ESAIM: PROCEEDINGS AND SURVEYS, March 2016, Vol. 53, p. 248-270
M. Campos Pinto and F. Charles, Editors

AN APPROACH TO INCREASE RELIABILITY OF HPC SIMULATION, APPLICATION TO THE GYSELA5D CODE*, **

Julien Bigot[1], Guillaume Latu[2], Thomas Cartier-Michaud[2], Virginie Grandgirard[2], Chantal Passeron[2] and Fabien Rozar[3,1,2]

Abstract. Reproducibility of results is a strong requirement in most fields of research for experimental results to be called science. For results obtained through simulation software using high performance computing (HPC) this translates as code quality requirements. While there are many works focusing on software quality, these typically do not take the specificities of HPC scientific simulation software into account. This paper presents an approach to introduce quality procedures in HPC scientific simulation software while remaining the less invasive as possible so as to ease its adoption. The approach relies on quality procedures including human code review and automated testing and offers a dedicated procedure to help correct defects found this way. These procedures are integrated in a development work-flow designed to improve the traceability of defects. By implementing this approach for the development of the Gysela code, we show that it is indeed viable and that the return on investment is positive. We also identify multiple reusable elements developed for this experiment that should reduce the cost of adopting the approach for other codes as well as some aspects that can still be improved to ensure a widespread propagation of the approach in the community.

1. Introduction

It is more and more common to identify simulation as the "third pillar of science" [7] together with theory and experimentation. In the case of experimentation, results reproducibility is a strong requirement. For simulation however, this concept is still not as widespread. Codes tend to evolve at a very fast pace which often leads to the introduction of subtle bugs together with new features. In this context it can be difficult to identify the exact set of features and defects present in the code used to produce a given result.

This can be improved by paying careful attention to the quality traceability of software. In terms of software development, this means identifying and tracking defects (also known as bugs) so as to correct them as soon as possible. These goals ensure that the cost of handling defects remains manageable and are therefore largely shared in the software industry. This industry has thus proposed multiple solutions to handle this problem. These solutions do however not take into account specificities of simulation codes and especially those that arise when using high performance computing (HPC). Some work do especially focus on code quality for HPC scientific simulation software, especially those proposing dedicated programming models and software architectures to

* *We would like to thank the Inria Continuous Integration platform (`https://ci.inria.fr`) team for their support*

** *This work was supported by the ANR funding through the GYPSI and G8-Exascale Nu-FuSE projects.*

[1] Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, 91191 Gif-sur-Yvette, France

[2] CEA, IRFM, F-13108 Saint-Paul-lez-Durance

[3] Inria

improve maintainability. These approaches do however require invasive changes in the code or at least in the habits of their developers which has hindered their adoption.

In this paper we propose a different approach to improve the situation of HPC scientific simulation software reliability. This approach intends at minimizing the impact on the software architecture. We focus on the development work-flow instead and propose a work-flow designed with HPC scientific simulation software in mind. This proposition takes the tracking of defects into account at every step. It integrates software quality assurance processes including code peer reviews and automated software testing. It finally includes an approach designed to ease the correction of defects detected by these procedures. We implement this approach to evaluate it in the framework of the development of the GYSELA [2, 10, 19] code.

This is not the first time these aspects are studied in the field of HPC scientific simulation software. There are indeed workshops entirely dedicated to the question[1] and some work identify lists of best-practices for quality in HPC scientific simulation software [13]. Some communities have also already implemented quality procedures such as the climate modeling community in the form of an automated testing process known as the "trusting process" [4]. This paper does however go further by specifying a complete approach and evaluating it through an application on a real use-case.

The remainder of the paper is organized as follows. Section 2 presents our approach with the development work-flow we propose, the process to detect and track defects and the process to ease their correction. Section 3 evaluates the approach through its real-life application to the GYSELA code. Finally, Section 4 concludes and presents some perspectives.

2. An approach to increase reliability of HPC simulation

There is a rich literature on the various approaches that can be taken to increase the quality of code. This section presents some specificities of HPC scientific simulation software that require these approaches to be adapted. It proposes a dedicated development work-flow to enable the application of quality procedures for this category of codes and focuses on two of its aspects: **a)** processes to identify that a given version of the code suffers from a defect and **b)** processes to localize the actual defect in the code so as to be able to correct it.

2.1. HPC scientific simulation software specificities

Scientific simulation software is typically developed by scientists as a tool to study and validate a proposed theory for the simulated field (physics, chemistry, biology, etc.). As such, the software and its quality is not understood as a goal in itself but rather as a mean to obtain results and as a computerized version of a model, best represented by equations. This is accentuated by the fact that software development is usually not the main competence of the specialists of the simulated domain in charge of development. As results are obtained through simulation or experiments, the model is refined, the equations modified and the code adapted in consequence. The result of an execution of the code is therefore expected to change along its life and can not be predicted in the general case; each execution can provide new, previously unknown results.

Amongst scientific simulation software, there is a sub-category of codes that require a large amount of computation. This leads to a focus on the performance aspect so as to run in a reasonable time; this is known as high-performance computing (HPC). It is not unusual for executions of these codes to represent an amount of computation that would require centuries on a sequential personal computer. They are therefore executed on supercomputers and have to be parallelized in both shared memory (*e.g.* OpenMP) and distributed memory (*e.g.* MPI). Computer scientists are sometimes involved in the development due to this added complexity; however even in this case, the development usually remains lead by the domain specialists as new features can only be introduced by them. This combination of multiple very specialized fields of knowledge in each code means that there is often no single person capable of a complete understanding of the code.

[1] One can for example cite the *SEHPCCSE* (`http://se4science.org/workshops/sehpccse15/`), *SEH4PCS* (`http://se4science.org/workshops/se4hpcs15/`) and *SE4Science* (`http://se4science.org/workshops/se4science16/`) workshops.

The use of supercomputers as an execution platform has other implications because they are expensive and therefore shared between multiple users. This first means that the time allocated for their use by each user is constrained and that a choice has to be made on how much to use for tests intended to improve code quality versus actual result production. This also means that the platform is not under the full control of the developers and that each supercomputer can have its specificities and even a single platform can change due to an update or other change decided by the system administrator.

Another specificity of supercomputers as compared with other execution platforms such as desktop computers is that there remains some diversity in their hardware. This can be observed by looking at the evolution of the Top500 [1] list of the 500 most powerful supercomputers in the world. Three distinct processor architectures (Intel & AMD x86, IBM POWER and Fujitsu SPARC64) are represented amongst the five most powerful computers and this is even amplified by the use of accelerators in the form of GPUs or MICs co-processors. In addition, the history of HPC shows a continuous evolution of the dominant architecture that moved from vector processing, multi-core processing, distributed memory parallelism and now seems to move towards many-core units such as GPUs or MICs. Codes have to be able to efficiently use this large range of hardware and their quality should be validated on all the architectures actually used, increasing the cost of this validation.

On the other hand, HPC scientific simulation software codes present some advantages. Firsts, they are typically run in batch mode requiring not human interaction during their execution thus easing their automated testing. In addition, as scientific simulations, the result of their execution should mimic an experiment in the underlying simulated domain and therefore be reproducible. This should provide a way to validate these results. Finally and contrary to most fields of computing, the users of these codes are usually also developers or at least have a basic understanding of the code, this enables a much better feedback from the users when a defect is discovered.

To summarize, HPC scientific simulation software is a niche of computing with specificities that impact the application of quality procedures that can not simply be reused as-is from other fields but must be adapted. Codes tends to be complex to understand, developed by people for whom computer science is not the main field of competence and executed on very complex and diverse architectures which makes quality procedures all the more important. Their specification evolves together with the code and the platforms used to execute them are a scarce resource making their validation difficult. On the other hand, their execution is non-interactive, their results should be reproducible at some level and their users usually have some understanding of the implementation, which provides opportunities that should be taken into account to ease the validation process.

2.2. **A development Work-flow geared toward HPC scientific simulation software**

A work-flow formalizes the various steps executed during the development of the code and provides some rules regarding their ordering. This is required to implement efficient quality procedures; without knowing which procedure has been applied on a given code version or how much change the code has undergone since the last testing of its quality, one can hardly provide any information. A development work-flow at least has to specify how to identify code versions and their relationships. It can then specify additional information that can be attached to each code version as well as constraints and procedures that have to be applied. Information that can be attached to a code version for example includes its purpose (*e.g.* development or production) or a list of defects affecting the version. Procedures that can be specified for example include tests that might have to be applied to the code before marking it as production-ready.

2.2.1. *Tools*

Although such a work-flow can be implemented without any specific tool, following complex procedures without any support is cumbersome and typically leads to errors. Dedicated tools have thus been designed to handle this task: the version control systems (VCSs) such as CVS or SVN. They enable to store steps in the development of the code (*revisions*), and to associate them with some metadata such as a publication date, author, comment, etc... These VCSs also order revisions so as to to retrieve their development order. This is

however limited to a linear history which fail to capture concurrent developments. The concept of Distributed VCS (DVCS) that handles non-linear history is therefore more and more widespread.

The DVCS concept is implemented in tools such as Git, Mercurial or Bazaar for example[2]. In this approach, the graph of revisions is a directed acyclic graph (DAG). Each revision is typically either a new development based on a single previous revision or the integration of the work from multiple previous revisions (a.k.a. a *merge*). Versions are still ordered since a given revision can not be based on itself even by transitivity, however this is only a partial ordering. While more complex in its concepts, this approach eases tracking multiple development branches by distinct developers, for distinct features or for various maturity levels of the code. It can also better automate[3] the merge of concurrent development from multiple branches than centralized VCSs[4].

There is however at least one aspect of work-flows that is not directly supported by VCSs: tracking the identified defects with the code revisions they impact. There are tools to handle this aspect in the form of dedicated databases known as bug (or more generally "issue") trackers. Well known examples are for instance Bugzilla, Trac or Jira[5]. Issues can be used to track defects, but also more general requirements of the code such as non-functional behavior or missing features. The issue trackers provide a database of issues with metadata associated to each issue such as for example a category, a list of revisions impacted or comments by the users. They also offer an interface to interact with this database, typically in the form of a web interface.

Software forges integrates tools (including issue tacking) around a VCS in the form of a website. Some are provided as a service such as SourceForge or GitHub while others are provided as a software that one can install on its own server such as FusionForge or GitLab[6]. Another classification can be made between forges that are VCS agnostic such as SourceForge or FusionForge and those that are closely linked to a specific VCS, Git in the case of GitHub or GitLab. In our opinion, this second category is much more interesting as it can support a close integration between the various tools of the forge. On both GitHub and GitLab, one can for example directly interact with the issue database from Git and mark an issue as solved in a specific revision on the code by using a dedicated tag in the commit message.

To support the work-flow proposed in this paper, we have made the choice to rely on Git as it seems to be one of the most popular free and open-source DVCSs [3, 22]. In order to be able to retain the control of our data, we have chosen to use a forge that can be installed locally. Amongst the Git dedicated forges provided as installable software, we have chosen GitLab that seems to be a pretty active project with new releases every month and that provide many of the tools we rely on: a web interface for Git and an issue tracker, but also a review tool as will be discussed in Section 2.3 and a wiki that we use for documentation[7].

2.2.2. *Work-flow*

Many work-flows have been designed around Git to formalize the way to use it, such as for example *git-flow* [8], the *GitHub Flow* [12] and the *GitLab Flow* [21]. A common property of all these work-flows is that they rely on the concept of feature branch. A feature branch starts from the shared development branch, focuses on the implementation of a well identified feature only and is destroyed as soon as it is merged back in the shared development branch. The advantage of this approach is that is enables to record intermediate steps of the development of a feature without impacting any shared branch with unfinished work.

---

[2]The page `https://en.wikipedia.org/wiki/List_of_version_control_software` provides a more exhaustive list of both centralized and distributed VCSs

[3]Automated merge typically relies on the `diff` and `patch` tools for text files that consider two changes compatible if they affect far enough lines and require human intervention otherwise. More specific tools with a better understanding of the underlying content can be used and this is a requirement in order to provide any support for automated merge of binary files.

[4]A centralized VCS can automatically merge two branches if they have never modified the same line. A DVCS can do the same but it can also refer to its history to replay choices already made during previous merges, an information centralized VCSs lack.

[5]The page `https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems` provides a more exhaustive list of issue-tracking systems

[6]The page `https://en.wikipedia.org/wiki/Forge_(software)` provides a more exhaustive list of forges in both categories.

[7]This aspects is not covered in this paper.

Another common property of these work-flows is that they identify multiple categories of changes for which different procedures are applied. These categories are typically those described in *semantic versioning* [17]: "backwards-compatible bug fixes", "backwards-compatible new functionalities" and "incompatible API changes". While these categories are well suited to the case of libraries, this is not so true for HPC scientific simulation software where other categories can be identified:

- minor bug-fixes that correct an identified defect with no change of results and little impact on the code,
- optimization and improvements that have no impact on simulation results (but can have major impact on non-functional properties such as performance),
- changes that might have small impact on simulation results due to the way things are computed, floating-point rounding errors and such,
- change on the model such as addition of new physics that can have large impact on results.

Amongst those work-flows, the main specificity of the *GitHub flow* is that it targets simplicity and does not contain a concept of release. Instead, it relies on the concept of continuous delivery where the main development branch is supposed to always be deployable to production and new features are moved to production as often as possible. This is well suited for web services deployed centrally where a single version of the code is used in production at any time. For HPC scientific simulation software on the other hand, once a simulation is started with a given version of the code, it is important to get access to the corrections of defects but not to impact the simulation with new physics. This requires a clear distinction between bug-fixes and new features and a concept of release.

git-flow on the other hand is a featureful work-flow with support for releases based on five different types of branches (*feature*, *develop*, *release*, *hotfix* and *production*). This requires the developers to strictly follow complex procedures and can lead to defects in the code if these are not correctly applied. In the case of HPC scientific simulation software for example, there is no need to distinguish between the *release*, *production* and `hotfix` branches. Finally, the *GitLab flow* offers an interesting discussion on the two previous work-flows and proposes interesting strategies to handle common problems, but since its purpose is to tackle a large range of cases, it does not define a single well formalized work-flow.

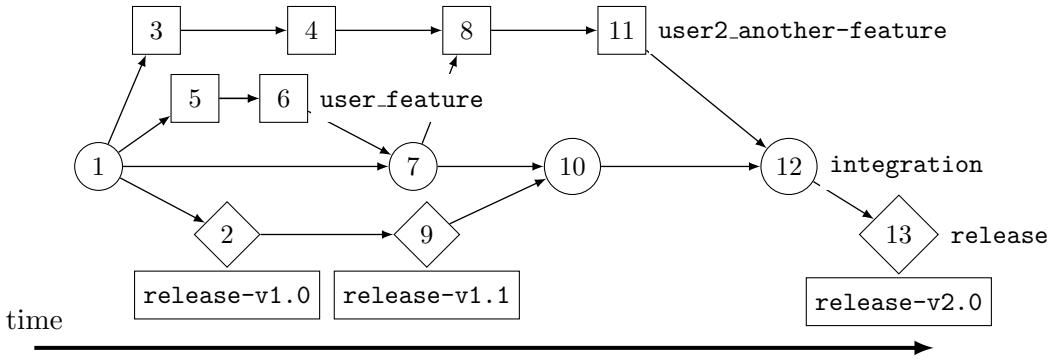

FIGURE 1. The *HPCSSS Gitflow*. Each circle, square or diamond represents a commit and numbers a possible ordering. An arrow from a commit to another means that the first one is a parent of the second. Text in bold represent branches with commits that were made in this branch in the same horizontal line. Rectangles on the bottom of the figure represent tags.

These observations have lead us to propose the *HPCSSS Gitflow* with the specificities of HPC scientific simulation software in mind. It is based on the previously presented work-flows and can be understood either as a simplified *git-flow* or as the *GitHub flow* with support for releases. The work-flow is presented in Figure 1.

It is based on two main branches: the `integration` (○) and `release` (◇) branches. The `release` branch contains the code used for production while the `integration` branch is used to integrate new developments and to test them. Both branches are intended to be always stable.

Development of new features happens in *feature branches* (□) like `user_feature` or `user2_another-feature` started from the `integration` branch. Each developer can work independently of others in its branch but should still merge work done during the development in the `integration` branch as exemplified by commit 8 on the figure. As soon as a feature is complete, it must be merged in the `integration` branch.

When one of the new features that accumulate in the `integration` branch is required in production, a new release is produced by making a commit in the `release` branch and tagging it with an appropriate version number. Limiting the number of such releases reduces the number of different code versions used in production and thus the cost of support. In order to keep the `release` branch readable as a linear history, we have designed a new git command that automatically generates this commit with a message summarizing all the messages of feature merged into the `integration` branch since the last release.

Finally, for minor bug-fixes with no impact on the results, the correction can be made directly on the `release` branch so that users can access this fix without having to include all the new features present in the `integration` branch since the last release (commit 9 in the figure). The resulting version is tagged and the `release` branch merged into `integration` so that the bug-fix is integrated in the next release (commit 10).

This work-flow distinguishes between bug-fixes and other kind of changes but it makes no difference between the three other categories of changes previously identified. Instead, we advocate for this distinction to be handled by a distinct version number the "code version" stored in the code and made of two parts: a major and a minor number. When changes are made on the model simulated, for example by adding new physics, the major number is incremented and the minor reset to zero. When changes are made that only have a minor impact on the results due to the way computations are executed, the minor number is incremented. For changes that have no impact on the results, this version number is kept as-is.

Tracking this independently of the release number has two advantages. First it allows to track changes at a finer grain, for example by identifying the feature whose integration lead to a change of code version. Second, it handles the case where two releases only include changes with no functional impact on the results at all (non-functional aspects such as performance might change): while the release number is increased, the code version remains the same.

By providing a well identified process for the development of code, this work-flow enables when a defect is identified to list a set of code revisions impacted. In turn, this enables to determine whether a given simulation result can be trusted or whether it might result from a code defect. In itself however, the work-flow does not provide any mean to detect whether a code revision suffers from defects or to ease their correction. It does notwithstanding exhibit some development steps that are good candidate to apply quality procedures in a consistent way. These include:

- early in the development of feature branches when the developer wants feedback on the chosen approach,
- just before a feature branch is merged into the `integration` branch to give feedback on the opportunity of the merge,
- regularly on the `integration` branch to ensure its quality,
- just before the `integration` branch is released to give feedback on the opportunity of the release,
- at each and every execution.

It also provides information on code history that can help locating defects so as to ease their correction. These aspects are treated in the two following sections.

2.3. **Defect identification process**

The first difficulty to specify a defect identification process is to clearly define what constitutes a defect. In its narrower definition, a defect or *software fault* is an incorrect piece of code that can lead to a *failure* at execution such as a crash or incorrect results. Additional *non-functional* aspects can however also be included in the definition of a defect. For example in the case of HPC, a piece of code that slows down execution could make it impossible to run on any existing hardware. In this paper we consider any flaw in the code that can lead to unwanted behavior at execution, be it functional or non-functional, to be a defect.

Once the definition of a defect is clear, one must analyze the question of their visibility, that is, if a defect is present, under what condition does it trigger a failure that can be detected? This discussion is required to design test with a good coverage and this is therefore the first aspect presented in the remaining of this section. Then we study the definition of procedures that can be applied to detect defects in a code and their integration in the work-flow and we then focus on the case of regression testing for which we propose a dedicated approach adapted to the specificities of HPC scientific simulation software. Finally, we present some tools that can help implementing the proposed procedures.

2.3.1. *Defect visibility*

Defect visibility can be impacted by multiple elements that can lead a given defect to either stay silent or lead to a failure. We have identified three such elements:

- the underlying hardware and software stack used,
- the algorithms activated in the code,
- to values manipulated by these algorithms.

The hardware and software stack covers all external dependencies of the code including the hardware, compilers and libraries. Various implementations of a specification, be that of a CPU instruction set, language or library API can react differently to uses not covered by the specification. For example, one implementation of MPI might crash when given an invalid size for an array while another could silently ignore that. The algorithms activated in the code cover the case of most HPC scientific simulation software where various modules implementing distinct subparts of the underlying domain model can be activated. If such a module contains a defect, this will not be detected unless the module is activated and some defects might even lie at the interface between multiple modules thus leading to a failure only in the case where a specific combination of modules are activated. Finally,the code can for example contain implementations of operators that do not accept the full range of values they should; a defect that can only be detected if such values are indeed used. An example could be a naive implementation of the binomial coefficients based on the computation of factorials that would exceed the range of integers even for rather small inputs for which the binomial coefficient could be stored without problem. Another example would be a defect lying in the code handling the special case of a zero value for input that would only be triggered when such a value occurs.

These three elements impacting the visibility of defects are typically controlled by parameters specified at three different steps of the testing process. First, the choice of the execution platform used to execute a test determines the hardware and software stack that will be available for combination. Then, at compilation, further choices can be made amongst multiple choices that can be available for the stack, such as in the case where multiple compilers or implementations of a library are available. Some codes also offer compilation parameters used to statically choose between multiple code path, typically in the form of `#ifdef`s. Finally, the last set of choice is made by the configuration provided to the code at execution. This can choose between multiple code path relying on plain dynamic `if`s (or other comparable approaches such as dynamic polymorphism in object-oriented code) rather than `#ifdef`s. This is also where the values that the code will manipulate are specified.

These three axes: execution platform, compilation parameters and execution parameters define the code testing domain. In most cases the domain is large enough that an exhaustive exploration, even for the smallest of tests, would lead to unrealistic testing time. An analysis thus has to be done of the potential impact of each parameter on each test so as to determine the best compromise of domain exploration versus testing time.

2.3.2. *Proposed tests*

Methods to detect whether code suffers from a defect is a research domain in itself, Table 1 for example presents a list of existing procedures with their efficiency as determined by experience in the industry. Looking at these figures, one can observe that the most efficient procedures are unfortunately hardly applicable to HPC scientific simulation software as they typically rely either on a formal definition of the underlying model that changes too fast to be useful in HPC scientific simulation software or on beta testing that does not apply when

Removal Step	Lowest Rate	Modal Rate	Highest Rate
High-volume beta test (> 1000 sites)	65%	75%	90%
Modeling or prototyping	35%	65%	80%
Formal code inspections	45%	60%	70%
Formal design inspections	45%	55%	65%
Personal desk-checking of code	20%	40%	60%
System test	25%	40%	55%
Informal design reviews	25%	35%	40%
Integration test	25%	35%	40%
Low-volume beta test (< 10 sites)	25%	35%	40%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Informal code reviews	20%	25%	35%
Regression test	15%	25%	30%

TABLE 1. Defect detection rates, extract from *Code Complete* [16], adapted from *Programming Productivity* [14], *Software Defect-Removal Efficiency* [15] and *What We Have Learned About Fighting Defects* [20]. Approaches that are difficult to apply to HPC scientific simulation software such as beta testing at large scale or formal verification have been grayed.

the distinction between users and developers is blurry. Even by including those, no single procedure can catch all defects and they should be combined for maximum efficiency as further discussed in [16].

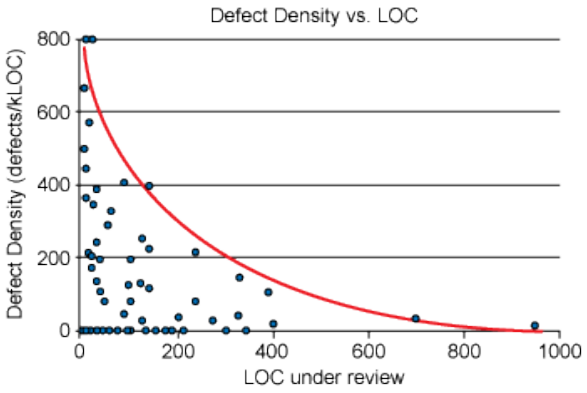

FIGURE 2. As the amount of code under review increases reviewers become less effective at finding defects assuming a constant true number of defects per line of code, extract from *The Best Kept Secrets of Peer Code Review* [6]

The integration of these quality procedures in the work-flow must take into account multiple parameters. First, the sooner a defect is detected, the cheaper it is to correct; it is much easier to correct an invalid design during the early conception phase than once multiple development based on this design have been integrated and released. Then, each procedure has a cost in the form of either developer or computer time and can often not be executed as often as one would like. Finally, some tests loose interest when too much change happened between two of their executions; for example regression tests compare execution results to ensure that no change and thus no regression happened, this can not be used if a new feature has been introduced. The choice of which procedure to use at what step of the work-flow thus depends on the actual procedure.

Personal desk-checking of code, informal design and code reviews are *static tests* that do not require running the code and can not be automated as they rely on a human action. Personal desk-checking of code is an

individual procedure that should be applied as often as possible, ideally at each commit, but that can not be enforced in the work-flow. The reviews on the other hand involve multiple people and can be enforced, but they would be very expensive to apply to the complete code-base. With the hypothesis that a given code version is correct however, one can limit the review to changes to the code since that version. In addition, as shown in Figure 2, the bigger the changes, the less effective the review process is. This pleads in favor of code reviews as frequent as possible to ensure that the amount of change to review remains reasonable. We therefore propose to apply these procedures whenever a feature branch is merged in the `integration` branch.

Other static tests that are not mentioned in the table consist in static code check. This category of tests check that the code complies with pre-defined quality standards and avoid well-known mistakes. There is a large range of tools that implement this kind of tests, especially those based on the CLang compiler front-end[8]. Unfortunately, the HPC scientific simulation software are in large part implemented in Fortran for which the CLang-based tools provide no support; as a matter of fact, there is a scarcity of tools for this language. A static test that can be applied for most language do however exist in the form of compilation: it tests whether the code can indeed be compiled and the list of errors and warnings generated by the compiler. These are usually rather inexpensive and can be run often, for example during the development of a feature branch and at least before integration. All the tests described hereafter are *dynamic tests* that require the execution of some code.

Unit and component tests focus on one specific part of the code and test it independently of the rest of the application. This has multiple advantages. First it typically requires less execution time than running the whole application. Then, while the results of the execution of the whole application is typically not predictable in the general case, it is often build of a set of modules that are much better understood. For example, many scientific applications rely at some level on a linear algebra solver that is well specified. Finally, by testing only parts of the application, it is possible to limit the tests actually executed to those testing parts that have been modified since the last tested code revision. However, this requires the code to be developed in a modular way where each module can be compiled independently and with well defined interfaces so that values can be passed to it from a mock implementation rather than from the actual application; this is often not the case in HPC scientific simulation software which limits the use of these tests. Depending on their cost, they can be run during feature development or at integration only but in any case, they should leverage history information to only test modified parts.

System tests execute the whole application and validate its result. In the case of HPC scientific simulation software this means validating that the result are in compliance with the underlying model of the studied domain. This is a complex process and only specialists of the domain can write such tests. While the more meaningful tests are, the more difficult they usually are to design, some basic tests are often developed early in the life of simulation software, such as conservation law validation. In order to be meaningful, these tests do however require enough time-steps so that a divergence can be observed and distinguished from the noise of rounding errors. Another interesting approach consists in recomputing the time derivative computed in the code to compare them to those provided by the equations [5]. The domain must also be refined enough so that numerical errors are lower than the observed difference. As a result, the computation cost of these test is typically very high and they can only be executed parsimoniously on the integration branch at a fixed interval or even only before a release. These can however sometimes be integrated in normal runs as they might only incur a limited overhead to an otherwise normal execution. Other tests that can be executed this way are non-functional tests such as execution time measurement or memory consumption.

Regression tests compare the results of executions of two versions of the code in which no new feature has been added. They verify that the results of both versions are indeed comparable. Ideally this can be implemented as unit tests but as discussed above, in the case of HPC scientific simulation software software modularity is often low, making this approach difficult. Another solution is to execute the complete code and to compare the results. The execution can be limited to very coarse discretization and a minimal number of iterations. This yields results that typically have no meaning as a simulation of the underlying domain. However, if the results are expected to be exactly the same, at a bit-exact level, from one execution to the other, this is enough for

[8] `http://clang-analyzer.llvm.org/`

comparison. One does not need to wait for any convergence process to happen, either a divergence appears even of a single bit or the result is exactly identical and errors can not accumulate. This makes it possible to run this type of tests much more frequently than realistic system tests, they can typically be executed at least on a feature branch integration or even during a feature branch development. This does however induce two questions: what execution results to use as a reference and how to achieve bit-exact result reproducibility? The reference results should be those of the first version of the code that has the exact same set of feature as the tested version. In our proposed work-flow, this is the first revision of code in the Git history that shares the same "code version" as the tested version. When a new feature is introduced, the code version is upgraded and regression tests can not be used anymore, it is the responsibility of the user to ensure the quality of this change, for example by relying on other types of tests including the more expensive system tests. Achieving bit-exact result reproducibility is more complex and is discussed in the following section.

To summarize, we propose to include five distinct types of quality procedures in our work-flow. Human code reviews should be applied as often as possible at an individual level and at feature branch integration for team reviews. Static tests should be executed at least at branch integration and but can also be applied during their development. Unit tests should be executed at the same steps as static tests but can be limited to tests on the part of the code that did indeed change. Regression tests can also be executed at the same steps, but only in the case no new feature has been introduced, that is if the "code version" number did not change. Finally, the more expensive system tests execution frequency should be chosen to balance their interest with their execution cost and can typically be executed regularly on the integration branch or just before release. Some of these system tests can also be included in the normal code-path if they do not incur too much overhead so as to leverage the time used for production runs.

2.3.3. *Achieving bit-exact reproducibility*

When the underlying model does not change, *i.e.* at feature parity from a code point of view, HPC scientific simulation software executions should provide comparable results given a well defined input. While results are comparable, they are however often not similar at the bit level in practice since there are multiple source of variation of the results. Some codes such as those relying on monte-carlo approaches do for example use random numbers in their implementation that while they should lead to comparable result on a macroscopic level do induce small variations. Another common source of non bit-reproducibility is due to timing issues in parallel codes; distinct conditions at execution can lead to different ordering of execution and potentially difference in results. Finally, a last source of difference is due to rounding errors that can change depending on the optimization applied to floating point operations.

Regarding random numbers, one must note that it is very difficult to generate real numbers with a computer, dedicated hardware has to be used and this is typically a very slow process. In fact codes do usually rely on pseudo-random numbers instead that given the right algorithm can provide well understood distribution and other properties. Pseudo-random number generator algorithms generate numbers from a sequence and use an internal state to determine where in the sequence the next number should be chosen. This internal state can be initialized by what is known as a *seed* to make the behavior of the generator reproducible from one execution to the other. While fixing the seed might not be a good idea in the general case, offering this option enables to provide a version of the code where this aspect is not a source of non-reproducibility.

Timing issue in multi-thread or multi-process applications makes it especially hard to offer a reproducible execution in the general case. If a value can be modified by multiple threads executing in parallel, proving that any correct ordering produces the same result for this value can be particularly complex. Fortunately, both MPI and OpenMP, the typical technical implementations used in HPC scientific simulation software offer synchronous programming models that do not suffer from this problem. In their typical use, the program consist in parallel sections that do only work on memory local to the current thread and pre-defined synchronization points where data is exchanged in a well defined order. There are however ways to use both model that do not provide reproducible execution in practice such as sharing a variable with locking in OpenMP or using one-sided communications or polling to handle messages in their reception order rather than in a pre-defined order in

MPI. There is no general way to make programs using such strategies bit-reproducible, in most cases however the parallelism does not induce non-reproducibility.

The last source of non-reproducibility is due to rounding approximations that change for floating-point number depending on the ordering of operations. Operation reordering are applied for optimization purpose and can mostly be linked to two sources. The first is the compiler and the second are libraries that offer high-level operations such as reductions.

Regarding compiler reordering, it would sound logical that as long as a piece of code is not changed, the binary does not change either thus leading to the exact same computations and results at run-time. Additional parameters do however impact compilation. Optimizations are so interleaved that a change in one part of the code can impact optimizations applied in completely unrelated parts of the code. Even without any change of the code, compilers often have a timeout for expensive optimizations in order not to spend too much time on a single specific optimization but this timeout uses wall-clock time and the amount of actual CPU time can change thus leading to different optimizations. To ensure bitwise reproducibility of results, any optimization impacting floating point reordering must thus be prevented. Fortunately, all compilers we have tested have options to prevent such reordering[9]. These options typically force the order of operations encoded in the binary to match that found in the source code at the cost of a less efficient code.

Reduction operations provided by libraries can also induce operation reordering for performance reasons. This is especially true for parallel libraries that can take timing issue into account and thus leads to the previously identified problem but inside libraries rather than in the user code itself. MPI will for example use a reduce tree that depends on the network and OpenMP might apply the reduction operation in the order threads make the data available. This can be circumvented by implementing reductions by hand instead of relying on optimized ones. In the case of OpenMP, this has to be done in each and every code as OpenMP a compiler extension and that we don't know of any option that would trigger this behavior for well-known compilers. In the case of MPI on the other hand, we have implemented a fully MPI 2.2 compatible implementation of the `reduce` and `allreduce` operations that can either be fully bit-deterministic at the price of degraded performance and scalability or fall back on the underlying MPI implementation. The deterministic `reduce` applies a `gather` followed by a local deterministically ordered reduction. The `allreduce` simply applies this reduce followed by a broadcast.

These small changes enable to compare results at a bit level between two executions at the cost of running a slightly different code than the one used in production and of being slower. A defect located in the part of the code that is modified for it to be deterministic will therefore be oversight by this approach, however these changes should be small and localized enough that this should prove to be exceptional. Regarding performance, the increase in execution time will be more than compensated by the reduction in problem size that can be treated to obtain usable results.

2.3.4. *Tools*

Tools exist that support both human code review and automated testing. Gerrit or Review Board are well known examples of code review tools[10] but there is also one directly provided with GitLab that we have therefore chosen to use. When a branch is proposed for integration in another (a "merge request" in the GitLab terminology), these tools offer a web-page to display the difference between both branches and offer a comment system to make remarks and suggest improvements. The GitLab tool supports both comments that are either global or associated with a specific line of code as presented on Figure 3. Comments of this category are automatically hidden if the code line is later modified so as not to pollute the review page with outdated comments.

Automated tests on the other hand could be implemented with simple tools such as cron and shell scripts but there are platforms specifically designed to integrate them in VCS-based work-flows. These are known as "continuous integration platforms" after the testing approach that lead to their development. They are

[9]for GNU gfortran, we use `-fno-cx-limited-range -frounding-math -fno-reciprocal-math -fno-associative-math -fno-unsafe-math-optimizations -fprotect-parens`, for Intel ifort `-fp-model-source` and for IBM xlf `-qstrict`

[10]The page `https://en.wikipedia.org/wiki/List_of_tools_for_code_review` offers a more exhaustive list.

FIGURE 3. GitLab interface to comment on a specific line of code in a merge request. Lines highlighted in red are those removed by the merge while those in green are added. The stronger highlight is applied when only part of a line changes.

however not limited to integration tests but offer a unified way to describe any kind of test and to interface with multiple execution machines. Two such platforms are for example Jenkins or Buildbot[11]. GitLab also integrates a continuous integration tool known as Gitlab-CI, but in our opinion this is still too young and has not reached feature parity with other tools. In this paper, we base our examples on Jenkins.

In all these tools, tests are always described by two main aspects: a trigger that specifies when to run the test and a description that specifies what to do. Pre-defined triggers usually exist, for example **a)** to interface with VCSs so as to execute the test whenever an commit happens in a given branch, **b)** to execute the test at a specified periodicity (potentially restrained to the case there has been at least one commit since the last test execution), **c)** to execute the test when one or more previous tests have finished execution thus building a complete test work-flow. The description of the test itself can be split in subparts that define:

(1) how to download the data required for the test (source, input parameters, expected results, etc.),
(2) on which execution machine to run the test,
(3) what to execute,
(4) and how to report the result.

Once again, the developer work is simplified by pre-defined tools that typically support download from multiple sources, including VCSs, description of tests in specific formats or as general purpose batch scripts and many ways to report the result of execution, on the platform website, via emails or what might be the most interesting, on the merge request page of GitLab, thus providing direct feedback on the opportunity of its acceptation. The tools also act as batch schedulers by queuing the tests to execute and only issuing one (or more if so specified) test at a time on each execution machine.

2.4. **Defect localization process**

The process previously identified enables one to detect whether a defect has been introduced in the code. In order to correct it, one does however have to identify where exactly the defect is located in the code base. We propose two approaches in that purpose, one relies on the code history to find out the exact code change that lead to the defect, the other relies on the comparison of a valid execution and the invalid one to find out at which point of the execution the results start diverging.

[11]The page `https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software` provides a more exhaustive list of platforms

2.4.1. *Code history analysis*

The use of feature branches for development allows to track the history of each development at a much finer grain than with a linear history. The result is that each commit should be much smaller in term of number of lines of code changed and that identifying the commit that results in a defect can provide a good idea of its location in the code base. In order to automate this process, one first has to design an automatic test that can classify a given version of the code as either impacted or not. In the case where the defect has been detected during the automated testing process, this is already available, otherwise it makes sense to add it as a new test anyway. Git then offers the `bisect` command to automatically walk the code history. It tests the minimal number of code revisions required to find out the exact commit responsible for the introduction of the defect.

This approach does however suffer from a limitation regarding the policy used to make commits in feature branches and that is not covered by our proposed work-flow. On one hand, a strategy that emphasis the commit of stable versions only can lead to very large changes. In that case, while it is possible to identify the commit where the defect has been introduced, the situation is similar to that of a linear history where commits can be large enough that this gives only little information regarding the exact location of the defect in the code base. On the other hand, a strategy where commits are made very frequently often leads to the commit of intermediate code versions that are work in progress and are expected not to generate correct results or to not even compile at all. In that case, the hypothesis that underlies the `bisect` command is broken, there is no single commit that triggers the change from a correct code to an incorrect one in the feature branch, instead the versions can alternate between these two states and this approach can not be used. In order to apply the approach, one has to restrict it to commits of branches expected to remain stable such as the `integration` branch and this once again leads to large changes where finding the origin of the defect can be difficult.

2.4.2. *Execution trace comparison*

In order to circumvent this limitation, we propose another complementary approach to identify the location of a defect. This approach compares the execution of two versions of the code, one where the defect is absent and one where it appeared. Each version of the code is run in the same configuration and a trace of these executions is written, following the most important variables as they are modified. This allows to pinpoint when exactly the results start diverging. The more similar the versions are, the best result the approach gives and it is thus typically used to compare the closest versions possible as identified by the Git `bisect` strategy.

In order to compare the two executions, we have designed a dedicated library called `Cktrace`. The library provides a macro `DBG_CKSUM_PRINT(var)` to instrument the code[12] as demonstrated on Figure 4a. Each call to this macro prints four bits of information to a dedicated file: the source file name and line number, the variable name and a value as shown on Figure 4b. For scalar variables the actual value of the variable is printed, for arrays the size of the array and a FNV-0[13] [9] hash of its content are printed. This makes it possible to follow how variables evolves while not generating an unreasonable amount of data for large arrays.

When comparing two versions of the code where modifications have been applied, the line numbers or file names might change, a refactoring might even change the name of variables, however the values are expected not to change. The order in which the variables appear in the trace is the same as the order in which the `DBG_CKSUM_PRINT(var)` macro is called. For reasonably small changes to the code, this order is typically preserved thus leading to comparable traces.

In order to compare two traces, the following algorithm is applied. First, the values appearing in both traces are compared excluding other informations (line numbers, variable and file name). The first value that differs between executions is likely to be related to the divergence in results. The associated filename, line number and variable name can then be used to see if the divergence is actually a defect or if it was expected. In this later case, this divergence can be ignored and the process is iterated until an unexpected divergence is found.

[12]This is currently limited to Fortran but could be made available for other languages such as C easily

[13] The FNV-0 has is interesting because **a)** it allows to easily spot arrays containing zero only and **b)** it implements the avalanche effect, even the smallest change to a single bit of a variable leads to a clearly observable change in the hash.

```
1  #include "cktrace.inc"
2  program main
3    real, pointer:: field(:,:,:,:)
4    integer:: iter
5
6    call alloc_field(field)
7    DBG_CKSUM_PRINT(field)
8    call init_field(field)
9
10   do iter= 1, MAX_ITER
11     DBG_CKSUM_PRINT(iter)
12     DBG_CKSUM_PRINT(field)
13     call simulate(iter,field)
14   enddo
15 endprogram
```

(A) Usage of `Cktrace` in `main.F90`

```
 === 0000000.trace ===
# main.F90:7
field = 0x0000000000000000:512
# main.F90:11
iter = 1
# main.F90:12
field = 0xB0A352C7491456F1:512
# main.F90:11
iter = 2
# main.F90:12
field = 0xD50AF4623D80D146:512
# main.F90:11
iter = 3
# main.F90:12
field = 0x6C4F1A80543E7800:512
...
```

(B) Trace generated by `Cktrace`

FIGURE 4. Example of use of the `Cktrace` library, both the `field` array and the `iter` scalar are traced. One can follow the value of the `iter` scalar and see that the 512 elements of the `field` array are initialized to zero on line 7.

The current file format generated by `Cktrace` is a plain text file. The comparison process thus relies on a combination of the `grep` and `diff` unix tools. `grep` allows to filter out lines concerning non looked for aspects (such as file name and line numbers in a first time) while diff allows to identify difference between the result of this filtering. A work in progress is to move to a somewhat more structured file format (JSON is foreseen) in order to provide a more integrated difference identification tool.

The limitation of this approach is that the location of a change can only be pinpointed at the granularity of the code executed between two calls to the `Cktrace` instrumentation macro. This means that in order to accurately locate a defect, extensive instrumentation of the code is required. Unfortunately, such an amount of instrumentation can make the code less readable.

In order to be able to provide extensive instrumentation of the code while not impacting its readability, we limit this instrumentation to a dedicated branch. This branch is called `debug` and mirrors the `integration` branch with additional `Cktrace` instrumentation in many places. Keeping this branch up to date requires frequent merges from the `integration` branch, however these are typically trivial merges. This can thus be automated through Jenkins. Manual intervention is only required in the rare cases where a conflict occurs.

When a regression is identified in a feature branch, the `debug` branch in merged with this feature branch to generate a new `feature-debug` branch. Since the `debug` branch is very close to the `integration` branch on which the feature branch is based, this typically generates only very limited conflicts. The two branches `debug` and `feature-debug` then both contain the `Cktrace` instrumentation and can be used to locate the defect. Once this is done, one can go back to the original feature branch to correct it.

3. Applying the proposed approach to the Gysela code

We have applied the approach presented in the previous sections to the Gysela code. This section presents this experiment and offers feedback on this experience. We first describe the code and the platform we have set-up and we characterize the testing domain. Then we present the four categories of tests implemented: static tests, bitwise comparison, unit tests and full system tests. Finally, we discuss this experiment, the gains and costs of the approach.

3.1. The Gysela code

Gysela is a GYrokinetic, SEmi-LAgrangian code for the simulation of turbulence developing in Tokamak plasma. At its core is the ion distribution function that describes the quantity of particles in the simulation space. The main algorithm employed in the code is a self-consistent coupling between advections and quasi-neutrality. The *advection* solver (based on the Vlasov/Boltzmann equation) computes the movement of the loaded particles from the electromagnetic fields. The *Quasi-neutrality* solver (based on the Poisson equation) computes the electromagnetic fields from the distribution of particles. The Quasi-neutrality solver works in the 3 dimensions of space. An advection solver based on the kinetic setting would work in 6 dimensions, 3 dimensions of space and 3 dimensions of velocity. Gysela however relies on the gyrokinetic approach that reduces this dimensionality and thus the amount of memory and computation required. The result is an advection solver that works in "only" 5 dimensions. This requires the addition of a gyroaverage operator in the coupling between the advection and quasi-neutrality solver.

The code base is not overwhelmingly large with about 60 k lines of Fortran90 and 13 k lines of C code. As for most 5D gyrokinetic codes however, the amount of computing power required for runs of Gysela leads to the usage of the largest existing computers. Runs of Gysela have lasted up to a full month on a complete supercomputer (Jade at CINES, France) and runs on the full Juqueen (Jülich, Germany) machine have demonstrated a good scalability with a 91% relative efficiency at 1 835 008 threads compared to 65 536. To achieve such parallel efficiency, advanced numerical schemes are used and the parallelization (using MPI and OpenMP) represents a large part of the code-base, tightly coupled with the physics related code. This makes the resulting code difficult to write or even understand for both physicists, applied mathematicians and computer scientists.

As many research codes, Gysela is used as a testbed for new physics and numerical schemes. Multiple alternative or optional algorithms have been integrated and accumulated in the code over its more than 10 years of history. Ensuring that all the combinations remain functional becomes a challenge. This is even more true since new algorithms and parallelization approaches are developed everyday, making the code a fast moving target. There have been more than 174 k line modifications distributed over more than 1700 commits (excluding merge) in the core repository over the last year. During the same time, 33 release have been produced: 16 feature releases and 17 bugfix releases.

As most simulation codes, Gysela started as a single person project and slowly grew, there are now about 5 daily developers and in the order of 10 users that also sometimes modify the code. The daily developers are typically applied mathematicians and computer scientists spread over multiple locations and laboratories in France. The users are mostly physicists and applied mathematicians that run the code, study it, test local modifications and propose new features, usually reworked by the daily developers before inclusion in releases.

3.2. Platform setup

The platform we use to implement the described approach consists in an installation of GitLab hosted by *Maison de la Simulation*[14] and in an installation of Jenkins provided by *CI-Inria*[15]. We use GitLab to store our issue database and for merge requests where we apply a human code review process as described in Section 2.3. Together with Jenkins, *CI-Inria* offers virtual machines (VMs) used as execution machines. The *CI-Inria* platform provides each project with its own independent Jenkins instance with full administrative rights which enables to use distinct execution machines with specific credentials for each project. We have therefore been able to also use the *Poincare* cluster of *Maison de la Simulation* (IDRIS, Orsay, France) as an additional execution machine and to connect Jenkins to our GitLab so that tests are launched automatically whenever a merge request is made on GitLab. The results of tests execution is also made accessible as a comment on the merge request thus making GitLab the main entry point for developers.

[14] `https://gitlab.maisondelasimulation.fr/`
[15] `https://ci.inria.fr/`

The provided VMs can be configured with either one or two cores running at 1 or 2 GHz each and with 1, 2 or 4 GB of RAM. They are also completely administered by members of the project (root access). We have chosen to use the largest possible configuration (2×2 GHz + 4 GB RAM) which in the context of HPC remains small. The OS we use is Debian GNU/Linux that provides all the software dependencies of GYSELA (GCC and GFortran compilers, OpenMPI and HDF5). This does however not match production environments where software specifically tailored for the hardware is usually available (especially for the compiler and MPI library). Multiple VMs with distinct software can be created to test a larger range of case, however since none of them would really match production configurations we did not explore this approach. These VMs have not been designed to be interconnected through the network and we do not use multiple VMs together for the execution of parallel MPI jobs. We have created 4 VMs however, but only in order to accelerate the testing process by running distinct tests on multiple VMs in parallel. *Poincare* is a cluster of 92 dual E5-2670 Sandy Bridge nodes (2×8 cores at 2.6 GHz) with 32 GB RAM each, connected by a QLogic QDR Infiniband interconnect. The software environment is managed through the `module` command that provides the Intel compiler, intelmpi and HDF5 thus satisfying all GYSELA dependencies. This is much closer to production environments than the VMs. In order to take into account the shared nature of *Poincare* and disturb other users the least possible, we currently limit our use of this platform to 4 tests in parallel and only submit parallel tests during the night.

The use of this machine does however raise a technical issue since it is accessed through a batch scheduler (LoadLeveler) to share it between its users. Batch schedulers offer an asynchronous interface where a job is submitted by the user, then waits for execution resources to become available and only then is executed. Jenkins like other CI tools implements a similar mechanism and each Jenkins job is expected to be synchronous. When interfacing Jenkins with a batch scheduler like LoadLeveler, the job waits in the Jenkins queue until Jenkins executes it which results in the submission of a LoadLeveler job. Once this is done, Jenkins expects to be able to check the result of the execution, however the job might still be waiting in the LoadLeveler queue and the results might thus not be available yet. We have worked around this problem by using an option of LoadLeveler[16] that waits for the end of the job execution before returning control to the user, thus offering a synchronous interface. This solution does however require a process to run, doing nothing during the whole execution of the LoadLeveler job which is fragile if the scheduler is restarted and leads to a waste of resources if a job is submitted from another job as happens for GYSELA restarts. A better solution would require to integrate the Jenkins and LoadLeveler queues by allowing to submit jobs in both directions.

It would also be very interesting to use the exact same supercomputers as those used in production in addition of these two platforms (VMs and *Poincare*). This does however raise a security issue as all members of a given project have access to the credentials used to connect to the execution machines and are all able to submit jobs using this single account. This is considered as sharing an account amongst multiple users and goes against the security policy of all major computing centers. We are currently investigating possible solutions in collaboration with computing centers technical and security teams. One possible direction would be to change the automated execution platform from Jenkins to Buildbot that has a different approach to the way execution slaves are spawned on the execution machines. While Jenkins connects (typically through ssh) to the execution machine to launch the slaves, in Buildbot the slaves are manually launched on each execution machine and thus require no central storage of any shared credential.

To summarize, the platform we have set-up fulfills the needs of the approach presented in this paper. Setting it up has however required a rather large investment in time. The Jenkins software is sometimes complex to use, especially when it comes to its administration for installing plugins or connecting with other tools such as GitLab. While connecting to the VMs is well integrated in the *CI-Inria* platforms, using other execution machines also requires to figure out tricky technical details, integration with the batch scheduler of shared machines is still work in progress. Overall the installation of GitLab from scratch and the configuration of both Jenkins and the execution machines to support all tests described here required an investment of time that can be estimated to around one month of full-time work for a single engineer. This was however not done at a single time but as an incremental work where each new test required additional work, the initial investment

[16] `llsubmit -s` specifies that the llsubmit command will not exit until all steps of the job have completed.

to get a simple merge request procedures with compilation work only required a few days. We haven't even found a solution to use real supercomputers yet. We do however believe that most of our work is reusable. In order for the approach to be adopted by a large part of the HPC community, a pre-configured platform that simplifies these technical aspects should be provided. This is something we are trying to achieve together with the national computing centers.

3.3. **The Gysela testing domain**

Before describing the tests implemented for Gysela, let us characterize its testing domain as described in Section 2.3.1. While there are typically about ten different execution platforms (excluding personal machines) used to run Gysela in production at any time, the number of platforms available for testing is limited to two: the *CI-Inria* VMs and *Poincare*.

At compilation, the compiler, its options and libraries versions are selected and choices are made amongst the various alternative algorithms. We have chosen to limit ourselves to a single version of compiler and libraries on each execution platform. The Gysela CMake based build-system supports three compiler configurations: one that matches that of production, one for deterministic runs and the last one for debugging. It also exposes 12 pre-processor macros that enable or disable a code path resulting in 2^{12} distinct configurations. In addition, a legacy plain Makefile buildsystem remains available that offers four compilation configurations. This results in a plain total of $3 \times (2^{12}) + 4 = 12\,292$ possible compilations.

At execution, a Fortran namelist is specified that choses between some alternative implementations and specifies the value for some constants. A few data fields can also be initialized from HDF5 files referenced from the namelist. There are about 200 input fields in the namelist, including booleans, integers, floating point, character strings and small arrays of these (typically, their size is the number of species simulated). This results in an extremely large number of potential configurations.

3.4. **Static tests**

Since static analysis tools are limited for Fortran, we have developed small scripts to test two simple aspects of the code. First, a naive dead-code detection is made at the procedure level by identifying procedures defined but never called. Second, a scripts checks that the coding rules of the project concerning things such as case consistency between declaration and use or indentation are correctly applied. Each of these scripts returns a status whether a problem has been found in the code and in such a case, a list of the problems found. Since the code is not even compiled to apply these tests, they do not depend on any of the previously identified parameters: execution platform, compilation options or execution parameters. Each test only takes a few seconds to run.

Another static test applied is a compilation test that returns a status whether the code can be fully compiled and the list of errors and warnings generated by the compiler. As a static test, this is not affected by the execution parameters but the hardware and software stack and compilation options used impact the test. This test is rather inexpensive, taking between 2 and 3 minutes for a compilation depending on the parameters. We execute it on both *Poincare* and the VMs with each of the three set of compiler and linker options. The combinatorial of testing all combinations of pre-processor macros would however require multiple days or even weeks. Fortunately, the successful compilation of each code path is nearly unrelated to the activation status of the others and they can be tested independently. We therefore test 13 cases, one where each of the 12 macro is activated and one with none active. In additions, we test the 4 compilation presets of the plain Makefile-based compilation process on the 2 stacks. This results in $2 \times (3 \times 13) + 4 = 82$ distinct compilation tests. With 41 tests by execution machine taking about 2.5 minutes each and 4 parallel execution on each machine, the full process usually takes slightly less than 30 minutes to complete.

These tests are fast enough to execute them frequently and they are run for every merge request as well as for every commit in the `integration` branch.

3.5. **Bitwise comparison tests**

Gysela is a code well suited to be made bit-reproducible. Pseudo randomly generated values are used but only at initialization and the generator is always set-up with a fixed seed. There is no timing issue between MPI processes or OpenMP threads because we use a pure synchronous model. We use hand-made deterministic OpenMP reductions for portability since historically some OpenMP reduction implementations were unreliable. For MPI, we rely on our deterministic wrapper described in Section 2.3.3 in the deterministic compilation mode. We also use the compiler options identified in this section to obtain a deterministic rounding model.

As dynamic tests, the bitwise comparison testing domain includes the execution platform, the compilation parameters and the execution parameters. The compilation parameter space is slightly reduced however as the code must be compiled in deterministic mode for the tests to be possible. This still leaves an extremely large range of parameters that can not be exhaustively explored and choices have to be made regarding the configurations to test. The process we have adopted is to design a small basic test and derive new tests from it whenever a defect is discovered that was not covered by the existing tests.

The base test uses the default compilation options and has been designed to run on all execution machines including the VMs. Since the VMs only have two cores, this is the maximum number of threads that can be used, but they can either be used as two OpenMP threads inside a single MPI process or inside distinct MPI processes. We have chosen this second option which in in Gysela has more impact on the executed code path than the number of OpenMP threads. The bitwise comparison approach enables to used a very small domain size, we use 1 M points ($N_r = 32, N_\theta = 64, N_\varphi = 16, N_{v_\parallel} = 16, N_\mu = 2$). Similarly, the number of iterations can be reduced to a minimum as long as all parts of the code are executed at least once such as the computation and output to disk of derived values in Gysela, known as "diagnostics". We use 2 iterations and have set-up the configuration so that diagnostic are executed every two iterations.

Additional tests have been designed to catch defects that this base test did not detect. These tests typically differ from the base by changing the value of a parameter in the namelist used to choose between alternative algorithm in the code. This type of change does indeed have a much larger impact on code coverage than the values used to initialize the data manipulated by the code. The value of parameters in the namelist are also changed much more frequently in production than those provided as pre-processor macros. As a result of this process, the parameters explored in the tests are: the number of species (2 values), the quasi-neutrality implementation (4 values), the collision operator (4 values), the low frequency filter (3 values) and a parallelization choice (2 values). When possible, multiple parameters have been combined to reduce the number of runs executed and there are currently 13 different tests run on each platform.

These tests are defined as shell scripts that all follow the same pattern: execution of the code, fetching of reference results and comparison of the execution results with reference. This script is stored together with the various namelist to test in the same repository as the code. Executing the code requires interfacing with batch schedulers but this aspect was already handled by a script provided for the use of Gysela by humans in production, the only change that was required was the support of a synchronous mode as described in Section 3.2. This leaves two technical questions open: the approach used for comparing results and the storage of the reference data.

We have chosen to use the HDF5 checkpoint files generated at the end of Gysela runs for comparison since by definition they contain the whole state of the simulation. In addition to small scalar describing the iteration number and such, the most important field stored in the Gysela checkpoint files is the ion distribution function. HDF5 provides the `h5diff` command that compares arrays inside HDF5 files at the precision specified as parameter. Comparing the ion distribution function with the `h5diff` command at bit-exact precision thus offers a good comparison tool.

The reference results are binary files that would prove to be a problem if stored in the code repository since its technology (Git) is optimized for text files. In addition, each reference is specific to a given execution platform and can be recomputed by re-executing the version of the code that was used to generate it in the first place. As a result, we have chosen to only store the reference for the current state of the `integration` branch in a specific

directory of each execution machine. When a developer changes the code version number in the repository, thus invalidating the reference results, he is responsible for recomputing the reference results with the new code.

Each execution lasts about 20 seconds on the VMs and about 30 seconds on *Poincare* due to the overhead of the batch scheduler. The compilation takes about 2 minutes but is factorized to be executed only once for all executions. With 13 distinct tests launched on each execution architecture with a four-way parallelism each, this means that the whole set of tests requires less than 4 minutes to execute. The size of all reference results currently amount to about 210 MB for each platform which leaves room to add new tests if required. The tests are run for each merge request on the VMs, but only at night if there was an actual commit in the `integration` branch on *Poincare* since they rely on the compute nodes that we only use at night for tests.

3.6. **Unit tests**

Like many HPC scientific simulation software, Gysela has been designed in a very monolithic way and extracting unit tests is often difficult. The unit tests available are therefore limited to tests of: **1)** the deterministic MPI reduction implementation, **2)** the gyroaverage operator implementation and **3)** the checkpoint-restart mechanism.

The deterministic MPI reduction implementation has been described is Section 2.3.3 and reproduces the pre-defined MPI interface. It is therefore as independent of Gysela as MPI and can be unit tested. The test compares the results of reductions using the deterministic implementation with those of the underlying implementation and returns an error in case the difference between the two crosses a threshold. It tries to use a large range of features to validate the API as much as possible, including using the `INPLACE` parameter, various kind of Fortran arrays (normal, allocatable, pointers) and different lower and upper bounds. While this test is very limited, the number of lines of code required to implement it is surprisingly large in comparison. This is due to the fact that no dedicated unit test library is used and that all the unit testing logic is implemented directly in the code. The unit testing libraries for Fortran we have tried did not help much in that regard; they typically try to mimic the design of unit testing libraries of object-oriented languages and due to the lack of features such as reflexivity and limited use of object in Fortran this fails. This test requires less than one second to execute.

Another test focuses on the implementation of the gyroaverage operator. Roughly speaking, this operator consists in computing an average of a given physical quantity over a circle (the so-called *Larmor* circle) in a $2D$ plane. Several implementations of this operator are available in Gysela. They can be distinguished in two categories: the *Padé* approximation and direct methods that perform integration on the gyro-ring using interpolation methods (Hermite or cubic splines). Recent development focuses on the optimization of the Hermite interpolation based gyroaverage. To ensure the validity of the computation done by the different optimized versions, a unit test was required. This test is based on the knowledge of an analytical formula of the gyroaverage for a particular family of function $f_{m,n}$. The test consists to initialized the input $2D$ plane with a chosen $f_{m,n}$ function, compute its gyroaverage, and finally compute the distance between the numerically gyroaveraged plane and its analytical solution thanks to a L_1 and L_2 norm. The development process starts with a reference implementation of the gyroaverage based on Hermite interpolation. So every optimized implementations must reproduce the same results of the reference implementation on the unit test. Between 15 and 20 seconds are required to test the 12 gyroaverage methods available in Gysela including compilation.

The last test is not strictly speaking a unit test because is executes the complete code due to the lack of modularity of Gysela. It does however specifically validate the checkpoint-restart mechanism of the code. The approach we have implemented consist in comparing the results of two executions of the code with similar parameters, except one stops after a checkpoint in the middle and restarts for the second half of the run while the other executes the complete run without any restart. The comparison of the two executions uses the same approach as for bitwise comparison. The namelist file used is the same as that used for bitwise comparison tests except with a restart inserted. The time required for their execution is therefore very close at about 2 minutes.

Unlike the bitwise comparison tests, this one does not require any data to be stored since the results of two executions are compared.

Since they do not test the actual code, the domain of unit tests is not impacted by GYSELA compilation and execution options. Only the execution platform impacts their result. They are also very fast to execute and are therefore integrated in the tests executed for any merge request. Since their number is very small currently and their execution very fast, we have not implemented any tool to limit the execution of unit tests to the parts of the code that did actually change.

3.7. **System tests**

A fist system test has been implemented that relies on the fact that GYSELA uses a semi-Lagragian numerical scheme to solve the gyrokinetic Vlasov equation. Under the assumption of a fine discretization in both time and space, it exhibits good properties of energy conservation the two regimes of plasma simulation: linear and non-linear[17]. In this context, the kinetic and potential energies are macroscopic variables that can be well measured. The dynamics of these two energies is sensitive to any modification of the distribution function, which is the main unknown of the simulation. They characterize the plasma dynamics state quite well. To test both linear and non-linear regimes in realistic geometries however, no analytical case can be used, comparison with a reference case is therefore a good solution. A divergence typically reflects a defect in one component of the application such as the Vlasov solver, the initialization stage or the Poisson solver. This test is however known to have limitations with respect to the defects it can detect and will typically not handle errors happening at radial boundary conditions, errors located in initialization routines or sections of the code that are not used in these simulations.

The test we have designed therefore measures the kinetic and potential energies during a simulation. Unlike for bitwise tests, elements such as the values of data manipulated by algorithms, dimension of the mesh and number of iterations have a large impact on the quality of this test. Some defects can for example yield large errors in the non-linear case that would be completely hidden in linear case. Five distinct namelist files have therefore been designed so as to explore interesting state of the simulation, they all use a mesh ($N_r = 64, N_\theta = 128, N_\varphi = 32, n_{v\|} = 48, N_\mu = 2$) during 130 iterations. The potential and kinetic energy at each time-step are stored in an ASCII file whose total size is in the order of 20 KB. A script compares these values with a reference and a difference larger than a few percents of the maximum absolute value indicates that the simulations diverges. Since the reference file is small and not expected to change, we consider it part of the test itself and store it with it in the code repository. This test requires about 12 hours of sequential execution for each input file and more memory than the VMs contain. It is thus only executed on *Poincare* where it is parallelized over 64 cores and ends up lasting 11 minutes for each of the five executions or a bit less than 15 minutes total with the four-way parallelism. The test is only executed at night when there was a commit to the `integration` branch.

In addition, we use the MTM library [18] in the code that traces memory allocation and thus detects any memory leak. Another tool is used to take performance measurements of the code at the module level. These two tools are activated for all runs, production or tests and their result is available as part of the test results. The execution time in particular is automatically traced by Jenkins over code revisions of the `integration` branch thus making it possible to identify the change that lead to a performance drop when this happens.

Two additional tests that rely on an even deeper understanding of the physical model are currently being designed [11]. The first one is known as the Rosenbluth-Hinton test and consists in computing the linear evolution of a zonal flow component for an initial electrostatic perturbation. The second one is a linear benchmark based on Cyclone DIII-D case for global codes. The first problem with both these tests is that due to the 5D nature of the code, their execution is very costly. Finding the execution parameters that reduce this time to a minimum is indeed a complex work. For the Rosenbluth-Hinton test, we managed to reach a situation where the test can be executed in about 30 minutes on 64 cores of *Poincare*. Execution of the Cyclone test on the other hand, still requires 1.7 Mh.core on a Blue Gene/Q architecture or about 10% of the yearly allocation for GYSELA. It makes

[17] simplified geometry, no sources, no collisions are assumed.

its execution as a regular automated test impossible an we are still investigating possible change of parameters to make the test less expensive. Another problem with these tests is the complexity of validating their result. The development of a script validating the results of the Rosenbluth-Hinton is in progress, however this requires applying various post-processing steps and requires much more work than a simple value comparison.

3.8. **Analysis**

Before this process was adopted, each developer had to test its code himself. With access to different machines, he only tested a small subset of the possible configurations. The two main tests that were performed were a compilation test and a result comparison test with the last code revision using a threshold he chose. Each developer had to determine what was acceptable in term of compiler warnings, or result similarity. As a result, it was difficult to identify a single version of the code that even compiled with all possible combination of compilation options and was usable for all users.

The new approach we have implemented has a cost in term of both the set-up of the platform and the development of the tests. This however typically relies on the same work that is required to correct defects when they appear. The additional work required to automate the test is very light in regard of the test design. Learning the new work-flow and implementing code reviews also takes time for developers. While we have no hard number about the defects that have been prevented from appearing, the feeling of developers is that time spent on quality was reduced. They consider that the time gained by having to spend less time debugging the code largely overweights the time spent implementing the work-flow.

Regarding compilation, the only compilation problem that still sometimes appear are related to platforms not tested by the process such as the Blue Gene/Q architecture. The number of tests grew fast at first to catch new identified defects but stabilized in the last 10 month because fewer and fewer defect remain uncaught. Determining how much remains to be done could be determined by a formal analysis of the test coverage with a tool such as gcov[18] but we have not done so yet.

In addition, the new work-flow greatly increases the confidence in new development by beginner or new developers and can therefore augment the amount of new features accepted in the code. What might be the most important improvement however is the increase in confidence of results and the clear identification of those impacted by the defects that are found. This improves the quality of physics publications based on Gysela and reduces the risk of having to retract a paper.

4. Conclusion and future work

This paper has presented an approach to increase the confidence in simulation results by applying quality assurance procedures to HPC scientific simulation software. Classical approaches need to be adapted to take into account the many specificities of this field regarding the hardware used, the code itself and its developers. The approach we propose is organized around a work-flow and introduces quality procedures based on both human review and automated testing as well as a dedicated process to ease debugging. We have applied this approach to the Gysela code and have noted a large improvement in the code quality since we started using it (less defects that reach production and faster correction of those that do). The tests we have implemented are static tests, bitwise comparison tests, unit tests as well as complete system tests.

Applying this approach requires a non negligible investment that might prevent other codes from doing it. The return on investment is however very positive in multiple aspects. The amount of computation time lost in production due to an error is greatly reduced. Developers can focus on implementing new features rather than correcting defects. The confidence in the results is increased and physicists spend less time figuring out if a given result is an error or is actually physically relevant. There are two directions we are currently exploring to extend this work.

The first direction is to extend the quality and quantity of tests executed in the Gysela automated testing procedure. We for example plan to add a code coverage analysis to asses the amount of code actually tested.

[18] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

The more interesting tests we now target do however rely on more invasive approaches. Unit tests are designed for as many modules as possible in the code; this requires that the code be made much more modular. Since GYSELA is a complex piece of code this takes time, but this is an aspect we are working on for some time now and the infrastructure is ready for new code to be added in a more decoupled way similarly to what has been done for the new implementations of the gyroaverage operator. We are also investigating the use of dedicated programming models that should make this modularization easier such as software components and task based programming.

The second direction we are exploring is to provide reusable elements everywhere it makes sense to ease the adoption of the approach by new codes. Some part of the approach are specific to each code that use it such as for example all tests that rely on a deep understanding of the underlying simulated model. Other parts on the other hand can be reused from one code to the other. This is the case for multiple elements we intend to make available to the community, including the `Cktrace` library, the deterministic MPI reductions or the deterministic compilation framework. In addition, we are in the process of making an automated testing platform available in collaboration with the computing centers so as to handle the technical aspects once for all code and to enable the use of the same supercomputers as those used in production for testing.

REFERENCES

[1] Top500 supercomputer site. `http://www.top500.org`. Accessed: 2015-12-01.

[2] Julien Bigot, Virginie Grandgirard, Guillaume Latu, Chantal Passeron, Fabien Rozar, and Olivier Thomine. Scaling gysela code beyond 32k-cores on Blue Gene/Q. In *ESAIM: PROCEEDINGS*, volume CEMRACS 2012 of *43*, pages 117–135, Luminy, France, July 2012.

[3] Black Duck Software, Inc. Compare repositories - Open HUB. `https://www.openhub.net/repositories/compare`, 2015. Accessed: 2015-02-05.

[4] P. Brockmann and A. Caubel. Trusting dashboard. `http://webservices.ipsl.jussieu.fr/trusting/`. Accessed: 2015-01-26.

[5] T Cartier-Michaud, P Ghendrih, Y Sarazin, J Abiteboul, H Bufferand, G Dif-Pradalier, X Garbet, V Grandgirard, G Latu, C Norscini, C Passeron, and P Tamain. PoPe (Projection on Proper elements) for code control: verification, numerical convergence and reduced models. Application to plasma turbulence simulations. working paper or preprint, October 2015.

[6] Jason Cohen. *The Best Kept Secrets of Peer Code Review*, chapter Code Review at Cisco Systems: The largest case study ever done on lightweight code review process; data and lessons., pages 63–87. Smart Bear Inc., 2006.

[7] President's Information Technology Advisory Committee. Computational science: Ensuring america's competitiveness. Report to the President, June 2005. `https://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf`.

[8] Vincent Driessen. A successful git branching model. `http://nvie.com/posts/a-successful-git-branching-model/`, Jan 2010. Accessed: 2015-01-25.

[9] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, and Donald Eastlake. The FNV Non-Cryptographic Hash Algorithm. Internet-Draft (work in progress) 1654, Internet Engineering Task Force, April 2014.

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

[11] Virginie Grandgirard, Jérémie Abiteboul, Julien Bigot, Thomas Cartier-Michaud, Nicolas Crouseilles, Charles Erhlacher, Damien Esteve, Guilhem Dif-Pradalier, Xavier Garbet, Philippe Ghendrih, Guillaume Latu, Michel Mehrenberger, Claudia Norscini, Chantal Passeron, Fabien Rozar, Yanick Sarazin, Antoine Strugarek, Eric Sonnendrücker, and David Zarzoso. A 5D gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations. working paper or preprint, July 2015.

[12] GitHub Guides. Understanding the github flow. `https://guides.github.com/introduction/flow/index.html`, Dec 2013. Accessed: 2015-01-25.

[13] Michael A. Heroux and James M. Willenbring. Barely sufficient software engineering: 10 practices to improve your cse software. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, pages 15–21, Washington, DC, USA, 2009. IEEE Computer Society.

[14] C. Jones. *Programming Productivity*. McGraw-Hill, 1986.

[15] C. Jones. Software defect-removal efficiency. *Computer*, 29(4):94–95, Apr 1996.

[16] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[17] Tom Preston-Werner. Semantic versioning 2.0.0. `http://semver.org/`, Jun 2013. Accessed: 2015-02-05.

[18] Fabien Rozar, Guillaume Latu, Jean Roman, and Virginie Grandgirard. Toward memory scalability of GYSELA code for extreme scale computers. *Concurrency and Computation: Practice and Experience*, 27(4):994–1009, November 2014.

[19] Y. Sarazin, V. Grandgirard, J. Abiteboul, S. Allfrey, X. Garbet, Ph. Ghendrih, G. Latu, A. Strugarek, and G. Dif-Pradalier. Large scale dynamics in flux driven gyrokinetic turbulence. *Nuclear Fusion*, 50(5):054004, 2010.

[20] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *VIII International Symposium on Software Metrics (METRICS'02)*, pages 249–258, Washigton, DC, USA, June 2002. IEEE Computer Society.
[21] Sytse Sijbrandij. Gitlab flow. `https://about.gitlab.com/2014/09/29/gitlab-flow/`, Sep 2014. Accessed: 2015-01-25.
[22] Ian Skerrett. Eclipse community survey 2014 results. `https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/`, Jun 2014. Accessed: 2015-02-05.