Generating and Scaling a Multi-Language Test-Suite for MPI

Julien Adam, Jean-Baptiste Besnard, Paul Canat, Sameer Shende, Hugo Taboada, Adrien Roussel, Marc Pérache, Julien Jaeger

Generating and Scaling a Multi-Language Test-Suite for MPI

Julien Adam
adamj@paratools.com
ParaTools SAS
Bruyères-le-Châtel, France

Jean-Baptiste Besnard
jbbesnard@paratools.fr
ParaTools SAS
Bruyères-le-Châtel, France

Paul Canat
pcanat@paratools.fr
ParaTools SAS
Bruyères-le-Châtel, France

Hugo Taboada
hugo.taboada@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France
Université Paris-Saclay, CEA
Laboratoire en Informatique Haute
Performance pour le Calcul
et la simulation
91680, Bruyères le Châtel, France

Sameer Shende
sameer@paratools.com
ParaTools SAS
Bruyères-le-Châtel, France

Adrien Roussel
adrien.roussel@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France
Université Paris-Saclay, CEA
Laboratoire en Informatique Haute
Performance pour le Calcul
et la simulation
91680, Bruyères le Châtel, France

Marc Pérache
marc.perache@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France
Université Paris-Saclay, CEA
Laboratoire en Informatique Haute
Performance pour le Calcul
et la simulation
91680, Bruyères le Châtel, France

Julien Jaeger
julien.jaeger@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France
Université Paris-Saclay, CEA
Laboratoire en Informatique Haute
Performance pour le Calcul
et la simulation
91680, Bruyères le Châtel, France

ABSTRACT

High-Performance Computing (HPC) is currently facing significant challenges. The hardware pressure has become increasingly difficult to manage due to the lack of parallel abstractions in applications. As a result, parallel programs must undergo drastic evolution to effectively exploit underlying hardware parallelism. Failure to do so results in inefficient code. In this constrained environment, parallel runtimes play a critical role, and their testing becomes crucial. This paper focuses on the MPI interface and leverages the MPI binding tools to develop a multi-language test suite for MPI. By doing so and building on previous work from the Forum document editors, we implement a systematic testing of MPI symbols in the context of the Parallel Computing Validation System (PCVS), which is an HPC validation platform dedicated to running and managing test suites at scale. We first describe PCVS, then outline the process of generating the MPI API test suite, and finally, run these tests at scale. All data sets, code generators, and implementations are made available in open-source to the community. We also set up a dedicated website showcasing the results, which self-updates thanks to the Spack package manager.

EUROMPI '23, September 11–13, 2023, Bristol, United Kingdom

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Distributed programming languages**; • **Computing methodologies → Parallel programming languages**.

KEYWORDS

HPC, test, api, MPI, validation

ACM Reference Format:
Julien Adam, Jean-Baptiste Besnard, Paul Canat, Hugo Taboada, Sameer Shende, Adrien Roussel, Marc Pérache, and Julien Jaeger. 2023. Generating and Scaling a Multi-Language Test-Suite for MPI. In *Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting (EUROMPI '23), September 11–13, 2023, Bristol, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3615318.3615329

1 INTRODUCTION

Programming a supercomputer has always been a challenging task. Indeed, as in any field, handling cutting-edge hardware requires specific care. In some aspects, High-Performance Computing (HPC) is special as it tends to combine relatively long-lasting technologies to create cutting-edge software. For example, MPI is 25 years old and is still productively used on hardware that is far from what was available when it was first released. This state of affairs can be explained by the highly specialized nature of the domains tackled by HPC, where the supercomputer is just a tool managed by super-specialists in their field – the simulation code is a means to an end. However, this may lead to poor performance due to the increasing

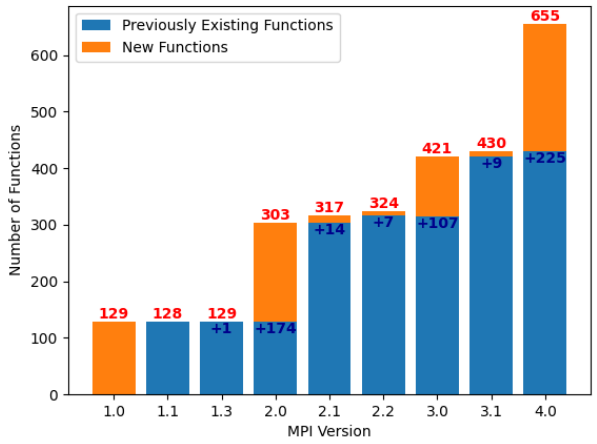


Figure 1: Evolution of the MPI standard size over its versions. Red values, total number of functions and blue values, functions added by the current MPI revision.

complexity of architectures and runtimes if the empirical use of computing resources is not managed efficiently.

With hardware differentiation, a given program has to target multiple kinds of computing resources. This means the code has to encompass the complexity of this hardware. Portability is often antinomic to performance, and programmers have to couple their program with multiple runtimes specific to a given kind of resource and sometimes fixed hardware. In such a changing landscape, the stability of the runtime's Application Programming Interface (API) is becoming of paramount importance. Not only may your program not compile, but you also need to assess the presence of some parallel features before using them - feature sets linked to the chosen runtime. For example, the OpenMP standard is released before being implemented in the various compilers at different paces, and programmers are required to read the documentation before choosing a feature.

Figure 1 shows the evolution of the number of functions in the MPI standard over its revisions, we have gathered these information in a structured manner on a dedicated website (see **https://mpicheck.pcvs.io/std/**). Major leaps are clearly visible, such as MPI-IO and RMA for 2+ [5, 17], MPI-T [11, 16] and non-blocking collectives [9, 10] for 3+, and eventually large counts [8] for 4+. Note ten functions have been removed from the standard between 2.2 and 3.0 and one between 1.0 and 1.1. From the perspective of MPI developers, it can be daunting to implement all 655 functions in the new standard, with the risk of missing one. Similarly, from the user's perspective, a new MPI primitive may only be available in a certain runtime release, making it difficult to identify the corresponding version.

In this paper, we pose the question of how to guide both end users and developers through the increasingly complex landscape of supported features and versions. To address this, we propose leveraging the Parallel Computing Validation System (PCVS) runtime, an HPC-oriented massively parallel testing and reporting framework with unique features suited for this task. Specifically, we implement a set of generated tests that cover exhaustively the MPI runtime interfaces (C, Fortran `mpif.h`, `use mpi`, and `use mpi_f08` to pinpoint levels of support in the various MPI runtime implementations. The code is available as open source, and the results are publicly distributed at the **https://mpicheck.pcvs.io/** URL, which we plan to maintain and enhance in the future by pulling new versions from the Spack package manager.

2 PARALLEL COMPUTING VALIDATION SYSTEM

The Parallel Computing Validation System (PCVS) is a testing framework that was originally developed to test the MPC MPI runtime [15]. As a result, it was designed with specific requirements to enable recompilation and parallel execution of tests. Coupled with the JCHRONOSS scheduler [2], it enables the massively parallel execution of test suites. Recently, PCVS has been completely rewritten from Perl to Python in order to provide a richer set of features, particularly on the analysis side and JCHRONOSS has now been embedded, also in Python. In this section, we provide more details on PCVS to highlight what makes it unique and specific in the HPC testing field. We first describe how PCVS can be used to run tests from a simple YAML syntax. Then, we detail how tests are executed in parallel on a supercomputer. Finally, we outline PCVS's analysis and reporting capabilities.

2.1 Workflow

PCVS is a command-line tool based on a compact YAML syntax that describes how a given test/job must be compiled and executed in parallel. However, this test description in PCVS is only half of what is required to run it. The notion of a "profile" is also required, which details how to map tests to the target system. This aspect is key to the retargeting capabilities of the PCVS runtime. PCVS was designed to run a given validation benchmark on arbitrary systems without requiring changes to the benchmark itself. Such changes, such as modifications to execution parameters, can quickly become combinatorial and can lead to a lack of portability of the overall test suite (i.e. hardcoded values). In contrast, PCVS carefully decouples the target system from the tests themselves and will only run the tests that intersect the two configurations.

Listing 1: Test description for Lulesh

```
lulesh:
    build:
        variants: [ 'openmp' ]
        files: '@BUILDPATH@/Makefile'
        make:
            target: 'all'
    run:
        program: 'lulesh2.0'
        iterate:
            n_node:
                values: [1, 2, 4]
            n_mpi:
                values: {op: 'powerof', of: 3}
            n_omp:
                values: [2, 4, 8]
```

As shown in Listing 1, the YAML description in PCVS covers both the build and execution of the test. In this example, the build

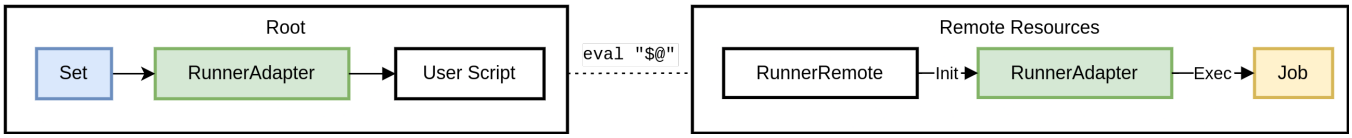


Figure 2: Illustration of the PCVS configuration interaction.

process invokes the OpenMP flags from the global profile and proceeds to run the Makefile in the current directory. On the execution side, the target binary name is specified, and the criterions are defined. Criterions are attributes defined globally or per program. It allows handling in a compact way a range of values a runtime or a program can take. Combined altogether, it creates a matrix of test-case scenarios to generate. From a single program definition, it may build thousands of test runs thanks to this parameterization. Criterions can be either bounded sets, such as the `n_omp` criterion in this case, or open sets, such as the `n_mpi` criterion, which includes all the powers of 3. Furthermore, this YAML description can be generated from any script using standard output, allowing for the handling of more complex scenarios that may not fit PCVS's standard semantics. Such YAML node is called a "Test Expression".

The PCVS runtime takes Test Expressions (TE) as input and generates a set of tests by combining the test parameters in a way that is compatible with the target execution environment. This process is guided by a machine profile, which defines the available resources on the target system. The resulting test set is the intersection of the TE specifications with machine profile. This approach allows test suites to be scaled dynamically from a single laptop to a large supercomputer without the need for manual editing of the test parameters.

PCVS is designed to work with different runtime environments and provides a way to add wrapper commands, such as `srun` and `mpirun`, to execute distributed jobs. However, some runtime environments may not support certain combinations of test parameters, such as a number of nodes greater than the number of MPI processes. To address this, PCVS allows users to define filter plugins that can remove impossible test combinations after the TE is evaluated. The runtime provides standard plugins for mpirun, and users can customize them by editing the configuration in the form of a Python script.

To summarize how PCVS handles tests, it can be noted that PCVS manages test compilation, allowing a test suite to be run independently of the target system. Additionally, PCVS can be coupled with the Spack package manager to compile and run tests, runtimes, and compilers. The ability to retarget a test suite to different systems at various scales is also a key feature for test-suite portability. Once a test or program is integrated into PCVS, it can be run in a portable way on any compatible system, and different compilers or compilation options can be propagated to the whole test suite through the change of a single option.

2.2 Scheduler

PCVS has an internal scheduler that works using a greedy heuristic. It launches the largest tests first, in terms of nodes by default, but this metric can be changed to any global criterion. Moreover, dependencies are resolved in a depth-first fashion, recursively unfolding the first dependency, running compilation tests first, to make an executable available for dependent tests. Parallel execution is handled by the target runtime, usually using either `srun` or a combination of `salloc` and `mpirun`. As far as parallel execution is concerned, PCVS either fills all the configured nodes or limits itself to a given number of parallel launches in order to remain graceful with the batch manager. Thanks to this very simple approach, PCVS is able to scale a given test suite to either a multi-core system or a massively parallel machine.

This approach quite simple is having a major drawback or requiring every job to allocate its own resources, maximizing the batch-manager allocation cost overhead. To circumvent this issue, PCVS provides a multi-level scheduling, packing jobs into sets, then unfolded and executed in a new execution context upon user request, as shown by the figure 2. Allocation wrappers are defined through profiles to be called when a set is launched. A common workflow of such script is to invoke `salloc` or `srun`. The way jobs are packed into sets is implemented through plugins, and can be customized to match the goal to achieve. A plugin is natively provided to optimize compilation launches by allocating single nodes and running a pool of compilation tests to spread the workload over cores.

2.3 Result Analysis and Reporting

Although PCVS was initially designed to build and run jobs, a more sophisticated way of assessing job success became necessary. While return code values are meaningful in many scenarios, more advanced capabilities such as pattern identification may be more appropriate. PCVS offers a dedicated syntax for TEs, detailing how a test should be considered successful, based on additional result attributes like targeted expected time, expected or not specific patterns, and even a custom script to achieve the greatest possible flexibility.

When designing PCVS, it was clear that the ability to process and store results was equally important to the ability to run the test suite itself. Given the constraints of HPC environments, the ability to move results around and persist them over time is crucial. To implement collaborative test suites and efficient result sharing, PCVS relies on a Git repository called a "bank". This method of sharing takes advantage of UNIX file-system or SSH-based access control, which simplifies access control to potentially sensitive application or platform-dependent data. Teams can share a single test repository and accumulate results over a long period. Git is particularly suited to handling change histories, so it is easy to track over time and assign responsibility for a given test-suite run.

On the storage side, PCVS relies on a hierarchy of JSON files matching the run test-suite tree. Each JSON file is mapping a test and its path is the exact test name it had during execution. Each JSON file contains identification (names, id, command line), its results raw information like the return code, the elapsed time or the output streams from the program (possibly truncated using a configurable threshold) and extra data, either attached to PCVS or as requested by the TE (tags, artifacts, metrics). Through artifacts a job may submit data to be persistently stored within job results directly from the test specification, and query or even build analysis based on information stored in it. From a Git Perspective, a commit maps to a single run storage while a branch is a list of runs, stored in order, associated with a given profile. To distinguish multiple independent runs relying on the same profile (as it may occur while testing MPI runtimes), a prefix can be added to the branch name to avoid any ambiguity as shown in Listing 2.

Listing 2: Example output for bank listing command

```
$ pcvs bank show mpicheck
Projects contained in bank '/path/to/bank':
- mpich   : 1 distinct testsuite(s)
  * mpich/194dc6e6a7cb78e56e1a59: 4 run(s)
- openmpi : 1 distinct testsuite(s)
  * openmpi/194dc6e6a7cb78e56e1a59: 5 run(s)
```

It is therefore possible to build a history of the test results for later query thanks to Git's logging capabilities on the temporal axis. In practice, a test-suite can be directly submitted to a *bank* after its execution, but it is also possible to work with regular archives for the case of disconnected networks.

As depicted in Listing 3 PCVS comes with a Python interface that we call the PCVS Domain-Specific Language to consume theses *banks*. It allows a high level analysis of the multiple test suites, including bank exploration, test listing and naturally individual test inspection; including captured values and artefacts. Thanks to these facilities it is then much easier to extract trends or develop custom analysis to add them to the validation pipeline.

Listing 3: Sample PCVS DSL python script extracting metadata from a given test serie

```
bank = dsl.Bank("/path/to/bank.git")
serie = bank.get_serie("mpich/194dc6e6a7cb78")
status = {'SUCCESS': [], 'FAILURE': [],
    'ERR_OTHER': []}
today = datetime.now()
# iterate over runs from last month
for run in serie.find(Serie.Request.RUNS,
          today - timedelta(days=30),
          today):
    # iterate on each test
    for job in run.jobs:
        status[job.state].append(job.name)
```

One aspect provided by the Git-inspired architecture is the possibility of having a centralized repository where all results are pushed. This repository does not require a server per se and is simply backed up by the file system. In such a configuration, scripts or even monitoring daemons can probe this repository using the DSL to generate higher-level validation metrics without having to develop specific analyses during each measurement. This allows for the deployment of dedicated monitoring and validation tools at the team level with simple scripts. For example, a simple crontab calling a Python script leveraging the DSL language could be used to send an email to the developer team, summarizing or warning about potential issues. Similarly, for metric tracking and visual reporting, it is easy to implement a Prometheus exporter consuming the performance database (updating only on push) to feed a Grafana dashboard for continuous display. Overall, this file-based storage with the availability of a Python consumer interface is facilitating the interoperability of the test-suite results. Such analyses may then be used as validation attributes for later execution of this same test.

3 MOTIVATION

This paper introduces PCVS, a testing framework that provides comprehensive handling of recompilation and test environment, which we refer to as *retargeting*. We demonstrate how this feature facilitates the validation of runtimes with diverse sources and constraints.

To demonstrate the capabilities of PCVS, we developed a systematic testing framework for the MPI standard API using the Spack package manager. Our framework runs exhaustive symbol tests to assess the presence of specific features in a given version. We present the test suite results in the PCVS reporting server, which uses Python Flask to provide a user-friendly browsing experience.

The MPI test suite is designed to ensure the correct implementation of MPI functions and facilitate their implementation. With the large count support, hundreds of functions were added to MPI, and they need to be present to fully support version 4.0. However, MPI runtimes currently do not support all MPI 4.0 features, and it is unclear if all features will be released at once. Therefore, end-users need to know which implementation can run a code using the latest features.

This paper is structured as follows: we first present related work, followed by a general presentation of our testing infrastructure. In the second part, we describe the tests we developed for MPI. Finally, we present the test results derived from these test suites for various versions of the respective runtimes.

4 RELATED-WORK

Validation in High-Performance Computing is a field in rapid expansion [13]. And there are several motivating factors for this expansion. First, machines are becoming more and more complex and hybridization is confirmed to be a requirement for increased performance. Transitively, parallel software willing to take advantage of diverse hardware needs to address multiple abstraction layers. And thus, these layers result in increasing code complexity. Managing this complexity is then crucial to be sure the program (1) runs as expected and (2) remains portable on a wide range of machines.

More generally, software development is always a complexity management undertaking and several abstractions have been developed to cope with it. First, agile software development methodologies in general have always advocated for a form of tests. Either simple integration tests or more ambitiously unit tests shaping requirements; Overall, tests are a vector of visibility and the less unknown there are the better the system is under control. Transitioning to HPC, due to the domain specific nature of the scientific codes targeting such platforms, development methodologies used to be of lesser priority. Physicists, mathematicians are first interested

in scientific results and the code is mostly part of the process and not an end in itself. However, at one point when the code is not actionable anymore development methodologies are a requirement to still produce results. This is where we see HPC is standing now and, why validation is becoming a dynamic field. Several solutions have been developed to address the issue of improving software quality in HPC software.

Pavilion2 [1] test framework is specifically designed to run and analyze tests on HPC systems. Similar to PCVS, it uses YAML-based input files to configure both system and test codes, and includes a powerful result parser model for fine-grained job assessment. It is highly customizable through plugins. Its own scheduler allows each test-case scenario to be adapted to the targeted resource. JUBE [14] focuses on running and analyzing benchmarks in a systematic way, with the ability to adapt to the heterogeneous systems through a minimal set of information in XML or YAML-based format. Beeswarm [18] [4] is a solution designed to automate application workflows and integrate with multiple CI providers, based on BEE [3], an autonomous containerization environment and workflow orchestration system. Unlike PCVS, it is not intended for end users but rather for programmatic use in automating the deployment of reproducible applications for validating production developments. Finally, ReFrame [12] is a python-based test framework for writing regression tests or benchmarks, targeting HPC systems by adding abstraction layers between test definition and underlying hardware, through a powerful support for batch managers.

PCVS offers similar capabilities but with additional support for dynamic retargeting of benchmarks to new environments without the need for editing, ensuring reproducibility and fair comparison in evaluating implementation divergence between runtimes.

5 SYSTEMATIC API VALIDATION WITH PCVS

PCVS has been designed as a validation tool capable of fully recompiling and running test cases. This is particularly important for runtimes because recompilation is necessary due to opaque linking wrappers, such as `mpicc` and the lack of MPI ABI. When entering a test in PCVS, you must not only describe how it is run (from the final binary) but also how it is compiled. Although this requires extra test integration effort, capturing this information has the advantage of allowing external parameters to be easily changed. For example, running with another compiler, changing some optimization flags, or replacing the BLAS implementation can now be done on a global profile basis rather than on per-test basis. This "retargeting" capability enables PCVS to run tests on multiple kinds of machines and return results in a unified manner.

In High-Performance Computing, there are several core interfaces that are essential for most production codes. These interfaces provide parallelism abstractions to run programs in a portable way on different machines. The Message Passing Interface (MPI) is commonly used for internode parallelism, and OpenMP is used to provide shared-memory parallelism. The availability of these interfaces is crucial for HPC codes and is generally a key component in machine procurement. In the rest of this paper, we propose to build a reliable test suite to assess the completeness of the MPI interface. This is intended for implementation developers as it is easy to overlook one of the hundreds of functions defined in MPI.

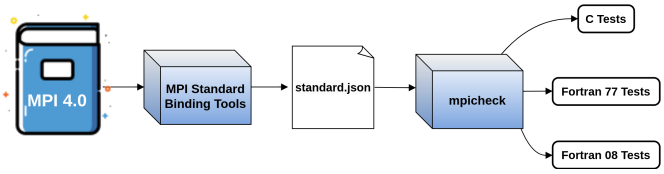


Figure 3: Test generation using the standard meta-data: `mpitest` is used to output the interface tests.

Additionally, end-users can also use this test suite to validate a given MPI implementation and ensure it supports the latest features of the standard. In the case of MPI, it is known to be backward compatible, meaning that deprecation rarely results in actual removal from the standard. One of the rare exceptions is the C++ bindings, which were deprecated in MPI 2.2 and removed in MPI 3.0. Even in cases of removal, implementations often keep the feature for longer, which can be challenging to track down as it is linked to the individual implementation changelog.

Comparatively, OpenMP is slightly less streamlined, particularly due to recent additions linked with accelerators. Unlike MPI, the OpenMP standard is ahead of the implementation, which means that the standard is released before being available in compilers. As a result, it is necessary to carefully check the compiler to see if it provides the required level of support to ensure your primitives are unfolded. Moreover, OpenMP is a "pragma" language, meaning that if the "pragma" is ignored, the code may remain totally valid but sequential. Consequently, if the runtime does not provide target support and warnings are overlooked during the compilation, the code may forget to use GPUs.

Although PCVS was originally designed to validate MPI implementations during their continuous integration cycle by running several tests and benchmarks at scale, this paper focuses on a smaller test set that targets standard support levels for MPI. This means that we do not validate features by themselves, but we simply check if the runtime allows us to compile and run a program depending on a given symbol in the MPI case. In the rest of this section, we will further describe our MPI test suite. Finally, we will summarize our testing infrastructure, outlining the reporting side and how the results are generated from periodic tests.

5.1 `mpicheck` Test Generation

As shown in Figure 3, we used the binding tools included in the MPI standard itself to validate the MPI interface. Since MPI 4.0, several parts of the standard document are generated through Python scripts. This means that there is a common description available for all MPI functions and symbols. We used this data, made available by the MPI Forum document editors (although not publicly at the time of writing), to generate systematic symbol tests for all MPI functions in MPI 4.0 with high fidelity to the standard. We also manually added functions that were deprecated by previous standards to achieve good MPI coverage. Additionally, we manually labeled each function with the versions of MPI in which it is present using the "label" capability in PCVS. These labels define meta-test groups, allowing for cross-validation of test results.

```c
#include <mpi.h>
int main(char argc,
         char**argv)
{
    int var_0;
    MPI_Request var_1[2];
    MPI_Status var_2[2];
    int ret;
    ret = MPI_Waitall( var_0, var_1, var_2);
    ret = PMPI_Waitall( var_0, var_1, var_2);
    return 0;
}
```

(a) C code

```
program main
include 'mpif.h'
INTEGER var_0
INTEGER var_1(10)
INTEGER var_2(10, 10)
INTEGER var_3
call mpi_waitall( var_0, var_1, var_2, var_3)
call pmpi_waitall( var_0, var_1, var_2, var_3)
end program main
```

(b) Fortran `mpif.h` code

```
program main
use mpi_f08
INTEGER :: var_0
TYPE(MPI_Request), DIMENSION(10) :: var_1
TYPE(MPI_Status), DIMENSION(10) :: var_2
INTEGER :: var_3
call mpi_waitall( var_0,  var_1, var_2, var_3)
call pmpi_waitall( var_0, var_1, var_2, var_3)
end program main
```

(c) Fortran `use mpi_f08` code

Figure 4: Example of generated code for the `MPI_Waitall` function.

5.2 Generated Tests

As far as test generation is concerned, we unfold the standard metadata to write interface tests for C and Fortran. As shown in Figure 4, this process relies on simple code generation to create individual test files (1) creating the right arguments for the call and then (2) calling for the MPI function. This test then checks if the symbol is exported by the MPI interface and if the arguments are correctly provided. However, this automatic test generation does not validate the correct implementation of the call itself as it would have required more extensive testing incompatible with scaffolding. We consider this second approach as future work and we consider leveraging Large Language Models (LLMs) to do so procedurally. In summary, what is tested is the compilation of the given symbol with the provided compiler, configurable as per the PCVS profile and target MPI runtime. These tests are declined for the various levels of support in the standard and reported in a structured manner using these labels.

5.3 Handling for Large Counts

MPI Large Counts [8] are one of the main contributor to the increase of the interface in MPI 4.0. Large counts consist of a new

```c
const void *var_0;
void *var_1;
MPI_Count var_2; /* Large Count */
MPI_Datatype var_3;
MPI_Op var_4;
MPI_Comm var_5;
int ret;
ret = MPI_Allreduce_c(var_0, var_1, var_2,
                      var_3, var_4, var_5);
ret = PMPI_Allreduce_c(var_0, var_1, var_2,
                       var_3, var_4, var_5);
return 0;
```

Figure 5: Sample code generated for large-count function `MPI_Allreduce_c` in a C code.

set of functions suffixed `_c`, mostly in C, and featuring large integer count parameters. Indeed, previous versions of MPI hit size barriers when manipulating large arrays, particularly MPI-IO where sizes can quickly overpass 32 GB (2^{32} * `sizeof(double)`). Previous mitigation for this has been to use derived datatypes to increase the multiplicative factor of the datatype size. However, new functions in the interface now featuring 64 bit parameters allow a more direct expression of these larger sizes. In Fortran, large count is not officially supported nor by the Forum or MPI implementations for `use mpif.h` or `use mpi`. Interface polyformism is used in Fortran with `use mpi_f08` to accomplish large count support. There is still a need to validate the polymorphism through calls with `MPI_COUNT_KIND` arguments. As presented in figure 5, the large count test is very similar to the test of the other functions, except that it relies on `MPI_Count` (instead of int) for its count datatype. The test is generated by the bindings description, unfolding the function twice to account for the large count implementation when necessary.

5.4 Integration with Spack

As depicted in Figure 6, to pull the last release of a given runtime, we relied on the Spack package manager. This package manager, specifically tailored for HPC was designed to build packages from their sources. Naturally, all the main HPC runtime and compilers are provided through this interface. The Spack repositories are periodically probed for a new version and if there is one, the test suite is run and submitted to the bank. This result bank saves in Git format all the results including those from other versions. Conjointly, the reporting portal monitor the result bank and render results when new data are available. This approach has the advantage that it is very generic. We just need to provide a list of MPI runtimes as input and by calling Spack we can install and switch between multiple versions.

6 TEST-SUITE RESULTS

In this section, we present the results of our validation of the MPI interface across several runtimes and versions. We will begin by describing our testing platform, followed by presenting the results we obtained in terms of standard coverage. Finally, we will conclude by characterizing the acceleration achieved through the use of the PCVS test execution scheduler when running tests in parallel.

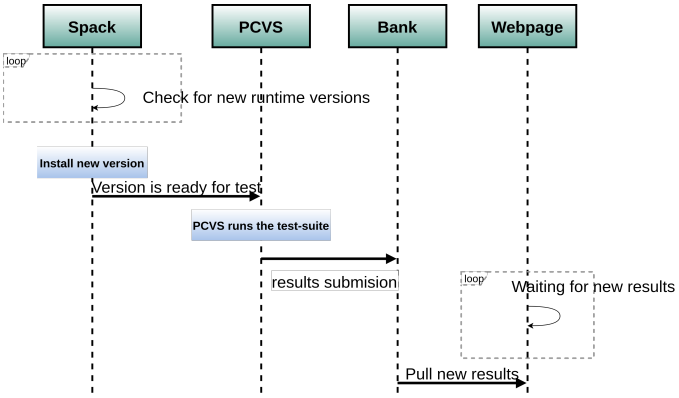


Figure 6: General PCVS validation pipeline.

6.1 On the Test Environment

Our MPI test suite can be run on any computer, as it mainly involves running 1988 compilations, resulting in a sequential runtime of approximately 15 minutes. However, when validating multiple runtimes across various versions, this time increases quickly, making parallel execution beneficial. It is worth noting that since our test suite is incremental, and new tests are added when a new MPI version is released in Spack, the need for parallelism is not critical. Our main goal is to demonstrate the parallelism capabilities of the PCVS runtime, which reduces the time to obtain results.

6.2 MPI Test-Suite

We have run the PCVS exhaustive interface test suite on multiple implementations of MPI, Open MPI [6] and MPICH [7] which are production level implementations and MPC [15], a more research oriented implementation. To do so, as previously discussed we leveraged the Spack package manager to compile and deploy the runtimes. We used the default configuration without specific tweaks. We then simply run the test suite after loading the respective MPI implementations. We have thus targeted the various implementations over C, Fortran. The test suite runs for all functions on the interface and test standards from 1.0 to 4.0 included. As 4.0 is very recent it is expected that several failures are linked to the absence of implementation and not to runtime errors. Tests are grouped using labels including standard levels, type of tests (e.g. collective, rdma, …) and presented firsthand on a per implementation basis.

As shown in Figure 7, presenting results on MPC, PCVS features a web-based reporting interface which allows the exploration of the results from multiple test suites, this web results is maintained available for trial at the following address **https://mpicheck.pcvs.io/**, where we plan to maintain the Spack-based test suite over time. Thanks to this interface, it is possible to explore the various MPI runtimes, looking at both successes and errors while filtering tests by *tags*. Available tags include, MPI standard versions and some general groups of functions (e.g. collectives, datatype, ...).

Figure 8 presents examples of errors caused by a missing function, `MPI_Isend_c`, which is an implementation of `MPI_Isend` designed for large counts. Note the Fortran test names are suffixed with `_c` to differentiate them from non large-count ones but do not

Figure 7: Sample output from the PCVS web-based result explorer, showing the tag-based view of MPC's result.

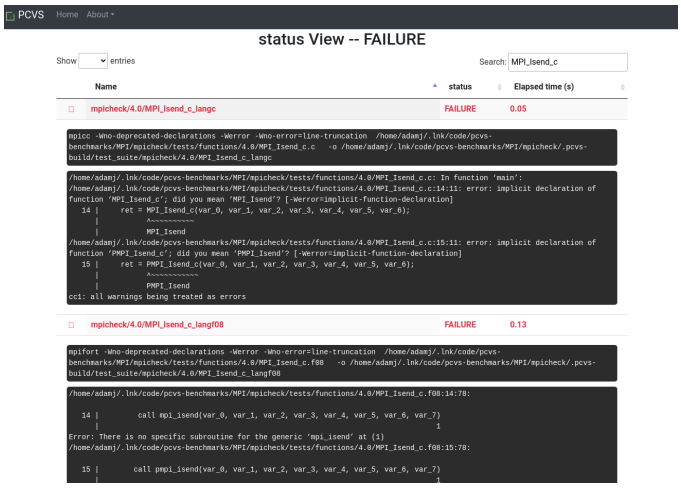
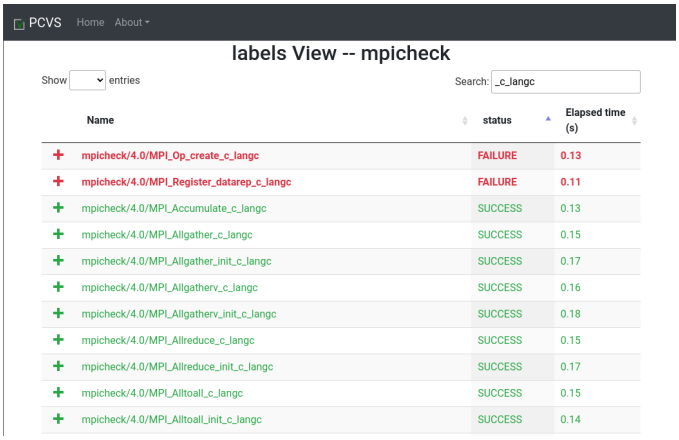
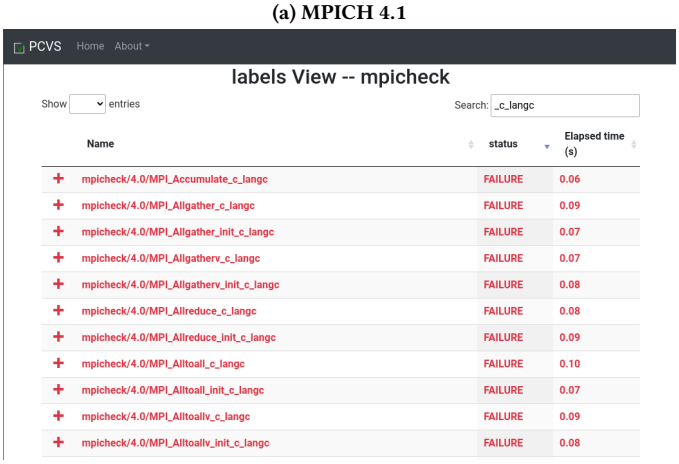


Figure 8: Output for missing large count `MPI_Isend_c` on OpenMPI 5.0-rc11.



(a) MPICH 4.1



(b) Open MPI 4.1.5

Figure 9: Comparison of C large count implementation results sorted by error/success.

translate to actual function names. In the case of the version of OpenMPI we tested (5.0-rc11), this function seems to be missing. When we compare the level of support for large counts between the two most recent versions of Open MPI and MPICH in Spack, we see differences, as shown in Figure 9. While MPICH has implemented large counts, this version of Open MPI has not, leading to a discrepancy in support. A complete list of runs for most of latest releases of the following MPI implementations are freely reachable at **https://mpicheck.pcvs.io/**: MPICH, OpenMPI, IntelMPI, mvapich2 and MPC.

Overall, with respect to features prior to version 4.0, we have not observed any specific shortcomings in either of the two production-level MPI implementations (i.e., Open MPI and MPICH). However, as shown in Figure 7, despite most failures are also large counts, MPC still lacks a few functions and encounters issues with Fortran bindings. Particularly for functions with strings and callbacks, requiring a fix to our custom bindings generator. This test suite is a tool we use to guide us in addressing these remaining issues. Thanks to the filtering capabilities provided in the reporting interface, it is easy to identify missing functions, which can help implementors avoid forgetting to implement them. For each MPI symbol, the function is present in multiple forms, depending on the targeted language (C, Fortrans `mpif.h`, `use mpi` and `use mpi_f08` are supported) and with or without the profiling-enabled call. It leads to at least 8 MPI call scenario to build. With so many possible implementations, it is easy to overlook one of them if the process is manual. Our test suite, which generates tests for all of these functions, can be particularly helpful for implementors. Similarly, our results could guide end users in assessing the level of support for a given runtime with respect to the recent MPI features. For example, Figure 9a and Figure 9b could be compared to determine the level of support for a specific MPI implementation. We plan to implement a dedicated analysis model to automate test-suite comparisons leveraging the test bank to provide historical data, while offering solutions to compare two runtimes.

6.3 Deploying the Test-Suite

The best way to get PCVS is through the PyPI repository:

```
# After this command you should have
# 'pcvs' in your path
pip3 install pcvs
```

You may clone mpicheck and (optionally) generate the PCVS-enabled test-suite with the generator script:

```
# Retrieve the test-suite
git clone https://github.com/cea-hpc/pcvs-benchmarks
cd pcvs-benchmarks
# Optional (pre-generated)
# will create ./MPI/mpicheck/tests/
cd ./MPI/mpicheck/
python3 mpicheck.py
```

You can then run the test suite and then start the reporting server on **http://localhost:5000** with:

```
# Create a default profile with MPI template
# It implies to find MPI wrappers into PATH
pcvs profile create -t mpi user.mpi
# Run the directory using this profile
# Results are stored locally in a hidden dir.
pcvs run -p mpi MPICHECK:./MPI/mpicheck/tests/functions
# Display results on localhost:5000
pcvs report
```

More information may be found at **https://pcvs.readthedocs.io/**. Using these simple instructions, you should be able to reproduce our results on your local MPI implementation, leading to the execution of the test suite as shown in figure 10. It is the exact same process we have followed to expose such results on **https://mpicheck.pcvs.io/**.

6.4 PCVS Scalability Test

To assess the performance improvement provided by the PCVS runtime, we conducted parallel MPI tests on a small prototype supercomputer consisting of 40 nodes. The tests were run on a partition with Bi-Socket AMD Rome and 128 cores per node connected via Infiniband interconnect. PCVS offers two levels of parallelism. The first level is based on allocations and involves running N parallel srun processes, each assigned a linear subset of the test suite. The second level of parallelism is within the allocation, where the individual runners can spawn a fixed number of parallel jobs, thereby taking advantage of node-level parallelism.

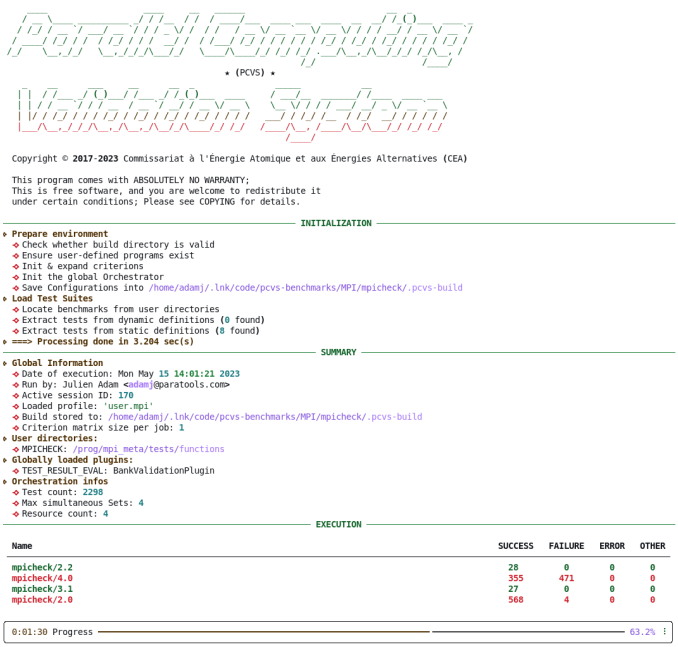


Figure 10: Sample output for PCVS running the mpicheck test-suite.

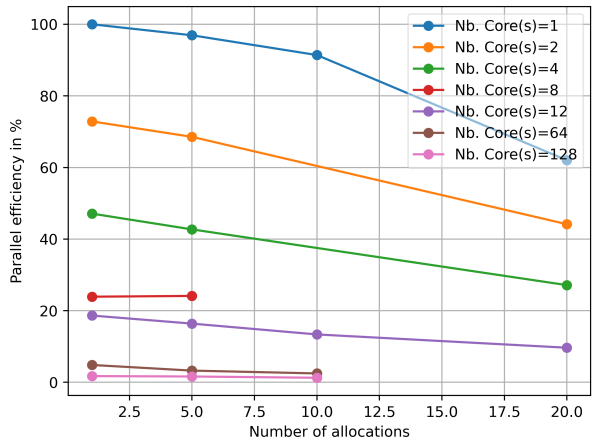


Figure 11: Evolution of the parallel efficiency in function of both the number of allocations and intra-node parallelism.

Figure 11 illustrates the achieved parallel efficiency in various combinations of the two levels of parallelism. It is evident that the efficiency decreases more with the increase of intra-node parallelism as compared to the gains obtained through parallel allocations. For instance, 20 allocations perform better than a single allocation with 12 cores. This observation indicates that the second level of parallelism in PCVS incurs significant sequential overhead, which makes it unfavorable for short jobs such as compilations. We intend to address this issue and expect that the scheduling overhead will be much more negligible for longer jobs.

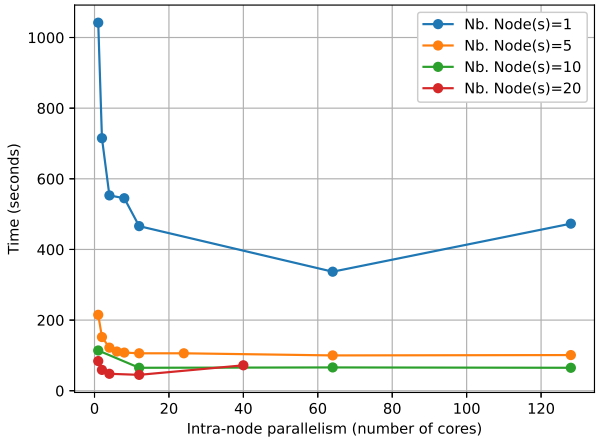


Figure 12: Comparison of inter and intra-node acceleration when running the MPI test-suite.

Despite this limitation on the second level of parallelism, Figure 12 presents the results of the acceleration in terms of total time. As previously discussed, most of the gains come from the number of allocations and not from intra-node parallelism. Thanks to the parallel execution of PCVS, we were able to reduce the duration of the test suite from 17 minutes and 22 seconds (1042 seconds) to 84 seconds using 20 cores, resulting in a parallel efficiency of 62%.

7 CONCLUSION

In this paper, we presented our implementation of an exhaustive MPI test suite generated from the standard metadata. To run these tests, we utilized the Parallel Computing Validation System (PCVS), a parallel test-suite runner designed for HPC. We highlighted the increase in the number of MPI functions over time and the need for automated testing to track symbols in the various bindings. Our classification of the MPI standard functions, helped gather tests on a standard-revision basis. We described the architecture of PCVS, its capabilities, and implementation details. We then outlined the implementation of tests in different output languages (C and Fortran, with support for any revision). Additionally, we presented and discussed the results of the test suite, followed by an evaluation of its scalability, reducing the execution time to less than one minute for the whole interface test suite.

In summary, our implementation of the exhaustive MPI test-suite integrated with PCVS has proven to be efficient and scalable for testing MPI functions across different bindings and languages. Our work provides a valuable resource for the HPC community, both for implementors to assess the presence of numerous functions of the standard in various interfaces and for end users to check if their version of the runtime contains the desired features.

8 FUTURE WORK

This paper presented the first iteration of our validation system. We started with symbols as they are one of the simplest thing to test. We will soon complement the tests with the constants and datatypes defined by the standards. In a second time, as these tests are only interface tests, allowing to check if the symbol is present, and not if it is operational, we plan to complement them with functional tests. As it can be cumbersome to write tests validating all the functions of the MPI standard, we plan to leverage a Large Language Model (LLM) to generate the corresponding code, as such models are surprisingly good at manipulating code. In addition as MPI is working on defining an ABI, we plan to implement the corresponding tests as part of our interface test suite, the same could be done for other languages bindings such as RUST and Python.

On the implementation side, future work will focus on enhancing the second level of parallelism in PCVS to benefit longer jobs and on incorporating the test suite in continuous integration and development pipelines. We are also willing to enhance the reporting side as we would like to build a website showcasing the results in a more integrated manner, as of now it is simply the regular PCVS reporting interface and tests presentation can be improved.

ACKNOWLEDGMENTS

We thank the MPI Forum Committee for its recent work to uniformize the Document through a proper Python Interface, allowing third-party tools to automatically extract information relatively to the API, paving the way to automatize many tasks around MPI and its numerous implementations. We would like to thanks Ken Raffenetti from Argonne National Laboratory for his contribution with calling convention complexities with the MPI Fortran interface.

REFERENCES

[1] 2020. Pavilion2. **https://pavilion2.readthedocs.io/**

[2] Julien Adam and Marc Pérache. 2016. A Parallel and Resilient Frontend for High Performance Validation Suites. In *International Conference on Vector and Parallel Processing*. Springer, 248–255.

[3] Jieyang Chen, Qiang Guan, Xin Liang, Louis Vernon, Allen McPherson, Li-Ta Lo, and James Ahrens. 2017. Docker-Enabled Build and Execution Environment (BEE): an Encapsulated Environment Enabling HPC Applications Running Everywhere. (12 2017).

[4] Jieyang Chen, Qiang Guan, Li-Ta Lo, Patricia Grubel, and Tim Randles. 2020. BeeSwarm: Enabling Scalability Tests in Continuous Integration. arXiv:2007.10491 [cs.DC]

[5] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. 1996. MPI-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing: Second International Euro-Par Conference Lyon, France, August 26–29 1996 Proceedings, Volume I 2*. Springer, 128–135.

[6] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. 2006. Open MPI: A flexible high performance MPI. *Lecture notes in computer science* 3911 (2006), 228.

[7] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 22, 6 (1996), 789–828. **https://doi.org/10.1016/0167-8191(96)00024-5**

[8] Jeff R Hammond, Andreas Schäfer, and Rob Latham. 2014. To INT_MAX... and beyond! Exploring large-count support in MPI. In *2014 Workshop on Exascale MPI at Supercomputing Conference*. IEEE, 1–8.

[9] Torsten Hoefler, Prabhanjan Kambadur, Richard L Graham, Galen Shipman, and Andrew Lumsdaine. 2007. A case for standard non-blocking collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting, Paris, France, September 30-October 3, 2007. Proceedings 14*. Springer, 125–134.

[10] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 1–10.

[11] Tanzima Islam, Kathryn Mohror, and Martin Schulz. 2014. Exploring the capabilities of the new MPI_T interface. In *Proceedings of the 21st European MPI Users' Group Meeting*. 91–96.

[12] Vasileios Karakasis, Victor Holanda Rusu, Andreas Jocksch, Jean-Guillaume Piccinali, and Guilherme Peretti-Pezzi. 2017. A regression framework for checking the health of large HPC systems. In *Proceedings of the Cray User Group Conference*.

[13] Verónica G Vergara Larrea, Michael J Brim, Arnold Tharrington, Reuben Budiardja, and Wayne Joubert. 2020. Towards acceptance testing at the exascale frontier. In *Proceedings of the Cray User Group 2020 conference*.

[14] Sebastian Lührs, Stephan Graf, Alexander Schnurpfeil, Wolfgang Frings, and Kay Thust. 2015. *JUBE–A Flexible, Application-and Platform-Independent Environment for Benchmarking*. Technical Report. Jülich Supercomputing Center.

[15] Marc Pérache, Patrick Carribault, and Hervé Jourdren. 2009. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Matti Ropo, Jan Westerholm, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 94–103.

[16] Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D Malony, Hari Subramoni, and Dhabaleswar K Panda. 2017. MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU. In *Proceedings of the 24th European MPI Users' Group Meeting*. 1–11.

[17] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*. 23–32.

[18] Jake Tronge, Jieyang Chen, Patricia Grubel, Tim Randles, Rusty Davis, Quincy Wofford, Steven Anaya, and Qiang Guan. 2021. BeeSwarm: Enabling Parallel Scaling Performance Measurement in Continuous Integration for HPC Applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1136–1140. **https://doi.org/10.1109/ASE51524.2021.9678805**