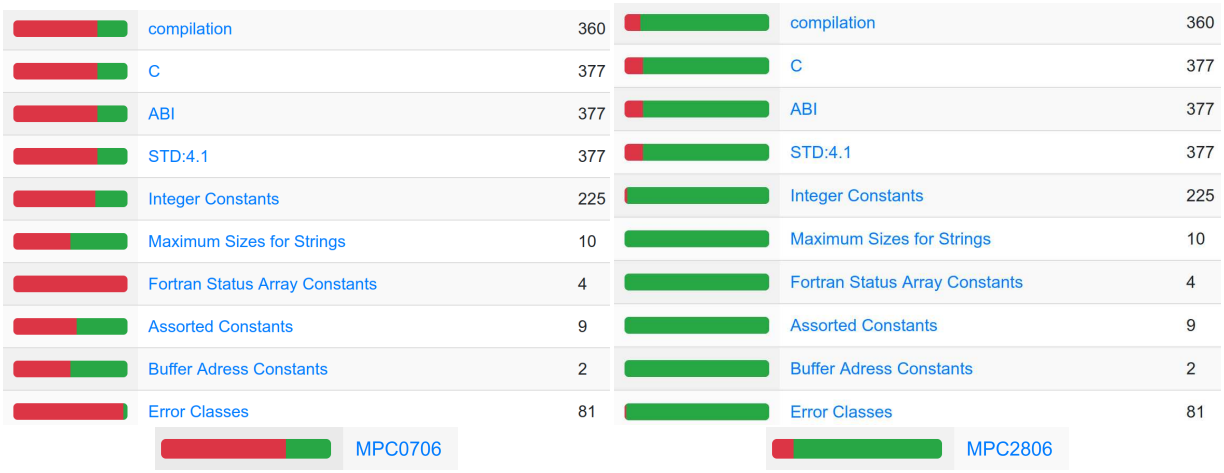
Implementing the MPI ABI in the MPC MPI Runtime

Corentin Beaulieu
corentin.beaulieu@cea.fr
CEA, DAM, DIF
Arpajon, France

Julien Jaeger
julien.jaeger@cea.fr
CEA, DAM, DIF
Arpajon, France

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



(a) Before the ABI efforts. **(b) After the ABI efforts.**

Figure 1: PCVS output for the ongoing port of the ABI in the MPC runtime.

ABSTRACT

MPI is the de-facto standard for message-passing and more generally distributed memory programming in High-Performance Computing (HPC). For the last 25 years, this programming interface has continuously renewed itself to provide the best performance to parallel programs. With a rigorous standardization process and backward compatibility, MPI is known as a reliable interface. However, recent evolutions have pushed for implementing an Application Binary Interface (ABI) in the standard, for several reasons that we will quickly cover in the context of the MPC MPI runtime. In this poster, we first outline our vision of the ABI benefits for MPC and more generally MPI and then detail how we implement it in the context of this specific runtime. We then conclude with the opportunities for a more open MPI ecosystem thanks to such ABI.

KEYWORDS

MPI, ABI, Testing

EuroMPI 2023, September 2023, Bristol UK

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Corentin Beaulieu, Julien Jaeger, and Jean-Baptiste Besnard. 2018. Implementing the MPI ABI in the MPC MPI Runtime. In *Proceedings of EuroMPI 2023.* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

For the last 25 years, MPI has been setting the standard for message passing in High-Performance Computing, enabling distributed memory computation at scale. In computer science, such longevity is exceptional and it testifies to the exceptional capabilities and usefulness of the MPI standard. Meanwhile, the HPC landscape is about to rapidly change which the rise of new drivers in the computing demands: machine learning. While simulation usefulness is not to be questioned, driving HPC since the start[2], Machine Learning (ML) payloads are probably the use-case which will leads to the largest computing demand in the near future[18, 21]. Despite ML being mostly based on Python frameworks and GPUs[1, 19], there are no reasons it shall not leverage more efficient native languages and eventually distributed memory to further accelerate its results.

This general introduction is bringing us to the implementation of the ABI inside MPI[16]. Indeed, we think that MPI has to become more agile in terms of target languages and ecosystem to make it a reliable candidate for ML payload willing to scale out of a single node. Due to its longevity, MPI's usefulness is demonstrated for tightly bound HPC payloads, the question is then how to extend it to other payloads and more precisely other programming languages.

2 TOWARDS A MULTI-LANGUAGE INTERFACE

We think that one of the first characteristics of ML payloads is that they generally use higher-level languages and solely run their compute-intensive kernels using native languages (Numpy[22], Cuda). MPI usage for such payloads is then associated with the availability of expressive bindings in the language of interest, mostly Python. This process is currently cumbersome as it is difficult to target the various runtimes of interest (OpenMPI and MPICH) as they do not share the same ABI. For this reason, the definition of an MPI ABI is a facilitator for such effort – ensuring the wide availability of MPI for Python (and many others) bindings.

HPC is used to a limited set of scenarios, OpenMP, MPI, Fortran, C, and C++[3, 12] while ML is much more agile in terms of computational expressivity[17] – relying on a wide range of runtimes. There is then certainly an area of convergence keeping the best of both worlds, the flexibility of higher-level languages and the efficiency of compiled languages. Growing interest[6, 9, 14] for Python[8], Rust[7, 13] and Julia[5] in HPC is testimony to the need for new abstractions and alternative ways of defining computing.

Following this statement, we considered that the ability to gracefully interact with other languages is a priority for MPI and therefore, this need translated to the implementation of the MPI ABI in the MPC MPI runtime which is a research vehicle for MPI.

3 MULTI-PROCESSOR COMPUTING (MPC)

MPC[20] is a thread-based implementation of MPI. It aims at transparently converting regular MPI processes (bound to UNIX processes) to thread-based processes (bound to user-level threads). This process is supported by a dedicated modified compiler which translates global variables to Thread-Local Storage (TLS) automatically[4]. This enables a single UNIX process to run the same binary multiple times inside a shared-address space. In addition, MPC is also featuring an OpenMP runtime[10] and an implementation of User-Level Threads (ULT) compliant with Pthreads to capture all active wait state into the scheduler.

4 IMPLEMENTING THE ABI

As far as the ABI is concerned it is based on pointer handlers and specifies values for all the handles in the space of the first system page (less than 4KB). Implementing the ABI is then matching all the handle types in MPI with pointers and following all the values as stated by the MPI standard. The reference implementation for this is done by Hammond in the context of Mukautuva[11], it is what we follow to make a first implementation in MPC.

4.1 Current State of MPC

Due to constraints in the Fortran handles, MPC originally used integer-based handles to manage its whole interface. This allowed the direct implementation of F77 bindings without any form of conversion[23]. The drawback is naturally that the conversion is systematic for C and C++ while not being compulsory. A recent refactoring has defined pointer-based handles for communicators and MPI windows[15], but all handles must be completed to match the ABI constraints. It is this process that the poster outlines, with a focus on how we monitor the change and progress thanks to test-driven development.

4.2 Outline of ABI Changes

Changing from integer handles to pointers is a relatively simple process for C as practically all handles were already internally structs and there were translation mechanisms to move from and to these integers. These structs have now to be exposed as opaque structs (which are also normalized) to follow the proposed ABI. This then leads to changes in the outer layers of the MPI interface, removing conversions and eventually removing some logic and tests in the MPC runtime, simplifying the C interface logic by directly consuming the opaque struct.

5 TEST-DRIVEN APPROACH

Due to the number of cases to handle, we have started our implementation of the ABI inside MPC by working on a generic test suite for the new ABI. As the ABI is still a moving target and under the active definition in the standard, we needed to define a dynamic way of testing. We then use a JSON file which is unfolded thanks to a dedicated script into multiple tests. This stands both for types and values.

Listing 1: Json defining the test for an MPI_Datatype handle.

```
{"name": "MPI_Datatype", "value": "struct MPI_ABI_Datatype*",
 "kind": "type", "lang": "c" }
```

Using the code defined in the Listing 1, the test of Listing 2 is unfolded to validate that the target handle has the right type. It uses the C11 `_Generic` extension to validate the equality of the handle typedef and its opaque struct name. As far as the predefined values are concerned this is simply based on asserts.

Listing 2: Test generated from Listing 1.

```
#include <mpi.h>
#include <assert.h>
#include <stddef.h>
#include <stdint.h>

int main(void)
{
    MPI_Datatype var;
    int sametype = _Generic(var, struct MPI_ABI_Datatype *:1, ⟷
          default: 0);
    assert(sametype);
    return 0;
}
```

With the test, we also leverage the PCVS testing runtime to run all the tests in parallel, while generating a report of our porting efforts.

5.1 State of the Implementation

As presented in Figure 1 which shows a report of the ABI test suite both before and after the beginning of our porting efforts we are currently in the process of fully implementing the ABI in MPC. Overall the process is relatively straightforward and we do not encounter practical issues to do so. It is mostly a repetitive and error-prone process due to the large number of functions and handle values in MPI. For this reason, we believe that the ABI test suite which we have built to guide our efforts will be useful for the wider MPI community ensuring all ABI requirements are fulfilled.

For this reason, this code will be part of the http://mpicheck.pcvs.io automated test suite as soon as ABI constraints start to be drafted as a standard.

6 CONCLUSION

In this poster, we described how MPC is implementing the future MPI ABI. We first motivated the in-depth changes we see in the HPC landscape in the near future and outlined how important language bindings will be in this changing context. We have defined a complete and automatically scaffolded test suite for the MPI ABI, test-suite that can be used by other MPI runtimes to ensure their compliance. This test-suite will be released in open-source as part of mpicheck shortly.

Following this effort, we are also considering implementing parts of the MPC runtime in other languages, we have efforts undergoing to reimplement the user-level scheduler in Rust and we want to continue this modularization. It is also important to note that thanks to the ABI it is now possible to create portable plugins to replace part of the MPI interface. Such plugins can also be in any language as long as they satisfy the C ABI. We see this use of new languages as an opportunity to experiment with other languages which may provide opportunities for more idiomatic implementations and transitively more actionable codes.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Gordon Bell, David H Bailey, Jack Dongarra, Alan H Karp, and Kevin Walsh. 2017. A look back on 30 years of the Gordon Bell Prize. *The International Journal of High Performance Computing Applications* 31, 6 (2017), 469–484.

[3] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. 2020. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* 32, 3 (2020), e4851.

[4] Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D Maloney. 2016. Introducing task-containers as an alternative to runtime-stacking. In *Proceedings of the 23rd European MPI Users' Group Meeting*. 51–63.

[5] Simon Byrne, Lucas C Wilcox, and Valentin Churavy. 2021. MPI. jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, Vol. 1. 68.

[6] Valentin Churavy, William F Godoy, Carsten Bauer, Hendrik Ranocha, Michael Schlottke-Lakemper, Ludovic Räss, Johannes Blaschke, Mosè Giordano, Erik Schnetter, Samuel Omlin, et al. 2022. Bridging HPC Communities through the Julia Programming Language. *arXiv preprint arXiv:2211.02740* (2022).

[7] RustMPI Contributors. 2023. RustMPI: MPI bindings for Rust. https://github.com/rsmpi/rsmpi. GitHub repository.

[8] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. 2008. MPI for Python: Performance improvements and MPI-2 extensions. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 655–662.

[9] Patrick Diehl, Steven R Brandt, Max Morris, Nikunj Gupta, and Hartmut Kaiser. 2023. Benchmarking the Parallel 1D Heat Equation Solver in Chapel, Charm++, C++, HPX, Go, Julia, Python, Rust, Swift, and Java. *arXiv preprint arXiv:2307.01117* (2023).

[10] Manuel Ferat, Romain Pereira, Adrien Roussel, Patrick Carribault, Luiz-Angelo Steffenel, and Thierry Gautier. 2022. Enhancing MPI+ OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support. In *International Workshop on OpenMP*. Springer, 3–16.

[11] Jeff Hammond. 2023. Mukautuva: A Demonstrator for the MPI ABI. https://github.com/jeffhammond/mukautuva. GitHub repository.

[12] Atsushi Hori, Emmanuel Jeannot, George Bosilca, Takahiro Ogura, Balazs Gerofi, Jie Yin, and Yutaka Ishikawa. 2021. An international survey on MPI users. *Parallel Comput.* 108 (2021), 102853.

[13] Matthias Kübrich. 2020. Integration and Test of RUST Tool Support for MPI. (2020).

[14] Wei-Chen Lin and Simon McIntosh-Smith. 2021. Comparing julia to performance portable parallel programming models for hpc. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 94–105.

[15] MPC Development Team. 2023. MPC MPI Runtime - Release 4.2.0. https://mpc.hpcframework.com/download/. Official website.

[16] MPI ABI Working Group. 2023. MPI ABI Working Group. https://github.com/mpiwg-abi. Accessed on: July 7, 2023.

[17] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. 2019. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review* 52 (2019), 77–124.

[18] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[20] 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: 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings 16*. Springer, 94–103.

[21] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[22] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering* 13, 2 (2011), 22–30.

[23] Junchao Zhang, Bill Long, Kenneth Raffenetti, and Pavan Balaji. 2014. Implementing the MPI-3.0 Fortran 2008 binding. In *Proceedings of the 21st European MPI Users' Group Meeting*. 1–6.

Implementing the MPI ABI in the MPC Runtime

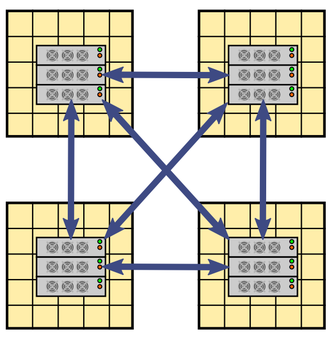
Towards inter implementations operability

C. Beaulieu[1], J. Jaeger[1,3], J.B. Besnard[2]

1 CEA, DAM, DIF, F-91297 Arpajon, France
2 ParaTools SAS, 91680 Bruyères-le-Châtel, France
3 Labratoire en Informatique Haute Performance pour le Calcul et la Simulation, 91680 Bruyères-le-Châtel, France

1 MPI

- Most used library for parallel distributed computing
- Standardized API in C and Fortran



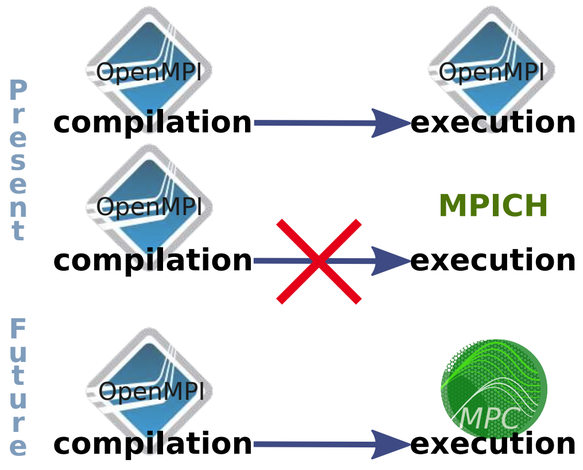
Message Passing Interface

2 Why Standardize the ABI?

Two different implementations are most likely to be incompatible

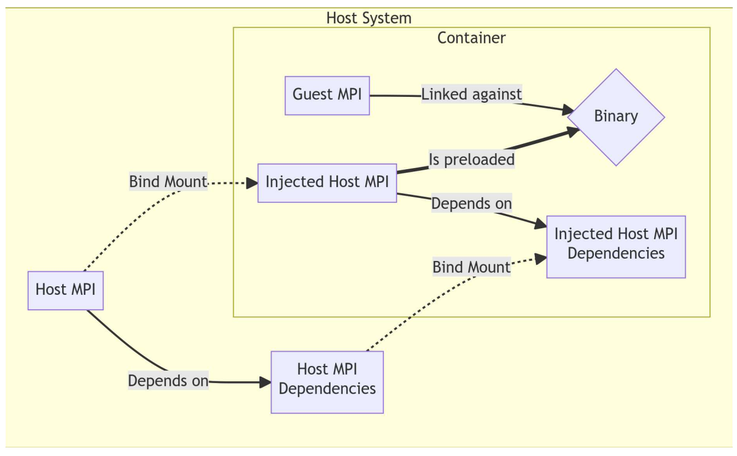
Simplify the use of MPI

Have to match compilation and linkage



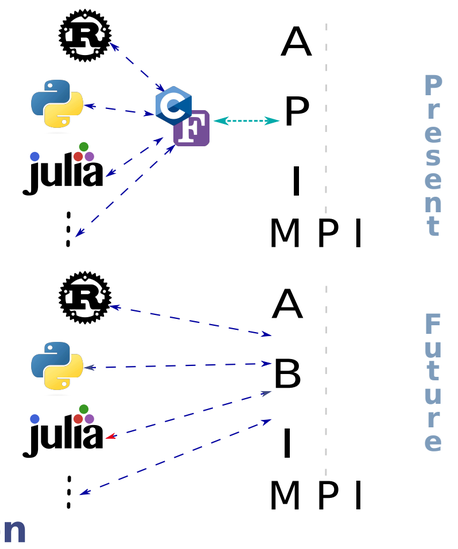
Help containerize

Two MPI implementations : Host and container



Open up to new languages

Must use the language specific API

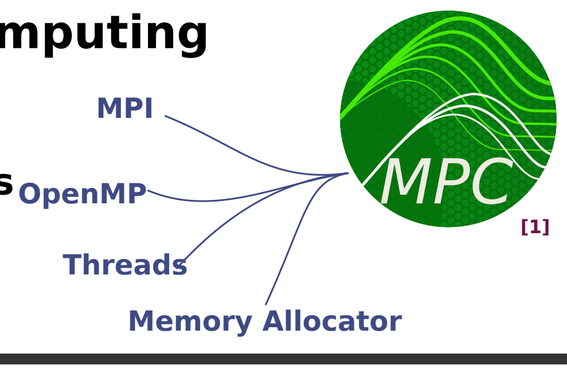


Improve profiling and debugging

Packet gestion

3 Multi-Processor Computing

- Unified HPC framework
- Based on user-level threads
- Multiple modules



https://mpc.hpcframework.com/

4 Parallel Computing Validation System

- Validation engine
- High scalability
- Precise results visualization

PCVS [2]

Get it

```
$ pip3 install pcvs
```

Run a test suite

```
$ pcvs run -p abi MPC:./abi-tests
```

https://pcvs.hpcframework.com/

5 MPICHECK

- Test suite
- Validate the MPI API
- Look for symbols
- Categorized by
  - language
  - standard version
  - type of function

Get it

```
$ git clone https://
github.com/
```

Generate the suite

```
$ python3
```

Progress	Name	Test Count
	compilation	2298
	c	634
	functions	2298
	STD:2.2	1143
	STD:3.0	1445
	STD:3.1	1472
	STD:4.0	2298
	topology	76
	f90	557
	f77	555

https://mpicheck.pcvs.io/

6 Check the ABI

- Values from Mukautuva
- Listed in MPI Standard, Annex A

Test description in JSON

```
{"name": "MPI_Datatype", "value": "struct
MPI_ABI_Datatype *",
```

Example of C test

```
MPI_Datatype var;
int sametype = _Generic(var, struct MPI_ABI_Datatyp
        default: 0);
assert(sametype);
```

Python Script

PCVS yaml configuration file

- Standard categories

PCVS

Validate the suite

Get results and visualize it

	OpenMPI 4.1.5	MPICH 4.1.1
compilation	360	360
C	377	377
ABI	377	377
STD:4.1	377	377
Integer Constants	225	225
Maximum Sizes for Strings	10	10
Fortran Status Array Constants	4	4
Assorted Constants	9	9
Buffer Adress Constants	2	2
Error Classes	81	81

Not defined

	Mukautuva
compilation	360
C	377
ABI	377
STD:4.1	377
Integer Constants	225
Maximum Sizes for Strings	10
Fortran Status Array Constants	4
Assorted Constants	9
Buffer Adress Constants	2
Error Classes	81

360 Tests

7 Implementing the ABI

	MPC 4.1.0
compilation	360
C	377
ABI	377
STD:4.1	377
Integer Constants	225
Maximum Sizes for Strings	10
Fortran Status Array Constants	4
Assorted Constants	9
Buffer Adress Constants	2
Error Classes	81

Common

- Structures
- Handles
- Constants
- Types

- Test-driven
- Modified constants
- Modified handles

MPC after modifications

8 Conclusion

The standardization of the MPI ABI is a new layer of constraints. Nevertheless, it will allow new possibilities for the standard to evolve and conquer new users. The first drafts and works on it has allowed us to build a test suite to check the compliance with an ABI and to modify MPC MPI implementation to get closer to what the ABI may be.

[1] https://github.com/cea-hpc/mpc

[2] https://github.com/cea-hpc/pcvs

[3] https://github.com/jeffhammond/mukautuva