with SMT is always higher than the IPC obtained with one thread

[9] multiple hardware threads are handled by each core

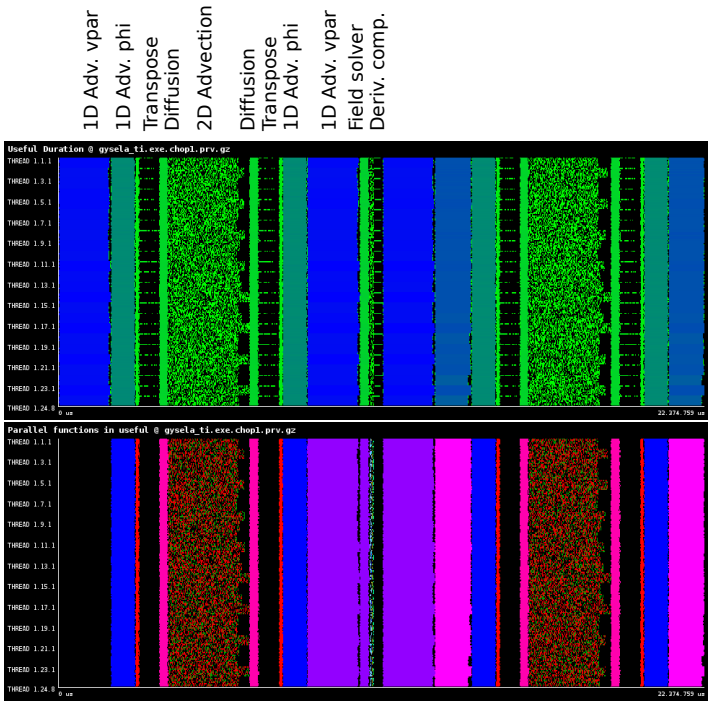



Figure 14: Snippet of a run with 2 threads per core (SMT), Top: Paraver useful duration plot, Bottom: Parallel functions plot

per core without SMT. For these kernels, the accumulated 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 slots are often unused (for example due to memory or network lantencies).

**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. 14 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:

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

b) 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.

c) The transpose steps show a lot of black regions (threads remaining idle). Fig. 15 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.

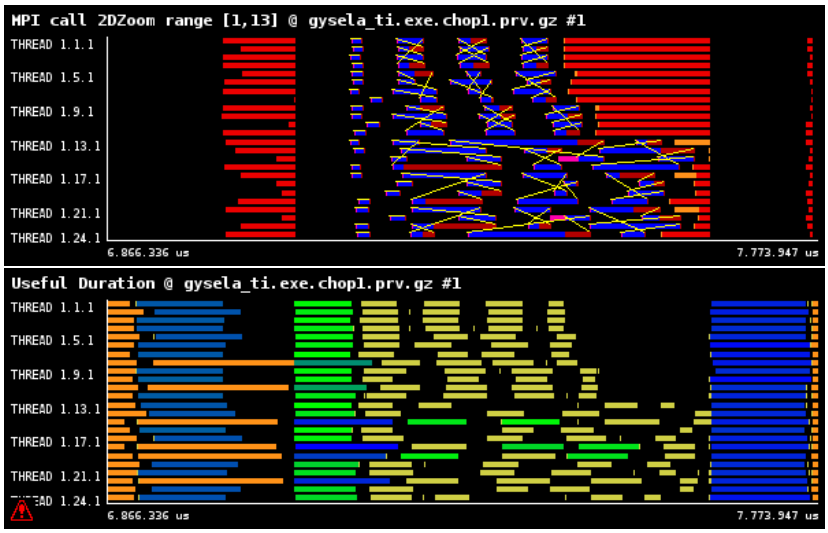



Figure 15: Zoom on the Transpose kernel (only MPI ranks are displayed), Top: Paraver MPI calls plot, Bottom: useful duration plot

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 3 and Fig. 16 exhibits associated measurements.

a) The 2D advection kernel is composed of OpenMP regions. 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 $\varphi, v_{\|}$ 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.

b) 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 dynamic clock frequency scaling. 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. 16 that, now, diffusion is sandwiched between the transpose step and the 2D advection, without global synchronization.

c) 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. 14, 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. 16, 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. 16 compared to Fig. 14. 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 3 also illustrates the achieved gain in term of elapsed time. If one compares to Table 2, the timings are reduced with one or

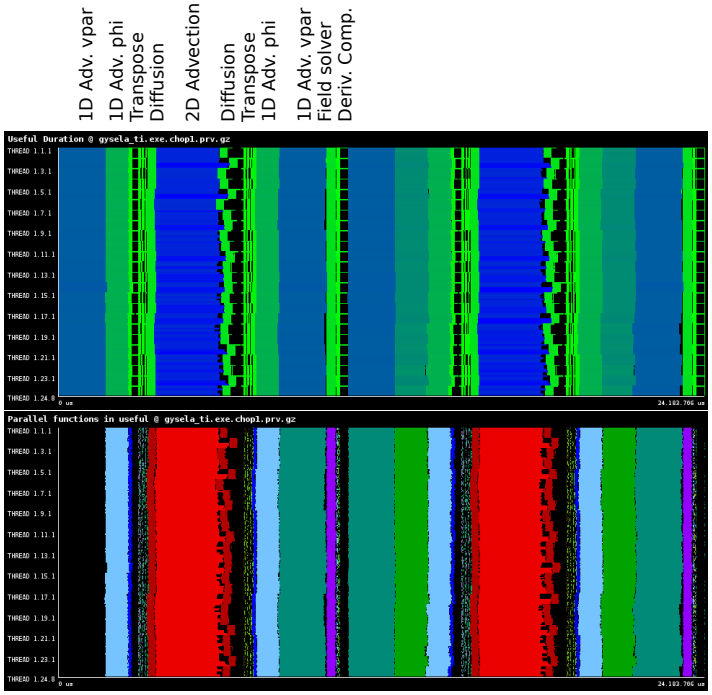



Figure 16: Snippet of a run with 2 threads per core (SMT), after optimizations are done, Top: Paraver useful duration plot, Bottom: Parallel functions plot

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 accumulate 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 3: Time measurements and gains achieved after optimizations that remove some synchronizations and some OpenMP overheads. Minirun of 8 times steps.

### 2.3.4 Conclusion

We have introduced the local spline method to be used with Semi-Lagrangian scheme for Vlasov equation. It employs patches decomposing the velocity and space domains, each patch being devoted to a process. Some adapted Hermite boundary conditions allow to reconstruct a good approximation of the global solution. Moreover, this kind of strategy allows to avoid a time consuming part (communication costs between processes) which is the transposition step. Several numerical results demonstrate the accuracy and the good scalability of the method. The local spline method shows good behavior for simulations of a high intensity beam in periodic or alternating gradient focusing fields (paraxial Vlasov equation on the 4D transverse phase space), but also on the GYSELA application in the simple 4D cylindrical setting. However, local splines show strong limitation whenever large displacements appear with realistic toroidal geometry.

In this latter situation, a good implementation of the transpose algorithm helps reducing this pseudo-CFL condition and represents a scalable and robust solution to handle different physics regimes (including 5D simulation with toroidal geometry). Global execution times are satisfactory with transpose: GYSELA achieves a good parallel computation scalability up to 64k cores combining several levels of parallelism and a hybrid OpenMP/MPI approach. Overheads related to extra communication costs remain reasonable for now. The expanse of network bandwidth use needs to be put in perspective of exascale challenges. Overlapping these communications will be addressed soon to alleviate the pressure over network throughput.

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 accumulated throughput of all available integer units and floating-point units, especially in the Vlasov solver. 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 and modern memory organization and management introduce 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 at a very fine grain of parallelism. We currently investigate task parallelism to go further in improving load balancing and in removing global synchronization in-between global steps.

## 2.4 Field solver

The Vlasov solver is the main hotspot of the application that represents a large fraction of total execution time. However, the field solver is also able to become a bottleneck on very large parallel machine if the computations are not well distributed or if the amount of communications grows too much within the field solver or in-between Vlasov and field solvers. As the Vlasov solver came close to an excellent parallel efficiency in 2011, the field solver started becoming a limiting factor. This Section describes a set of numerical schemes and parallel algorithms that were co-designed to achieve higher, sustainable levels of performance up to a large number of cores. Thus, we continued to push forward parallel scalability considering the persistent growth in the size of supercomputers.

### 2.4.1 Introduction

A parallel gyrokinetic code needs to finely couple a parallel Vlasov solver with a parallel field solver to be an efficient method. The role of a quasi-neutrality (QN) solver is to give the electric potential $\phi$ taking as an input the particles density, whereas the Vlasov solver moves the particle density forward in time. Moreover, the numerical resolution of the QN equation requires the solving of a 3 dimensional equation, involving domain decomposition and communications that may penalize an efficient parallel implementation [35]. One of the possible ways to treat numerically the quasi-neutrality equation consists in the use of Fast Fourier Transform, other choices are multigrid or use of a direct solver for a finite element method for example. Even if the FFT approach is not adapted to general geometries [163], in periodic direction they remains a fast, simple and accurate method for polar coordinates (and a circular plasma). First, one solves $N_\varphi$ independent $2D$ Laplacian type equations in the poloidal plane. Second, a simple $1D$ ordinary differential equation has to be solved on the average of the electric potential on the magnetic surface along the toroidal dimension $\varphi$.

I proposed an adapted communication scheme to reduce the communication cost between the Vlasov solver and the QN solver. As a consequence, a global parallelized gyrokinetic solver is then obtained, the performance of which is presented in the following. We will consider here cylindrical setting instead of toroidal setting introduced more recently and explained later on [3,7], this choice will allow us to match the background of the papers on which is based this Section, *i.e.* [26, 29, 91, 92, 93]. Mainly, the difference of toroidal vs cyclindrical is that with cylindrical setting: $\mathcal{J}_{\mathrm{x}}$ the jacobian in space of the system does not depend on $\theta$, and that: $\langle\cdot\rangle_{\mathrm{FS}}$ denoting the flux surface average degenerates into the average along $\theta, \varphi$ directions.

The rest of the Section is organized as follows: First, we present the QN equation together with the gyroaverage operator. Then, we describe various methods, and their impact on the GYSELA performance.

### 2.4.2 Using 2D FFT in the field solver and solution to avoid it

#### Quasi-Neutrality equation

In tokamak configurations, the plasma quasi-neutrality approximation is assumed in many cases [139, 142]. This leads to $n_i = n_e$ where $n_i$ (resp. $n_e$) is the ionic (resp. electronic) density. Under some assumptions, electron inertia can be ignored, which means that an adiabatic response of electrons are supposed. The QN solver includes two computation parts. First, the function $\tilde{\rho}$ is derived taking as input function $f$ through Eq. (2.3) (p.15). Specific methods, that will be described afterwards, are used to evaluate the gyroaverage operator $J_0$ on $(f - f_{eq})$ included in Eq. (2.3). Second, the 3D potential $\phi$ is found in computing discrete Fourier transforms of $\tilde{\rho}$, followed by solving of tridiagonal systems and inverse Fourier transforms in order to treat Eq. (2.2) (p.15).

**Gyroaverage operator**

Let us now detail the computation of $\tilde{\rho}$. Starting from the ionic guiding center distribution function $f = f(r, \theta, \varphi, v_{//}, \mu)$, we can obtain the ionic density on the particle coordinates thanks to a *gyroaverage* operator, one of the cornerstone of the gyrokinetic theory. The gyroaverage operator transforms the so-called guiding-center distribution into the actual particle distribution. It enables to take into account effects relative to the finite Larmor radius, which is the radius of gyration of the gyro-center (motion which is faster than the turbulence we are looking at). After some computations in Fourier space, this operator makes appear the Bessel operator and leads to (2.3) (p.15). In the sequel, we focus on the computation of $g = g(r, \theta, \varphi)$ satisfying

$$\bar{g}(r, \theta, \varphi) = J_0(k_\perp \sqrt{2\mu}) g(r, \theta, \varphi) \tag{2.6}$$

Please note that $J_0$ depends on $\mu$ value, this point will be important for several parallel algorithms. The numerical resolution of such a problem is based on a Padé approximation which is currently performed for the Bessel function $J_0$

$$J_0(k_\perp \sqrt{2\mu}) \approx \frac{1}{1 + \frac{(k_\perp \sqrt{2\mu})^2}{4}}. \tag{2.7}$$

This approximation is correct for small wavenumbers $k_\perp$ and keeps $J_0$ finite in the opposite limit $|k_\perp| \to +\infty$. Injecting the Padé approximation (2.7) in (2.6), a Fourier transform enables to use the equivalence between $(ik_\perp)$ and $\nabla_\perp$. Finally, we can obtain $g$ by solving the following implicit equation

$$\left[1 - \frac{\mu\, B}{2\, {\omega_c}^2\, m_i} \nabla_\perp^2\right] \bar{g}(r, \theta, \varphi) = g(r, \theta, \varphi).$$

**Numerical methods for gyroaveraging**

Let us consider a function $f$ which is defined on a global domain $[r_0, r_{Nr}] \subset \mathbb{R}$. In the following, we will use the notation $r_i = r_0 + i\, dr$, where $dr$ is the mesh size: $dr = (r_{Nr} - r_0)/(Nr + 1)$. Let us now restrict the study of $g : r \to g(r)$ on an interval $[r_0, r_{Nr}]$, $Nr \in \mathbb{N}$, where all $r_i$ are known. Our goal is to get the gyroaverage $\bar{g}$ from a known $g$ function. Let us do a Fourier transform in variable $\theta$. Each $k$ Fourier mode of $\bar{g}$ is the solution of the equation of the following equation (after some approximations [10, 139]):

$$\left[1 - \frac{\mu\, B_0}{2\omega_c^2\, m_i} \left(\frac{\partial^2}{\partial r^2} - \frac{k^2}{r^2}\right)\right] \bar{g}^{\,k}(r, \varphi) = g^{\,k}(r, \varphi)\ .$$

We assume that we have Neumann boundary conditions, and that ${g_1}^m = {g_2}^m$ and ${g_{Nr}}^m = {g_{Nr-1}}^m$. In the code, the hypothesis $B_0 = 2\omega_c^2\, m_i$ is done in a set of simplified geometries where $B$ is considered constant in the poloidal plane. It gives the following system to solve for each $\varphi$:

$$\begin{pmatrix} \alpha{+}\beta_2 & \alpha & 0 & & \\ \alpha & \beta_3 & \alpha & 0 & \\ & \ddots & \ddots & \ddots & \\ & 0 & \alpha & \beta_{Nr-2} & \alpha \\ & & 0 & \alpha & \alpha{+}\beta_{Nr-1} \end{pmatrix} \begin{pmatrix} \bar{g}_2^k(\varphi) \\ \bar{g}_3^k(\varphi) \\ \vdots \\ \bar{g}_{Nr-2}^k(\varphi) \\ \bar{g}_{Nr-1}^k(\varphi) \end{pmatrix} = \begin{pmatrix} g_2^k(\varphi) \\ g_3^k(\varphi) \\ \vdots \\ g_{Nr-2}^k(\varphi) \\ g_{Nr-1}^k(\varphi) \end{pmatrix} \tag{2.8}$$

with

$$\begin{cases} \zeta & = \frac{1}{\Delta r^2} \\ \alpha & = -\frac{\mu}{2}\, \zeta \\ \beta_j & = 1 + \frac{\mu}{2}\, (2\, \zeta + \frac{k^2}{r_j^2}) \end{cases} \tag{2.9}$$

The solving of system (2.8) allows one to apply the gyroaverage operator on function $g$. A $LU$ decomposition is done once for the tridiagonal matrix of system (2.8). The matrices $L$ and $U$ are used every time a gyroaveraging is needed with a computational complexity of $\Theta(N_r)$ for the solving procedure and $\Theta(N_r\, log(N_\theta))$ for the Fourier transforms.

**Solver based on 2D Fourier transform (initial method)**

The QN equation (2.2) can be solved in the Fourier space [139] along the two periodic directions ($\theta$, $\varphi$) taking as input the values of $\tilde{\rho}$. Let $\phi$ and $\tilde{\rho}$ be defined in terms of Fourier expansion as:

$$\begin{array}{ll} \phi(r, \theta, \varphi) = & \sum_u \sum_w \hat{\phi}^{u,w}(r) e^{i\, u\, \theta} e^{i\, w\, \varphi} \\ \tilde{\rho}(r, \theta, \varphi) = & \sum_u \sum_w \hat{\rho}^{u,w}(r) e^{i\, u\, \theta} e^{i\, w\, \varphi} \end{array} \tag{2.10}$$

In this wave number representation, the QN equation could be written as:

$$-\frac{\partial^2 \hat{\phi}^{u,w}(r)}{\partial r^2} - [\frac{1}{r} + \frac{1}{n_0(r)}\frac{\partial n_0(r)}{\partial r}]\frac{\partial \hat{\phi}^{u,w}(r)}{\partial r} + \left(\frac{u^2}{r^2} + \frac{(1-\delta_{u=0,w=0})}{Z_i\, T_e(r)}\right)\hat{\phi}^{u,w}(r) = \hat{\rho}^{u,w}(r)$$

Let $N_r$ be the number of radial points. We want to compute each $\hat{\phi}^{u,w}$ such as:

$$\begin{pmatrix} m_2 & o_2 & 0 & & \\ p_3 & m_3 & o_3 & 0 & \\ & \ddots & \ddots & \ddots & \\ & 0 & p_{Nr-2} & m_{Nr-2} & o_{Nr-2} \\ & & 0 & p_{Nr-1} & m_{Nr-1} \end{pmatrix} \begin{pmatrix} \hat{\phi}_2^{u,w}(r) \\ \hat{\phi}_3^{u,w}(r) \\ \vdots \\ \hat{\phi}_{Nr-2}^{u,w}(r) \\ \hat{\phi}_{Nr-1}^{u,w}(r) \end{pmatrix} = \begin{pmatrix} \hat{\rho}_2^{u,w}(r) \\ \hat{\rho}_3^{u,w}(r) \\ \vdots \\ \hat{\rho}_{Nr-2}^{u,w}(r) \\ \hat{\rho}_{Nr-1}^{u,w}(r) \end{pmatrix} \tag{2.11}$$

with

$$\begin{cases} p_i & = -(\frac{1}{\Delta r^2} - \frac{\alpha(r_i)}{2\Delta r}) \text{ where } \alpha(r_i) = \frac{1}{r} + \frac{1}{n_0(r_i)}\frac{d\, n_0(r_i)}{dr} \\ m_i & = \frac{2}{\Delta r^2} + \frac{m^2}{r_i^2} + (1-\delta_{u=0,w=0})\frac{1}{Z_i\, T_e(r_i)} \\ o_i & = -(\frac{1}{\Delta r^2} + \frac{\alpha(r_i)}{2\Delta r}) \\ \hat{\rho}_i^{u,w} & = \hat{\rho}^{u,w}(r_i) \end{cases} \tag{2.12}$$

We assume here vanishing Dirichlet boundary conditions in $r$ direction [10]: $\hat{\phi}_1^{u,w}(\varphi) = \hat{\phi}_{Nr}^{u,w}(\varphi) = 0$. The system (2.11) is solved using a $LU$ decomposition of the matrix (the decomposition can be computed only once).

**Solver based on 1D Fourier transform (new method)**

The previous formulation of the QN solver using 2D Fourier transforms has a main drawback from a parallelization point of view. In order to solve the equation (2.11) for a given couple $(u, w)$, one must compute 2D FFTs that require to know all values of $\tilde{\rho}$ in dimensions $\theta$ and $\varphi$. Then, systems are solved along dimension $r$, thus the 3 dimensions are tightly coupled. There is no simple domain decomposition of $\tilde{\rho}$ that leads to the design of a QN solver with a good load balance and that induces few communication with such 2D FFTs. The manageable algorithms perform global transpositions of $\hat{\rho}$ and $\hat{\phi}$ before or after FFT transforms (as it is shown in Algo. 3 hereafter). These transpositions constitute an overhead that one should avoid. So, we are looking for another method that does not need to consider all values in one of the three dimensions, and then uncouples the dimensions $(r, \theta, \varphi)$.

The main advantages of the method that follows (from a work distribution point of view) is to consider only 1D FFTs in $\theta$ dimension and to *uncouple hardly* all computations in $\varphi$ direction. The equation (2.2) averaged on $(\theta, \varphi)$ dimensions gives :

$$-\frac{\partial^2 \langle\phi\rangle_{\theta,\varphi}(r)}{\partial r^2} - [\frac{1}{r} + \frac{1}{n_0(r)}\frac{\partial n_0(r)}{\partial r}]\frac{\partial \langle\phi\rangle_{\theta,\varphi}(r)}{\partial r} = \langle\tilde{\rho}\rangle_{\theta,\varphi}(r) \tag{2.13}$$

A Fourier transform in $\theta$ direction gives:

$$\begin{array}{rl} \phi(r,\theta,\varphi) = & \sum_u \hat{\phi}^u(r,\varphi) e^{i\,u\,\theta} \\ \tilde{\rho}(r,\theta,\varphi) = & \sum_u \hat{\rho}^u(r,\varphi) e^{i\,u\,\theta} \end{array} \tag{2.14}$$

The equation (2.2) can then be rewritten as:

for $u > 0$ :

$$-\frac{\partial^2 \hat{\phi}^u(r,\varphi)}{\partial r^2} - [\frac{1}{r} + \frac{1}{n_0(r)}\frac{\partial n_0(r)}{\partial r}]\frac{\partial \hat{\phi}^u(r,\varphi)}{\partial r} + \frac{u^2}{r^2}\hat{\phi}^u(r,\varphi) + \frac{\hat{\phi}^u(r,\varphi)}{Z_i\, T_e(r)} = \hat{\rho}^u(r,\varphi) \tag{2.15}$$

for $u = 0$ :

$$\frac{\partial^2 \langle\phi\rangle_\theta(r,\varphi)}{\partial r^2} - [\frac{1}{r} + \frac{1}{n_0(r)}\frac{\partial n_0(r)}{\partial r}]\frac{\partial \langle\phi\rangle_\theta(r,\varphi)}{\partial r} + \frac{\langle\phi\rangle_\theta(r,\varphi) - \langle\phi\rangle_{\theta,\varphi}(r)}{Z_i\, T_e(r)} = \langle\tilde{\rho}\rangle_\theta(r,\varphi) \tag{2.16}$$

The equation (2.13) allows one to directly find out the value of $\langle\phi\rangle_{\theta,\varphi}(r)$ from the data $\langle\tilde{\rho}\rangle_{\theta,\varphi}(r)$. Let us define the function $\Upsilon(r,\theta,\varphi)$ as $\phi(r,\theta,\varphi) - \langle\phi\rangle_{\theta,\varphi}(r)$. Substracting equation (2.13) to equation (2.16) leads to

$$-\frac{\partial^2 \langle\Upsilon\rangle_\theta(r,\varphi)}{\partial r^2} - [\frac{1}{r} + \frac{1}{n_0(r)}\frac{\partial n_0(r)}{\partial r}]\frac{\partial \langle\Upsilon\rangle_\theta(r,\varphi)}{\partial r} + \frac{\langle\Upsilon\rangle_\theta(r,\varphi)}{Z_i\, T_e(r)} = \langle\tilde{\rho}\rangle_\theta(r,\varphi) - \langle\rho\rangle_{\theta,\varphi}(r) \tag{2.17}$$

Let us notice that $\hat{\phi}^0{}_{(r,\varphi)} = \langle\Upsilon\rangle_{\theta}{}_{(r,\varphi)} + \langle\phi\rangle_{\theta,\varphi}{}_{(r)}$. So, the solving of equations (2.13) and (2.17) allows one to compute $\langle\phi\rangle_{\theta,\varphi}{}_{(r)}$, $\langle\Upsilon\rangle_{\theta}{}_{(r,\varphi)}$ and $\hat{\phi}^0{}_{(r,\varphi)}$ from the quantities $\langle\tilde{\rho}\rangle_{\theta}{}_{(r,\varphi)}$ and $\langle\tilde{\rho}\rangle_{\theta,\varphi}{}_{(r)}$. Then, the

[10] in the GYSELA code, Neumann boundary conditions are also available.

equation (2.15) is sufficient to compute $\hat{\phi}^{u>0}(r,\varphi)$ from $\tilde{\rho}$. Let us notice that one has the equality $\hat{\phi}^{u>0}(r,\varphi) = \hat{\Upsilon}^{u>0}(r,\varphi)$. The different equations are solved using a LU decomposition in the same way that we have done previously in system (2.11), decomposition of the matrix is computed only once. Moreover, variable $\varphi$ acts as a parameter in equation (2.15), this allows what was expected: computations can be parallelized. This approach is able to replace the one based on 2D FFTs with now an effective parallelization along $\varphi$.

### 2.4.3 Algorithms for the QN solver

Hereafter, two algorithms are proposed to solve the QN equation based on the description of the methods given in the last paragraphs. These algorithms will be denoted by `2d_fft` and `p1d_fft`. To distribute computation of part 1, all algorithms use a single formulation to obtain $\tilde{\rho}$ data structure from integrals on $f$. Techniques described in the previous subsections are used to compute the part 2 that derives $\phi$. Each solution will be presented in a peculiar paragraph. We assume two main hypothesis concerning the data distribution: i) initially each process knows the value of a block $f(r = local, \theta = local, \varphi = *, v_{//} = *, \mu = value)$, ii) at the end the whole 3D data $\phi$ will be known on all processes. In forthcoming works, we will remove the latter hypothesis to only produce blocks $\phi(r = local, \theta = local, \varphi = *)$ on each process. But data dependencies in GYSELA prevent us to do so as a first stage (see 2.4.4 for the second stage).

#### Partial parallel algorithm with 2D FFT

The algorithm that follows (Algo. 3, p.32), focuses singly on the parallelization of part 1. The part 2, that performs the $\phi$ computation, uses an non-parallelized approach with 2D FFTs. The data structure $\tilde{\rho}_1$ and $\tilde{\rho}_2$ store partial results in order to obtain $\tilde{\rho}$.

```
1  Input : local block f(r = local, θ = local, φ = *, v_// = *, μ)
2
3        (* part 1 *)
5  Computation : ρ̃_1 by integration in dv_// of f
6    (parallelization in μ, r, θ)
7  Send local data ρ̃_1(r = local, θ = local, φ = *, μ)
8  Redistribute ρ̃_1 / Synchronization
9  Receive block ρ̃_1(r = *, θ = *, φ = local, μ = *)
10 for local φ values do in parallel
11 |    (parallelization in φ)
12 |  Computation : ρ̃_2 for a given φ by application of operator J_0 on ρ̃_1
13 |    Fourier transform in θ, Solving of LU systems in r
14 |  Computation : ρ̃ for a given φ by integration in dμ of ρ̃_2
15 Send local data ρ̃(r = *, θ = *, φ = local)
16 Broadcast of ρ̃ / Synchronization
17 Receive global data ρ̃(r = *, θ = *, φ = *)
18
19       (* part 2, not parallelized with MPI *)
20 Computation : 2D FFTs of ρ̃ on dimensions (θ,φ)
21   Computation of ρ̂
22 Computation : Solving of LU systems on dim. r ,
23   Computation of φ̂
24 Computation : 2D inverse FFTs 2D on dim. (θ,φ)
25
26 Outputs : φ(r = *, θ = *, φ = *)
```

**Algorithm 3:** Partial parallelization of QN solver (`2d_fft`)

Let #P be the number of processes. The 5D data $f$ has size $(\mathrm{N_r}\,\mathrm{N_\theta}\,\mathrm{N_\varphi}\,\mathrm{N_{v\|}}\,\mathrm{N_\mu})$, whereas the 3D data $\tilde{\rho}$ has size $(\mathrm{N_r}\,\mathrm{N_\theta}\,\mathrm{N_\varphi})$. Because of the 2D FFTs, the computation complexity of part 2 is in $\Theta(\mathrm{N_\theta}\,\mathrm{N_\varphi}\,\mathrm{N_r}\,ln(\mathrm{N_\theta})\,ln(\mathrm{N_\varphi}))$. The data structures $\tilde{\rho}$ and $\phi$ have size $\Theta(\mathrm{N_\theta}\,\mathrm{N_\varphi}\mathrm{N_r})$. We have chosen to perform the part 2 redundantly on each process. The parallelization of part 2 would imply adding communication to redistribute data after line 21 and before line 24. The parallelization with these extra communication can not be scalable on many processes because of communication over complexity costs ratio. This Algo. (3) will be taken as a reference to evaluate the other algorithms presented afterwards.

The communications in part 1 consist in a global transposition of data $f$ (line 8) and a broadcast of distributed data $\tilde{\rho}$ (line 16). The respective costs in term of global volume exchanged are $\mathrm{N_\theta}\,\mathrm{N_\varphi}\,\mathrm{N_r}\,\mathrm{N_\mu}$ double precision floating point values for the first one and $\mathrm{N_\theta}\,\mathrm{N_\varphi}\,\mathrm{N_r}\,(\#\mathrm{P}-1)$ for the second one.
The method that evaluates the gyroaverage operator introduced in section 2.4.2 requires that we know, for a given couple $(\varphi,\mu)$, all values $\tilde{\rho}_1(r=*,\theta=*,\varphi,\mu)$. Furthermore, we have to integrate over $d\mu$ (line 14) to get $\tilde{\rho}$ ; to avoid extra communication, it is safe to put all $\tilde{\rho}_1(r=*,\theta=*,\varphi,\mu=*)$ on

the same process in order to perform the integrals locally. So, with our gyroaveraging method (that couples $r$ and $\theta$ dimensions), the parallel algorithm we have introduced induces small communication cost and good load balance. It mainly uses a parallel loop in $\varphi$ at line 10.

We now assume that $\tilde{\rho}_1$, which is the output of part 1 computations, is distributed along with $\varphi$ over the parallel machine because of previous arguments about the constraints of gyroaverage[11]. As we expect that each process finally knows the data $\phi$, then a global communication will transfer parts of locally computed $\phi$ structure between processes. With these hypothesis, the communication cost of $\mathrm{N}_\theta\,\mathrm{N}_\varphi\,\mathrm{N}_\mathrm{r}\,(\#\mathrm{P}-1)$ is best reduced. So, from our actual knowledge, the proposed algorithm for part 1, does the minimum possible volume of communication together with a reasonable computation distribution. The parallel overhead is low for this solution and it is difficult to further reduce it.

### Full parallel algorithm with 1D FFT

The last algorithm version `f1d_fft`, presented here, describes a full parallel algorithm (excluding a small redundant computation line 23). We will use the intermediate function $\Upsilon(r,\theta,\varphi) = \phi(r,\theta,\varphi) - \langle\phi\rangle_\theta(r,\varphi)$. The main idea is to compute $\langle\phi\rangle_{\theta,\varphi}$ on its own, and also to perform the computations $\langle\Upsilon\rangle_{\theta(r=*,\varphi)}$ in parallel. At the end we combine the results to get $\phi(r=*,\theta=*,\varphi)$. The amount of communication added at line 18 is $O(\mathrm{N}_\mathrm{r}\,\mathrm{N}_\varphi\,\#\mathrm{P})$. This is negligible when one considers other communication costs. Nevertheless, a synchronization of processes is also induced.

1 *Input :* local block $f(r=local,\theta=local,\varphi=*,v_{//}=*,\mu)$
2
3 *(* part 1*)*
4 *Computation :* $\tilde{\rho}_1$ by integration in $dv_{//}$ of $f$
5 (parallelization in $\mu,r,\theta$)
6 **Send** local data $\tilde{\rho}_1(r=local,\theta=local,\varphi=*,\mu)$
7 **Redistribute** $\tilde{\rho}_1$ **/ Synchronization**
8 **Receive** block $\tilde{\rho}_1(r=*,\theta=*,\varphi=local,\mu=*)$
9 **for** *local* $\varphi$ *values* **do in parallel**
10 (parallelization in $\varphi$)
11 *Computation :* $\tilde{\rho}_2$ for a given $\varphi$ by application of operator $J_0$
12 Fourier transform in $\theta$, Solving of LU systems in $r$
13 *Computation :* $\tilde{\rho}$ for a given $\varphi$ by integration in $d\mu$ of $\tilde{\rho}_2$
14 *Computation :* accumulation of $\tilde{\rho}$ values to get $\langle\tilde{\rho}\rangle_\theta(r=*,\varphi)$
15
16 *(* part 2*)*
17 **Send** local data $\langle\tilde{\rho}\rangle_{\theta(r=*,\varphi=local)}$
18 **Broadcast of** $\langle\tilde{\rho}\rangle_\theta$ **/ Synchronization**
19 **Receive** $\langle\tilde{\rho}\rangle_{\theta(r=*,\varphi=*)}$
20 *Computation :* Solving of LU system to find $\langle\phi\rangle_{\theta,\varphi}$ from $\langle\tilde{\rho}\rangle_\theta$, eq. (2.13)
21 **for** *local* $\varphi$ *values* **do in parallel**
22 (parallelization in $\varphi$)
23 *Computation :* 1D FFTs of $\tilde{\rho}$ on dimension ($\theta$)
24 *Computation :* Solving of LU systems for $\hat{\phi}$ modes ($u>0$), eq. (2.15)
25 *Computation :* Solving of LU system for $\langle\Upsilon\rangle_{\theta(r=*,\varphi)}$, eq. (2.17)
26 *Computation :* Adding $\langle\phi\rangle_{\theta,\varphi}$ to $\langle\Upsilon\rangle_{\theta(r=*,\varphi)}$ gives $\hat{\phi}^0{}_{(r=*,\varphi)}$
27 *Computation :* inverse 1D FFTs on $\hat{\phi}^0$ and $\hat{\phi}^{u>0}$ to get $\phi_{(r=*,\theta=*,\varphi)}$
28 **Send** local data $\phi(r=*,\theta=*,\varphi=local)$
29 **Broadcast of values / Synchronization**
30 **Receive** global data $\phi(r=*,\theta=*,\varphi=*)$
31 *Outputs :* $\phi(r=*,\theta=*,\varphi=*)$

**Algorithm 4:** Full Parallelization of QN solver (`f1d_fft`)

### Performance Analysis

Numerical experiments (2009-2010) were performed on a cluster of 932 nodes owned by CCRT/CEA, France. Each node hosted eight Itanium2 1.6Ghz cores and offered 24GB of shared memory. Performance of the different versions of the QN solver for one 5D test case are presented in Table 4 ($\mathrm{N}_\mathrm{r}=256,\mathrm{N}_\theta=64,\mathrm{N}_\varphi=256,\mathrm{N}_{\mathrm{v}_\parallel}=16,\mathrm{N}_\mu=32$). Note that the Vlasov solver of the GYSELA code uses a parallelization based on a domain decomposition in dimension $\mu$ and $r$ (not possible to parallelize along $\theta$ at this time). So, the number of cores #P is given by the product of $\mathrm{p}_\mu$ the number of $\mu$ values with $\mathrm{p}_\mathrm{r}$ the number of block in dimension $r$. The number of $\mu$ values in the presented test case is $\mathrm{p}_\mu=32$, then our parallel program uses a minimum of 32 cores. Then, the relative speedups shown in Table 4 considers as a reference the execution times on 32 cores of four computation nodes.

In order to give fine performance results, we will subdivide the algorithms into small recognizable parts. The computation of $\tilde{\rho}$ (part 1) consists in a communication part and parallelized integral

[11] this strategy could be revised since the work of N. Bouzat in 2016 that considers a gyroaverage operator based on Hermite interpolation (see Section 2.6).

calculations. For the different versions, the solver that gives $\phi$ depending on $\tilde{\rho}$ (part 2) could be decomposed into communication steps plus two types of computation: the redundant ones and the parallel ones. Finally, one can have a look to time costs of the QN solver considering three measures: i) the time spent in communication (the larger timing among all processes is given in Table 4), ii) the time spent in sequential computations (each process has exactly the same work to do), iii) the time spent in parallel tasks (the maximum among all processes is taken).

| Nb. procs. (#P) | 32 | 128 | 256 |
|---|---|---|---|
| $p_r$ | 1 | 4 | 8 |
| **Algo.** `2d_fft` | | | |
| comm | 0.350 s | 0.687 s | 0.767 s |
| solve_seq | 4.208 s | 4.613 s | 4.535 s |
| solve_par | 3.908 s | 1.076 s | 0.555 s |
| Rel. speedup solve_par+_seq | 1.0 | 1.4 | 1.6 |
| Total `2d_fft` | 8.444 | 6.299 | 5.771 |
| Rel. speedup `2d_fft` | 1.0 | 1.3 | 1.5 |
| **Algo.** `f1d_fft` | | | |
| comm | 0.377 s | 0.593 s | 0.668 s |
| solve_seq | 0.003 s | 0.006 s | 0.018 s |
| solve_par | 4.078 s | 1.039 s | 0.528 s |
| Rel. speedup solve_par+_seq | 1.0 | 3.9 | 7.7 |
| Total `f1d_fft` | 4.375 s | 1.603 s | 1.178 s |
| Rel. speedup `f1d_fft` | 1.0 | 2.7 | 3.7 |

Table 4: Comparison of the algorithms introduced. Time measurements for one call to the QN solver in seconds and relative speedup are given (compared to performance on 32 cores).

The execution time measurements of algorithms `2d_fft` and `f1d_fft` are given in Table 4. For the reference algorithm `2d_fft`, there is only one computation part that is parallelized: the $\tilde{\rho}$ computation (part 1). The timing results shows this part is scalable. A major problem is that the `solve_seq` becomes the dominant calculation as soon as #P goes above a threshold. The Amdhal's law limits the overall performance of this algorithm with usual parameter sets. As a consequence, the relative speedup on 256 cores, 1.5 compared with 32 cores as a reference, is extremely low. The communication part is partly responsible for that; it has an overall cost in $\Theta(N_\theta\, N_\varphi\, N_r\, N_\mu + N_\theta\, N_\varphi\, N_r\, (\#P - 1))$ that increases with #P.

The `f1d_fft` algorithm improves definitely the previous `2d_fft` algorithm. A reduced amount of work to perform (1D FFT instead 2D FFT) is observed in the first column for #P = 32. In the `f1d_fft` algorithm, almost all computations are parallelized and the speedup of the computation part (`solve_par+_seq`) is impressive: 7.7 compared to 8 in the ideal case. The remaining parallel overhead comes from the small sequential computation of `solve_seq` and above all communication `comm`. This QN solver is efficient and reduces on 256 cores the recorded time of 5.7 s with `2d_fft` to 1.2 s with `f1d_fft`.

One could think about using other methods such as multigrid or a direct solver for the part 2 of the solver. However, it will not diminish the cost of communications required for the calculation of $\tilde{\rho}$ and for the final broadcast. Hence, we cannot expect that using alternate methods to yield a large enhancement of the parallel performance as long as communication costs remain steady.

**Conclusion**

Parallelizations of a quasi-neutral Poisson solver have been shown. The parallel performance of two solving methods is demonstrated. The last method achieves good scalability up to 256 cores (MPI only approach, no OpenMP here). Thus, we were able to remove a major performance bottleneck from the field solver and this paved the way for running the application on hundreds of processors.

Nevertheless, the communication induced by the coupling of the quasi-neutral solver and the Vlasov code remains quite high. More work has to be done in order to reduce these communication costs to further improve this solver. Avoiding the final broadcast of $\phi$ which is a 3D data structure is a possible solution (see next Section). In addition, combining MPI with OpenMP will help to further reduce the costs (see hereafter).

### 2.4.4 Highly scalable field solver

The parallel solution proposed in the previous Section for QN solver improves a lot the scalability compared to initial status. Nevertheless, beyond thousands of cores, the field solver is still one of

the main bottleneck for the scalability of the application. The communications gathering data at the beginning of the field solver and the broadcast at the end are costly. Furthermore, several 3D data structures are replicated on each MPI process that increases artificially the memory footprint. This memory problem is painful and it practically prohibits many large physical cases because of *memory exhaust* issues. I proposed some improvements that are described in this Section to solve these difficulties and also to add OpenMP in addition to MPI [29].

### Data distribution issues

In the Vlasov solver and at the beginning of QN solver each process knows the values of a subdomain $f(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *, \mu = \mu_{value})$. At the end of the QN solver, the field $\phi$ is an output that one would like to distribute over the parallel machine. In previous Section and paper [91], we had simplified the problem of data dependencies and provided an unsophisticated solution: broadcasting the entire $\phi$ data structure to all processes. Then, in each MPI process, the derivatives of gyroaveraged electric potential are computed redundantly. Nevertheless, this strategy leads to a bottleneck for large platforms (typically more than 4k cores). Indeed, the broadcast involves growing communication costs along with the number of processes, and the sequential treatment of derivatives computation (Amdahl's law) becomes also problematic. These two overheads are unnecessary, even they greatly simplify the implementation of several subroutines and reduces also the complexity of data management. Now, we will only consider a subdomain of $\phi$ to be sent to each process. Also, a distributed algorithm computes the derivatives of $J_0\,\phi$, as you will see in Algo. 6.

*Input :* local block $f(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *, \mu = \mu_{value})$
    *(* task 1*)*
*Computation :* $\tilde{\rho}_1$ by integration in $dv_\parallel$ of $f$ `/* parallel in` $\mu, r, \theta$ `*/`
**Send** local data $\tilde{\rho}_1(r = [i_{start}, i_{end}]), \theta = [j_{start}, j_{end}], \varphi = *, \mu = \mu_{value})$
**Redistribute** $\tilde{\rho}_1$ **/ Synchronization**
**Receive** block $\tilde{\rho}_1(r = *, \theta = *, \varphi = [s^\varphi_{start}, s^\varphi_{end}], \mu = [s^\mu_{start}, s^\mu_{end}])$
    *(* task 2*)*
**for** $\varphi = [s^\varphi_{start}, s^\varphi_{end}]$ and $\mu = [s^\mu_{start}, s^\mu_{end}]$ **do** `/* parallel in` $\mu, \varphi$ `*/`
    *Computation :* from $\tilde{\rho}_1$ at one $\varphi$, compute $\tilde{\rho}_2$ applying $J_0$
      (Fourier transform in $\theta$, Solving of LU systems in $r$) $\forall \mu \in [s^\mu_{start}, s^\mu_{end}]$
    *Computation :* $\tilde{\rho}_3$ for a given $\varphi$ by integration in $d\mu$ of $\tilde{\rho}_2$
**if** $[s^\mu_{start}, s^\mu_{end}] \neq [0, N_\mu - 1]$ **then**
    **Send** local data $\tilde{\rho}_3(r = *, \theta = *, \varphi = [s^\varphi_{start}, s^\varphi_{end}])$
    **Reduce Sum** $\tilde{\rho}$ += $\tilde{\rho}_3$ **/ Synchronization**
    **Receive** summed block $\tilde{\rho}(r = *, \theta = *, \varphi = [g_{start}, g_{end}])$
    *(* task 3*)*
**for** $\varphi = [g_{start}, g_{end}]$ **do** `/* parallel in` $\varphi$ `*/`
    *Computation :* accumulation of $\tilde{\rho}$ values to get $\langle\tilde{\rho}\rangle_\theta\,(r = *, \varphi)$
**Send** local data $\langle\tilde{\rho}\rangle_\theta(r=*,\varphi=[g_{start},g_{end}])$
**Broadcast of** $\langle\tilde{\rho}\rangle_\theta$ **/ Synchronization**
**Receive** $\langle\tilde{\rho}\rangle_\theta(r=*,\varphi=*)$
    *(* task 4*)*
*Computation :* Solving of LU system to find $\langle\phi\rangle_{\theta,\varphi}$ from $\langle\tilde{\rho}\rangle_{\theta,\varphi}$, eq. (2.13)
**for** $\varphi = [g_{start}, g_{end}]$ **do** `/* parallel in` $\varphi$ `*/`
    *Computation :* 1D FFTs of $\tilde{\rho}$ on dimension $(\theta)$
    *Computation :* Solving of LU systems for $\hat{\phi}^m$ modes $(\forall m > 0)$, eq. (2.15)
    *Computation :* Solving of LU system for $\langle\Upsilon\rangle_\theta(r=*,\varphi)$, eq. (2.17)
    *Computation :* Adding $\langle\phi\rangle_{\theta,\varphi}$ to $\langle\Upsilon\rangle_\theta(r=*,\varphi)$ gives $\hat{\phi}^0(r=*,\varphi)$
    *Computation :* inverse 1D FFTs on $\hat{\phi}^0$ and $\hat{\phi}^{m>0}$ to get $\phi(r=*,\theta=*,\varphi)$
**Send** local data $\phi(r = *, \theta = *, \varphi = [g_{start}, g_{end}])$ to the $N_\mu$ communicators
**Broadcast of values / Synchronization**
**Receive** global data $\phi(r = *, \theta = *, \varphi = [q_{start}, q_{end}])$
*Outputs :* $\phi(r = *, \theta = *, \varphi = [q_{start}, q_{end}])$

**Algorithm 5:** Parallel algorithm for the QN solver

### Parallel algorithm descriptions

The Algo. 5 describes a scalable parallel algorithm of the QN solver. It improves previous approaches of Algo. 4 in introducing a better work distribution, and also in reducing the amount of the final communication. The main idea of the algorithm is to get $\langle\phi\rangle_{\theta,\varphi}$ for solving Eq. (2.17), and then to uncouple computations of $\hat{\phi}^m$ along the $\varphi$ direction in the QN solver. Finally, each locally computed $\phi(r, \theta, \varphi)$ values are sent to the process that is responsible for it. This algorithm has parameters: mappings (domain decompositions) $s$, $g$ and $q$ that will be detailed. The computation sequence is: integrate $f$ over $v_\parallel$ direction (task 1), compute right-hand side $\tilde{\rho}$ in summing over $\mu$ (task 2), perform averages $\langle\tilde{\rho}\rangle_\theta(r=*,\varphi=*)$ and $\langle\tilde{\rho}\rangle_{\theta,\varphi}(r=*)$ (task 3), solve QN equation and produces $\phi$ slices (task 4).

*Input :* local block $\phi(r=*,\theta=*,\varphi=[q_{start},q_{end}])$
*(* task 1*)*
**for** $\varphi=[q_{start},q_{end}]$ and $\mu=\mu_{value}$ **do** `/* parallel in` $\mu,\varphi$ `*/`
  *Computation :* $A_0(r=*,\theta=*,\varphi)=J_0\,\phi(r=*,\theta=*,\varphi)$
  $A_1(r=*,\theta=*,\varphi)=\frac{\partial A_0(r=*,\theta=*,\varphi)}{\partial r}$
  $A_2(r=*,\theta=*,\varphi)=\frac{\partial A_0(r=*,\theta=*,\varphi)}{\partial \theta}$
**Send** local data $A_0|A_1|A_2(r=*,\theta=*,\varphi=[q_{start},q_{end}])$
**Redistribute** $A_0|A_1|A_2$ **inside** $\mu$ **communicator/ Synchronization**
**Receive** blocks $A_0|A_1|A_2(r=[i_{start},i_{end}],\theta=[j_{start},j_{end}],\varphi=*)$
*(* task 2*)*
**for** $r=[i_{start},i_{end}]$, $\theta=[j_{start},j_{end}]$, $\mu=\mu_{value}$ **do** `/* parallel in` $\mu,r,\theta$ `*/`
  *Computation :* $A_3(r,\theta,\varphi=*)=\frac{\partial\varphi}{\partial A_0(r,\theta,\varphi=*)}$
*Outputs :* $A_0|A_1|A_2|A_3(r=[i_{start},i_{end}],\theta=[j_{start},j_{end}],\varphi=*)$

**Algorithm 6:** Parallel algorithm to get derivatives of the potential

The Algo. 6 follows immediately the QN solver. It applies the gyroaverage operator on distributed $\phi$ data and then computes its derivatives along spatial dimensions. These 3D fields are named $A_1, A_2, A_3$ in the algorithm. They are redistributed in a communication step in order to correctly feed the Vlasov solver. In the task 2, derivatives along $\varphi$ direction are computed (all values are known along $\varphi$).

### Mapping functions

The two presented algorithms use three different mappings to distribute computations and data on the parallel machine. These mappings concern $\varphi$ and $\mu$ variables and are illustrated in Fig. 17 and 18 for a large testbed and a small one respectively. Let #C be the number of cores used for a simulation run and #P be the number of MPI processes. The number of threads #T per MPI process is fixed, so we have #C = #P #T. Each rectangle on the Figs. 17 and 18 represents a MPI process. A

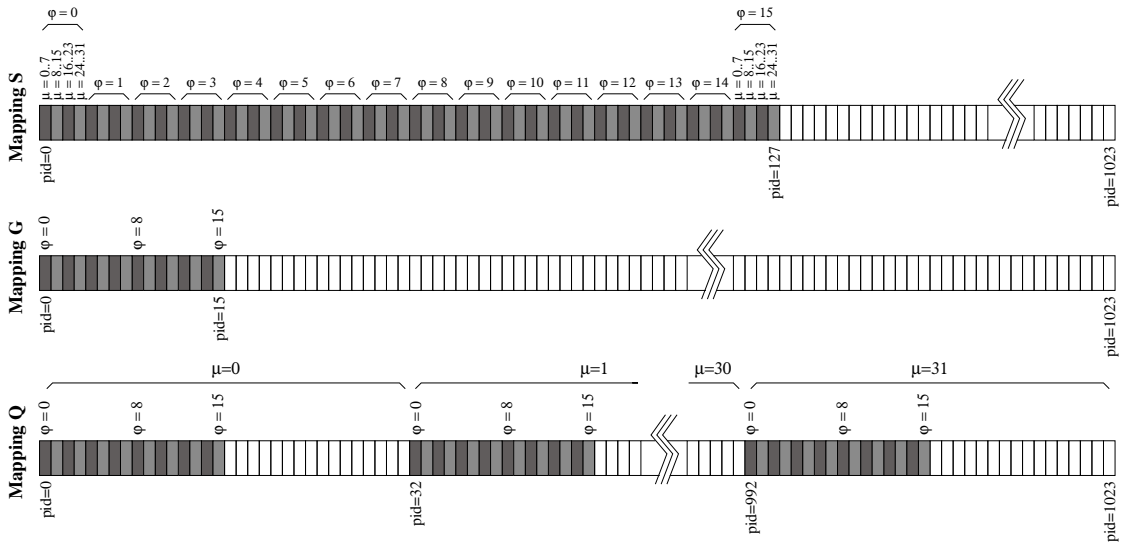



Figure 17: Mappings of $\mu,\varphi$ ($\mathrm{N}_\mu=32,\mathrm{N}_\varphi=16$) on processes, large testbed (#P = 1024)

process filled in dark or light gray has computations to perform, whereas the white color denotes an idle process. These mappings also implicitly prescribe how communication schemes exchange data during the execution of the two algorithms. We give here a brief description of these mappings:

- **Mapping S** - It defines the ranges $\varphi\in[s^{\varphi}_{start},s^{\varphi}_{end}]$ and $\mu\in[s^{\mu}_{start},s^{\mu}_{end}]$. In the task 2 of QN solver, we use this mapping to distribute the computation of the gyroaverage $J_0$. The maximal parallelism is then obtained whenever each core has at most one gyroaverage operator to apply. We have considered in the example shown in Fig. 17 that each MPI process hosts #T = 8 threads, so that a process can deal with 8 gyroaveraging simultaneously ($s^{\mu}_{end}-s^{\mu}_{start}+1=8$). This distribution is computed in establishing a block distribution of domain $[0,\mathrm{N}_\mu-1]\times[0,\mathrm{N}_\varphi-1]$ over #C cores.
- **Mapping G** - It defines the range $[g_{start},g_{end}]$ for the $\varphi$ variable. A simple block decomposition is used along $\varphi$ dimension. For a large number of cores, this distribution gives to processes 0 to $\mathrm{N}_\varphi-1$ the responsibility to compute the $\mathrm{N}_\varphi$ slices of $\phi$ data structure (task 4 of the QN

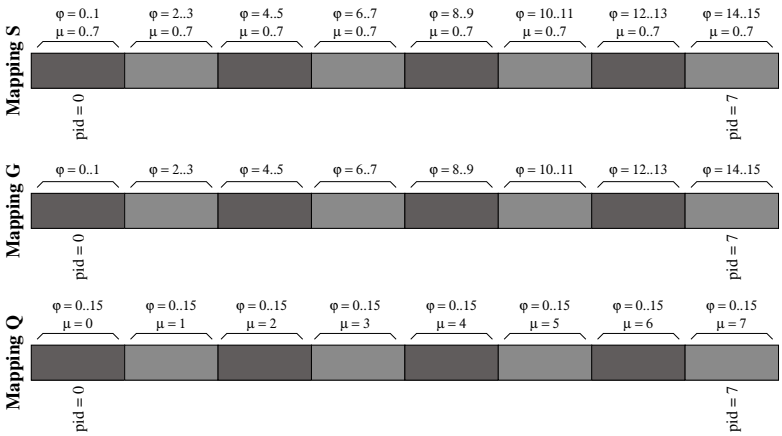



Figure 18: Mappings of $\mu, \varphi$ ($\mathrm{N}_\mu = 8, \mathrm{N}_\varphi = 16$) on processes, small testbed (#P = 8)

solver). For a small testbed (see Fig. 18), this mapping is identical to S mapping ($[s^\mu_{start}, s^\mu_{end}] = [0, N_\mu - 1]$). Then, we save the communication step in task 2 of QN solver.

- **Mapping Q** - The mapping Q defines the range $\varphi \in [q_{start}, q_{end}]$ in each MPI process. Inside each $\mu$ communicator, it creates a block decomposition along $\varphi$ dimension. It is designed to carry out the computation of the gyroaverage of $\phi$, together with the computation of its derivatives (Algo. 6). These calculations depend on the value of $\mu$. It is cost effective to perform them inside $\mu$ communicators: we need to only locally redistribute data inside the $\mu$ communicator at the end of Algo. 6 in order to prepare the input 3D data fields for the Vlasov solver.

#### Communication costs

A fine communication costs analysis [29] can be conducted. Algo. 5 and 6 have an overall cost in $\Theta(\mathrm{N}_r\,\mathrm{N}_\theta\,\mathrm{N}_\varphi\,\mathrm{N}_\mu)$ floats. In short, the annoying #P multiplication factor has been suprressed.

### 2.4.5 Large scale experiments

#### Performance - scalability on Jaguar Machine

Timing measurements were performed on CRAY-XT5 Jaguar machine (DOE, USA) during year 2011. This machine had 18 688 XT5 nodes hosting dual hex-core AMD processors, 16 GB of memory. Table 5 reports timing of the QN solver extracted from GYSELA runs. $\mathrm{N}_\mu$ remains constant while $\mathrm{p_r}$ and $\mathrm{p}_\theta$ are increased. A small case was run from 256 cores to 4096 cores. The parameters are the following:

$$N_r = 128, N_\theta = 256, N_\varphi = 128, N_{v_\parallel} = 64, N_\mu = 32\ .$$

In Table 6, timings for a bigger test case are presented. Its size is

$$N_r = 512, N_\theta = 512, N_\varphi = 128, N_{v_\parallel} = 128, N_\mu = 32\ .$$

For this second case, the parallel testbed was composed of 4k cores up to 64k cores. In the tables, the `io[1-4]` steps state for communications associated with task[1-4] respectively, whereas `comp[1-4]` stand for computation costs relative to task[1-4]. The Vlasov solver of the GYSELA code is parallelized using a domain decomposition in $\mu, r, \theta$. The number of processes #P equals $\mathrm{N}_\mu \times \mathrm{p_r} \times \mathrm{p}_\theta$.

*General observations* We map a single MPI process per node. The main idea for this OpenMP parallelization has been to target $\varphi$ loops. This approach is efficient for the computation task 1 of the QN solver. But in the tasks 3, 4, this strategy competes with the MPI parallelization that also uses the $\varphi$ variable for its domain decomposition. Thus, above $N_\varphi$ cores, no parallelization gain is expected. This fact is not the hardest constraint up to now: communication costs are the critical overhead, much more than computation distribution for these tasks (see Table 6). An additional improvement has been to add a parallelization along the $\mu$ direction in task 2. Even if this change adds a communication step (`io2`) that can be avoided, it is worthwhile on large platforms. Notably, we see that `comp2` scales beyond $N_\varphi = 128$ cores in Table 6.

*Comments for the small case* In Table 5, the communication costs for exchanging $\tilde{\rho}_1$ values (`io1` - task 1) is reduced along with the involved number of nodes. This is explained by the fact that the cumulative network bandwidth increases with a larger number of nodes, while the amount of data exchanged remains the same. The communication cost associated with `io3` is mainly composed of synchronization of nodes and broadcasting the 2D slice $\langle\tilde{\rho}\rangle_\theta\,(r = *, \varphi = *)$. The `io4` communication involves a selective send of $\phi$ slices to each process; and one can note the same decreasing behavior

| Nb. cores | 256 | 1k | 4k |
|---|---|---|---|
| Nb. nodes | 32 | 128 | 512 |
| comp1 | 2300 ms | 580 ms | 100 ms |
| io1 | 300 ms | 160 ms | 90 ms |
| comp2 | 170 ms | 43 ms | 13 ms |
| io2 | 0 ms | 0 ms | 8 ms |
| comp3 | 0 ms | 0 ms | 2 ms |
| io3 | 17 ms | 42 ms | 43 ms |
| comp4 | 4 ms | 3 ms | 4 ms |
| io4 | 100 ms | 40 ms | 35 ms |
| Total time | 2900 ms | 870 ms | 300 ms |
| Relative eff. | 100% | 83% | 60% |

Table 5: Time measurements for one call to the QN solver - Small case

| Nb. cores | 4k | 16k | 64k |
|---|---|---|---|
| Nb. nodes | 512 | 2k | 8k |
| comp1 | 2200 ms | 570 ms | 95 ms |
| io1 | 450 ms | 300 ms | 470 ms |
| comp2 | 320 ms | 180 ms | 180 ms |
| io2 | 30 ms | 60 ms | 70 ms |
| comp3 | 3 ms | 2 ms | 3 ms |
| io3 | 150 ms | 140 ms | 130 ms |
| comp4 | 30 ms | 30 ms | 30 ms |
| io4 | 180 ms | 100 ms | 65 ms |
| Total time | 3400 ms | 1400 ms | 1000 ms |
| Relative eff. | 100% | 61% | 21% |

Table 6: Time measurements for one call to the QN solver - Big case

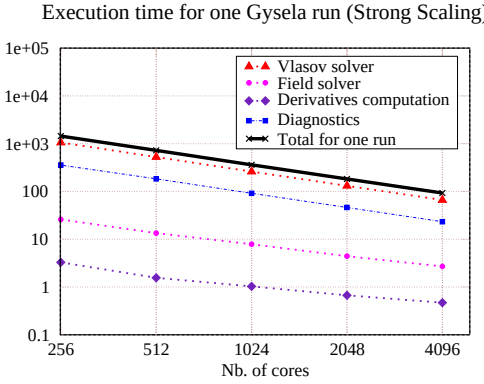



Figure 19: Small case - timings - Jaguar

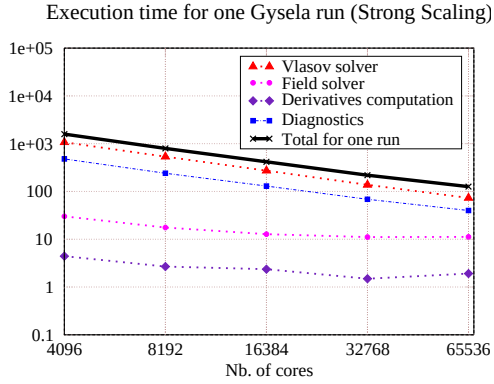



Figure 20: Big case - timings - Jaguar

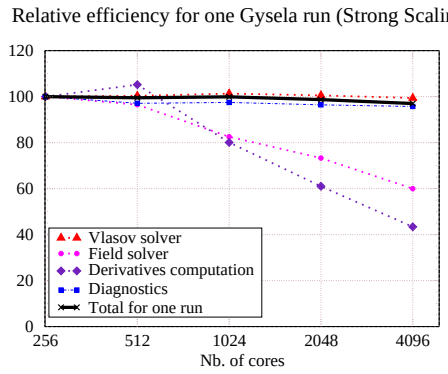



Figure 21: Small case - efficiency - Jaguar

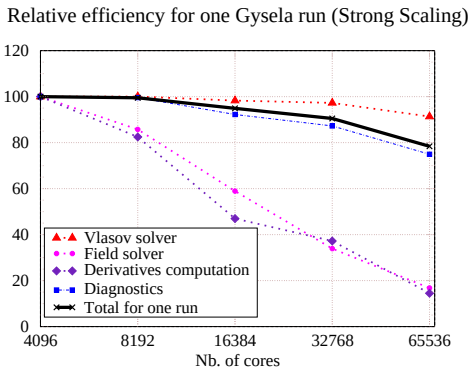



Figure 22: Big case - efficiency - Jaguar

depending on the number of cores observed for `io1`. The `comp1` calculation is a big CPU consumer as it scales well with the number of cores. The `comp3` is negligible time and `comp4` is a small computation step, time measurements are nearly constant for all number of cores shown. As already said, $N_\varphi$ cores is the upper bound of the parallel decomposition here. The relative efficiency for the overall QN solver is 60% at 4k cores which is a good result for this solver that collects/redistributes data between all cores.

*Comments for the big case* The relative speedups shown in Table 6 consider as a reference the execution times on 4k cores. Communication costs are larger than in the small test case. Badly, `comp2` and `comp4` do not scale well (the case is not big enough). Only `comp1` and `io4` parts behave as one can expect. In future works, we expect improving the scalability of this algorithm: reduction of communication costs is the first candidate (we investigate compression methods). Even if improvements can be found, a good property of this solver is that the execution time globally decreases along with the number of cores. We see in Fig. 22 that this cheap cost of the QN solver brings a good overall scalability. GYSELA reaches 78% of relative efficiency at 64k cores. in Fig. 19 and 20, timings for short runs of

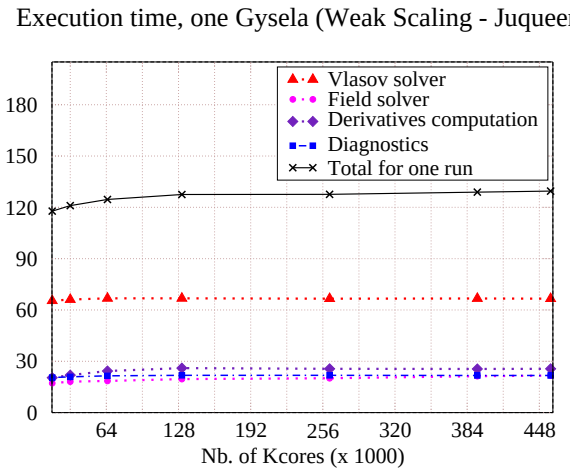



Figure 23: Weak scaling, timings - domain size $512\times1024\times128\times128\times N_\mu$ with $N_\mu \in [2-56]$

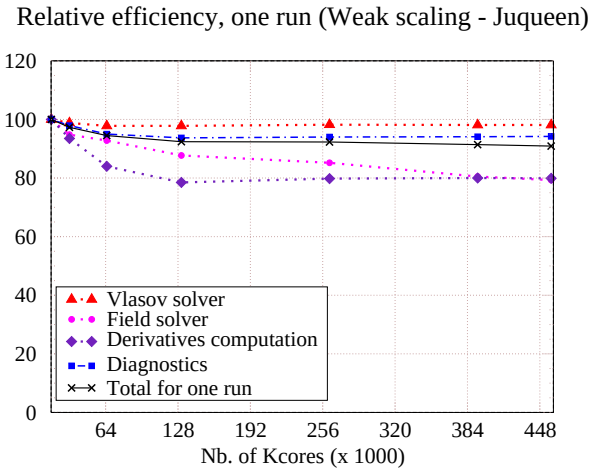



Figure 24: Weak scaling, efficiency - domain size $512\times1024\times128\times128\times N_\mu$ with $N_\mu \in [2-56]$

GYSELA are presented (initialization step is omitted). The field solver and derivatives computation of the gyroaveraged $\phi$ are low compared to Vlasov solver and diagnostics costs. Then, their limited scalability at high core counts does not impact significantly the scalability of the overall code. Let us remark the excellent scalability for the small case (Fig. 21) with 97% of overall efficiency at 4k cores.

**Performance - scalability on Juqueen Machine**

*Architecture* The experiments we describe now have been conducted in 2013 on Blue Gene/Q machines JUQUEEN at JSC/IAS, Jülich, Germany. The Blue Gene/Q architecture is based on 209 TeraFlop/s peak performance racks grouping 1024 nodes each. JUQUEEN is composed of 24 racks ranking #5 in the November 2012 Top500 list [189]. Each compute node contains a single 17-cores processor running at 1.6 GHz. One core is dedicated to the operating system while the others are available for the user code. The cores support Simultaneous MultiThreading (SMT) and can handle up to 4 hardware threads each and execute up to two instructions per cycle, one logical instruction and one floating point instruction. Each core has its own four-wide SIMD floating point unit capable of executing up to height operations per cycle resulting in 12.8 GFlop/s. Each node contains 16 GB of 1.33 GHz memory and 32 MB of L2 cache (16 $\times$2 MB banks). The theoretical peak memory bandwidth is 42.6 GB/s, but running the STREAM benchmark shows an effective bandwidth of 30 GB/s. The Blue Gene/Q network is a re-configurable 5D mesh or torus. Each node is connected to two neighbors in each of the 5 dimensions via a full duplex 1.8 GB/s link resulting in 18 GB/s accumulated bandwidth for each node. When making a reservation, a rectangular partition of the full machine is electrically isolated and attributed to the user. This prevents any interaction between two users especially in terms of performance.

*Specific improvements for Blue Gene/Q* This paragraph presents our efforts [26] to increase the computational efficiency of GYSELA on Blue Gene/Q. First, a fine analysis led us to use 64 threads by process (4 threads by core). The rationale is quite similar to the one presented in Sections 2.3.3 and 4.3, the use of SMT maximizes the number of instructions that a core can process each cycle. Second, in order to increase the efficiency when using more than 16 threads per process (quite unusual back at this time), we analyzed the parallel loops. Some loops do not scale well, because at some point, the are less iterations than available threads. We identified three parallel loops concerned by this problem. We were able to introduce novel parallel algorithms based on nested loop parallelization ensuring that over 64 parallelizable iterations are yielded. Third, when analyzing the performance, we also identified two global communications whose performance did not reach our expectations. The first communication is used to transpose back and forth the data from a distribution in $r$, $\theta$ and $\mu$ to a distribution in $\varphi$, $v_\parallel$ and $\mu$, just before and after the 2D advection of the Vlasov solver (transpose algorithm of Section 2.3.2). The second communication is also used to redistribute the data from the $r$, $\theta$ and $\mu$ distribution into a specific distribution (along $\varphi, \mu$ variables) onto a subset of processes within the Field solver. We managed to reduce significantly time dedicated to these communications by evaluating/benchmarking several communications schemes and selecting one among the most non-blocking ones. Also, we suppressed the global MPI collectives whenever possible.

*Weak scaling* A weak scaling test is illustrated in Figs. 23 and 24 using a test case with the following parameters: $N_r = 512$, $N_\theta = 1024$, $N_\varphi = 128$, $N_{v_\parallel} = 128$, $N_\mu$ taking several values from $N_\mu = 2$

to 56. In this test, the ratio of communication over computation remains almost constant for the different number of cores considered for almost all parts of the code, which is an interesting property. However, in the *Field solver* and *Derivatives computation*, the communication costs tends to grow a little bit more than linearly in the number of cores. It explains partly why the relative efficiency of these parts drop a bit along with the number of cores. Another reason is that these regions include a lot of complex communication schemes that involve large amount of transfer compared to computations to be done. All other parts behave quite perfectly in terms of relative efficiency. We end up with a very good relative efficiency of 92% for 458k cores and 1832k threads (the whole machine) compared to 16k cores for the weak scaling test. This experiment has been performed in the framework of a PRACE preparatory access (April 2012 - Nov. 2012), and through a dedicated CPU allocation via our collaboration with Juelich (G8-Exascale NUFUSE project in 2013). This was the largest scaling experiment we ever launched.

Thanks to the new parallel algorithms designed for the Field solver, we avoid the storage of complete 3D field data in memory of each process, but we need only distributed 3D data instead. Each thread inside a MPI process share these reduced amount of 3D data. In addition to this favorable data distribution, another modification has occurred in the present version: a set of different 3D and 2D data buffers have been removed and replaced by a shared large 1D array which is recycled for several uses. Table 7 shows the memory consumption. It can be observed that the memory scalability is greatly improved. At 65k cores, there is more than a factor 2 between the memory consumption of the two versions. The overall memory consumption has also diminished.

**Memory scalability**

| Nb. cores | 4k | 8k | 16k | 33k | 65k |
|---|---|---|---|---|---|
| **Previous version** | | | | | |
| 3D data struct. | 2.79 | 2.75 | 2.73 | 2.72 | 2.72 |
| All data struct. | 11.55 | 8.07 | 6.38 | 5.48 | 5.02 |

| Nb. cores | 4k | 8k | 16k | 33k | 65k |
|---|---|---|---|---|---|
| **Present version** | | | | | |
| 3D data struct. | 1.07 | 0.82 | 0.58 | 0.39 | 0.24 |
| All data struct. | 9.50 | 5.83 | 3.93 | 2.86 | 2.27 |

Table 7: Memory footprint (GB) by node, big case. Left side: previous, Right side: new version

**Conclusion**

We have described the parallelization of a quasi-neutral Poisson solver used into a full-$f$ gyrokinetic 5D simulator, for which the parallel performance of the numerical method has been demonstrated. It achieves a good parallel computation scalability up to 458k cores (Weak scaling) and has a reduced impact on overall execution time on 65k cores (Strong scaling). Several levels of parallelism and a hybrid OpenMP/MPI approach are combined. The coupling of the quasi-neutral solver and the Vlasov code were improved in reducing communication costs and introducing better computation distribution compared to previous approaches [35, 91, 139]. The modifications result also in savings in the memory occupancy, which is a big issue when GYSELA users wish to run very large cases. Further improvements in terms of memory footprint will be shown in the next Section. What we learned throughout the different solutions evaluated for the Poisson solver is that taking all constraints to scale up to 1832k threads require a very fine analysis of the possible numerical schemes, the data flow and all the associated costs: computations, amounts of memory, communications and so on. Also, each time a parallel bottleneck is pushed away, another one appears at a larger scale. This means we have to regularly revise the assumptions and the structure of the code in order to reach better scalability. There is a kind of duty to deal with the complexity of managing readability of the code, parallel efficiency and user support all at once. There is no programming model or framework that can really handle that comprehensively. A set of homemade verification tools and a continuous integration platform have helped us a lot to handle code evolution more smoothly [19], which also reduced the time devoted to debugging.

## 2.5 Improving memory scalability

### 2.5.1 Introduction

The architecture of supercomputers will considerably change in the next decade. Since 2004, CPU frequency does not increase anymore. The problem due to processor's power consumption and heat dissipation forms what is now known as the "Power Wall". Consequently the on-chip parallelism is dramatically increasing to offer more performance. Instead of doubling the clock-speed every 18-24 month, the number of cores per compute node follows the same law. These new parallel architectures are expected to exhibit different Non Uniform Memory Access (NUMA) levels. One trend for a fraction of these machines is to offer less and less memory per core. Amount of memory available per core

has been identified as one of the exascale challenges [181] and is one of our main concerns hereafter. The NVRAM and 3D-stacked DRAM technologies could possibly modify this issue in a near future, but this is not yet clear. In this Section we especially focus on the memory consumption of GYSELA. This is a critical point to simulate ever-growing physical cases while using a constrained amount of available memory.

The goal of the work presented here is to analyze and to reduce the memory footprint of GYSELA to improve its memory scalability. Even if an application scales well and can already use a large amount of cores, the study of the memory consumption becomes critical. In fact, the larger the simulation, the more the cost of structures which do not scale becomes the main limiting factor to handle bigger problems (a kind of Amdhal's law). Brief state of the art is given in [4,83] about possible approaches. It is an important aim because in 2011 we were not able to launch very big case due to the memory footprint of GYSELA per node beeing too large for any existing supercomputers, while the main distributed data structure (distribution function) can fit into memory. The memory overheads associated to parallel management were to high. We present a tool which provides a way to generate memory traces for a parallel application and a visualization of the associated memory allocations/deallocations in off-line mode. Another tool allows us to predict the memory peak depending on input parameters. This is helpful in a production context to check whether the needs of a future simulation fit into available memory. Another topic is to define a methodology and a versatile and portable library to assist the developer to optimize memory usage in scientific parallel applications. In the following, we introduce the method we apply to decrease the memory footprint of GYSELA [4,83]. Then, we briefly describe the dedicated module implemented to generate a trace file of allocation/deallocation process. It also illustrates the visualization and prediction tool capabilities to handle the data of the trace file. The enhanced memory scalability of GYSELA is illustrated.

### 2.5.2 Analysis of memory footprint

During a GYSELA run, each MPI process is associated with a single $\mu$ value and is responsible for a part of the distribution function as a 4D array and the electric field as a 3D array. The remaining of the memory consumption is mostly related to arrays used to store precomputed fields, arrays that store intermediate results for diagnostics, MPI user buffers to concatenate data for MPI send/receive calls and user buffers to store temporary results within OpenMP regions. The biggest arrays are allocated during the initialization of GYSELA and are deallocated at the end of the application.

In order to better understand the memory behavior of GYSELA, we have logged each allocation (allocate statement) by storing the array name, type and size. Using these data we have performed a *strong scaling* study presented in Table 8 (16 threads per MPI process). This study consists in running the application with a large enough mesh, evaluating the memory consumption (for each MPI process) and then increasing step by step the number of MPI processes used for the simulation. Let say we use $x$ Giga Bytes of memory per process. for a given simulation with $n$ processes. In the ideal case, one can expect that the same simulation with $2n$ processes would use $\frac{x}{2}$ Giga Bytes of memory per process. In this case, the memory *scalability* is considered perfect. But in practice, this is generally not the case because of memory overheads that do not scale.

| Number of cores<br>Number of MPI processes | 2k<br>128 | 4k<br>256 | 8k<br>512 | 16k<br>1024 | 33k<br>2048 |
|---|---|---|---|---|---|
| 4D structures | 209.2<br>67.1 % | 107.1<br>59.6 % | 56.5<br>49.5 % | 28.4<br>34.2 % | 14.4<br>21.3 % |
| 3D structures | 62.7<br>20.1 % | 36.0<br>20.0 % | 22.6<br>19.8 % | 19.7<br>23.7 % | 18.3<br>27.1 % |
| 2D structures | 33.1<br>10.6 % | 33.1<br>18.4 % | 33.1<br>28.9 % | 33.1<br>39.9 % | 33.1<br>49.0 % |
| 1D structures | 6.6<br>2.1 % | 3.4<br>1.9 % | 2.0<br>1.7 % | 1.7<br>2.0 % | 1.6<br>2.3 % |
| Total per MPI process in GBytes | 311.6 | 179.6 | 114.2 | 82.9 | 67.4 |

Table 8: Strong scaling: allocation sizes (in GB per MPI process) of data allocated during initialization stage and percentage with respect to total size for each kind of data

Table 8 shows the evolution of the memory consumption along with the number of cores for a single MPI process. The percentage of memory consumption compared with the total memory of the process is given for each type of data structure. The dimensions of the mesh are set to $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. This mesh is bigger than the meshes used in production nowadays, but match further needs, especially those expected for kinetic electron physics.

The last case with 2048 processes requires 67.4 GB of memory per MPI process. We often launch a single MPI process per node, and we can notice that the memory required is largest than, for example the 64 GB of a Bull Helios [12] node or the 16 GB of an IBM BG/Q node. Table 8 also illustrates that 2D structures and many 1D structures do not benefit from parallelization. In fact, the memory cost of the 2D structures does not depend on the number of processes at all, but rather on the mesh size and the number of threads. On the last case with 33k cores, the cost of the 2D structures is the main bottleneck, as it takes 49 % of the whole memory footprint. Finally, we can point out that memory overheads observed in large simulations are due to various reasons. Additional extra memory is needed, for example, to store briefly some coefficients during an interpolation (for the Semi-Lagrangian solver of the Vlasov equation). Several MPI buffers are used whenever the application deals with the sending and the receiving of temporary data. Often, before using MPI subroutines, a copy to reorganize and change the shape of data is required to send or receive them. But these overheads can be reduced. To improve the memory scalability we need to precisely understand the memory consumption of Gysela.

### 2.5.3 A method and tools to reduce the memory footprint

#### Step-by-step reduction of memory footprint

There are at least two ways to reduce the memory footprint of a parallel application. On one hand, we can first increase the number of nodes used for the simulation, since the size of data structures which benefit from a domain decomposition will decrease along with the number of MPI processes. On the other hand, we can manage more finely the array allocations in order to reduce the memory costs that do not scale with the number of threads/MPI processes and to limit the impact of all allocated data at the memory peak. In fact, this approach consists in reducing the impact of memory overhead at the memory peak, and therefore improves the memory scalability of the application.

To achieve the reduction of the memory footprint and to push back the memory bottleneck, we choose to focus on the second approach. In the original version of the Gysela code, most of the variables are allocated during the initialization phase. This procedure is rightful for structures which are for *persistent variables* in opposition to *temporary variables* that can be dynamically allocated. In this configuration, one can determine at the beginning of the simulation the memory space required without actually executing a complete simulation. This allows a user to know if the case can be submitted. Secondly, it avoids execution overheads due to dynamic memory management. But a disadvantage of this approach is that variables used locally in one or two subroutines consume their memory space during the whole execution of the simulation. As the memory space becomes a critical point when a large number of cores are employed, we have allocated a large subset of these variables as temporary variables with dynamic allocation. This has reduced the memory peak with a negligible impact on the execution time. However, one issue with dynamic allocations is the ability to predict the memory space required to run a simulation. Thanks to the prediction tool described hereafter, we recover this advantage while doing dynamic allocations. Furthermore, the new prediction tool is used in combination with the IMPR method. It was applied to reduce the memory peak of Gysela:

1 **Step 0** Choose the target configuration of the application (input parameter set),
2 **Step 1** Locate the moment of the memory peak during the execution,
3 **Step 2** Identify the data structures allocated at the memory peak,
4 **Step 3** Evaluate which one can be deallocated. If none of them can be simply deallocated, reconsider the algorithms used and try to improve them,
5 **Step 4** After adding deallocations or modifying algorithms, go back to "**Step 0**".

**Method 7:** Incremental Memory Peak Reduction - IMPR method

The **Step 0** of this method has a direct impact on the other steps. From the point of view of the memory consumption in Gysela, the crucial parameters are the size of the mesh and the number of MPI processes. Depending on these parameters, the memory peak settles at a specific location. The prediction tool presented later on allows us to take the data from a run of reference and to change them offline to explore different configurations easily.

In **Step 1**, the location of the memory peak means to locate the lines of the source code and the call stack where the maximum memory consumption is reached. In order to identify the allocated data structure in **Step 2**, we need to track the allocations/deallocations of all data structures used during a run which will be done with the dedicated library.

For **Step 3**, the developer has to pay attention to the data dependencies to understand which arrays can potentially be removed whenever the memory peak occurs. There are different ways to diminish memory consumption. You can decide to reuse an existing temporary array, or to do more

[12] http://www.top500.org/system/177449

computation instead of saving a result in memory, or you can notice data duplication that can be avoided, or you can notice that some allocated buffers can be deallocated during a short time slot. If you does not succeed, a way of reducing the memory consumption is to change the algorithm and/or the numerical scheme. In the GYSELA framework, we have mainly target the data structures that penalize the application at large scale (because they do not benefit of domain decomposition).

Once the source code is modified in **Step 4**, one can occasionally observe a shift of the memory peak in time. So, for the next iteration of the method, one can focus on another part of the application. We loop back to the **Step 0** until the footprint is low enough.

Let us notice that in order to use the prediction tool in offline mode, a strong assumption must hold: all memory allocations of the application must be independent of any variables that can vary by a non-deterministic process. In other words, given an input set of parameters, the memory behavior of the application is unique and deterministic (it does not depend on environment or intractable process). This assumption is fulfilled in the GYSELA application. If this assumption is not true for a given application, it would require to perform the different steps of the method without the prediction tool. To follow up the memory consumption of GYSELA and to measure the memory footprint reduction, two different tools have been developed: a FORTRAN module to generate a trace file of allocations/deallocations (part of **MTM** library [83] developed by F. Rozar, PhD) and a visualization + prediction Python script which exploits the trace file. The information retrieved from the execution of GYSELA thanks to the instrumentation module is crucial for this memory analysis.

### Trace File

Various data structures are used in GYSELA and in order to handle their allocations/deallocations, a dedicated FORTRAN module was developed to log them into a file: the *dynamic memory trace*. As in the current implementation the MPI processes have almost the same dynamic memory trace, we analyze a single trace file for the allocations/deallocations of MPI process `0`.

*Overview of generic tools* In the community of performance analysis tools dedicated to parallel applications, different approaches exist. Almost all of them rely on *trace files*. A trace file collects information from the application to represent one aspect of its execution: execution time, number of MPI messages sent, idle time, memory consumption and so on. In order to obtain these information, the application has to be instrumented. The instrumentation can be made at 4 levels: in the source code, at compilation time, at link-time or during execution (just in time). The Scalasca performance tool [135] is able to instrument at compilation time. This approach has the advantage to cover all the code parts of the application and it allows the customization of the retrieved information. This systematic approach gives a full detailed trace but the main purpose of Scalasca is to locate the bottleneck of an application, not to study the memory scalability. Also by using automatic instrumentation, it can be difficult to retrieve the expression of an allocation, like we do in our library. The tool set EZTrace [103] offers the possibility of intercepting calls to a set of functions. This tool can quickly instrument an application thanks to a link with third-party libraries at link-time. The same issue occurs in Scalasca. The Pin [164] tools, DynamoRIO [111] or Maqao [126] use the Just In Time (JIT) approach to instrument the application during the execution. The advantage here is the genericity of the method. Any program can be instrumented this way, but in this approach, retrieving the expression of the size of an allocation is also an issue.

*Implementation* The dedicated tools we have developed allow us to measure the performance of GYSELA, from the memory point of view. These tools permit to deal with the memory scalability issue unlike the previous generic tools. The source code is instrumented thanks to a FORTRAN module to generate a trace file at the execution. A visualization tool has been developed to deal with the provided trace file. It offers a global view of the memory consumption and an accurate view around the memory peak to help the developer identify problems and then reduce the memory footprint. The terminal output of the post processing script gives valuable information about the arrays allocated when the memory peak is reached. Given a trace file, we can also extrapolate the memory consumption as a function of the input parameters. This allows one to investigate the memory scalability. As far as we know, there is no equivalent tool to profile the memory behavior in the HPC community. The instrumentation module offers an interface, *take* and *drop*, which wraps the calls to `allocate` and `deallocate`. For each allocation and deallocation, the module logs the name of the array, its type, its size and the expression of number of elements. The expression is required to make prediction afterwards, during a post-mortem analysis.

### Visualization tool

In order to address the memory consumption issues, we have to identify the parts of the code where the memory usage reaches its peak. The log file can be large, *i.e* several Mega Bytes. A Python tool

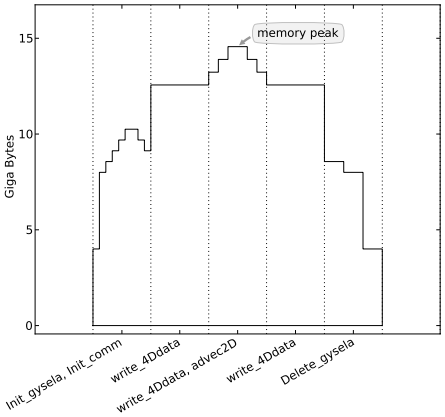



Figure 25: Evolution of the dynamic memory consumption during Gysela execution

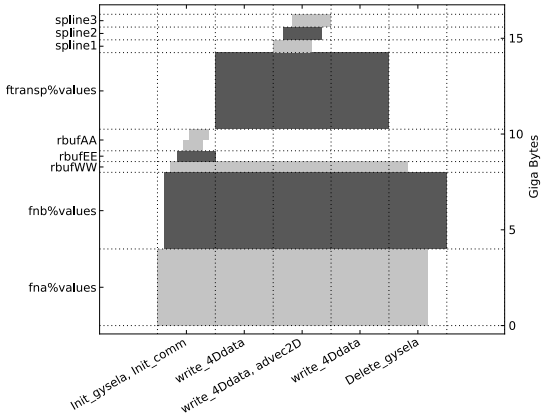



Figure 26: Allocation and deallocation of arrays used in different Gysela subroutines

helps the developer to understand the memory cost of the handled algorithms, and gives him some hints about how and where it is meaningful to decrease the memory footprint. This information is provided through two kinds of plots.

Fig. 25 plots the *dynamic* memory consumption in GB along time. The X axis represents the chronological entry/exit of instrumented subroutines. The Y axis gives memory consumption in GB. The number of subroutines plotted on the graph has been shortened in order to improve the readability. In Fig. 26, the X axis remains identical to the previous Fig., while the Y axis corresponds to the names of the array. Each array is associated with a horizontal line of the picture. The allocation of an array matches a rectangle, filled in dark or light gray color in its corresponding line. The width of rectangles depends on the subroutines where allocation/deallocation happens. In Fig. 25 one can locate in which subroutine the memory peak is reached. in Fig. 26 one can then identify the arrays that are actually allocated when the memory peak is reached.

#### Prediction tool

To anticipate the memory requirements when running a given simulation, we need to predict the memory consumption with any given input parameter set. Thanks to the expressions of array size and the value or expression of numerical parameters contained in the trace file, we can model the memory behavior off-line. The idea here is to reproduce allocations from the trace file with a given input set of parameters. By changing the input parameters, our Python prediction tool precisely establishes the Gysela memory consumption for a MPI process on larger meshes. It also gives the possibility to calibrate the quantity of computational resources required by a given simulation case. The prediction tool allows us to explore machine configurations which do not exist yet, such as the exascale ones.

### 2.5.4 Applying the method on Gysela

#### Achieving memory reduction step-by-step

In order to reduce the memory footprint of Gysela, we follow the IMPR method. We are willing to cut down the memory peak at each iteration of this method. The code changes we consider consist of moving allocations/deallocations of a set of data structures and in changing some algorithms. To track the changes concerning the location of the memory peak and the amount of memory usage after each optimization step, a reference run has been defined that uses 64 MPI processes and 16 threads per process, leading to a total of 1024 cores. In this subsection, we will use for our analysis the maximum of memory consumption given for one MPI process. The mesh of the reference case is set to

$$N_r = 512,\ N_\theta = 1024,\ N_\varphi = 128,\ N_{v_\parallel} = 128,\ N_\mu = 2\,. \tag{2.18}$$

Originally, a large fraction of the data structure allocations in Gysela are done at application startup. These persistent data structures are allocated in the initialization step and deallocated in the exit step. In this original setting, a Gysela specific wrapper logs some of the `allocate` function calls. The name, the size and the dimensions of each allocated data structure are recorded in a trace file during the Gysela initialization. In this persistent allocation approach, these information are added up to give directly the maximum of memory consumption soon after the application launch. However, this is no longer true when we insert the dynamic management of the data structure allocations. Starting with the persistent allocation trace file, we obtain Table 8. It appears that 1D and 2D structures represent a large amount of the memory usage for a large run. With this original version,

GYSELA exhibits a maximum memory consumption of 14.14 GB. In the following, we will observe and reduce this peak memory consumption focusing on the reference case.

Let us consider a first optimization looking at data structures allocated that appear in trace files. Several 1D and 2D data structures that store intermediate computations take most of the memory space. We remove the initial allocation for these data and bring the allocation/deallocation combination closer to the usage of these data structures. The aim is to move the allocations of these data structures as close as possible to their utilization. We keep in mind that frequent calls to allocation/deallocation primitives can lead to execution time overheads. We introduced some dynamic management of allocations for 1D and 2D data structures avoiding most inner loops region. The maximum memory consumption is then 13.53 GB.

Subsequently, we investigate a second memory optimization looking at data structures allocated at the memory peak. We notice that some 2D arrays which store the spline coefficients are allocated during the whole run while they are used only during the Vlasov solving step (the main computation kernel). In the same way than the previous optimization, we move the allocations/deallocations of the spline coefficients closer to their effective use, in order to decrease the memory peak (not located in the Vlasov solver). This modification reduces the memory peak down to 11.89 GB.

In the third phase of IMPR Method, we again list the data structures allocated at the memory peak. In this list, some 2D arrays store matrices that are used in the subroutines that compute the gyroaverage operator. Those matrices store precomputed values to accelerate the computation of the gyroaverage operator (precomputations are done at each GYSELA startup). Two changes are performed: moving the allocations/deallocations of those 2D arrays near their usage and setting these arrays to their correct values just before use. Although the dynamic management of allocation and the setting of these structures introduce a small overhead in execution time, this allows us to decrease the memory peak to 11.63 GB.

After these successive improvements, we have reduced the memory peak by 17% on the reference case. As our goal is to deal with larger meshes, we expect to provide a way to reach a better reduction. The next step described in the following subsection will highlight part of the data dependencies of GYSELA. We will identify an array that can be deallocated during a slot time to shift the memory peak location.

#### Further memory optimizations using data dependencies

We again consider the IMPR Method to alleviate memory usage. The visualization tool generates the pictures of Fig. 27. These pictures are intentionally simplified to improve readability. The curves (upper part) that give memory consumption along with the time help the developer to locate the memory peak in the `advec2D` subroutine (a part of the Vlasov solver). In the lower part of Fig. 27, the biggest allocated data structures are shown. One can wonder which of these data can be deallocated.

After an accurate analysis of the `advec2D` subroutine, we notice that inside the 2D advection step, a transposition of the main unknown (4D distribution function) is done to achieve the computation of splines coefficients on the poloidal plane. By the way, we noticed that at the memory peak, the transposed data structure (named `ftransp\%values`) and the original distribution function (named `fnb\%values`) contain the same information stored into two different containers. Then we manage to deallocate `fnb\%values` temporarily during the memory peak. No extra computations is required here, but the readability of the code is altered a little bit with this change. We then obtain the outputs of Fig. 28. Practically, the memory peak is reduced to 10.66 GB. This improvement is characterized by a move of the memory peak. If one compares Figs. 27 and 28, it is noticeable that the memory peak is located in a different subroutine.

#### Memory scalability

The Table 9 details the strong scaling test using the latest stable release-v5.0 of GYSELA that includes the new dynamic allocation scheme and the algorithmic improvements. The prediction tool allows us to reproduce the Table 8 on the initially targeted mesh $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. Table 9 shows the memory consumption at the memory peak. It is obtained by keeping the mesh size constant and changing the number of MPI processes. For this strong scaling study in the GYSELA framework, we have to modify the domain decomposition setting in the $r$ and $\theta$ directions to increase the number of MPI processes. Our prediction script replays the allocations/deallocations of the trace file by recomputing the size of each array considering the new domain decomposition. The numbers given in Table 9 are obtained thanks to the prediction tool and moreover, they were checked with different test cases such as the bigger one that uses 33k cores.

As one can see on the biggest case (33k cores), the consumption of the 2D structures is reduced to 20.8%. Also the memory gain on this case is of 50.8% on the global consumption compared to

Table 8. The 4D structures contain the most relevant data used during the computation and they consume most of the memory as they should. The memory overheads have been globally reduced, which improves the memory scalability of GYSELA and allows us to run larger simulations.

Based on the feedback of different runs, the tools integrated in GYSELA and explained here do not introduce too much execution time overheads. Nevertheless, the origin of these overheads can be identified as, firstly the cost of input/output activities on disk, and secondly the more frequent calls to the `allocate` subroutine. The study and reduction of overheads due to dynamic allocators will be the purpose of future works.

**Using prediction tool for production runs**

Before launching a simulation on a supercomputer, it is helpful to know how to check the set of important parameters for a given parallel application. Indeed, the waiting time in queue before the job can start may be quite long (several days) when a lot of computing resources are requested. Therefore, one may wonder if the application will start with well-designed parameters and if it will exceed the available memory. With the prediction tool, it is possible to evaluate for a given mesh size and an input parameter file if a GYSELA simulation can be run (*i.e* fit into the available memory) or not. To illustrate this point, we have made a larger reference run on the Juqueen machine and saved the associated memory trace file. The Juqueen machine is an IBM BlueGene/Q with 16 GB of memory per node. Moreover to be able to exploit most of the computational capacity of this machine, we commonly use 64 threads per node (16 cores).

In this large reference run, we use 256 MPI processes and 64 OpenMP threads (4k cores). Usually a GYSELA run is set such that a single MPI process is launched per node. We also choose this setting for the other tests described in this subsection. The mesh for this run is given by

$$N_r = 256,\ N_\theta = 256,\ N_\varphi = 128,\ N_{v_\parallel} = 64,\ N_\mu = 4\,. \tag{2.19}$$

Using the trace file of this run, the behavior of the memory consumption of GYSELA can be

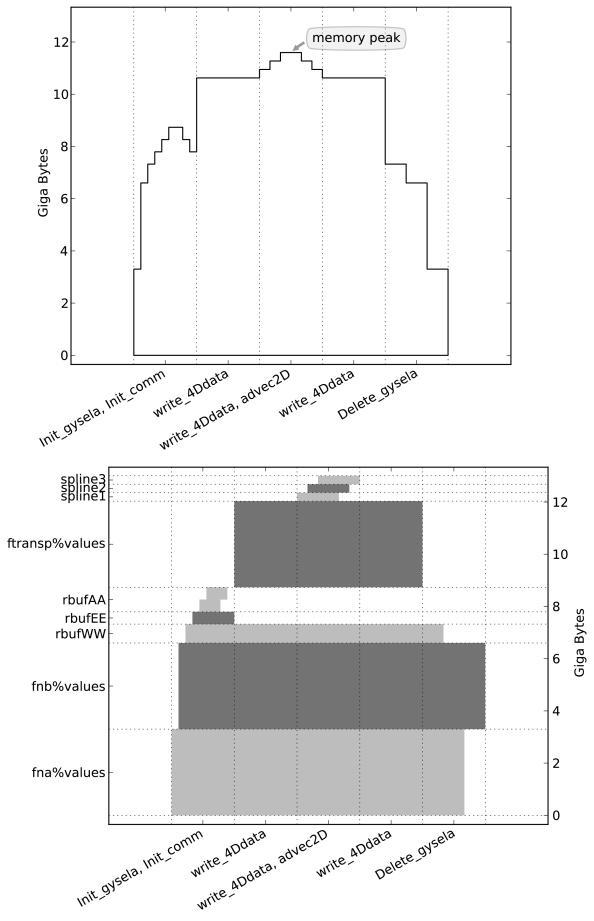



Figure 27: Original run on 1024 cores, visualization of the memory trace

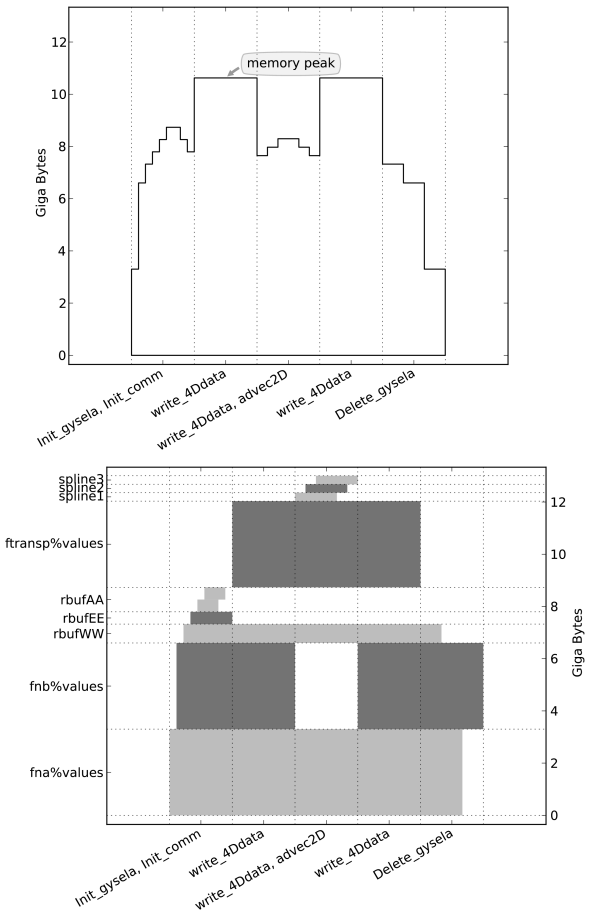
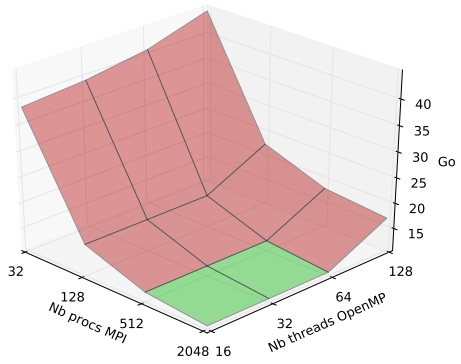



Figure 28: Run performed after improvement, visualization of the memory trace

| Number of cores<br>Number of MPI processes | 2k<br>128 | 4k<br>256 | 8k<br>512 | 16k<br>1024 | 33k<br>2048 |
|---|---|---|---|---|---|
| 4D structures | 207.2<br>79.2% | 104.4<br>71.5% | 53.7<br>65.6% | 27.3<br>52.2% | 14.4<br>42.0% |
| 3D structures | 42.0<br>16.1% | 31.1<br>21.3% | 18.6<br>22.7% | 15.9<br>30.4% | 11.0<br>32.1% |
| 2D structures | 7.1<br>2.7% | 7.1<br>4.9% | 7.1<br>8.7% | 7.1<br>13.6% | 7.1<br>20.8% |
| 1D structures | 5.2<br>2.0% | 3.3<br>2.3% | 2.4<br>3.0% | 2.0<br>3.8% | 1.7<br>5.1% |
| Total per MPI process in GBytes | 261.5 | 145.9 | 81.8 | 52.3 | 34.2 |

Table 9: Strong scaling: memory allocation size and percentage with respect to the total for each kind of data at the memory peak




| MPI procs \ OpenMP threads | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| 32 | **38.47** | **39.36** | **41.12** | **44.64** |
| 128 | **16.78** | **16.79** | **16.80** | **22.60** |
| 512 | **12.51** | **12.52** | **12.53** | **18.26** |
| 2048 | **11.24** | **11.24** | **11.26** | **17.07** |

Figure 29: The memory peak (GB) along with nb of MPI procs and nb of OpenMP threads

predicted. We would like to find out a list of possible deployments able to handle the following target mesh:

$$N_r = 1024,\ N_\theta = 1024,\ N_\varphi = 128,\ N_{v_\parallel} = 64,\ N_\mu = 4\,. \tag{2.20}$$

We change the parameters of the mesh directly in the trace file and also the number of MPI processes and the number of OpenMP threads. Scanning a set of values, we output the memory peak per MPI process with the prediction tool. The obtained results are plotted on Fig. 2.5.4. With the 16 GB available on a Juqueen node, one can observe that beyond 512 processes and below 64 threads, the Gysela code fit into the memory and can handle the targeted mesh.

Some data structure used by the application for which the parallelism is induced by the domain decomposition of the mesh can take advantage of this decomposition. This means that the cost of this kind of structure decreases as the number of process increases. On Fig. 2.5.4, one can notice that the behavior of the memory consumption along the MPI process axis is consistent with the expected behavior. While the application uses a larger number of processes, the memory costs tend to diminish. But, a saturation effect arises between 128 and 2048 MPI processes. This is due to some arrays that do not scale along with the number of subdomains in $(r, \theta)$. After an analysis of the memory consumption along time, it appears that the memory peak is not located in the same subroutine for both cases for 64 and 128 threads.

### 2.5.5 Conclusion

The work described in this Section focuses on a memory modeling and tracing module and some post processing tools which enable us to improve greatly the memory scalability of Gysela. With this framework, the understanding of the memory footprint behavior along time is achievable. Also, the generated trace file can be reused to extrapolate the memory consumption for different input sets of parameters in offline mode; this aspect is important both for end-users who need greater resolutions or features with greedy memory needs, and for developers to design algorithms for exascale machine.

With these tools, a reduction of 50.8% of the memory peak has been achieved and the memory scalability of the Gysela has been improved. The designed set of tools provides the post processing scripts to visualize the trace file and to perform predictions. To use the prediction feature, the pattern of the memory footprint must depend only on an input set of parameters, not on computed values during the execution. However the visualization script does not require this assumption. As our method leads the developer to allocate his data structures dynamically, studying the different

allocators in the user space to optimize the execution time overhead would be desirable to minimize data movements and reuse cache memory if possible.

## 2.6 Gyroaverage operator

The current gyroaverage implementation used in GYSELA has been improved in 2015-2016, enhancing the precision of the operator thanks to Hermite interpolation [5,17,83] instead of Padé approximation (quick description already given p.30, the work has been done in collaboration with V. Grandgirard, M. Mehrenberger, F. Rozar, C. Steiner). One main advantage of this approach is to avoid going back and forth in the Fourier space like it was the case for Padé approximation. Thereupon this method is much more local in space (one does not anymore all points allong $\theta$ direction locally), which is beneficial for parallelization purposes. In the present Section, a new parallelization scheme for the gyroaverage operator is described [15, 80] (cooperative work with N. Bouzat, M. Mehrenberger, J. Roman). It mainly avoids costly transpositions of the full 5D function using halo exchange instead. Though the computational cost remains the same, the communication cost is much smaller. The overall algorithm is also improved by cleverly interleaving communications and computations, thus allowing for a reduction of communication costs and a more efficient thread parallelization. The execution time of the gyroaverage is up to twice as fast as before. The benefit of an improved scheme providing the overlap of communications by computations is also shown.

### 2.6.1 Initial setting

#### Context

The gyroaverage operator mimics the cyclotronic motion projected in the $r\times\theta$ poloidal plane. The gyroaverage operator represents a costly part of two main components: within the *diagnostics* to get the gyroaverage on the whole distribution function, and in the *field solver* – despite the speedup is expected to be smaller in this second component. In the *diagnostics*, the data distribution is such that the $r$ and $\theta$ dimensions are distributed among the MPI processes and the $\varphi$ and $v_\parallel$ dimensions are entirely contained in each process. Thus, each process knows the value of a block $f(r=local, \theta=local, \varphi=*, v_{//}=*, \mu=value)$. In the previous implementation of the gyroaverage operator, a Padé approximation method was used. It requires the whole $r\times\theta$ plane to be stored locally in memory because of Fourier transform along the $\theta$ direction and finite differences along $r$. Therefore, it was necessary to transpose the whole distribution function between MPI processes before and after the computation of the gyroaverage. The method based on Hermite interpolation does not have this strong constraint, as gyroaverage can be computed only with values close to the target point, which permits us to avoid the costly transpositions. Moreover it does not damp the small variations of the function as the Padé approximation did for large $k_\perp\rho$, accuracy is enhanced [83, p. 93].

The use of the Fourier transform reduces the gyroaveraging operation by a multiplication in the Fourier space by the Bessel function. Good approximations of the Bessel function have been proposed such as the widely used *Padé* expansion. The Padé approximation enables to recover a good approximation for small radii [116]. However, for a larger Larmor radius and ordinary wave-numbers, the Padé approximation truncates the oscillations of the Bessel function, over-damps the small scales, and then introduces bias. Also the use of Fourier transform is not applicable in general geometry as we would like to employ in a near future; therefore it can not be employed in realistic tokamak equilibrium. These two limitations are overcome using an interpolation technique on the gyro-circles, which is the background of this study [5, 17].

#### Description of Hermite based gyro average

We will now describe the numerical framework and approximations that are made for the gyroaverage operator. Let us consider a Larmor radius $\rho$, a grid in polar coordinates $r\times\theta$ (poloidal plane) and a function $f$ defined over this grid. The gyroaverage operator $\mathcal{J}_\rho$ consists, for each point $P$ of the plane, in a weighted integral of the value of $f$ over the circle of radius $\rho$ and center $P$. In a discrete space, this translates as a mean of $N_\mathcal{L}$ points uniformly distributed on the circle of Larmor and interpolated with the Hermite method. The precision of the operator directly depends on $N_\mathcal{L}$. An example is shown in Fig. 30 for $N_\mathcal{L}=5$. To compute the gyroaverage at the point ●, $N_\mathcal{L}$ points ▲ are placed on the circle of radius $\rho=\sqrt{2\mu}$. As these points are unlikely to coincide with a mesh point, an Hermite interpolation is performed for each of them using the values at the four corners ■ of the cell in which

they are contained. Thus the gyroaveraged value of $f$ at a point $(r_i, \theta_j)$ of the grid writes

$$\mathcal{J}_\rho(f)(r_i,\theta_j) \simeq \frac{1}{N_\mathcal{L}} \sum_{k=1}^{N_\mathcal{L}} \mathcal{H}(f)(x_k, y_k) \tag{2.21}$$

with $\mathcal{H}$ being the Hermite interpolation function, $x_k = r_i cos(\theta_j) + \rho cos(\theta_j + k\frac{2\pi}{N_\mathcal{L}})$ and $y_k = r_i sin(\theta_j) + \rho sin(\theta_j + k\frac{2\pi}{N_\mathcal{L}})$ being the coordinates of the points on the Larmor circle in Cartesian coordinates. When one of these points is outside the mesh, a radial projection is done on the closer border (inner or outer). Given one of the $N_\mathcal{L}$ points ▲ of coordinate $(\tilde{r}, \tilde{\theta})$, the computation of the interpolation $\mathcal{H}$ requires the value, the derivatives in $r$ and $\theta$ and the cross-derivatives of each of the corners of the containing cell. The 2D interpolation behaves as if two 1D interpolations along $r$ and $\theta$ were performed. First, an interpolation along $r$ is performed from the two corners of each $\theta$ side of the cell, $\theta_1$ and $\theta_1 + 1$, to the points of same respective $\theta$ coordinate and same $r$ coordinate as the target point ($r = \tilde{r}$). Then a second interpolation along $\theta$ is performed from these two new points to reach the target point $(\tilde{r}, \tilde{\theta})$. The coefficients used for the Hermite interpolation are detailed in [5, 17].

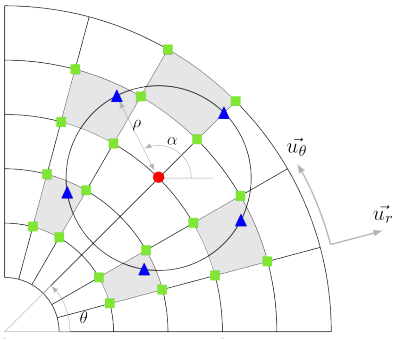



Figure 30: Computation of the gyroaverage.

**Gysela implementation**

We will now go through the implementation of the numerical scheme. For a given point, the computation of the gyroaverage requires a certain set of points which coefficients in the Hermite interpolation are stored in the matrix $M_{coef}$. Each line $i$ of the matrix corresponds to the coefficients of the value and three derivatives of all the points used in the interpolation of the point ($r$=$r_i, \theta$=0). These coefficients are also valid for every $\theta$-value thanks to radial symmetry. Furthermore, $M_{coef}$ is sparse as for each $r_i$, only a few cells (gray cells in Fig. 30) along the Larmor circle center in ($r$=$r_i, \theta$=0) are involved in the gyroaverage. Their number depends on $N_\mathcal{L}$, $\rho = \sqrt{2\mu}$ and $r$.

In the actual implementation, $M_{coef}$ is represented as an array of dimension $N_r$ where each cell contains: an array of indexes indicating which point is involved in the interpolation for the given $r$ and of an array of corresponding coefficients. The size of $M_{coef}$ is then at most of $36 N_r N_\mathcal{L}$ elements [80]. Gysela simulations usually use $N_\mathcal{L} = 8$. Once the initialization is done and $M_{coef}$ is computed, the gyroaverage operator is called several times during each time step on the whole poloidal plane. A matrix $M_{fval}$ is built at the beginning of a call by computing the derivatives at each point of the plane registered in $M_{coef}$. In short, the number of operations to compute the gyroaverage of one point depends, in most cases and for a given $r$ value, on the number of different points involved in the interpolations of the $N_\mathcal{L}$ points on the Larmor circle (see [80]). The cost of gyroaveraging one poloidal plane is then $\Theta(N_\mathcal{L} N_r N_\theta)$, so the total cost for the full 5D distribution function is $\Theta(N_\mathcal{L} N_r N_\theta N_\varphi N_{v_\parallel} N_\mu)$.

In Gysela the distribution of data is twofold. A MPI process $P_{i,j}$, located on the $i$-th $r$ row and $j$-th $\theta$ column of MPI processes, either has data $\mathbf{D}_1(r = *, \theta = *, \varphi = \varphi_i^\tau \to \varphi_{i+1}^\tau, v_\parallel = v_j^\tau \to v_{j+1}^\tau, \mu = \mu_{i,j}^\tau)$ or $\mathbf{D}_2(r = r_i^\tau \to r_{i+1}^\tau, \theta = \theta_j^\tau \to \theta_{j+1}^\tau, \varphi = *, v_\parallel = *, \mu = \mu_{i,j}^\tau)$ where $r_i^\tau = i \times (N_r / N_{\texttt{procr}})$ and similarly for $\theta_j$, $\varphi_i$ and $v_j$. $N_{\texttt{procr}}$ is the number of MPI processes in the radial direction. Several processes share the same $\mu_{i,j}^\tau$. The gyroaverage, as implemented in previous version, requires the full poloidal plane in local memory, *i.e.* distribution $\mathbf{D}_1$. However the data distribution when the operator is called is $\mathbf{D}_2$. It thus requires a costly transposition of the full distribution function before the gyroaverage and after as the computation performed subsequently requires the $\mathbf{D}_1$ distribution. This is the reason why a gyroaverage operator which can handle directly $\mathbf{D}_2$ distribution and avoiding transpose can drastically reduce the volume of communication. It also gives possible solutions for the future where Gysela would possibly consider only $\mathbf{D}_2$ and never $\mathbf{D}_1$. In addition, this technique can be well combined with non-uniform mesh (that we target for the future) and Lagrange interpolants that will be presented in Section 3.3.4.

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

**Algo. 8:** Hermite gyroaverage in *diagnostics*

Algo. 8 shows how the basic Hermite interpolation gyroaverage based on transpositions was integrated in GYSELA into the diagnostic before modification. $N_{l\varphi}$ and $N_{lv_\parallel}$ are the dimensions of the local subdomain in $\varphi$ and $v_\parallel$. $f_{tmp}$ exactly corresponds to $f$ but using distribution $\mathbf{D}_1$. During the preprocessing step 5, the function to be gyroaveraged is built from the 5D distribution function. The same goes for the post-process step 7 where some macro-data are gathered on the gyroaveraged function (*e.g* fluid momentum, velocity integrals over $v_\parallel$ and $\mu$).

### 2.6.2 Parallelization with Halo Exchange

This section details the new solution for the gyroaverage operator using Hermite interpolation. The algorithm has been changed to fit data distribution $\mathbf{D}_2$, and some optimizations have been done for the parallelization. The numerical analysis and core computations remain the same as before while communication schemes are improved in *diagnostics*.

#### Halo and ghost points

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

Indeed, the computation of the gyroaverage for points on the border of a $r \times \theta$ patch requires a certain number of values from other MPI processes depending on the Larmor radius and on the discretization of the mesh. Considering Fig. 31, the ● point of process 2 requires values and derivatives from points ■ located on the processes 1, 3 and 4. ■ point also requires neighboring points to compute their derivatives as seen in 2.6.1. The number of exchanged points also depends on the $r$ coordinate of the point to be gyroaveraged. The closer it is to the inner circle of the plane, the narrower the meshing in $\theta$ becomes and so the more cells the Larmor circle is likely to intercept. The more cells the Larmor circle intercepts, the larger the halo will be and thus the communication. The computation still mainly depends on $N_{\mathcal{L}}$ but now also loosely relies on the Larmor radius.

The halo consists of all the points located on neighbor processes (ghost points) needed by a process to be able to apply the gyroaverage on its local subdomain. Its size must be as small as possible so that its communication cost would be smaller than the cost of a full transposition. Otherwise the main gain we target would be lost. For each specific subdomain, the number of ghost points in dimension $r$ is $N_{ghost\_r}$, and $N_{ghost\_\theta}$ in dimension $\theta$, $N_{deriv}$ is the number of points used to compute the derivatives (here two). The formula for these quantities and the size of the halo are given in [15,80] (illustration is given Fig. 32).

It is worth noticing that the current implementation does not take into account cases with very small $r_{min}$, or where the poloidal plane is divided between a large number of processes in $\theta$. In these cases the Larmor circle of a corner of a patch intercepts cells further away than neighboring subdomains. Larmor circle intercepts too many subdomains for small $r$ values as the mesh narrows along $\theta$ near the center. This limitation should be handled soon by decimating points near $r = 0$ to diminish the cost of communications at this location.

#### Block communication and OpenMP parallelization

Several optimizations for communications and for the parallelization of the computation can be done. Let us recall that for a process $P_{i,j}$, we have the data distribution $\mathbf{D}_2(r = r_i^\tau \rightarrow r_{i+1}^\tau, \theta = \theta_j^\tau \rightarrow \theta_{j+1}^\tau, \varphi = *, v_\parallel = *, \mu = \mu_{i,j}^\tau)$; in our setting it means that every process has to compute the gyroaverage for $N_\varphi \times N_{v_\parallel}$ patches.

Until now, all the communications were performed during the transposition steps (see Algo. 8), before and after the computation. The same could be done by exchanging the halos for every poloidal plane beforehand, but GYSELA consumes much memory and the gyroaverage operator is called within the part of the code where the memory peak is reached. Then, it is preferable to exchange the halos only when the corresponding planes are about to be processed. However, to reduce the initialization

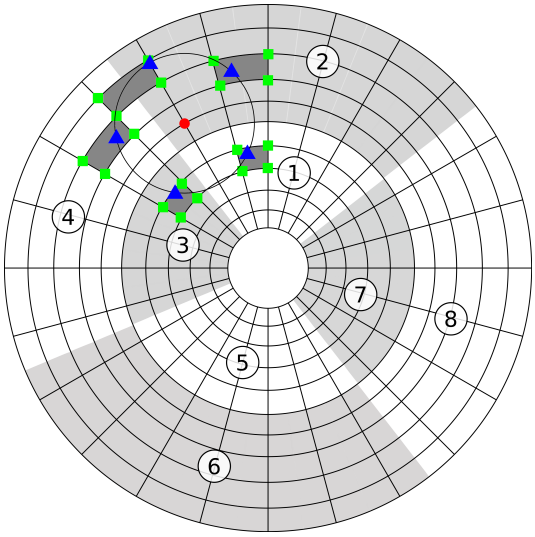



Figure 31: Data distribution over MPI processes and gyroaverage example.

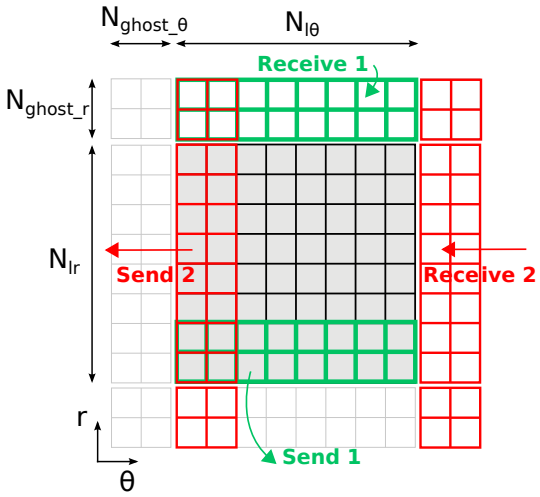



Figure 32: Subdomain representation (local points are in gray) and halo communication scheme.

costs of each communication, it is also interesting to perform them by grouping the halos of several poloidal planes together. Thus, once the communication has ended, several planes (a block) are ready to be gyroaveraged and they can be computed in parallel in a multi-threaded loop. The size of a block of halos can be tailored so that it is a trade-off between memory footprint, thread parallelization, communication performance and readibility of the code. Implementation is detailed in Algo. 9.

During the initialization step 6, the local subdomains of the current block are copied in the temporary array $f_{block}$ which is big enough to store the local subdomains plus their halos received during the communication step 7. Step 11 performs the backward operation, retrieving the gyroaveraged local subdomains and storing them back into the function $f$ as well as gathering data *via* the post-process for the diagnostic. For a given $i_{block}$, the $v_i$ and $\varphi_i$ refers to the $v_\parallel$ and $\varphi$ coordinates of the planes composing the block according to the formula

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

where $bs$ is the size of a block. We will denote $N_{block} = \frac{N_\varphi N_{v_\parallel}}{bs}$. In step 10, the gyroaverage operator used is the same as the one described in Section 2.6.1, but it is applied to a function whose size corresponds to the local subdomain size plus the halo. In our implementation, we ensure that each plane of the block is initialized by the thread which gyroaverages it in order to maximize memory locality and affinity.

**Data:** Distribution function $f$, block size $bs$, $N_{ghost\_r}$, $N_{ghost\_\theta}$, $N_{lr}$ and $N_{l\theta}$
**Result:** Gyroaveraged distribution function $\mathcal{J}_0.f$
1 **begin**
2 Declare $f_{block}$ = `array`$(bs,\ N_{lr} + 2N_{ghost\_r},\ N_{l\theta} + 2N_{ghost\_\theta})$
3 **for** $i_{block} : 0 \rightarrow \frac{N_\varphi N_{v_\parallel}}{bs} - 1$ **do**
4 **OpenMP parallel processing**
5 **for** $i : 0 \rightarrow bs - 1$ **do**
6 $f_{block}(i)$ = `preprocess`$(f(r_{min} \rightarrow r_{max}, \theta_{min} \rightarrow \theta_{max}, \varphi_i, v_i, \mu))$
7 `send_receive_halo`$(f_{block})$
8 **OpenMP parallel processing**
9 **for** $i : 1 \rightarrow bs$ **do**
10 `gyroaverage`$(f_{block}(i))$
11 $f(r_{min} \rightarrow r_{max}, \theta_{min} \rightarrow \theta_{max}, \varphi_i, v_i, \mu)$ = `postprocess`$(f_{block}(i))$

**Algo. 9:** Halo based gyroaverage, processed by block

Concerning the communication step 7, one process exchanges data with the processes which are before and after it in $r$ (down and up), with those before and after him in $\theta$ (left and right) and finally with the four other neighboring processes "in the corners". The number of communications can be reduced from eight to four by avoiding the corners using the scheme pictured at Fig. 32. For readability, only the $r$ down and $\theta$ left phases have been pictured. First, each process sends and receives the requested data to its neighbors in $r$ (in green), second it sends and receives its data in $\theta$ plus some of those received during the previous step (in red). Thus the communications with the processes located in the corners are avoided at the cost of a synchronization in the middle of the communication phase. The communications are carried out using the `MPI_Sendrecv()` routine to send

the halos for all the planes of a block in one step. A communication scheme using non-blocking MPI routines has also been evaluated, but proves to be less performing in the benchmark we conducted. In the end, the new implementation is expected to be faster as the communication costs are reduced compared to the original version based on transposition though the computational cost is a bit higher. The memory footprint is also reduced as the function distribution was previously fully duplicated, whereas now, we only need one buffer $f_{block}(i)$ with relatively small size.

#### Performance results

In the following, the performance of the new solution, described in Section 2.6.2, is compared to the one of the original approach of Section 2.6.1. The simulations presented here were performed (2016) on the Poincaré cluster located at Maison de la Simulation, France. Nodes are composed of two Intel(R) Xeon(R) E5-2670 with 8 cores and 32GB of shared memory each. In this study, we consider $N_{\mathcal{L}}$ constant and equal to 8 as it is the value which is used in usual production runs.

The free parameter of the new parallel algorithm is the block size, *i.e.* the number of poloidal planes for which halo exchange is performed in one communication and for which computation is parallelized at thread level. It is interesting to study the block size which allows to achieve the best performance, it has been done [80]. Fig. 33 shows the execution times of the gyroaverage operator for the previous version and for the new one. The mesh size is $(1024 \times 1024 \times 64 \times 32 \times 1)$ with $\mu = 4.0$ and a block size equal to 128. The number of cores is changed by increasing alternatively the number of MPI processes along $r$ and $\theta$. The number of threads per process is constant and equal to 8. The halo based version is almost twice as fast compared to original transposition based version.

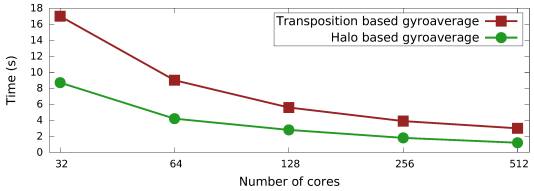



Figure 33: Strong scaling and comparison of execution time of the Hermite gyroaverage based on transposition and the gyroaverage based on halo exchange within *diagnostics*.

Benchmarks tend to show an equal amount of communication and computation time for large numbers of cores. Thus, it would be interesting to be able to absorb these communication costs by performing them concurrently with the computation.

### 2.6.3 Overlapping communication with computation

This section details how the performance of the gyroaverage is improved by overlapping communications and computations in the operator. It is performed through finer grain parallelization, one thread doing communication, while the other threads are computing.

#### Algorithm, complexity and expected speedup

As seen in Section 11, large production simulations usually show communication times and computation times which are relatively close one to each other. It is then possible to further improve the algorithm and decrease execution time by performing communications and computations simultaneously. Similar and more complete analyzes are performed in [119] and solutions for a good calibration of the parameters of the overlapping are suggested.

Using the improved blocked version of the Hermite gyroaverage (Section 2.6.2), the idea is to start the initialization and communication of the next block while performing the computation of the current one. The different steps are detailed in Algo. 10. At line 3 the different blocks to be processed are built. Lines 14 and 15 consists in the initialization and communication of the current data block. Lines 19 and 20 represents the computation and post-processing of the previous data block. There is one more iteration in the $i_{block}$ loop than the total number of blocks so that the first iteration only performs the communication for the first block to initialize the macro-pipeline.

**Data:** Distribution function $f$, block size $bs$
**Result:** Gyroaveraged distribution function $\mathcal{J}_0.f$

```
1  begin
2      L_{φv_∥} = {∅}
3      for i_block : 0 → (N_φ N_{v_∥})/bs − 1 do
4          L_tmp = {∅}
5          for i : 0 → bs − 1 do
6              v_i = modulo(i_block × bs + i, N_φ)
7              φ_i = (i_block × bs + i) ÷ N_φ
8              L_tmp = L_tmp ⊕ f(r_min → r_max, θ_min → θ_max, φ_i, v_i, μ)
9          L_{φv_∥}(i_block) = L_tmp
10     for i_block : 0 → (N_φ N_{v_∥})/bs do
11         OpenMP parallel processing
12             if i_block ≠ (N_φ N_{v_∥})/bs then
13                 OpenMP thread 1
14                     preprocess(L_{φv_∥}(i_block))
15                     async_send_receive_halo(L_{φv_∥}(i_block))
16             if i_block ≠ 0 then
17                 wait_comm(L_{φv_∥}(i_block − 1))
18                 OpenMP threads 2 → n
19                     gyroaverage(L_{φv_∥}(i_block − 1))
20                     postprocess(L_{φv_∥}(i_block − 1))
21
```

**Algo. 10:** Patched gyroaverage with overlapping

There are three ways algorithms with overlap can behave according to the relative size of the different execution times. Either the computation of a block and the communication of a block have the same duration (a), or communication is longer than computation (b), or computation is longer than communication (c). We consider preprocess and postprocess times to be included in the communication and computation times, respectively, assuming they are negligible. An excellent behavior is obtained for cases where communication and computation have the same execution times (a). In this case the global execution time can be ideally decreased by almost a factor 2 with overlapping.

### Implementation

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

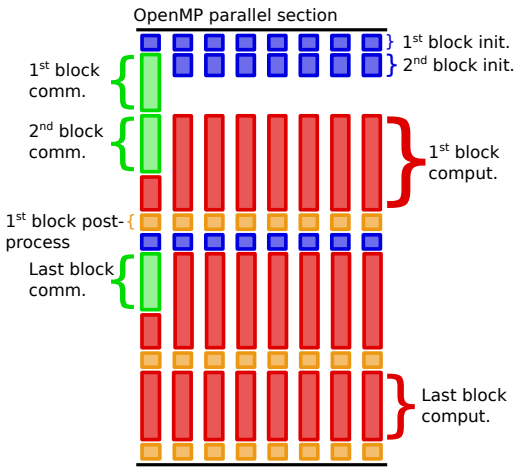



Figure 34: Behavior of the OpenMP threads (thread number in abcissa, time in ordinate) in the main loop for the algorithm with overlapping

| Versions | $\mu$ values | |
|---|---|---|
| | 2.6667 | 8. |
| Transp. version | 81.74s | 81.14s |
| Halo version | 36.91s | 51.75s |
| Overlap version | 32.66s | 44.40s |

Table 10: Total execution time of the 9 calls to the diagnostic on a typical production run with two $\mu$ values

To get the expected overlapping behavior and design a portable approach, the communications are performed by a dedicated thread (usually called master thread in OpenMP terminology). Moreover the loop scheduling of OpenMP has been set to dynamic, *i.e.* once a thread is done with its assigned loop iteration, it requests others to the scheduler. This way, no index of the parallelized loop is

assigned beforehand to the master thread so the computation can be performed entirely even if the communications are longer. And in the case where the communications are shorter, this scheduling allows the master thread to join the computation loop once it has performed the communications. However, this could lead to a loss of cache locality as one plane is no longer pre-processed, computed and post-processed by the same thread. Indeed, OpenMP static scheduling ensures that a thread is assigned the same loop indexes in any loop which has the same first and last index, which is not the case for the dynamic scheduling. Fig. 34 pictures the scheduling of the work load between the threads in the case where the communication time for a block is shorter than computation time.

**Performance results**

In the following, the performance of the solution with overlapping, is compared to that presented in Sections 2.6.2 and 2.6.1. The dimensions of the mesh used are $(1024 \times 1024 \times 64 \times 32 \times 1)$. The performance detailed in this section were performed on the Helios cluster located at the IFERC center in Rokkasho, Japan which has an architecture very close to that of the Poincaré machine (Intel(R) Xeon(R) E5-2680 instead of E5-2680).

Tab. 11 shows how the version with overlapping of the gyroaverage scales with the number of cores and is compared to the halo version. The algorithm with overlapping is faster than the halo algorithm and scales in a similar way. The speedup results behave as expected in previous sections: the time gained over the halo version when the optimal block size of 64 (few blocks) is used is around 10%, and it reaches 100% with numerous small blocks of size 4. Moreover, one can see that the performance of the version with overlapping seems to be much less dependent on the block size. It is a big benefit to use the same size for any value of $\mu$ (see Section 11) without having to seek for optimal block size.

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

Table 11: Scaling of the execution time of the gyroaverage operator for the halo and version with overlapping with different block sizes (between parenthesis).

Evaluating the efficiency of the improvements detailed in this paper on an overall GYSELA execution requires a test case with several $\mu$ values, similar to usual production runs. Table 10 shows the total time spent in the diagnostic in which the halo and overlap gyroaverage algorithms have been implemented. The execution covers 24 time steps, and the diagnostic is performed every 3 time steps. The improvement achieved by the halo version is great but highly depending on the value of $\mu$. Nonetheless, $\mu$ never takes a value above 16 in production runs and the halo version still demonstrates improvements at this point. The performance gain achieved by the new implementations is effective, as the total execution time of diagnostic take 5.7% of the total time in the transposition's version, down to 2.7% in the halo version and down to 2.4% in the version with overlapping.

The gain in terms of memory is also significant. Indeed, in the transposition's version, the transpositions were performed on a copy of the distribution function, thus making the memory footprint of same magnitude as the size of the function to be gyroaveraged. With the halo version, the memory cost of the operator is only the amound needed by one block of poloidal plane; given $N_P$ MPI processes, this represents one $N_P$-th of the size of the function to be gyroaveraged. The version with overlapping uses twice as much memory because it stores two blocks of data in parallel for the macro-pipeline, but it has still a smaller footprint than the transposition's version.

**Discussion**

The parallelization algorithm detailed above gives good results in terms of speedup and memory usage. The communication costs are well diminished and they are also partially overlapped. Also, the hermite scheme that we consider obtains better accuracy compared to previous Padé approximation. However, as an extension, we would like to consider the situation where $r_{min}$ is small. Considering MPI processes on the inner border of the plane and the gyroaverage of their points located at $(r_{min} \approx 0, \theta = *)$, the problem is that the Larmor circle will intercept cells further away than neighboring subdomains in $\theta$ direction. The current implementation is not ready for that. We are going to solve this issue in reducing the number of points near $r_{min}$ to diminish the communication cost (based on the work [81]). A non-uniform mesh will cover the poloidal plane (see Section 3.3.4), and tiles will be used to subdivide this non-uniform mesh. The tiles will be dispatched over several MPI process, this strategy will replace the uniform rectangular $r \times \theta$ patches with a basic block domain decomposition. We will handle the

center, in the vicinity of $r = 0$, with a single tile. This will solve the problems arising near magnetic axis in terms of large communication costs. The price to pay is to overhaul the current basic uniform mesh and the existing parallel domain decomposition (work in-progress by N. Bouzat).

## 2.7 Conclusion

Achieving high sustained performance on large machines requires attention to several factors. First, scalable algorithms and implementation are needed, *i.e* restitution time is reduced in inverse proportion to the number of computing elements. Second, one expect a good per-core performance on contemporary hardware (GPU or CPU). Third, memory consumption should not exhibit large parallel overheads in order for one application to be able to run large cases whenever a sufficiently high number of nodes are available. Fourth, one ought to consider a set of actions to check the accuracy and health of the code on a regular basis. In this chapter, we had a walkthrough of solutions to improve parallel algorithms in the Vlasov, Poisson and Gyroaverage solvers of the Gysela application. Performance bottlenecks have been incrementally removed. Hybrid programming with MPI+OpenMP helped us a lot to access several levels of parallelism, to achieve more flexibility in term of deployment, while allowing some memory savings. A fine memory study with specific method and tool has enabled to reach an extreme scale on the largest european supercomputer Juqueen. The topic of sequential optimizations is not covered here even if several actions have been achieved to improve memory accesses and a fine computation organization within the code, a fraction of these works will be approached in Section 4. The question of testing and having ways to check numerical results even for a complex parallel code will be raised in the next Chapter 3.

For effective use of parallel systems, it is essential to obtain a good match between algorithm requirements and architecture resources. As the processor technology evolves rapidly, to design adapted parallel algorithms is a work that never comes to an end. Thus, Gysela ought to evolve and adapt to many-core or GPU setting soon. It will need to switch to other programming models and runtime systems to tackle such challenges. This aim will require an even larger interdisciplinary approach compared to the current status.

The optimization of the checkpoint-restart procedures, the implementation of non-blocking writes on the parallel file system, and improvements of IO efficiency were studied in Gysela [25, 26, 79], but not displayed here. Here is a list of studies concerning parallel algorithms that I also conducted while not presented in this document and not related to Gysela [13, 21, 23, 28, 32, 34, 38, 39, 41].

# Chapter 3

# Importance of numerical schemes in scientific applications

*"The scientist does not study nature because it is useful;*
*he studies it because he delights in it,*
*and he delights in it because it is beautiful."*

Jules Henri Poincaré
The Value of Science

In order to provide reliable results, the methods and schemes should be tuned to lower the error induced by the numerical methods and also to enhance the accuracy if ever possible. The works described in this chapter address improvements that go along these directions and give the opportunity to better describe physics phenomena [1,27,81,85]. Furthermore, reproducibility of results is a strong requirement in most fields of research for experimental results to be called science. For results obtained through simulation software using high-performance computing (HPC) this translates as code quality demands. But codes tend to evolve at a very fast pace which often leads to the introduction of subtle bugs with new features all at once. In this framework, paying careful attention to the quality traceability of software is important. Automated testing of the code offers a way to help correct defects. This explains the rationale that pushed us to design specific measurements, establish mathematical relations that permit to characterize the correctness of our simulation results. These checks were accompanied further by putting into place a continuous integration platform (hosted by Inria) that automatically checks on a HPC machine a set of medium-sized Gysela cases [19].

In the first Section, a study will show how we improved conservation properties in Gysela. Second, the aligned method is presented that permits to diminish execution time and to enhance accuracy. The third Section deals with adapting the geometry of the poloidal plane to get more realistic simulations and describes a way to access to non-circular shaped plasma. At last, we give some physics achievements over the recent years with Gysela, and we draw some conclusions and perspectives.

## 3.1 Improving conservation properties

In gyrokinetic turbulent simulations, the knowledge of some stationary states can help reducing numerical artifacts. Considering long-term simulations, the qualities of the Vlasov solver and of the radial boundary conditions have an impact on the conservation properties. In order to improve mass and energy conservation mainly, the following methods are investigated here: fix the radial boundary conditions on a stationary state, use a 4D advection operator that avoids a directional splitting, interpolate with a delta-f approach. The combination of these techniques in Gysela led to a net improvement of the conservation properties in 5D simulations.

### 3.1.1 Introduction

Inaccurate description of the gyrokinetic equilibrium can yield nonphysical excitation of zonal flow oscillations [98]. Moreover, as stated in [152,98,124,151], it is important to define the initial condition using a relevant gyrokinetic equilibrium, especially in the context of collisionless full-f simulations and long-term simulations. However, at long simulation times, irrespective of the choice for the initial state (local or canonical Maxwellian), the turbulence robustly develops with identical statistical properties [124]. In the following, accuracy aspects are investigated for the Vlasov solver used in the Gysela code. If proper care is not taken for both the Vlasov solver and the gyrokinetic initial equilibrium, one can observe that some conservation properties are not satisfied, for example total

mass or energy. Adaptations we have done on the radial boundary conditions permit to avoid abnormal particle gains and losses. Then, several new features are presented: an operator splitting (linear versus non-linear terms), a more accurate computation of particle displacement fields that are part of the semi-Lagrangian scheme, a 4D interpolation scheme, a delta-f interpolation technique. Afterwards, numerical and experimental investigations show how the solutions we propose fix several problems. This study reinforces trust in the GYSELA code and allows one to better understand the role of some separate components of the solvers. Also, we see that radial boundary conditions have a crucial role with toroidal geometry. Moreover, understanding the external radial boundary condition is especially important for physics purposes. Indeed, the so-called scrape-off layer[1] has its own physics and needs a specific submodel. A long-term goal, currently undergoing, is to couple GYSELA with another submodel (or code) that will mimic the SOL physics specifically.

**Gyrokinetic Vlasov equation**

Let $\mathbf{z} = (r, \theta, \varphi, v_\parallel, \mu)$ be a variable in the 5D phase space, the distribution function of the guiding-center is $f(\mathbf{z}, t)$. The gyrokinetic Vlasov equation reads (no Source nor Collisions here, simplified version compared to Eq. (2.1) p.14):

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

The time evolution of the gyro-center coordinates $(\boldsymbol{x}, v_\parallel, \mu)$ are (collision-less electrostatic):

$$\frac{\mathrm{d}x^i}{\mathrm{d}t} = v_\parallel \boldsymbol{b}^* \cdot \boldsymbol{\nabla} x^i + \boldsymbol{v}_{Es_{GC}} \cdot \boldsymbol{\nabla} x^i + \boldsymbol{v}_{Ds} \cdot \boldsymbol{\nabla} x^i \tag{3.1}$$

$$m \frac{\mathrm{d}v_\parallel}{\mathrm{d}t} = -\mu \boldsymbol{b}^* \cdot \boldsymbol{\nabla} B - e \boldsymbol{b}^* \cdot \boldsymbol{\nabla} (J_0 \, \phi) + \frac{m v_\parallel}{B} \boldsymbol{v}_{Es_{GC}} \cdot \boldsymbol{\nabla} B \tag{3.2}$$

where $x^i$ corresponds to the $i$-th covariant coordinate of $\boldsymbol{x}$, $\boldsymbol{B}$ is the magnetic field (notation $B$ is the magnitude of $\boldsymbol{B}$), $\boldsymbol{J}$ stands for the plasma current density. Vacuum permittivity is denoted $\mu_0$. The $B^*_\parallel$ and $\boldsymbol{b}^*$ terms are defined as:

$$B^*_\parallel = B + \frac{m v_\parallel}{e \, B} \mu_0 \boldsymbol{b} \cdot \boldsymbol{J} \tag{3.3}$$

$$\boldsymbol{b}^* = \frac{\boldsymbol{B}}{B^*_\parallel} + \frac{m v_\parallel}{e B^*_\parallel} \frac{\mu_0 \boldsymbol{J}}{B} \tag{3.4}$$

The advection terms are:

$$\boldsymbol{v}_{Es_{GC}} \cdot \boldsymbol{\nabla} x^i = v^i_{Es_{GC}} = \frac{1}{B^*_\parallel} [J_0 \, \phi, x^i], \qquad \boldsymbol{b}^* \cdot \boldsymbol{\nabla} x^i = b^{*i} = \frac{\boldsymbol{B} \cdot \boldsymbol{\nabla} x^i}{B^*_\parallel} + \frac{m v_\parallel}{e B^*_\parallel} \frac{\mu_0 \boldsymbol{J} \cdot \boldsymbol{\nabla} x^i}{B},$$

$$\boldsymbol{v}_{Es_{GC}} \cdot \boldsymbol{\nabla} B = -\frac{1}{B^*_\parallel} [B, J_0 \, \phi], \qquad \boldsymbol{v}_{Ds} \cdot \boldsymbol{\nabla} x^i = v^i_{Ds} = \left( \frac{m v_\parallel^2 + \mu B}{e B^*_\parallel B} \right) [B, x^i] \, .$$

The Poisson bracket is defined by $[F,G] = \boldsymbol{b} \cdot (\boldsymbol{\nabla} F \times \boldsymbol{\nabla} G)$. The term $v_{E_{GC}}$ represents the electric $E \times B$ drift velocity of the gyro-centers and $v_D$ the curvature drift velocity. The Jacobian in phase space is $\mathcal{J}_{\mathbf{x}} . \mathcal{J}_{\mathbf{v}}$ with $\mathcal{J}_{\mathbf{x}}$ the jacobian in configuration space and $\mathcal{J}_{\mathbf{v}} = 2\pi \, B^*_{||}(r, \theta, v_\parallel)/m$ the jacobian in velocity space. Some references concerning the framework and related works are [10, 65, 66, 95, 132, 139].

**Quasi-neutrality equation**

*Description of the QN equation*

In an electrostatic code, the field solver reduces to the numerical solving of a Poisson-like equation [142]. In tokamak configurations, the plasma quasi-neutrality (denoted QN) approximation is currently made (see description p. 15). We note $n_0$ the initial equilibrium density (integral over phase space - except $r$ - of a reference equilibrium distribution function $f_{\mathrm{ref}}$ that will be defined afterwards[2]). We recall the QN equation already given by Eq. (2.2) (p. 15).

$$-\frac{1}{n_0(r)} \nabla_\perp \cdot \left[ \frac{n_0(r)}{B_0} \nabla_\perp \phi(r, \theta, \varphi) \right] + \frac{1}{T_e(r)} \left[ \phi(r, \theta, \varphi) - \langle \, \phi \, \rangle_{\mathrm{FS}} \right] = \tilde{\rho}(r, \theta, \varphi) \tag{3.5}$$

[1] also called SOL which is the plasma region close to the vessel wall and characterized by open field lines.

[2] In the following, $f_{\mathrm{ref}}$ will also be denoted $f_{eq}$ and refers to a canonical Maxwellian.

where $\tilde{\rho}$ is defined by

$$\tilde{\rho}(r,\theta,\varphi) = \frac{1}{n_0(r)} \int\int \mathcal{J}_{\mathbf{v}}\, J_0\,(f - f_{\rm ref})(r,\theta,\varphi,v_{\parallel},\mu)\, dv_{\parallel}\, d\mu. \tag{3.6}$$

with $\langle . \rangle_{\rm FS}(r)$ the average on the flux surface labeled by $r$, and $f_{\rm ref}$ representing a reference distribution function. By assumption $\phi=0$ for this distribution function $f=f_{\rm ref}$. Let us formally define what are the flux surface operator (denoted $\langle\, g\,\rangle_{\rm FS}$) and the $(\theta,\varphi)$-average operator (denoted $\bar{g}$) applied on a given function $g$

$$\begin{gathered} \bar{g}\,(r) = \tfrac{1}{4\pi^2} \int\int\, g\, {\rm d}\theta\, {\rm d}\varphi, \\ \langle\, g\,\rangle_{\rm FS}(r) = h(r) \int\int\, \mathcal{J}_{\mathbf{x}}(r,\theta)\, g\, {\rm d}\theta\, {\rm d}\varphi \\ \text{with } h(r) = \tfrac{1}{4\pi^2 \int\, \mathcal{J}_{\mathbf{x}}(r,\theta)\, {\rm d}\theta} \end{gathered} \tag{3.7}$$

Within the GYSELA setting we use in this work, we assume the following conservation property is well preserved at any time step (mass conservation)

$$\int\int\int \mathcal{J}_{\mathbf{x}}(r,\theta)\, n_0(r)\, \tilde{\rho}\, dr\, {\rm d}\theta\, {\rm d}\varphi = 0. \tag{3.8}$$

Let us remark that the variables $\phi$, $f$, $\tilde{\rho}$ depend also on time $t$. The function $f_{\rm ref}$ is fixed at startup very close to the initial distribution function $f^{t=0}$, and $f_{\rm ref}$ do not change over time.

*Description of the QN solver*

We now describe the extension that handles the toroidal setting instead of the cylindrical one given earlier. The equation (3.5) can be written as

$$\mathcal{P}\phi + \frac{1}{T_e}\left[\phi - \langle\, \phi\,\rangle_{\rm FS}\right] = \tilde{\rho} \tag{3.9}$$

where $\mathcal{P}$ is defined as

$$\mathcal{P} = -\frac{1}{n_0(r)} \nabla_{\perp} \cdot (n_0(r) \nabla_{\perp}) = -\left\{ \frac{\partial^2}{\partial r^2} + \left[ \frac{1}{r} + \frac{1}{n_0(r)} \frac{{\rm d}n_0(r)}{{\rm d}r} \right] \frac{\partial}{\partial r} + \frac{1}{r^2} \frac{\partial^2}{\partial \theta^2} \right\}$$

By applying the $(\theta,\varphi)$-average operator to the previous equation (3.9) and by using the fact that $\overline{\langle\, \phi\,\rangle_{\rm FS}} = \langle\, \phi\,\rangle_{\rm FS}$ then:

$$\mathcal{M}\bar{\phi} + \frac{1}{T_e}\left[\bar{\phi} - \langle\, \phi\,\rangle_{\rm FS}\right] = \bar{\tilde{\rho}} \tag{3.10}$$

where

$$\mathcal{M} = -\left\{ \frac{\partial^2}{\partial r^2} + \left[ \frac{1}{r} + \frac{1}{n_0(r)} \frac{{\rm d}n_0(r)}{{\rm d}r} \right] \frac{\partial}{\partial r} \right\}$$

One has the relations $\mathcal{M}\bar{\phi} = \mathcal{P}\bar{\phi}$ and $\mathcal{M}\langle\, \phi\,\rangle_{\rm FS} = \mathcal{P}\langle\, \phi\,\rangle_{\rm FS}$. Let $\Upsilon$ be $\Upsilon = \phi - \bar{\phi}$ then, by subtracting (3.10) to (3.9), and by using appropriate boundary conditions (we will discuss in next subsection what kind of boundary conditions we use), we obtain an equation on $\Upsilon(r,\theta,\varphi)$ ($\forall\theta \in [0,2\pi]$ and $\forall\varphi \in [0,2\pi]$):

$$\begin{cases} \left(\mathcal{P} + \frac{1}{T_e}\right) \Upsilon(r,\theta,\varphi) = \varrho(r,\theta,\varphi) \\ \qquad \text{with} \quad \varrho = \tilde{\rho} - \bar{\tilde{\rho}} \qquad \forall r \in [r_{\min}, r_{\max}] \\ \Upsilon(r_{\min},\theta,\varphi) \text{ and } \Upsilon(r_{\max},\theta,\varphi) \text{ are given} \\ \qquad \text{by boundary conditions.} \end{cases} \tag{3.11}$$

First, the unknown $\Upsilon$ can be solved in this equation without knowing $\langle\, \phi\,\rangle_{\rm FS}$ and $\bar{\phi}$. Second, to have access to the main unknown $\phi$, we would like now to compute $\langle\, \phi\,\rangle_{\rm FS}$ and $\bar{\phi}$ (because $\phi = \Upsilon + \bar{\phi}$). The equation (3.10) can be rewritten as

$$\begin{aligned} \mathcal{P}\left(\bar{\phi} - \langle\, \phi\,\rangle_{\rm FS}\right) + \mathcal{P}\langle\, \phi\,\rangle_{\rm FS} + \\ \frac{1}{T_e}\left[\bar{\phi} - \langle\, \phi\,\rangle_{\rm FS}\right] = \bar{\tilde{\rho}} \end{aligned} \tag{3.12}$$

Then, using the fact that $\langle\, \Upsilon\,\rangle_{\rm FS} = \langle\, \phi\,\rangle_{\rm FS} - \langle\, \bar{\phi}\,\rangle_{\rm FS} = \langle\, \phi\,\rangle_{\rm FS} - \bar{\phi}$, the previous equation leads to the following system (omitting boundary condition issues):

$$\mathcal{P}\langle\, \phi\,\rangle_{\rm FS} = \bar{\tilde{\rho}} + \left(\mathcal{P} + \frac{1}{T_e}\right) \langle\, \Upsilon\,\rangle_{\rm FS} \tag{3.13}$$

We solve this equation (3.13) to get $\langle\,\phi\,\rangle_{\rm FS}$. We obtain the expression of the electric potential $\phi$ as[3]:

$$\phi(r,\theta,\varphi) = \Upsilon(r,\theta,\varphi) + \underbrace{\langle\,\phi\,\rangle_{\rm FS}(r) - \langle\,\Upsilon\,\rangle_{\rm FS}(r)}_{\bar{\phi}(r)} \tag{3.14}$$

To summarize, the solving of (2.2) can be replaced by the solving of two simpler equations: (3.11) then (3.13). First, we solve (3.11) in $\Upsilon$. The variable $\varphi$ plays the role of a parameter, we can solve a set of 2D Poisson-like equations (in the poloidal plane). Each 2D problem reduces to a projection in Fourier space in $\theta$ direction and then finite differences in the radial direction. Second, equation (3.13) is treated as a differential equation only depending on the radial direction. Third, a sum is performed to get $\phi$ (with (3.14)). See also Section 2.4 for further details.

*New boundary conditions*

A first set of boundary conditions (denoted BC$_1$) is Dirichlet at $r_{\rm min}$ and $r_{\rm max}$. In (3.11), we impose $\Upsilon(r_{\rm min},\theta,\varphi)=0$ and $\Upsilon(r_{\rm max},\theta,\varphi)=0$ ($\forall\theta\in[0,2\pi]$, $\forall\varphi\in[0,2\pi]$). Concerning (3.13), we set $\langle\,\phi\,\rangle_{\rm FS}(r_{\rm min}) = \langle\,\phi\,\rangle_{\rm FS}(r_{\rm max}) = 0$ (we have then also $\bar{\phi}(r_{\rm min}) = \bar{\phi}(r_{\rm max}) = 0$). These conditions are easy to set up but there is one major drawback, the same potential (flux averaged) is forced at $r_{\rm min}$ and $r_{\rm max}$ which does not allow for the system to freely set a global radial gradient for $\langle\,\phi\,\rangle_{\rm FS}$ from $r_{\rm min}$ to $r_{\rm max}$. We will exemplify soon why it is a problem in term of physics.

A second set of boundary conditions (denoted BC$_2$) alleviates the constraint on the radial gradient of $\langle\phi\rangle_{\rm FS}$. As previously, we impose $\Upsilon(r_{\rm min},\theta,\varphi)=0$ and $\Upsilon(r_{\rm max},\theta,\varphi)=0$ ($\forall\theta\in[0,2\pi]$, $\forall\varphi\in[0,2\pi]$). Although this assumption simplifies the solver, the impact on the solution has not been yet evaluated. Then, we fix a Neumann condition at $r_{\rm min}$ and Dirichlet at $r_{\rm max}$ for Eq. (3.13): $\frac{\partial}{\partial r}\langle\phi\rangle_{\rm FS}(r_{\rm min})=0$ and $\langle\,\phi\,\rangle_{\rm FS}(r_{\rm max})=0$. Let us assume that $r_{\rm min}$ is small enough (*i.e* near 0) and therefore that $\mathcal{J}_{\mathbf{x}}(r_{\rm min},\theta)$ does not depend on $\theta$. In this configuration (verified in practice) for $r$ small enough, $\bar{\phi}(r)\approx\langle\,\phi\,\rangle_{\rm FS}(r)$ thanks to Eq. (3.8). We end up with some interesting properties:

$$\left\{\begin{array}{l}\partial_r\bar{\phi}(r_{\rm min})\approx\partial_r\langle\,\phi\,\rangle_{\rm FS}(r_{\rm min})=0\\ \bar{\phi}(r_{\rm max})=\langle\,\phi\,\rangle_{\rm FS}(r_{\rm max})=0\ .\end{array}\right.$$

To show the impact of boundary conditions on a simulation, let us consider a simulation in which ITG turbulence has grown up. We consider the distribution function and associated $\tilde{\rho}$ at one given time step. In Fig. 35, we plot the $\bar{\phi}(r)$ function using the two different boundary conditions for the same right-hand side $\tilde{\rho}$ (close to $r_{\rm min}$ and $r_{\rm max}$, $\tilde{\rho}$ is zero, whereas in the center of the radial domain where the turbulence is located, $\tilde{\rho}$ is non zero). For BC$_1$, the radial derivative of $\bar{\phi}(r)$ is non-zero at low $r$, that gives a net poloidal flow (because $v_{E_{GC}}$ has non-zero $\theta$ component) near $r_{\rm min}$. However, in BC$_2$ configuration $\bar{\phi}(r)$ is nearly constant near $r_{\rm min}$, then the spurious flow along $\theta$ direction disappears, which is desirable.

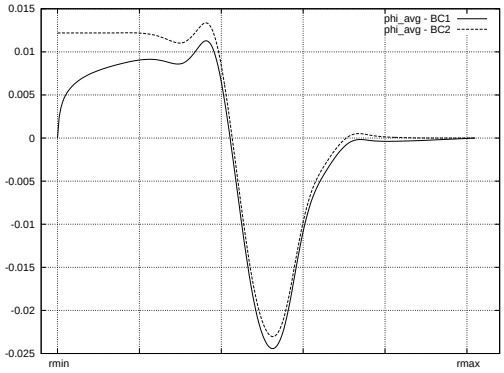



Figure 35: $\bar{\phi}(r)$ profiles for a ITG simulation at a given time step (turbulence has already grown up) using BC$_1$ and BC$_2$. Radial derivative at $r_{\rm min}$ is wrong for original BC$_1$.

### Defining proper radial boundaries

We have already described Vlasov and quasi-neutrality solvers that are two main components of the GYSELA code. Let us notice the computational domain is not formally closed in the radial and

[3]Remark: In slab geometry (old versions of GYSELA), we used to suppose $\langle\,\phi\,\rangle_{\rm FS}=\bar{\phi}$ and $\langle\,\Upsilon\,\rangle_{\rm FS}=0$.

velocity directions. We will describe which changes have been done to prevent particles from escaping the computational domain at radial boundaries.

*Description of the problem*

The situation with eddies, turbulence located near the radial boundaries $r_{\min}$ or $r_{\max}$ is difficult to handle. It commonly leads to electric potential structures that generate fluxes of particles in or out the computational domain. Taking this into account in the mass and energy balance is tricky. Also, the derivatives of $\phi$ have to be evaluated at $r_{\min}$ and $r_{\max}$ for computing displacements of particles, it is complex to get them accurately.

Thus, we have retained the following practical solution: to impose the distribution function equal to a reference function in the vicinity of radial boundaries. The reference distribution function that we consider is a stationary solution of the Vlasov equation corresponding to a vanishing electric potential. It follows that simple radial boundary conditions for Vlasov and QN solvers are accessible, for example Dirichlet and Neumann are both possible. Also, as $\tilde{\rho}$ (see Eq. (3.6)) will then be zero in the vicinity of $r_{\min}$ and $r_{\max}$, the $\phi$ potential is likely to be nearly constant in this area. Then, the influx/outflux of particles due to electric potential through radial boundaries of the computational domain should be null (not because of physics phenomena, but due to the new boundary conditions). This solution improves the setup of both Vlasov and QN solver and helps to close the computational domain in radial direction. Hereafter, we describe a method that one can use to force the distribution function near the radial boundaries.

*Adapting the Vlasov solver at radial boundaries*

Let us decompose $f$ as the sum of an equilibrium function $f_{eq}$ and a perturbation $\delta_f$:

$$f = f_{eq} + \delta_f$$

We will see afterwards how to build such a $f_{eq}$ function. We define a radial mask function $\mathcal{H}(r)$ (see also Fig. 36)

$$\begin{cases} \mathcal{H}(r_{\min}) = \mathcal{H}(r_{\max}) = 0 \\ \forall r \in [r_{\min}, r_{\max}], 0 \leq \mathcal{H}(r) \leq 1 \end{cases}$$

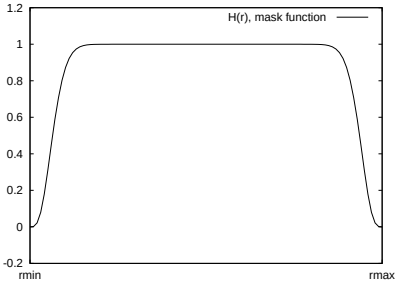



Figure 36: The mask function $\mathcal{H}$ (abscissa is radius $r$)

This function is smooth and equal to 1 almost everywhere except in the vicinity of radial boundaries. Near $r_{\min}$ and $r_{\max}$, the function $\mathcal{H}$ smoothly drops to zero. With the distribution function $f$, one can define:

$$\begin{cases} \delta_f = f - f_{eq} \\ f_\dagger = f_{eq} + \mathcal{H}\,\delta_f \\ \delta_{f_\dagger} = (1 - \mathcal{H})\,\delta_f \end{cases} \tag{3.15}$$

With this formulation, we have the property: $f = f_\dagger + \delta_{f_\dagger}$. The main benefit is that: $f_\dagger$ is equal to $f_{eq}$ at radial boundaries, and $f_\dagger$ is equal to $f$ in the center of the radial domain. The improvement of radial boundary conditions consists in using $f_\dagger$ in the Vlasov solver instead of $f$ (because $f_\dagger$ is equal to $f_{eq}$ at $r_{\min}, r_{\max}$ and $f_{eq}$ is invariant by Vlasov equation).

Practically, the method consists in removing $\delta_{f_\dagger}$ from $f$ to get $f_\dagger$ before the Vlasov step, and then solving Vlasov on $f_\dagger$ only. As $\delta_{f_\dagger}$ function contains a $(1 - \mathcal{H})$ factor, $\delta_{f_\dagger}$ is zero almost everywhere except in the vicinity of $r_{\min}$ and $r_{\max}$. The $\delta_{f_\dagger}$ function is designed to contain a relative small set of particles that are likely to escape the computational domain. We discard these particles from the Vlasov solver on purpose. After the Vlasov step, we add back $\delta_{f_\dagger}$ to $f_\dagger$ to recover $f$ and the same total number of particles. According to this procedure, $\delta_{f_\dagger}$ does not evolve in time through Vlasov solver, we can interpret this quantity as a reservoir of particles that has left the computational domain (the domain is truly open at $r_{\max}$).

Please note that the reference $f_{eq}$ of Eq. (3.15) can be changed occasionally during a simulation in order to fit the macroscopic evolution of parameters such as temperature/density profiles. This approach closes the computational domain in radial direction, as the particles are trapped into $\delta_{f_\dagger}$ when they approach the radial boundary limits. Then, mass conservation is obtained, the main drawback concerns the physics model that is altered near $r_{\min}$ and $r_{\max}$.

*Adapting the QN solver at radial boundaries*

For the quasi-neutrality solver, we would like to avoid large derivatives of the potential $\phi$ in the vicinity of radial boundaries. To achieve this goal, we are considering the previous decomposition $f = f_\dagger + \delta_{f_\dagger}$. The $f_\dagger$ function is built in order to be equal to $f_{eq}$ near $r_{\min}$ and $r_{\max}$. Then, we would like to use $f_\dagger$ and discard the $\delta_{f_\dagger}$ contribution in the right-hand side (RHS) of the QN equation. However, we have also to take into account the $\delta_{f_\dagger}$ part in order that the total mass is unchanged. As we want to consider $\delta_{f_\dagger}$ as something that does not participate anymore in the dynamics of the system we transform this quantity in a passive mass in the RHS of QN equation. To do that, we just have to rewrite Eq. (3.6) in the following way:

$$\tilde{\rho}(r,\theta,\varphi) = c_i + \frac{1}{n_0(r)} \int \mathcal{J}_{\mathbf{v}}\, d\mu \int dv_\parallel J_0\,(f_\dagger - f_{eq}), \tag{3.16}$$

with

$$c_i = \frac{\int \mathcal{J}_{\mathbf{v}}\, \mathcal{J}_{\mathbf{x}}\, J_0\,(f_{eq} - f_\dagger)\, dv_\parallel\, d\mu\, dr\, d\theta\, d\varphi}{\int\, \mathcal{J}_{\mathbf{x}}\, n_0(r)\, dr\, d\theta\, d\varphi}\ .$$

The term $c_i$ is built to recover mass conservation defined in Eq. (3.8) (see also [167] for a focus on this problem in a reduced setting). These changes alter locally the electric potential, near $r_{\min}$ and $r_{\max}$. Nevertheless, the energy balance is preserved, conservation of total energy remains true. In practice, the term $c_i$ is expected to be relatively small[4], because it represents the fraction of the particles that has left the computational domain through radial boundaries, it is small in practice.

*Numerical results: conservation issues in a toroidal 4D simulation*

Let us consider a simulation with a single value $\mu \neq 0$. Due to magnetic curvature/gradient, the drift velocities at large $v_\parallel$ transport some turbulent eddies straight to $r_{\min}$ and $r_{\max}$. In this configuration, we will look at the impact of the solutions we have just proposed for boundary conditions, versus the original code setting. The test case and initial conditions will be given in the upcoming Section 3.1.4.

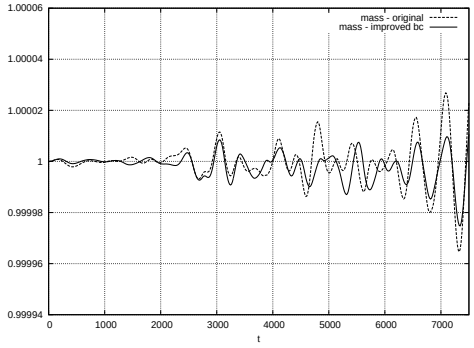



Figure 37: Evolution of the mass for a 4D toroidal test case, $\mu = 3$. The mass is normalized to 1 at $t = 0$.

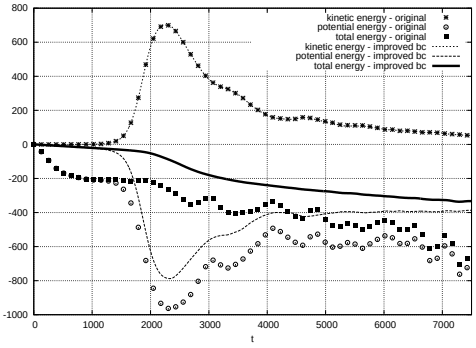



Figure 38: Evolution of the energies for a 4D toroidal test case, $\mu = 3$.

The specific treatment of radial boundary conditions improves the time evolution of energy conservation (as shown in Fig. 38). The energy curves report the relative energy: the energy at time step $t$ minus the energy at time step 0. Ideally, the total energy should remain 0 throughout the simulation. The curve denoted "`total energy - improved bc`" is closer to zero than the original one denoted "`total energy - original`". In the early time steps (from 0 up to $2000\,\Omega^{-1}$), total energy conservation is improved significantly: the total energy in the original version (black squares) rapidly departs from 0, whereas with the new boundary conditions the total energy (black thick curve) remains closer to zero. We will see afterwards that other reasons explain why total energy is not well conserved after $t = 2000\,\Omega^{-1}$.

The impact on mass conservation is however negligible (see Fig. 37), meaning that mainly particles with large velocity modulus are concerned by the new boundary conditions (they represent a small percentage of the total mass, but quite a significant part of kinetic energy).

*Numerical results: distribution function cut in a toroidal 4D simulation*

We can now have a closer look at the distribution function itself. In Fig. 39, the poloidal cross section of $\delta_f = f - f_{eq}$ is shown at a specific location in phase space ($v_\parallel = -5\,v_{th0}$, $\varphi = 0$, $t = 64\,\Omega^{-1}$, zoom on the center of poloidal cross section). A structure characterized by negative values of $\delta_f$ goes through the center of the poloidal cross section (top-down flow). On the left hand side, the original

[4]One can also consider to set $c_i$ to zero, the conservation of Eq. (3.8) is no more assured but the possible impact of $c_i$ on the radial profile of $\phi$ is decreased (Eq. (3.10)).

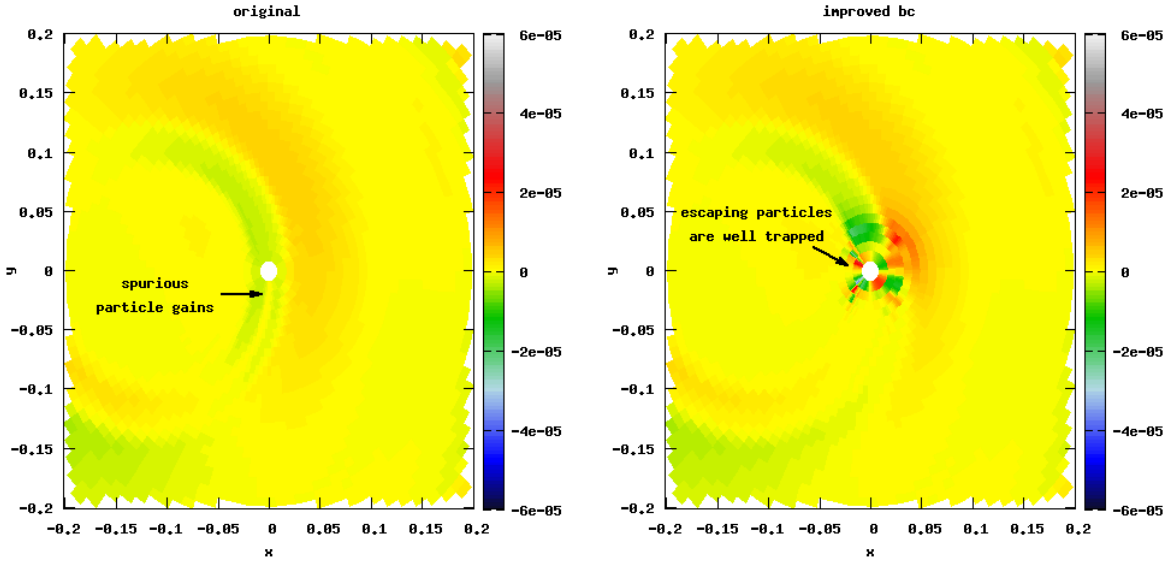



Figure 39: Poloidal cross section (in $x, y$ coordinates) of $\delta_f$ with a zoom on the center of the domain, location in phase space : $v_\parallel = -5\, v_{th0},\ \varphi = 0,\ \mu = 3,\ t = 64\,\Omega^{-1}$. Left plot: original code, right plot: improved boundary conditions.

code has an external boundary in the center of the grid (white disk) that absorbs some of these negative values and a kind of tail is generated (this is a wrong behavior that leads to particle gains). On the right hand side, the improved boundary conditions imply that many particles are trapped before they reach the central disk. These trapped particles create fine holes and bumps (located all around the center). They are not taken into account in the Vlasov solver (because we use $\delta_{f_\dagger}$ during Vlasov solve instead of $\delta_f$), but they are kept in the distribution function to compute macroscopic values (such as kinetic energy, mass, ...). Hence, these particles do not flow out the computation domain which is a good property. This process of collecting particles is not due to the physics model, but an artificial mechanism that we add to avoid numerical issues (it alters locally the transport process).

### 3.1.2 New numerical schemes

A set of new features are presented now: an operator splitting (linear versus non-linear terms), the accurate precomputation of some displacements that are part of the semi-Lagrangian scheme, the 4D interpolation scheme, the *delta-f interpolation* technique. The objectives were twofold:

- to evaluate pros and cons compared to the directional Strang splitting that is used in the current version of the GYSELA code,
- to combine this 4D advection with a *delta-f* approach for the interpolation.

#### Global separation of linear/nonlinear terms

The equations (3.1) and (3.2) can be split into two parts, using the same kind of procedure as described in [151]. The first part includes the nonlinear terms that depend on the electric potential. The second part comprises all other terms. One can solve these two parts separately. On the first hand, the nonlinear operator is described by (3.19), (3.20), on the second hand, the linear operator is presented in (3.17), (3.18).

**Linear operator $\mathcal{L}$**

$$\frac{\mathrm{d}x^i}{\mathrm{dt}} = v_\parallel \boldsymbol{b}^* \cdot \boldsymbol{\nabla} x^i + \boldsymbol{v}_{Ds} \cdot \boldsymbol{\nabla} x^i \qquad (3.17)$$

$$m\frac{\mathrm{d}v_\parallel}{\mathrm{dt}} = -\mu\, \boldsymbol{b}^* \cdot \boldsymbol{\nabla} B \qquad (3.18)$$

**Nonlinear operator $\mathcal{N}$**

$$\frac{\mathrm{d}x^i}{\mathrm{dt}} = \boldsymbol{v}_{Es_{GC}} \cdot \boldsymbol{\nabla} x^i \qquad (3.19)$$

$$m\frac{\mathrm{d}v_\parallel}{\mathrm{dt}} = -e\, \boldsymbol{b}^* \cdot \boldsymbol{\nabla}(J_0\, \phi) + \frac{m v_\parallel}{B}\, \boldsymbol{v}_{Es_{GC}} \cdot \boldsymbol{\nabla} B \qquad (3.20)$$

The linear operator exhibits large displacements at large modulus of parallel velocity, and also induces shear flows. These features can interfere with the nonlinear dynamics that possibly involves small displacements. Moreover, as the dynamics generated by the two operators are different, the accuracy problems have possibly not the same characteristics for the two operators; and the limitations (CFL-like conditions) on the time step are also not the same. Ideally, one should be able to fix the numerical scheme and time step of linear and nonlinear operators independently in order to achieve a given accuracy.

The current semi-Lagrangian scheme implemented in GYSELA code uses Strang splitting in the Vlasov solver. This is a directional splitting already described p. 15. For this setting we have divergence-free property of the full advection field of the Vlasov gyrokinetic equation, but not for each substep of the directional splitting (we do not add specific sources as proposed in [150]). Even if the scheme is not conservative from a mathematical point of view, we have noticed that errors do not accumulate much in our gyrokinetic simulations, and that mass is well conserved over long simulation times. Conservative approaches have also been tried out in semi-Lagrangian framework [95, 117], but up to now we have encountered some difficulties on the three following points: to guarantee the divergence-free property at discrete level, to preserve stationary states, to deal with radial boundary conditions.

Another point, we have remarked that the Strang splitting may lead to some troubles at high parallel velocities. Indeed, during one single directional substep, some large shifts in $\varphi, r$ and $\theta$ appear in the linear terms at high $|v_\parallel|$ (typically several spatial cells). Then, for not so large $\Delta t$, the evaluation of electric fields $E$ that depends on spatial location is not done at the exact spatial position at each substep of the directional splitting. To correct this, a possibility is to take smaller time step to recover small displacements in the linear operator and then a reasonable accuracy in the evaluation of $E$. The splitting between linear and nonlinear parts corrects this problem. The nonlinear operator is applied alone, thus the linear operator and its large shifts at high $v_\parallel$ modulus does not interact badly with the nonlinear solver. This approach is a little bit more expensive than the previous approach in term of computational cost. But it is counter-balanced by the fact that one can take a larger time step $\Delta t$.

**Precomputation of particle trajectories**

The foot of a characteristic ending at a grid position $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})$ is needed for the advection in the semi-Lagrangian method. Since the fields acting on particles for the *linear* part $\mathcal{L}$ do not depend on time $t$, one can approximate the foot of a particle trajectory - denoted $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})^\star$ - once for all, for a given time step $\Delta t$. We have used Runge-Kutta time integration scheme RK2 with a small time step to precompute these particle trajectories backward in time. This approach is possible for the linear terms, but not for non-linear terms $\mathcal{N}$ that depend on $\phi(t)$. Let us choose a $\delta t$, such as $M\,\delta t = \Delta t$ with $M \in \mathbb{N}^*$ and $M$ large enough. One can build a series as follows (using a $\alpha$ field deduced from (3.17) and (3.18) at p.62):

$$\begin{pmatrix} r^{n+\frac{1}{2}} \\ \theta^{n+\frac{1}{2}} \\ \varphi^{n+\frac{1}{2}} \\ v_\parallel^{n+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} r^{n} \\ \theta^{n} \\ \varphi^{n} \\ v_\parallel^{n} \end{pmatrix} - \frac{\delta t}{2}\,\alpha \begin{pmatrix} r^{n} \\ \theta^{n} \\ \varphi^{n} \\ v_\parallel^{n} \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} r^{n+1} \\ \theta^{n+1} \\ \varphi^{n+1} \\ v_\parallel^{n+1} \end{pmatrix} = \begin{pmatrix} r^{n} \\ \theta^{n} \\ \varphi^{n} \\ v_\parallel^{n} \end{pmatrix} - \delta t\,\alpha \begin{pmatrix} r^{n+\frac{1}{2}} \\ \theta^{n+\frac{1}{2}} \\ \varphi^{n+\frac{1}{2}} \\ v_\parallel^{n+\frac{1}{2}} \end{pmatrix}.$$

The initial condition is set to $r^0 = r_i,\ \theta^0 = \theta_j,\ \varphi^0 = \varphi_k,\ v_\parallel^0 = v_{\parallel\, l}$. After $M$ steps of Runge-Kutta iterations, it gives

$$\begin{pmatrix} r_i \\ \theta_j \\ \varphi_k \\ v_{\parallel\, l} \end{pmatrix}^\star = \begin{pmatrix} r^M \\ \theta^M \\ \varphi^M \\ v_\parallel^M \end{pmatrix}.$$

Because these trajectories can be computed only once when the simulation starts, $M$ can be taken quite large (we will assume $M = 64$ in the following). These precomputations do not impact significantly the global simulation time. We just need to store the results of these precomputations in memory and/or on the parallel file system. Other time integration schemes have been tried: RK3, RK4, and also larger values of $M$, no significant impact was observed in term of accuracy.
One of the main benefit of this method is: the feet of the characteristics are determined more accurately than the Taylor expansion of $\alpha$ that is used in the current version of GYSELA, because the time step $\delta t$ is a much smaller time step than $\Delta t$. One should also mention that this approach resembles to the *subcycling* technique [97] that is able to limit particle motion to one cell per cycle in PIC codes, and also improves accuracy and simplifies sorting.

**Interpolations by tensor product in 4D**

In this Subsection, we will describe a strategy that performs 4D interpolations using tensor product of cubic B-splines. Let us consider a one-dimensional function $g$ which is defined on a global domain $[x_{\min}, x_{\max}] \subset \mathbb{R}$. Suppose that we know the values $(g(x_i))_{\forall i=[0,N]}$ and we want to interpolate this discretized $g$ function with cubic splines. The projection $s$ of $g$ onto the cubic splines basis reads

$$g(x) \simeq s(x) = \sum_{\nu=-1}^{N+1} \eta_\nu \mathcal{B}_\nu(x),$$

where $\mathcal{B}_\nu$ is the cubic B-spline. The interpolating spline $s$ is uniquely determined by $(N+1)$ interpolating conditions

$$g(x_i) = s(x_i), \quad \forall i = [0, N],$$

and by the boundary conditions on the first derivative at both ends of the interval in order to obtain a $\mathcal{C}^1$ global approximation (periodic boundary conditions or others are also possible but not described here)

$$f'(x_0) \simeq s'(x_0), \quad f'(x_N) \simeq s'(x_N).$$

The $\eta_\nu$ coefficients are the solution of a linear system. A LU decomposition is used to find these unknowns $(\eta_\nu)_{\nu\in[-1,N+1]}$ depending on the inputs $(g(x_i))_{i\in[0,N]}$. Practically, as $\mathcal{B}_\nu$ are compactly supported, the interpolation of $g$ at a single location $x$ is computed with the formula:

$$g(x) \simeq s(x) = \sum_{\nu=m^0-1}^{m^0+2} \eta_\nu \mathcal{B}_\nu(x),$$
$$\text{with} \quad m^0 = \lfloor N \, \frac{x - x_{\min}}{x_{\max} - x_{\min}} \rfloor \, .$$

Let us consider now a 4D function $g$ that will be projected on $s$ in cubic spline basis. Using a tensor product of cubic B-spline in 4D, one can construct spline coefficients $\eta_{\nu^0,\nu^1,\nu^2,\nu^3}$, with $\nu^d \in [0, N^d]$, $\forall d = 0, 1, 2, 3$ . These new $\eta$ coefficients can be found by using LU decompositions (four LU decompositions, one for each dimension). The interpolations in four dimensions are computed using the following expression (with $m^d$ well chosen depending on $x^d$ and on the discretization):

$$s(x^0, x^1, x^2, x^3) = \sum_{\nu^0=m^0-1}^{m^0+2} \sum_{\nu^1=m^1-1}^{m^1+2} \sum_{\nu^2=m^2-1}^{m^2+2} \sum_{\nu^3=m^3-1}^{m^3+2} \eta_{\nu^0,\nu^1,\nu^2,\nu^3} \; \mathcal{B}_{\nu^0}(x^0)\, \mathcal{B}_{\nu^1}(x^1)\, \mathcal{B}_{\nu^2}(x^2)\, \mathcal{B}_{\nu^3}(x^3)$$

**Four-dimensional numerical scheme**

The usual way to perform a single time step in the GYSELA code consists of a series of directional advections: $(\hat{v_\parallel}/2, \hat{\varphi}/2, \hat{r\theta}, \hat{\varphi}/2, \hat{v_\parallel}/2)$. Let us now consider an avoidance of this Strang splitting. Let us suppose that we advance in time *only the linear part* $\mathcal{L}$ and the feet of the characteristics are computed for all grid points in phase space (see 3.1.2). The Algo. 11 sketches the corresponding 4D numerical scheme. Parallelization of the computations will not be detailed here.

1 Compute the spline coeff. $\eta^n$ of the 4D function $f^n$;
2 **for** *All grid points* $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})$ **do**
3 | Get precomp. foot of characteristic $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})^\star$;
4 | Interpolate $f^n$ at location $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})^\star$;
5 | $f^{n+1}(r_i, \theta_j, \varphi_k, v_{\parallel\, l}) \leftarrow$ the interpolated value;

**Algo. 11:** 4D numerical scheme (given $\mu$ value)

Let us remark that the $\mathcal{N}$ operator uses the directional splitting method and not 4D advections. It implies large computational costs to estimate 4D displacement fields for operator $\mathcal{N}$ that we wish to avoid. In contrast, the 4D displacement fields for the linear operator $\mathcal{L}$ is computed once at the beginning, so that its overall computational cost becomes negligible for long simulation runs.

The parallel domain decomposition that we have designed for this scheme is along $r$ and $\theta$ directions (MPI parallelization). In the parallel setting (*pid* the index of the process), only local points located in $[r_{pid}^{min}, r_{pid}^{max}] \times [\theta_{pid}^{min}, \theta_{pid}^{max}]$ are taken into account at line 2. If the foot of one characteristic falls outside the local domain, communications are generated to ask the process able to perform this interpolation to send back the result. In addition to this MPI approach, a OpenMP fine grain parallelization has been set up on the $\varphi$ loop.

**Delta-f interpolation strategy**

The delta-f interpolation strategy is one of the goals that is targeted to achieve the conservation of invariant/equilibrium states. The Algo. 12 describes how we have used delta-f techniques to improve the interpolation accuracy.

This algorithm requires that $f_{eq}$ is invariant for the Vlasov equation. If $f_{eq}$ verifies this property, the value of $f_{eq}$ at the foot of any characteristic is equal to the value of a given grid points,

1 If needed, update $f_{eq}$;
2 Compute the spline coeff. $\eta^n$ of the 4D function ${\delta_f}^n = f^n - f_{eq}$;
3 **for** *All grid points* $(r_i, \theta_j, \varphi_k, v_{\parallel l})$ **do**
4 | Get precomputed foot $(r_i, \theta_j, \varphi_k, v_{\parallel l})^\star$;
5 | Interpolate ${\delta_f}^n$ at location $(r_i, \theta_j, \varphi_k, v_{\parallel l})^\star$;
6 | $f^{n+1}(r_i, \theta_j, \varphi_k, v_{\parallel l}) \leftarrow$ interp. ${\delta_f}^n$ value $+ f_{eq}(r_i, \theta_j, \varphi_k, v_{\parallel l})$;

**Algo. 12:** 4D delta-f scheme (given $\mu$ value)

*i.e* $f_{eq}(r_i, \theta_j, \varphi_k, v_{\parallel l})^\star = f_{eq}(r_i, \theta_j, \varphi_k, v_{\parallel l})$. With such method, we are able to conserve perfectly an the equilibrium state $f_{eq}$. Furthermore, if we suppose that $f^n$ is near $f_{eq}$, $\delta_f^n$ is small. The first Algo. 11 will generate interpolation errors that grow along with the spatial derivatives of $f_{eq}$. The new Algo. 12 has smaller interpolation errors that are only proportional to derivatives of $\delta_f$. Please note that one may need to update the $f_{eq}$ function if this function is too far from $f$ (first line of the algorithm). However, in the simulations we consider here, we do not need to perform this step, as $f$ remain close to $f_{eq}$.

Remark: The same delta-f scheme could also have been combined with the Strang splitting we have usually. But, as $f_{eq}$ is not preserved at each directional substep, special care should be taken.

#### Global algorithm for one time step

To sum up the different approaches that will be compared afterwards, a global algorithm for the time stepping is presented in Algo. 13 and 14. The Algo. 13 is the original algorithm with the Strang splitting and the time integration scheme described p. 16. The Algo. 14 corresponds to the new approach with $\mathcal{L}/\mathcal{N}$ splitting, the 4D interpolation in the $\mathcal{L}$ operator, and possibly the delta-f interpolation inside the $\mathcal{L}$ operator.

1 $\phi^n \leftarrow$ QN solver on $f^n$;
2 $f^{n+1/2} \leftarrow$ Strang split of $(\mathcal{L}+\mathcal{N})(f^n, \phi^n)$ over $\Delta t/2$;
3 $\phi^{n+1/2} \leftarrow$ QN solver on $f^{n+1/2}$;
4 $f^{n+1} \leftarrow$ Strang split of $(\mathcal{L}+\mathcal{N})(f^n, \phi^{n+1/2})$ over $\Delta t$;

**Algo. 13:** Time stepping without $\mathcal{L}/\mathcal{N}$ splitting

1 $f^\star \leftarrow \mathcal{L}(f^n)$ over $\Delta t/2$;
2 $\phi^\star \leftarrow$ QN solver on $f^\star$;
3 $f^\dagger \leftarrow$ Strang split of $\mathcal{N}(f^\star, \phi^\star)$ over $\Delta t/2$;
4 $\phi^\dagger \leftarrow$ QN solver on $f^\dagger$;
5 $f^\sharp \leftarrow$ Strang split of $\mathcal{N}(f^\dagger, \phi^\dagger)$ over $\Delta t$;
6 $f^{n+1} \leftarrow \mathcal{L}(f^\sharp)$ over $\Delta t/2$;

**Algo. 14:** Time stepping with $\mathcal{L}/\mathcal{N}$ splitting

Algo. 14 is therefore more expensive than Algo. 13. First, the number of steps shown is a bit larger (6 versus 4). Second, the linear operator in the new version requires 4D interpolations which are a bit more expensive (algorithmic cost) compared to the original directional splitting. Thus, the new Algo. 14 (with delta-f interpolation) roughly doubles execution time compared to original Algo. 13. The memory footprint is also increased in Algo. 14, by less than a factor two, because the feet of the characteristics in operator $\mathcal{L}$ are temporarily stored in memory.

### 3.1.3 Gyrokinetic simulations - reduced settings

In this Section, we evaluate the new methods presented previously on several simple test cases. In the following, the default timing unit is the ion cyclotron period $\Omega^{-1} = m_i/(Z_i e B)$.

#### Unperturbed motion of particles

We are interested in numerical methods that are able to conserve equilibrium states of the Vlasov equation. First, it is a property that ensures a good level of accuracy of the Vlasov solver in full-scale simulations, but, second, such invariant property is also important by itself in order to establish reference scenarii to test/verify the code. This aspect is crucial for large HPC codes with a lot a feature combined all together. It helps detect bugs by checking violations against invariance. Generally speaking, conservation properties are crucial to distinguish correct execution from bad one for a scientific simulation code. In addition, this also is helpful for debugging purposes.

We focus in this Section on gyrokinetic models with one single $\mu$ value and with $\mu \neq 0$ (*i.e.* non drift-kinetic models, the case with $\mu = 0$ is simpler [89] and is not considered here). Let us assume the following hypothesis: considering the global separation of linear/nonlinear terms, we discard $\mathcal{N}$ and keep only $\mathcal{L}$. The quasi-neutrality solver is also switched off. Therefore, we use no field solver, (equivalent to impose $\phi$=0). In this configuration, we estimate essentially the quality of the initial distribution function and the quality of the Vlasov solver for the linear terms ($\mathcal{L}$ operator), as we will see afterwards.

### Test case 1

**Description of the test case.** Let us consider an initial distribution function that should be a steady state for the Vlasov equation. An equilibrium solution of the collisionless gyrokinetic equation must satisfy some conditions. In an axisymmetric toroidal configuration, a gyrokinetic Vlasov equilibrium is defined by three constants of motion: the magnetic moment $\mu$, the energy $\mathcal{E}=m\,v_\parallel^2/2+\mu\,B(r,\theta)$ (assuming $\phi$ is null), and the canonical toroidal angular momentum $P_\varphi=e\,\psi(r)+m\,I v_\parallel/B(r,\theta)$ (where $I$ is a constant used in the definition of $\boldsymbol{B}=\frac{I}{B}(\boldsymbol{e_\varphi}+\frac{r}{q(r)R_0}\boldsymbol{e_\theta})$). The $\psi(r)$ function is defined thanks to the safety factor $q(r)$ by the relation: $d\psi/dr=-B_0\,r/q(r)$.

Let us assume that we have set up an initial equilibrium. One can estimate both the accuracy of this equilibrium and of the Vlasov solver by:

- measuring the difference between the initial function and the distribution function at a given time step after several Vlasov solving steps.
- a convergence study in time and space discretizations for a simulation that includes several Vlasov solving steps.

For the sake of simplicity and to reduce the computational costs, the simulations presented in this Section are set up to $N_\mu = 1$. Let us initialize a simulation using a steady state. For an axisymmetric equilibrium, the distribution function is constant along the $\varphi$ direction. It should remain constant using the $\mathcal{L}$ operator. Our goal is to quantify the numerical error induced by the numerical scheme, knowing that the difference to the initial distribution function should remain zero. The test case setting is characterized by (with $\rho^\star$ the gyroradius normalized to the plasma size): $\mu$=3., $\rho^\star$=.01, $N_r$=256, $N_\theta$=256, $N_\varphi$=16, $N_{v_\parallel}$=128. The initial distribution function $f_{init} = f_{eq}$ is taken as :

$$\begin{cases} \forall P_\varphi \in [-\infty, P_{\varphi_1}], & f_{eq}(\mathcal{E},\mu,P_\varphi) = 0 \\ \forall P_\varphi \in [P_{\varphi_2}, \infty], & f_{eq}(\mathcal{E},\mu,P_\varphi) = e^{-\mathcal{E}} \\ \forall P_\varphi \in [P_{\varphi_1}, P_{\varphi_2}], & f_{eq}(\mathcal{E},\mu,P_\varphi) = e^{-\mathcal{E}}\,\frac{1}{2}\left(1+\cos\left(\frac{\pi\,(P_{\varphi_2}-P_\varphi)}{P_{\varphi_2}-P_{\varphi_1}}\right)\right) \end{cases}$$

where $P_{\varphi_1}$ and $P_{\varphi_2}$ are well chosen in order to localize the large gradient of the distribution function in the middle the radial profile. We will look at the norms $u_p(t) = \|f^t - f_{eq}\|_p$ with $p = 1, \infty$. The thermal velocity is denoted $v_{th}$. The timing unit is the ion cyclotronic time $\Omega^{-1}$.

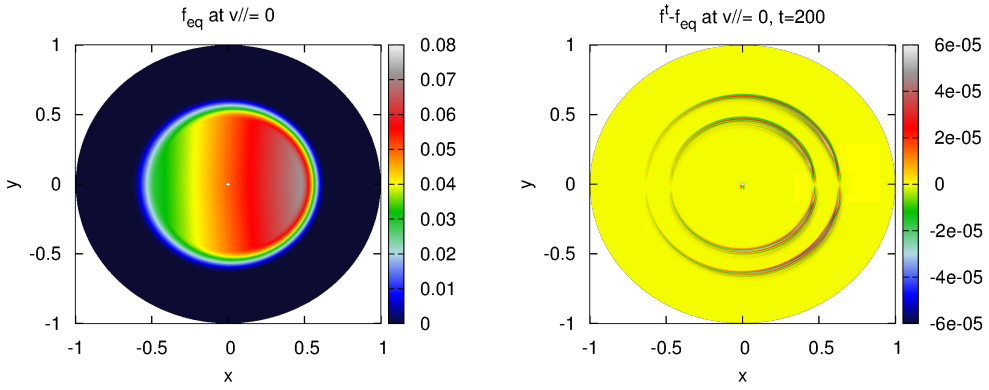



Figure 40: Time evolution of a steady state distribution function without 4D interpolation and without delta-f interpolation techniques, poloidal cut at $v_\parallel = 0, \varphi = 0$ (ideally $f^t - f_{eq}$ remains zero)

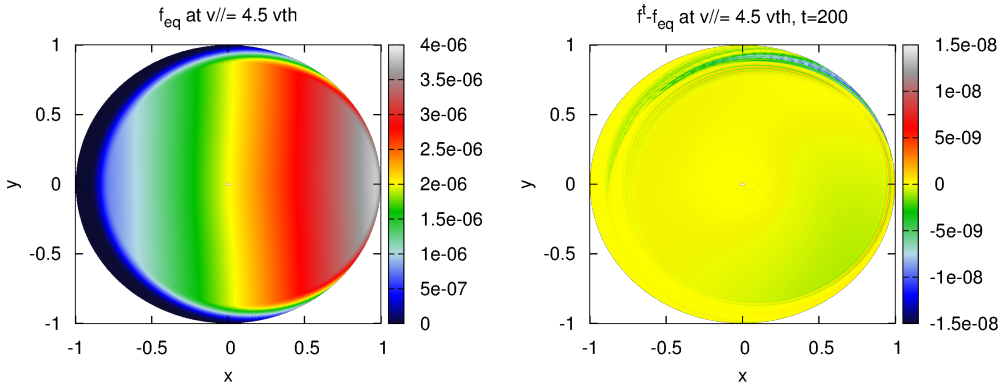



Figure 41: Time evolution of a steady state distribution function without 4D interpolation and without delta-f interpolation techniques, poloidal cut at $v_\parallel = 4.5\,v_{th}, \varphi = 0$ (ideally $f^t - f_{eq}$ remains zero)

Poloidal cuts are presented in Fig. 40, with ($v_\parallel$=0, $\varphi$=0 fixed) at $t$=0 and $t$=200. These correspond to simulations using standard approach with Strang splitting, none of the new numerical schemes is

used. One can observe on this illustration that numerical artifacts develop on $f^t$ in a poloidal cut at $v_\parallel = 0$. These approximation errors come from the interpolation operator and from the computation of the feet of characteristics. These errors are small compared to the mean value of $f^t$, but errors can be located where $f^t$ is close to zero (relative error is large).

On Fig. 41, a poloidal cut at $v_\parallel = 4.5\, v_{th}$ is presented. Let us consider energetic particles, some of them are able to encounter radial boundaries. This Fig. shows that numerical problems arise at external boundary (right plot, $t = 200$). The difficulty is the following: in the semi-Lagrangian scheme one looks for the position of one given particle back into the past and estimates the value of distribution function at this location. If this particle was outside the computational domain one time step ago, one has however to approximate the value of the distribution function with an ad-hoc procedure. As the equilibrium function $f_{eq}$ is not defined outside the computational domain[5], our method is to stick escaping particles to the last radius $r = r_{\max}$. It is this procedure that induces the biggest numerical perturbation originating on the external boundary condition.

Using 4D advection technique combined with delta-f interpolation, the numerical artifacts shown in Fig. 40 and 41 do not appear *at all* (thus it is not shown here). Indeed, the interpolation during the advection is performed on the distribution function $\delta_f^t = f^t - f_{eq}$, and $\delta_f^t$ is equal to zero at the beginning of the simulation. The $\delta_f^t$ function remains exactly zero over all time step. This approach solves perfectly the problem of preserving the invariant state $f_{eq}$.

### Test case 2

**Description of the test case.** The previous test case was focusing on preserving a steady state solution, but in practice the distribution function used in a realistic simulation can often be represented as the sum of a steady state solution plus a perturbation $f = f_{eq} + \delta_f$. There are multiple methods and several criteria to establish the best $f_{eq}$ to use, we will not discuss this issue here. However, the following case will focus on the benefits brought by delta-f interpolation technique to represent the time evolution of a small perturbation over of a steady state solution of Vlasov. Let us define $f_{init}$ as $f_{eq} + \delta_f^0$ with $f_{eq}$ reads

$$\begin{cases} \forall P_\varphi \in [-\infty, P_{\varphi_1}], & f_{eq}(\mathcal{E},\mu,P_\varphi) = 0 \\ \forall P_\varphi \in [P_{\varphi_2}, \infty], & f_{eq}(\mathcal{E},\mu,P_\varphi) = e^{-\mathcal{E}} \\ \forall P_\varphi \in [P_{\varphi_1}, P_{\varphi_2}], & f_{eq}(\mathcal{E},\mu,P_\varphi) = e^{-\mathcal{E}} \frac{1}{2}(1+\cos(\frac{\pi\,(P_{\varphi_2}-P_\varphi)}{P_{\varphi_2}-P_{\varphi_1}})) \end{cases}$$

and $\delta_f^{t=0}$ is a smooth perturbation which is localized in the area of a single point $P^i = (\mathcal{E}^i, P_\varphi^i, \theta^i)$. The $\delta_f^{t=0}$ function does not depend on $\varphi$, this test case is axisymmetric and at any time step the solution is identical for each $\varphi$ value. On Fig. 42, the $f_{eq}$ and $\delta_f^{t=0} = f^{t=0} - f_{eq}$ functions are plotted.

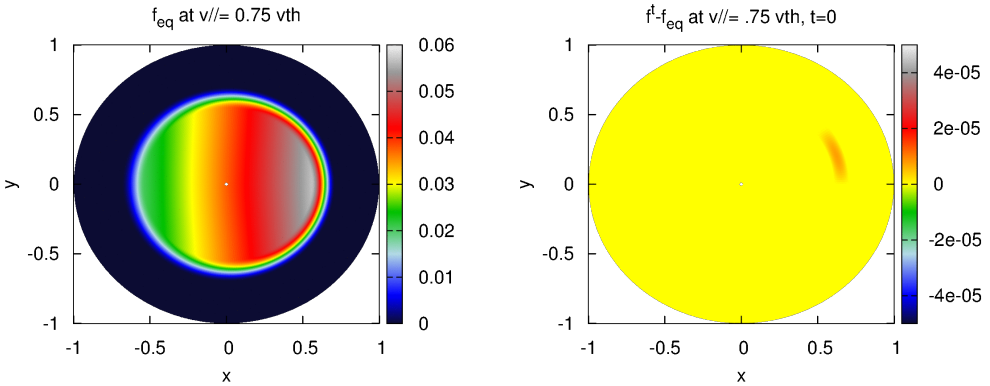



Figure 42: Initial state of test case 2, poloidal cuts at $v_\parallel = 0.75 v_{th}, t = 0$

After a few time steps, the distribution function $\delta_f^{t=20}$ is shown in Fig. 43. In the left panel, the original version of the code is used, whereas in the right panel the 4D advections with delta-f interpolation schemes are switched on. Without the delta-f approach, artifacts spoil the signal: in the center at $r = r_{\min}$, and also in the regions where the cubic splines have difficulties fitting the $f$ slope. With the delta-f interpolation, these problems disappear. There is no difficulty to handle feet of the characteristics outside the domain because it is assumed that one has $\delta_f = 0$ outside, which is a well known value. The artifacts due to the slope of $f_{eq}$ do not arise because the interpolation operator acting on $\delta_f^t$ does not see this slope.

To conclude, the delta-f interpolation method tends to reduce the numerical error. The main drawback is that we assume that: the distribution function is very near an equilibrium function at

[5] there is no great impediment, it would be possible to define a specific extension to define $f_{eq}$ outside.

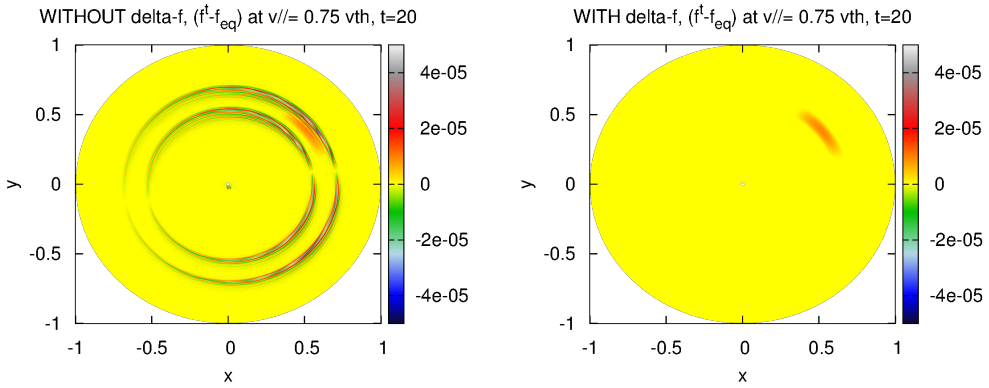



Figure 43: State at $t = 20$, poloidal cuts at $v_{\|} = 0.75 v_{th}$ of original code (left), modified version with delta-f interpolation (right)

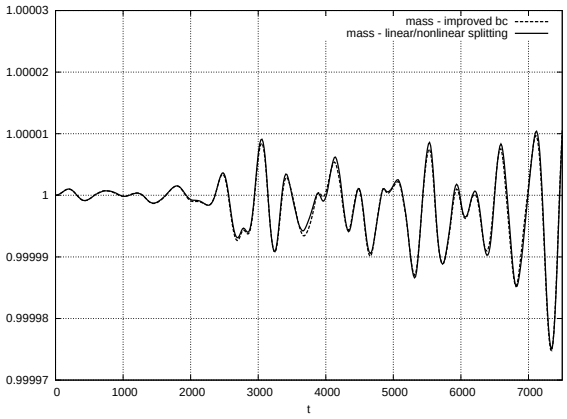



Figure 44: Evolution of the mass (it should remain 1) for a 4D toroidal test case, $\mu \neq 0$

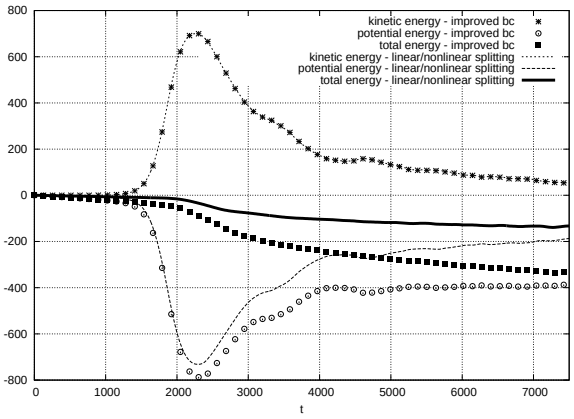



Figure 45: Evolution of the energies for a 4D toroidal test case, $\mu \neq 0$

radial boundaries ($\delta_f$ almost zero). The efficiency of the method is increased if one can compute a steady state function close to the current distribution function.

### 3.1.4 Gyrokinetic simulations - full-scale 4D settings

In the previous paragraphs, the field solver of non-linear terms were switched off. Now, we consider a full-scale case including non-linear terms and quasi-neutrality solver, giving access to non-linear physics. The simulation presented earlier p. 61 is the reference framework on top of which we will evaluate here a set of new methods. The improved boundary conditions is switched on in all the experiments presented hereafter.

#### Description of the setting

For the sake of simplicity, we will describe in this section only 4D experiments (with $\mu \neq 0$). The initial distribution function $f_{init}$ has been chosen close to an equilibrium $f_{eq}$ in order to maximize the benefit of the delta-f interpolation approach. The initial distribution function $f_{init}$ is equal to $f_{eq}$ plus a small perturbation (a bath of modes). The distribution $f_{eq}$ is a function of the three motions invariants only, $P_\varphi, \mathcal{E}, \mu$. The $f_{eq}$ is computed such that the radial profile of temperature $T_i(r)$ (averaged over $\theta, \varphi$ dimensions) and density $n_i(r)$ match the ones we prescribed in input. The radial gradients of these profiles are chosen so that Ion Temperature Gradient (ITG) instability develops. The turbulence drive is ensured by thermal baths at the radial edges in $r_{\min}$ and $r_{\max}$ that are imposed during the whole simulation. Some of the key parameters of the following simulation are: $\mu$=0.4, $\rho^\star$=.02, aspectratio=3, $N_r$=128, $N_\theta$=128, $N_\varphi$=64, $N_{v_\|}$=92.

#### Impact of the Linear/Non-Linear splitting

Let us evaluate the improvements brought by all the techniques described in Section 3.1.2. First, the new splitting introduces a clear separation between linear terms and non-linear terms, which improves quantitatively the time integration of particle trajectories. Second, the precomputation of particles displacements leads to an improved localization of the foot of characteristics in the semi-Lagrangian

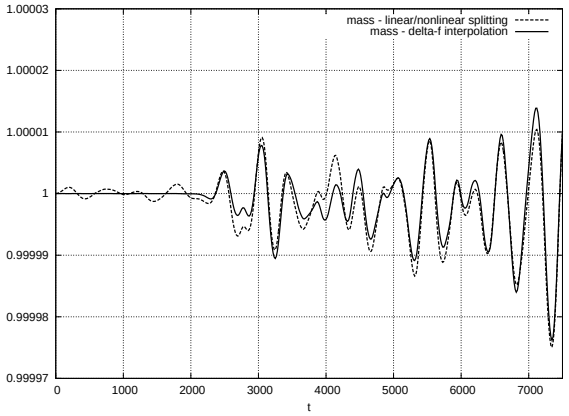



Figure 46: Evolution of the mass (it should remain 1) for a 4D toroidal test case, $\mu \neq 0$.

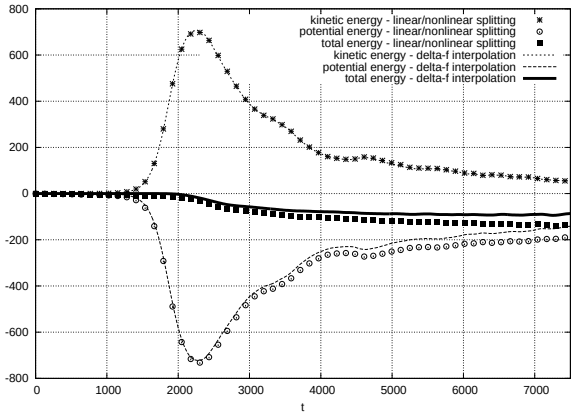



Figure 47: Evolution of the energies for a 4D toroidal test case, $\mu \neq 0$

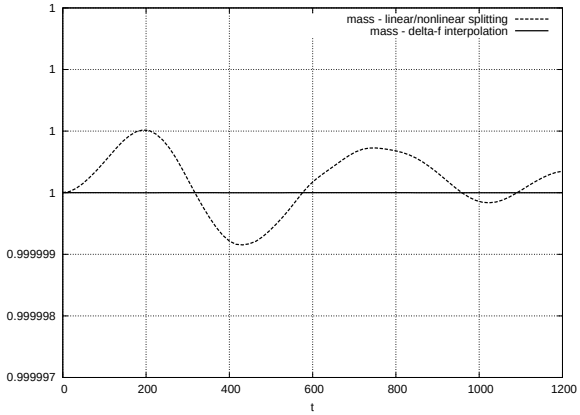



Figure 48: Zoom on the first time steps, mass for a 4D toroidal test case, $\mu \neq 0$

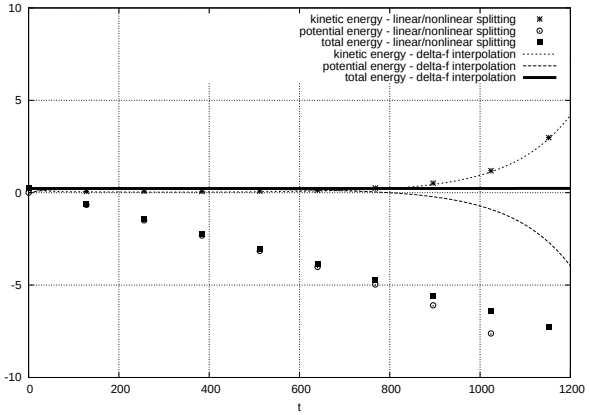



Figure 49: Zoom on the first time steps, energies for a 4D toroidal test case, $\mu \neq 0$

scheme, and then on the accuracy of the Vlasov solver. Third, the 4D advection avoids the possible discrepancy due to the directional splitting occurring at too large time steps.

The combination of these three techniques significantly improves the energy conservation property, as shown in Fig. 45 (relative total energy is set to zero at $t = 0$). The total energy obtained with these techniques (black thick line) is closer to zero than the total energy previously observed (black plain squares). However, potential energy and mass curves (Fig. 44) are almost not modified. We deduce that these modifications mainly correct the particle trajectories of particles that have high $|v_\parallel|$.

**Impact of the delta-f interpolation approach**

We consider the delta-f interpolation in addition to the previous setting.
With delta-f version, the mass remains almost at the reference initial value on the period from $t = 0$ to $t = 2000$ (see Fig. 46 and a zoom on early time steps in Fig. 48). This is a quite desirable behavior. However, once we reach the non-linear saturation phase ($t > 2000$), the mass conservation begins to degrade. Furthermore, Fig. 49 shows a zoom on the energies during the beginning of the simulation. The total energy of delta-f interpolation version (black thick line) is a lot better than the previous simulation (black squares). Overall, in order to simulate fine phenomena that develop close to a steady state (for example $t < 1000$ here), the delta-f interpolation technique seems to be required.

Let us analyze more precisely the mass curves. Each of the operators $\mathcal{L}$ and $\mathcal{N}$ should theoretically conserve the mass independently. One can track during a simulation how much each operator degrades the mass. On specific simulations, we took a time step sufficiently low in order to have well converged simulations in time and space. In these simulations, we have observed:

- the operator $\mathcal{L}$ induces mass conservation errors (without delta-f strategy) in the beginning of the simulation before the saturation phase (meaning $t \leq 2000$ here). These errors are larger than those of operator $\mathcal{N}$. Using the delta-f approach, these errors caused by $\mathcal{L}$ are reduced by several orders of magnitude.
- the operator $\mathcal{N}$ is predominantly responsible for the lack of conservation of the mass in the saturation phase (meaning $t \geq 2000$ here).

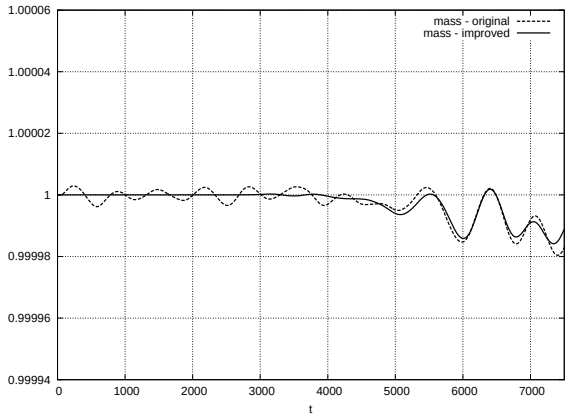



Figure 50: Evolution of the mass for a 5D toroidal test case

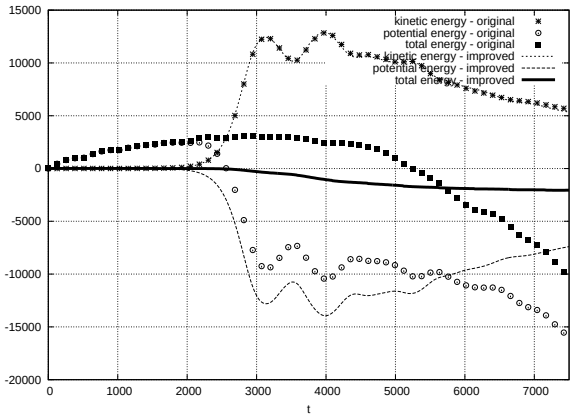



Figure 51: Evolution of the energies for a 5D toroidal test case

The framework of this work is not a conservative approach that would enforce mass or energy conservation (this kind of approach has been proposed in other simulation codes). Therefore, we just observe these quantities as indicators of the quality of a simulation. In Fig. 47, even with the improvement we have put in place, the relative total energy is not perfectly constant in time. The main reasons are the following: the toroidal setting (as opposed to slab geometry) introduces more complex particle trajectories, the high energetic particles carry much energy (they contribute much to kinetic energy) and are also difficult to handle (they need a small time step to be well tracked), the filamentation in phase space occurs together with numerical diffusion degrading the total energy.

In Figs. 50 and 51, a reference 5D case in the original version of the code is compared to the same case with the new version of the code that includes specific treatment for boundary conditions, operator splitting (separation of linear/non-linear operators), delta-f interpolation in 4D. It is noteworthy that the analysis of the different improvements that have been carried out for the 4D simulations in the previous Sections are also true in 5D setting.

### 3.1.5 Conclusion

Several numerical schemes have been investigated to improve the conservation properties expected in a gyrokinetic code. The semi-Lagrangian code Gysela was enhanced by these schemes in terms of energy conservation, mass conservation and in preserving some stationary states. The first technique targeted the radial boundary conditions. Radial boundaries were adapted in order to prevent particles from escaping the computational domain. Second, the Vlasov solver has been split into two separate steps: the linear part and the non-linear part. This change improved energy conservation and the modeling of the fastest particles. Third, a 4D advection technique combined with a delta-f interpolation scheme and an accurate precomputation of the feet of characteristics allow: first, a better representation of a distribution function close to a stationary state, second, a net improvement on energy and mass conservations. The long-time stability of such systems is a challenging area in applied mathematics. Having a set of methods to check and ensure a set of conservation properties helps a lot to build confidence in the numerical methods and in the simulation tool.

It would be a major contribution to add collision operator in this framework. A possible mechanism to extend this work would be to switch cleverly between a local Maxwellian as reference distribution function in the collision operator [190] and a canonical Maxwellian for the Vlasov operator. Another means would be to use a better equilibrium that takes into account both operators [108].

## 3.2 Aligned interpolation method

### 3.2.1 Introduction

In a tokamak, due to the large confining magnetic field, a fast homogenization of the different physical quantities occurs along the magnetic field lines; this leads to very smooth and small variations along the field lines, whereas the scale length of the variations is very small (comparable to the gyro-radius) in a perpendicular direction. This strong *anisotropy* should be taken into account for more efficient simulations. It is typically done by using field aligned coordinates in many gyrokinetic codes. The fundamental idea is to use coordinates that follow field lines. With such coordinates a flux tube (a tube with a surface parallel to $\boldsymbol{B}$), which is bent by magnetic curvature and twisted by magnetic shear, is mapped into a rectangular domain. However this approach has the drawback of needing a non-conformal correction after one turn, either in the poloidal or the toroidal direction, which yields

a break of symmetry. More importantly, field-aligned coordinates become singular when approaching the separatrix[6] (see Fig. 52) in a divertor configuration, with potentially serious consequences on the robustness of the numerical algorithm that employs them.

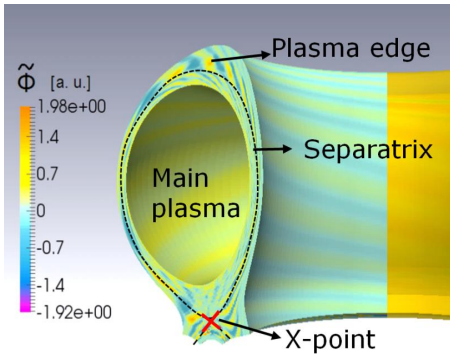



Figure 52: 3D view of 1/4 torus in a TOKAM3X simulation, the electric potential fluctuations are shown.

A very promising alternative, which is very flexible in regard to the choice of coordinates, has been introduced by Hariri-Ottaviani [146]. The main idea is to compute the derivatives locally along the field lines, getting the needed values for finite differences by interpolation to the intersection points of a field line with the poloidal planes. We are interested here in a thorough numerical investigation of this idea in the context of gyrokinetic simulations using semi-Lagrangian methods. Then, we adapted this approach to the semi-Lagrangian context. We have validated this method on the constant oblique advection equation and then on $4D$ and $5D$ models with oblique magnetic field in cylindrical and toroidal geometries. The strength of this method is that one can reduce the number of points in the longitudinal/parallel direction (along the field lines). Numerical experiments show that field-aligned interpolation leads to considerable memory savings for the same level of accuracy; substantial extra savings are also expected in much larger reactor-scale simulations (ongoing work).

Firstly, we give the numerical algorithms that are employed for performing interpolation and differentiation in a "field-aligned" fashion. Second, we will describe the constant velocity oblique advection to show the main feature of the method. Finally, the GYSELA setting in toroidal geometry (circular tokamak) is presented and the aligned method is benchmarked against a standard (*i.e.* non-aligned) version. The work shown here [1] has been carried out through a collaboration involving CEA (V. Grandgirard, M. Ottaviani, myself), IPP-Garching (Y. Guçlu, E. Sonnendrücker) and University of Strasbourg (M. Mehrenberger).

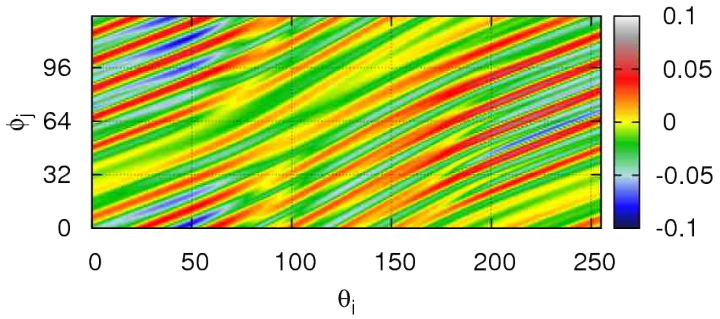



Figure 53: Aligned method aim: interpolate along field line that show smooth variations in the $(\theta, \varphi)$ plane. Electric potential plotted here.

### 3.2.2 Description of the numerical tools

To describe the 2D field-aligned interpolation method, we consider a magnetic flux surface at $r = r_0$, parameterized by the angular coordinates $(\theta, \varphi) \in [0, 2\pi] \times [0, 2\pi]$. This setting is natural in toroidal geometry. Our goal is to interpolate a sufficiently regular function $g(\theta, \varphi)$, $2\pi$-periodic in both coordinates, at an arbitrary location $(\theta^\star, \varphi^\star)$. We assume that the values $g(\theta_i, \varphi_j)$ are known on the uniform 2D grid $(\theta_i, \varphi_j) = (i\,\Delta\theta, j\,\Delta\varphi)$ with $\Delta\theta = 2\pi/N_\theta$, $\Delta\varphi = 2\pi/N_\varphi$ and $(i, j) \in [0..N_\theta - 1] \times [0..N_\varphi - 1]$. By periodicity we can extend this to $(i, j) \in \mathbb{Z}^2$. There exists a unique index $j^\star \in \mathbb{Z}$ and $0 \leq \beta < \Delta\varphi$ such that $\varphi^\star = \varphi_{j^\star} + \beta$ . We then define

$$\varphi_{j^\star+k} = \varphi_{j^\star} + k\,\Delta\varphi, \quad k = \varsigma, .., \xi \ .$$

We will use information stored in the 1D slices $g(\theta = *, \varphi = \varphi_{j^\star+k})_{k=\varsigma,..,\xi}$ to perform the aligned interpolation at $(\theta^\star, \varphi^\star)$. Let us define a function $\mathit{fieldline}_\theta(\theta, \varphi, j)$ that gives a $\theta$-value that corresponds to the intersection of the field line (or an approximation of the field line) that passes by the point $(\theta, \varphi)$ and the line $(\theta = *, \varphi_j)$. This function is the cornerstone of the method, as it provides a way to interpolate using values that are close to each other, because the locations of these values are

[6] Boundary between closed and open field lines, separating the toroidally confined region from the region where field lines connect to material surfaces.

aligned to the physical structures. The $fieldline_\theta$ function is chosen such as all interpolated points $h_k$ are aligned on a single field line.

The first stage of the method is to compute $\varrho_{\theta^\star,\varphi^\star}(k)_{k=\varsigma,..,\xi}$ by interpolating $g$ at positions $(fieldline_\theta(\theta^\star,\varphi^\star,j^\star+k),\varphi_{j^\star+k})_{k=\varsigma,..,\xi}$. We currently employ cubic splines to interpolate along the $\theta$ direction on the 1D slices $g(\theta=*,\varphi=\varphi_{j^\star+k})_{k=\varsigma,..,\xi}$. The formula for $fieldline_\theta$ that we have been using so far is the linear approximation

$$fieldline_\theta(\theta^\star,\varphi^\star,j^\star+k)=\theta^\star+\iota(r_0)\,(\varphi_{j^\star+k}-\varphi^\star)\ ,$$

which is the equation of a straight line. This expression is a good approximation in the case of the circular tokamak we consider with GYSELA (because of its medium-large aspect-ratio). If this assumption is not fulfilled, one can take a more accurate description for the field line. The $fieldline_\theta$ function can be easily changed in the code: it is effectively a parameter of the method. The impact on the simulation would be a small additional cost compared to the very cheap linear approximation, along with an improvement in the accuracy of the method.

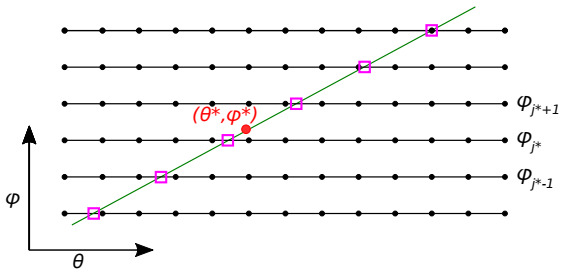



Figure 54: Illustration of the aligned interpolation scheme for a target point at position $(\theta^\star,\varphi^\star)$; the squares are located at $(\theta=fieldline_\theta(\theta^\star,\varphi^\star,j^\star+k),\varphi=\varphi_{j^\star+k})_{k=\varsigma,..,\xi}$; the values at square positions are interpolated using values known at black small points; the value at the red circle position $(\theta^*,\varphi^*)$ is interpolated using values known at the square positions.

The second stage of the method consists in interpolating $g(\theta^\star,\varphi^\star)$ using the values aligned on the parallel direction we just get: $\varrho_{\theta^\star,\varphi^\star}(k)_{k=\varsigma,..,\xi}$. To achieve this, we use Lagrange polynomials of degree $2d+1$ taking $\varsigma=-d,\xi=d+1$. The pseudo-code implementation of the scheme is presented in Algo. 15, and an illustration is given in Fig. 54.

**Input** : $g$, $theta^\star$, $phi^\star$
**Output** : $g^\dagger$

**for** $j=0,N_\varphi$ **do**
  $\eta(i=*,j)\leftarrow$ spline coefficients for $g(i=*,j)$
**for** $j=0,N_\varphi$ **do**
  **for** $i=0,N_\theta$ **do**
    `// foot of characteristics phi*, theta*`
    `// are computed beforehand`
    $\varphi^\star\leftarrow phi^\star(i,j)$; $\theta^\star\leftarrow theta^\star(i,j)$;
    $j^\star\leftarrow$ index of the left grid point close to $\varphi^\star$ ;
    **for** $k=-d,d+1$ **do**
      $\theta_k\leftarrow fieldline_\theta(\theta^\star,\varphi^\star,j^\star+k)$;
      $\varrho_k\leftarrow$ 1D spline interpolation along $\theta$ at $\theta_k$ using $\eta(i=*,j^\star+k)$;
    $g^\dagger(i,j)\leftarrow$ 1D Lagrange interpolation using values $(\varrho_k)_{k=-d,d+1}$

**Algo. 15:** Aligned interpolation in 2D

**Input** : $g$, $\epsilon$
**Output** : $dg/d\varphi$

**for** $j=0,N_\varphi$ **do**
  $\eta(i=*,j)\leftarrow$ spline coefficients for $g(i=*,j)$
**for** $j=0,N_\varphi$ **do**
  **for** $i=0,N_\theta$ **do**
    **for** $k=-d,d+1$ **do**
      $\theta_k^+\leftarrow fieldline_\theta(\theta_i,\varphi_j+\epsilon,j+k)$;
      $\theta_k^-\leftarrow fieldline_\theta(\theta_i,\varphi_j-\epsilon,j+k)$;
      $\varrho_k^+\leftarrow$ 1D spline interpolation along $\theta$ at $\theta_k^+$ using $\eta(i=*,j+k)$;
      $\varrho_k^-\leftarrow$ 1D spline interpolation along $\theta$ at $\theta_k^-$ using $\eta(i=*,j+k)$;
    $\varrho^+\leftarrow$ 1D Lagrange interpolation at $(\theta_i,\varphi_j+\epsilon)$ using values $(\varrho_k^+)_{k=-d,d+1}$;
    $\varrho^-\leftarrow$ 1D Lagrange interpolation at $(\theta_i,\varphi_j-\epsilon)$ using values $(\varrho_k^-)_{k=-d,d+1}$;
    $\frac{dg}{d\varphi}(i,j)\leftarrow\frac{\varrho^+-\varrho^-}{2\epsilon}$

**Algo. 16:** Derivatives along $\varphi$, aligned scheme

### Field-aligned computation of derivatives

In the GYSELA code, we need to evaluate $\phi$ derivatives with respect to $\varphi$ of the electric potential $\phi(r,\theta,\varphi)$. It is essential to compute the non-linear terms appearing in the advection equations, but also in the diagnostics that compute a set of macroscopic physical variables. In order to do so with a reduced number of points in the $\varphi$ direction (authorized by the aligned interpolation approach), a scheme has to be designed to get an accurate approximation of these derivatives. We have evaluated two alternatives to estimate $\partial\phi/\partial\varphi$: the first one relies on separately computing the polar derivative $\partial\phi/\partial\theta$ and the parallel derivative $\boldsymbol{b}\cdot\nabla\phi$, and then performing a projection using

the formula $\partial\phi/\partial\varphi = \left[\boldsymbol{b}\cdot\nabla\phi - (b_\theta/r)\partial\phi/\partial\theta\right]R/b_\varphi$; the second one performs a field-aligned interpolation similar to Algo. 15 to compute two accurate values of $\phi(r,\theta,\varphi\pm\epsilon)$, and then employs the finite difference formula $\partial\phi/\partial\varphi(r,\theta,\varphi) \approx [\phi(r,\theta,\varphi+\epsilon) - \phi(r,\theta,\varphi-\epsilon)]/2\epsilon$. Algo. 16 describes the second solution, which is effectively used in the GYSELA code since we observed higher accuracy for this approach in actual simulations.

### 3.2.3 Constant velocity oblique advection

To exemplify how the method works we will now go through a simple transport equation in a setting that includes a strong anisotropy.

#### Equation

Here the oblique magnetic field $\mathbf{B}$ whose norm is $B$ (which depends on $r$) writes

$$\mathbf{B} = B\mathbf{b},\ \mathbf{b} = b_z\hat{\mathbf{z}} + b_\theta\hat{\boldsymbol{\theta}},\ b_\theta = \frac{c}{\sqrt{1+c^2}},\ b_z = \frac{1}{\sqrt{1+c^2}},\ c = \frac{\iota r}{R},$$

and is parameterized by $B_0 := Bb_z$ and the rotational transform iota which satisfies

$$\iota = \frac{b_\theta/r}{b_z/R} = \frac{1}{q},$$

where $q$ is called the safety factor[7]. To demonstrate the advantages of the aligned interpolation that we introduce, we intend to solve the constant oblique advection equation which is a lot more simple to manipulate and understand than the full gyrokinetic setting:

$$\partial_t f + v\mathbf{b}\cdot\nabla f\ . = 0, \tag{3.21}$$

This equation is a subpart of the gyrokinetic equations (fast motion along field lines) if one focuses on advection terms along $(\theta,\varphi)$ variables. We will use an interpolation that is aligned along the direction of the magnetic field $\mathbf{b}$. Note that another possible strategy would be to adapt the grid to magnetic field lines (see [109, 143] for example); one strength of the present approach is that the location grid points does not need to be changed, which permits to ease the implementation in an existing code. Also, the aligned approach may or may not be activated depending on the physical case under consideration giving the user more flexibility.

#### Numerical scheme

Writing $\varphi = \frac{z}{R}$, we have to solve for $g := g(t,\theta,\varphi) = f(t,r,\theta,R\varphi,v)$, the constant oblique advection equation (3.21). We have

$$\partial_t f + v\nabla_\parallel f = \partial_t f + v\frac{b_\theta}{r}\partial_\theta f + vb_z\partial_z f = 0,$$

which leads to

$$\partial_t g + v\frac{b_\theta}{r}\partial_\theta g + v\frac{b_z}{R}\partial_\varphi g = \partial_t g + \tilde{v}\left(\iota\partial_\theta g + \partial_\varphi g\right) = 0, \tag{3.22}$$

with $\tilde{v} = \frac{vb_z}{R}$, and $g$ is $2\pi$ periodic in $\theta$ and $\varphi$.
Let $\Delta t \in \mathbb{R}^+$, and $t_\ell = \ell\Delta t,\ \ell\in\mathbb{N}$. We have the relation

$$g(t_\ell + \Delta t,\theta,\varphi) = g(t_\ell,\theta - \iota\tilde{v}\Delta t,\varphi - \tilde{v}\Delta t).$$

Let $N_\theta, N_\varphi \in \mathbb{N}^*$, and $\theta_i = \frac{2\pi i}{N_\theta}$, $\varphi_j = \frac{2\pi j}{N_\varphi}$, which can be defined for $i,j\in\mathbb{R}$. We suppose to know values $g_{\ell,i,j} \simeq g(t_\ell,\theta_i,\varphi_j)$, for $i = 0,\dots,N_\theta-1$, $j = 0,\dots,N_\varphi-1$. By periodicity, we assume $i,j\in\mathbb{Z}$.

#### Numerical results

In the case of constant oblique advection, we have to solve (3.22). We consider an initial function with a well defined helicity[8] $g(0,\theta,\varphi) = g_0(m\theta - n\varphi)$, so that

$$\mathbf{b}\cdot\nabla f = \left(m\frac{b_\theta}{r} - n\frac{b_z}{R}\right)g_0'(m\theta - n\varphi) = k_\parallel g_0'(m\theta - n\varphi),$$

[7] When $\iota = 0$, we are close to the setting of the classical drift kinetic model given in [139, 115] for example.
[8] magnetic helicity is a measure of twist and linkage of magnetic field lines. It gives the extent to which the field lines wrap and coil around one another.

where

$$k_\parallel := \frac{b_z}{R}\,(n - \iota m) = \frac{b_z}{qR}\,(m - qn)\ .$$

In order to have $\mathbf{b}\cdot\nabla f$ bounded, we look for situations where

$$|m - qn| \leq 1,$$

as in real tokamaks, it is assumed that $k_\parallel$ will typically be in the range of $[-\frac{1}{q\,R}, \frac{1}{q\,R}]$. For simplicity purposes, We will use in the sequel $g_0 = \sin$.

The displacement due to advection equation has a main parameter $\Delta t$. An extended set of various $\Delta t$ will be investigated because the error of one given numerical scheme depends on it. We choose a value for safety factor which is a non-rational surface case, $q = \sqrt{2}$. We look at four important configurations (targeting gyrokinetic simulations) for the initial functions with $k_\parallel$ close to 0 or 1.

$$\begin{array}{lll} A : (n = -5,\ m = -7) & k_\parallel \approx 0.07 & \frac{1}{q\,R} \\ B : (n = -24,\ m = -34) & k_\parallel \approx -0.06 & \frac{1}{q\,R} \\ C : (n = -5,\ m = -6) & k_\parallel \approx 1.07 & \frac{1}{q\,R} \\ D : (n = -24,\ m = -33) & k_\parallel \approx 0.94 & \frac{1}{q\,R} \end{array} \tag{3.23}$$

In Figs. 55 and 56, the abscissa of the plots are $N_\varphi$, and the ordinate are the $L^\infty$ norm which is the maximum of the difference of the function computed after *one time step* versus the analytical function which is here

$$g(\Delta t, \theta, \varphi) = \sin(m\,(\theta - \Delta t) - n\,(\varphi - q\,\Delta t))\ .$$

The maximum ($L^\infty$-norm) is over all the grid points. We show the results of the aligned versus the standard (non-aligned) scheme. Some parameters are fixed for this study: $N_\theta = 400$, $q = \sqrt{2}$, $\Delta t = 0.9$.

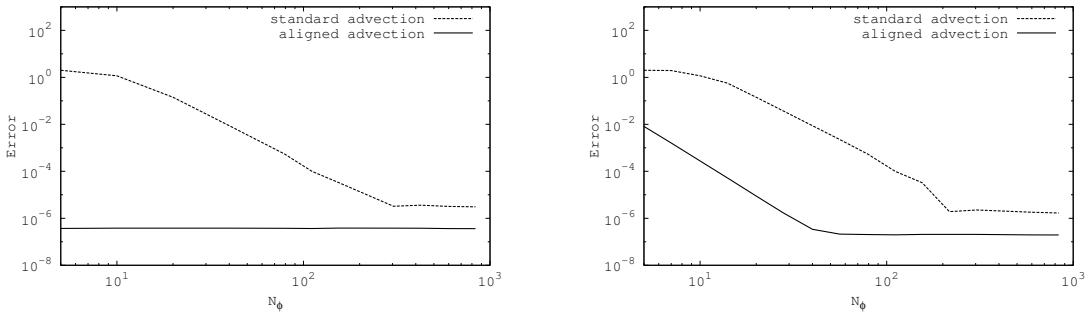



Figure 55: Error in $L^\infty$-norm compared to the analytical solution for advection $N_\theta = 400$, $q = \sqrt{2}$, $n = -5$ and $m = -7, k_\parallel \approx 0.07$ (left), versus $m = -6, k_\parallel \approx 1.07$ (right)

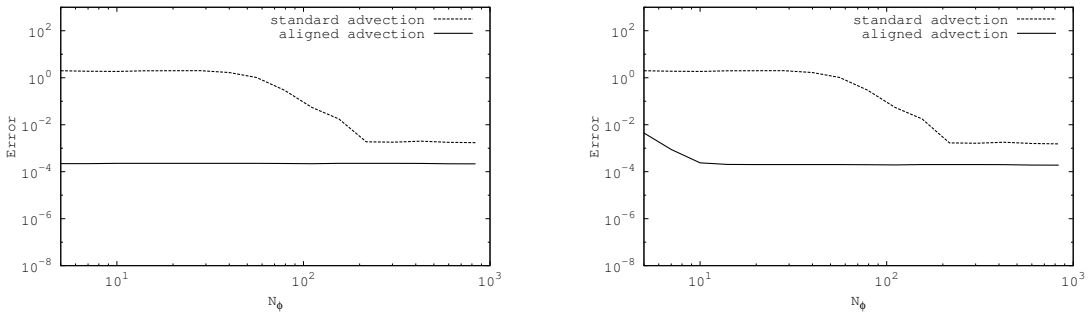



Figure 56: Error in $L^\infty$-norm compared to the analytical solution for advection $N_\theta = 400$, $q = \sqrt{2}$, $n = -24$ and $m = -34, k_\parallel \approx -0.06$ (left), versus $m = -33, k_\parallel \approx 0.94$ (right)

For left-hand side plots on Figs. 55-56, $k_\parallel$ is near 0 and the aligned method is very accurate, even if one takes a low value for $N_\varphi$. If one considers now the right-hand side plots, $k_\parallel$ is close to $\frac{1}{qR}$. Even if the aligned method gives lower error than standard method, the error is bigger at small $N_\varphi$ than with lower $k_\parallel$.

Fig. 55 considers lower frequency than Fig. 56. The behavior in the two cases are similar except that there is a shift of the curves along the $\varphi$ direction. This is expected, and what is interesting is that the aligned method behaves well (low error is achieved) even with low $N_\varphi$ values. In Fig. 57, the

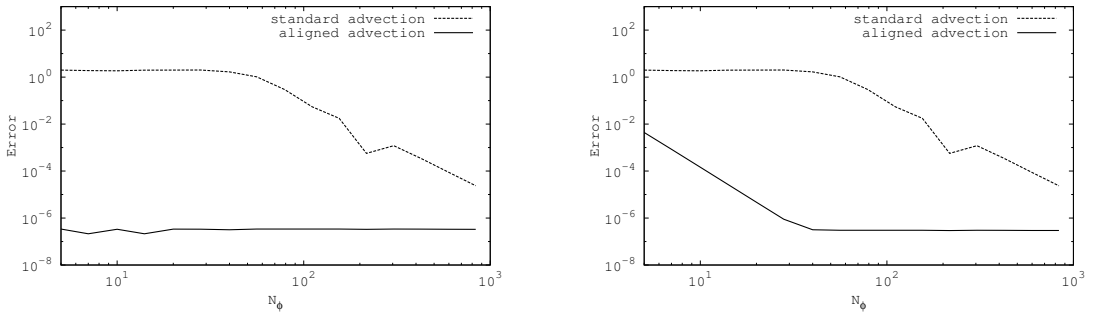



Figure 57: Error in $L^\infty$-norm compared to the analytical solution for advection $N_\theta = 2000$, $q = \sqrt{2}$, $n = -24$ and $m = -34, k_\parallel \approx -0.06$ (left), versus $m = -33, k_\parallel \approx 0.94$ (right)

discretization along $\theta$ has been refined ($N_\theta = 2000$) compared to Fig. 56. The asymptotic error (at large $N_\varphi$) is lowered. The aligned method for $k_\parallel$ close to zero (left-hand side plot) is very accurate with an $L^\infty$ error lower than $10^{-6}$. As the gain of the aligned method is effective to improve accuracy or to reduce the number of points in $\varphi$ direction, we will now look to the integration achieved within GYSELA and associated results.

### 3.2.4 Aligned interpolation within Gysela

#### Parallel algorithms

The Algo. 17 sketches the main features concerning the Vlasov solver that we are interested in. The usual way to perform a single Vlasov solving in the GYSELA code consists of a series of directional advections: $(\hat{v_\parallel}/2, \hat{\varphi}/2, \hat{r\theta}, \hat{\varphi}/2, \hat{v_\parallel}/2)$. This solver uses two parallel domain decompositions for the distribution function $f$. The 1D advections along $\varphi$ and $v_\parallel$ are performed with a first domain decomposition that retains all points of $f$ along these two dimensions $(\varphi, v_\parallel)$ locally in the MPI process. Then, a transpose of the distributed data structure $f$ is performed that involves large collective communications. Then, the 2D advection along both $r$ and $\theta$ dimensions can be done, this step uses a domain decomposition along $\varphi$, $v_\parallel$ and $\mu$ directions. After a second transposition of $f$, two 1D advections are again performed.

| | |
|---|---|
| 1D advection in $v_\parallel$ | $(\forall(\mu,r,\theta) = [local], \forall(\varphi, v_\parallel) = [*])$; |
| 1D advection in $\varphi$ | $(\forall(\mu,r,\theta) = [local], \forall(\varphi, v_\parallel) = [*])$; |
| **Transpose** $f$; | |
| 2D advection in $(r,\theta)$ | $(\forall(\mu,\varphi,v_\parallel) = [local], \forall(r,\theta) = [*])$; |
| **Transpose** $f$; | |
| 1D advection in $\varphi$ | $(\forall(\mu,r,\theta) = [local], \forall(\varphi, v_\parallel) = [*])$; |
| 1D advection in $v_\parallel$ | $(\forall(\mu,r,\theta) = [local], \forall(\varphi, v_\parallel) = [*])$; |

**Algo. 17:** Standard GYSELA Vlasov solver

| | | |
|---|---|---|
| 1: | 1D advection in $v_\parallel$ | $(\forall(\mu,r,\theta) = [local], \forall(\varphi, v_\parallel) = [*])$; |
| 2: | **Transpose** $f$ + halo in $\varphi$; | |
| 3: | 2D *aligned* advec. (**comm.**) in $(\theta,\varphi)$ | $(\forall(\mu,\varphi,v_\parallel) = [local], \forall(r,\theta) = [*])$; |
| 4: | 2D advection in $(r,\theta)$ | $(\forall(\mu,\varphi,v_\parallel) = [local], \forall(r,\theta) = [*])$; |
| 5: | 2D *aligned* advec. (**comm.**) in $(\theta,\varphi)$ | $(\forall(\mu,\varphi,v_\parallel) = [local], \forall(r,\theta) = [*])$; |
| 6: | **Transpose** $f$ + halo in $\varphi$; | |
| 7: | 1D advection in $v_\parallel$ | $(\forall(\mu,r,\theta) = [local], \forall(\varphi, v_\parallel) = [*])$; |

**Algo. 18:** New *aligned* Vlasov solver

In order to depart from the original algorithm to accommodate the aligned strategy, one can list the different constraints that must be taken into account. First, to use the aligned advection approach in the $(\theta,\varphi)$ plane, it is of utmost importance to treat these two directions in a single step. Second, 2D advections in $(r,\theta)$ can not be suppressed or transformed into a simple advection along the $r$ direction, because the non-linear terms in $r$ and $\theta$ interact tightly (displacements in $\theta$, $r$ are intertwined). Third, to evaluate a new algorithm and a new Strang splitting, we should not undermine the existing parallelization strategy (to keep it simple within the GYSELA code).

The proposed Algo. 18 fulfills these constraints. The advections along $v_\parallel$ are unchanged. The aligned advections along $(\theta,\varphi)$ (lines 3, 5) replace the previous advections along the $\varphi$ direction. A first part of the advective terms along the $\theta$ direction are treated line 3 in the new aligned advection along $(\theta,\varphi)$. The 2D advection along $(r,\theta)$ (line 4) is modified in order to keep the other terms (second complementary part) in the $\theta$ direction. Compared to the standard algorithm, the communications included in the 2D aligned advection constitute a reasonable overhead. Another overhead comes from the computation of the 2D interpolations themselves that replace 1D interpolation along $\varphi$ direction. A sketch of the 2D aligned interpolation is given in Algo. 19. The solution presented here improves the previously published solution [1] that induced more memory/computation/communications costs. Indeed, some extra steps, *e.g.* reassemby of $(\theta = *, \varphi = *)$ planes, have been removed. In the new version Algo. 19, ghost zones are transfered along $\varphi$ direction in order to compute locally interpolations during $(\theta,\varphi)$ advections avoiding extra transposition communication steps. Typically, we use Lagrange $5^{th}$ polynomial for the 1D aligned interpolation in Algo. 15. It follows that we work with 3 or 4 ghost points at each local $\varphi$ boundary. In practice, each MPI process initiates a communication

at beginning of Algo. 19 with its two neighbors along $\varphi$ direction to retrieve $f$ values in the form $f\big((\mu, v_\parallel) = [local], (r, \theta) = [*], \varphi = value\big)$.

We then have a robust parallel solution that does not require an entire overhaul of the GYSELA code. Nevertheless some extra communications are created that we measure in the following discussion. We now describe in detail the equations to be solved at each step of Algo. 18.

**Input** : $f\ ((\mu, \varphi, v_\parallel) = [local], (r, \theta) = [*])$, displacement fields
**Output** : $f^\dagger\ ((\mu, \varphi, v_\parallel) = [local], (r, \theta) = [*])$

**Transfer ghost zones** of $f$ along $\varphi$ directions for the local subdomain (typically 3 or 4 extra points at each $\varphi$-boundary);
**for** *all local points* $P$ *in* $(\mu, \varphi, v_\parallel) = [local], \forall (r, \theta) = [*]$ **do**
  $P^\star \leftarrow$ get foot of the characteristics of point $P$ (using displacement fields);
  **if** $P^\star$ *is inside the local domain;*
  **then**
    $f^\dagger(P) \leftarrow$ 2D interpolation along $(\theta, \varphi)$ of $f(P^\star)$;
  **else**
    **send** $P^\star$ coordinates to responsible process and **get** $f(P^\star)$ asynchronously;
**Wait** all pending asynchronous communications

**Algo. 19:** New *aligned* advection along $(\theta, \varphi)$

Following [7] we define $\mathbf{b}^*(\mathbf{x}, v_\parallel) := \mathbf{B}/B^*_\parallel + (v_\parallel/B^*_\parallel)(\mathbf{J}/B)$, where $\mathbf{J}(\mathbf{x}) = \nabla \times \mathbf{B}$ is the equilibrium plasma current, and reformulate the phase-space flow field as

$$\mathbf{u}(t, \mathbf{x}, v_\parallel, \mu) = v_\parallel \mathbf{b}^* + \mathbf{v}_D + \mathbf{v}_E, \tag{3.24a}$$

$$a_\parallel(t, \mathbf{x}, v_\parallel, \mu) = -\mathbf{b}^* \cdot \nabla\Big(\mu B + J_0\, \phi\Big) + \frac{v_\parallel}{B}\, \mathbf{v}_E \cdot \nabla B, \tag{3.24b}$$

where $v_\parallel \mathbf{b}^*$ represents the streaming velocity, $\mathbf{v}_D$ the curvature drift velocity and $\mathbf{v}_E$ the $\mathbf{E} \times \mathbf{B}$ drift velocity:

$$\mathbf{v}_D(\mathbf{x}, v_\parallel, \mu) := \frac{v_\parallel^2 + \mu B}{B^*_\parallel B}\, \mathbf{b} \times \nabla B, \qquad \mathbf{v}_E(t, \mathbf{x}, v_\parallel, \mu) := \frac{1}{B^*_\parallel}\, \mathbf{b} \times \nabla J_0\, \phi.$$

After normalization, the 5D gyrokinetic Vlasov equation with flow field is expressed with toroidal coordinates $(r, \theta, \varphi)$ and decomposed into three separate advection equations:

*A.* 1D advection along $v_\parallel$, which is left untouched compared to the original algorithm,

$$\partial_t f + a_\parallel \partial_{v_\parallel} f = 0;$$

*B.* 2D advection on a magnetic flux surface $(\theta, \varphi)$, where field-aligned interpolation is used,

$$\partial_t f + \big[v_\parallel \mathbf{b}^* \cdot \nabla\theta\big]\, \partial_\theta f + \big[(v_\parallel \mathbf{b}^* + \mathbf{v}_D + \mathbf{v}_E) \cdot \nabla\varphi\big]\, \partial_\varphi f = 0;$$

*C.* 2D advection on a poloidal plane $(r, \theta)$, where a subset of terms along $\theta$ are retained,

$$\partial_t f + \big[(v_\parallel \mathbf{b}^* + \mathbf{v}_D + \mathbf{v}_E) \cdot \nabla r\big]\, \partial_r f + \Big[(\mathbf{v}_D + \mathbf{v}_E) \cdot \nabla\theta\Big] \partial_\theta f = 0.$$

An illustration is given in Fig. 58, where we plot the streamlines for the advection field of Equation $B$, in the limit of vanishing electric field ($\mathbf{v}_E = 0$). We consider the same geometrical parameters of the test-cases in the next section (namely $a = 40$, $[r_{\min}, r_{\max}] = [0.1\, a, a]$, and $R_0 = 3\, a$), and we set $q_\varnothing = 1$ (safety factor at magnetic axis $r = 0$) and $q_a = 2.5$ in our parabolic safety factor profile $q(r)$, which yields a local value $q(r_p) = 1.45375$. For brevity we set $\mu = 0$, select a single magnetic flux surface at $r = r_p$, and focus on four different values of $v_\parallel \in \{-2, -1, 1, 2\}$. We recall that all velocities are normalized to the thermal velocity $v_{T_0}$. For comparison we also plot the streamlines of the magnetic field passing through the same points at $\varphi = 0$: as expected, the misalignment between the advection velocity and the magnetic field grows with $v_\parallel$, but remains reasonably small throughout this range of meaningful velocities. Finally, in the background is shown a set of straight lines with $d\theta/d\varphi \equiv \iota(r_p)$ (as shown previously in Fig. 54), which approximates the magnetic field on this flux surface: these are used for field-aligned interpolation of the distribution function $f$.

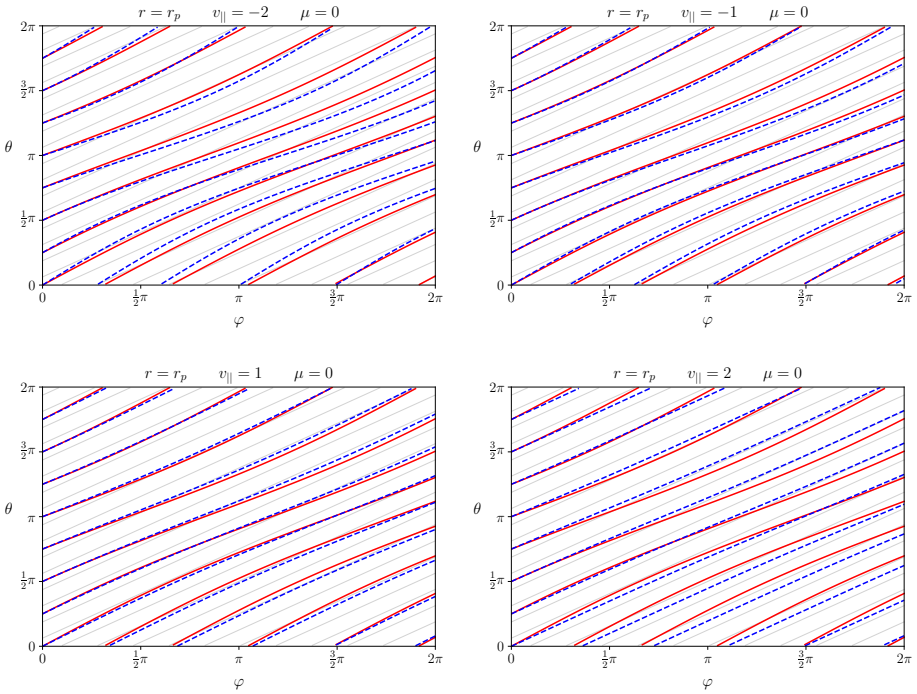



Figure 58: Equation $B$ in GYSELA's operator splitting (aligned advection in $(\theta, \varphi)$ coordinates): we compare the streamlines of the advection field assuming $\mathbf{v}_E = 0$ (dashed blue lines) to the streamlines of the magnetic field (solid red lines). In the background we plot a set of straight lines with $d\theta/d\varphi \equiv \iota(r)$ (solid gray lines) that approximate the magnetic field on this flux surface. For this comparison we have chosen $r = r_p$ and $\mu = 0$, and we calculate the advection field for four different values of $v_\parallel \in \{-2, -1, 1, 2\}$. The safety factor is $q(r_p) = 1.45375$.

### Numerical results

In order to have accurate and converged simulations, in this section we use a setup with a relatively large value of $\rho^\star = 1/40$ ($\rho^\star$ beeing the gyroradius normalized to the plasma size), and we consider a single $\mu$-value of $\mu = 0$. We investigate two physical cases with geometrical parameters

$$a = 40, \quad r_{\min} = 0.1\,a, \quad r_{\max} = 1.0\,a, \quad R_0 = 3\,a\ ,$$

that differ in their safety factor profiles $q(r)$. The parallel velocity domain is truncated at $v_{\max} = 6.3$. Benchmarks have been realized with the 4D toroidal version of the GYSELA code, on a fine computational domain of size

$$N_r = 256, \quad N_\theta = 256, \quad N_\varphi =< not\ fixed >, \quad N_{v_\parallel} = 48.$$

In order to keep the time integration error low compared to interpolation errors, a small time step $\Delta t = 1$ was chosen.

A first case with an almost constant safety factor $q(r)$, slowly varying between $q(0) = 1$ and $q(a) = 1.1$, is illustrated by Figs. 59, 61, and 62. A second case with a safety factor strongly depending on $r$, varying between $q(0) = 1$ and $q(a) = 2.5$, is illustrated by Figs. 60, 63, and 64. The second case could be slightly more difficult to handle for the aligned approach, because the $\mathbf{b}$ direction depends on the $r$ position and on $q(r)$. Indeed, for each hyper-plane at a given $r$, the aligned advection algorithm uses possibly a different direction than for another $r$ value. Figs. 59 and 60 report the time evolution of the electrostatic energy (integrated over the domain). Figs. 61 and 63 show the electrostatic energy on the magnetic flux surface at $r = 127\Delta r \approx r_p = 0.55a$, at time $t = 1672$. Figs. 62 and 64 show the electrostatic energy on the poloidal plane at $\varphi = 0$, also at time $t = 1672$.

One can see on Fig. 59 that the standard approach with $N_\varphi = 128$ gives a similar result compared to the aligned method with $N_\varphi = 32$. The two other curves with standard method and $N_\varphi = 32$ and $N_\varphi = 64$ are not converged along the $\varphi$ direction and give substantially different potential energy dynamics. Figs. 61 and 62 corroborate this fact by showing different cuts of the electric potential. In Fig. 61, the two graphs at middle and bottom position show quite identical structures. It is important to notice that we have reconstructed finely the graph with $N_\varphi = 32$ in order to recover a fine resolution on the plots (through 4 aligned interpolations per original grid point, leading to a virtual $N_\varphi = 128$). In order to do that, we use Algo. 15 with $(\theta^\star, \varphi^\star)$ being the grid points on the fine mesh. As stated in

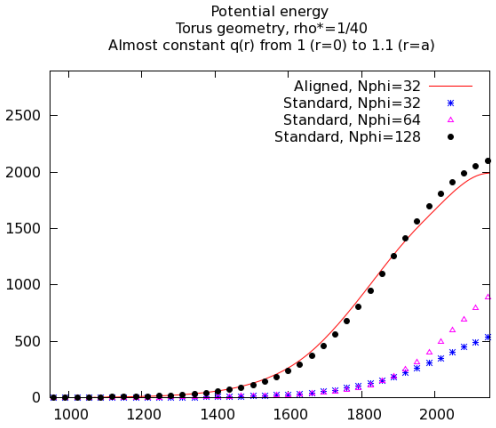



Figure 59: Potential energy plots for aligned or standard strategies. Toroidal configuration with almost constant safety factor along $r$.

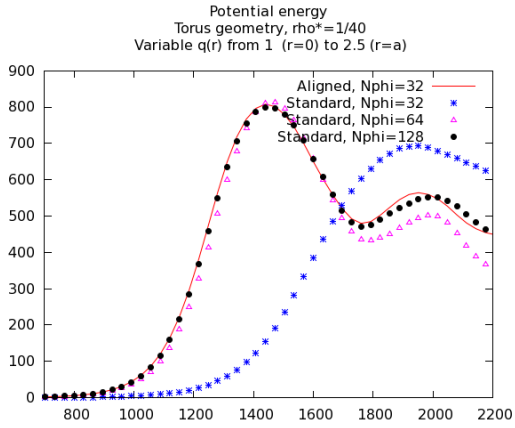



Figure 60: Potential energy plots for aligned or standard strategies. Toroidal configuration with safety factor depending on $r$ coordinate.

Section 3.1 (see also [7]), global conservation of mass and energy in toroidal geometry is quite difficult to achieve in practice, due to boundary conditions within GYSELA. Therefore, these quantities have not been used to estimate the benefits of the aligned method here.

Figs. 60, 63 and 64 show results for the second simulation with a strongly varying safety factor. Conclusions are quite analogous to the first simulation. On the left-hand side, one can see elongated structures along the parallel direction, which constitute the rationale that justifies why the aligned method reduces interpolation approximation errors. For these two simulations, we conclude that the aligned approach works well and permits to reduce by a factor of 4 the number of grid points in the $\varphi$ direction for these cases at $\rho^\star = 1/40$ ($\rho^\star$ beeing the gyroradius normalized to the plasma size). We also expect that, as $\rho^\star$ is further reduced to approach the ITER values of the order of $10^{-3}$ [104], the benefit of aligned method will increase. Thus, our method could allow larger savings in the number of grid points along toroidal direction when employed in the context of realistic simulations of reactor scale devices.

### Execution times comparison

As a matter of comparison between the standard and aligned methods, Table 12 gives typical execution times of GYSELA for four short runs that employ the same configuration and grid size already described in Section 3.2.4 ($N_r = 256$, $N_\theta = 256$, $N_{v_\parallel} = 48$). For the aligned scheme we take $N_\varphi = 32$, while for the standard scheme we consider three different simulations with $N_\varphi \in \{32, 64, 128\}$. The time breakdown of specific regions of the code are shown in addition to the total run time.

| Execution Time | Aligned $N_\varphi = 32$ | Standard $N_\varphi = 32$ | Standard $N_\varphi = 64$ | Standard $N_\varphi = 128$ |
|---|---|---|---|---|
| advections | 38.2 | 17.0 | 35.0 | 68.8 |
| transposes + advec. comm. | 10.1 | 5.9 | 11.9 | 22.6 |
| others | 17.3 | 15.8 | 25.5 | 45.0 |
| total run time | 65.8 | 38.6 | 72.4 | 136.4 |

Table 12: Time (in seconds) of a short GYSELA run in the same configuration described in Section 3.2.4.

Let us compare the timings for the aligned and standard methods at $N_\varphi = 32$. Firstly we observe that the execution times for the transposes and advection communications are higher with the aligned scheme. This is explained by the fact that i) there are ghost zones along $\varphi$ direction (Algo. 19) required in addition to the original algorithm, ii) there are several additional communications to transmit feet of characteristics and asynchronous send/receive of distribution function values. The advection steps are also more expensive with the aligned scheme, because two 1D advections along $\varphi$ per call to the Vlasov solver are replaced by two 2D aligned advections along $(\theta, \varphi)$.

Nevertheless, one can see that the aligned strategy with $N_\varphi = 32$ is already competitive against the standard approach with $N_\varphi = 64$ in terms of total run time, with the big benefit of requiring approximately two times less memory to store the distribution function. Section 3.2.4 has shown that the aligned approach with $N_\varphi = 32$ is more accurate than the standard approach with $N_\varphi = 64$ (at least in the linear phase), we can conclude that there is a clear gain of field-aligned interpolation in GYSELA. This aligned approach will be helpful to handle kinetic electron simulations.

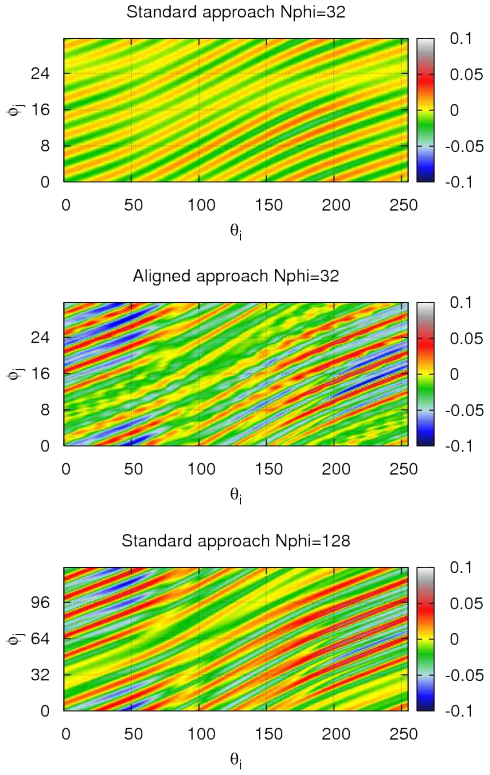



Figure 61: Toroidal configuration with $q(a) = 1.1$: flux-surface cross-section of electric potential at $r \approx r_p$ and $t = 1672$. Standard simulation with $N_\varphi = 32$ (top), Aligned simulation with $N_\varphi = 32$ (middle), Standard simulation with $N_\varphi = 128$ (bottom).

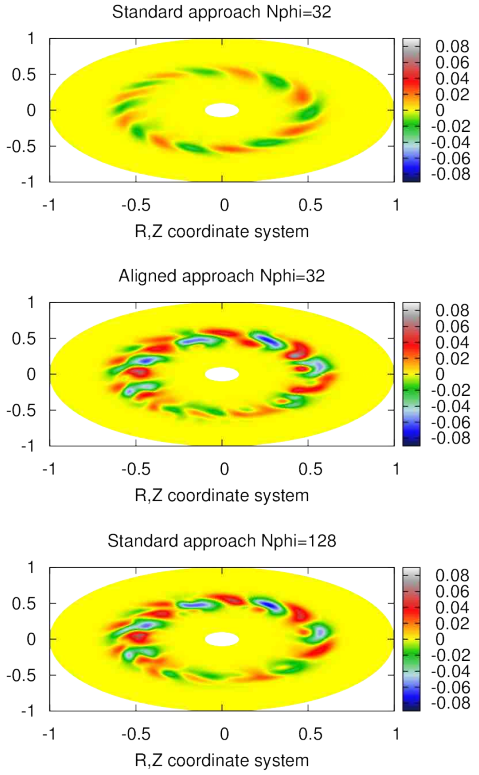



Figure 62: Toroidal configuration with $q(a) = 1.1$: poloidal cross-section of electric potential at $\varphi = 0$ and $t = 1672$. Standard simulation with $N_\varphi = 32$ (top), Aligned simulation with $N_\varphi = 32$ (middle), Standard simulation with $N_\varphi = 128$ (bottom).

### 3.2.5 Conclusion

We have described a semi-Lagrangian method based on field-aligned interpolation, for the solution of the gyrokinetic Vlasov equation. The application of interest is the numerical simulation of magnetically confined plasmas in fusion devices. Thanks to the smooth variation of the solution in the direction of the magnetic field, field-alignment enhances the accuracy of the interpolation. Then, considering a given level of accuracy, this allows one to reduce the number of discretization points along the toroidal direction $N_\varphi$.

The scheme were implemented into two semi-Lagrangian codes, Selalib and Gysela, mainly considering the 4D gyrokinetic Vlasov equation in the zero-Larmor-radius limit, but also on standard gyrokinetic simulations (results not presented here). In our benchmarks against the standard (non-aligned) scheme, we have observed large reductions in memory footprint, as well as a reduction of number of points needed along toroidal direction. Our estimates [1] suggest that these gains will be even larger in reactor-scale simulations (small $\rho^\star$ value).

Field-aligned interpolation does not pose any constraint on the 2D poloidal grids, and the use of magnetic flux coordinates is not necessary. Accordingly, the magnetic axis, as well as the X-point in a divertor configuration, do not pose theoretical problems. Therefore the presented algorithms are able to be extended to more complex magnetic geometries.

## 3.3 Realistic geometry in the poloidal plane

### 3.3.1 Aim for describing complex geometry

A tokamak is a toroidal device for plasma confinement in which a strong toroidal magnetic field is imposed by a system of external coils, while a poloidal magnetic field is generated by a strong toroidal current flowing through the plasma. The sum of these toroidal and poloidal fields results in a helical geometry of the magnetic field lines. In the region we are interested in, which is the confined plasma in the core of the tokamak, the magnetic field lines are closed and are wrapped around closed surfaces.

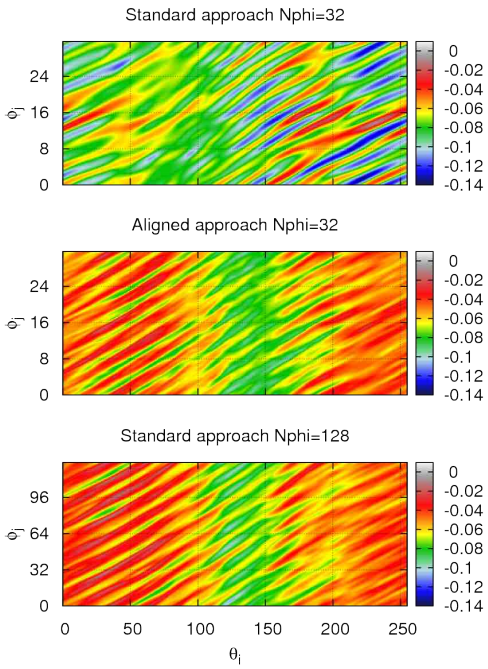



Figure 63: Toroidal configuration with $q(a) = 2.5$: flux-surface cross-section of electric potential at $r \approx r_p$ and $t = 1672$. Standard simulation with $N_\varphi = 32$ (top), Aligned simulation with $N_\varphi = 32$ (middle), Standard simulation with $N_\varphi = 128$ (bottom).

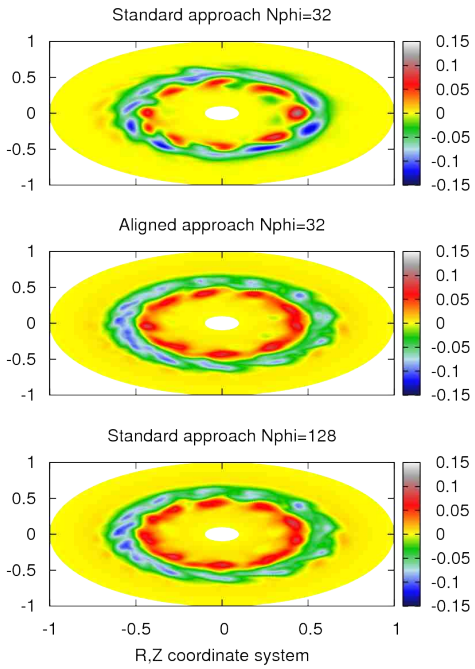



Figure 64: Toroidal configuration with $q(a) = 2.5$: poloidal cross-section of electric potential at $\varphi = 0$ and $t = 1672$. Standard simulation with $N_\varphi = 32$ (top), Aligned simulation with $N_\varphi = 32$ (middle), Standard simulation with $N_\varphi = 128$ (bottom).

These so-called *magnetic surfaces* are nested around the plasma center and can usually be considered axisymmetric and described by a constant section around the torus. The shape of the magnetic surfaces has a strong impact on the physics involved in the confinement of the plasma [99]. Therefore, in order to accurately describe the transport processes in a tokamak plasma, a fine description of the geometry of magnetic surfaces is helpful to reduce computation costs or improve accuracy.

The strong magnetic field in a tokamak means that the motion of particles will be restricted in the direction perpendicular to the magnetic field lines, but free along this (so-called *parallel*) direction. Moreover, the large perturbations along the magnetic field lines play an important role in transport processes and must be included in the model. Thus, it appears that even small discretization errors can corrupt numerical results, for instance by causing a parallel heat flux to leak into the transverse direction [137]. These issues are particularly critical when modeling nonlinear phenomena such as turbulence, which is frequently studied in the fusion community through the development of gyrokinetic codes [132]. As a first approach, many gyrokinetic codes adopted a simplified description of the geometry of magnetic surfaces, for instance using basic polar coordinates, which implies that realistic tokamak geometries were not taken into account. Since several years, an ongoing effort has been initiated to include more realistic geometries. As a long-term objective, we expect to design a new Vlasov solver for GYSELA which would extend the code's capability to perform simulations in complex tokamak geometry (within the last closed magnetic surface for X-point, double null, or D-shaped plasma) while providing a fine description of this geometry.

### 3.3.2 Reduced setting for testing diffeomorphism

Classical finite element method (FEM) with straight lines describing the edges are not adapted to the complex geometry of magnetic surfaces in a tokamak, where curved elements appear necessary for an accurate description of these surfaces. In order to advance towards the long-term objective of designing a gyrokinetic Vlasov solver in complex geometry, we worked on the feasibility of using *diffeomorphism/mapping* in a simplified setting. The Vlasov solver is implemented on a simplified and reduced phase space (1D in space and 1D in velocity), instead of the 5D phase space of gyrokinetic codes. This reduced model is directly relevant for GYSELA, as *mapped* meshes would be used – at least as a first step – to generate the 2D meshes describing magnetic surfaces in the poloidal plane. We consider a Vlasov-Poisson problem modeling the focusing of a heavy ion beam in an axisymmetric

geometry around its optical axis. The advantage of this basic model is that numerical experiments can be performed in a small-sized phase space domain using simple Dirichlet boundary conditions. Since the solution remains very localized, one can easily investigate different mesh geometries with flexibility concerning boundary conditions. Moreover, as in the case of gyrokinetic simulations, small-scale filamentation develops in phase-space, which must be correctly captured by the solver independently of the chosen mesh. The aim of the paper we wrote [27] and briefly summarized here was to validate numerically the Vlasov solver for different underlying meshes, as well as to measure the impact of the chosen mesh on the computational costs.

Diffeomorphisms and mapping techniques provide methods to go from a reference coordinate system (*e.g.* $(s,t) \in [0,1]^2$) to a physical coordinate system (*e.g.* $(x,y) \in \Omega$). To do so, one has to choose a *shape function* or a *mapping* to go from the reference system to the physical system. A common example is the use of a polar mapping to map a rectangular grid onto a circular domain. In our context, we choose to work on a uniform grid in the reference coordinate system.

The need of an accurate representation of the geometry is not an exclusive matter of a specific application domain but is quite common in scientific computing. *Diffeomorphisms* associated to finite elements have proved to be useful to deal with complex geometries. The idea is to perform a shape transformation in order to map a computational domain, for example a rectangular grid, to a potentially complex geometrical domain. At a given mesh resolution, a well chosen diffeomorphism adapted to the geometry can reduce numerical errors. In this context, the use of B-splines, NURBS and isoparametric approaches may provide powerful tools. The inconvenient of B-spline mappings is the inverse problem: given a point in the physical domain, what is the corresponding parametric point? Iterative algorithms are needed to perform this inverse mapping. Another idea is to use an analytic inverse, whenever it is possible, for example in the case of a polar grid. The present work focuses on two isoparametric meshes: an oval mesh with an analytic inverse mapping and a mesh defined by Bézier elements which are a specific type of NURBS or B-splines.

Our reference test case is the study of beam focusing using the paraxial Vlasov-Poisson model [128]. This model is common in accelerator physics and describes the propagation of a particle beam along a linear optical axis. While the beam is considered stationary, the propagation velocity in the direction $z$ of the optical axis is assumed constant (thus $z$ is formally replaced by time in the equations). We also assume that the beam is symmetric around its optical axis. Therefore, we solve the Vlasov-Poisson system in cylindrical geometry with the radius $r$ as the only space coordinate and $v_r$ the velocity in the radial direction. In order to avoid issues with boundary conditions around $r = 0$, we consider a symmetric domain in $r$ with the condition $f(-r, v_r) = f(r, -v_r)$ where $f$ is the particle distribution function in phase-space.

Following the normalization in [40], we solve the Vlasov equation coupled with a Poisson equation. We consider the case where a proton beam is focused by applying a periodic external electric field. As an initial distribution function, we will use the semi-gaussian (gaussian in velocity, localized in space) formulation (details are given in [27]). In order to solve the 2D Vlasov-Poisson system in complex geometry, we have used the LOSS code previously implemented [14,9,30], extending it for the paraxial beam test case and for various geometries to a new version named ISOLOSS. The ISOLOSS code uses a semi-Lagrangian numerical scheme, the time integration is performed using a predictor-corrector scheme.

### 3.3.3 Results for ISOLOSS code and diffeomorphisms

#### Analytic mapping

As a first step, we consider the case where the mapping is performed by a fully analytic diffeomorphism. This diffeomorphism maps a rectangle $(s,t) \in [0,1] \times [0, 2\pi[$ (in the reference space) to an ellipse in the physical space. The following parametric equations are taken as a mapping function:

$$x(s,t) = r_{max} \times s \times \cos(t) \qquad y(s,t) = v_{max} \times s \times \sin(t)$$

The inverse mapping is also given by analytical expressions:

$$s(x,y) = \sqrt{\frac{y^2}{v_{max}^2} + \frac{x^2}{r_{max}^2}} \qquad t(x,y) = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \text{ and } y \geq 0 \\ \arctan(\frac{y}{x}) + 2\pi & \text{if } x > 0 \text{ and } y < 0 \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ \frac{3\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \end{cases}$$

The computational cost of moving forward and backward from the reference space to the physical space is quite low. A direct or inverse mapping represents a fraction of the cost induced by one 2D

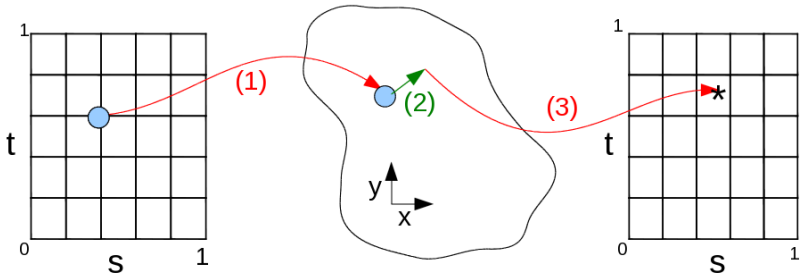



Figure 65: The Semi-Lagrangian method in complex geometry: (1) map the position in the reference space into physical space, (2) follow the characteristic backward in physical space, (3) map the obtained position back in the reference space to perform the interpolation

interpolation. Therefore, as we shall see in the numerical experiments, the overhead in the ISOLOSS code is not so large compared to the original solution that does not require a diffeomorphism.

### Bezier mapping

A Bézier surface (also known as Bézier patch) of order $(n, m)$ is defined by several control points $(\mathbf{k}_{i,j})_{i\in[0,n],j\in[0,m]}$. It maps the unit square $[0,1]^2$ into a surface embedded within a space of the same dimensionality as the control points. In our application, we take the control points $\mathbf{k}_{i,j}$ in $\mathbb{R}^2$ in order to represent a 2D geometry. A Bézier surface is parametric and the equations describing it depend on parameters that are not explicitly part of the geometry. Hence, a point $P$ of coordinates $(s,t)$ on the patch is localized at the following position in the physical space:

$$\mathbf{P}(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{m} \mathcal{B}_i^n(s)\mathcal{B}_j^m(t)\,\mathbf{k}_{i,j} \qquad \text{with} \qquad \mathcal{B}_i^n(u) = \binom{n}{i} u^i(1-u)^{n-i}\ .$$

$\mathcal{B}_i^n(u)$ are known as the Bernstein basis of polynomials of degree $n$. As a first step, we decide to use biquadratic Bézier patches. In our work, the main issue is the inverse mapping of positions from physical space to reference space, whereas the direct mapping from reference space to physical space is very simple. The inverse mapping transformation has to be performed for each grid point during the advection step. There is no generic analytical solution for curved elements and this operation happens to be costly. Several ideas and solutions that we have tested for inverse mapping calculation are inspired from computer graphics papers [134,172]. We typically have chosen to use *Newton* method and *clipping* algorithm in this work but we will not describe them in detail (please see [27]). It is a quite fast iterative method with a few iterations. The semi-Lagrangian method must be adapted to use mapping techniques. In the approach chosen here, we keep the expressions of the advection equations in the physical space, rather than rewriting these equations in the reference space. Therefore, it is necessary to move backward and forward between the reference space and the physical space. Fig. 65 outlines the adapted semi-Lagrangian method when using parametric surfaces.

From a given position $(s,t)$ in the reference space, we map the corresponding position in physical space. The characteristics are then followed backward in physical space, and the foot of the characteristics must be mapped back into the reference space before performing the cubic spline interpolation. The main issue in this algorithm is the inverse mapping of positions from physical space to reference space. This transformation has to be performed for each grid point during the advection step. Another issue when solving the Vlasov-Poisson system on an isoparametric mesh is the computation of velocity integrals to obtain the density, needed to solve the Poisson equation, we will not discuss this point here. There are also ways to reduce the computational costs associated to retrieving the Bézier patch surrouding a given location (see [27]).

### Experimental results

*Numerical results* We consider the evolution of a non stationary beam. The underlying uniform 2D grid used for the simulation consists in $2^{18}$ points covering entirely or partially the spatial domain $(r, v_r) \in [-6,6] \times [-2.5, 2.5]$. We intend to compare the speed and accuracy on three geometries:

a) *Original LOSS*: no diffeomorphism is employed and this case corresponds to the initial simulator configuration. The computational grid is rectangular and its size is $512 \times 512$.

b) *Analytical mapping*: an analytic shaping function maps an oval in phase space to a reference domain $[0,1] \times [0,1]$. The reference domain has a uniform grid mesh of size $1024 \times 256$ with a

larger number of points in the angle direction. Especially in this configuration, the periodicity along the angle direction has to be taken into account.

c) *Bezier mapping*: a set of Bézier patches are built; the shape of the computational domain is in-between the shapes of the previous oval and rectangle configurations. The reference domain has a $512 \times 512$ size.

The different computational grids are sketched in Fig. 66 where an undersampling has been applied in order to improve readability.

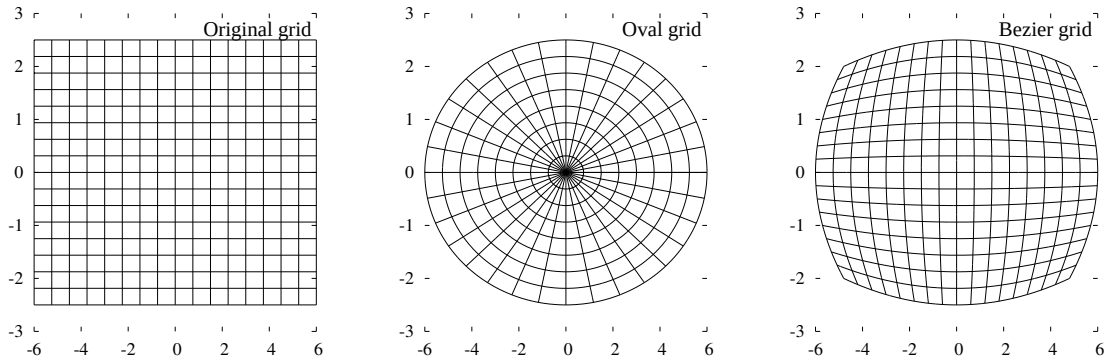
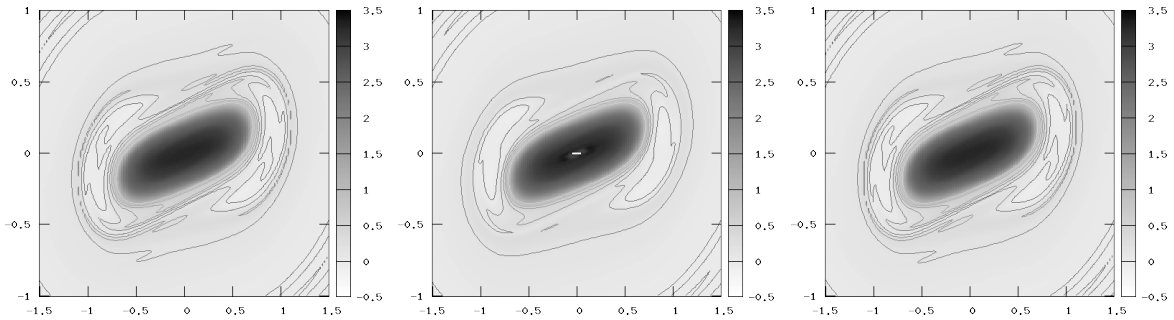



Figure 66: Three computational grids are tested for the paraxial beam problem: (a) regular Cartesian grid, (b) oval grid associated to an analytical inverse mapping, (c) grid defined by Bézier elements



(a) Original (b) Analytic/Oval (c) Bézier

Figure 67: Final state of the distribution function in phase space in the three configurations

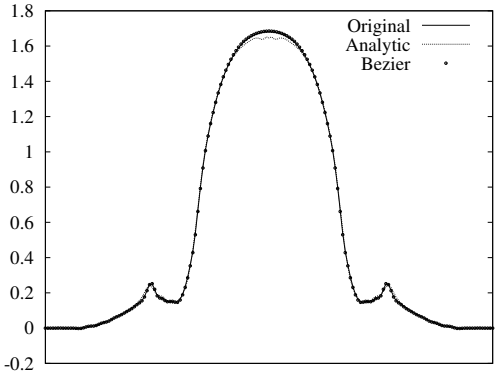



Figure 68: Final state of the density function $\rho(r)$ in the three configurations

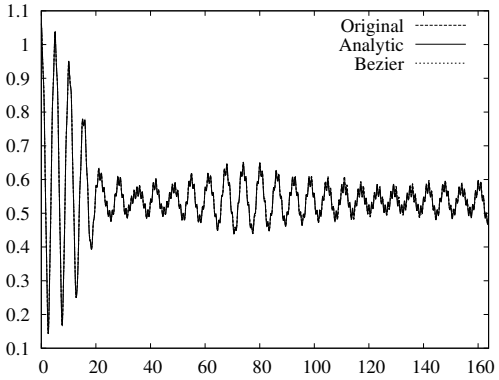



Figure 69: Evolution in time of the $R_{rms}$ quantity in the three geometry configurations

The final state of the distribution function is shown in Fig. 67 for the different computational grids. For each configuration, the simulator manages to follow the evolution of filamentation. The distributions functions are quite similar, but the analytic mapping exhibits some clear differences with the two other settings. The aspect and singularity of the oval mapping close to the center of the domain clearly trigger numerical problems. These problems are also illustrated in Fig. 68 where

the final state of density $\rho(r)$ is presented in the three geometries. The Bézier and the original curves almost overlay, but the analytic curve has a different behavior at the peak of the density ($r = 0$) and on the lateral small shoulders. The $R_{rms}$ quantity is displayed in Fig. 69, it measures the effective beam focalization. The three configurations give nearly identical $R_{rms}$ curves over time. Finally, we conclude that diffeomorphisms do not decrease accuracy directly, but care must be taken to well choose the mapping function.

*Performance issues*

Performance of the different geometry settings has been investigated on a desktop computer. A 3.0 GHz Intel Core 2 Duo E8400 processor was used to run numerical experiments on only one core. In Table 13, the timings of each part of the code for one simulation of 16384 time steps are gathered. The `Field solve` column sums time used to compute the density $\rho$ density and the self-consistent field. The `Spline coeff.` column corresponds to computing spline coefficient before interpolation can be done. The `Advection` column comprises the trajectories computation to find the origins of the characteristics and the interpolation costs. The `Total` column sums the first columns, ignoring the preprocessing phase of the simulation and the diagnostics.

| | Field solve | Spline coeff. | Advection | Total |
|---|---|---|---|---|
| Original LOSS | 2.3 | 16.4 | 78 | 97 |
| Analytical mapping | 10.1 | 18.4 | 115 | 143 |
| Bezier mapping | 9.2 | 17.3 | 275 | 301 |

Table 13: Timings (seconds) for 2 geometry settings and simulation over 1024 time steps

The field solver and $\rho$ calculation take much more time when a mapping is employed. This is due to the fact that without mapping the computation $\rho(r)$ is very light, as it is only a sum of $f(r, v)$ along dimension $v$; whereas with mapping there are extra calculations to be done. The computation of spline coefficients takes nearly the same amount of time in each case. The computation costs are almost identical and cache effects are the critical factor that modulates slightly the costs of computing spline coefficients. Concerning the advection step, the cost of the inverse mapping explains the observed differences in computation time. The analytic inverse mapping involve a 50% time increase compared to the original LOSS code, and Bézier inverse mapping leads a 250% time increase.

If we look forward to the application of this technique in the GYSELA code, this overhead must be rescaled. The 2D interpolations represents approximately 5%-10% of the execution time in a GYSELA run. A rough estimate of the overhead induced by implementing the Bézier setting in the GYSELA Vlasov solver is thus $13\% - 26\%$. Other overheads should also be considered for the field solver using the upgraded geometry setting, but no simple estimate of this cost can be proposed.

The semi-Lagrangian scheme combined with isoparametric analysis successfully solves the Vlasov equation on a reduced beam test case. Small-scale filamentations in phase-space are well captured by the new solver. Quantitative analysis shows that only minor issues in accuracy are caused by the mesh geometry, except if domain is too much refined close to $r = 0$. Although the overhead in term of execution time may appear quite large, it can in fact be reasonable if the advantages brought by a more accurate description of the domain geometry outweigh this penalty. In the following, we expect to overcome the main issues remaining: adapted solution near magnetic axis $r = 0$, improved analytic mapping to have realistic plasma configuration and low inverse mapping overhead. In addition, we target to have the grid lines truly aligned with magnetic surfaces (the same as we have currently in GYSELA). It prevents spurious flows along radial direction that may happen if the radial and angle directions are mixed in the poloidal plane. Another point that will be under consideration in the next part is that a non-uniform grid and dedicated solution can help to solve problems arising at $r = 0$.

### 3.3.4 Avoiding central hole and introducing reduced polar grid

In the present work (described partly in [81]), a new variant for the interpolation method is proposed that can handle the mesh singularity in the poloidal plane at $r = 0$ (polar system is used for the moment in GYSELA). Also, the hole in the center of poloidal plane for $r < r_{\min}$ is suppressed which enhances realism of GYSELA simulations. Then, we introduced a non-uniform meshing of the poloidal plane instead of uniform one in order to save memory and computations. The interpolation method, the gyroaverage operator, and the Poisson solver are revised in order to cope with non-uniform meshes. A mapping (in exactly the same spirit that the one described in the previous paragraph) that establish a bijection from polar coordinates to more realistic plasma shape is used to improve realism. Convergence studies were performed [81] to establish the validity and robustness of our new approach. This study is going further beyond what was presented earlier in p.59 and the following pages. We are now expecting to remove completely the $r_{min}$ boundary condition.

**Avoiding $r = 0$ singularity**

*Original polar mesh*

We fix $N_r$, the number of points in the radial direction and $N_\theta$, the number of points in the poloidal direction. The original polar mesh, as it is defined in GYSELA, is such as $r_i = r_{min} + i\Delta r$ with $r_{min} > 0$, $i \in [\![0, N_r - 1]\!]$, $\Delta r = \frac{r_{\max} - r_{\min}}{N_r - 1}$, and also $\theta_j = \frac{j\,2\pi}{N_\theta}$ with $j \in [\![0, N_\theta - 1]\!]$. It is worth noting that $r_{min}$ and $r_{max}$ act as boundary conditions. For each operator that is applied within the poloidal domain, specific ad-hoc approaches are setup to handle what is happening in the central hole $r \in [0, r_{min}]$. We do not detail here the set of ad-hoc boundary conditions (see [3]).

*Enhancing $r=0$ handling for Field solver*

Following the work of [159], we have set $r_{\min} = \frac{\Delta r}{2}$ and $r_{\max} = r_{\min} + (N_r - 1)\Delta r$ so that $\Delta r = \frac{r_{\max} - r_{\min}}{N_r - 1} = \frac{r_{\max} - \frac{\Delta r}{2}}{N_r - 1}$, which leads to $\Delta r = \frac{r_{\max}}{N_r - \frac{1}{2}}$, and the radial points are $r_i = r_{\min} + i\,\Delta r = (i + \frac{1}{2})\Delta r, \quad i = 0, \ldots, N_r - 1$.

Let us first consider 2D Poisson equation on a disk $\Omega = \{(r, \theta) : 0 < r < r_{\max}$ with $r_{\max} \in \mathbb{R}$ and $0 \le \theta \le 2\pi\}$ where $\Omega$ is described by a uniform circular mesh $\Omega_k$. To overcome the singularity problem at $r = 0$, one of the trick consists in taking $r \in [r_{\min}, r_{\max}]$ with $r_{\min} = \Delta r/2$ and a half-integered grid in radial direction and an integered grid in poloidal direction. We choose to stay with finite differences to solve this problem. In this section, we consider the 2D Poisson equation in polar coordinates on a domain $\Omega$,

$$\frac{\partial^2 f}{\partial r^2} + \frac{1}{r}\frac{\partial f}{\partial r} + \frac{1}{r^2}\frac{\partial^2 f}{\partial \theta^2} = R(r, \theta) \tag{3.25}$$

with Dirichlet boundary conditions $f(r = r_{\max}, \theta) = g(\theta)$ on $\partial\Omega$. Note that the boundary values are defined on the grid points and $f_{i,j} = f(r_i, \theta_j)$, $R_{i,j} = R(r_i, \theta_j)$. Using the centered difference method to discretize, one has:

$$\frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{(\Delta r)^2} + \frac{1}{r_i}\frac{f_{i+1,j} - f_{i-1,j}}{2\Delta r} + \frac{1}{r_i^2}\frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{(\Delta\theta)^2} = R_{i,j} \tag{3.26}$$

but also at $i = 0$, the following equation is solved:

$$\frac{f_{1,j} - 2f_{0,j} + f_{-1,j}}{(\Delta r)^2} + \frac{1}{r_0}\frac{f_{1,j} - f_{-1,j}}{2\Delta r} + \frac{1}{r_0^2}\frac{f_{0,j+1} - 2f_{0,j} + f_{0,j-1}}{(\Delta\theta)^2} = R_{1,j} \tag{3.27}$$

The trick is that the coefficient of $f_{-1,j}$ is canceled in this last equation because $r_0 = \frac{\Delta r}{2}$. The scheme does not need any extrapolation for $f_{-1,j}$, so no pole condition is needed. I integrated a variant with Fourier transform along $\theta$ and second order in $r$ (see [160]) into the field solver of GYSELA with success. We no more have the Dirichlet nor Neumann boundary conditions at $r_{min}$. This eliminates spurious artifacts at this location on the electric potential which is a keypoint for turbulence modeling. Differences of previous and new solution is presented in Fig. 70, 71, 72, 73 with a zoom at a given time step on the region $r = 0$ to highlight the artifacts. It is important to notice that the settings of the two compared simulations are strictly identical, even $r_{min}$ value is exactly the same.

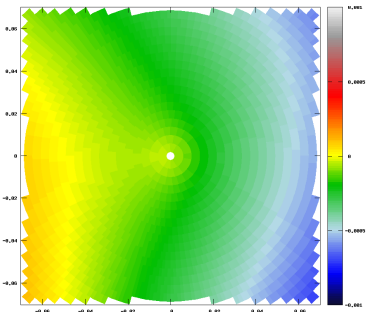


Figure 70: Electric Potential - Neumann at $r_{min}$ (old), artifact in the center

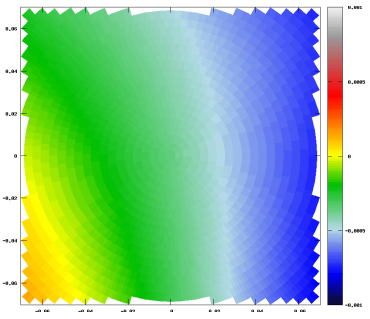


Figure 71: Electric Potential - Lai & Wang trick at $r_{min}$ (new), artifact removed

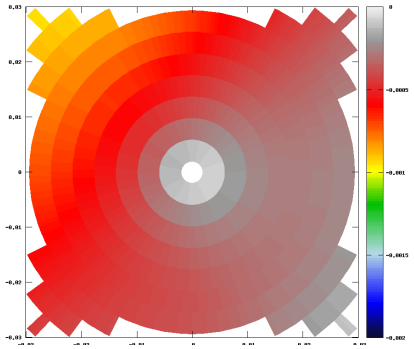


Figure 72: Electric Potential - Neumann at $r_{min}$ (old), artifact in the center

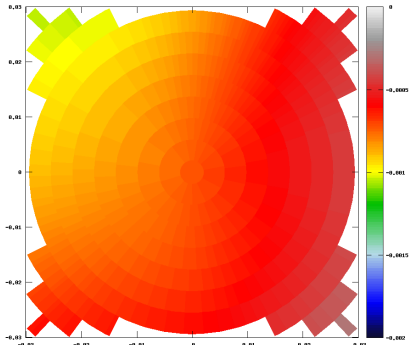


Figure 73: Electric Potential - Lai & Wang trick at $r_{min}$ (new), artifact removed

*Enhancing $r=0$ handling for Vlasov solver*

The issue for $r$ close to 0 within Vlasov solver is mainly located in advection step in the poloidal plane $(r,\theta)$. Indeed, one has to perform interpolations at $r < r_{min}$, but also the calculation of foot location has to deal with an evaluation also at $r = r_{min}$ with reasonable accuracy.
We propose to switch from the usual interpolation scheme near the magnetic axis to a bilinear interpolation along $x, y$ (Cartesian coordinates). Thus, we remove the dependency along $\theta$ direction induced by polar coordinates and then avoid the singularity at $r = 0$. In order to go from usual interpolation for $r \gg 0$ to bilinear interpolation near $r = 0$, we use a simple linear combination. The weighting coefficients $w_{center}(r)$, $w_{large\_r}(r)$ for this linear combination depend on $r$ value. We impose the constraint $w_{center}(r) + w_{large\_r}(r) = 1, \forall r$.

$$\begin{aligned} interpolate(r,\theta) = & w_{center}(r) \;\; bilinear\_interpolate(r,\theta) \;\; + \\ & w_{large\_r}(r) \;\; 2D\_cubic\_splines\_interpolation(r,\theta) \end{aligned}$$

The kind of weighting functions we employ are shown in Fig. 74. The bilinear interpolation is based on only 4 values located at $(r_1, \pi/8)$, $(r_1, 3\pi/8)$, $(r_1, 5\pi/8)$ and $(r_1, 7\pi/8)$.

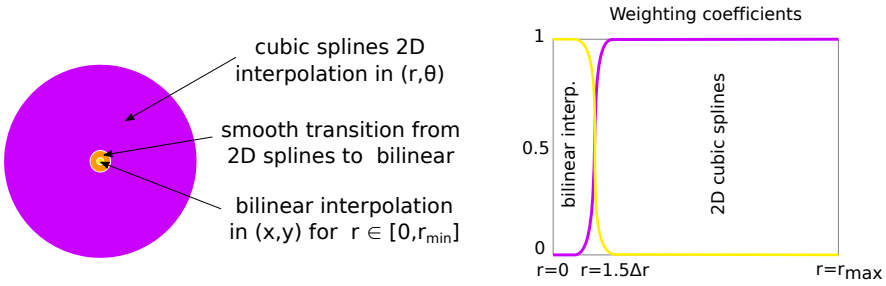



Figure 74: Linear combination of two interpolation operators

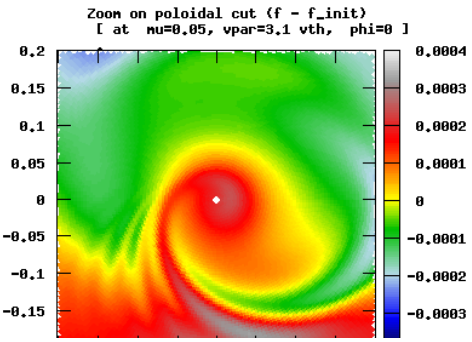



Figure 75: No interp. in $[0, r_{min}]$ (old), artifact at $r = 0$

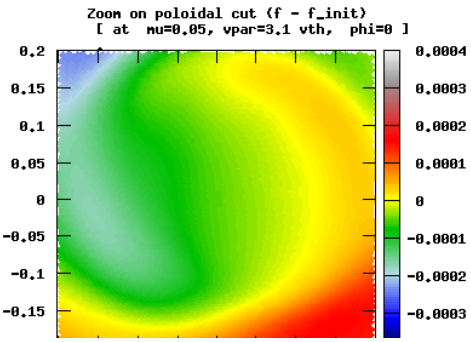



Figure 76: Interp. in $[0, r_{min}]$ (new), expected behavior at $r = 0$

Some illustrations of the new solution with mixed interpolators are given in Figs. 75, 76. Let us notice than the simulations performed use the new methods both in Vlasov and Poisson solvers. The improvements are valuable because one observes a unpolluted solution near the center with usual GYSELA settings. Nevertheless, the behavior at $r = 0$ could be even better if we invest in a higher order interpolator in the center. This solution has been integrated and adopted by the GYSELA users.

It removes many issues associated with artificial boundary conditions that was set at $r_{min}$ in the past. For example, physicists have no more needs for artifical diffusion operator along $r$ and $\theta$ direction in a buffer region sticked to $r_{min}$ boundary.

**Lagrange interpolants**

Semi-Lagrangian solvers need an interpolation scheme as an inner component. In the literature, meteorology community for weather forecast models and plasma physics community have investigated a number of numerical schemes. Most commonly, cubic splines is employed in practice. However, this approach induces 1) costs due to the LU decomposition which are not trivial to vectorize on modern architectures and 2) a synchronization between spline coefficient derivation and effective interpolations that one may want to avoid. Alternate approaches as high-order Lagrange interpolants have started gaining interest recently. Such a local and compact stencil method fit well the current processor architectures. Hence, computations tend to be cheaper and cheaper in comparison to memory accesses and FLOPs achieved by high-order methods tends to increase along with the order [157,176]. Then one can afford a bit more computations if the number of memory accesses remains low. But high-order (order larger than 4) are expected in order to get closer to spline accuracy [127]. Nevertheless, very high-order can lead to other problems especially with data that include sharp gradients or high frequency in space phenomena: Runge's phenomenon, spurious oscillations, or artificial undershoot/overshoot. These kinds of issues happen less frequently with cubic splines that tends to foster smoothness because of $\mathcal{C}^1$ regularity. Furthermore, as soon as spurious oscillations appear, they tend to remain in the computational domain if no specific processes amortize them.

Lagrange interpolants have been tested within the advections of Gysela. Mainly we considered polynomials of degree 5 and 7 that are the best compromises so far. Let $g$ be a discrete function (defined on $x \in [x_q, x_r]$).

$$L^{qr}(x) = \sum_{j=q}^{r} \mathcal{L}_j^{qr}(x), \qquad \mathcal{L}_j^{qr}(x) = g(x_j) \prod_{k=q,\ k!=j}^{k=r} \frac{(x - x_k)}{(x_j - x_k)}$$

One has the following properties:

- $\forall j \in [q, r], \quad L^{qr}(x_j) = g(x_j)$,
- One considers $q - r + 1$ points (6 or 8) for $L^{qr}$, degree of the polynomial is $q - r$ (5 or 7)

With tensor product, we have also access to 2D interpolation for the 2D advections along $(r, \theta)$. Assuming the cache memory is large enough, one can estimate the average cost of the interpolation needed for each grid point in each advection. In Table 3.3.4, the needs in term of floating-point operation is summarized, whereas Table 3.3.4 gives some execution times associated with the different interpolation choices (domain size was $256 \times 256 \times 128 \times 128 \times 8$, number of nodes is 64 on INTEL Broadwell or INTEL KNL - Marconi machine - 2017).

| Kind of interpolation | Mem. load | Mem. store | Multiply | Add | Divide |
|---|---|---|---|---|---|
| 1D spline | 1 | 1 | 26 | 16 | 1 |
| 1D Lagrange 8-pts | 1 | 1 | 44 | 35 | 0 |
| 1D Lagrange 6-pts | 1 | 1 | 30 | 25 | 0 |
| 2D spline | 1 | 1 | 60 | 40 | 2 |
| 2D Lagrange 8-pts | 1 | 1 | 144 | 124 | 0 |
| 2D Lagrange 6-pts | 1 | 1 | 90 | 74 | 0 |

| | Broadwell machine | | | KNL machine | | |
|---|---|---|---|---|---|---|
| Interpolation | 1D advections | 2D advections | Total time | 1D advections | 2D advections | Total time |
| spline | 109 | 120 | 615 | 175 | 142 | 842 |
| Lagrange 8-pts | 72 | 67 | 527 | 92 | 90 | 720 |
| Lagrange 6-pts | 62 | 61 | 513 | 80 | 84 | 716 |

The timings show that Lagrange interpolants permit to reduce the execution times compared to the splines. The Lagrange 6-pts saves more than 45% (Broadwell and KNL) on advection timings versus splines. This result is quite counterintuitive if one considers the costs in term of floating-point operations (Table 3.3.4). Nevertheless, the algorithm for Lagrange approach is much simpler and needs less intermediate data structures than the splines do. In addition, vectorization by the compiler of the most inner loop is much easier with Lagrange, and this is a key element on modern architectures.

**Introducing reduced/non-uniform grid**

Changing the mesh of the poloidal plane while keeping a polar coordinate system should allow us first, to loosen the meshing in order to reduce the typical concentration of points near the center $r = 0$ and second, to have the mesh match more closely the magnetic surfaces of the plasma. We should then have an improvement in execution time by reducing the overall number of points as well as an improvement in accuracy thanks to the grid being closer to the typical pattern of simulated phenomena. The non-uniform meshing will also allows us to focus on a specific location of the plane that we want to solve by using more points there and only solving roughly elsewhere (for specific physics experiments).

*New non-uniform polar mesh*

The new poloidal grid that we want to use is sketched in Fig. 77. The idea is to have, for each different circle labeled by $r$ coordinate, a different number of points in the $\theta$ direction. For each one of the $r_i$ we choose a number of points along $\theta$: $N_{\theta_{[i]}}$, and a grid spacing: $\Delta_{\theta\,[i]}$. For instance, in Fig. 77 (p.88), the first layer (inner circle) has four points $N_{\theta_{[1]}} = 4$, the second to fourth layers have eight points $N_{\theta_{[2,3,4]}} = 8$ and the remaining layers have sixteen points $N_{\theta_{[5-7]}} = 16$. This allows either to have a density of grid point which is nearly uniform, or to model finely a subset of the plane which is better solved with more grid points. This meshing or quite similar approaches have already been used in a set of papers [149,168,169], but the settings or equations were different. Therefore, we have mainly only retained the meshing strategy while redesigning the operators.

*Mapping*

The previous approach can be combined with a general mapping, the polar mapping being only a special case. We focus here on mappings with analytical formula and whose inverse can also be expressed by a formula (to shorten execution time) which was one of the concluding points of [27]. This is of course the case for the polar mapping, but we can also find other more general cases, that can have relevance for the description of the geometry of a tokamak. We consider here the case of a large aspect ratio tokamak equilibrium, and the mapping that derives from it, as in [100,130]. For a large aspect ratio mapping [100], one can take the formula

$$\begin{aligned} x &= R_0 + r\cos(\omega) - \delta(r) - E(r)\cos(\omega) + T(r)\cos(2\omega) - P(r)\cos(\omega) \\ y &= r\sin(\omega) + E(r)\sin(\omega) - T(r)\sin(2\omega) - P(r)\sin(\omega), \end{aligned}$$

where $\delta$, $E$, $T$ stand for Shafranov shift, elongation and triangularity. The $P$ notation corresponds to a relabeling of the surfaces. To simplify, we take here $P = 0$, $T = 0$, $\omega = \theta$, together with $E(r) = E_0 r$ and $\delta(r) = \delta_0 r^2$; this clearly restricts the range of accessible geometries, but enables one to get an explicit and simple formula for the inverse. We get

$$\begin{aligned} x &= R_0 - \delta_0 r^2 + (1 - E_0) r\cos(\theta) \\ y &= (1 + E_0) r\sin(\theta). \end{aligned} \tag{3.28}$$

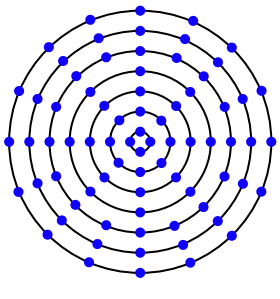


Figure 77: New poloidal grid. The number of points in $\theta$ direction (angle) depends on the radial position (distance to the center).

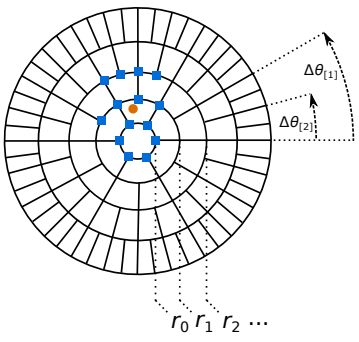



Figure 78: Interpolation of a point ● of the poloidal plane with a stencil of 16 points ■ (Lagrange of order 3).

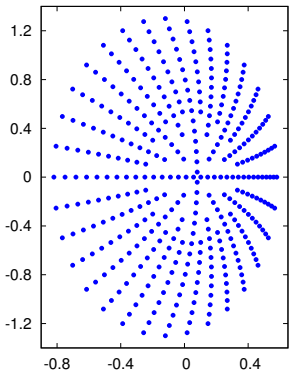



Figure 79: New mapped grid using large aspect ratio equilibrium, with non-uniform meshing along $\theta$.

The inverse mapping can be explicitly given

$$\tilde{y} = \frac{y}{1+E_0},\ \tilde{x} = \frac{x-R_0}{1-E_0},\ \tilde{\delta}_0 = \frac{2\delta_0}{1-E_0}, r = \left( \frac{2\left(\tilde{x}^2+\tilde{y}^2\right)}{1-\tilde{\delta}_0\tilde{x}+\sqrt{(1-\tilde{\delta}_0\tilde{x})^2-\tilde{\delta}_0^2(\tilde{x}^2+\tilde{y}^2)}} \right)^{1/2},$$
$$\theta = \mathrm{atan2}(\tilde{y}, \tilde{x}+\frac{\tilde{\delta}_0}{2}r^2)\,.$$

We refer to [143,144] for some other works concerning the semi-Lagrangian method combined with a mapping. Fig. 79 shows a non-uniform grid combined with this specific mapping.

*2D Interpolation*

The interpolation along $(r,\theta)$ is defined as a tensor-product of 1D interpolation operators. Lagrange interpolation along $r$ and $\theta$ directions is used. We mostly use the value of points located closely to targeted coordinates, but some specific conditions have to be taken into account. If the radial position $r$ goes above $r_{max}$ then a Dirichlet condition is used. If $r$ is located in the interval $r \in [0, \frac{\Delta r}{2}[$, the interpolation scheme has to be adapted because we are crossing the most inner radius of the grid; we basically continue the stencil on the radially opposite side (as if we were considering negative values of $r$). Considering a non-uniform mesh, we need to take into account the cases where the interpolation stencil covers several radii having different number of points along $\theta$. An illustration of the algorithm used is shown in Fig. 78. We always use the closest known points, leading to a good accuracy. To adapt the interpolation calculation to non-uniform meshes, one only need to first perform the interpolation in $\theta$ before the one along $r$ direction. It allows to easily take into account that the number $N_{\theta_{[i]}}$ depends on radius $r_i$. The Algorithm and detailed scheme are given in [81]. Convergence studies have been conducted that demonstrate that the interpolation behaves correctly on a non-uniform mesh with results close to what is observed for uniform mesh (if mesh is well chosen). Then, advection and gyroaverage operators that are based on interpolation are also accurate numerically (additional convergence studies have been done with success in [81]).

*Field solver*

From Eqs. (3.26), (3.27), we have designed a new Poisson solver using finite difference approach along $\theta$ and $r$ for non-uniform meshes (second order in space). A key point to setup this solver was to use Lagrange interpolation to reconstruct missing values because of the non-uniformity of the mesh in the right hand side . The Poisson solver relies now on a sparse solver, and CPU time depends tightly on the performance of the library used. This solution is worthwhile [81], but we expect to improve it in the near future: to reduce execution time and memory footprint, to go to higher order along $\theta$ dimension.

*Discussion*

Several operators of the GYSELA application have been redesigned to cope with non-uniform/reduced grid in the poloidal plane. In addition, a diffeomorphism (mapping technique) has been put into place to model realistic plasma instead of circular configuration. The positive results of this proof-of-concept study justifies that we should try to assemble these new components into a new GYSELAX prototype which is foreseen for 2019. The reduced mesh should subdivide the poloidal plane into a set of tiles with a fixed $\Delta\theta$ in each tile. We wish to have very efficient algorithm on each small tile, *i.e.* non-uniformity should not add negative impact on performance compared to the current version. In addition, we want to move to HPC task-based programming models (instead of OpenMP loop parallelism) and split the computational work into small units of computation called tasks associated to tiles. Each task requires data as input and produces output data. Tasks inputs and outputs are connected according to data dependencies that are explicitly given. In this framework, a runtime is responsible for scheduling task according to their dependencies and the targeted hardware. This approach offers several advantages compared to the usual statically scheduled fork-join model or bulk synchronous model. First, it enables one to express task-parallelism in addition to data parallelism, by decoupling some workloads, load balancing can be improved and execution time be reduced. Second, it eases performance portability in the case of work imbalance between processing units (whether on homogeneous or heterogeneous hardware). Third, it helps removing the costly global or collective barriers in large scale application by providing a way to convey synchronization at micro-scale instead of macro-scale. Fourth, we will be able to specialize some algorithms for specific regions of space (specific tiles) and to embed them into peculiar tasks. Thus, one can have an efficient algorithm for most of the poloidal plane (majority of tiles will have efficient implementations with regular/uniform cells), and some specialized procedure for managing boundary conditions and frontiers in some tiles. Also, the workload that should not be identical among the tasks will be balanced automatically by the runtime.

## 3.4 Physics results over the past years in Gysela

Gysela's physics results over the past years are briefly summarized hereafter (extra information is available in [140]). These achievements would not have been accessible without the joint research and development in the fields of applied mathematics, numerics and HPC. Help and involvement of several partners have been a springboard for improving the code, among them I should mention the strong cooperation with: Bordeaux-Nancy-Strasbourg universities, INRIA (France), IPP (Germany).

Several years ago, the gyrokinetic simulations indicated quantitative differences with the results of a number of reference fluid simulations [125]. If differences and more accuracy of the gyrokinetic approach were to be expected, the importance of the changes – especially for the ITER configuration – indicated that gyrokinetics was an essential model for carrying out simulations of magnetically confined plasmas. Gysela draws on the achievements of the physics group of CEA/IRFM concerning fluid simulations, notably based on the skills gained in designing two previous codes: TRB [133], ETAI3D [173] and TOKAM2D [179]. An important point inherited from former studies is to consider that the controlling parameter of magnetic fusion experiments is the energy flux. In this context, the transport processes are building up on the plasma temperature profile between the center and the periphery. The case is analogous to that of a residential heater for which the gradient of the temperature from the radiator to the outside depends strongly on the quality of the house's heat insulation. The simulations describing this transport impose a so-called *global* approach, that is to say that one has to consider the whole volume of the plasma to be really relevant. Another important component is to represent with enough accuracy a volumetric heat source, this was done through the *flux-driven* approach. Gysela was able to access this setting and associated physics around 2010. In this regime we found in gyrokinetic simulations evidence of avalanche transport [62, 68, 69, 121]. It is characterized by convection of heat on large scales with high intermittency, similar to periodic relaxations.

The study of transport effects in a forced flow was developed gradually with Gysela. To begin with, the confinement of thermal energy, the cornerstone of the ITER experimental program, was examined in the Gysela simulations. Turbulence appears to self-organize via a wide range of processes. It has been shown that there is a transport that conflict with confinement. This transport develops from the center to the periphery and depends on the distance to an instability threshold. Due to the intrinsic stiffness of the problem, only the regimes near marginality are really of interest and are especially well addressed by flux-driven systems. The organization of these events, their spatial structure, the elements triggering these relaxations (or blocking them) constitute an important field of research which mobilizes a part of the resources of the Gysela team. Within Gysela, the interplay of several phenomena (avalanches, zonal flows ...) can be triggered at *mesoscale*. This possibility is not offered by many gyrokinetic codes.

The self-organizing elements of turbulent transport are characterized by transverse flows (*i.e.* within the magnetic flux surface, but perpendicular to the magnetic field). The transport induced by these flows is beneficial for confinement. Consequently, on initial examination, the higher the fraction of turbulent energy dedicated to these flows, the greater the losses of confinement will be reduced (a beneficial effect) up to a certain extent. Gysela allows modeling precisely these flows. In particular, the symmetry of the tokamak imposes a law of conservation on the *momentum*. The Gysela team have shown that the conservation of toroidal momentum, is rightfully observed in the simulations [65]. On this occasion, it was also proved that this momentum, generated in opposite dipoles was transported by avalanches together with heat. This study determined a rather modest momentum confinement time and thus showed the key role of the boundary conditions and the driving role of the momentum dipoles in the overall properties of the plasma [55, 62].

In addition to toroidal momentum, an important part of the plasma dynamics is carried by flows developping on flux surfaces, with a strongly sheared poloidal component (transversely to the toroidal flows aforementioned). These so-called *zonal flows* also participate in the self-organization of turbulent transport by shearing avalanches, thus weakening the coherence of the latter, and reducing the efficiency of transport. A dynamic organisation between zonal flows and avalanches is commonly observed, limiting the avalanche transport efficiency to mesoscales. This avalanche regulation physics constitutes a complex dynamical system that involves several actors who feedback on themselves. On the other hand, the main damping mechanism known is related to collisions (even if collisionless plasma hypothesis is quite usual for hot plasma). It has also become very important to model the collisions and to reproduce in the simulations some of the physics of collisional transport. Several steps have been taken in Gysela in this direction, with operators of collisions increasingly thorough. After a first generation favoring the effect of collisions for a single species of particle and along a single velocity direction [64, 49, 45], a new stage was reached in 2017 with an operator acting in all directions of velocity and valid for collisions between different species. The transport of impurities is

one of the important transport aspects for plasmas like those of ITER and WEST (tokamak hosted by CEA/IRFM). Among the impurities, tungsten plays a decisive role. In the plasma facing components [9] of current tokamaks there is tungsten in the areas most exposed to plasma. It has a harmful impact for the performance of the tokamak due to radiative losses. Beyond a concentration of the order of 1/10000, its presence limits the temperature of the plasma by radiating the available energy. It is therefore imperative to ensure a high purity of the plasma and to understand and control the transport of tungsten. This control ought to be effective from the periphery of the plasma (where the source of tungsten is located) up to the center (where its presence will limit the performance). The physics of tungsten transport is characterized by a comparable role in amplitude of collisional and turbulent transport. Collision operators built into GYSELA provide access to the involved mechanisms. The upgrade of GYSELA to handle two species instead of a single species in 2013 made it possible to study this physics. More specifically, the competition between the collisional transport and the turbulent transport has been examined, it can exhibit synergetic or couteracting effects [123, 45, 190].

One of the objectives of the study of confinement in plasmas is to find efficient scenarios where turbulent transport is controlled. Thanks to experimental know-how, a set of regimes where turbulent transport is reduced was found by the fusion community. Transport barriers fall within this framework, which consists of layers with large temperature gradients over a narrow radial region characterized by a strong reduction in turbulence. Two types of barriers have been identified, the external transport barriers (ETB) close to the periphery of the plasma, called H-mode, which have not been reproduced so far in simulations[10] and internal transport barriers (ITB). The external barrier is already a key element for the reference scenario of ITER. However, internal barriers may play a future role in achieving a thermonuclear regime (they are considered as advanced scenarii for ITER operation). It is therefore important to address the issue of turbulence control in GYSELA simulations. Three turbulent transport regulation mechanisms were examined. Firstly, the forcing of a polarization of the plasma made it possible to trigger a transport barrier in a gyrokinetic simulation. The heating mechanism associated with the polarization phenomenon has also triggered relaxation events of the transport barrier [56]. We thus observe a fine combination of the beneficial aspect of the transport barriers with the harmful aspect of the relaxations of the barriers. This ambivalent result, which corresponds to the experimental reality, is found in the mechanism of turbulence control by flows generated by fast particles in the plasma (from heating or fusion reactions). In this case, the effects of reducing the turbulent transport appear transiently, often followed by opposite effects. Implementing this type of turbulence control remains an important research topic and simulation is an important tool. Finally, a self-organization of transverse avalanche flows was observed in simulations with GYSELA. The so-called "staircases" develop in particular schemes of self-organization of turbulence. It constitutes a set of weak barriers in fairly regular schematic, characterized by relaxations of small amplitudes and thus an improvement of the confinement. The characterization with GYSELA of an improved confinement regime allowed finding signatures of this regime in the experiments [50].

The experiments with the WEST tokamak have just started at the IRFM. They are done in a toroidal volume whose ratio between the small radius and the large radius (named *aspect ratio*) is greater than in the majority of existing tokamaks elsewhere (average is $A = 3$, WEST is $A = 5$). The impact of this control parameter on plasma confinement has not been studied experimentally. Exploring this effect was therefore a modeling challenge before experiments in WEST began. To carry out this study, the know-how of the GYSELA team was mobilized to study quasi-stationary discharges that is to say by adjusting heating sources to expected losses. This study led to the demonstration of an improvement in confinement in the WEST-specific aspect ratio in contrast with the empirical studies which seemed rather to extrapolate a reverse effect. The first transport studies on WEST, that will come soon, are expected to invalidate or confirm the simulation results. The study of the self-organization of turbulence leading to these global properties is in progress.

This overview of the physical results of GYSELA shows that this research tool has made it possible to study confinement under several aspects: heat transport, large scale flows and impurities, proximity to marginal threshold, all of which are decisive factors in anticipating and understanding the future experiments in ITER. This research is thus closely associated with the major issues of magnetic confinement fusion.

Verification and benchmark of GYSELA against other codes started a long time ago because numerical and physical verification is a major concern for complex codes. A difficulty in full-f code is the delicate choice of the initial distribution function. Therefore, the impact of the initial state on turbulence and transport has been addressed. For two strongly different initial states, it has been found that the steady turbulent regime exhibits nearly identical statistical properties [122, 124]. The code has been also benchmarked in the linear and non-linear regimes against other codes as well as

[9] the components taped to the wall that see the plasma.

[10] excepting a controversial recent paper [112].

against theoretical predictions. At each stage, whenever it was possible, comparisons with analytical results have been performed: the linear results of the 4D drift-kinetic version have been validated with the linear dispersion relation results [139], neoclassical results have been recovered with our simplified collision operator [64, 63], the radial force balance equation has been recovered analytically from the conventional first order gyrokinetic equations [65] and successfully recovered numerically [64, 139], local conservation equations for density, energy and toroidal momentum have been derived [55, 65], GAMs and EGAMs[11] and effects of fast particles have been investigated by means of analytical theory and numerical simulations [59, 61, 106].

Conventional verification tests for gyrokinetic codes have been also successfully reproduced [10]. The Projection on Proper elements (PoPe) method [48] has already proven its capability to verify kinetic codes of plasma turbulence. We foresee in the near future to use this method in order to investigate the accuracy of Gysela, as well as to test model reduction methods. All along the improvement of the physics of the code, turbulence analysis have been confronted with other gyrokinetic code results [62, 67, 76]. Finally, in terms of validation, the code has been confronted to tokamak experiments, qualitatively, but also quantitatively with a comparison with Tore Supra (CEA/IRFM tokamak) results [50, 131, 185]. Overall results obtained in physics have been enabled by a major commitment of physicists, mathematicians and computer scientists.

In order to consider kinetic electrons, which bring much more physics than adiabatic electrons that we have as for now, there is still progress to be made. If one considers that the ratio of Larmor radii between ions (deuterium) and electrons is about 60, one should increase the mesh size by a factor $60^3$ and decrease the time step by a factor 60. Then, achieving a global flux-driven simulation with both kinetic ions and electrons remains a challenge in term of CPU time. Exascale resources are really needed to step forward in this direction. To go along this path, new efficient numerical schemes (see Section 3.3.4), new physics modeling (modeling scrape-off layer more precisely) and new programming approaches (task programming, adaptive graininess of computation, overlap of main communication hot spots, vectorization, see Section 2.6) are currently investigated. Preliminary physics results have been achieved on the trapped electron mode (TEM) and the transition between ITG and TEM instabilities is currently under investigation.

## 3.5 Conclusion

Conservation properties have been studied to better understand the overall behavior of a gyrokinetic code. Some techniques permitted to enhance energy conservation, mass conservation and to better preserve some stationary states. Accuracy was improved in a subset of cases where boundary conditions have been specifically modified. This work has improved the confidence users can have in the code and it has allowed for the definition of verification procedures for automated regression testing. Furthermore, I contributed to the design of the field-aligned method which is a numerical scheme that exploits properties of the physics processes to diminish the amount of stored data. Without requiring in-depth modifications of the plasma physics code, it led to an improvement both in term of accuracy and in term of reducing the execution time and memory consumption. The new expected physics to come, kinetic electrons that are very demanding, will directly benefit from this contribution. Several operators of the Gysela application have been redesigned to cope with non-uniform/reduced grid in the poloidal plane. In addition, a diffeomorphism (mapping technique) has been put into place to model realistic plasma (within the last closed magnetic surface) instead of previous circular configuration. Successful convergence studies have been performed within this new framework. The encouraging results of this proof of concept study justify that we should try to assemble these new components within an efficient framework to lower the execution time and to prove it is competitive. In this regard, task-parallelism is what we are targeting to face both the challenge of increasing complexity of the hardware and the difficulty to manage a non-uniform mesh. High-order Lagrange interpolators reveal to be a fast and accurate alternative compared to cubic spline. Clearly mathematical researches impact and interact strongly with high-performance computing field. And these two topics are tightly coupled to physics modeling. In Section 3.4, we gave a set of physics results that have been achieved thanks to this joint work.

Other works I did about numerical schemes are not presented in this document [5, 6, 11, 12, 24, 33, 37]. Such studies provide a way to check and verify HPC code, but they also trigger paths to reduce execution time or improve accuracy. These are important cirteria for parallel simulation tools that require large amounts of CPU time on supercomputers (*i.e.* energy and money). Support for joint works between applied mathematicians and computer scientists should be promoted to design algorithms and numerical schemes that have both good numerical and fine performance behaviors.

[11] geodesic acoustic modes: oscillations of the electric field whose importance in tokamak plasmas is due to their role in the regulation of turbulence.

# Chapter 4

# Towards exascale

*"If the world does not suit you,*
*you just have to change it. "*

Ayerdhal
Parleur ou les chroniques d'un rêve enclavé

Since several years now performance of parallel applications is usually far less than the achievable theoretical peak performance (many HPC applications achieve less than 5% of computational peak). Theoretical modeling and profiling tools can help to understand which level of performance one can expect for a given kernel on a particular hardware. Nevertheless, these FLOPs depend on the mix of integer/float operations and vectorized/scalar calculations, but they depend also on memory requirements of the computations, and other constraints. One of these tools is the roofline analysis[1], it requires a deep knowledge of the floating point operations done within the kernel, and also requires to take care of the specificities of the targeted architecture. Optimizing a code to extract much of the computational power requires detailed consideration of the the specific computer on which the code is to be run. Thus, performance portability is also a real issue, that comes in addition to the work of optimizing a single kernel on a given architecture. In this chapter, several studies are presented that show some aspects that should be handled in order to target the porting of existing application on novel architectures for HPC, such as GPU or many-core processors. As the semiconductor industry continues its march to exascale computing, HPC systems are becoming more complex. Within a processor, there are an increasing number of cores but also a broad array of on-chip resources ranging from memory, network, compute elements, vectorized units, inputs-outputs. The capabilities of networks, both in term of latency and bandwidth can not compete with the high CPU speed. We encounter difficulties to fully harness the hardware capabilities. We will see three example of applications that have been ported, optimized for accelerators. Some issues and a range of solutions will be highlighted.

## 4.1 Oil exploration applications on a GPU cluster

### 4.1.1 Introduction

The challenge that the oil and gas industry must face for hydrocarbon exploration requires the development of leading edge technologies to recover an accurate representation of the subsurface. *Seismic modeling* and *reverse time migration* (RTM) based on the full wave equation discretization are tools of major importance because they give an accurate representation of complex wave propagation areas. The recent development in graphics processing unit (GPU) technologies with unified architecture and general-purpose languages coupled with the high and steadily increasing performance throughput of these components made general-purpose processing on GPUs (GPGPUs) an attractive solution. In the analysis presented in [8] and summarized hereafter, we describe the algorithmic approach specific to GPUs and expose implementation details and performance results of a real-world industrial application. This collaborative work has been conducted with R. Abdelkhalek and H. Calandra from Total company and O. Coulaud and J. Roman from INRIA.

Numerical *Seismic modeling* aims at simulating seismic wave propagation in a geological medium in order to generate synthetic seismograms that are the seismograms that a set of sensors would record, given an assumed structure of the subsurface. Among the numerous approaches to seismic modeling, direct methods based on approximating the geological model by a numerical mesh are of particular interest. Besides, *Reverse time migration* is a technique for creating seismic images, providing imaging of so-called turning and prismatic waves. RTM was introduced in the 1980s, but

[1] combination of theoretical and practical modeling of the performance of an application on a target architecture [192].

despite showing promising imaging capabilities, its high computational cost prevented it from being used in practice, until recent advances in HPC technologies. RTM is based on the simulation of waves propagation: both source's and receivers' wavefields are propagated respectively forward and backward in time. These wave-fields are then compared using an imaging condition for corresponding time steps in order to form the subsoil image to detect oil deposit.

*Governing equations*

The three-dimensional (3D) acoustic wave Equation (4.1) links the pressure field $u(x, y, z, t)$ to the density $\rho(x, y, z)$ and the velocity $c(x, y, z)$,

$$\frac{1}{c^2\,\rho}\frac{\partial^2 u}{\partial t^2} = \nabla.\left(\frac{1}{\rho}\nabla u\right). \tag{4.1}$$

Using finite difference methods to solve the wave equation is one way among others to tackle direct methods. The way this equation is derived, among a regularly meshed domain: we write a cascaded first-order spatial difference expression to compute the second time difference of the wavefield:

$$\frac{\partial^2 u}{\partial t^2} = c^2\,\rho\left[\frac{\partial_-}{\partial x}\left(\frac{1}{\rho}\frac{\partial_+ u}{\partial x}\right) + \frac{\partial_-}{\partial y}\left(\frac{1}{\rho}\frac{\partial_+ u}{\partial y}\right) + \frac{\partial_-}{\partial z}\left(\frac{1}{\rho}\frac{\partial_+ u}{\partial z}\right)\right]. \tag{4.2}$$

The $\partial_-$ and $\partial_+$ symbols denote the spatial difference operators that are centered halfway between grid points either forward or backward in the direction of the spatial difference. We use operators with a 3D stencil width of 8. So, for example, the first derivative of u with respect to x evaluated at $(i + 1/2)\Delta x$ is written as

$$\frac{\partial_+}{\partial x}u\left(\left(i+\frac{1}{2}\right)\Delta x\right) = \sum_{n=0}^{3} a_n[u((i+1+n)\Delta x) - u((i-n)\Delta x)], \tag{4.3}$$

where the $a_n$ coefficients are the eighth-order finite difference operator optimized coefficients. When the density is considered to be constant in all the domain, Equation (4.1) is simplified to $\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \Delta u$. This approximation is especially carried out during migration process. The discretization of this equation is carried out with a second-order-in-time leapfrog scheme and an eighth-order centered difference scheme in space with either Taylor or optimized coefficients, which leads in 2D to

$$U_{i,j}^{n+1} = 2U_{i,j}^{n} - 2U_{i,j}^{n-1} + c^2\Delta t^2\left[\frac{1}{\Delta x^2}\sum_{m=-4}^{4} b_m U_{i+m,j}^{n} + \frac{1}{\Delta y^2}\sum_{m=-4}^{4} b_m U_{i,j+m}^{n}\right]. \tag{4.4}$$

Discretization of these equations on CPU is now briefly sketched. At each grid point, the stencil is applied to compute the Laplacian value. Thus, to update one grid point, 25 data read accesses are performed from the wave-field's last update ($U^n$) and one from ($U^{n-1}$). Data are contiguous along x-axis. When a domain is too large to fit into the cache size (almost always), data accesses along y-axis and z-axis become expensive. In specific cases (constant density) cache blocking technique is used along y-axis. In Algo. 20, we need to loop over all the grid points twice. During first sweep, three 3D arrays are filled with the forward first derivatives along each axis. Then, backward first derivatives are computed using these arrays and the density array. The stencil is then twice as large as in the constant density case.

Algo 20 describes the sequential variable density seismic modeling CPU implementation. The reference CPU implementation that we use to evaluate our GPU accelerated solution is parallel. It is based on a standard domain decomposition along $x$, $y$ and $z$. Ghost nodes are exchanged via non-blocking Message Passing Interface (MPI) communications. Ghost node thickness is determined by the stencil used to solve the wave equation: four planes in the constant density case and eight planes in the variable density case.

In practice, because the simulated domain cannot extend infinitely, damping zones are added at the borders of the domain to avoid reflections. In these zones, we use perfectly matched layers (PMLs) to simulate nonreflecting boundaries. This classical technique is based on a modified wave equation with a solution decreasing exponentially in the damping layers. PML numerical implementation requires solving two coupled discretized equations: the first one updates the pressure wavefield array and the second updates an artificial damping function array.

```
for time = t_start to t_end do
    add source term;
    forall domain grid points do
        compute forward first derivatives;
    forall domain grid points do
        compute backward first derivatives;
        update wavefield;
    save seismogram;
```

**Algo. 20:** Variable density seismic modeling

```
for time = t_start to t_end do
    add source term;
    forall domain grid points do
        take forward time step;
    save boundaries(time);
for time = t_end to t_start do
    read saved boundaries(time);
    forall domain grid points do
        backward time step source wavefield;
    add receiver term;
    forall domain grid points do
        backward time step receiver wavefield;
    forall domain grid points do
        apply imaging condition;
```

**Algo. 21:** Reverse Time Migration algorithm

The RTM algorithm is listed in Algo. 21. It relies on the constant density approach to simulate the source wavefield (in both the forward and the backward phases) and the receivers wavefield. The imaging condition applied during backward recursion consists in the accumulation of the cross-correlation between source and receiver wavefields over the time iterations. The work described here treat the wave propagation in a variable density domain on a cluster of GPU with PML boundary conditions. This configuration within a production code was a novelty in 2011.

### 4.1.2 Graphics processing unit implementation

#### Architecture and global setting

We used NVIDIA Tesla S1070 blades which are composed of four Tesla 10 (T10) GPUs embedded with 4 GB of memory per GPU. Each pair is connected to a host node via a PCIe 2.0 bus, acting as coprocessors or computing devices. Each T10 GPU can be seen as a set of 30 multiprocessors. Each multiprocessor is composed of eight streaming processors running in a single instruction, multiple data like way. The processors inside a multiprocessor have access to 16,384 32-bit registers (divided among processors) and to 16-kB shared memory space that can be seen as a cache memory with very low latency (4 clock cycles per read/write if no conflict). All the GPU's processors have read/write access to the off-chip global memory (nearly 4 GB) but with a higher latency up to 600 clock cycles. We focus on the Compute Unified Device Architecture (CUDA) that we used for our implementations. CUDA defines a thread hierarchy in order to organize threads in a geometric topology. Thus, threads are grouped into 1D, 2D, or 3D thread blocks. These blocks rearrange into 1D or 2D grids. This topology matches the thread organization to the GPU structure. Threads of the same block are run on the same multiprocessor, making them able to have access to the multiprocessor's shared resources (same shared memory space, texture, and constant memory cache) and to coordinate their activities by using a barrier function. Each thread is then identified according to its thread ID and the block ID of the block it belongs to. Threads of consecutive thread IDs are grouped into warps of 32 threads.

#### Framework and constraints

The concept of occupancy is of major importance when designing CUDA kernels. Occupancy is defined as the ratio of active warps per multiprocessor to the maximum number of active warps (32 for the T10 GPU). The number of active warps is defined by the availability of shared resources inside the multiprocessor. Other important metrics are the global memory accesses and the instruction throughputs to be compared with the theoretical peaks to get the memory over instruction throughput ratios. These ratios help us understand the limitations of each kernel implementation.

#### Host-graphics processing unit communication

For the pressure field to be updated at a given iteration $t_{n+1}$, pressure field updates at iterations $t_{n-1}$ and $t_n$ are required. Data in the GPU global memory are persistent across different kernel launches. Time evolution of the wavefield is then performed in the GPU memory by logically swapping $t_n$ and $t_{n+1}$ arrays. Only the ghost nodes[2] need then to be exchanged between host and GPU at each time step when partitioning the domain among several GPUs. Exchanging only ghost nodes instead of the whole domain, although requiring more programming effort, largely reduces the amount of data exchange and thus computing time. For example, transferring a whole 3D $528 \times 2548 \times 1067$ domain to/from GPU takes approximately 0.1 s, which is equivalent to the time needed for a single constant density modeling time iteration for this domain. Our implementation allows decomposing the domain

[2]back to 2011, MPI communications could not be initiated directly between GPUs, as CUDA-aware MPI and GPUDirect RDMA allow it now.

along the three axes. However, the location of the ghost nodes to be exchanged (which depends on the domain decomposition geometry) has a strong impact on the CPU-GPU communication time. In fact, exchanging contiguous data in the physical memory (corresponding to a decomposition along the slowest varying dimension) is much faster than any other ghost node exchange. This is illustrated in Table 14. We compare the time needed to transfer all the ghost nodes (eight-element width) of a $512 \times 512 \times 512$ data volume, from host to device and from device to host, when using different techniques. CUDA API provides routines to perform linear memory, 2D and 3D data transfers between the host and the device. We started by considering a single transfer of the whole data cube (Single Transfer). Then we used 2D memory copies in a loop along the slowest varying direction to transfer only the ghost nodes (MemCpy2D). This reduces the total size of transferred data but largely increases the number of transfers and thus latency time. More than 70% of the total transfer time was spent transferring the two ghost node sides where data was the least contiguous (left and right – or x-direction – in our implementation). To reduce this cost, we considered a third strategy on the basis of rearranging data of these ghost nodes by performing a transposition on the GPU side (MemCpy2D+Transpose). This technique showed to be the best approach because it was 2.33× faster than the previous one for CPU-to-GPU data transfers and 3 faster for GPU-to-CPU data transfers. Using 3D memory copies either to transfer the whole data cube (Single MemCpy3D) or to transfer only the ghost nodes (Partial Transfer 3D) showed poor performance.

| Transfer technique | HtoD | DtoH |
|---|---|---|
| Single Transfer | 90.9 | 96.3 |
| MemCpy2D | 63.3 | 96.3 |
| MemCpy2D+Transpose | 27.2 | 31.2 |
| Single MemCpy3D | 93.5 | 98.3 |
| Partial Transfer 3D | 55.4 | 215.6 |

Table 14: Host-GPU transfer time (ms) using different techniques from host to device (HtoD) and from device to host (DtoH).

**Computing kernel design**

We designed mainly three kernels dedicated to : i) constant density modeling, ii) variable density modeling, iii) Reverse time migration. Several techniques and strategies have been applied in order to shorten the execution time (not detailed here): maximizing occupancy, padding to ensure global memory coalescing, reducing registers and shared memory use, boundary condition treatment, CPU-GPU communications. Therefore, several variants of the kernels and variants of subpart of the kernels have been setup and benchmarked. Among all optimizations done, several ones were really worthy. First, to limit frequent memory access, we used shared memory: data are first copied to shared memory (texture fetches in 2D, global memory in 3D), then read by threads of the same block. This classical technique reduces global memory accesses for the same data but increases and puts pressure on shared memory use. Second, to limit pressure on shared memory and thus increase occupancy, we adopted a novel strategy: instead of dedicating a thread to each grid point, we used a sliding window algorithm, sliding over planes in the $z-$direction. We used shared memory to store the $(x, y, z = z_{current})$ plane to be updated, so that all the threads can compute Laplacian value along $x-$ and $y-$direction. For the $z-$direction, each thread loads into $float4$ registers the four pressure values under and over the current plane. At each loop iteration, corresponding to a shift along $z-$direction, wavefield values are shifted. Thus, at a given iteration, only the $(x, y, z = z_{current+4})$ plane is read from global memory. Third, the kernel managing PML boundary conditions was split into two kernels launched one after the other. It permitted to reduce register use and then improve occupancy. Fourth, some dedicated kernels has been setup to manage boundary conditions especially, other kernels deal with the inner domain. This way, a conditional is avoided within the kernels (the conditional is at the level of the caller to the kernel) and it also limits the register use. Fifth, for RTM implementation the domain to be updated on the GPU is decomposed into streams. Each stream consists in a data transfer from host to GPU, a kernel launch, and a data transfer from GPU to CPU. While executing a stream's kernel, the data corresponding to the next stream are transferred, thus overlapping the cost of the communication.

### 4.1.3 Benchmarks

**Graphics processing unit cluster test bed**

The GPU cluster test bed is composed of 10 Xeon bisocket quad-core nodes (INTEL) coupled with five NVIDIA Tesla S1070 servers. Each node is connected via one PCIe gen 2.0 bus to the Tesla server. The Tesla server is composed of four T10 GPUs. Each node has access to two GPUs via the

same bus. All the GPU computing times indicated hereafter, take into account the kernel execution time as well as the CPU-GPU communication needed to initiate and finish the kernel. We now review and analyze some performance results. The reference time is the time to process the entire domain without using any subdomain decomposition: the CPU reference time is given by running a sequential implementation on a single CPU core, and the GPU reference time is given by using one GPU card.

**Constant and variable density modeling**

We consider a 3D test case (hereafter referred to as 3DModel) of grid dimensions $521 \times 254 \times 1067$ with $\Delta x = 12.5$, $\Delta y = 12.5$, and $\Delta z = 10$ in meters. Results reported in Table 15 show that the CPU/GPU time ratio decreases with the number of subdomains increasing because the computing time decreases and the communication time becomes more predominant (both CPU-GPU and MPI communications are involved). Yet, only computing time is reduced when using GPUs.

From Fig. 80, many of the factors that may impact our seismic modeling variable density code performance may be inferred. In this Fig., the time required for one time iteration is reported and decomposed into MPI communication time, host-GPU communication time, and GPU kernels' execution time. Several measures, corresponding to different domain decomposition geometries, are reported for the 3DModel. As a rule of thumb, we can notice that decomposing along the slowest varying dimension is the most effective strategy. In fact, it reduces the overhead of both MPI and host-GPU transfers because, in both cases, contiguous data are exchanged.

| | Constant density | | | Variable density | | |
|---|---|---|---|---|---|---|
| Number of subdomains | CPU time | GPU time | Ratio | CPU time | GPU time | Ratio |
| 1 | 3.89 | 0.146 | 26.71 | 8.83 | 0.205 | 43.07 |
| 2 | 1.59 | 0.083 | 19.15 | 5.09 | 0.130 | 39.07 |
| 4 | 0.94 | 0.046 | 20.43 | 2.11 | 0.078 | 26.98 |
| 8 | 0.47 | 0.034 | 15.16 | 1.06 | 0.053 | 20.03 |

Table 15: Modeling averaged time in seconds for one iteration on 3DModel.

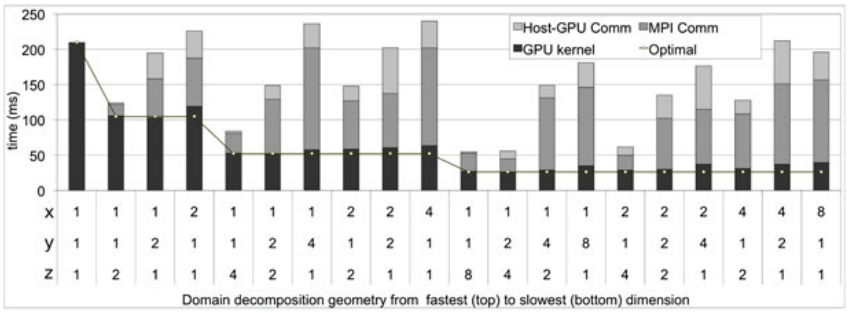



Figure 80: Variable density average time step for different domain decomposition geometries.

**Reverse time migration**

We report in Table 16 detailed times and speedup of the RTM when increasing subdomains number. FWD time refers to the time spent during the forward sweep, whereas BWD time refers to the backward sweep where the source and the receivers wavefields are propagated backward in time and the imaging condition (correlation of the two wavefields) is performed. The RTM implementation involves more host-GPU communications than the modeling. This reduces the obtained CPU/GPU time ratio very significantly. The results reported are obtained when using four CUDA streams in the forward sweep and two in the backward sweep. Stream implementation showed to be 1.4 faster than the non-streamed implementation.

| | CPU | | | GPU | | | Ratio | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of subdomains | FWD | BWD | Total | FWD | BWD | Total | FWD | BWD | Total |
| 1 | 923.9 | 1802.1 | 2726.0 | 74.0 | 92.8 | 166.8 | 12.5 | 19.4 | 16.3 |
| 2 | 526.9 | 953.3 | 1480.2 | 47.1 | 73.4 | 120.5 | 11.2 | 13.0 | 12.3 |
| 4 | 269.9 | 490.4 | 760.2 | 32.6 | 61.2 | 93.8 | 8.3 | 8.0 | 8.1 |
| 8 | 122.0 | 190.6 | 312.6 | 25.3 | 52.3 | 77.6 | 4.8 | 3.6 | 4.0 |

Table 16: RTM exec. times in s for one shot (3700 iterations) on a $288 \times 118 \times 338$ domain.

**Discussion**

We ported a seismic modeling application and reverse time migration on a GPU cluster. We compared our GPU implementation with the original CPU version on several test cases and obtained a performance increase of 16 for RTM and up to 43 for modeling. This shows that GPGPU solutions are worth being used in a large-scale industrial context. GPGPU is paving the way for many-core CPU systems, fine-grained parallelism constraints are likely to be similar, and concepts and approaches developed for GPUs may be useful in the many-core era.

We observed that the CUDA compute capability together with the GPU device have a large impact on performance. Auto-tuning techniques would be suitable to adapt and tune the best kernel depending on a peculiar system. Such kind of auto-tuning has been worthwhile for optimizing kernels in this work [82]. In addition, it is expected that the kernels are updated/upgraded when a new generation of GPUs appears.

Keeping equivalent versions of a GPU and a CPU implementation of several kernels is quite challenging. Maintenance, support and development are much more complicated in such context. High-level programming language (HMPP, OpenACC) can help a bit but portability, readability, efficiency and maintenance issues remain a challenge.

## 4.2 Vlasov-Poisson application on GPU

### 4.2.1 Introduction

**Context**

The work described here [30] highlights adjustments needed for a semi-Lagrangian Vlasov-Poisson code to tackle a GPU device. It followed a former study made on the LOSS code described in a set of papers [9, 14, 35]. A classical approach in the semi-Lagrangian community involves the use of cubic splines to achieve the numerous interpolations induced by this scheme. The local spline method, already presented in Chapter 1, allows one to decouple computations for parallelizing them. Only relatively small MPI inter-processor communication costs were induced and these codes scaled well over hundreds of cores. A mixed-precision method has been setup to reduce execution time and improve the overall performance on GPU, as we shall see.

**Equations**

We consider a reduced model for two physical dimensions, corresponding to $x$ and $v_x$ such as $(x, v_x) \in \mathbb{R}^2$. The 1D variable $x$ represents the configuration space and the 1D variable $v_x$ stands for the velocity along $x$ direction. Moreover, the self consistent magnetic field is neglected ($v_x$ considered to be relatively small). The Vlasov-Poisson system then reads:

$$\frac{\partial f}{\partial t} + v_x \,.\nabla_x f + (E + v_x \times B)\ .\nabla_{v_x} f = 0, \tag{4.5}$$

$$-\varepsilon_0 \nabla^2 \phi = \rho(x,t) = q \int f(x, v_x, t) d\, v_x, \qquad E(x,t) = -\nabla\phi. \tag{4.6}$$

where $f(x, v_x, t)$ is the particle density function, $\rho$ is the charge density, $q$ is the charge of a particle (only one species is considered) and $\varepsilon_0$ is the vacuum permittivity, $B$ is the applied magnetic field. Eq. (4.5) and (4.6) are solved successively at each time step. The density $\rho$ is evaluated in integrating $f$ over $v_x$ and Eq. (4.6) gives the self-consistent electrostatic field $E(x,t)$ generated by particles. The physical domain is defined as $\mathcal{D}_p^2 = \{(x, v_x) \in [x_{\min}, x_{\mathrm{Max}}] \times [v_{x_{\min}}, v_{x_{\mathrm{Max}}}]\}$. For the sake of simplicity, we will consider that the size of the grid mapped on this physical domain is a square indexed on $\mathcal{D}_i^2 = [0, 2^j - 1]^2$. Periodic extensions is implemented as boundary condition. For the sake of simplicity, we focus here on the very classical linear Landau damping test case (with $k = 0.5, \alpha = 0.01$) which highlights the accuracy problem one can expect in Vlasov-Poisson simulation. The initial distribution function is given by

$$f(x, v_x, 0) = \frac{e^{-\frac{{v_x}^2}{2}}}{\sqrt{2\,\pi}} (1 + \alpha\, \cos(k\, x))\ .$$

**Numerical scheme**

The Vlasov Equation (4.5) can be decomposed by splitting. It is possible to solve it, through the following elementary advection equations:

$$\partial_t f + v_x \partial_x f = 0, \quad (\hat{x} \ \text{ operator}) \qquad \partial_t f + \dot{v_x} \partial_{v_x} f = 0. \quad (\hat{v}_x \ \text{ operator})$$

Each advection consists in applying a shift operator. A splitting of Strang is employed to get a second order accuracy in time. We took the sequence $(\hat{x}/2, \hat{v}_x, \hat{x}/2)$, where the factor $1/2$ means a shift over a reduced time step $\Delta t/2$. Algo. 22 shows how the Vlasov solver of Eq. (4.5) is interleaved with the field solver of Eq. (4.6).

**Local spline method**

Each 1D advection (along $x$ or $v_x$) consists in two substeps (Algo. 23). First, the density function $f$ is processed in order to derive the cubic spline coefficients. The specificity of the local spline method (already described18) is that a set of spline coefficients covering one subdomain can be computed concurrently with other ones. Thus, it improves the standard approach that requires a coupling between all coefficients along one direction. Second, spline coefficients are used to interpolate the function $f$ at specific points. This substep is intrinsically parallel whether with the standard spline method or with the local spline method: one interpolation involves only a linear combination of four neighboring spline coefficients.

In Algo. 23, $x^o$ is called the origin of the characteristic. With the local spline method, we gain concurrent computations during the spline coefficient derivation (line 2 of the algorithm). Our goal is mainly to port Algo 23 onto GPU, we need then a highly parallel form adapted to the CUDA framework.

**Input :** $f_t$
**Output:** $f_{t+\Delta t}$
*// Vlasov solver, part 1*
1 1D Advection, operator $\frac{\hat{x}}{2}$ on $f(.,.,t)$
*// Field solver*
2 Integrate $f(.,.,t+\Delta t/2)$ over $v_x$
3 to get density $\rho(.,t+\Delta t/2)$
4 Compute $\phi_{t+\Delta t/2}$ with Poisson solver
5 using $\rho(.,t+\Delta t/2)$
*// Vlasov solver, part 2*
6 1D Advection, operator $\hat{v_x}$ (use $\phi_{t+\Delta t/2}$)
7 1D Advection, operator $\frac{\hat{x}}{2}$

**Algo. 22:** One time step

**Input :** $f$
**Output:** $f$
**forall** $v_x$ **do**
  $a(.) \leftarrow$ spline coeff. of sampled function $f(., v_x)$
  **forall** $x$ **do**
    $x^0 \leftarrow x - v_x.dt$
    $f(x, v_x) \leftarrow$ interpolate $f(x^0, v_x)$ with $a(.)$

**Algo. 23:** Advection in $x$ dir., $dt$ time step

## 4.2.2 Specific improvements for GPGPU

**Improvement of numerical precision**

In 2009, one had to consider mostly single precision (SP) computations to get good performance out of a GPU. The double precision (DP) was much slower than single precision (SP) on the available GPU devices at this time. However, it remains true nowadays that SP calculation behave faster than DP on many computing units. In addition, the use of double precision increases pressure on memory bandwidth by a factor two. In many targeted physical configurations, using SP without caution leads to unacceptable numerical results [30].

It turns out that we were able to modify the scheme to reduce numerical errors even with only SP operations during the advection steps. To do so, a new function $\delta f(x, v_x, t) = f(x, v_x, t) - f_{\text{ref}}(x, v_x)$ is introduced. Working on the $\delta f$ function could improve accuracy if the values that we are working on are sufficiently close to zero. Then, the reference function $f_{\text{ref}}$ should be chosen such that the $\delta f$ function remains relatively small (in $L_\infty$ norm). Let us assume that $f_{\text{ref}}$ is a constant along the $x$ dimension. For the Landau test case, we choose $f_{\text{ref}}(v_x) = \frac{1}{\sqrt{2\,\pi}}\, e^{-\frac{{v_x}^2}{2}}$. As the function $f_{\text{ref}}$ is constant along $x$, the $x$-advection applied on $f_{\text{ref}}$ leaves $f_{\text{ref}}$ unchanged. Then, it is equivalent to apply $\hat{x}$ operator either on function $\delta f$ or on function $f$. Working on $\delta f$ is worthwhile ($\hat{x}$ operator): for the same number of floating point operations, we increase accuracy in working on small differences instead of large values. Concerning the $\hat{v_x}$ operator however, both $f_{\text{ref}}$ and $f$ are modified. For each advected grid point $(x, v_x)$ of the $f^\star$ function, we have ($v_x^o$ is the foot of the characteristic):

$$f^\star(x, v_x) = f(x, v_x^o) = \delta f(x, v_x^o) + f_{\text{ref}}(v_x^o), \qquad \delta f^\star(x, v_x) = f^\star(x, v_x) - f_{\text{ref}}(v_x)\,,$$
$$\delta f^\star(x, v_x) = \delta f(x, v_x^o) - (f_{\text{ref}}(v_x) - f_{\text{ref}}(v_x^o)).$$

Working on $\delta f$ instead of $f$ changes the operator $\hat{v_x}$. We now have to interpolate both $\delta f(x, v_x^o)$ and $(f_{\text{ref}}(v_x) - f_{\text{ref}}(v_x^o))$. In doing so, we increase the number of computations (only in $\hat{v_x}$ operator); because in the original scheme we had only one interpolation per grid point $(x, v_x)$, whereas we have two in the

new scheme. In spite of this cost increase, we enhance the numerical accuracy using $\delta f$ representation (see Fig. 81). A sketch of the $\delta f$ scheme is shown in Algo 24.

**Input :** $\delta f_t$
**Output:** $\delta f_{t+\Delta t}$

1D advection on $\delta f$, operator $\frac{\hat{x}}{2}$

Integrate $\delta f(.,.,t+\Delta t/2) + f_{\text{ref}}(.)$
to get $\rho(.,t+\Delta t/2)$
Compute $\phi_{t+\Delta t/2}$,
with Poisson solver on $\rho(.,t+\Delta t/2)$

1D advection on $\delta f$, operator $\hat{v_x}$
→ stored into $\delta f$
Interpolate $f_{\text{ref}}(v_x) - f_{\text{ref}}(v_x^o)$
→ results added into $\delta f$
1D advection on $\delta f$, operator $\frac{\hat{x}}{2}$

**Algo. 24:** One time step, $\delta f$ scheme

**Input :** $f_t$ in global memory of GPU
**Output:** $f_{t+dt}$ in global memory of GPU

`// A) Load from global mem. to shared mem.`
Each thread loads 4 floats from global mem.
Floats loaded are stored in *shared memory*
Boundary conditions are set (extra floats are read)
Synchro.: 1 thread block owns $n$ *vectors* of 32 floats
`// B) LU Solver`
1 thread over 8 solves a LU system (7 are idle)
Synchro.: 1 block has $n$ vectors of spline coeff.
`// C) Interpolations`
Each thread computes 4 interpolations
`// D) Writing to GPU global memory`
Each thread writes 4 floats to global mem.

**Algo. 25:** Skeleton of an advection kernel

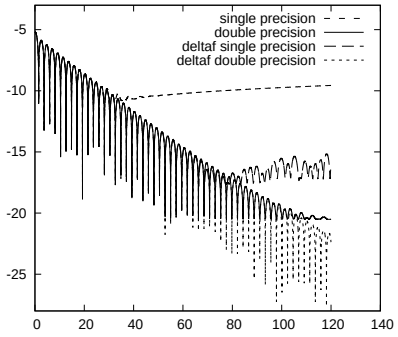



Figure 81: Electric energy for Landau test case $1024^2$, using $\delta f$ representation or standard representation.

### Data placement

We perform the computation on data $\delta f$ of size $(2^j)^2$. Typical domain size varies from $128 \times 128$ (64 KB) up to $1024 \times 1024$ (4 MB). The whole domain fits easily in global memory of current GPUs. In order to reduce unnecessary overheads, we decided to avoid transferring 2D data $\delta f$ between the CPU and the GPU as far as we can. So we kept data function $\delta f$ onto GPU global memory. CUDA computation kernels update it in-place. For diagnostics purposes only, the $\delta f$ function is transferred to the RAM of the CPU at a given frequency.

### Spline coefficients computation

Spline coefficients (of 1D discretized functions) are computed on patches of 32 values of $\delta f$. A smaller patch would introduce significant overhead because of the cost of first derivative computations on the patch borders [35]. A bigger patch would increase the computational grain which is a bad thing for GPU computing that favors scheduling large number of threads. The 2D domain is decomposed into small 1D vectors (named "patches") of 32 $\delta f$ values. To derive the spline coefficients, tiny LU systems are solved. The assembly of right hand side vector used in this solving step can be summarized as follows: keep the 32 initial values, add 1 more value of $\delta f$ at the end of the patch, and then add two derivatives of $\delta f$ located at the borders of the patch. Once the right hand side vector is available (35 values), two precomputed matrices $L$ and $U$ are inverted to derive spline coefficients (using classical forward/backward substitution). We decided not to parallelize this small $LU$ solver: a single CUDA thread is in charge of computing spline coefficients on one patch That point could be improved in the future in order to use several threads instead of one.

### Parallel interpolations

On one patch, 32 interpolations need to be done (except at domain boundaries). These interpolations are decoupled. To maximize parallelism, one can even try to dedicate one thread per interpolation. Nevertheless, as auxiliary computations could be factorized (for example the shift $v_x.dt$ at line 4 of Algo. 23), it is relevant to do several interpolations per thread to reduce global computation cost.

The number of such interpolations per thread is a parameter that impacts performance. This blocking factor is denoted $K$.

#### Data load

The computational intensity of the advection step is not that high. During the *LU* phase (*spline coefficients computation*), each input data is read and written twice and generates two multiplications and two additions in average. During the *interpolation step*, there are four reads and one write per input data and also four multiplications and four additions. The low computational intensity implies that we could expect shortening the execution time in reducing loads and writes from/to GPU global memory. So, there is a benefit to group the spline computation and the interpolations in a single kernel. Several benchmarks have confirmed that with two distinct kernels (one for building splines and one for interpolations) instead of one, the price of load/store in the GPU memory increases. Thus, we now describe the solution with only one kernel.

#### Domain decomposition and fine grain algorithm

We have designed three main kernels. Let us give short descriptions: `KernVA` operator $\hat{v_x}$ on $\delta f(x, v_x)$, `KernVB` adding $f_{\text{ref}}(v_x) - f_{\text{ref}}(v_x^o)$ to $\delta f(x, v_x)$, `KernX` operator $\hat{x}$ on $\delta f(x, v_x)$. The main steps of `KernVA` or `KernX` are given in Algo. 25. The computations of $8\,n$ threads acting on $32\,n$ real number values are described (it means $K = 4$ hardcoded here).
First `A)` substep reads floats from GPU global memory and puts them into fast GPU shared memory. When entering the `B)` substep, all input data have been copied into shared memory. Concurrently in the block of threads, small *LU* system are solved (but 87% of the threads stays idle). Spline coefficients are then stored in shared memory. In substep `C)`, each thread computes $K = 4$ interpolations using spline coefficients. This last task is the most computation intensive part of the simulator. Finally, results are written into global memory.

### 4.2.3 Performance

#### Machine

In order to develop the code and perform small benchmarks, a cheap personal computer has been used (in 2009). The CPU is a dual-core E2200 Intel (2.2Ghz), 2 GB of RAM, 4 GB/s peak bandwidth, 4 GFLOPS peak, 1 MB L2 cache. The GPU is a GTX260 Nvidia card: 1.24 Ghz clock speed, 0.9 GB global memory, 95 GB/s peak bandwidth, 750 GFLOPS peak, 216 cores.

| Substeps in one time step | CPU (deltaf 4B) | GPU (deltaf 4B) |
|---|---|---|
| X Advection | 5123 $\mu s$ (*1.0*) | 172 $\mu s$ (*29.7*) |
| V Advection | 4850 $\mu s$ (*1.0*) | 144 $\mu s$ (*33.7*) |
| Field computation | 133 $\mu s$ (*1.0*) | 93 $\mu s$ (*1.4*) |
| Complete Iteration | 10147 $\mu s$ (*1.0*) | 546 $\mu s$ (*18.6*) |

Table 17: Small $256^2$ Landau test case- time breakdown within a time step (speedup in parentheses) averaged over 5000 calls

| Substeps in one time step | CPU (deltaf 4B) | GPU (deltaf 4B) |
|---|---|---|
| X Advections | 79600 $\mu s$ (*1.0*) | 890 $\mu s$ (*90*) |
| V Advections | 89000 $\mu s$ (*1.0*) | 1000 $\mu s$ (*89*) |
| Field computation | 1900 $\mu s$ (*1.0*) | 180 $\mu s$ (*11*) |
| Complete Iteration | 171700 $\mu s$ (*1.0*) | 2250 $\mu s$ (*76*) |

Table 18: Large $1024^2$ Landau test case - Time breakdown within a time step (speedup in parentheses) averaged over 5000 calls

#### Benchmark

Let us first have a look on performance of the $\delta f$ scheme. We consider the small testbed (E2200-GTX260), and a reduced test case ($256^2$ domain). The simulation ran on a *single* CPU core, then on the 216 cores of the GTX260. Timing results and speedups (reference is the CPU single core) are given in Table 17. The speedup is near 30 for the two significant computation steps, but is smaller for the field computation. The field computation part includes two substeps: first the integral computations over the 2D data distribution function, second a 1D Poisson solver. The timings for the integrals are bounded up by the loading time of 2D data from global memory of the GPU (only one addition to do per loaded float). The second substep that solves Poisson equation is a small sequential 1D problem. Furthermore, we loose time in launching kernels on the GPU (25 $\mu s$ per kernel launch included in timings shown).
In Table 18, we look at a larger test case with data size equal to $1024^2$. Compared to a single CPU core, the advection kernels have speedups from 75 to 90 for a GPU card (using 260 000 tiny threads). Here, the field computation represents a small computation compared to the advections and the low speedup for the field solver is not a real penalty. A complete iteration reaches a speedup of 76 on GPU over CPU. But one should mitigate this result by the fact that an OpenMP parallelization on a multi-core CPU would have lowered this speedup by almost a factor 2 on 2 cores.

### 4.2.4 Discussion

It turns out that $\delta f$ method is a valid approach (under some assumptions on the reference $f_{\text{ref}}$) to perform a semi-Lagrangian Vlasov-Poisson simulation using only 32-bit floating-point precision instead of classical 64-bit precision. A very fine grain parallelization of the advection step is presented that scales well on thousands of threads. Such kind of very fined-grain algorithm is also expected to exploit foreseen exascale devices. We have discussed the kernel structure and the trade-offs made to accommodate the GPU hardware.

The application is bounded up by memory bandwidth because computational intensity is small. It is well known that algorithms of high computational intensity are able to be efficiently implemented on GPU. We have shown an algorithm of low computational intensity that can also benefit from GPU hardware. Our GPU solution reaches a significant speedup of overall 76 compared to a single core CPU execution. Optimizations done are dedicated to GPU. In practice, it means that we kept and maintain separate codes for the two versions of the main kernels: CPU (OpenMP+Fortran version) and GPU (CUDA+C+Fortran version).

## 4.3 Xeon Phi to accelerate applications

### 4.3.1 Introduction

#### Context

Adapting the code to new parallel architectures is a key issue. It is important to understand new hardware, their advantages and limitations. This Xeon Phi coprocessor is the first one (KNC), largely commercialized, that implements the Many Integrated Cores (MIC) architecture. This architecture is appealing, as the compiling and execution steps are quite similar to those of standard INTEL processors, and the theoretical peak performance and memory bandwidth are high. Furthermore, we had the opportunity to get an access to the Helios machine dedicated to Fusion community (Japan), where such devices were available (starting in 2014). This work [2,20] describes the challenges presented by porting parts of the GYSELA code to the INTEL Xeon Phi coprocessor, as well as opportunities for optimization, vectorization and tuning that can be applied to many other applications.
To simplify our study, we have extracted a mini-application from the GYSELA code with a reduced code line count, and with less physics submodels. The mini-application is much easier to modify in order to optimize and to tune the code. We tried a *top-down* approach to improve the performance of the costly computational part located in a single subroutine. Many standard optimization techniques were investigated on this subroutine, but the investigations was too rough and tedious, we did not reach the level of performance that one can expect from Xeon Phi (KNC).
Then, we investigated a *bottom-up* approach to overcome the performance limitation we observed. To finely study the optimization aspects, we designed very small computation kernels, easy to modify, that are representative of the costly part of the mini-application. The description of these kernels, the optimizations we have done on them and benchmarks results we obtained, are summarized hereafter. Finally, the lesson learned from the porting of the tiny computation kernels has been used to globally improve the performance of the mini-application. Finally, optimized kernels were integrated back into the mini-application.

#### Hardware and setting

The benchmarks presented in this study have been mostly obtained on the Helios machine on IFERC computing facility (Rokkasho, Japan). The hardware we used consists of Xeon Phi cards with 60 cores (1.052 GHz), where each core is capable of executing four concurrent threads. The chip handles an instruction set that operates on 512 bits wide vector registers. Because of the in-order execution model and a latency of vector instructions which is greater than 2 on this Xeon Phi, one needs 2 to 4 threads per core to get good performance. Each core executes alternatively these threads in a round-robin manner in order to hide the instructions, pipelines and memory latencies. Also thread pinning plays a significant role to achieve performance. High latencies to access cache memory are observed on Xeon Phi in combination with smaller cache sizes (compared to CPU) brought us to the conclusion that efficient cache reuse strategies are much more difficult to implement on Xeon Phi (KNC) than on Sandy Bridge. The host processor (Sandy Bridge, INTEL Xeon E5-2600), *i.e* the host on which the Xeon Phi is plugged, consists of two 2.7 GHz processors with eight cores each and with two admissible threads per core.
We chose the so-called “native” or “symmetric mode”, similar to CPUs programming models and not the “offload” mode which is closer to the CPU+GPU model with a host interacting with a coprocessor.

The reason this "native" model is available for Xeon Phi (KNC) while there is no such model for GPU accelerators is that the Xeon Phi runs a complete operating system whereas GPUs are operated by the system running on the host. In this model, each coprocessor is seen as a separate node and the MPI application can be deployed on both the CPU and MIC nodes. The largest advantage of this strategy resides in its ease of use: the application developer stays within the well known MPI programming framework. The hardware non-homogeneity (CPU versus Phi) also leads to possible load imbalance issues. Depending on the computation and for a given load, the execution time will likely be different on the host and on the coprocessor. For the native mode, as the large majority of MPI applications distributes equally the same computations with the same load on the different nodes, some tasks will finish earlier than others then leaving their computing unit idle. A strategy to alleviate this would consist in modifying the MPI application such that different groups of MPI tasks can have different loads. Runtime system like StarPU, XKaapi, ompSs, PaRSEC can help reducing the imbalance also. Indeed, once the developer has expressed the parallelism of his application as a set of tasks, the runtime system schedules them dynamically on the available computing units and performs the required data transfers.

**Challenges**

The types of problems the Xeon Phi (KNC) is well suited for are intensive numerical computations. Additionally, for better performance the computations can use one of the highly optimized vector math libraries that were implemented using assembly language constructs tuned specifically for the Xeon Phi architecture, or very well vectorized code.
Also, local CPU caches should be used as much as possible. Maintaining locality of data in caches is a key factor to achieve performance. This is a major difficulty because the L2 cache is about 25 MB over all 60 cores on the Xeon Phi (and over the possibly 240 threads), which means much less cache memory per core than on the Sandy Bridge host. One can already notice the impressive factor four in memory bandwidth and factor 5.8 in peak performance which separates the Sandy Bridge (single socket) from the Xeon Phi. On the other hand, the memory per core shrinks by a factor 30 from 4GB/core to 130MB/core. Additionally, the peak performance per core also shrinks from 21.6 GFlops/s to 16.6 GFlop/s. These peak performance numbers both assume the usage of the vectorized fused multiply-add assembly instruction, i.e. one addition and one multiplication are performed on all the elements of the vector registers given in parameter.
As we will see, to reach a certain level of performance, the parallelization effort is much more important on the Xeon Phi (KNC) compared to Sandy Bridge.

### 4.3.2 Application framework

**Numerical scheme & algorithmic analysis**

We have chosen to present here configurations with only a single $\mu = 0$ value. It means, we consider the 4D drift-kinetic model which is the backbone of the 5D gyrokinetic models and relevant to build numerical schemes. To simplify the analysis and avoid possible problems directly due to MPI communication that had bad performance on this KNC cluster, we considered a mini-application that does not use MPI at all and is only parallelized using the OpenMP paradigm. Such a mini-app offers more flexibility than the original application while retaining key algorithms and performance characteristics [148].
We now investigate an alternative of the the classical Strong splitting of Gysela with a a series of directional advections: $(\hat{v_\parallel}/2, \hat{\varphi}/2, \hat{r\theta}, \hat{\varphi}/2, \hat{v_\parallel}/2)$. This alternative consists in 4D advection, Algo. 26 is very close to Algo. 11 presented earlier (p.64). The OpenMP parallelization and SIMD vectorization is one of the challenge together with the fact this kernel is computation intensive.

$\eta(r = *, \theta = *, \varphi = *, v_\parallel = *) \leftarrow$ compute spline coeff. from the
    4D function $f^n(r = *, \theta = *, \varphi = *, v_\parallel = *)$;
**for** *All grid points* $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})$ **do**
    $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})^\star \leftarrow$ foot of characteristic that ends at $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})$ ;
    Interpolate $f^n$ at location $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})^\star$ using $\eta$ coeff.;
    $f^{n+1}(r_i, \theta_j, \varphi_k, v_{\parallel\, l}) \leftarrow$ the interpolated value;

**Algo. 26:** 4D advection scheme with semi-Lagrangian scheme

Let us briefly summarize the algorithmic complexity of the two approaches: splitting scheme (denoted *Split*), 4D advection scheme (denoted *Nosplit*). To oversimplify this short analysis, we will focus only on the interpolation operator, and the spline coefficients computation. We leave aside the

cost due to the computation of the feet of the characteristics. Also, we assume that we have a constant displacement field for the advection (no costly computation required). Let us denote $N_{all}$ the number of points in the domain, *i.e.* $N_{all} = N_r\, N_\theta\, N_{v_\parallel}\, N_\varphi$.
For the *Split* case, each 1D advection requires: $7\, N_{all}$ floating point operations (it means 3 additions and 4 multiplications per grid point and $N_{all}$ read memory accesses, and the same amount of write memory accesses. The 1D spline coefficients derivation costs $10\, N_{all}$ FLOP (a solver performs a single sweep down and up, using a small LU decomposition). We assume that each advection is performed with a *very good* locality in cache memory that prevents from loading the same memory reference two times *during a single advection*, especially the spline coefficients remain in the cache. The 2D advection implies: $35\, N_{all}$ FLOP, $N_{all}$ read memory accesses, and the same amount of write memory accesses. The 2D spline coefficients derivation leads to $10\, N_{all}$ FLOP. Finally, the splitting scheme, with 5 advections, has the following cost: $113\, N_{all}$ FLOP, $5\, N_{all}$ in both read and write memory accesses.
Concerning the *NoSplit* case, the computational cost of the 4D interpolator is quite high: $595\, N_{all}$ FLOP. The spline coefficients derivation using a tensor product in four dimensions leads to $20\, N_{all}$ FLOP. Considering memory accesses, the distribution function $f^n$ is accessed at least one time to compute the spline coefficients. The spline coefficients are first written (lines 1 and 2 of Algo. 26) and then read in order to compute the interpolations. Finally, the interpolation result is written (line 6). So in total, the algorithm performs $2\, N_{all}$ read memory accesses (at least) and $2\, N_{all}$ write memory accesses, and means $615\, N_{all}$ FLOP.
If one compares both methods, the *Nosplit* case computes 6 times more FLOP but performs less memory accesses. At first approximation, the 1D and 2D advections are mainly memory-bound kernels (as we will see afterwards), whereas the 4D approach is clearly CPU-bound. For computing units that can perform a very large number of FLOP per transferred byte (as the Xeon Phi is), the 4D approach is better suited than the splitting approach. In the following, we will mainly target the 4D algorithm for the porting on Xeon Phi which seems to be best suited.

#### Profiling and improvement strategy

In practice, we have used the `OpenMP` paradigm on a shared memory node throughout all the codes presented hereafter. A small run with this mini-app using 4D advections on one single Sandy Bridge node takes 10 hours for the following domain size: $N_r = 128, N_\theta = 256, N_\varphi = 32, N_{v_\parallel} = 64$ with 4000 time steps. A profiling of the application using a single process with 16 threads has been performed on the Sandy Bridge partition with the Scalasca toolkit. As a result 98% of the run time is concentrated in one single routine and can be split into:

- Spline construction 10%
- Vlasov Solver (4D advection) 89% split into:
  - Computation of the feet of the characteristics 41%
  - **Interpolations with 4D stencil** using spline coefficients and feet 59%

We first tried a *top-down* approach to improve the performance of the 4D interpolation kernel. We extracted the most computation intensive part of the kernel in a single subroutine of less than 200 lines of code. Then, a lot of optimization techniques were carried on on this subroutine to get a vectorization of parts of the kernel and cache-friendly behavior. We have sped up the initial version by a factor 2 on medium test cases on the Xeon Phi (domain size $128 \times 128 \times 64 \times 32$, 45s for one advection step with the initial version and 25s after some improvements have been done). Nevertheless, on a Sandy node with 16 cores and a similar test case, the code is eight times *faster* (3.3s per advection step) than on one Xeon Phi (KNC) device (25s). Even though we have explored many techniques, we failed to reach the level of performance one can expect from Xeon Phi (*i.e.* Sandy should have been two times *slower* than Phi on well optimized/vectorized code).
Therefore, we switched to a different approach further discussed in the following. In this *bottom-up* approach, we start with the study of small kernels that can reach high performance on Phi. And then, from this experience we build up more complex kernels while trying to keep the same level of performance.

### 4.3.3 Evaluating Xeon Phi with simple kernels

#### Description of simple advection kernels

The choice of interpolation method in a semi-Lagrangian scheme is crucial. It determines the numerical quality of the scheme and its computational cost. The tensor product is employed in this work to achieve multi-dimensional interpolations. Therefore, we need to fix first the method for the 1D

interpolation. We will investigate the cubic spline scheme, which is known to be a well-balanced compromise between cost and quality for plasma simulations. Nevertheless, cubic splines lead to rather complex computational kernel. Thus, we have chosen to first look at Lagrange polynomials. The Lagrange interpolator is simpler and thus gives us the opportunity to easily investigate several optimization alternatives. In addition the $3^{th}$ degree (4 points) Lagrange polynomial is close to cubic spline in terms of computational cost (except that we do not need to compute spline coefficients) and in terms of data access pattern. That will give us hints to optimize cubic splines afterwards. We will focus on 4 kernels based on Lagrange polynomials of order 4 using tensor product in 1, 2, 3 and 4 dimensions. Then, we will tackle 4D spline interpolation. We intend to evaluate the interpolation kernels within the semi-Lagrangian scheme. In order to address this issue, we define a much simpler equation than the gyrokinetic setting described earlier as well as an analytical test case. The advection equation that we consider is the following (with $D \in [1..4]$ depending on the kernel under evaluation):

$$\partial_t f + v\,\partial_x f = 0, \quad x \in [0, 2\pi]^D, \quad t \geq 0\,. \tag{4.7}$$

We assume a constant velocity field $v = \begin{pmatrix} v_1 & \dots & v_D \end{pmatrix}^T$.

The simplest interpolation kernel is shown on Fig. 82. The function `access_f` is an accessor to get/set the value of a distribution function. The input distribution function is `f0` (at time $t$), the output is `f1` (time $t + \Delta t$). The variables `coeff[1-4]` are set depending on the velocity field. For the sake of simplicity, the velocity is assumed to be small (or $\Delta t$ small) in order to have a foot of characteristic in the left or right cell near the departure point $(x_1, x_2, x_3, x_4)$. The kernel in Fig. 83 performs the same calculation as Fig. 82 but using dedicated intrinsics [153], that enables to use Xeon Phi SIMD instruction set.

```
#pragma omp parallel for collapse(3)
 for (x1=0; x1<Nx1; x1++) {
  for (x2=0; x2<Nx2; x2++) {
   for (x3=0; x3<Nx3; x3++) {
#pragma vector nontemporal (f1)
#pragma vector always
    for (x4=0; x4<Nx4; x4++) {
     access_f(f1,x4,x3,x2,x1) = // OUTPUT data f1
       coef1 * access_f(f0,x4-1,x3,x2,x1) + // INPUT data f0
       coef2 * access_f(f0,x4  ,x3,x2,x1) +
       coef3 * access_f(f0,x4+1,x3,x2,x1) +
       coef4 * access_f(f0,x4+2,x3,x2,x1);
    } } } }
```

Figure 82: Lagrange 1D code - with directives

```
#pragma omp parallel for collapse(3)
 for (x1=0; x1<Nx1; x1++) {
  for (x2=0; x2<Nx2; x2++) {
   for (x3=0; x3<Nx3; x3++) {
    for (x4=0; x4<Nx4; x4++) {
      ptread = &(acces_f(f0,x4,x3,x2,x1));
      // read input data
      tmpr2 = _mm512_load_pd  (ptread);
      tmpr1 = _mm512_loadunpacklo_pd(tmpr1, ptread-1);
      tmpr1 = _mm512_loadunpackhi_pd(tmpr1, ptread-1+8);
      tmpr3 = _mm512_loadunpacklo_pd(tmpr3, ptread+1);
      tmpr3 = _mm512_loadunpackhi_pd(tmpr3, ptread+1+8);
      tmpr4 = _mm512_loadunpacklo_pd(tmpr4, ptread+2);
      tmpr4 = _mm512_loadunpackhi_pd(tmpr4, ptread+2+8);
      // 1+2+2+2=7 flop per loop iteration
      tmpw = _mm512_mul_pd(tmpr1,coeff1);
      tmpw = _mm512_fmadd_pd(tmpr2,coeff2,tmpw);
      tmpw = _mm512_fmadd_pd(tmpr3,coeff3,tmpw);
      tmpw = _mm512_fmadd_pd(tmpr4,coeff4,tmpw);
      // write output data
     _mm512_store_pd (&(access_f(f1,x4,x3,x2,x1)), tmpw);
```

Figure 83: Lagrange 1D code - with intrinsics

Fig. 84 shows the interpolation with the 2D tensor product of Lagrange polynomial (order 3). The number of FLOP grows from 7 in the inner loop of Fig. 82 to 35 in Fig. 84.

```
#pragma omp parallel for collapse(3)
 for (x1=0; x1<Nx1; x1++) {
  for (x2=0; x2<Nx2; x2++) {
   for (x3=0; x3<Nx3; x3++) {
#pragma vector nontemporal (f1)
#pragma vector always
    for (x4=0; x4<Nx4; x4++) {
     access_f(f1,x4,x3,x2,x1) =
       coefb1 * (coefa1 * access_f(f0,x4-1,x3-1,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3-1,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3-1,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3-1,x2,x1)  ) +
       coefb2 * (coefa1 * access_f(f0,x4-1,x3  ,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3  ,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3  ,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3  ,x2,x1)  ) +
       coefb3 * (coefa1 * access_f(f0,x4-1,x3+1,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3+1,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3+1,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3+1,x2,x1)  ) +
       coefb4 * (coefa1 * access_f(f0,x4-1,x3+2,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3+2,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3+2,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3+2,x2,x1)  )
    } } } }
```

Figure 84: Lagrange 2D code - with directives

**Benchmark of simple kernels an solutions to get fast kernels**

We compare the performance of several kernels on a Xeon Phi(KNC) in native mode, against one Sandy Bridge node (two sockets, 16 cores). In theory, the Phi can outperform Sandy by a factor 3 on compute-bound kernels (considering ratio of peak processing performance, the peak is established according to INTEL specification sheet), and a factor 2 for memory-bound kernels (ratio of the STREAM benchmark performance). Table 19 shows the obtained results. The first two kernels (Lagrange 1D

| Kernel/archi. | | Processing perf. | | Memory bandwidth | |
|---|---|---|---|---|---|
| Lagrange 1D | **Phi** | 46 GFLOPS | (5% peak) | 106 GB/s | (81% STREAM) |
| | **Sandy** | 25 GFLOPS | (7% peak) | 57 GB/s | (81% STREAM) |
| Lagrange 2D | **Phi** | 250 GFLOPS | (25% peak) | 111 GB/s | (85% STREAM) |
| | **Sandy** | 134 GFLOPS | (39% peak) | 59 GB/s | (84% STREAM) |
| Lagrange 3D | **Phi** | 228 GFLOPS | (23% peak) | 25 GB/s | (19% STREAM) |
| | **Sandy** | 156 GFLOPS | (46% peak) | 17 GB/s | (25% STREAM) |
| Lagrange 4D | **Phi** | 160 GFLOPS | (16% peak) | 4.3 GB/s | (3.3% STREAM) |
| | **Sandy** | 145 GFLOPS | (42% peak) | 3.9 GB/s | (5.6% STREAM) |

Table 19: Performance of interpolation kernels

and 2D) are dominated by the memory bandwidth constraint. They achieve on both architectures more than 80% of the maximal bandwidth, which is very satisfactory. The measurement of maximal bandwidth was established with STREAM benchmark (TRIAD operator) and we measure 70 GB/s on Sandy, 130 GB/s on Phi. Practically, one can thus achieve the expected speedup of Phi over Sandy on a memory-bound kernel.
The 3D and 4D interpolation kernels are clearly compute-bound, they have several hundreds of FLOP per grid points to perform. The Lagrange 3D implementation has good performance, but the Phi gets a 1.5 speedup over Sandy which is not the expected 2 or 3. The Lagrange 4D kernel is not fast enough for the Phi implementation. The performance is close to the Sandy one, while one can expect a good speedup for this kind of kernel (Phi was designed to tackle computation intensive algorithms). By replacing the complex accesses for memory reads by simpler but fake accesses, performance is improved a lot (but the results are wrong of course). This simple test shows that a major problem comes from the 4D stencil that implies complex memory access pattern. Several techniques have been tested to optimize and tune the performance of the interpolation kernels and to achieve the results obtained in Table 19. In the following we summarized quickly some of the different investigated techniques and approaches to fasten the code.
For *Memory-bound* kernels, prefetch mechanism is important. In practice, it means loading data in advance thanks to ad-hoc directives or intrinsics. It reveals to be crucial to scan of a large set of possible combinations of prefetch parameters. It is also interesting to tune the kernels in order to work on aligned data. We have studied the memory access pattern in the inner loop to avoid any cache trashing, this is also a key factor. On Phi, we can observe large variations of execution time from one run to the other (10% is common). Then, interpreting the impact of any optimization requires to get good statistics with a large number of runs.
For *Compute-bound* kernels, Cache blocking (loop tiling) is crucial to save computation time. As caches are small on Phi, designing algorithms that are cache friendly is quite a hard task. Nevertheless, we manage to save execution time with proper loop tiling on the 3D kernels (50% gain). Splitting the body of a 10-40 lines loop into multiple loops of less than 10 lines of code each (if it is semantically correct) has led to effective speedups. The positive effects are much stronger on Phi than on Sandy Bridge. For these 3D and 4D kernels, we observe that the best performance is achieved on Phi with 170 up to 240 threads. This very fine grain parallelism has to be compared to the 16 threads that are sufficient on Sandy (2 sockets). Small modifications in the code can lead to vectorization with bad performance at some location. Whenever this kind of event happens, the code can slow down by a factor of 4 suddenly. Sensitivity to compiler version and vectorization problem is really high, as far as performance is concerned.

**Discussion**

A partial conclusion of this study is that achieving performance even on a simple kernel is a challenge on Phi (KNC). Compared to the relative easiness to get a reasonably efficient code on Sandy, the programmer has a lot of constraints to fulfill on Phi. The developer has to interact finely with the compiler and profiling tools to improve the performance to an acceptable level. If it is possible to take care of some of the important points on a simplified kernel code, it can be very difficult to satisfy

some constraints throughout a long program. The non exhaustive list of theses points could be: good vectorization, fine grain parallelism (threads), data alignment, cache blocking (adapted to small caches), prefetching, loop splitting. Applying these hints in a long code, in which the computation time is distributed over several routines, is not simple. These optimization issues can pose a serious challenge to programmers that want to fully use the hardware.

### 4.3.4 Evaluating Xeon Phi with a Gysela kernel

#### Description of Gysela kernel

This part builds up on the experience gained on simple kernels to target a more realistic kernel that uses a 4D tensor product of splines instead of Lagrange polynomials and a variable displacement field for the feet of characteristics. The kernel we design in this section is not a synthetic kernel anymore but is instead a complete implementation of Algo. 26 (p.103) intended to be included in the Gysela code. Designing a complete kernel increases the amount of code that has to be optimized compared to the small (typically less than 150 lines of code) kernels, this is the first source of increased complexity. Another complexity comes from the use of a variable displacement field for the feet of the characteristics. In the previous simple kernels we took purposely a very small displacement. This shift generates an indirect access which induces a cost and prevent from using efficient prefetching. Also, this computation of the foot mixes floating-point (for the values) and integer (for the location) calculations that depend on each other. A sketch of the code is presented in Fig. 85 for the outer loops and Fig. 86 for the innermost kernel.

```
1  do ith_blk=0, nb_blk_th ! loop blocking in theta
2   do ivpar=0, Nvpar
3    call feet_computations_with_openmp(...)
4   !$OMP PARALLEL DO COLLAPSE(2)
5     do iphi=0, iNphi
6       do ith=ith_blk*th_bsize,(ith_blk+1)*th_bsize-1
7         call interpolations_vectorized_kernel(...);
8       end do
9     end do
10  end do
11 end do
```

Figure 85: Advection kernel external loops

```
1  #define R_BSIZE 8
2  subroutine interpolations_vectorized_kernel(...,spline coeff.)
3   do ir_outer=0, Nr, R_BSIZE
4     ! retrieve grid cell containing the foot, compute spline basis
5  !dir$ simd
6     do ir_inner=0,R_BSIZE-1
7       ir=ir_outer+ir_inner
8       r_foot=...; th_foot=...; vpar_foot=...; phi_foot=...;
9       ir_star      = map_on_grid(r_foot)
10      ith_star     = map_on_grid(th_foot)
11      ivpar_star   = map_on_grid(vpar_foot)
12      iphi_star    = map_on_grid(phi_foot)
13      sbasis(1:16) = compute_spline_basis(*_star,*_foot)
14    end do
15    ! interpolate in combining spline basis and spline coeff.
16    psum(0:R_BSIZE-1) = 0.
17    do <nest_of_four_loops>
18 !dir$ simd
19      do ir_inner=0,R_BSIZE-1
20          coeff = load spline coeff. located at *_star (with unit stride)
21          psum(ir_inner) = psum(ir_inner) + coeff(...) * sbasis(...)
22      end do
23    end do
24    f1(ir_outer:ir_outer+R_BSIZE-1,ith,iphi,ivpar)=psum(0:R_BSIZE-1)
25  end do
26 end subroutine interpolations_vectorized_kernel
```

Figure 86: Advection kernel inner loop nest

For the design of the advection kernel, we have several constraints coming from the Gysela mini-app and from the Phi hardware. We targeted a set of good properties: cache friendly, fine grain parallelism, computations in the innermost loop that are well vectorized by SIMD directives. Regarding the part of the kernel presented in Fig. 85, the strategies applied include a blocking for the loop in the $\theta$ dimension. It has been split in two loops (lines 1 and 6) in order to improve cache locality. The outermost loops along $\theta$ blocks (line 1) and $v_{\|}$ (line 2) directions are not parallelized with `OpenMP`, instead, the loops along $\varphi$ (line 5) and inside blocks of $\theta$ (line 6) are. Thus, several data are shared between the threads that fit into cache memory, which is not the case if parallelization would occur at outer loops. This improves the use of cache memory (in the inner loops) which is a crucial element for the Phi. Similarly, we interlaced inside this loop nest the **feet computations** and the **interpolations** in order to reuse the data shared by these routines and to improve temporal locality. Due to the high number of threads on Phi, getting sufficient loop counts requires the domain size along the OpenMP parallelized dimensions to be large enough.

The part of the kernel presented in Fig. 86 corresponds to the part executed by a single thread where the parallelism comes for SIMD vectorization. The loop along $r$ has been blocked with the iteration along $r$ blocks appearing at line 3 so as to help vectorization by having the innermost loop the same size as that of a processor vector. This vectorization is further favored by iterating over small arrays the size of a vector such as `psum` (line 16) in the code to accumulate results inside a loop. The upper bound of the loop along $r$ blocks (line 3) is fixed at compile time which accelerates computations on Phi a lot (not true on Sandy). The body of the iterations inside these blocks has

been split in two parts (lines 6 and 19) which enables the compiler to apply better optimizations as seen in the previous section. The loading of spline coefficients (line 20) has also been tuned in order to read in memory with unit-stride. Finally, the choice amongst the possible SIMD directives (*i.e.* `vector` or `simd`) has been made by testing both as their impact on performance is hard to predict.

**Benchmark**

In order to evaluate this kernel in a way that can be compared with the previous section, we first focus on the interpolation part only (spline coefficients are computed during initialization). To reach that goal, simplified precomputed feet are used which lead to incorrect results but induce no computing cost. Table 20 shows the results of a benchmark with a domain of size $N_r = 128, N_\theta = 128, N_\varphi = 128$, $N_{v_\parallel} = 64$. 80 GFLOPS are obtained on Phi which is more than twice the 33 GFLOPS obtained on Sandy. Given the ratio of performance between the two architecture this shows that the optimization for the Phi are indeed well done. Reaching this level of performance however required a huge investment in time. The reason was that each optimization reduces execution times just a little bit (sometimes it is even hard to measure). But once all of these optimization are active simultaneously, there is a net performance improvement. The percentage of the theoretical attainable hardware peak is

| Kernel/archi. | | Processing perf. | | Memory bandwidth | |
|---|---|---|---|---|---|
| Advec 4D (without foot comp.) | **Phi** | 80 GFLOPS | (7% peak) | 2.7 GB/s | (2% STREAM) |
| | **Sandy** | 33 GFLOPS | (9% peak) | 1.1 GB/s | (1.6% STREAM) |

Table 20: Performance of advection kernel (236 threads on Xeon Phi and 16 on Sandy Bridge)

however not as good as that of the kernels presented in the previous section. This is most likely due to the integer computations and memory indirections required for this version. In addition, all the memory accesses for the advection kernel cannot be well aligned (the foot can be located anywhere). It is worth noting that the present results have been obtained with a domain size well suited to the Phi. The performance in GFLOPS is decreasing for smaller domain sizes. This dependency on input parameters is not as sensitive on Sandy. The pressure and the constraints on the computing units and the cache hierarchy seem to be higher on Phi than on Sandy and a small imbalance can lead to severe performance issue on Phi. Finally, additional issues come from the compiler version. From one version to another, the INTEL compiler does not employ the same SIMD instructions and optimizations for a given code. We have noted that a well optimized code can suffer a large slowdown (factor 3 has been observed) when changing the compiler version.

**Splitting versus no-splitting**

Let us now come back on the analysis done in subsection 4.3.2. The Strang splitting scheme with four 1D advections and one 2D advection (*Split*) is clearly memory-bound and this scheme has been used for a long time in GYSELA. On the other hand, the most recent 4D advection scheme (*Nosplit*) leads to a larger number of floating-point operations but requires ideally less memory transfers. Now that several tuning operations have been done on the 4D kernel, the *Nosplit* version has reached a quite high optimization level on Sandy Bridge architecture. After this work on the 4D kernel, we observe on Sandy Bridge that the *Nosplit* case is slightly faster than the *Split* case. The execution time is reduced by 1% to 20% depending on the domain size. The 4D advection is now an interesting solution because, despite its expensive computations, execution time is better than for the splitting scheme. This gain is even larger for large domain sizes. On the one hand, the *Nosplit* case exhibits much more compact kernels in term of the number of code lines. This is definitely an advantage from the software engineering point of view. On the other hand, the *Nosplit* case also has drawbacks. First, it requires temporary buffers to store the feet of the characteristics (extra memory consumption). Second, tiling sizes have to be specifically tuned for each new machine in order to benefit from maximal cache effects. Considering the communication costs of the *NoSplit* case in a future application using a MPI parallelization, as the parallel algorithm is not yet known, it is difficult to make accurate predictions. Nevertheless, the constraints and data dependencies are quite similar to those of the *Split* case, therefore we do not expect that the amount of communications will be completely different.

**Discussion**

The final obtained performance was satisfactory as the computing time for 4D kernel was twice as fast on Xeon Phi (KNC) as on a dual-socket Sandy bridge node. But the other parts of the GYSELA code are also difficult to vectorize/optimize/parallelize for the Xeon Phi. To port the whole production

version of GYSELA code on such architecture would require a large effort of overhauling the application. All along the optimization process we went through, the performance evaluation have been performed on both the Sandy Bridge and the Phi computing units. For the best cases with the simplest kernels, we managed to reach similar fractions of the peak performance on Phi as on Sandy Bridge. In other cases however, and especially when the code complexity builds up, we did not manage to reach that goal.
Regardless the final result, the investment in terms of code engineering to reach these performance levels is very large. The level of parallelism to extract is much higher than on Sandy Bridge: 177 and up to 236 OpenMP threads are required on Phi versus 16 on Sandy Bridge. Extracting this fine grain parallelism is difficult for some kernels, especially for small domain sizes. Many other aspects have to be looked at, including vectorization, prefetching, data alignment, cache locality, cache trashing and memory access patterns while these have much less impact on Sandy Bridge. In addition, compiler optimizations are very important to get good performance but are very difficult to predict. Some good practice can help such as loop splitting, but no golden rule can be extracted and in practice, one often has to look at the generated assembly code to assess its quality. This problem is further amplified by the fact that the same code can yield very different performance results depending on the compiler version. Part of these conclusions are also valid for Xeon Phi KNL, the latest version of Xeon Phi. However, good performance were much easier to achieve because of hardware improvements, and vectorizing compiler enhancement on KNL [77]. Such considerations are relevant to anticipate strategies for optimizing future exascale devices, and the main issue of performance portability.

# Conclusion

Exascale architectures are foreseen to be supercomputers based on different building blocks as the last three decades. Since several years, the number of many-core systems is rising, whether as accelerators or co-processors, which are mostly Nvidia GPUs or the Intel Xeon Phi[3]. This is driving related improvements in power efficiency, which is necessary in the run-up to exascale. Also, pressures on efficiency, performance, scalability, and programmability for such machines are mounting. Floating point performance-driven world are becoming less and less relevant, whereas metrics like data movement is already critical, and will continue to remain as such. Since 2010, most of the parallel applications do not make great use of the resources and run at a small fraction of the peak performance because all processing units are really complex to exploit: vectorization is required, complex memory hierarchy, communication and memory access costs become prominent, compiler tricks and sophisticated deployment strategy are key elements. For many production codes it will take years to restructure them for reaching efficiency on the new architectures.
We have reached a turning point. There are still a lot of unknowns for the upcoming architectures and programming models, some awaited difficulties, but also some trends that should continue in the years to come and that we may trust. The ratio between the computation capacity, the communication over the network, the memory bandwidth and latency evolves rapidly and the codes should follow and adapt their algorithms to these changes (*e.g.* communication-avoiding algorithms, communication-computation overlap). The use of SIMD instructions is clearly a tidal wave that programming models, compilers and applications have to handle carefully to eventually get performance [77]. Mixed-precision algorithms that segregate computations with single precision versus double precision typically are and will remain useful. These can be smartly combined with the vectorization and should bring some speedups if applicable. The collective communications and data movements have large relative costs that can be mitigated by data centric computing or task programming approaches. Also, depending of the targeted architecture, one will expect the granularity of computation can be tuned. This aspect has a large impact on the existing programs but also at the level of algorithms. New algorithms can already been designed to go in this direction. Portability of efficiency is also difficult to achieve, because hardware and systems have some fine specificities that have to be taken into account. One way to deal with this issue is to articulate the program around identified kernels. The algorithms of the kernels can be parameterized and auto-tuned through specific tools such as BOAST [191] for example. Thus, one can ensure that the most sensible parts can be well optimized. Another approach would be to use the principle of separation of concerns (SoC) that aims at separating a computer program into distinct sections. One section could be related to the logic and the algorithms, another section could describe the optimization or the details of the targeted hardware, another section could manage data movements. An added value is the ability to improve or modify one section of code without having to know the details of the other sections. Kokkos is representative of this kind of

[3]Intel has announced that it abandons its next-generation Xeon Phi chip, code-named Knights Hill, in favor of a new microarchitecture specifically designed for exascale. It is foreseeable than certain technologies used in Xeon Phi will be recycled in next generations of processors, and that many-core will not disapear.

approach and proposes a C++ library for HPC applications. One interesting point is the ability for testing many different data structures that are described in a specific section and optimizing them, without modifying the main program. Nowadays, large HPC codes are often employing a wide set of technologies, *e.g.* MPI calls, OpenMP or OpenACC directives, SIMD directive, IO library, Linear Algebra libraries, and so on. Efforts should be initiated to deal with code complexity, whether with new programming models, or with a set of good practices that permits to avoid all the issues of developing, maintaining a code, deploying it on the parallel machine with such a large number of products and dependencies. Also, I/O bottleneck in HPC is becoming worse as the amount of data flowing within application continues to grow. Applications rely on parallel file systems (PFS) to obtain good I/O performance. Recently two new players have come: SSD and NVRAM technologies. These fast, non-volatile storage technologies, can act as a cache for accessing PFS, but also as an extension of the main memory with different characteristics. In addition, such persistent storage should allow HPC applications to keep very large data sets temporarily before post-processing, helping for the so-called in-situ or in-transit processing. There is a need for having an abstract view of these memory and storage hierarchies in order to better manage the data transfer across the various levels, to achieve it more transparently and more efficiently compared to today. Efficient tools and software stacks are also lacking. Compression techniques can also be valuable and complementary in this context. Most of the previous topics related to exascale area are quite exciting and challenging. But as technology is moving fast, the adaptation of libraries, software stacks and applications should keep pace and be ready to adapt swiftly with the most current supercomputers.

# Chapter 5

# Perspectives

*"Every block of stone has a statue inside it*
*and it is the task of the sculptor to discover it."*
*Michelangelo*

**Efficiency, portability, readability**
One of the constant challenge facing the parallel application developer is to find a compromise between efficiency, portability and code readability. The complexities of hardware, of applications and the difficulty to choose a programming model have also to be taken into account. A possible solution to these many-faceted difficulties would be to choose a good enough programming model that largely abstracts the low-level architectural details to increase portability, readability. But this, without masking everything to the point of not being able to express effective algorithms and get performance. For production applications, one delicate question is the availability on supercomputers of the tools associated to this programming model, the quality of support, the guarantee of sustainability. I currently investigate a solution based on MPI + a task-based programming model (typically using OpenMP 4.5 and associated runtime) for Gysela that should replace the MPI+OpenMP (loop parallelism) approach which is of common use in many applications. Combined to this change, a structuring of the application around a set of well defined kernel is also of importance. It will permit specialization of some kernels for specific configurations, related to an application feature or related to hardware constraints. *PhD of N. Bouzat (2015-2018) focuses on this subject, other collaborations are related: J. Bigot (Maison de la Simulation), J . Richard (INRIA Avalon), J. Roman (INRIA Hiepacs) and M. Mehrenberger (INRIA Tonus).*

**Combining numerical schemes, switching kernels**
One may want to change the computation kernels according to some circumstances. For example, the selection criteria can be a function of space, a function of time or be based on performance measurements even other constraints of use (*e.g.* CFL or boundary conditions). The decision may be taken at compile time or dynamically during runtime. The semi-Lagrangian method is not optimal in some cases in terms of numerical precision and/or in terms of computational cost. A possible way to make a good use of this approach would be to have the Eulerian scheme and the semi-Lagrangian scheme implemented into different kernels. The switching between these would happen depending on a multiple criteria (space, time, performance, system dynamics, linear versus non-linear part). This kind of kernel breakdown and clever kernel selection had already proven to be a valuable choice in the oil application described in Section 4.1 for shortening execution time. This approach can be combined with auto-tuning techniques to automate the choice of kernels. Furthermore, next-generation systems demand a more detailed analysis of the interplay among execution time and other metrics. Metrics such as power, performance, energy may all be targeted together and traded against one another.

**Unsplit 4D advection**
The 4D advection operator for semi-Lagrangian simulation has already proved worthwhile several times and happens to be quite popular [7, 20, 158, 193] (see also Sections 3.1 and 4.3). This scheme is appealing on several aspects compared to Strang directional splitting: execution time, compactness, less sensitivity on the time step, higher computational intensity. Investigations should continue in order to establish if one can achieve good scalability at a large number of cores with this scheme, and estimate the benefit in term of accuracy.
In addition, I am considering the possibility to have mesh refinement in phase space and 4D advection can be a great lever. Mesh refinement in Gysela would permit more accurate description of the external radial boundary of the plasma, and less accuracy near the magnetic axis. This objective can help diminishing execution time, but more importantly it will enable physicists to model the scrape-off layer (SOL) where small grid cells are required. The 4D advection can be combined with Strang splitting for glueing patches with different resolutions together. It can also be a valuable choice to

replace completely the Strang splitting to avoid the hurdles associated with directional splitting in this setting.

**Shaped plasma**
Shaping of magnetic flux surfaces in tokamaks have an influence on turbulence. Magnetic field shaping in tokamaks is mainly due to elongation, triangularity, shift for the core plasma simulation. I have developed some techniques to upgrade GYSELA in order to take into account this refined geometry instead of circular plasma currently handled (see Section 3.3). Overhauling the application is targeted to go along this path. This improvement will allow for greater realism in term of physics. Non-uniform meshing will also allow for memory saving with an adaptive method which is also a topic I like [12,28,41,96]. *A collaboration is already underway with M. Mehrenberger (University of Strasbourg) and E. Sonnendrücker (IPP, Germany) to handle complex poloidal geometries [5, 17, 27, 81].*

**Kinetic electrons, huge simulations**
Considering kinetic electrons as an additional species in GYSELA leads to large costs. Computational grid has to be increased a lot and very fast electrons induces a sharp reduction of the time step. Field-aligned interpolations help to diminish a bit the grid size along $\varphi$ dimension (see Section 3.2) but even with this support, global full-f simulations are extremely demanding in term of CPU time. Introducing new levels of parallelism is mandatory to handle huge poloidal planes (at least $4096 \times 4096$ points in the $(r, \theta)$ plane typically). Investigation of parallel algorithms with domain decomposition along the 5 dimensions have began to prepare simulations with kinetic electrons. Furthermore, the collision operator will possibly exceed Vlasov in term of costs in this new setting, this may cause a paradigm shift for the parallelization choices. The new parallelization of the gyroaverage operator is a step towards this goal (see Section 2.6).

**Addressing new architectures**
Auto-tuning techniques can be employed to generate optimized versions automatically together with keeping readability of the code. A collaboration with INRIA Corse team has started on this subject to improve the portability of performance on several architectures of a reduced set of kernels [82]. Kernels should ideally be able to use several grain sizes in order to fit to architectures or runtimes that obtain good parallel performance with certain kernel sizes and/or with some knowledge on the shared resources. In this regards, CPU, GPU and many-core architectures have their own constraints. This implies that many algorithms have to be overhauled and *parameterized* with that in mind. A good target would be to improve performance portability through *automatic adaptation* of parallelism granularity depending on a platform description and/or auto-tuning techniques. All in all, adapting algorithms and numerical schemes is unavoidable to maintain the pace of performing well on modern architectures [2, 8, 18, 20, 26].

**Exascale**
One issue for upcoming architectures is about dealing with large data flow through deep storage hierarchy, which is a major topic whenever considering big amount of data and several levels of storage. A second issue concerns strategies and tools for task programming and load balancing on many-core's clusters. It is important to improve core utilization in dynamically revising the deployment of several application's parts. Vectorization and mixed precision algorithms are topics that I want to keep up exploring [30, 77]. Investigating *separation of concerns* (SoC) approaches for HPC is interesting to distinguish between the conception of algorithms on one side, to the effective memory access patterns on the other side. This could help a lot for designing and maintaining complex codes on complex machines, for improving readibility while accessing a good level of performance. Several fields are involved in building an application that satisfies all the constraints for exascale: application domain, mathematics, algorithmic, runtime systems, middleware, hardware. Much coordination and trade-offs are anticipated among the different domains in order to fulfill at best the numerous criteria.

Then, applications will need to be agile in evaluating and adopting technologies that are most promising along the way. It will even require *exploration* of new computing paradigms as we move to extreme parallelism and heterogeneity (all the parameters of the programing environment, and the specifications of the hardware is not yet fixed and known). Investing in new control layers and system software support (e.g., for asynchronous heterogeneous *tasking* and *data movement*) is critical for addressing the disruption of large on-node heterogenous parallelism [180]. One key component to help for this issue is the concept of *mini-application*. A mini-application attempts to capture meaningful aspects of a fraction of a large application. The goal is to provide a sufficiently small set of code lines (*e.g.* less than 2000 lines of code) in order to try different strategies, various algorithms or schemes quickly and at a relative low human cost. Indeed, this is much important to converge towards good solutions satisfying numerous constraints at a reasonable cost, but also to have means to keep the pace of change in software/hardware. The success of a mini-application framework lies in both *flexibility* and *simplicity* of execution. It is a self-contained program that embodies essential

performance characteristics of a part of the main application [148]. A well-written mini-app will allow that improvements obtained on it be included easily into the main application. One way to do that is to have common *kernels* (small sets of code lines encapsulating computationally intensive parts) shared by the mini-app and the main application. Lastly, preparing an application for a transition from petascale to exascale systems will require a large investment in terms of software research and development and agile capabilities.

# References

## Computer Science and Applied Math. journal papers


[1] G. Latu, M. Mehrenberger, Y. Güçlü, M. Ottaviani, and E. Sonnendrücker. Field-aligned interpolation for semi-Lagrangian gyrokinetic simulations. *Journal of Scientific Computing*, 74(3):1601–1650, Mar 2018.

[2] Y. Asahi, G. Latu, Takuya Ina, Yasuhiro Idomura, V. Grandgirard, and X. Garbet. Optimization of fusion kernels on accelerators with indirect or strided memory access patterns. *IEEE Trans. Parallel Distrib. Syst.*, 28(7):1974–1988, 2017. http://doi.ieeecomputersociety.org/10.1109/TPDS.2016.2633349.

[3] V. Grandgirard, J. Abiteboul, J. Bigot, Th. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, Ch. Ehrlacher, D. Esteve, X. Garbet, Ph. Ghendrih, G. Latu, M. Mehrenberger, Cl. Norscini, Ch. Passeron, F. Rozar, Y. Sarazin, E., A. Strugarek, and D. Zarzoso. A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207:35–68, 2016. https://doi.org/10.1016/j.cpc.2016.05.007.

[4] 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. https://doi.org/10.1002/cpe.3429.

[5] C. Steiner, M. Mehrenberger, N. Crouseilles, V. Grandgirard, G. Latu, and F. Rozar. Gyroaverage operator for a polar mesh. *Eur. Phys. J. D*, 69(1):18, 2015. https://doi.org/10.1140/epjd/e2014-50211-7.

[6] N. Crouseilles, M. Kuhn, and G. Latu. Comparison of numerical solvers for anisotropic diffusion equations arising in plasma physics. *J. Sci. Comput.*, 65(3):1091–1128, 2015. https://doi.org/10.1007/s10915-015-9999-1.

[7] G. Latu, V. Grandgirard V., J. Abiteboul, N. Crouseilles, G. Dif-Pradalier, X. Garbet, Ph. Ghendrih, M. Mehrenberger, Y. Sarazin, and E. Sonnendrücker. Improving conservation properties of a 5D gyrokinetic semi-Lagrangian code. *Eur. Phys. J. D*, 68(11):345, 2014. https://doi.org/10.1140/epjd/e2014-50209-1.

[8] R. Abdelkhalek, H. Calandra, O. Coulaud, G. Latu, and J. Roman. Fast seismic modeling and reverse time migration on a graphics processing unit cluster. *Concurrency and Computation: Practice and Experience*, 24(7):739–750, 2012. http://dx.doi.org/10.1002/cpe.1875.

[9] N. Crouseilles, G. Latu, and E. Sonnendrücker. A parallel Vlasov solver based on local cubic spline interpolation on patches. *J. Comput. Physics*, 228(5):1429–1446, 2009. http://dx.doi.org/10.1016/j.jcp.2008.10.041.

[10] V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Pradalier, Ph. Ghendrih, N. Crouseilles, G. Latu, E. Sonnendrücker, N. Besse, and P. Bertrand. Computing ITG turbulence with a full-f semi-Lagrangian code. *Communications in Nonlinear Science and Numerical Simulation*, 13(1):81 – 87, 2008. "Vlasovia 2006: The Second International Workshop on the Theory and Applications of the Vlasov Equation", https://doi.org/10.1016/j.cnsns.2007.05.016.

[11] N. Crouseilles, M. Gutnic, G. Latu, and E. Sonnendrüker. Comparison of two Eulerian solvers for the four-dimensional Vlasov equation: Part I and II. *Communications in Nonlinear Science and Numerical Simulation*, 13(1):88 – 99, 2008. Vlasovia 2006: The Second International Workshop on the Theory and Applications of the Vlasov Equation, http://dx.doi.org/10.1016/j.cnsns.2007.03.017.

[12] N. Besse, G. Latu, A. Ghizzo, E. Sonnendrücker, and P. Bertrand. A wavelet-MRA-based adaptive semi-Lagrangian method for the relativistic Vlasov-Maxwell system. *J. Comput. Physics*, 227(16):7889–7916, 2008. https://doi.org/10.1016/j.jcp.2008.04.031.

[13] S. Genaud, P. Gançarski, G. Latu, A. Blansché, C. Rattanapoka, and D. Vouriot. Exploitation of a parallel clustering algorithm on commodity hardware with P2P-MPI. *The Journal of Supercomputing*, 43(1):21–41, 2008. https://doi.org/10.1007/s11227-007-0136-2.

[14] N. Crouseilles, G. Latu, and Eric Sonnendrücker. Hermite spline interpolation on patches for parallelly solving the Vlasov-Poisson equation. *Applied Mathematics and Computer Science*, 17(3):335–349, 2007. http://dx.doi.org/10.2478/v10006-007-0028-x.

## Computer Science and Applied Math. conf. proceedings

[15] N. Bouzat, F. Rozar, G. Latu, and J. Roman. A new parallelization scheme for the hermite interpolation based gyroaverage operator. In *16th International Symposium on Parallel and Distributed Computing, ISPDC 2017, Innsbruck, Austria, July 3-6, 2017*, pages 70–77, 2017.

[16] G. Hautreux et al. Pre-exascale architectures: Openpower performance and usability assessment for french scientific community. In *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, pages 309–324, 2017.

[17] F. Rozar, C; Steiner, G. Latu, M. Mehrenberger, V. Grandgirard, J. Bigot, Th. Cartier-Michaud, and J. Roman. Optimization of the gyroaverage operator based on hermite interpolation. *ESAIM: Proc.*, 53:191–210, 2016. https://doi.org/10.1051/proc/201653012.

[18] G. Latu, J. Bigot, N. Bouzat, J. Giménez, and V. Grandgirard. Benefits of SMT and of parallel transpose algorithm for the large-scale Gysela application. In *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2016, Lausanne, Switzerland, June 8-10, 2016*, page 10, 2016. http://dl.acm.org/citation.cfm?id=2929912.

[19] J. Bigot, G. Latu, Th. Cartier-Michaud, V. Grandgirard, Ch. Passeron, and F. Rozar. An approach to increase reliability of HPC simulation, application to the Gysela5D. *ESAIM: Proc.*, 53:248–270, 2016. https://doi.org/10.1051/proc/201653015.

[20] G. Latu, M. Haefele, J. Bigot, V. Grandgirard, Th. Cartier-Michaud, and F. Rozar. Evaluating kernels on Xeon Phi to accelerate Gysela application. *ESAIM: Proc.*, 53:211–231, 2016. https://doi.org/10.1051/proc/201653013.

[21] 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 - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 545–557, 2015. https://doi.org/10.1007/978-3-662-48096-0_42.

[22] F. Rozar, G. Latu, and J. Roman. Achieving memory scalability in the Gysela code to fit exascale constraints. In *Parallel Processing and Applied Mathematics - 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013, Revised Selected Papers, Part II*, pages 185–195, 2013. https://doi.org/10.1007/978-3-642-55195-6_17.

[23] M. Kuhn, G. Latu, S. Genaud, and N. Crouseilles. Optimization and parallelization of Emedge3D on shared memory architecture. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, pages 503–510, 2013. https://doi.org/10.1109/SYNASC.2013.72.

[24] Th. Cartier-Michaud, Ph. Ghendrih, V. Grandgirard, and G. Latu. Optimizing the parallel scheme of the Poisson solver for the reduced kinetic code TERESA. *ESAIM: Proc.*, 43:274–294, 2013. https://doi.org/10.1051/proc/201343017.

[25] O. Thomine, J. Bigot, V. Grandgirard, G. Latu, Ch. Passeron, and F. Rozar. An asynchronous writing method for restart files in the Gysela code in prevision of exascale systems. *ESAIM: Proc.*, 43:108–116, 2013. https://doi.org/10.1051/proc/201343007.

[26] J. Bigot, V. Grandgirard, G. Latu, Ch. Passeron, F. Rozar, and O. Thomine. Scaling Gysela code beyond 32K-cores on Bluegene/Q. *ESAIM: Proc.*, 43:117–135, 2013. https://doi.org/10.1051/proc/201343008.

[27] J. Abiteboul, G. Latu, V. Grandgirard, A. Ratnani, E. Sonnendrücker, and A. Strugarek. Solving the Vlasov equation in complex geometries. *ESAIM: Proc.*, 32:103–117, 2011. https://doi.org/10.1051/proc/2011015.

[28] G. Latu. Sparse data structure design for wavelet-based methods. *ESAIM: Proc.*, 34:240–276, 2011. A course available at https://doi.org/10.1051/proc/201134005.

[29] G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable quasineutral solver for gyrokinetic simulation. In *PPAM (2)*, pages 221–231, 2011. http://dx.doi.org/10.1007/978-3-642-31500-8_23.

[30] G. Latu. Fine-grained parallelization of a Vlasov-Poisson application on GPU. In *Euro-Par Workshops*, pages 127–135, 2010. http://dx.doi.org/10.1007/978-3-642-21878-1_16.

[31] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu. Fast seismic modeling and reverse time migration on a GPU cluster. In *2009 International Conference on High Performance Computing & Simulation, HPCS 2009, Leipzig, Germany, June 21-24, 2009*, pages 36–43, 2009. https://doi.org/10.1109/HPCSIM.2009.5192786.

[32] R. Abdelkhalek, H. Calandra, O. Coulaud, G. Latu, and J. Roman. FDTD Based Seismic Modeling and Reverse Time Migration on a GPU Cluster. In *9th International Conference on Mathematical and Numerical Aspects of Waves Propagation - Waves 2009*, Pau, France, 2009. https://hal.inria.fr/inria-00407782.

[33] M. Campos Pinto, S. Jund, G. Latu, S. Salmon, and E. Sonnendrücker. Exact Charge Conservation in a High-Order Conforming Maxwell Solver coupled with Particles. In *9th International Conference on Mathematical and Numerical Aspects of Waves Propagation - Waves 2009*, Pau, France, 2009. https://hal.inria.fr/inria-00591052.

[34] M. Haefele, F. Zara, G. Latu, and J.-M. Dischler. A dedicated compression scheme for large multidimensional functions visualization. In *1st International Workshop on Super Visualization (IWSV08)*, Ile de Kos, Greece, June 2008. https://hal.inria.fr/inria-00591076.

[35] G. Latu, N. Crouseilles, V. Grandgirard, and E. Sonnendrücker. Gyrokinetic semi-Lagrangian parallel simulation using a hybrid OpenMP/MPI programming. In *PVM/MPI*, pages 356–364, 2007. http://dx.doi.org/10.1007/978-3-540-75416-9_48.

[36] V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Praladier, Ph. Ghendrih, N. Crouseilles, G. Latu, E. Sonnendrücker, N. Besse, and P. Bertrand. Gysela, a full-f global gyrokinetic semi-Lagrangian code for ITG turbulence simulations. In *Proceedings of Theory of Fusion Plasmas, Varenna*, 2006.

[37] M. Gutnic, M. Mehrenberger, E. Sonnendrücker, O. Hoenen, G. Latu, and E. Violard. Adaptive 2D Vlasov simulation of particle beams. In *Proceedings of ICAP 2006*, 2006. epaper.kek.jp/ICAP06/PAPERS/THMPMP02.PDF.

[38] G. Tessier, J. Roman, and G. Latu. Hybrid MPI-Thread implementation on a cluster of SMP nodes of a parallel simulator for the propagation of powdery mildew in a vineyard. In *High Performance Computing and Communications, Second International Conference, HPCC 2006, Munich, Germany, September 13-15, 2006, Proceedings*, pages 833–842, 2006. https://doi.org/10.1007/11847366_86.

[39] A. Calonnec, G. Latu, J.-Marc Naulin, J. Roman, and G. Tessier. Parallel simulation of the propagation of powdery mildew in a vineyard. In *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, volume 3648 of *Lecture Notes in Computer Science*, pages 1254–1264, 2005. https://doi.org/10.1007/11549468_137.

[40] E. Sonnendrücker, M. Gutnic, M. Haefele, G. Latu, and J.L. Lemaire. Vlasov Simulation of Beams and HALO. In *Proceedings of the Particle Accelerator Conference, 2005*, pages 581–585, 2005.

[41] M. Haefele, G. Latu, and M. Gutnic. A parallel Vlasov solver using a wavelet based adaptive mesh refinement. In *34th International Conference on Parallel Processing Workshops (ICPP 2005 Workshops), 14-17 June 2005, Oslo, Norway*, pages 181–188, 2005. https://doi.org/10.1109/ICPPW.2005.13.

## Plasma Physics journal papers

[42] G. Dif-Pradalier, E. Caschera, Ph. Ghendrih, Y. Asahi, P. Donnel, X. Garbet, V. Grandgirard, G. Latu, Cl. Norscini, and Y. Sarazin. Evidence for global edge-core interplay in fusion plasmas. *Plasma and Fusion Research*, 12:1203012–1203012, 2017.

[43] Y. Asahi, V. Grandgirard, Y. Idomura, X. Garbet, G. Latu, Y. Sarazin, G. Dif-Pradalier, P. Donnel, and Ch. Ehrlacher. Benchmarking of flux-driven full-f gyrokinetic simulations. *Physics of Plasmas*, 24(10):102515, 2017.

[44] G. Dif-Pradalier, G. Hornung, X. Garbet, Ph. Ghendrih, V. Grandgirard, G. Latu, and Y. Sarazin. The ExB staircase of magnetised plasmas. *Nuclear Fusion*, 57(6):066026, 2017.

[45] D. Esteve, Y. Sarazin, X. Garbet, V. Grandgirard, S. Breton, P. Donnel, Y. Asahi, C. Bourdelle, G.Dif-Pradalier, C. Ehrlacher, C. Emeriau, Ph. Ghendrih, C. Gillot, G. Latu, and C. Passeron. Self-consistent gyrokinetic modeling of neoclassical and turbulent impurity transport. *Nuclear Fusion*, 58(3):036013, 2018.

[46] D. Zarzoso, P. Migliano, V. Grandgirard, G. Latu, and Ch. Passeron. Nonlinear interaction between energetic particles and turbulence in gyro-kinetic simulations and impact on turbulence properties. *Nuclear Fusion*, 57(7):072011, 2017.

[47] J. A. Morales, M. Becoulet, X. Garbet, F. Orain, G. Dif-Pradalier, M. Hoelzl, S. Pamela, G. T. A. Huijsmans, P. Cahyna, A. Fil, E. Nardon, Ch. Passeron, and G. Latu. Edge localized mode rotation and the nonlinear dynamics of filaments. *Physics of Plasmas*, 23(4):042513, 2016.

[48] Th. Cartier-Michaud, Ph. Ghendrih, Y. Sarazin, J. Abiteboul, H. Bufferand, G. Dif-Pradalier, X. Garbet, V. Grandgirard, G. Latu, Cl. Norscini, Ch. Passeron, and P. Tamain. Projection on proper elements for code control: Verification, numerical convergence, and reduced models. application to plasma turbulence simulations. *Physics of Plasmas*, 23(2):020702, 2016.

[49] D. Estève, X. Garbet, Y. Sarazin, V. Grandgirard, T. Cartier-Michaud, G. Dif-Pradalier, Ph. Ghendrih, G. Latu, and Cl. Norscini. A multi-species collisional operator for full-f gyrokinetics. *Physics of Plasmas*, 22(12):122506, 2015. `http://dx.doi.org/10.1063/1.4937373`.

[50] G. Dif-Pradalier, G. Hornung, Ph. Ghendrih, Y. Sarazin, F. Clairet, L. Vermare, P.-H. Diamond, J. Abiteboul, T. Cartier-Michaud, C. Ehrlacher, D. Estève, X. Garbet, V. Grandgirard, O.-D. Gürcan, P. Hennequin, Y. Kosuga, G. Latu, P. Maget, P. Morel, C. Norscini, R. Sabot, and A. Storelli. Finding the Elusive ExB Staircase in Magnetized Plasmas. *Physical Review Letters*, 114:085004, 2015.

[51] F. Orain, M. Bécoulet, G. T. A. Huijsmans, G. Dif-Pradalier, M. Hoelzl, J. Morales, X. Garbet, E. Nardon, S. Pamela, Ch. Passeron, G. Latu, A. Fil, and P. Cahyna. Resistive reduced MHD modeling of multi-edge-localized-mode cycles in tokamak X-Point plasmas. *Phys. Rev. Lett.*, 114:035001, Jan 2015.

[52] Ph. Ghendrih, Cl. Norscini, Th. Cartier-Michaud, G. Dif-Pradalier, J. Abiteboul, Y. Dong, X. Garbet, O. Gürcan, P. Hennequin, V. Grandgirard, G. Latu, P. Morel, Y. Sarazin, A. Storelli, and L. Vermare. Phase space structures in gyrokinetic simulations of fusion plasma turbulence. *The European Physical Journal D*, 68(10):303, 2014.

[53] M. Bécoulet, F. Orain, G. T. A. Huijsmans, S. Pamela, P. Cahyna, M. Hoelzl, X. Garbet, E. Franck, E. Sonnendrücker, G. Dif-Pradalier, Ch. Passeron, G. Latu, J. Morales, E. Nardon, A. Fil, B. Nkonga, A. Ratnani, and V. Grandgirard. Mechanism of edge localized mode mitigation by resonant magnetic perturbations. *Phys. Rev. Lett.*, 113:115001, Sep 2014.

[54] Ph. Ghendrih, G. Dif-Pradalier, Cl. Norscini, Th. Cartier-Michaud, D. Estève, X. Garbet, V. Grandgirard, G. Latu, Ch. Passeron, and Y. Sarazin. Self organisation of plasma turbulence: impact on radial correlation lengths. *Journal of Physics: Conference Series*, 561(1):012008, 2014.

[55] J. Abiteboul, Ph. Ghendrih, V. Grandgirard, Th. Cartier-Michaud, G. Dif-Pradalier, X. Garbet, G. Latu, Ch. Passeron, Y. Sarazin, A. Strugarek, O. Thomine, and D. Zarzoso. Turbulent momentum transport in core tokamak plasmas and penetration of scrape-off layer flows. *Plasma Physics and Controlled Fusion*, 55(7):074001, 2013.

[56] A. Strugarek, Y. Sarazin, D. Zarzoso, J. Abiteboul, A. S. Brun, Th. Cartier-Michaud, G. Dif-Pradalier, X. Garbet, Ph. Ghendrih, V. Grandgirard, G. Latu, Ch. Passeron, and O. Thomine. Unraveling quasiperiodic relaxations of transport barriers with gyrokinetic simulations of tokamak plasmas. *Phys. Rev. Lett.*, 111:145001, Oct 2013.

[57] A. Strugarek, Y. Sarazin, D. Zarzoso, J. Abiteboul, A. S. Brun, Th. Cartier-Michaud, G. Dif-Pradalier, X. Garbet, Ph. Ghendrih, V. Grandgirard, G. Latu, Ch. Passeron, and O. Thomine. Ion transport barriers triggered by plasma polarization in gyrokinetic simulations. *Plasma Physics and Controlled Fusion*, 55(7):074013, 2013.

[58] F. Orain, M. Becoulet, G. Dif-Pradalier, G. Huijsmans, S. Pamela, E. Nardon, Ch. Passeron, G. Latu, V. Grandgirard, A. Fil, A. Ratnani, I. Chapman, A. Kirk, A. Thornton, M. Hoelzl, and P. Cahyna. Non-linear magnetohydrodynamic modeling of plasma response to resonant magnetic perturbations. *Physics of Plasmas*, 20(10):102510, 2013.

[59] D. Zarzoso, Y. Sarazin, X. Garbet, R. Dumont, A. Strugarek, J. Abiteboul, Th. Cartier-Michaud, G. Dif-Pradalier, Ph. Ghendrih, V. Grandgirard, G. Latu, Ch. Passeron, and O. Thomine. Impact of energetic-particle-driven geodesic acoustic modes on turbulence. *Phys. Rev. Lett.*, 110:125002, Mar 2013.

[60] X. Garbet, D. Esteve, Y. Sarazin, J. Abiteboul, C. Bourdelle, G. Dif-Pradalier, Ph. Ghendrih, V. Grandgirard, G. Latu, and A. Smolyakov. Turbulent acceleration and heating in toroidal magnetized plasmas. *Physics of Plasmas*, 20(7):072502, 2013.

[61] R. J. Dumont, D. Zarzoso, Y. Sarazin, X. Garbet, A. Strugarek, J. Abiteboul, Th. Cartier-Michaud, G. Dif-Pradalier, Ph. Ghendrih, J-B. Girardo, V .Grandgirard, G. Latu, Ch. Passeron, and O. Thomine. Interplay between fast ions and turbulence in magnetic fusion plasmas. *Plasma Physics and Controlled Fusion*, 55(12):124012, 2013.

[62] S. Ku, J. Abiteboul, P.H. Diamond, G. Dif-Pradalier, J.M. Kwon, Y. Sarazin, T.S. Hahm, X. Garbet, C.S. Chang, G. Latu, E.S. Yoon, Ph. Ghendrih, S. Yi, A. Strugarek, W. Solomon, and V. Grandgirard. Physics of intrinsic rotation in flux-driven ITG turbulence. *Nuclear Fusion*, 52(6):063013, 2012.

[63] X. Garbet, J. Abiteboul, A. Strugarek, Y. Sarazin, G. Dif-Pradalier, Ph. Ghendrih, V. Grandgirard, C. Bourdelle, G. Latu, and A. Smolyakov. Thermodynamics of neoclassical and turbulent transport. *Plasma Physics and Controlled Fusion*, 54(5):055007, 2012.

[64] G. Dif-Pradalier, P. H. Diamond, V. Grandgirard, Y. Sarazin, J. Abiteboul, X. Garbet, Ph. Ghendrih, G. Latu, A. Strugarek, S. Ku, and C. S. Chang. Neoclassical physics in full distribution function gyrokinetics. *Physics of Plasmas*, 18(6):062309, 2011.

[65] J. Abiteboul, X. Garbet, V. Grandgirard, S. J. Allfrey, Ph. Ghendrih, G. Latu, Y. Sarazin, and A. Strugarek. Conservation equations and calculation of mean flows in gyrokinetics. *Physics of Plasmas*, 18(8):082503, 2011.

[66] Y. Sarazin, V. Grandgirard, J. Abiteboul, S. Allfrey, X. Garbet, Ph. Ghendrih, G. Latu, A. Strugarek, G. Dif-Pradalier, P.H. Diamond, S. Ku, C.S. Chang, B.F. McMillan, T.M. Tran, L. Villard, S. Jolliet, A. Bottino, and P. Angelino. Predictions on heat transport and plasma rotation from global gyrokinetic simulations. *Nuclear Fusion*, 51(10):103023, 2011.

[67] L. Villard et al. Gyrokinetic simulations of turbulent transport: size scaling and chaotic behaviour. *Plasma Physics and Controlled Fusion*, 52(12):124038, 2010.

[68] Y. Sarazin, A. Strugarek, G. Dif-Pradalier, J. Abiteboul, S. Allfrey, X. Garbet, Ph. Ghendrih, V. Grandgirard, and G. Latu. Flux-driven gyrokinetic simulations of ion turbulent transport at low magnetic shear. *Journal of Physics: Conference Series*, 260(1):012017, 2010. `http://stacks.iop.org/1742-6596/260/i=1/a=012017`.

[69] Y. Sarazin, V. Grandgirard, J. Abiteboul, S. Allfrey, X. Garbet, Ph. Ghendrih, G. Latu, A. Strugarek, and G. Dif-Pradalier. Large scale dynamics in flux driven gyrokinetic turbulence. *Nuclear Fusion*, 50(5):054004, 2010.

[70] X. Garbet, J. Abiteboul, Y. Sarazin, A. Smolyakov, S. Allfrey, V. Grandgirard, Ph. Ghendrih, G. Latu, and A. Strugarek. Entropy production rate in tokamak plasmas with helical magnetic perturbations. *Journal of Physics: Conference Series*, 260(1):012010, 2010.

[71] V. Grandgirard, Y. Sarazin, P. Angelino, A. Bottino, N. Crouseilles, G. Darmet, G. Dif-Pradalier, X. Garbet, Ph. Ghendrih, S. Jolliet, G. Latu, E. Sonnendrucker, and L. Villard. Global full-f gyrokinetic simulations of plasma turbulence. *Plasma Physics and Controlled Fusion*, 49(12B):B173, 2007.

[72] Y. Sarazin, V. Grandgirad, G. Dif-Praladier, E. Fleurance, X. Garbet, Ph. Ghendrih, P. Bertrand, N. Besse, N. Crouseilles, E. Sonnendrücker, G. Latu, and E. Violard. Impact of large scale flows on turbulent transport. *Plasma Phys. Control Fusion*, 48:B179–B188, december 2006.

## Plasma Physics conference proceedings

[73] Ph. Ghendrih, Th. Cartier-Michaud, G. Dif-Pradalier, D. Esteve, X. Garbet, V. Grandgirard, G. Latu, Cl. Norscini, and Y. Sarazin. Collisions in magnetised plasmas. *ESAIM: Proc.*, 50:81–112, 2015.

[74] Dif-Pradalier et al. Further details on the plasma ExB staircase. In *7th IAEA Technical Meeting on Theory of Plasmas Instabilities, Frascati, Italy*, France, 2015.

[75] Y. Sarazin et al. Understanding momentum transport in tokamak plasmas. In *IAEA Fusion Energy Conference*, Saint Petersburg, october 2014.

[76] X. Garbet et al. Beyond scale separation in gyrokinetic turbulence. In *21st IAEA Fusion Energy Conference*, Chengdu, China, october 2006.

## Submitted papers - reports - documents I contributed to

[77] G. Latu, Y. Asahi, J. Bigot, T. Fehér, and V. Grandgirard. Scaling and optimizing the Gysela code on a cluster of many-core processors. Submitted to Europar 2018 conference, CEA, February 2018. `https://hal.inria.fr/hal-01719208`.

[78] P. Donnel, X. Garbet, Y. Sarazin, V. Grandgirard, Y. Asahi, N. Bouzat, E. Caschera, G. Dif-Pradalier, Ch. Ehrlacher, Ph. Ghendrih, G. Latu, and Ch. Passeron. A multi-species collisional operator for full-F global gyrokinetic codes: Numerical aspects and validation with the GYSELA code. Research Report, CEA, January 2018. `https://hal.archives-ouvertes.fr/hal-01687586`.

[79] M. Thevenin, O. Thomine, and G. Latu. Compression de données numériques, 2017. WO Patent App. PCT/EP2016/081,284, Brevet - `https://www.google.com/patents/WO2017103002A1?cl=fr`.

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

[81] N. Bouzat, C. Bressan, V. Grandgirard, G. Latu, and M. Mehrenberger. Targeting realistic geometry in Tokamak code Gysela. *Submitted to - ESAIM: Proc.*, 2017. `https://hal.archives-ouvertes.fr/hal-01653022`.

[82] J. Bigot, V. Grandgirard, G. Latu, J.-F. Mehaut, L.-F. Millani, Ch. Passeron, S. Quinito Masnada, J. Richard, and B. Videau. Building and auto-tuning a kernel: an experiment with Boast and StarPU in the Gysela code. *Submitted to - ESAIM: Proc.*, 2017.

[83] Fabien Rozar. *Towards highly scalable parallel simulations for turbulent plasma physics.* Theses, Université de Bordeaux, November 2015. `https://tel.archives-ouvertes.fr/tel-01271032`.

[84] Xavier Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems.* Theses, Université de Bordeaux, February 2015. `https://tel.archives-ouvertes.fr/tel-01222565/file/LACOSTE_XAVIER_2015.pdf`.

[85] G. Latu, M. Mehrenberger, M. Ottaviani, and E. Sonnendrücker. Aligned interpolation and application to drift kinetic semi-Lagrangian simulations with oblique magnetic field in cylindrical geometry. Research report, IRMA, December 2014. `https://hal.inria.fr/hal-01098373`.

[86] Matthieu Kuhn. *Parallel computing and numerical methods for boundary plasma simulations.* Theses, Université de Strasbourg, September 2014. `https://tel.archives-ouvertes.fr/tel-01272267/file/Kuhn_Matthieu_2014_ED269.pdf`.

[87] Rached Abdelkhalek. *Hardware acceleration for seismic imaging : modeling, migration and interpretation*. PhD thesis, Université Sciences et Technologies - Bordeaux I, December 2013. https://tel.archives-ouvertes.fr/tel-01159517/file/ABDELKHALEK_RACHED_2013.pdf.

[88] G. Latu, M. Becoulet, G. Dif-Pradalier, V. Grandgirard, M. Hoelzl, G. Huysmans, X. Lacoste, E. Nardon, F. Orain, Ch. Passeron, P. Ramet, and A. Ratnani. Non regression testing for the Jorek code. Research Report RR-8134, Inria, November 2012. https://hal.inria.fr/hal-00752270.

[89] G. Latu, V. Grandgirard, J. Abiteboul, M. Bergot, N. Crouseilles, X. Garbet, Ph. Ghendrih, M. Mehrenberger, Y. Sarazin, H. Sellama, E. Sonnendrücker, and D. Zarzoso. Accuracy of unperturbed motion of particles in a gyrokinetic semi-Lagrangian code. Rapport de recherche RR-8054, Inria, September 2012. http://hal.inria.fr/hal-00727118.

[90] M. Sauget and G. Latu. Dynamic Load Balancing for PIC codes using Eulerian/Lagrangian partitioning. Research report, University of Strasbourg, 2011. http://arxiv.org/abs/1706.08362.

[91] G. Latu, V. Grandgirard, N. Crouseilles, R. Belaouar, and E. Sonnendrücker. Some parallel algorithms for the Quasineutrality solver of Gysela. Research Report RR-7591, Inria, April 2011. https://hal.inria.fr/inria-00583521.

[92] G. Latu, N. Crouseilles, and V. Grandgirard. Parallel bottleneck in the Quasineutrality solver embedded in Gysela. Research Report RR-7595, Inria, April 2011. https://hal.inria.fr/inria-00583689.

[93] G. Latu, V. Grandgirard, N. Crouseilles, and G. Dif-Pradalier. Scalable Quasineutral solver for gyrokinetic simulation. Rapport de recherche RR-7611, Inria, May 2011. http://hal.inria.fr/inria-00590561/PDF/RR-7611.pdf.

[94] J. Guterl, J.-P. Braeunig, N. Crouseilles, V. Grandgirard, G. Latu, M. Mehrenberger, and E. Sonnendrücker. Test of some numerical limiters for the conservative PSM scheme for 4D Drift-Kinetic simulations. Research Report RR-7467, Inria, November 2010. https://hal.inria.fr/inria-00540948.

[95] J.-P. Braeunig, N. Crouseilles, V. Grandgirard, G. Latu, M. Mehrenberger, and E. Sonnendrücker. Some numerical aspects of the conservative PSM scheme in a 4D drift-kinetic code. Research report, Inria, 2011. https://hal.archives-ouvertes.fr/hal-00650343.

[96] Matthieu Haefele. *Simulation adaptative et visualisation haute performance de plasmas et de faisceaux de particules*. PhD thesis, Université de Strasbourg, 2007. http://www.haefele.fr/matthieu/publis/2007_haefele_phd_A4.pdf.

## References of the literature

[97] J.C. Adam, A. Gourdin Serveniere, and A.B. Langdon. Electron sub-cycling in particle simulation of plasma. *Journal of Computational Physics*, 47(2):229 – 244, 1982.

[98] P. Angelino, A. Bottino, R. Hatzky, S. Jolliet, O. Sauter, T. M. Tran, and L. Villard. On the definition of a kinetic equilibrium in global gyrokinetic simulations. *Physics of Plasmas*, 13(5):052304, 2006.

[99] P. Angelino, X. Garbet, et al. Role of Plasma Elongation on Turbulent Transport in Magnetically Confined Plasmas. *Physical Review Letters*, 102(19), 2009.

[100] P. Angelino, X. Garbet, L. Villard, A. Bottino, S. Jolliet, Ph. Ghendrih, V. Grandgirard, B. F. McMillan, Y. Sarazin, G. Dif-Pradalier, and T. M. Tran. The role of plasma elongation on the linear damping of zonal flows. *Physics of Plasmas*, 15(6), 2008.

[101] S. Ashby et al. The opportunities and challenges of exascale computing. Summary report of the Advanced Scientific Computing Advisory Commitee, 2010. https://science.energy.gov/~/MEDIA/ASCR/ASCAC/PDF/REPORTS/EXASCALE_SUBCOMMITTEE_REPORT.PDF.

[102] J. A. Åström, A. Carter, J. Hetherington, K. Ioakimidis, E. Lindahl, G. Mozdzynski, R. W. Nash, P. Schlatter, A. Signell, and J. Westerholm. Preparing scientific application software for exascale computing. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, PARA'12, pages 27–42. Springer-Verlag, Berlin, Heidelberg, 2013.

[103] C. Aulagnon, D. Martin-Guillerez, F. Rué, and F. Trahay. Runtime function instrumentation with EZTRACE. In *Euro-Par 2012: Parallel Processing Workshops*, pages 395–403. Springer, 2013.

[104] R. Aymar, P. Barabaschi, and Y. Shimomura. The ITER design. *Plasma Phys. Control. Fusion*, 44:519–565, 2002.

[105] N. Besse and M. Mehrenberger. Convergence of classes of high-order semi-Lagrangian schemes for the Vlasov-Poisson system. *Mathematics of Computation*, 77(61):93–123, 2008.

[106] A. Biancalani, A. Bottino, C. Ehrlacher, V. Grandgirard, G. Merlo, I. Novikau, Z. Qiu, E. Sonnendruecker, X. Garbet, T. Goerler, S. Leerink, F. Palermo, and D. Zarzoso. Cross-code gyrokinetic verification and benchmark on the linear collisionless dynamics of the geodesic acoustic mode. `https://arxiv.org/abs/1705.06554`, submitted to Phys. Plasma, 2018.

[107] C.K. Birdsall and A. Langdon. *Plasma physics via computer simulation*. McGraw-Hill, New York, NY, USA, 1985.

[108] M. Bostan. High magnetic field equilibria for the fokker-planck-landau equation. *Ann. Inst. H. Poincaré Anal. Non Linéaire*, 33(4):899–931, 2016.

[109] J.-Ph. Braeunig, N. Crouseilles, M. Mehrenberger, and E. Sonnendrücker. Guiding-center simulations on curvilinear meshes. *Discrete and Continuous Dynamical Systems - Series S*, 5(2):271–282, 2012. `http://aimsciences.org/journals/displayArticlesnew.jsp?paperID=6582`.

[110] A. J. Brizard and T. S. Hahm. Foundations of nonlinear gyrokinetic theory. *Rev. Mod. Phys.*, 79(2):421–468, 2007.

[111] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *[CGO 2003]*, pages 265–275. IEEE, 2003.

[112] C. S. Chang, S. Ku, G. R. Tynan, R. Hager, R. M. Churchill, I. Cziegler, M. Greenwald, A. E. Hubbard, and J. W. Hughes. Fast low-to-high confinement mode bifurcation dynamics in a tokamak edge plasma gyrokinetic simulation. *Phys. Rev. Lett.*, 118:175001, Apr 2017.

[113] Ch. 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.

[114] 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*, pages 60–60, 2005.

[115] N. Crouseilles, P. Glanc, S. Hirstoaga, E. Madaule, M. Mehrenberger, and J. Pétri. A new fully two-dimensional conservative semi-lagrangian method: applications on polar grids, from diocotron instability to itg turbulence. *The European Physical Journal D*, 68(9):252, Sep 2014. `http://dx.doi.org/10.1140/epjd/e2014-50180-9`.

[116] N. Crouseilles, M. Mehrenberger, and H. Sellama. Numerical solution of the gyroaverage operator for the finite gyroradius guiding-center model. *Communications in Computational Physics*, 8(3):484–510, September 2010.

[117] N. Crouseilles, M. Mehrenberger, and E. Sonnendrücker. Conservative semi-Lagrangian schemes for Vlasov equations. *Journal of Computational Physics*, 229(6):1927 – 1953, 2010.

[118] N. Crouseilles, Th. Respaud, and E. Sonnendrücker. A forward semi-lagrangian method for the numerical solution of the Vlasov equation. *Computer Physics Communications*, 180(10):1730–1745, 2009.

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

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

[121] G. Dif-Pradalier, P. H. Diamond, V. Grandgirard, Y. Sarazin, J. Abiteboul, X. Garbet, Ph. Ghendrih, A. Strugarek, S. Ku, and C. S. Chang. On the validity of the local diffusive paradigm in turbulent plasma transport. *Phys. Rev. E*, 82:025401, Aug 2010.

[122] G. Dif-Pradalier, V. Grandgirard, Y. Sarazin, X. Garbet, and Ph. Ghendrih. Defining an equilibrium state in global full-f gyrokinetic models. *Communications in Nonlinear Science and Numerical Simulation*, 13(1):65 – 71, 2008. Vlasovia 2006: The Second International Workshop on the Theory and Applications of the Vlasov Equation, http://dx.doi.org/10.1016/j.cnsns.2007.05.004.

[123] G. Dif-Pradalier, V. Grandgirard, Y. Sarazin, X. Garbet, and Ph. Ghendrih. Interplay between gyrokinetic turbulence, flows, and collisions: Perspectives on transport and poloidal rotation. *Phys. Rev. Lett.*, 103:065002, Aug 2009.

[124] G. Dif-Pradalier, V. Grandgirard, Y. Sarazin, X. Garbet, Ph. Ghendrih, and P. Angelino. On the influence of initial state on gyrokinetic simulations. *Physics of Plasmas*, 15(4):042315, 2008.

[125] A. M. Dimits, G. Bateman, M. A. Beer, B. I. Cohen, W. Dorland, G. W. Hammett, C. Kim, J. E. Kinsey, M. Kotschenreuther, A. H. Kritz, L. L. Lao, J. Mandrekas, W. M. Nevins, S. E. Parker, A. J. Redd, D. E. Shumaker, R. Sydora, and J. Weiland. Comparisons and physics basis of tokamak transport models and turbulence simulations. *Physics of Plasmas*, 7(3):969–983, 2000.

[126] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J. Acquaviva, W. Jalby, et al. MAQAO: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[127] F. Filbet and E. Sonnendrücker. Comparison of Eulerian Vlasov solvers. *Computer Physics Communications*, 150(3):247 – 266, 2003. http://dx.doi.org/10.1016/S0010-4655(02)00694-X.

[128] F. Filbet and E. Sonnendrücker. Modeling and numerical simulation of space charge dominated beams in the paraxial approximation. *Mathematical Models and Methods in Applied Sciences*, 16(5):763, 2006.

[129] F. Filbet, E. Sonnendrücker, and P. Bertrand. Conservative Numerical Schemes for the Vlasov Equation. *Journal of Computational Physics*, 172(1):166 – 187, 2001.

[130] R. Fitzpatrick, C.G. Gimblett, and R.J. Hastie. On the '1-1/2-d' evolution of tokamak plasmas in the case of large aspect ratio. *Plasma Physics and Controlled Fusion*, 34(2):161, 1992.

[131] X. Garbet, J. Abiteboul, S. Benkadda, C. Bourdelle, A. Casati, F. Clairet, N. Dubuit, G. Falchetto, C. Fenzi, S. Futatani, V. Grandgirard, R. Guirlet, O. D. Gurcan, P. Hennequin, S. Heuraux, G. T. Hoang, C. Honor, R. Sabot, Y. Sarazin, J. L. Segui, A. Smolyakov, E. Trier, L. Vermare, , and D. Villegas. Measurements and modeling of turbulent transport in the tore supra tokamak. *AIP Conference Proceedings*, 1308(1):75–84, 2010.

[132] X. Garbet, Y. Idomura, L. Villard, and T.H. Watanabe. Gyrokinetic simulations of turbulent transport. *Nuclear Fusion*, 50(4):043002, 2010. http://stacks.iop.org/0029-5515/50/i=4/a=043002.

[133] X. Garbet and R. E. Waltz. Heat flux driven ion turbulence. *Physics of Plasmas*, 5(8):2836–2845, 1998.

[134] M. Geimer and O. Abert. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *Proceedings of WSCG*, pages 71–78, 2005.

[135] M. Geimer, F. Wolf, B.-JN Wylie, E. Ábrahám, D. Becker, and B. Mohr. The SCALASCA performance toolset architecture. *CCPE*, 22(6):702–719, 2010.

[136] Francis X Giraldo and Beny Neta. A comparison of a family of eulerian and semi-lagrangian finite element methods for the advection-diffusion equation. In *In Computer Modelling of Seas and Coastal Regions III, (Edited by J.R. Acinas and C.A. Brebbia*, pages 217–229, 1997. http://faculty.nps.edu/bneta/papers/new_sisl.pdf.

[137] A.H. Glasser, I.A. Kitaeva, V.D. Liseikin, V.S. Lukin, and A.N. Simakov. Harmonic grid generation for the tokamak edge region. In *Proceedings of EPS, Spain*, 2005.

[138] T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told. The global version of the gyrokinetic turbulence code gene. *J. Comput. Physics*, 230(18):7053–7071, 2011.

[139] V. Grandgirard, M. Brunetti, P. Bertrand, N. Besse, X. Garbet, P. Ghendrih, G. Manfredi, Y. Sarazin, O. Sauter, E. Sonnendrücker, 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.

[140] Virginie Grandgirard. *The GYSELA project: A semi-Lagrangian code addressing gyrokinetic full-f global simulations of flux driven tokamak plasmas.* Hdr, Université de Strasbourg, Nov 2016. `https://publication-theses.unistra.fr/public/hdr/2016/2016_Grandgirard_Virginie.pdf`.

[141] 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.

[142] T. S. Hahm. Nonlinear gyrokinetic equations for tokamak microturbulence. *Physics of Fluids*, 31(9):2670–2673, 1988.

[143] A. Hamiaz, M. Mehrenberger, A. Back, and P. Navaro. Guiding center simulations on curvilinear grids. *ESAIM: Proc.*, 53:99–119, 2016. `http://dx.doi.org/10.1051/proc/201653007`.

[144] A. Hamiaz, M. Mehrenberger, H. Sellama, and E. Sonnendrücker. The semi-lagrangian method on curvilinear grids. *Communications in Applied and Industrial Mathematics.*, 7(3):99–137, 2016.

[145] G. W. Hammett and Fr. W. Perkins. Fluid moment models for Landau damping with application to the ion-temperature-gradient instability. *Phys. Rev. Lett.*, 64(25):3019–3022, 1990.

[146] F. Hariri and M. Ottaviani. A flux-coordinate independent field-aligned approach to plasma turbulence simulations. *Computer Physics Communications*, 184(11):2419 – 2429, 2013. `http://dx.doi.org/10.1016/j.cpc.2013.06.005`.

[147] R. Hatzky, Tr. Minh Tran, A. Könies, R. Kleiber, and S. J. Allfrey. Energy conservation in a nonlinear gyrokinetic particle-in-cell code for ion-temperature-gradient-driven modes in theta-pinch geometry. *Physics of Plasmas*, 9(3):898–912, 2002.

[148] M.-A. Heroux, D.-W. Doerfler, P.-S. Crozier, J.-M. Willenbring, H.-C. Edwards, A. Williams, M. Rajan, E.-R. Keiter, H.-K. Thornquist, and R.-W. Numric. Improving performance via mini-applications - SANDIA REPORT SAND2009-5574, September 2009. `http://www.cs.sandia.gov/~maherou/docs/MantevoOverview.pdf`.

[149] B. Holman and L. Kunyansky. A second-order finite difference scheme for the wave equation on a reduced polar grid, 2015.

[150] F. Huot, A. Ghizzo, P. Bertrand, E. Sonnendrücker, and O. Coulaud. Instability of the time splitting scheme for the one-dimensional and relativistic Vlasov–Maxwell system. *Journal of Computational Physics*, 185(2):512–531, March 2003. `http://dx.doi.org/10.1006/jcph.2002.0079`.

[151] Y. Idomura, M. Ida, T. Kano, N. Aiba, and S. Tokuda. Conservative global gyrokinetic toroidal full-f five-dimensional Vlasov simulation. *Computer Physics Communications*, 179(6):391 – 403, 2008.

[152] Y. Idomura, S. Tokuda, and Y. Kishimoto. Global gyrokinetic simulation of ion temperature gradient driven turbulence in plasmas with canonical maxwellian distribution. *Nuclear Fusion*, 43:234–243, 2003.

[153] INTEL. Intel xeon phi coprocessor inst. set archi. ref. manual. `https://software.intel.com/en-us/mic-developer`.

[154] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual, September 2015.

[155] 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.

[156] S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, T.M. Tran, B.F. Mcmillan, O. Sauter, K. Appert, Y. Idomura, and L. Villard. A global collisionless PIC code in magnetic coordinates. *Computer Physics Communications*, 177(5):409 – 425, 2007.

[157] A. Klöckner, T. Warburton, and J. S. Hesthaven. *High-Order Discontinuous Galerkin Methods by GPU Metaprogramming*, pages 353–374. Springer Berlin Heidelberg, 2013. https://doi.org/10.1007/978-3-642-16405-7_23.

[158] J.-M. Kwon, D. Yi, X. Piao, and P. Kim. Development of semi-lagrangian gyrokinetic code for full-f turbulence simulation in general tokamak geometry. *J. Comput. Phys.*, 283(C):518–540, February 2015. http://dx.doi.org/10.1016/j.jcp.2014.12.017.

[159] M.-Ch. Lai. A note on finite difference discretizations for poisson equation on a disk. *Numerical Methods for Partial Differential Equations*, 17(3):199–203, 2001. http://dx.doi.org/10.1002/num.1.

[160] M.-Ch. Lai and W.-Ch. Wang. Fast direct solvers for poisson equation on 2d polar and spherical geometries. *Numerical Methods for Partial Differential Equations*, 18(1):56–68, 2002. http://dx.doi.org/10.1002/num.1038.

[161] 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.

[162] W. W. Lee. Gyrokinetic approach in particle simulation. *Physics of Fluids*, 26(2):556–562, 1983.

[163] Z. Lin and W. W. Lee. Method for solving the gyrokinetic Poisson equation in general geometry. *Phys. Rev. E*, 52(5):5646–5652, Nov 1995.

[164] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K̃. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[165] K. Madduri, E.-J. Im, K. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Computing*, 37(9):501–520, 2011.

[166] B. F. McMillan, S. Jolliet, T. M. Tran, L. Villard, A. Bottino, and P. Angelino. Long global gyrokinetic simulations: Source terms and particle noise control. *Physics of Plasmas*, 15(5):052308, 2008.

[167] M. Mehrenberger, C. Steiner, L. Marradi, N. Crouseilles, E. Sonnendrücker, and B. Afeyan. Vlasov on GPU (VOG project). *ESAIM Proceedings*, 43:37–58, 2013.

[168] A. Mock. Subgridding scheme for FDTD in cylindrical coordinates. In *Progress In Electromagnetics Research Symposium Proceeding*, 2011.

[169] K. Mohseni and T. Colonius. Numerical treatment of polar coordinate singularities. *Journal of Computational Physics*, 157(2):787 – 795, 2000.

[170] N. Nethercote and J. Seward. VALGRIND: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[171] Y. Nishimura, Z. Lin, J. Lewandowski, and S. Ethier. A finite element Poisson solver for gyrokinetic particle simulations in a global field aligned mesh. *J. Comput. Phys.*, 214:657–671, 2006.

[172] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *SIGGRAPH Comput. Graph.*, 24(4):337–345, 1990.

[173] M. Ottaviani and G. Manfredi. The gyro-radius scaling of ion thermal transport from global numerical simulations of ion temperature gradient driven turbulence. *Physics of Plasmas*, 6(8):3267–3275, 1999.

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

[175] P. Peterson. F2PY: a tool for connecting FORTRAN and PYTHON programs. *International Journal of Computational Science and Engineering*, 4(4):296–305, 2009.

[176] S. Del Pino, B. Després, P. Havé, H. Jourdren, and P.F. Piserchia. 3D finite volume simulation of acoustic waves in the earth atmosphere. *Computers and Fluids*, 38(4):765 – 777, 2009.

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

[178] H. Qin. A short introduction to general gyrokinetic theory. Technical Report 4052, PPPL Report, 2005.

[179] Y. Sarazin and Ph. Ghendrih. Intermittent particle transport in two-dimensional edge turbulence. *Physics of Plasmas*, 5(12):4214–4228, 1998.

[180] V. Sarkar et al. Future high performance computing capabilities - summary report of ASCAC, December 2017. `https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/201712/ASCAC-Future-HPC-report.pdf`.

[181] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science – VECPAR 2010*, LNCS 6449, pages 1–25. Springer, 2011.

[182] 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.

[183] A. Staniforth and J. Côté. Semi-lagrangian integration schemes for atmospheric models: A review. *Monthly Weather Review*, 119(9):2206–2223, 1991.

[184] St. 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*, 9(5), 2008.

[185] A. Storelli et al. Comprehensive comparisons of GAM characteristics and dynamics between tore supra experiments and gyrokinetic simulations. *Phys. Plasmas*, 22(6), 2015.

[186] G. Strang. On the Construction and Comparison of Difference Schemes. *SIAM Journal on Numerical Analysis*, 5:506–517, September 1968.

[187] 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.

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

[189] Top500. Top 500 supercomputer sites. http://www.top500.org/.

[190] Thibaut Vernay. *Collisions in Global Gyrokinetic Simulations of Tokamak Plasmas using the Delta-f Particle-In-Cell Approach: Neoclassical Physics and Turbulent Transport.* PhD thesis, EPFL, Suisse, 2013.

[191] Br. Videau, K. Pouget, L. Genovese, Th. Deutsch, D. Komatitsch, Fr. Desprez, and J-Fr Méhaut. BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. *IJHPCA*, 32(1):28–44, 2018.

[192] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[193] Y. Xu, L. Ye, Z. Dai, X. Xiao, and Sh. Wang. Nonlinear gyrokinetic simulation of ion temperature gradient turbulence based on a numerical lie-transform perturbation method. *Physics of Plasmas*, 24(8):082515, 2017. `http://dx.doi.org/10.1063/1.4986395`.

[194] D. Zarzoso, X. Garbet, Y. Sarazin, R. Dumont, and V. Grandgirard. Fully kinetic description of the linear excitation and nonlinear saturation of fast-ion-driven geodesic acoustic mode instability. *Physics of Plasmas*, 19(2):022102, 2012.