A C++20 Interface for MPI 4.0

A.C. Demiralp, P. Martin, N. Sakic, M. Krüger, T. Gerrits
RWTH Aachen University, Germany

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.

This work presents a **modern and idiomatic header-only**
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
- the ability to **express MPI requests as futures** with continuations to describe sequential non-blocking communication

Automatic Lifetime Management

- **Managed contructors** take the ownership of MPI objects after instantiation
- **Unmanaged constructors** take in existing MPI objects and don not take ownership by default
- **Copy constructor** when duplication functions are provided by MPI
- **Move constructor** whenever possible

```cpp
mpi::environment environment;
const auto& communicator = mpi::world_communicator;
std::vector<std::int32_t> data_container(3);
if (communicator.rank() == 0) {
    data_container = {1, 2, 3};
    communicator.send(data_container, 1);
}
if (communicator.rank() == 1)
    communicator.receive(data_container, 0);
```
C++ Interface

Code example of MPI setup and sending/receiving a std::vector

Usage of Modern C++ Features

- **std::function** for callbacks additionally enables the usage of lambdas with caputres
- **std::variant** for arguments that take a variety of MPI objects
- **std::optional** for optional parameters and indeterminate results
- **strongly typed enums** based on MPI enums for additional type safety

Reflection and Concepts

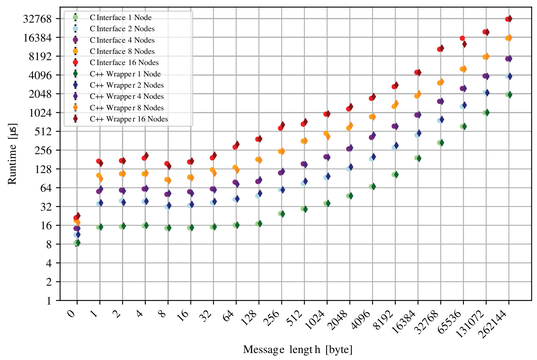
- **Automatic MPI data types** for PODs through non-boost PFR [2]
- **mpi::compliant** concept indicate MPI mappable types (Arithmetic types, enumerations, std::complex specializations and C-Arrays, std::arrays, std::pairs, std::tuple and aggregrate types of compliant types)
- **Sequential contigous containers** can be used for sending multiple values (i.e., std::string, std::span, std::valarray, std::vector)

```cpp
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.
```
C++ Interface

Seamless integration of MPI for use with modern C++ features such as futures

Futures and Error Handling

- **Requests can be std::futures** supporting the concurrency support library of the C++ standard
- **mpi::when_all / mpi__when_any** allows joining and syncing workflows that involve forking
- **Exceptions** are used when MPI functions fail
- **Default error codes** are available through the mpi::error namespace




```cpp
struct particle {
  std::uint64_t id;
  float position[3];
};

particle value = {};
if (communicator_rank == 0)
   value = {42ull, {1.0f, 2.0f, 3.0f}};

MPI_Datatype position_data_type;
MPI_Type_contiguous(3, MPI_FLOAT, &position_data_type);
MPI_Type_commit(&position_data_type);

MPI_Datatype particle_data_type;
std::array<std::int32_t, 2> block_lengths{1, 1};
std::array<MPI_Aint, 2> displacements{0, sizeof(std::uint64_t)};
std::array<MPI_Datatype, 2> data_types{MPI_UINT64_T, position_
data_type};
MPI_Type_struct(2, block_lengths.data(), displacements.data(),
                data_types.data(), &particle_data_type);
MPI_Type_commit(&particle_data_type);

MPI_Bcast(&value, 1, particle_data_type, 0, MPI_COMM_WORLD);

MPI_Type_free(&position_data_type);
MPI_Type_free(&particle_data_type);
```
C Interface

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

particle value = {};
if (communicator_rank == 0)
   value = {42ull, {1.0f, 2.0f, 3.0f}};

mpi::world_communicator.broadcast(value, 0);
```
C++ Interface

Code comparison of a simple messaging setup: Left using the standard MPI C-Interface. Right: Using our C++ Interface

Performance

We measure the performance of the library and compare it to equivalent functionality implemented with the raw C interface on the mpiBench [3] benchmark. Experiments with varying message length and node count are repeated and averaged.
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 other processes.

The results are shown in the plot. 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.

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.

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

or by scanning this QR code:

Acknowledgements

The authors gratefully acknowledge the German Federal Ministry of Education and Research (BMBF) and the individual state governments for supporting this work as part of the NHR funding.

References

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

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

[3] A. Moody and H. Subramoni, “mpiBench,” 2019. [Online]. Available: https://github.com/LLNL/mpiBench