PDI, an approach to decouple I/O concerns from high-performance simulation codes

Corentin Roussel, Kai Keller, Mohamed Gaalich, Leonardo Bautista Gomez, Julien Bigot

Research Report

PDI, an approach to decouple I/O concerns from high-performance simulation codes

Corentin Roussel*, Kai Keller†, Mohamed Gaalich*, Leonardo Bautista Gomez†, Julien Bigot* * Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, FR91191 Gif-sur-Yvette
Emails: {corentin.roussel, julien.bigot, mohammed.gaalich}@cea.fr † Barcelona Supercomputing Center, Barcelona, Spain
Emails: {leonardo.bautista, kai.keller}@bsc.es

Abstract—High-performance computing (HPC) applications manipulate and store large datasets for scientific analysis, visualization purposes and/or resiliency. Multiple software libraries have been designed for interacting with the parallel file system and in some cases with intermediate storage levels. These libraries provide different level of abstraction and have been optimized for different purposes. The best I/O library for a given usage depends on multiple criteria including the purpose of the I/O, the computer architecture or the problem size. Therefore, to optimize their I/O strategies, scientists have to use multiple API's depending on the targeted execution. As a result, simulation codes contain intrusive and library dependent I/O instructions interwoven with domain instructions. In this paper, we propose a novel interface that transparently manage the I/O aspects and support multiple I/O libraries within the same execution. We demonstrate the low overhead of our interface and perform experiments at scale with the Gysela production code.

Keywords—Separation of concern; I/O; HPC; Checkpointing; Software-engineering;

1 Introduction

Scientific simulations possess several programming aspects. One is the resolution of the system of equations that defines a physical models, another one is the post-processing of results, and a last one could be, for instance, data in- and out-put (I/O) operations. The first two aspects form the foundation of computational science.

I/O, on the other hand, is divided into the scientific regime, *what* to write (or read), and the technical regime, *how* to write (or read). In the following, the decoupling of these two distinct aspects is referred as *separation of concern*. On the one side, we have the concern corresponding to the physical model, which dictates where and what kind of data to read or write. And on the other side the rather technical concern, to handle the data in the most efficient and secure way. The latter also incorporates checkpointing to account for execution failures (e.g. caused by breakdown of hardware components, power outtakes, etc.).

Many different I/O dedicated libraries have been designed to encode the technical knowledge of data transfer and fault tolerance and hence a basic abstraction is already provided by the library API's. However, as we will explain in this article, some of the I/O concerns remain in the code.

To improve the separation of concern, we propose the parallel data interface (PDI), a novel interface that enables users to decouple the I/O strategies from the application code through a single API. PDI is not an I/O library by itself; it rather enables to use or implement different I/O strategies and to access them through a single API in a generic way. The API supports read- and write- operations using various I/O libraries within the same execution, and allows to switch and configure the I/O strategies without modifying the source (no re-compiling). However, it does not offer any I/O functionality on its own. It delegates the request to a dedicated library plug-in where the I/O strategy is interfaced. The range of functions and the performance of the underlying I/O libraries are not straitened, as shown in the evaluation (section 4).

In this section, we demonstrate that PDI significantly improves the separation of concerns. Using the production-level 5D Gyrokinetic semi-lagrangien (GYSELA) application [9], we show that, through PDI, one can decouple the semantic aspect of the I/O, from the performance aspect. We perform the evaluation for several strategies based on two I/O libraries, (FTI [2] & HDF5 [6]). We show that the additional abstraction layer comes at a very low cost with minimal overhead in terms of both, memory consumption and execution time.

As main contributions of this article we want to formulate: **1)** The design and implementation of PDI, a new interface that improve the separation of concern. It allows to access multiple I/O strategies and libraries without the need of modifying or recompiling the source code of the application. **2)** The adaptation of the GYSELA code to use either

FTI/HDF5 plain or embedded via PDI for checkpointing[1]. **3)** The evaluation at scale on different supercomputers with different architectures.

Our results show that PDI imposes a negligible overhead while running on over thousands of processes with a production-level application.

The remainder of the paper is organized as follows. Section 2 discusses the requirements for PDI and introduces its design while Section 3 presents its implementation. Section 4 evaluates PDI through an application at scale to the production code GYSELA. Section 5 reviews related work and Section 6 presents our conclusion.

2 PDI DESIGN

To clarify why a flexible interface such as PDI is a benefit for the HPC community, we would like to emphasize the significant amount of I/O libraries (section 5). As each library is optimized for some specific tasks, we remark that it is realistic to use different I/O libraries in a single scientific application. It could be very convenient to have an interface that allows to easily switch between I/O strategies. Those can be provided by the user or by an I/O library.

As mentioned in the introduction, HPC applications use I/O for multiple purposes. One can for example distinguish between diagnostics and checkpoints but other categories could be identified and even the checkpoint category could be further refined (for example, resiliency checkpoints for fault-tolerance purpose *vs.* segmentation checkpoints between two restarts of the simulation.) The purpose of the I/O is however not the only criterion that impacts the best strategy to use and one must also take into account other elements such as the architecture of the supercomputer used, the degree of parallelism, the memory size of the simulated problem, the developer time available for implementation, etc.

A given I/O library could theoretically provide coverage of all the desired features. However, those are very numerous due to the combinatorial complexity that comes from both codes requirements and the possibilities offered by execution environments. Therefore, such a library would have to provide complex API's that enable the code to finely define the expected behavior. Instead, many I/O libraries focus on more specific situations and are thus less complex. To reach the best performance for each situation, it is possible to use various API's and/or to change the way a given API is used every time the application runs in a different environment.

Modifying the I/O strategy in a code is a tedious work that is not the main concern (and often nor the main domain of expertise) of the application developer. I/O experts can provide new I/O strategies or optimized existing ones. However, for each new library that is used, and with each line of code that is added to handle a specific I/O concerns, the code becomes harder to understand and to maintain. Moreover, when the number of dependencies is increased, the portability of the code decreases. As a result, suboptimal I/O code is used in many applications at the cost of millions (if not billions) of dollars of wasted computer time every year.

1. In both cases we verify the validity of the written data sets.

The main goal of PDI is to separate the I/O aspects from the domain code and thus to improving the separation of concerns. PDI is a glue layer that sits in-between the implementation of these two aspects and interface both of them.

PDI has been designed in such a way that the separation of concern does not come at the expense of good properties of existing approaches. The implementation of a given I/O strategy through PDI should be as efficient and as simple (ideally less complex) than existing approaches, both from the user and I/O expert point of view. This should hold, whatever the level of complexity of the I/O strategy, from the simplest one where the implementation time is paramount, to the most complex one where the evaluation criterion is the performance on a specific hardware.

We therefore design PDI to act as a *lingua franca*, a thin layer of indirection that exposes a declarative API for the code to describe the information required for I/O and that offers the ability to implement the I/O behavior using these informations. In order to decouple both sides, we rely on a configuration file that correlates information exposed by the simulation code with that required by the I/O implementation and enables to easily select and mix the I/O strategies used for each execution. This approach brings important benefits as it improves the separation of concerns thanks to the two abstraction layers. It offers a simple API that allows a uniform code design while accessing and mixing the underlying I/O libraries.

We design the PDI API to support concepts as close as possible to those manipulated by simulation codes. Using PDI, the simulation code exposes information that it would handle in any case. It does not have to generate additional information dedicated to I/O management. This makes the API easy to grasp for code developers and prevents computations dedicated to I/O from cluttering the code, impacting readability or adding useless overheads.

The API limits itself to the transmission of information (required by the I/O implementation) that can only be provided during execution. Information that is known statically is expressed in the configuration file. The only elements that have to be described through the API are therefore: **1)** the buffers that contain data with their address in memory and the layout of their content, **2)** the time period along execution when these buffers are accessible either to be read or written. In addition, the API handles the transmission of control flow from the code to the library through an event system. Events are either generated explicitly by the code or generated implicitly when a buffer is made available or just before it becomes unavailable.

The data layout is often at least partially fixed, only some of its parameters vary from one execution to the other (*e.g.* the size of an array). We therefore support the description of this layout in the configuration file so as not to uselessly clutter the application code. The value of parameters that are only known during the execution can be extracted from the content of buffers exposed by the code.

Implementing an I/O strategy is done by catching the control flow in reaction to a event emitted by the simulation code and using one ore more of the exposed buffers. The name of the events and buffers to use come from the configuration file, ensuring a weak coupling between both

side. Two levels of API are offered. A low level API enables to react to any event and to access the internal PDI data structures where all currently exposed buffers are stored. A higher level API enables to call user-defined functions to which specific buffers are transmitted in reaction to well specified events.

When using the low-level API, it is the responsibility of the I/O code implementation to access the configuration file to determine the events and buffers to use. This API is well suited for the development of plugins that require a somewhat complex configuration because they are intended to be reused in multiple codes. This is typically the case when interfacing I/O libraries with declarative API's close to that of PDI where options in the configuration file are enough to match the API's.

User can implement their own I/O strategies that can be interfaced with PDI. When using user-defined functions, the name of the events and buffers passed to the function are specified in the configuration file in a generic way. The function itself does neither have access to the configuration file content nor to the list of shared buffers.

This approach is less flexible but much easier to implement. It is well suited when a specific code has to be written to use a given I/O library in a given simulation code as is often the case with libraries with imperative API's. It can be used to provide additional instructions that complement but are distinct from the library features.

In order to decouple this I/O implementation code both from PDI and from the simulation code, it is defined in dedicated object files that can either be loaded statically or dynamically (a plugin system). This means that PDI does not depend on any I/O library, only its plugins do. This also simplifies changing strategy from one execution to the other as the plugins to load are specified in the configuration file.

To summarize, PDI offers a declarative API for simulation codes to expose information required by the implementation of I/O strategies. The I/O strategies are encapsulated inside plugins that access the exposed information. A week coupling mechanism enables to connect both sides through a configuration file. This can be understood as an application of aspect oriented programing (AOP) to the domain of I/O in HPC. The locations in the simulation code where events are emitted are the *joint points* of AOP. The I/O behavior encapsulated in the plugins are the *advices* of AOP. The configuration file specifies which behavior to associate at which location and constitute the *pointcuts* of AOP.

3 PDI Implementation

PDI is freely and publicly available[2] under a BSD license. It is written in C and offers a C API with Fortran bindings to the simulation code. This covers uses from C, C++ and Fortran, the three most widespread languages in the HPC community. We present the C flavor in this section but the Fortran binding offers the exact same interface. The plugin API is currently limited to C but bindings for other languages (*e.g.* LUA, Python) are planned.

The simulation code API contain functions to initialize and finalize the library, change the error handling behavior, emit events and expose buffers as presented in Listings 1. The initialization function takes the library configuration (a reference to the content of a YAML [3] file) and the world MPI communicator that it can modify to exclude ranks underlying libraries reserve for I/O purpose. The error handling function enables to replace the callback invoked when an error occurs. The event function takes a character string as parameter that identifies the event to emit.

```
1  enum PDI_inout_t { PDI_IN=1, PDI_OUT=2, PDI_INOUT=3 };
2
3  PDI_status_t PDI_init(PC_tree_t conf, MPI_Comm *world);
4  PDI_status_t PDI_finalize();
5
6  PDI_errhandler_t PDI_errhandler(PDI_errhandler_t handler);
7
8  PDI_status_t PDI_event(const char *event);
9
10 PDI_status_t PDI_share(const char *name, void *data,
       PDI_inout_t access);
11 PDI_status_t PDI_access(const char *name, void **data,
       PDI_inout_t access);
12 PDI_status_t PDI_release(const char *name);
13 PDI_status_t PDI_reclaim(const char *name);
```

Listing 1. The PDI public API

```
1  PDI_status_t PDI_export(const char *name, void *data);
2  PDI_status_t PDI_expose(const char *name, void *data);
3  PDI_status_t PDI_import(const char *name, void *data);
4  PDI_status_t PDI_exchange(const char *name, void *data);
5
6  PDI_status_t PDI_transaction_begin(const char *name);
7  PDI_status_t PDI_transaction_end();
```

Listing 2. Simplified PDI API for buffer exposing

The most interesting functions of this API are however the buffer sharing functions. They support sharing a buffer with PDI identified by a `name` character string and with a specified `access` direction specifying that information flows either to PDI (`PDI_OUT`, read-only share), from PDI (`PDI_IN`, write-only share) or in both directions (`PDI_INOUT`.) The `PDI_share` and `PDI_access` function start a buffer sharing section while the `PDI_release` or `PDI_reclaim` function end it. `PDI_share` is used for a buffer whose memory was previously owned by the user code while `PDI_access` is used to access a buffer previously unknown to the user code. Reciprocally, `PDI_reclaim` returns the memory responsibility to the user code while `PDI_release` releases it to PDI.

In a typical code, the buffers are however typically shared for a brief period of time between two access by the code. The previously introduced API requires two lines of code to do that. The API presented in Listing 2 simplifies this case. Its four first functions define a buffer sharing section that lasts during the function execution only. The functions differ in terms of access mode for the shared buffer: `share(OUT) + release` for `PDI_export`; `share(OUT) + reclaim` for `PDI_expose`; `share(IN) + reclaim` for `PDI_import`; and, `share(INOUT) + release` for `PDI_exchange`.

This API has the disadvantage that it does not enable to access multiple buffers at a time in plugins. Each buffer sharing section ends before the next one starts. The two transaction functions solve this. All sharing sections enclosed between calls to these functions have their end delayed until

2. https://gitlab.maisondelasimulation.fr/jbigot/pdi

```
data:
  my_array: { sizes: [$N,$N], type: double }
metadata:
  N: int  # data id and type
  it: int
plugins:
  declh5: # plug-in name
    outputs:
      # data to write, dataset and file name
      my_array: { var: array2D, file: example_$it.h5,
        # condition to write
        select: ($it>0) && ($it%10) }
```

Listing 3. Example of PDI configuration file

```
main_comm = MPI_COMM_WORLD
call PDI_init(PDI_subtree, main_comm)
call PDI_transaction_begin("checkpt")
ptr_int=> N; call PDI_expose("N",ptr_int)
ptr_int=> iter; call PDI_expose("it",ptr_int)
call PDI_expose("my_array",ptr_A)
call PDI_transaction_end()
call PDI_finalize()
```

Listing 4. Example of PDI API usage

the the transaction ends. This effectively supports sharing of multiple buffers together. The transaction functions also emit a named event after all buffers have been shared and before their sharing section ends.

At the heart of PDI is a list of currently shared buffers. Each shared buffer has a memory address, a name, an access and memory mode and a content data type. The access mode specifies whether the buffer is accessible for reading or writing and the memory mode specifies whose responsibility it is to deallocate the buffer memory. The content data type is specified using a type system very similar to that of MPI and is extracted from the YAML configuration file.

The `data` section of the configuration file (example in Listing 3) contains an entry for each buffer, specifying its type. The type can be a scalar, array or record type. Scalar types include all the native integer and floating point of Fortran and C (including boolean or character types.) Array types are specified by a content type, a number of dimensions and a size for each dimension. They support the situation where the array is embedded in a larger buffer with the buffer size and shift specified for each dimension. Record types are specified by a list of typed and named fields with specific memory displacement based on the record address.

The types can be fully described in the YAML file, but this makes them completely static and prevents the size of arrays to change at execution for example. Any value in a type specification can therefore also be extracted from the content of an exposed buffer using a dollar syntax similar to that of bash for example. The syntax supports array indexing and record field access. For the content of a buffer to be accessible this way, it does however needs to be specified in the `metadata` section of the YAML file instead of its `data` section. When a metadata buffer is exposed, its content is cached by PDI to ensure that it can be accessed at any time including outside its sharing section.

The plugins to load are specified in the `plugins` section of the configuration file. Each plugin is loaded statically if linked with the application and dynamically otherwise. A plugin defines five function: an initialization function, a finalization function and three event handling functions. The event handling functions are called whenever one of the three types of PDI event occurs, just after a buffer becomes available, just before it becomes unavailable and when a named event is emitted.

The plugins can access the configuration content and the buffer repository. Configuration specific to a given plugin is typically specified under this plugin in the `plugins` section of the YAML file. The YAML file can however also contain configuration used by plugins in any section. It can for example contain additional information in a buffer description.

Three plugins are used in this paper. The *FTI* plugin interfaces the declarative FTI library, the *decl'H5* plugin interfaces a declarative interface built on top of HDF5 and the *usercode* plugin supports user written code as specified in Section 2. A plugin interfacing a declarative version of SIONlib is also available in the repository, but most imperative libraries are best accessed through the *usercode* plugin.

Let us now present an example to show how PDI usage works in practice. Listing 4 shows the use of the Fortran API to expose to PDI two integers: `N` and `it`, and an array of dimension `N`×`N`, `my_array`. The configuration file for this example is the one presented in Listing 3.

When the `PDI_init` function is called, the configuration file is parsed and the *decl'H5* plugin is loaded. This plugin initialization function is called and analyzes its part of the configuration to identify the events to which it should react. No plugin modifies the provided MPI communicator that is therefore returned unchanged. A transaction is then started in which three buffers are exposed: `N`, `it` and `my_array`. The *decl'H5* plugin is notified of each of these events but reacts to none. The transaction is then closed that triggers a named event to which the *decl'H5* plugin does not react as well as three end of sharing section events, one for each buffer. The *decl'H5* reacts to the end of the `my_array` sharing since this buffer is identified in the configuration file. It evaluates the value of the `select` clause and if nonzero writes the buffer content in a dataset whose name is provided by the `var` value ("array2D") to a HDF5 file whose name is provided by the `file` value ("example$it.h5".)

4 Evaluation

To evaluate PDI we check that we attain the main goals introduced in Sections 2. These goals are that: **1)** PDI should separate I/O concerns from the simulation code, **2)** changing and combining different I/O strategies should be transparent inside the application, **3)** the implementation of an I/O strategy through PDI should be (at least) as simple as without PDI, **4)** all the features of the I/O libraries used through PDI should be accessible, **5)** similar performance as with a direct use of a library should be accessible through PDI.

Points 1 and 2 ensure that PDI provides a new approach for I/O in a HPC context with added value compared to existing approaches in term of decoupling. Points 3 to 5 validate that PDI does not impose any artificial limitations

compared to existing approaches that would hinder its acceptability.

The tests are carried out using the GYSELA production code on different platforms. The remaining of this section describes the application and introduce our experiments, which used GYSELA code. We evaluate PDI approach both in terms of software engineering (Separation of concern and ease of use) and performance.

4.1 Example Application

GYSELA [9], [11] is a scientific code that models the electrostatic branch of the ion temperature gradient turbulence in tokamak plasmas. The code consists of around 60 000 lines of code: 95% Fortran and 5% C, parallelized using a hybrid MPI+OpenMP approach. It has run on up to 1.8 million threads on a Blue Gene/Q machine with 91% relative efficiency [4], [8] excluding I/O. At the heart of GYSELA is a self-consistent coupling between a 3D Poisson solver and a 5D Vlasov solver. The main data manipulated in the code from which all other values are derived is a 5D particle distribution function in phase space. The array containing this dataset is referred to as `f5D`.

A typical GYSELA run is sized based on memory requirements first, and typically uses from one thousand to a few thousands nodes (16,000 cores is a usual value). The wall-clock time for a complete simulation is then often in the range from one week to a few months. For executions of such duration, a checkpoint/restart mechanism is a requirement for both, fault-tolerance and segmenting, *i.e.* running the simulation in multiple subsequent jobs. Indeed, most job schedulers limit jobs duration to between 12 and 48 hours.

The complete simulation state can be derived from the 5D particle distribution function (`f5D`) and a few scalar values (time-step, etc.) These are thus the values stored in checkpoint files. The `f5D` array usually represents in the order of one quarter of the total memory consumed. Storing it too often would represent a large overhead both in term of time and storage space. Therefore, actual results exploited by physicists, take the form of *diagnostics*: smaller (0D-3D) values derived from the 5D distribution function and written to permanent storage regularly. In term of I/O these two parts of the code have different requirements.

Storing diagnostics is usually not an issue due to their relatively small size. On the contrary, storing checkpoints can consume up to 1h for the biggest simulations resulting in an overhead in the order of 10%. Depending on the simulation size and platform different strategies should be used to limit this overhead. Our experiments thus focus on the case of the checkpoints.

By default, GYSELA uses the HDF5 library to store checkpoints. Its portable file-format is used in GYSELA post-processing tools. Another option is the FTI multilevel checkpointing library that leverages node-local storage (such as SSDs) for burst buffering and can be configured to use a dedicated MPI task per node in order to overlap I/O with computation. This is mostly useful for intermediate fault-tolerance checkpoints as no overlap is possible for segmentation checkpoints performed at the end of a run. To store the dataset, FTI uses a binary file-format that is tuned for performance. This file format is not portable across different architectures and make it difficult to analyze data. From the authors point of view, the two libraries are complementary and the code would benefit from their combination to obtain the best from two very different approaches.

We have upgraded the GYSELA code to support FTI and PDI. Through PDI, we have implemented support for decl'H5, HDF5 and FTI. The decl'H5 approach simply specifies in the configuration the list of buffers to write in an HDF5 file whose name depend on the current time-step. The HDF5 approach adds additional logic to mimic GYSELA HDF5 use, alternating writing between two files and selecting the latest coherent file for reading. This logic and the HDF5 usage is implemented through dedicated functions passed to the usercode plugin. The FTI approach simply maps PDI event to FTI calls in the configuration and lets FTI use is pre-defined logic for checkpoints writing and reading.

Each approach has its advantages and drawbacks and targets different use-cases with different requirements such as simplicity for decl'H5, limited disk overhead and file portability for HDF5, I/O and computation overlap for FTI. In addition and as a baseline, we support execution with no checkpoint writing at all. Experiments done using no checkpointing, HDF5 and FTI without PDI are denoted *none*, *HDF5* and *FTI* respectively in the remaining of the section. Experiments done using no checkpointing, decl'H5, HDF5, FTI and a combination of HDF5 and FTI through PDI are denoted *none*, *decl'H5*, *HDF5*, *FTI* and *FTI+HDF5* with a PDI prefix.

Our experiments are organized as follows. We evaluate the simplicity of usage from the I/O strategy implementation side first and application code side second. In the process we also discuss the level of separation of concern offered by the approach. Finally, we evaluate performance on 4 different clusters and supercomputers: Nord (BSC, Spain), Curie (TGCC, France), Poincare (IDRIS, France) and Jureca (JSC, Germany).

4.2 Software Engineering Evaluation

No perfect indicator exist to measure the simplicity of use of a library. In our experiment we settle to measure the number of lines of code that even if not ideal gives a rough idea of the amount of work required. We also study the number of distinct functions used in a piece of code that gives an idea of the complexity for writing this code.

From		To: Native HDF5	To: Native FTI	To: PDI decl'H5	To: PDI HDF5	To: PDI FTI
Nat.	HDF5	**931**	895	891	885	869
	FTI	895	**216**	199	205	189
PDI	decl'H5	891	199	**238**	19	5
	HDF5	885	205	15	**242**	3
	FTI	869	189	31	33	**211**

TABLE 1
Number of lines in GYSELA to support a given I/O strategy. On the diagonal: total number of line. Non-diagonal: number of line modifications required to change the I/O strategy.

In Table 1 we report the number of physical line of source codes required in GYSELA to support the HDF5 and FTI I/O strategies with and without PDI. As expected, HDF5

requires more lines than FTI due to the declarative and domain specific nature of FTI. Due to its declarative nature, PDI requires a number of lines in the same order as FTI. Thus, while supporting different approaches PDI intrusiveness is small and comparable with a domain specific library.

Another element that can be noticed on the table is the effort required to change the implementation from one strategy to another. Using the native libraries (HDF5 or FTI) API's, only a very little amount of code (less than 2%) can be reused from one implementation to the other. Using PDI, the implementation of an additional strategy only requires very little additional lines of code (1 to 10%). The typical addition required is for example a pair of transaction enclosing calls or the exposure of an additional buffer. We would also like to strongly stress that the addition of these lines does not prevent other versions from working. A version supporting the combination of all three plugins requires less than 300 lines of code. Using PDI, a change of I/O strategy has therefore very limited impact on the code which strongly improves the separation of concern.

	To		
From	decl'H5	HDF5	FTI
decl'H5	**150**	6	1
HDF5	3	**153**	1
FTI	2	6	**148**

TABLE 2
Number of lines used to declare data in the PDI configuration. On the diagonal: total number of line. Non-diagonal: number of line modifications required to change the I/O strategy.

When using PDI, one does not only have to make function calls in the code, but also to provide a configuration file. The content of this file can be separated in two parts. The first area contains the declaration of data and metadata buffers exposed to PDI while the second contains the configuration of the various I/O strategies. In Table 2 we report the number of lines used to declare the data and the metadata. The number of lines to modify to implement a new I/O strategy is very low. These lines correspond to buffers that have to be exposed for one strategy but not in another. Once again, providing information for buffers that are not used in a given strategy is not a problem and a configuration file supporting all three approaches requires less than 160 lines. The data type declaration part of the configuration is therefore also only very slightly impacted by the choice of I/O strategy.

decl'H5	HDF5	FTI	FTI + HDF5
12	21 (665)	9	30 (665)

TABLE 3
Number of lines in the PDI configuration file dedicated to the management of the I/O strategy. The number of lines of user source code is given in parenthesis, when appropriate.

In Table 3 we report the number of lines required to specify the I/O strategy. This part only requires between 9 and 30 lines of configuration. The case of the HDF5 approach also require dedicated code to implement the same logic as in the original GYSELA code through the usercode plugin. The 665 lines amount to slightly more than two third of the 931 lines of the original code identified in Table 1. This is due to two aspects: C is slightly more concise than Fortran and PDI handles a part of the boilerplate code required for this implementation, letter the developer focus on the actual strategy implementation. The table also presents the FTI+HDF5 implementation that simply combines the FTI and HDF5 version to use FTI for intermediate fault-tolerance checkpoints and HDF5 for final segmentation checkpoints. From these figures, one can notice that choosing a given I/O strategy through PDI is very simple for declarative libraries and requires no more work than with a native approach for imperative ones. Once implemented through PDI, the combination of multiple strategies is also greatly eased and only requires selecting in which situation to apply what strategy.

To summarize, PDI improves the separation of concern by minimizing the number of lines related to the I/O strategy implementation impacting the simulation code and by making these lines independent of the strategy used. The description of the buffers content type in the configuration is also independent of the strategy. The strategy choice is relegated to the dedicated part of the configuration file. Its implementation for imperative libraries is provided through well defined functions passed to the usercode plugin. From this study, we can see that PDI clearly separates aspects related to the simulation on one side with the PDI API calls and the specification of the buffer content and elements related to the I/O strategy on the other side with the I/O strategy configuration and its potential implementation through the usercode plugin.

		To	Native		PDI		
	From		HDF5	FTI	decl'H5	HDF5	FTI
Nat.	HDF5		**24**	31	30	32	31
	FTI		31	**7**	13	15	14
PDI	decl'H5		30	13	**6**	2	2
	HDF5		32	15	0	**8**	0
	FTI		31	14	1	1	**7**

TABLE 4
Number of distinct API functions from FTI, HDF5 or PDI. On the diagonal: total number of distinct functions. Non-diagonal: number of distinct functions required to change the I/O strategy.

In Table 4 we give the number of different API functions required to use HDF5 or FTI in GYSELA natively or through PDI. Similarly to the number of lines analyzed in Table 1 the declarative API's of PDI and FTI minimize the complexity. This approach requires about use about a third of the number of distinct functions required for the imperative API of HDF5. Similarly to the observation made for the number of lines of code, the implementation of new strategies when using PDI has only very limited impact on the simulation code with 1 or 2 functions to add while with the use of native API's it requires invasive changes.

Once again, these number do not represent the complete complexity of an I/O strategy implementation. One must also take into account the I/O strategy configuration identified in Table 3. This is especially true for the usercode plugin HDF5 support relying on dedicated code implementing the same logic as the native HDF5 code and hence using a similar number of distinct functions from HDF5 API. One must however notice that this only one option amongst

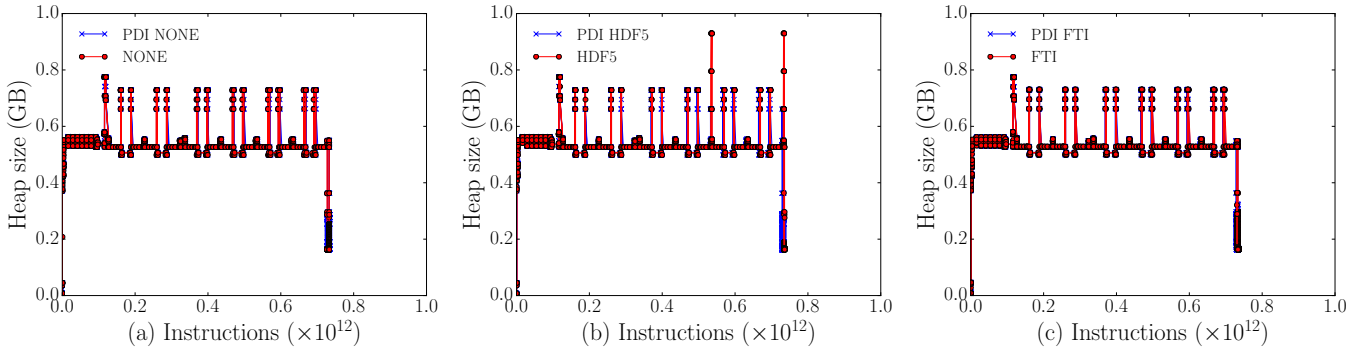

Fig. 1. Memory usage as a function of the number of instructions with and without PDI library for three I/O strategies: (a) no I/O, (b) using HDF5 library (c) using FTI library. Experiments are done on 4 nodes (64 cores) of the Nord supercomputer.

many. A user not willing to invest much time in I/O can rely on the decl'H5 approach and only switch to the usercode plugin when it becomes critical in term of performance.

From our observations, PDI will add less I/O directives and fewer lines inside the application code. It also reduce the number of distinct API required. Furthermore, the complexity and time to implement the PDI version of the code supporting both libraries together is much lower than using their native interface.

Finally, all the FTI or HDF5 features that were previously used inside GYSELA code are supported by the plug-ins. There is no loss of functionalities between the GYSELA version using the native API of the HDF5 version and the one using HDF5 through the PDI API.

To summarize, we have shown that PDI improves the separation of concern and that I/O aspects are mostly removed from the application code. In order to support PDI, an initial investment similar to that required to support a simple domain specific I/O library with a declarative interface is required. The number of lines to insert inside the application code and the number of API functions is minimized. Once this initial work done, the separation of concerns offered by PDI ensure that replacing the I/O strategy by another can be done at minimal cost and with minimal impact on the simulation code. These properties come at no loss of functionality and with the additional possibility to use multiple libraries for different purposes.

4.3 Performance Evaluation

In this section we check that PDI has no impact the performance of the application. We evaluate the memory overhead of PDI, present the results of a weak scaling experiment and compare the efficiency of five I/O strategies for four different use-cases. In all experiments, FTI-based strategies use one dedicated MPI task per node. GYSELA requires an even number of threads and we therefore use two cores less for the computation in this situation. All other strategies use all the cores available on the nodes.

4.3.1 Memory evaluation

In this section we perform a study on the memory overhead caused by PDI. The experiments are performed on the Nord supercomputer based on Marenostrum. Nord has two E5–2670 SandyBridge processor (2.6GHz) on each compute node with 128GB of DDR3 RAM. The total number of thread per node is 16. GYSELA and PDI use of the stack is negligible. Thus, the heap size during the simulation corresponds to the memory consumption. In Figure 1 we report the memory allocated on the heap for all MPI processes as a function of the number of instructions. The heap size over time is tracked using the massif tool of Valgrind [17]. We chose not to show the MPI task dedicated to FTI I/O due to their low maximum memory consumption (which is lower than 10MB).

For all curves we observe similar number of instructions with and without PDI. A first peak appears at the beginning of the initialization and is followed by a plateau. Since the computation uses buffers, oscillations appear. The heap size exhibits the same pattern between all cases except for the HDF5 cases that are using additional buffers to copy the main array (`f5D`) without the so-called ghost cells to do the checkpoint. Taken as a whole the curves show that there is no significant overhead in the memory consumption when using PDI. PDI only manipulate pointer to buffers and perform no additional copy.

4.3.2 Weak scaling study

In order to demonstrate that PDI does not impact the scaling of parallel I/O, we perform a weak scaling experiment on Curie supercomputer. Curie thin node have 2 eight-core Sandy Bridge E5-2680 (2.7 GHz) with 64 GB of RAM and one local SSD disk.

In Figure 2 we plot the wall clock time with and without PDI for the different I/O strategies (no I/O, HDF5, FTI) against the number of nodes. For each case one run consist of three iterations. Two checkpoints are saved, one at the end of the second iteration and one at the end of the run. For each checkpoint and on each node the simulation produces approximately 8.6GB of data resulting in a total data size ranging from 34.3GB for the 64 cores experiments up to 1.1TB for the 2048 cores runs.

PDI library does not add any significant overhead to the time to completion of GYSELA for the tested cases. When observed, the overhead is negligible and may be caused by the uncertainties of the measured. This result demonstrates that the impact of PDI on performance is negligible and especially that is has no noticeable impact on scalability.

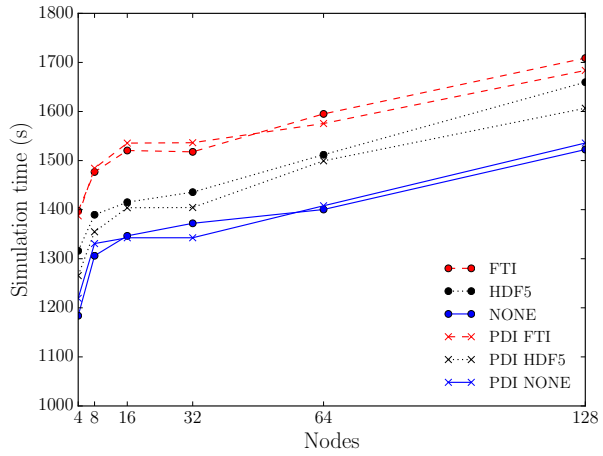

Fig. 2. Weak scaling study between 64 cores and 2048 cores with and without PDI for different I/O strategies: no I/O, using FTI or the gysela HDF5 strategy. Weak scaling studies are carried out on Curie supercomputer.

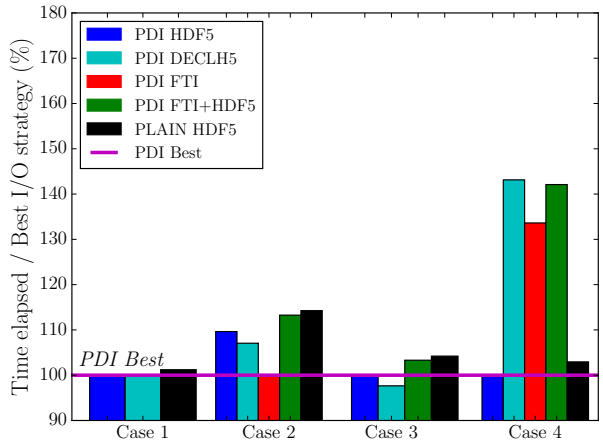

Fig. 3. Efficiency breakdown for various usage across different architectures. Case 1, uses 24 cores (1 node) of Jureca Supercomputer, the checkpoint size is 520 MB; case 2 and 3, use 16384 (1024 node) of Curie supercomputer, the checkpoint size is 4.4 TB; case 4, uses 64 cores (4 nodes) of Poincare supercomputer, the checkpoint size is 8.5 GB.

4.3.3 *Efficiency breakdown for four use cases*

On figure 3 we plot the wall clock time for different I/O strategies normalized by what we consider as the best approach, for four use cases. For each case one run consist of three iterations. Two checkpoints are saved, one at the end of the second iteration and one at the end of the run.

The first case corresponds to a small debug run on a single node of Jureca supercomputer that has two Intel Xeon E5-2680 v3 Haswell CPUs per node with 12 physical cores at 2.5 GHz and 128GB of DDR4 RAM. The second case corresponds to a large-scale simulation running on 16 384 cores of the Curie supercomputer. The third case corresponds to a simulation of the same scale on Curie but in a situation where the code is intended to be restarted on a different computer requiring the files to be portable. The fourth case corresponds to a medium-scale debug run on 4 nodes the Poincare cluster.

The first case uses the 24 physical cores of one node on the Jureca supercomputer and the main variable (`f5D`) requires 512MB/checkpoints. FTI can not run in its nominal mode on a single node as it requires at least four node for data replication. We do therefore not evaluate this approach in this case. In term of wall-clock time all evaluated approaches are very similar. For such a small case, filling the disk with checkpoints is not a concern and keeping all checkpoint files might actually be useful for debug purposes. The decl'H5 strategy that requires only minimal investment in term of development and no involvement of any I/O specialist does thus offer the best compromise in this case.

For the second case, the experiments consists of running GYSELA code for 3 iterations on 16384 cores (1024 nodes) of the Curie supercomputer. An intermediate checkpoint is written at the end of the second iteration and a segmentation checkpoint at the end of the run. Each checkpoint consumes 4398.0 TB of storage (around 4.3GB per node). We evaluate five checkpointing strategies that are meaningful: using FTI or decl'H5 library or using the native HDF5 based strategy in its initial version or through PDI. The last approach consists of using PDI to combine FTI to store the intermediate checkpoint and the HDF5 based approach to store the final checkpoint. The decl'H5 approach can not be realistically used in this case as it would quickly fill the disk for a longer run. From a performance point of view, the best option is FTI that therefore offers the best compromise in this case.

The third case consist in the exact same simulation with the additional constraint of producing portable checkpoint files at the end of the simulation. Once again the decl'H5 approach can not be used due to disk space limitations. The FTI strategy that uses a non-portable binary file format for the final checkpoint is not an option either. The remaining strategies all HDF5 for the final checkpoint and can be used. For this experiment, PDI using the native HDF5 approach is the more efficient approach and best solution. For a more realistic longer run, the ratio of intermediate checkpoints over final one would typically be higher and the FTI+HDF5 approach would likely be competitive.

For the last case, the experiments are done on four nodes of the Poincare cluster. Each nodes writes 2.1GB of data and he code therefore produces 8.4GB checkpoint files. Unlike Jureca or Curie, Poincare does not have SSD's on its compute nodes but HDD's. In this case, the burst buffering strategy of FTI is not competitive. A comparison between decl'H5 and the usercode HDF5 shows that the second is faster. This is likely due to an inefficient implementation "hyperslab" feature of the HDF5 library used in decl'H5 on the version of the library available on this cluster. The HDF5 approach with or without PDI are both using intermediate buffers and are more efficient on this specific cluster.

To summarize we have performed different experiments on 3 clusters. We observe that for each of the four use cases a different I/O strategy is preferred. Using native API's this situation would make the choice of the strategy to implement in the code difficult. Thanks to PDI, a single implementation makes it possible to access all strategies and choose the best one for each execution simply by changing the configuration file. The decl'H5 can be implemented with a very limited knowledge of I/O aspects and other strategies can be provided by expert with very limited impact

on the simulation code. This approach solves a problem often encountered in the development of high-performance simulation codes lead by experts of the simulated domain.

5 Related Work

There are many I/O libraries for various purposes. Operating systems offer standardized I/O services such as the POSIX I/O API on which language standard libraries typically build portable interfaces such as the ANSI C library. However, these where not designed with parallel codes in mind. Therefore, specialized libraries exist for parallel codes. Some of them are optimized for fast checkpointing in multiple storage levels, others aim to maximize data readability and portability, and yet others optimize for parallel writing in shared files. In the following we introduce some of the most widely used I/O libraries in the HPC community.

The message passing interface I/O API (MPI-IO) [22], [24] allows HPC applications, using the MPI standard, to write files in a parallel and collaborative fashion while keeping coherency. MPI-IO uses a combination of portable code and architecture-dependent optimized modules. MPI-IO covers a large spectrum of features, including collective and asynchronous I/O, non-contiguous accesses and atomicity semantics [23]. The library has been optimized, to work with multiple file systems [5] such as NFS, GPFS [18], Lustre and others. In addition to these features, MPI-IO has also implemented client-side caching, to reduce the amount of data transfer between client nodes and I/O servers. However, this also produces coherence problems when some files are accessed by multiple clients. Techniques, such as aligned accesses according to the system stripe size [13], have been studied and show important performance benefits. Nonetheless, writing on the file system is still bounded by the theoretical peak I/O bandwidth, which does not scale with the number of compute nodes (opposed to local storage).

The hierarchical data format (HDF5) [6] is perhaps the most widespread I/O library in scientific computing. It comes in two flavors: a sequential version and a parallel version, based on MPI-IO, that enables multiple parallel processes to work on a single coherent file [25]. HDF5 standardizes both, a library API and a self-describing file format. This makes it possible to move files from one machine to another without worrying about different architectures whose internal data representation might differ [10]; however, this means that the cost for transforming the data from the internal representation to the file representation has to be paid for every write. Checkpoint writing using HDF5 can either use the sequential or the parallel approach. The parallel approach makes it possible to write to a single file and thus reload it in a subsequent execution with a different domain decomposition. However, this incurs overheads due to synchronization. In contrast, writing one file per process requires to reuse the same domain decomposition for restart but can be much more efficient.

The network common data format (NetCDF) [19], [20] is another widespread library for scientific computing I/O, especially in the climate modeling community [21]. The last version of NetCDF is based on HDF5 and thus shares most of its advantages and drawbacks. It provides a parallel version [12] that relies on MPI-IO like HDF5. The library is well-known for its programming convenience and its parallel version provides high performance dataset handling.

The fault tolerance interface (FTI) [2] is a multilevel checkpointing library that aims to minimize the time to write checkpoints by using multiple storage levels combined with data replication and erasure codes [7]. The library attempts to offer a high-level interface to handle datasets and move the data transparently between different storage levels. FTI has shown over an order of magnitude performance improvement over plain writes to the PFS while checkpointing large datasets at high frequency. However, FTI uses a private binary file-format which is not convenient for data analysis, requires restarting with the same parallelization and does not guarantee checkpoint portability to other systems.

The scalable checkpoint/restart library (SCR) [16] is not a full-fledged I/O library as it does not handle the actual writing and reading of the checkpoints that has to be done using another I/O library. Instead, SCR offers an interface to move, replicate and encode checkpoint files written locally on each compute node. As with FTI, SCR offers orders of magnitude faster checkpointing compared with classic I/O libraries writing in the parallel file system (PFS). We note that some I/O libraries can write in local storage but then the user is responsible of keeping data integrity and file availability, which is difficult to guarantee in extreme scale systems. These multilevel checkpoint libraries offer those features transparently for the user.

There are many other HPC I/O libraries, such as for instance Damaris, DDN Infinite Memory Engine, SIONlib... Each one has been created to account for a specific need and provides interesting features in a specialized context. Citing them all would be close to impossible, but still one may intent to give a classification.

Amongst these libraries, some offer a process-local view of files (*e.g* the POSIX or ANSI C libraries) while other support coherent access across process boundaries (*e.g* MPI-IO or parallel HDF5). Some support I/O for most –if not any– purpose (*e.g* POSIX or HDF5) while others are specific to a specific goal (*e.g* FTI or SCR). We also observe that some libraries expose files as a stream of bytes (*e.g* POSIX or ANSI C) while others offer an API where typed data –objects– can be manipulated in the files (*e.g* HDF5 or NetCDF). Finally, some expose a very imperative API where the code explicitly specifies the operation to execute (*e.g* POSIX or HDF5) while others adopt a more declarative approach where many aspects of the actual action to take is not specified in the code (*e.g* FTI). As the actual action is not specified in the application, the separation of concern is increased. Two I/O libraries are using this appraoch to improve the separation of concern: ADIOS and XIOS.

The adaptable I/O system (ADIOS) [1], [14] and the XML I/O Server (XIOS) [15], are two I/O libraries that offer a unified interface to access different I/O strategies (thus improving the separation of concern). ADIOS is an I/O framework dedicated to performance in I/O of scientific code. It is based on two particular observations. First, scientists are not required to know the exact layout of their data as far as they can access it, second, in order to have

the best performance, users should not have to reorganize their I/O directives for each different platform they run their application on. ADIOS provides its own file format and is able to bufferize and/or compress data during I/O operation, to change output file format to one of the supported file format (*e.g.* HDF5). XIOS is an I/O library that offers a unified API for I/Os climate codes. XIOS includes many features such as management of history files and calendar, temporal and spatial post-processing that are often used by the climate scientists. Those features also make sense for other communities. XIOS currently supports only NetCDF4 file format and part of the performance for writing depends on the installation of the dependencies (NetCDF and HDF5).

The API's of both libraries are somewhat declarative which makes it possible to abstract from the choice of the I/O strategy in the domain code and the description of the actual strategy is provided through a dedicated XML configuration file. These libraries increase the separation of concern while providing a simple unified API. Both libraries support a wide range of features such as the dedication of compute nodes to I/O, the use of a single file or multiple files per process.

XIOS and ADIOS internal structures are designed to provide I/O performance and high level functionalities. Thus, they provide additional levels of data abstraction compared with the API native language (C and Fortran). For instance, data and mesh are distinct objects with specific treatment. The libraries internal structures and their data classifications restrict their use and thus, these libraries are mostly designed for post-processing and I/O performance issues in parallel. This limited scope still contains very complex issues that causes the libraries to become complex and to induce some memory overhead due to the use of internal buffers (for performance purpose). This means that porting the library to a new architecture is not always straightforward and it might not be the first choice of usage for very simple I/O strategies (e.g. where POSIX calls are possible) or when even minimal memory overhead matters.

I/O requirements of a given code evolve and may need specific features that are best handled outside of the previous library for the sake of simplicity. For instance, for fault tolerance purpose, one may need to detect missing or corrupted files, and, when appropriate, to recover application state from other checkpoint files that are not corrupted. Specific user instruction would be harder to implement using ADIOS internal plug-in system that is not designed to handle transparently fault tolerance issues. The additional effort will lead to the existence of additional I/O instructions in the application code and I/O concerns will be mixed with domain concerns. The separation of concern would be reduced.

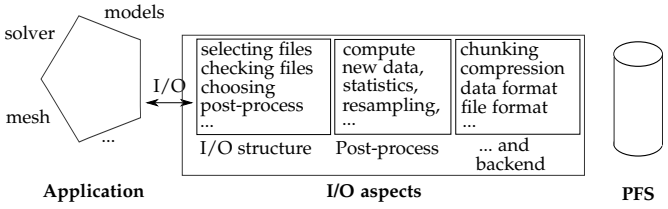

Fig. 4. The various level of abstraction for I/O

Figure 4 presents a schematic of the I/O components of an application code. One may observe that the aspects related to I/O are organized in different layers that are interacting directly with the application code on the left or with the PFS on the right. For the sake of conciseness many I/O features are not shown. Imperative libraries (such as NetCDF, MPI-IO) are located on the right side of this schematic. Declarative library (such as ADIOS and XIOS) internally managed several I/O features and provide a higher level of abstraction. While ADIOS and XIOS are closer to the application-side of the schematic, there is no declarative I/O library that have a general and global approach for any I/O strategy. I/O libraries are designed for a given purpose and the data and libraries structures are adapted to ease the associated tasks.

6 Conclusion

In this work we propose a novel interface, abbreviated PDI, that allows to separate most of the I/O concerns from the application code. PDI transparently manage the I/O aspects that are provided by external libraries or user codes. Our proposal does not impose any limitation on the underlying I/O aspects and decreases the programming effort required to perform and adapt I/O operations for different machines. We demonstrate that, thanks to PDI, scientists can use multiple I/O libraries within the same execution by simply changing a configuration file and without the need of modifying or recompiling the source code of the application. Our evaluation with GYSELA shows that PDI does not impose any significant overhead on the application or on the underlying I/O libraries and I/O strategies. A weak scaling experiments show that PDI behaves well even at large scale. In addition, comparison of five I/O strategies for four representative use cases shows that PDI make it possible to obtain the best performance from several strategies with a very limited development cost and no overhead.

As future work we would like to test PDI with other I/O libraries and in other use cases, such as in-situ visualization and scientific workflows.

7 Acknowledgment

We would first like to deeply thank Guillaume Latu for his help in understanding the GYSELA code and running benchmarks using it. We would also like to stress that this work was supported by the Energy oriented Center of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union. The research leading to these results has also received funding from the European Community Programme [H2020] under the Marie Slodowska-Curie Actions Fellowship DURO (2016-2018), grant agreement No. 708566 and it has been supported in part by the European Union (FEDER funds) under contract TTIN2015-65316-P.

References

[1] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[2] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 32. ACM, 2011.

[3] O. Ben-Kiki, C. Evans, and I. d. Net. *YAML Ain't Markup Language (YAML) Version 1.2, 3rd edition*. No Starch Press, 2009.

[4] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine. Scaling GYSELA code beyond 32K-cores on Blue Gene/Q. *ESAIM: Proceedings*, 43:117–135, Dec. 2013.

[5] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[6] 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*, pages 36–47. ACM, 2011.

[7] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 63–72. IEEE Computer Society, 2010.

[8] V. Grandgirard. High-Q club: Highest scaling codes on JUQUEEN – GYSELA: GYrokinetic SEmi-LAgrangian code for plasma turbulence simulations. online, March 2015.

[9] 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. *Computer Physics Communications*, 207:35–68, 2016.

[10] M. Howison. Tuning HDF5 for Lustre file systems. In *Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10), Heraklion, Crete, Greece, September 24, 2010*, 2012.

[11] G. Latu, J. Bigot, N. Bouzat, J. Gimenez, 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 '16, pages 10:1–10:10, New York, NY, USA, 2016. ACM.

[12] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel NetCDF: A high-performance scientific I/O interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[13] W.-k. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward. An implementation and evaluation of client-side file caching for MPI-IO. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

[14] 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*, pages 15–24. ACM, 2008.

[15] Y. Meurdesoif. XIOS fortran reference guide, February 2016.

[16] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[17] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind. In *2006 IEEE International Symposium on Workload Characterization*, pages 2–2. IEEE, 2006.

[18] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 58–58. IEEE, 2001.

[19] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[20] R. Rew, G. Davis, S. Emmerson, H. Davies, and E. Hartnett. NetCDF User's Guide, 1993.

[21] R. Rew, E. Hartnett, J. Caron, et al. NetCDF-4: software implementing an enhanced data model for the geosciences. In *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*, 2006.

[22] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.

[23] R. Thakur, W. Gropp, and E. Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, 2002.

[24] R. Thakur, E. Lusk, and W. Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical report, Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.

[25] Y. Wang, Y. Su, and G. Agrawal. Supporting a light-weight data management layer over HDF5. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 335–342. IEEE, 2013.