A C++20 Interface for MPI 4.0

Ali Can Demiralp
Visual Computing Institute
RWTH Aachen University
Aachen, Germany
demiralp@vis.rwth-aachen.de

Philipp Martin
IT Center
RWTH Aachen University
Aachen, Germany
pm.martin@itc.rwth-aachen.de

Niko Sakic
IT Center
RWTH Aachen University
Aachen, Germany
sakic@itc.rwth-aachen.de

Marcel Krüger
Visual Computing Institute
RWTH Aachen University
Aachen, Germany
krueger@vis.rwth-aachen.de

Tim Gerrits
Visual Computing Institute
RWTH Aachen University
Aachen, Germany
gerrits@vis.rwth-aachen.de

***Abstract*—We present a modern C++20 interface for MPI 4.0. The interface utilizes recent language features to ease development of MPI applications. An aggregate reflection system enables generation of MPI data types from user-defined classes automatically. Immediate and persistent operations are mapped to futures, which can be chained to describe sequential asynchronous operations and task graphs in a concise way. This work introduces the prominent features of the interface with examples. We further measure its performance overhead with respect to the raw C interface.**

***Index Terms*—Message Passing, Software Libraries, Application Programming Interfaces**

I. Introduction

The message passing interface (MPI) is the standard programming model for distributed computing today, yet it lacks an official C++ interface since version 3.0. Applications written in C++ have to rely on the C interface, which provides no encapsulation, requires manual memory and scope management, and prevents use of C++ idioms and features. As detailed by Rüfenacht et al. [1], many unofficial C++ interfaces such as Boost.MPI [2] and MPL [3], [4] exist. However these often target earlier versions of MPI (< 4.0), cover a subset of their respective specification, and tend to limit their usage of language features, serving mostly as RAII wrappers. Although the MPI forum is actively discussing the potential features of a new C++ interface [5], it is currently far from standardization.

This work presents a modern C++20 interface for MPI 4.0, covering the complete specification. The interface provides automatic lifetime management for each MPI object, meaningful defaults for each MPI function, compile-time generation of MPI data types from structures and classes, and the ability to express MPI requests as futures with continuations to describe sequential non-blocking communication. We implement the majority of the features requested by the users in [5], envisioning what an official C++ interface could look like today.

II. Implementation

The interface is implemented as a header-only library depending on the C interface. It consists of three major components; the core, IO and tool interfaces. The core component implements the chapters 1-13 of the standard [6], containing all point-to-point, collective and one-sided communication. The IO component implements chapter 14, the MPI-IO interface, covering the functions with the prefix *MPI_File_*. The tool component implements chapter 15, the profiling and tool information interface of MPI, containing the functions with the prefix *MPI_T_*.

The interface closely follows the C++ core guidelines [7], which encourage idiomatic use of the language and the standard library. The classes have two types of constructors; managed and unmanaged. Managed constructors instantiate a new MPI object and assume responsibility for its destruction. Unmanaged constructors accept an existing MPI object and do not assume responsibility for its destruction by default. Copy constructors are deleted unless MPI provides duplication functions (ending with *_dup*) for the object, whereas move constructors are available whenever possible.

All function pointers are converted to *std::function*s, which enables user data to be passed through captures rather than void pointer arguments. The library further contains scoped versions of each enumeration. Functions expecting enumerations as arguments use these scoped versions, which prevent passing erroneous values and provide code completion support. The arguments of functions that accept a variety of MPI objects are described with a *std::variant*, providing constrained type erasure. Optional arguments and indeterminate return values, such as the result of an immediate probe, are described using *std::optional*.

The library provides default arguments for most MPI functions, according to the standard where applicable. In several cases, the defaulted arguments are required to be moved at the end of the argument list. Furthermore, functions with a large number arguments accept description objects encapsulating the arguments instead.

User-defined classes must be registered as MPI data types prior to being used in communication. Our interface is capable of generating MPI data types for custom classes automatically,

```
struct custom_type
{
  std::uint64_t         id       ;
  std::array<float, 3> position;
}

custom_type custom;
if (communicator.rank() == 0)
{
  custom = custom_type {42, {1.0f, 2.0f, 3.0f}};
  communicator.send    (custom, 1);
}
if (communicator.rank() == 1)
{
  communicator.receive(custom, 0);
  // custom == custom_type {42, {1.0f, 2.0f, 3.0f}};
}
```
Listing 1. User-defined types can be used in communication without explicitly creating an MPI data type.

as seen in Listing 1. This functionality is based on PFR [8], which enables compile-time introspection of aggregate classes. Arithmetic types, enumerations and specializations of *std::complex* fulfill the *mpi::compliant* concept and are mapped to their MPI equivalents explicitly. Furthermore, C-style arrays, *std::arrays*, *std::pairs*, *std::tuples* and aggregate types consisting of compliant types are also compliant types themselves. The communication functions can be used with a single or a contiguous sequential container (i.e. *std::string*, *std::span*, *std::valarray*, *std::vector*) of compliant types.

The requests returned by the interface are castable into futures, which can be chained to express asynchronous sequential operations as seen in Listing 2. This feature serves as a bridge between the concurrency support library of the C++ standard and the non-blocking communication functionality of MPI. It further enables task graphs where forks are expressed as multiple futures started from the current context, and joins are expressed with *mpi::when_all* or *mpi::when_any* which forward the underlying requests to *MPI_WaitAll* or *MPI_WaitAny* respectively.

Error handling is performed by checking the return values of viable MPI functions for success, throwing an exception otherwise. It is optionally enabled at compile-time by defining a macro prior to inclusion of the library headers. The exceptions provide an error code, which derives from the error class as specified by the standard. Default error codes are available as variables scoped in the *mpi::error* namespace.

```
std::int32_t data = 0;
if (communicator.rank() == 0)
  data = 1;

auto status =
  mpi::future(communicator.immediate_broadcast(data, 0))
  .then([&] (mpi::future f)
  {
    auto status = f.get();
    if (communicator.rank() == 1)
      data++;
    return communicator.immediate_broadcast(data, 1);
  })
  .then([&] (mpi::future f)
  {
    auto status = f.get();
    if (communicator.rank() == 2)
      data++;
    return communicator.immediate_broadcast(data, 2);
  })
  .get(); // data == 3 in all ranks.
```
Listing 2. The requests returned from non-blocking calls can be cast into futures, which can be chained using .then() to express asynchronous sequential operations.

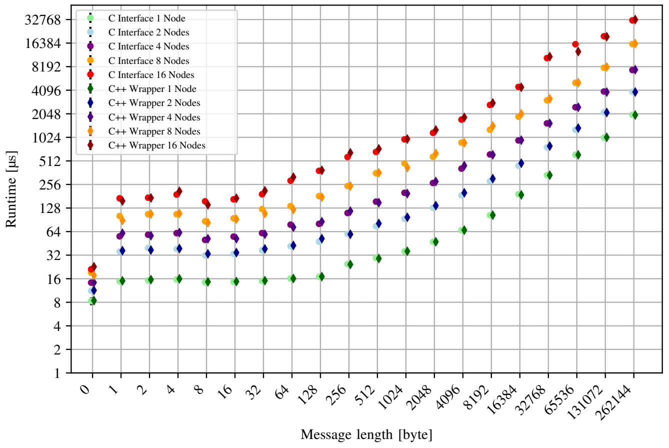

Fig. 1. The benchmark results. The runtime performance of the C and the C++20 interfaces for varying node counts and message lengths.

III. Performance

We measure the performance of the library and compare it to equivalent functionality implemented with the raw C interface. We utilize mpiBench [9], which measures the runtime of 11 MPI operations for varying message lengths. The benchmarks have been adapted to use our interface.

The experiments are controlled by three variables: The *interface* varies as C (using the original mpiBench) or C++20 (using the adapted version). The *message length* varies as 2^n where $0 < n < 18$. The *node count* varies as $1, 2, 4, 8, 16$. Measurements are taken for each combination of the three variables. Each measurement is repeated 10 times and averaged.

The experiments are ran on the RWTH Aachen CLAIX-2018 compute cluster. Each node is equipped with 2 Intel Xeon Platinum 8160 Skylake processors with 24 cores at 2.1 GHz. The network is provided by a high-speed RDMA Omni-Path interconnect. The nodes are exclusively reserved for the benchmarks to eliminate effects due to resource consumption of other processes.

The results are shown in Figure 1. Each data point represents the geometric mean over the 11 MPI operations. The slight variances in runtime could be attributed to network traffic which applies even in exclusive mode. The results of the two implementations do not show recognizable patterns that indicate a disparity in performance.

IV. Conclusion

We have presented a modern C++ interface for MPI and demonstrated that its performance overhead is negligible in comparison to the raw C interface. We continue to incorporate the additions and changes that are proposed as part of the 4.1 and 5.0 specifications as they are becoming available. For further detail, we refer the reader to the source code, distributed under the BSD 3-Clause license, accessible at https://github.com/vrgrouprwth/mpi.

REFERENCES

[1] M. Rüfenacht, D. Schafer, A. Skjellum, and P. V. Bangalore, "MPIs Language Bindings are Holding MPI Back," *ArXiv*, vol. abs/2107.10566, 2021. [Online]. Available: https://arxiv.org/pdf/2107.10566.pdf

[2] D. Gregor and M. Troyer, "Boost MPI," 2005. [Online]. Available: https://www.boost.org/doc/libs/master/doc/html/mpi.html

[3] H. Bauke, "MPL - A message passing library," 2015. [Online]. Available: https://github.com/rabauke/mpl

[4] S. Ghosh, C. Alsobrooks, M. Rüfenacht, A. Skjellum, P. V. Bangalore, and A. Lumsdaine, "Towards Modern C++ Language Support for MPI," in *2021 Workshop on Exascale MPI (ExaMPI)*, 2021, pp. 27–35. [Online]. Available: https://ieeexplore.ieee.org/document/9652833

[5] MPI Forum, "MPI Issues Repository - #288: What features do users need from an MPI C++ interface?" 2020. [Online]. Available: https://github.com/mpi-forum/mpi-issues/issues/288

[6] ——, "MPI: A Message-Passing Interface Standard Version 4.0," Jun. 2021. [Online]. Available: https://mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[7] The Standard C++ Foundation, "C++ Core Guidelines," 2015. [Online]. Available: https://github.com/isocpp/CppCoreGuidelines

[8] A. Polukhin, "Boost PFR," 2016. [Online]. Available: https://www.boost.org/doc/libs/master/doc/html/boost_pfr.html

[9] A. Moody and H. Subramoni, "mpiBench," 2019. [Online]. Available: https://github.com/LLNL/mpiBench