Checkpointing Optimisation to Prepare Future Exascale Plasma Turbulence Simulations

Méline Trochon, Julien Bigot, Virginie Grandgirard, Dorian Midou

Checkpointing Optimisation to Prepare Future Exascale Plasma Turbulence Simulations

Méline Trochon
Inria
Bordeaux, France
meline.trochon@inria.fr

Julien Bigot
CEA/MDLS
Gif-sur-Yvette, France
julien.bigot@cea.fr

Virginie Grandgirard
CEA/IRFM
Cadarache, France
virginie.grandgirard@cea.fr

Dorian Midou
CEA/IRFM
Cadarache, France
dorian.midou@cea.fr

***Abstract*—The advent of exascale computing has revolutionized high-performance computing (HPC) and enabled unprecedented advancements in nuclear fusion research. Simulations of plasma turbulence dynamics, such as the GYSELA code, now achieve unparalleled precision and complexity. However, this progress is accompanied by significant challenges in managing the exponential growth of data generated by these simulations. Traditional input/output (I/O) methods struggle to handle the massive data volumes, heightened concurrency, and fault-tolerance requirements inherent to exascale systems.**

This paper investigates the I/O bottlenecks inherent in exascale computing, with a particular focus on the checkpointing mechanisms of GYSELA. These mechanisms are critical for ensuring fault tolerance and must handle several terabytes of data efficiently to avoid undermining computational performance. We analyze the current implementation of GYSELA's checkpointing mechanism managed via the PDI data interface, identifying its limitations and proposing two alternative approaches aimed at enhancing scalability and resilience. Experiments conducted on pre-exascale architectures validate the efficiency of these methods through both strong and weak scaling benchmarks. We reduced the checkpointing execution time by a factor of four, achieving near-optimal bandwidth utilisation, and we have identified implementations well-suited for exascale architectures. Our findings suggest the potential for notable performance improvements and offer insights that could help optimise I/O operations in exascale simulations.

***Index Terms*—Checkpointing, I/O optimisation, Simulation code, Access Pattern optimisation**

I. Introduction

Controlled fusion offers the promise of sustainable and safe energy production on Earth, free of greenhouse gas emissions. In magnetic fusion devices, the power gain increases nonlinearly with the energy confinement time. The quality of the plasma energy confinement thus largely determines the size and therefore the cost of a fusion reactor. Unfortunately, small-scale turbulence limits the quality of confinement in most fusion devices. Hence modelling of turbulent transport is mandatory to find routes towards improved confinement regimes. Numerical simulations are based on a kinetic description of the plasma that can only be performed on most powerful HPC. The gyrokinetic GYSELA [1] code is one of these computational tools. It runs efficiently on several hundred thousand cores on current standard architectures. With a consumption of 150 million CPU hours per year, the code makes massive use of petascale computing capacities and manipulates petabytes of data.

The field of high-performance computing (HPC) has recently transitioned into the exascale era, characterized by systems capable of performing over 10^{18} floating-point operations per second (FLOPS). This extraordinary computational capacity paves the way for simulations with a complete description of electron dynamics that has been unattainable until now. However, there are still many challenges to overcome particularly in the realm of data management and input/output (I/O) operations.

In exascale systems, the volume of data generated by simulations will increase exponentially, driven by the need for higher resolution and fidelity to accurately capture intricate physical processes. Simulations will frequently involve millions of computational cores operating concurrently, producing vast quantities of intermediate and final data. Efficiently managing this data has emerged as a primary challenge, as traditional methods for data access, storage, and transfer struggle to meet the demands of such large-scale systems.

The challenges are not limited to data volume alone. The extreme concurrency of exascale systems, where thousands of processes require simultaneous access to storage resources, has significantly increased the complexity of I/O operations. Moreover, the need for fault tolerance in these massive systems has underscored the importance of robust checkpointing mechanisms. These mechanisms, which periodically save the state of a simulation to enable recovery in the event of a failure, now contend with terabytes of data. Achieving efficient checkpointing without introducing excessive delays is essential to maintaining the computational efficiency of exascale systems. To mitigate the overhead induced by these large I/O operations, GYSELA employs a data interface called PDI. This interface separates the technicalities of I/O operations from the source code and enables GYSELA to utilise specialised I/O libraries and adapt its I/O access patterns without requiring recompilation or modifications to the source code.

This paper investigates the I/O challenges of exascale computing with a focus on GYSELA and its checkpointing mechanisms. We begin by analysing the current implementation and its limitations in handling the increasing data volumes and concurrency demands. Next, we propose two alternative approaches designed to optimise checkpointing performance

by improving scalability and resilience. Finally, we present a detailed analysis of the results, offering broader strategies and best practices to enhance I/O performance not only for GYSELA but also for other large-scale simulation codes. Our findings aim to contribute to more efficient utilisation of exascale computing systems.

II. Background and related work

A. Related Work

With the evolution of large-scale simulations and AI-based applications towards data-centric workloads, efficient storage access has become even more decisive. Numerous studies, summarised in [2], have proposed strategies for optimising I/O operations, including data reading, result writing and checkpointing phases. To make I/O optimisations more accessible across diverse applications, Nicolae et al. [3] introduced the VeloC library, which supports adaptive asynchronous checkpointing. Other studies have focused on optimising specialised I/O libraries, such as ADIOS [4] and HDF5 [5].

In parallel, application-level optimisations have been developed to improve generic checkpointing performance. For example, Gossman et al. [6] enhanced checkpointing by grouping MPI processes, performing intra-group MPI communication, and then executing sequential I/O operations. Wang et al. [7] proposed a method to mitigate I/O contention during checkpointing by introducing random wait times, thereby reducing simultaneous write operations. These optimisations are complementary to our research, as this paper focuses specifically on access patterns to further enhance I/O performance.

Regarding GYSELA, several research activities have already addressed the optimisation of I/O operation. For example, Thomine et al. [8] computes the best checkpoint frequency regarding the mean time before failure (MBTF) and checkpoint's duration, which allow the user to easily obtain the best number of iterations between two checkpoints. Gueroudji et al. [9] implements in-situ asynchronous I/O operations by allocating I/O nodes which communicates with PDI and the DASK [10] library.

Several studies have explored application-level optimisations for specific simulation codes, focusing on enhancing I/O performance and scalability. For instance, Ross et al.[11] discuss the achievements and challenges associated with optimising I/O operations in large-scale simulation codes. Their work also emphasises the importance of accessibility in I/O solutions, aiming to make systems more user-friendly and easier to integrate into simulation workflows. Significant improvements for High-Energy Physics datasets were achieved in[12]. Similarly, Latham et al.[13] investigated targeted optimisations for the FLASH simulation code, demonstrating substantial performance enhancements. Van Gemmeren et al.[14] examined I/O strategies tailored to the ATLAS simulation code, highlighting the impact of domain-specific approaches on efficiency. Moreover, Sen et al. [15] explored strategies for fast I/O throughput in large-scale climate modelling applications, further contributing to the broader understanding of optimising I/O in complex simulations. However, as with this paper, these optimisations rely on characteristics unique to their respective simulation codes and are not directly applicable to GYSELA.

B. Background

1) GYSELA simulation code: The 1st principles 5D code GYSELA [1] (for GYrokinetic SEmi-LAgrangian) is developed at the IRFM/CEA since 20 years to model plasma turbulence and transport in controlled fusion devices. In terms of numerics, the code evolves the entire 5-dimensional (3 space coordinates $\boldsymbol{x}_G = (r, \theta, \phi)$, 2 velocity coordinates $(\upsilon_{||}, \mu)$) guiding-centre distribution functions in a full portion of torus. $\upsilon_{||}$ represents the velocity parallel to the magnetic field. The magnetic momentum μ (proportional to the perpendicular velocity) is an adiabatic motion invariant. The time-evolution of the distribution function of each species $f_s(r, \theta, \phi\upsilon_{||}, \mu)$ is governed by a 5D non-linear gyrokinetic Vlasov equation self-consistently coupled to a 3D quasi-neutrality equation (electrostatic and long wavelength limit of Maxwell's equations) and a 3D Ampère equation. One peculiarity of the GYSELA code is to be based on a backward semi-Lagrangian (SL) method, which is a mix between Particle-In-Cell (PIC) and Eulerian approaches. In this approach, the phase-space mesh grid is kept fixed in time (Eulerian method) and the Vlasov equation is integrated along its trajectories (Lagrangian method) using the invariance of the distribution function. From a numerical point of view, the IRFM team has demonstrated very good conservation properties [16]. From a parallelisation point of view, its Eulerian character is an advantage because there is no problem of load-balancing. On the other hand, the need to use a cubic spline interpolation which is non-local is clearly a disadvantage leading to a code much more difficult to parallelise than a PIC code. The code consists of approximately 60,000 lines, primarily written in Fortran, and is parallelised using a hybrid OpenMP and MPI approach. t has benefitted from continuous improvement of its computational scalability [17], [18] leading to efficient runs on several hundred thousand cores on standard CPU.

The GYSELA code produces very large amounts of data. A typical 5D mesh contains several hundreds of billions of points, which leads to 5D distribution functions of the order of 2 TB to be followed at each time iteration. Knowing that a simulation can run on several days or weeks (representing several tens of chained runs of more than 10,000 iterations), it is not conceivable to store the temporal evolution of these distribution functions f_s. Only time evolutions of 0 to 3D physical quantities (cross-section, fluid moments, ...) are saved all along the simulation while f_s are only saved at the end of each run in temporary files to allow the restart (checkpoint-restart). Even if the optimisation of diagnostics on exascale computer is still an open question, we will focus here on the checkpointing mechanism. As discussed in the following, this huge amount of data storage is already a bottleneck for petascale simulations. and as demonstrated by Thomine et al. [8], the importance of checkpointing mechanisms increases with the probability of execution failure, which grows as the number of cores allocated to a job rises. As the number

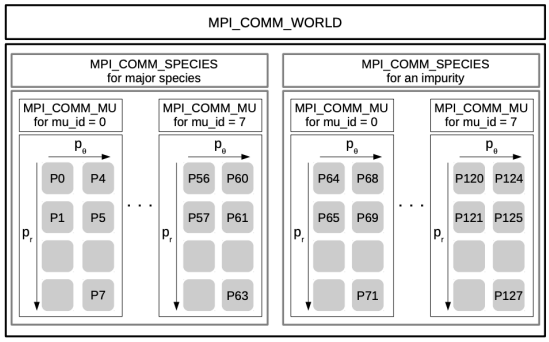

Fig. 1. MPI COMM WORLD communicator decomposition for two species. In this case, the number of MPI processes is equal to 128 from the paper [1]

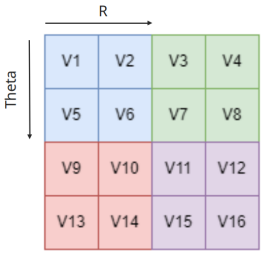

Fig. 2. Scheme of a 4D-matrix decomposition with 4 MPI processes Each color represents an MPI process, for example the blue MPI process stores the values (V1, V2, V5, V6), corresponding to the memory P0 in Fig. 1. In this array, the data assigned to a single MPI process is not contiguous, indicating that the local contiguity differs from the overall matrix contiguity.

of cores increases on exascale architectures, the MBTF is expected to decrease, highlighting the critical need for efficient and reliable checkpointing strategies to ensure fault tolerance.

2) The checkpointing mechanism: The GYSELA code runs efficiently on more than 500k CPU cores. It exhibits a relative efficiency (as defined in subsection IV-A) of 85% on more than 500k cores and 63% on $729\,088$ cores for a weak scaling from 1024 to $5\,696$ AMD EPYC nodes (test performed on CEA-HF supercomputer at Bruyère-le-Chatel/France) without I/O operations. However, the relative efficiency of I/O is 55% on $3\,072$ nodes deteriorates very strongly when the number of nodes is doubled and a crash occurred during the writing on $5\,696$ nodes. These results must be taken as preliminary because the tests were performed when the file system was not completely tuned. However, this maximum saving remains an achievement in itself with 13.2 Tbytes of data distributed in $24\,576$ files that were simultaneously written to disk. The checkpointing process took 1173 seconds, which is approximately one quarter of the time required for a single iteration. This remains a significant bottleneck that needs to be removed in order to achieve the performance necessary for exascale simulations.

Concerning the MPI parallelisation, an MPI communicator `'MPI_COMM_SPECIES'` is defined per species. Inside each one of the `'MPI_COMM_SPECIES'` communicators, an MPI communicator is defined for each value of the magnetic moment μ. Within each `'MPI_COMM_MU'` communicator, a 2D decomposition is performed such as each MPI process contains a sub-domain in (r, θ) dimensions (see Figure 1). Thus, a MPI process k is responsible for the subdomain defined by $Pk = f(r = [i_{start},\ i_{end}],\ \theta = [j_{start},\ j_{end}],\ \phi = *,\ v_{||} = *,\ \mu = \mu_{value})$, with $\mu_{value} \in [0, N_\mu - 1]$. The local values $i_{start},\ i_{end},\ j_{start},\ j_{end}$ corresponds to a classical sub-domain decomposition of (p_r, p_θ) blocks, with p_r (resp. p_θ) the number of sub-domains in r (resp. θ) direction. The local size of the matrix uses, in practice, one fourth of the total MPI process size, which is of the order of the GB.

As to improve this checkpointing mechanism, GYSELA first used the open source I/O library HDF5 [19], especially parallel HDF5 to manage the quantity of files created during I/O operations. Two different implementations were proposed. The first one, usually named one file per process (1FPP), uses HDF5 without collective features (meaning that all the process perform I/O independently). The second version uses Parallel HDF5 (PHDF5), which uses the MPI-IO library to optimise and write collective I/O operations. However, these implementations increased code complexity without delivering satisfactory performance. In fact, the implementation using PHDF5 took significantly more time to complete than the first implementation, making it unsuitable for practical use. As a consequence, the GYSELA developers decided to transfer the I/O management to a data interface named PDI [20], allowing the application code to be more focused on physics or mathematics, and making it more readable for the physics researchers. Also, the use of PDI allows modifications of the I/O strategies without recompiling the source code.

3) Specialised I/O libraries: The PDI [1] Data Interface [20] is a lightweight library designed for efficient data management. It provides a declarative API that exposes simulation data without requiring users to specify how to manipulate it. Additionally, it allows integration of external libraries via a configuration file called specification tree.

The idea behind of PDI is to simplify the source code of the application by keeping I/O operations simple to write and to manage. Also, through the utilisation of a specification tree, it permits the usage of several I/O libraries and dynamical change of I/O parameters. In our case, the "User application" is GYSELA, the "Events subsystems" are the checkpointing implementations, the "I/O library" is HDF5, with the plugin Decl'HDF5 and the "Specification tree" is a YAML file containing the description of the implementations.

The main functions used by GYSELA to interact with PDI, beyond the initialisation phase, are `PDI_share` that shares data with PDI by creating a lock of the memory space and triggering an I/O operation and `PDI_reclaim` used to de-lock the memory space. The function `PDI_expose` is equivalent to `PDI_share` followed by `PDI_reclaim`. The function `PDI_event` creates an event that handles the control transfer. This function is used to implement different types of I/O operations (independent, collective, etc..), enabling the

[1]Official documentation: https://pdi.dev/new-site/docs/

easy comparison of implementations by changing the event name in GYSELA input file. This also allow to have adaptive I/O operations throughout the job.

The specification tree defines the data structure, coordinates interactions between the code and PDI, and includes the configuration of plugins. This specification is provided to PDI via a YAML file. The specification tree is divided into three main sections: metadata, data, and plugins. The metadata section sets various values for which PDI keeps a copy. For instance the domain size and the domain decomposition parameters are considered as metadata. The data section describes the data layout, specifying the types of data expected during I/O operations. The plugins section lists the plugins to be loaded along with their configurations. PDI facilitates the loose coupling of simulation codes with external libraries by supporting libraries as plugins, which are configured through the specification tree. For instance, the Decl'HDF5 plugin for HDF5 can be configured to tailor the I/O operations as needed with variables as MPI communicators, attributes, or specific HDF5 features such as chunking, deflate compression, and more[2].

III. Optimising GYSELA's I/O performance

As said in the previous section, when performing I/O operations such as checkpoints, GYSELA exposes the data via an API and shares information of the I/O operation to PDI: its size, its quantity, its location, etc... Using these information and some parameters set in the specification tree, PDI can perform the operation, either using MPI functions, or other I/O libraries like HDF5 via plug-ins. Thus, the lines of code in the simulation code are solely informative, providing details about I/O events and the initialisation of the I/O operations inside the specification tree becomes the critical aspect to tune the checkpointing mechanism. GYSELA offers four different implementations of the checkpointing mechanism. In this research, we focused on fine-tuning the writing pattern of the checkpointing process, optimising its efficiency, while retaining the default collective writing operation performed by HDF5 without modifying its underlying parameters. :

- individual, synchronous I/O with one file per process (SYNC)
- collective I/O with small, non-contiguous writes, shared file access (PARA)
- collective I/O with one large contiguous write, shared file access (PACNT)
- collective I/O with smaller contiguous writes, shared file access (PARA_R)

Originally, two checkpoint's method were implemented using PDI which reproduced the method written in the source code. The first one uses the HDF5 library without MPI-IO and is fully independent, it does synchronous writing with one file created per process (SYNC). This method is very fast at small scale but generates contention on larger systems (too many concurrent streams, high load on the metadata server). The second implementation (PARA), which uses the HDF5 library with MPI-IO enabled, is fully parallelised and creates one output file per 4D-matrix. All the MPI processes in the same communicator `MPI_COMM_MU` write collectively in the same output file. The number of files written is lower and this technique is known to scale well on a large number of nodes. However, this approach is still far from the achievable peak performance.

The poor I/O bandwidth observed is primarily due to the configuration of the I/O operations. Specifically, to generate easily readable output files, the 4D matrices are fully reconstructed during the writing process. This approach results in MPI processes writing numerous small chunks of data because the matrices are not stored contiguously. On closer examination, each I/O operation writes data on the order of kilobytes, but the total data size reaches several gigabytes. Consequently, this process involves millions of individual write operations. While HDF5 attempts to optimise these operations through collective writes, it is unable to implement an efficient solution for handling such a high volume of small, fragmented writes.

To improve performance, the initial approach was to write contiguously rather than in small chunks of data. One efficient way to achieve this is by partitioning the data differently when writing to the output file. Similar to subfiling in HDF5, the idea is to introduce an additional outer dimension that stores each MPI process's data contiguously. In this scheme, the first dimension is incremented for each MPI process, following a linear partition. With this parallelised and contiguous implementation (PACNT), each MPI process performs a single large write operation, as opposed to the millions of small writes in the previous implementation (PARA). As a result, we expect PACNT to outperform PARA. However, the duration of this implementation compared to the independent approach execution time is unclear. Indeed, this implementation writes fewer files, especially on cases with huge parallelisation. The overhead of collective writing may slow it down too much, making it potentially longer than the independent implementation.

For this reason, we implemented another method (PARA_R), which is similar to PACNT, but instead of creating one file per μ matrix, we create one file per μ and θ values. Thus, this feature only parallelises in the r direction using the communicator `MPI_COMM_R`, creating `mu x Nbproc_theta` files, which is of the order of 1 000 files. That case, the number of files created is bigger than in the PACNT implementation, resulting in smaller collective writing to the same output file.

To conclude, since we do not perform our experiments at full scale, we expect the first independent implementation to achieve near optimal bandwidth, but we already know that it will not be a sustainable solution for future architectures. However, we have good expectations that the new implementations, which we believe are scalable on future architectures, will achieve performance comparable to the independent implementation. Additionally, these new implementations generate

[2]More details about Decl'HDF5 can be found in the documentation https://pdi.dev/main/Decl_HDF5_plugin.html

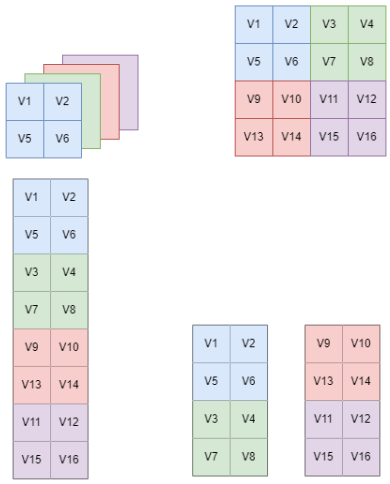

Fig. 3. Representation of each checkpointing implementation using the scheme in Fig. 2. The top left represents the SYNC implementation, the top right the PARA implementation, the bottom left the PACNT implementation and the bottom right the PAR_R implementation.

$Nbproc$ ($\approx 10k$) is the total number of MPI processes, N_μ (≈ 60) is the number of μ communicators and $Nbproc_\theta$ (≈ 10) is the number of processes regarding the dimension θ in each MPI process communicator. N_{local} (several MB) is the size of the multi-dimensional array store locally for each MPI processes and N_{*_loc} (≈ 100) is the local size regarding a dimension, except N_{r_loc} ($\leq$ 1KB) which is the size of the inner dimension's contiguous array. Finally, `number of blocks` represents the number of operations by MPI process, with Eq. representing $N_{v_{G\parallel}\_loc} \times N_{\phi\_loc} \times N_{\theta\_loc}$ ($\approx$ 1000k).

Implementation	file number	block length	number of blocks
SYNC	$Nbproc$	N_{local}	1
PARA	N_μ	N_{r_loc}	Eq.
PACNT	N_μ	N_{local}	1
PAR_R	$N_\mu \times Nbproc_\theta$	N_{local}	1

TABLE I
SUMMARY OF ALL CHECKPOINTING IMPLEMENTATIONS IN GYSELA.

fewer output files, improving data manageability and is an important consideration for non-computer science researchers using GYSELA in their work. Fig. 3 and Table I summarise all the chekcpointing implementations of GYSELA.

IV. Evaluation and Results

In this section, we investigate the four checkpointing methods implemented via PDI, utilising the French pre-exascale system, Joliot-Curie. To evaluate their performance, we conducted two types of benchmarks: weak scaling and strong scaling. We then compared the results across the different methods, analysed the outcomes, and derived conclusions regarding checkpointing mechanism for GYSELA in future exascale HPC systems.

A. Experimental methodology

To evaluate the performances of the different PDI implementations, weak scaling and strong scaling were conducted. Each test was repeated multiple times, and the results are presented as the mean and standard deviation corresponding to 90% of the data, providing a robust evaluation of the performance variability.

A key aspect of **Weak Scaling** is maintaining a constant memory footprint per process. This translates to a linear increase in problem size and compute resources. In our tests, we modified the size of the dimension μ, which affects the number of μ-communicators. Therefore, the number of processes writing to the same file remains constant across all test cases and the primary difference observed during this weak benchmark is the overall amount of data being stored simultaneously and the number of output files generated. We conducted two tests: one benchmark with 4 threads per MPI process to artificially increase the number of MPI processes while using fewer nodes, and another benchmark with 16 threads per MPI process to closely resemble production cases.

In **Strong Scaling** tests, the overall size of the problem is fixed while the compute resources increase. In our tests, we increased the number of MPI processes along (r, θ). Hence, the number of MPI processes increases inside μ communicators, creating a heavier parallelisation or more output files depending of the checkpointing method.

In our experiments, we evaluated the benchmarks using three key metrics: the total duration of checkpointing, the mean bandwidth (calculated as the total memory size of I/O operations divided by the duration), and relative efficiency. The relative efficiency is defined as the ratio of the mean bandwidth per MPI process for a given case to the mean bandwidth per MPI process of the smallest case. By construction, the relative efficiency will always be 100% for the smallest case.

The benchmarks were conducted on Joliot Curie, particularly the Rome partition. This partition consists of 2,286 dual-processor AMD Rome (Epyc) compute nodes at 2.6 GHz with 16 cores per socket and 8 sockets per node, for a total of 292,608 computing cores and a power of 11.75 Pflops. Each nodes has 228Go memory. The interconnection follows a *DragonFly* topology. The compute nodes are grouped into racks, each containing 96 nodes and two L1-switches, with 100 Gbits/s InfiniBand technology for the interconnection (HDR100). Six racks form an islet, each islet contains six L2-switches, all of them are connected to the L1-switches. And each L2-switch is connected to an L2-switch of the others islets. In total, there is four complete islets and the last one is shared with another partition. Lastly, the parallel file system uses Lustre [21], with a maximum bandwidth of 300GB/s.[3] During the tests, we use the defaults stripe parameters, with a stripe count of 1, a stripe size of 1MB and no offset.

B. Weak Scaling with 4 threads per MPI process

For this benchmark, only the number of μ communicators is increased. Thus, the mesh grid stays constant in each compute nodes, but it induces a constant increase in the total memory footprint, as shown in Table II. The number of threads allocated per MPI process is 4, and there is 32 MPI processes per node.

[3]More details about the architecture can be found in the documentation: https://www-hpc.cea.fr/tgcc-public/en/html/tgcc-public.html

Each lines represents a test case. The columns, from left to right, states the number of MPI processes, the number of allocated nodes, μ dimension points, the number of MPI processes allocated in each μ communicator along (r, θ), the size of the distribution f and the local memory footprint of f in an MPI process. The mesh grid used regarding $(N_r \times N_\theta \times N_\varphi \times N_{v_{G\parallel}} \times N_\mu)$ is 512x1024x128x128. The number of threads per MPI process is 4.

MPI proc.	nodes	N_μ	p_r, p_θ	total mem.	local size
1024	32	8	8×16	0.5 TB	500 MB
2048	64	16	8×16	1 TB	500 MB
8192	256	64	8×16	4 TB	500 MB
12288	384	96	8×16	6 TB	500 MB

TABLE II
OVERVIEW OF THE WEAK BENCHMARK CONFIGURATIONS.

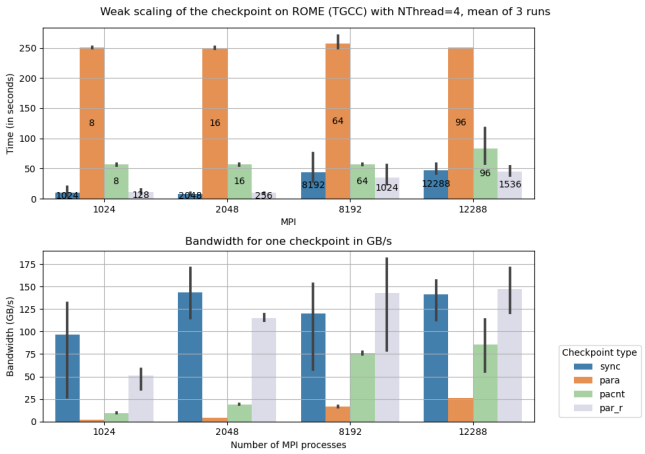

Fig. 4. Weak scaling plot with 4 threads per MPI process, averaged over three runs. The y-axis indicates the number of MPI processes, the x-axis shows the duration of each checkpoint in the top plot and bandwidth in the bottom plot. Each bar represents an implementation, from left to right, SYNC (in blue), PARA (in orange), PACNT (in green) and PAR_R (in grey). The number of output files created for each simulation is displayed on each bar.

The weak scaling shown in 4 presents optimisitic results. As expected, for the smallest case (1024 MPI processes), the independent implementation is achieving the best performance and we observe a fluctuating but high bandwidth around 125 Gb/s. But as the size of the problem gets bigger, time and variance start to increase, and as we have seen in Section II, this implementation is bound to fail for larger scale.

As for the first collective implementation (PARA), we can see that this implementation takes longer than the others. At this moment, it cannot be taken as a sustainable option as its bandwidth does not exceed 25 Gb/s, even if it shows a good scalability.

However, contiguous collective implementation (PACNT) has better results. It takes between 4 to 5 times less duration than the PARA implementation, and in the efficiency plot in Fig. 5, it shows a good efficiency (75% for the maximum number of MPI processes). Also, its bandwidth keeps increasing as the number of MPI processes grows, reaching almost 100 Gb/s for the biggest case. Since the last result can be considered as a pre-exascale I/O operations, we see that the runtime is still longer than the SYNC implementation. But, for the case using

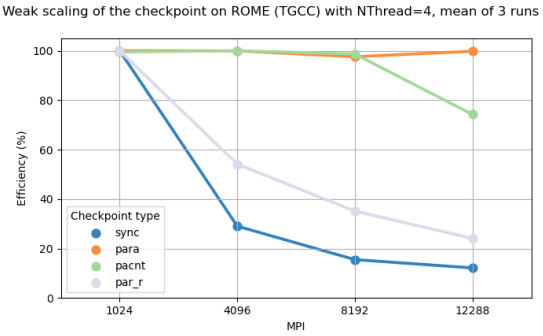

Fig. 5. The relative efficiency of the weak scaling with 4 threads per MPI process. The relative efficiency equation is defined in the previous subsection IV-A.

12288 MPI processes, the SYNC method writes in 12288 files whereas the PACNT method only writes in 96 files.

Lastly, PARA_R implementation also shows good results. Compared to the SYNC implementation, PARA_R has a longer duration in smaller cases (1024 and 2048 MPI processes) but requires less time in the larger cases (8192 and 12288 MPI processes), with a bandwidth near 150 Gb/s. For the 12288 MPI process case, PARA_R method writes in 1536 files which positions it as an in-between alternative compared to SYNC and PACNT methods. However, its efficiency decreases faster compared to the PACNT feature. Indeed, since it creates more and more files, this implementation is more subject to file system congestion.

To conclude, the results are satisfying. Indeed, the PACNT implementation has proven to be a viable method with a small number of files, and PARA_R performs better than the sequential implementation on large enough cases.

This benchmark highlights that a collective implementation (PARA_R) can achieve execution times comparable to SYNC implementation, and that implementations minimising the number of files opened concurrently maintain higher efficiency.

C. Weak Scaling with 16 threads per MPI process

Each lines represents a test case. The columns, from left to right, states the number of MPI processes, the number of allocated nodes, μ dimension points, the number of MPI processes allocated in each μ communicator along (r, θ), the size of the distribution f and the local memory footprint of f in an MPI process. The mesh grid used regarding $(N_r \times N_\theta \times N_\varphi \times N_{v_{G\parallel}} \times N_\mu)$ is 512x1024x128x128. The number of threads per MPI process is 16.

MPI proc.	nodes	N_μ	p_r, p_θ	total mem.	local size
1024	128	8	8×16	0.5 TB	500 MB
2048	256	16	8×16	1 TB	500 MB
4096	512	32	8×16	2 TB	500 MB
8192	1024	64	8×16	4 TB	500 MB

TABLE III
OVERVIEW OF THE WEAK BENCHMARK CONFIGURATIONS.

Similarly to the first weak scaling benchmark, this test also increases the number of μ communicators. However, unlike

the previous benchmark, each MPI process is allocated 16 threads, resulting in 8 MPI processes per node.

We tested only the independent implementation and the two collective, contiguous implementations (SYNC, PACNT, and PAR_R) because this benchmark uses a large number of nodes, and the PARA implementation performed poorly in the first weak scaling benchmark.

As shown in Fig. 6, the results are similar to those of the previous weak scaling benchmark, especially for the collective methods (PACNT and PAR_R). The SYNC implementation, as in the previous benchmark, is very fast for smaller cases. However, for larger cases, such as the one involving 4096 MPI processes, the bandwidth variation is significant. Specifically, the execution time in this case varies from 1.15 second for the fastest to 35.12 seconds for the slowest, resulting in a variation factor of approximately 30. While I/O operations inherently exhibit some variability, this level of fluctuation makes the implementation highly unstable. The results suggest that SYNC's performance depends heavily on external factors such as file system load, which are beyond our control, making it unreliable for larger cases.

Conversely, the PACNT implementation maintains a consistent execution time with minimal variation across all test cases, similar to the previous weak scaling benchmark. Additionally, as shown in Fig. 7, its efficiency remains close to 100% across all cases, demonstrating excellent scalability. However, its bandwidth, at approximately 80 Gb/s, is notably below the file system's maximum bandwidth.

The PAR_R implementation also produces results consistent with the previous benchmark. Its execution time increases gradually with the number of MPI processes, and its relative efficiency reaches up to 80% for the largest case. The observed bandwidth is approximately 350 Gb/s, approaching the maximum bandwidth of the parallel file system.

In summary, this benchmark confirms the trends observed in production runs. The SYNC implementation's execution time increases significantly for larger cases, with high variability making it unreliable. In contrast, the collective implementations demonstrate scalable performance, with one (PAR_R) achieving a bandwidth close to the maximum capacity of the file system.

This benchmark highlights the variability of the SYNC implementation, with a variation factor of approximately 30 for one test case. It also highlights that the PARA_R implementation's bandwidth is close to the file system maximum bandwidth.

D. Strong Scaling

For this scaling, only the MPI processes partitioning along (r, θ) is increased. Thus, the memory footprint of an MPI process is divided by two between each case, as shown in Table IV.

Since we have 16 threads per MPI process, each nodes contains 8 MPI processes and a job running on 768 MPI processes corresponds to one rack, 1536 MPI processes corresponds to two racks, 3072 MPI processes corresponds to four racks and 6144 MPI processes corresponds to 8 racks, which means two different islets. We can assume this because the SLURM scheduler generally attempts to reserve adjacent nodes to minimise communication costs, with a few exceptions such as in the case of node failure.

Looking at all the implementations, we observe:

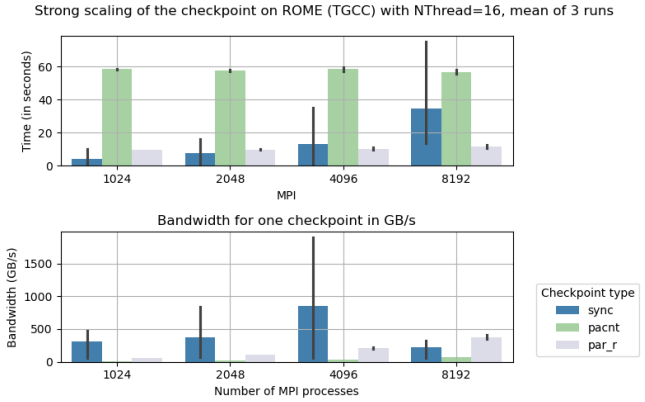

Fig. 6. Weak scaling plot with 16 threads per MPI process, averaged over 3 runs. The y-axis indicates the number of MPI processes, the x-axis shows the duration of each checkpoint in the top plot and bandwidth in the bottom plot. Each bar represents an implementation, from left to right, SYNC (in blue), PACNT (in green) and PAR_R (in grey). The SYNC implementation creates one file per process. The PACNT implementation creates one file per μ points, between 8 to 64 output files. Finally, the PAR_R methods creates sixteen times more files than the PACNT methods, between 128 to 1024, one per nodes.

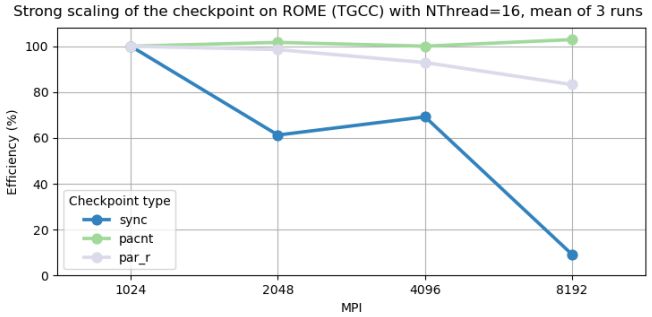

Fig. 7. The relative efficiency of the weak scaling with 16 threads per MPI process. The relative efficiency equation is defined in the previous subsection IV-A.

Each line represents a test case. The columns, from left to right, state the number of MPI processes, μ dimension points, the number of MPI processes allocated in each μ communicator along (r, θ), the size of the distribution f, the local memory footprint of f in an MPI process and the number of racks allocated for the simulation. The mesh grid used regarding $(N_r \times N_\theta \times N_\varphi \times N_{v_{G\|}} \times N_\mu)$ is 512x1024x128x128. The total size of the distribution function is 3 TB across all cases.

MPI proc.	nodes	N_μ	$p_r \times p_\theta$	local size	racks
768	96	48	4×4	4 GB	1
1536	192	48	8×4	2 GB	2
3072	384	48	8×8	1 GB	4
6144	768	48	8×16	500 MB	8

TABLE IV
OVERVIEW OF THE STRONG BENCHMARK CONFIGURATION.

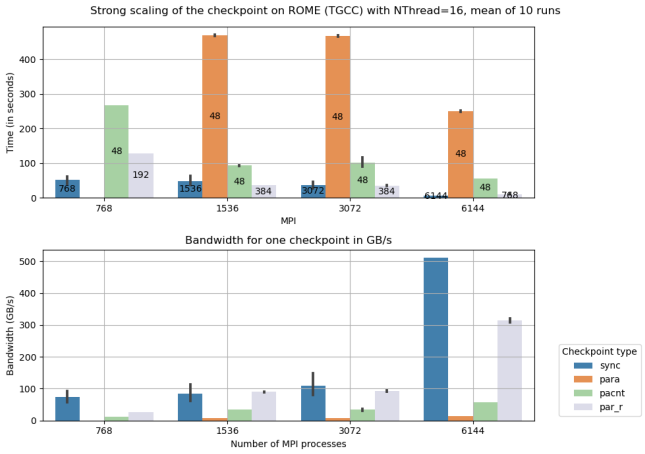

Fig. 8. Strong scaling plot, averaged over 3 runs. The y-axis indicates the number of MPI processes, the x-axis shows the duration of each checkpoint in the top plot and bandwidth in the bottom plot. Each bar represents an implementation, from left to right, SYNC (in blue), PARA (in orange), PACNT (in green) and PAR_R (in grey). The number of output files created for each simulation is displayed on each bar.

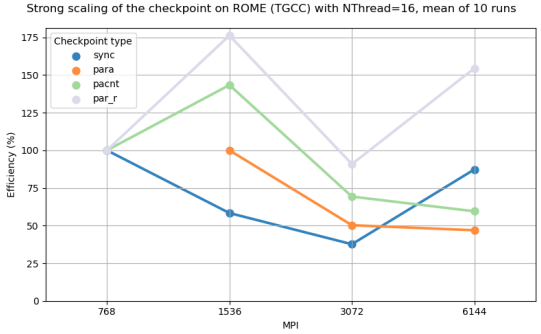

Fig. 9. The relative efficiency of the strong scaling. The relative efficiency equation is defined in the previous subsection IV-A.

- For the SYNC implementation, bandwidth improves, especially for the largest case (6144 MPI processes). However, efficiency drops significantly from case 1 to 3 due to metadata overhead, likely caused by the high number of output files, before recovering to 85% in the last case with two islets.
- For the PARA implementation, we could not execute the first case because of a SLURM limitation. For the other cases, we can see that from 2 to 4 racks, there is no time difference, then, from 4 racks to 8, the time is almost cut in half. The execution time is still at least four times higher than other implementations.
- For the last two implementations, we observe approximately the same behavior, from one rack to two, the time is cut in half, then from two to four racks, we do not observe any improvements, then from one islets to two, the time decreases again.

The efficiency metric, shown in Fig. 9, varies significantly across test cases compared to weak scaling. The efficiency of a single method does not follow a monotonic trend, indicating that the optimal parameters for maximum bandwidth depend on the number of nodes used. Choosing the best implementation involves balancing the number of files written simultaneously and the number of MPI processes writing collectively. While collective writing overhead remains constant, the overhead and variance from the number of written files depend on scaling size, with variance influenced by the number of islets used.

This benchmark underscores the role of the underlying architecture of the HPC center. Achieving an optimal balance between heavy collective writing and synchronous file writing depends on the system's architecture. Specifically, the architecture has an impact on the metadata performance, particularly on the overhead when writing on multiple files concurrently.

V. Conclusion

We have successfully identified effective methods for checkpointing, and the methodology outlined in this paper can be readily reproduced for use in other simulation codes. Moreover, in earlier tests, checkpointing took around 1,100 seconds, with a 55% efficiency, accounting for one-quarter of the total simulation time. In our tests, we reduced checkpointing time by 50% for the largest case between SYNC and PAR_R implementations. While we can't predict its performance at larger scales, we expect an improvement of up to 12.5% and, more importantly, less variation between checkpoints.

Further research can focus on optimising checkpointing at various levels. A key step is improving adaptability in the number of files created. This study aimed to balance the number of output files and the number of MPI processes writing collectively. One potential solution is to use the subfiling function in HDF5, which could be added to the existing plugin.

Another area for optimisation is the use of Lustre variables and HDF5 properties. We could better tune the file system variables and optimise the collective operations performed by HDF5 using MPI-IO.

Lastly, adaptive checkpointing is a promising direction. By calculating the optimal frequency for checkpoints, we could schedule them during periods of low parallel file system utilisation, reducing variability and improving overall I/O efficiency.

Acknowledgment

We would first like to thank Francieli Boito and François Tessier for their insights and help understanding I/O limitations and problematics. As part of the "France 2030" initiative, this work has benefited from a State grant managed by the French National Research Agency (Agence Nationale de la Recherche) attributed to the Exa-DoST project of the NumPEx PEPR program, reference: ANR-22-EXNU-0004. This project was provided with computing HPC and storage resources by GENCI at TGCC thanks to the grant 2024-A0160502224 on the supercomputer Joliot Curie's ROME partition.

REFERENCES

[1] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, C. Ehrlacher, D. Esteve, X. Garbet, P. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, C. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso, "A 5d gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations," vol. 207, pp. 35–68. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465516301230

[2] J. L. Bez, S. Byna, and S. Ibrahim, "I/o access patterns in HPC applications: A 360-degree survey," vol. 56, no. 2, pp. 46:1–46:41. [Online]. Available: https://dl.acm.org/doi/10.1145/3611007

[3] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 911–920, ISSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/document/8821049

[4] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, ser. CLADE '08. Association for Computing Machinery, pp. 15–24. [Online]. Available: https://dl.acm.org/doi/10.1145/1383529.1383533

[5] S. Byna, M. S. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "ExaHDF5: Delivering efficient parallel i/o on exascale computing systems," vol. 35, no. 1, pp. 145–160. [Online]. Available: https://doi.org/10.1007/s11390-020-9822-9

[6] M. J. Gossman, B. Nicolae, and J. C. Calhoun, "Modeling multi-threaded aggregated i/o for asynchronous checkpointing on HPC systems," in *2023 22nd International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 101–105, ISSN: 2379-5352. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10272431

[7] N. Wang, Q. Sun, Y. Liu, and D. Qian, "Mitigating i/o impact of checkpointing on large scale parallel systems," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 117–123. [Online]. Available: https://ieeexplore.ieee.org/document/8622785/citations#citations

[8] O. Thomine, J. Bigot, V. Grandgirard, G. Latu, C. Passeron, and F. Rozar, "An asynchronous writing method for restart files in the gysela code in prevision of exascale systems," vol. 43, pp. 108–116, publisher: EDP Sciences. [Online]. Available: https://www.esaim-proc.org/articles/proc/abs/2013/05/proc134307/proc134307.html

[9] A. Gueroudji, J. Bigot, and B. Raffin, "DEISA: Dask-enabled in situ analytics," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 11–20, ISSN: 2640-0316. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9680456

[10] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling." [Online]. Available: https://proceedings.scipy.org/articles/Majora-7b98e3ed-013

[11] R. Ross, R. Thakur, and A. Choudhary, "Achievements and challenges for i/o in computational science," vol. 16, no. 1, p. 501. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/16/1/069

[12] J. López-Gómez and J. Blomer, "Exploring object stores for high-energy physics data storage," vol. 251, p. 02066, publisher: EDP Sciences. [Online]. Available: https://www.epj-conferences.org/articles/epjconf/abs/2021/05/epjconf_chep2021_02066/epjconf_chep2021_02066.html

[13] R. Latham, C. Daley, W.-k. Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary, "A case study for scientific i/o: improving the FLASH astrophysics code," vol. 5, no. 1, p. 015001. [Online]. Available: https://dx.doi.org/10.1088/1749-4699/5/1/015001

[14] P. van Gemmeren, S. Binet, P. Calafiura, W. Lavrijsen, D. Malon, and V. Tsulaia, "I/o strategies for multicore processing in ATLAS," vol. 396, no. 2, p. 022054. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/396/2/022054

[15] K. Sen, S. Vadhiyar, and P. Vinayachandran, "Strategies for fast i/o throughput in large-scale climate modeling applications," in *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 203–212, ISSN: 2640-0316. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10487044?casa_token=vJAHb7gGEbIAAAAA:MxuysQBlQjCiVHyUsJCxMKEiExjc6FKgLhCi01fZ_aLsQ6i_qnvvKprGavdRQAqo3bJRkyCzJ1pZ

[16] V. Grandgirard, Y. Sarazin, X. Garbet, G. Dif-Pradalier, P. 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," vol. 871.

[17] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine, "Scaling gysela code beyond 32k-cores on bluegene/q," vol. 43, pp. 117–135, publisher: EDP Sciences. [Online]. Available: https://www.esaim-proc.org/articles/proc/abs/2013/05/proc134308/proc134308.html

[18] G. Latu, Y. Asahi, J. Bigot, T. Feher, and V. Grandgirard, "Scaling and optimizing the gysela code on a cluster of many-core processors," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 466–473, ISSN: 1550-6533. [Online]. Available: https://ieeexplore.ieee.org/document/8645933

[19] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11. Association for Computing Machinery, pp. 36–47. [Online]. Available: https://dl.acm.org/doi/10.1145/1966895.1966900

[20] C. Roussel, K. Keller, M. Gaalich, L. Bautista Gomez, and J. Bigot, "PDI, an approach to decouple i/o concerns from high-performance simulation codes." [Online]. Available: https://hal.science/hal-01587075

[21] "Lustre : A scalable , high-performance file system cluster." [Online]. Available: https://www.semanticscholar.org/paper/Lustre-%3A-A-Scalable-%2C-High-Performance-File-System/dc23ad6d4eb652718a2674486037454ec509eef5