Concepts for Designing Modern C++ Interfaces for MPI

C. Nicole Avans[1], Alfredo A. Correa[2], Sayan Ghosh[3], Matthias Schimek[4],
Joseph Schuchart[5], Anthony Skjellum[1], Evan D. Suggs[1], and
Tim Niklas Uhl[4(✉)]

[1] Tennessee Technological University, Cookeville, Tennessee, USA
`{cnavans42,askjellum,esuggs}@tntech.edu`
[2] Lawrence Livermore National Laboratory, Livermore, California, USA
`correaa@llnl.gov`
[3] Pacific Northwest National Laboratory, Richland, Washington, USA
`sg0@pnnl.gov`
[4] Karlsruhe Institute of Technology, Karlsruhe, Germany
`{schimek,uhl}@kit.edu`
[5] Stony Brook University, Stony Brook, New York, USA
`joseph.schuchart@stonybrook.edu`

Abstract. Since the C++ bindings were deleted in 2008, the Message
Passing Interface (MPI) community has recently revived efforts in build-
ing high-level modern C++ interfaces. Such interfaces are either built
to serve specific scientific application needs (with limited coverage to
the underlying MPI functionality), or as an exercise in general-purpose
programming model building, with the hope that bespoke interfaces can
be broadly adopted to construct a variety of distributed-memory scientific
applications. However, with the advent of modern C++-based hetero-
geneous programming models, GPUs and widespread Machine Learning
(ML) usage in contemporary scientific computing, the role of prospective
community-standardized high-level C++ interfaces to MPI is evolving.
The success of such an interface clearly will depend on providing robust
abstractions and features adhering to the generic programming principles
that underpin the C++ programming language, without compromising on
either performance or portability, the core principles upon which MPI was
founded. However, there is a tension between idiomatic C++ handling
of types and lifetimes and MPI's loose interpretation of object lifetimes/
ownership and insistence on maintaining global states.
Instead of proposing "yet another" high-level C++ interface to MPI,
overlooking or providing partial solutions to work around the key issues
concerning the dissonance between MPI semantics and idiomatic C++,
this paper focuses on the three fundamental aspects of a high-level inter-
face: type system, object lifetimes, and communication buffers, while also
identifying inconsistencies in the MPI specification. Presumptive solutions
can be unrefined, and we hope the broader MPI and C++ communities
will engage with us in productive exchange of ideas and concerns.

Keywords: Message Passing Interface · C++ · Concept-based Inter-
face.

1   Introduction

Many modern C++ projects rely on MPI extensively, but MPI cannot natively handle most C++ data structures or constructs [19]. We plan to derive a basic C++ interface that takes into account new features and capabilities of C++ without compromising either performance or portability. This paper seeks to derive key concepts and design considerations to promote the creation of a full modern MPI C++ interface, while highlighting issues complicating this endeavor.

C++ bindings were included in the MPI standard, but were deprecated by MPI 2.2 [17]. These bindings have been entirely removed in version 3.0 (2012), since they only added minimal functionality over the C bindings while adding significant maintenance complexity to the MPI specification [16]. C++ has changed significantly since the deprecation of C++ bindings in 2009, requiring a new look at its capabilities and the feasibility of a MPI C++ interface.

There are no guidelines now for developing efficient modern C++ interfaces over MPI (it is impossible to successfully enforce recipes such as C++ Core Guidelines [26] without standardizing the behavior of underlying MPI objects, which is usually left up to concrete implementations). Here, we expand on some general considerations for designing C++ interfaces over MPI in good faith, without imposing ad hoc rules that might limit productivity, performance, and/or portability.

Build on existing ideas. We consider, re-evaluate, formalize and extend upon ideas presented in existing language bindings (Section 2).

Derive a C++ representation of MPI's object model. We discuss how to bridge the gap between MPI objects and C++, particularly how to handle their life cycle and formalize mutability in terms of constness (Section 3.1).

Enabling type safety in MPI. The MPI C API currently does not utilize type information provided by the language. In Section 3.2, we categorize C++ types and how these different classes can support type safe compatibility with MPI.

Contiguous ranges as first class communication buffers. Section 3.3 describes rules defined through C++-20 concepts, which specify which kind of C++ containers are directly supported by MPI for communication operations. This enables direct support for many STL containers and provides a powerful interface to derive more complex abstractions. We also describe how to handle data ownership, which is necessary to conform to the best practices of resource management in modern C++ (namely, RAII) in Section 3.4.

Idiomatic error handling. We describe how C++ can enhance MPI error handling via compile-time checking and exceptions in Section 3.5.

These are the major aspects and principles to which a future C++ MPI interface should adhere. Rather than proposing a complete standardized interface, which risks repeating the shortcomings of the removed bindings, our primary goal is to define *semantic guidelines* and *core conceptual interfaces* that align idiomatic C++ with MPI. These concepts could serve as the foundation for a MPI standard side document. Standardizing concepts rather than concrete APIs offers the advantage of long-term maintainability and adaptability as both MPI and the C++ language evolve. A C++ interface could introduce additional

quality-of-life features to enhance productivity and provide safer abstraction, but these often introduce overheads that conflict with MPI's performance-portability while increasing the complexity of a specification. Still, we outline major ideas and provide suggestions on how they could be aligned to our previously introduced design considerations. Particularly, we focus on serialization of more complex data types (Section 4.1), first steps towards automatic life cycle handling of more complex MPI data types with direct mappings to C++ (Section 4.2) and how to increase MPI programmer's productivity by a sane set of defaults (Section 4.3).

In Section 5, we conclude our discussion with suggestions on how the MPI standard could support language interface designers (not limited to C++) by carefully extending the MPI specification.

2   Existing C++ Interfaces over MPI

Since the removal of the "official" C++ MPI interface, a large number of third-party library interfaces have emerged, joined by efforts to bridge the gap between MPI implementations, C++, and high-level performance portability frameworks (e.g., Kokkos). We will briefly outline notable works and their distinctive features in the following, as they serve as inspirations and proofs of concept for the technical discussion in the rest of this paper.

The `mpl` library [3] is a C++-17 based, header-only library meant to provide an easy to use MPI interface for C++ developers after the deprecation and removal of the C++ API in MPI 3.0. Recent work [10] have extracted communication-specific interfaces from the main `mpl` codebase (reducing about 4K LoC in main `mpl`), and consider it as a prototypical modern C++ interface for studying MPI-specific language bindings. To support bulk communication, `mpl` uses built-in `mpl::layout` class to manage derived data types.

Boost.MPI [13] offers a near one-to-one mapping of MPI-1 via free functions using communicators. Dynamic types are handled as skeletons containing address info, which must be updated if data is relocated (e.g., invalidated iterators).

B-MPI3 [7] wraps MPI-3 with a C++ interface based on communicators and member functions, emphasizing const-correctness and iterator-based ranges. It uses compile-time strategies to select communication methods: contiguous ranges use direct C-MPI calls; non-contiguous ones are copied; unsupported types are serialized via Boost.Serialization.

RWTH-MPI [9] is a C++ interface for MPI that supports contiguous STL containers as send/receive buffers and offers overloaded MPI procedures with automatic parameter inference. For custom types, it can auto-generate MPI data types using the PFR library [20]. Dynamic-size types are unsupported. While it covers MPI 4.0, its bindings largely mirror the C interface with limited added abstraction or safety.

KaMPIng [27] uses modern C++ features like move semantics to enhance safety, enabling return-by-value and memory-safe non-blocking communication. It constructs MPI data types automatically for STL and custom types when possible and supports compile-time C++ to MPI type mapping. Its named parameter

interface with compile-time defaults facilitates both high-level prototyping and low-level control, keeping the full MPI interface accessible.

The Enhanced Message Passing Interface (EMPI) [4] is based on modern C++, which is built on top of a customized version of Open MPI, eliding runtime checking overheads to directly map EMPI objects to low-level MPI objects within the Open MPI implementation. To enable RAII, EMPI proposes a *program context* that wraps the MPI environment, and specializes MPI group from the context for abstracting communication and synchronization.

Kokkos Comm [2] utilizes modern C++ language features to provide more intuitive support of inter-node communication of Kokkos Views. Templates provide the needed flexibility to adapt to diverse data layouts and memory spaces.

MPI Advance [5] provides lightweight libraries that complement available system MPI installations to leverage tuned performance while also implementing support for the newest features from the MPI standard (e.g., partitioned communication) and additional capabilities and optimizations beyond the current scope of the standard.

3   High-Level Design Considerations

Existing work on designing C++ interfaces to MPI has brought up a huge amount of interesting design concepts and ideas, but most of the time they are hidden behind the details of the actual implementation. In this section, we will take a step back and clearly define the underlying semantic concepts of modern and idiomatic C++ MPI interface and how MPI's design can be made "compatible" with the C++ language. These concepts can then be used by implementers to design a concrete interface.

We start by defining how MPI objects map to C++ (Section 3.1). Then we focus on how to model the actual data involved in communication, in terms of types (Section 3.2) and the actual memory involved (Section 3.3). We then discuss how combining C++'s ownership model and MPI to obtain additional memory safety (Section 3.4) and how to handle errors in an idiomatic way (Section 3.5).

3.1   Mapping the MPI Object Model to C++

MPI introduces a range of *MPI objects*, such as communicators, data types, and requests, which are represented in the C API using *opaque handles*. While this is the only viable approach in C, it hides key semantic properties of these objects, including ownership, lifetime, and identity. In contrast, C++ offers native language features that allow these properties to be expressed explicitly and safely.

For more than a decade, C++ MPI library designers have advocated for representing MPI objects as first-class C++ objects that act as proxies for the implementation-defined objects behind the handles [23].

It is worth noting that most MPI procedures operating on these objects can be expressed either as member functions or as free functions. Both choices are equally valid, and this decision is largely independent of the conceptual model

discussed here. In this work, we present communication operations as member functions, without advocating for one approach over the other.

Modern C++ idioms—particularly RAII (Resource Acquisition Is Initialization) [25, 16.5][26, E.6] and move semantics—naturally support resource management: MPI objects can be constructed via `MPI_*_create` in class constructors and released via `MPI_*_free` in destructors, with ownership safely transferable through move operations. However, the original MPI design deviates from this model because of its reliance on global state and implicitly managed global objects—an approach discouraged by the C++ Core Guidelines [26, I.3,I.22]. For example, global communicators such as `MPI_COMM_WORLD` are automatically initialized via `MPI_Init` and cannot be explicitly created or freed. Further, MPI prohibits reinitialization after `MPI_Finalize`, limiting modular or library-based usage patterns.

Sessions, Groups, and Communicators MPI 4.0 addressed some of these limitations by introducing the *Session Model*, an alternative to the traditional World Model for process management. Although originally designed to support better isolation in multi-threaded and multi-component environments, the session model aligns well with modern C++ object-oriented design. In this model, no global communicators are predefined; instead, communicators must be explicitly created from process sets via user code. This makes ownership explicit and fully under user control, enabling clean integration with RAII-based designs as shown in Fig. 1. For example, communicators and groups are constructed and destroyed as part of their enclosing scope.

```
mpi::session session{};
mpi::group group = session.group_from_pset("mpi://WORLD");
mpi::communicator comm{group}; // create communicator from group
comm.send(...);
```

Fig. 1. Example of the proposed object model using the MPI Session Model

const-Correctness of MPI Objects The concept of constness is important for both the compiler and, more crucially, libraries, as it indicates variable mutability and restricts modifications in certain contexts. This applies straightforwardly to the data being communicated: For instance, a receiving buffer cannot be `const`, which already enhances the program's safety, which we discuss in detail in Section 3.3. In a less obvious way, constness can also be applied to library objects of a MPI C++ interface, such as communicators, data types, and requests and procedure calls on them, making it even more important to discuss here.

When creating a C++ wrapper for MPI or any C interface that does not account for constness, adding the keyword `const` requires a good grasp of the interface semantics and precise knowledge of the implementation (internal mutation).

Without explicit guarantees from the MPI standard specification, it is difficult to gather sufficient information to determine whether a particular operation can be marked as `const`.

We want to illustrate this for MPI communicators. Most procedure calls leave a communicator in the same state as before, which would *naively* allow them to be specified as `const`. Conversely, one could argue that posting a message is not the same as not having posted it; in the former case, the message can be received on the other side, while in the latter, a receive operation may hang. We think this is a strong argument to mark communication calls as non-const. But this can still be challenged for procedures that send and receive in a single call (such as `send_recv` or collectives). [1]

To resolve these cases, we invoke a modern interpretation (post C++11): Since the communicator is likely undergoing internal mutation during any non-trivial operation, it is reasonable to conclude that most communication procedures should be marked as non-`const` at the communicator level.

```cpp
class mpi::communicator {
    communicator(communicator const& other) = delete;  // no copy-constructor
    ...
    auto duplicate() /*non-const*/ -> communicator;
    ...
    auto send(/*const data*/ ...) /*non-const*/;
    auto receive(/*mutable data*/ ...) /*non-const*/;
    auto broadcast(/*mutable data*/ ...) /*non-const*/;
    ...
    auto size() const;  // most likely can be marked const
                        // in a reasonable implementation
};
```

Fig. 2. Interpretation of const-ness in a MPI C++ interface

If we adhere to the conclusion that communication procedures should not be marked `const`, an important and somewhat unexpected result follows: a communicator class should *not* have a duplicate mechanism that is const on the original communicator. Incidentally, there will be no canonical copy constructor (taking original communicator as a constant reference). This is consistent, since even communicators duplicated from each other cannot receive or complete operations initiated in another communicator. Thus, two communicators are never copies of each other; they are at most alternative virtual fabrics for subsequent communication. Fig. 2 summarizes our discussions about the communicator class which also align with the interpretation of the B.MPI3 [7] interface.

For other parts of the library, the situation might be simpler. For example, most datatype manipulation is likely to take advantage of `const`, since datatype manipulation usually generates new types instances from other types instances.

---

[1] Note that this is a fundamental problem, not exclusive to const-qualified member functions; if the procedures are free functions the questions will still stand regarding the constness of the communicator argument parameter.

In summary, a key problem is that the MPI standard says little about the mutation of MPI "objects" (communicator, data types or request objects), which is further complicated by the existence of a global mutable state at the level of the environment. In most cases, the internal mutation seems only to be implied by common knowledge, which makes the decision on the correct usage of `const`-ness in a C++ interface difficult.

Design and conventions on this issue, have important ripple effect in the design of C++ MPI programs, specifically C++ classes that contain communicators and other MPI objects, which is common in programs written at a high level of abstraction [1,11]. Classes that use their own internal communicator even for non-mutating operations would require mutation of the communicator, internally at least (i.e., `mutable` attribute, and possibly synchronization).

3.2   Modeling and Mapping Types

Applications using MPI use a variety of data types that need to be communicated. The MPI standard distinguishes between *basic datatypes* and arbitrarily complex *derived datatypes*, which can be recursively constructed from other data types using type constructors (`MPI_Type_create_*`). C's lack of type introspection features forces users to always pass the type explicitly to a communication call, which is both tedious and error-prone, since type definitions need to be kept in sync with the actual data layout. Fortunately, for many C++ types, there is a one-to-one mapping to MPI data types. C++ defines a set of *fundamental types*: `void`, `std::nullptr_t`, *integral types* (including integers, character types and `bool`) and *floating-point types*. For integral and floating-point types, there exist matching predefined *basic data types* in MPI.

Using template-metaprogramming, a C++ MPI interface can therefore deduce an MPI type directly from a data buffer (as defined in Section 3.3), in case its underlying `value_type` is fundamental. This approach is implemented by all major MPI C++ bindings [3,7,13,27]. More complex types require explicit creation and a subsequent commit step, and have to be freed before MPI is finalized. To enable proper cleanup we again use RAII and represent data types as `mpi::datatype` objects, which support move construction and assignment and free the data type when the destructor is called.

To prevent users from using uncommitted data types in communication, we propose to encode the commit information as part of the type, similar to the approach of *rsmpi* [24]. The function `mpi::commit` takes an rvalue `mpi::datatype` object, and converts it to a `mpi::committed_datatype` as shown in Fig. 3. Accordingly, communication calls only accept data type objects of this type.

In addition to fundamental C++ types with direct mappings to predefined MPI basic types, there exists a broader class of C++ types that can still be safely mapped to MPI data types. These are precisely the types classified *trivially copyable* by the C++ standard. A type is trivially copyable if its binary representation can be safely copied byte-by-byte (e.g., via `memcpy`), without violating language rules and invoking undefined behavior. More concretely, an object of such type can be copied to an array of `char`, `unsigned char`, or `std::byte`, and then back into

```
struct MyType {
  int a;
  std::array<int, 3> b;
  double c;
  char d;
};
mpi::datatype type = mpi::datatype::for<MyType>();
mpi::committed_datatype struct_type = mpi::commit(std::move(type));
```

Fig. 3. Example for constructing a type for a C++ type and committing it

another object of the same type, which will hold the same value as the original. This property is essential for MPI communication: when transmitting data across processes, the memory content of a variable must be sent as a sequence of raw bytes over the network and reconstructed correctly on the receiving end. If the type is not trivially copyable, this round-trip may not preserve the original value or may result in undefined behavior. Thus, restricting to trivially copyable types ensures that the MPI type correctly matches the actual in-memory layout and semantics of the transmitted data.

These types can be safely supported by automatically constructing the corresponding MPI datatype using a compile-time reflection mechanism to call the correct MPI type constructors. This ensures that the C++ type is compatible with MPI's type system and enables safe type handling, even on heterogeneous systems. Although C++ currently lacks native language support for such reflection, concrete proposals are in place to introduce compile-time type reflection in C++26 [22]. In the meantime, third-party libraries such as Boost.PFR [20] can be used to implement this feature, although they have some limitations, for example for types using inheritance or private members.

Beyond fundamental and trivially copyable types, there remains a broad class of other (potentially user-defined) C++ types that do not have a direct or automatically derivable mapping to an MPI data type. For these types, the correspondence between memory layout and type semantics cannot be safely inferred by the prospective C++ MPI interface. Consequently, it becomes the user's responsibility to explicitly define how the type is laid out in memory and how that layout corresponds to an appropriate MPI datatype. While the handling of fundamental and trivially copyable types can be automated to reduce programming errors and ease MPI development, a prerequisite for such automation is a clear conceptual model of the data involved in communication. The core abstraction in this model is the *data buffer*, which we introduce in the following Section 3.3.

3.3   Modeling Memory Involved in MPI Communication

We now explore how a C++ MPI interface can offer idiomatic abstractions for the data sent or received in MPI communication operations. In MPI, this data is described in terms of a pointer to a memory region, an (MPI) datatype, the number of elements to be sent, and (for some collective operations) their displacements. While this number of parameters makes MPI flexible, it also leads to verbose

function calls. Further, the flexibility of MPI is only required in a few use cases and results in unnecessarily complicated code in other cases [27].

Particularly when using standard library containers such as `std::vector<T>` with the MPI C API, users have to access underlying raw pointers and sizes of the containers and pass them to MPI individually, while the actual vector object already comprises the whole send data context. This is both non-ergonomic to use and conflicts with the C++ Core Guidelines [26, I.13]. Most existing MPI C++ libraries therefore introduce support for a subset of standard library containers [3,7,9,13,27] as their core abstraction feature. However, this has certain shortcomings. First, this is often defined ad-hoc only including a fixed set of containers [9,13]. Second, design decisions within the approaches of supporting (standard library) containers introduce hidden, additional overheads through memory allocations or additional MPI calls [3,13,27]. For example, in an `MPI_Recv` call, Boost.MPI resizes the container into which it receives the data to match the size of the incoming message. If the user does not want this, they must instead pass raw pointers again. The KaMPIng library internally invokes `MPI_Probe` when the user issues a receive operation without a `count` argument. Such unexpected overheads are not desired from the perspective of a standardized C++ interface.

Therefore, in the following, instead of reiterating the specific approaches of existing MPI wrappers and defining how a particular container is "fitted" to support MPI, we will define a small set of underlying rules and categories for memory involved in communication in terms of C++ *concepts*. Concepts are a (modern) C++ language feature for specifying constraints on (custom) types. C++ provides easy-to-use mechanisms to check that types satisfy a given concept at compile time. This gives us well-defined semantics on which kind of containers can be used directly with MPI, deducing size and type information safely from C++, without imposing any additional overhead. Additionally, C++ objects not satisfying required concepts result in easy-to-understand error messages, as opposed to previous often very verbose and complicated error messages caused by template metaprogramming errors [8].

For each send or receive buffer involved in a communication call, MPI's C API expects a pointer to a contiguous memory location, a count argument and an MPI data type. C++ already provides type information, and automatically matching types is crucial for type safety. A high-level standardizable C++ interface, can use type introspection to provide such type safety, which we detail in Section 3.2. For now, let us assume that we can always deduce an MPI data type from a provided container. We call this basic building block of MPI communication (memory location, count and type) a *data buffer*. In the following, we describe a minimal interface of data buffer objects in terms of C++ concepts.

A data buffer object has to satisfy the concept `std::range`; that is, it must expose an iterator to the beginning (and to the end) of its underlying storage. Additionally, data buffers must satisfy the following (built-in) concepts:

```
namespace mpi {
  // Concept ensuring that t exposes MPI data type information
  template <typename T>
  concept Typed = requires(T t) {
      { mpi::datatype(t) } -> std::same_as<committed_datatype const&>;
  };

  template <typename T>
  concept DataBuffer =
      std::ranges::contiguous_range<T> &&
      std::ranges::sized_range<T> &&
      (std::is_fundamental_v<typename T::value_type> || Typed<T>);

  // Concept for data buffers storing send data
  template <typename T>
  concept SendDataBuffer =
      DataBuffer<T> &&
      std::ranges::input_range<T>;

  // Concept for data buffers encapsulating data to be received
  template <typename T>
  concept RecvDataBuffer =
      DataBuffer<T> &&
      std::ranges::output_range<T, typename T::value_type>;
}
```

Fig. 4. Concept definition for a data buffer

- `std::ranges::contiguous_range`, that is, its underlying memory has to be contiguous in memory. This captures MPI's requirement that data which is communicated must reside in *sequential storage* [2].
- `std::ranges::sized_range`, that is, a data buffer exposes a `size()` function returning the number of its elements.
- `Typed`, that is, either the data buffer exposes a constant reference to a committed MPI data type or the data buffer's `value_type` is a fundamental type (for which a direct mapping to predefined MPI data types exists and can directly be used if not overwritten by the user).

An example of how these constraints can be expressed in plain C++ is given in Fig. 4, which defines the concept `mpi::DataBuffer` in terms of the three concepts described above. A convenient property of this data buffer definition is that many C++ standard library containers like `std::vector` or `std::array` storing fundamental C++ types such as `char, int, double, ...` already satisfy the `mpi::DataBuffer` concept. This also holds for lightweight, non-owning views such as `std::span`. Hence, they can be directly used in an MPI C++ interface adhering to these principles.

For more complex cases, for example, to communicate a container of a custom C++ datatype which is `trivially_copyable`, we first have to construct and commit such a datatype (see Section 3.2) and then pass it to the data buffer. For communicating data using complex derived datatypes without a one-to-one

---

[2] An exception to this requirement is the usage of `MPI_BOTTOM` which relies on absolute memory addresses to describe data and should be treated separately.

mapping to the underlying C++ type, we not only have to construct and commit an MPI datatype but also pass the number of corresponding elements to the data buffer. For these more complex cases, there is no direct C++ standard library support but an MPI C++ interface could easily provide a lightweight *adapter* with multiple constructors not only accepting a contiguous `std::range`, but also a count and a `mpi::committed_datatype` parameter.

This also makes this interface extensible to support more complex containers from third-party libraries like `Kokkos::View` or `thrust::device_vector`, backed by memory located on accelerators such as GPUs. Therefore the data buffer concept naturally extends to the notion of *memory allocation kinds* [18], providing direct support for accelerator-aware MPI on C++ containers backed memory located on GPUs or other accelerators.

Figure 5 illustrates exemplary communication calls showcasing the different cases discussed above. Note that we do not suggest a concrete API or function signatures but rather want to show the underlying concept of using data buffers as an abstract description for communication data.

Orthogonal to the `mpi::DataBuffer` concept presented so far, we additionally have to model that data buffers describing send data have to be *readable* and data buffers for receive data have to be *writable*. This can be expressed using the (built-in) concepts, `std::ranges::in/output_range` (see Fig. 4).

To conclude, in many (simple) cases, this approach leads to clean and less convoluted code with direct C++ standard library integration. For more complex cases, we end up with the same number of parameters as in the C API, as we need to leverage its full flexibility. However, in all scenarios, we gain clear semantics directly documented *in* source code by using C++ concepts.

```cpp
std::vector<int> v = ...
comm.send(v, /*additional parameters*/);

std::vector<MyType> v = ... // static custom type defined in Fig. 3
mpi::datatype type = mpi::datatype::for<MyType>();
mpi::committed_datatype type_c = mpi::commit(std::move(type));
comm.send(mpi::buffer_adapter(v, type_c), /*additional parameters */);

std::vector<char> v = ...
int count = ...
mpi::datatype complex_type = ... // complex derived type manually defined by user
mpi::committed_datatype complex_type_c = mpi::commit(std::move(complex_type));
comm.send(mpi::buffer_adapter(v, complex_type_c, count), /*additional parameters */);

int local_count = ...;
comm.reduce(mpi::single_adapter(local_count), ...); // support for non-range single arguments
```

Fig. 5. Exemplary communication calls using the data buffer concept when MPI procedures are implemented as member functions.

Extending the Data Buffer Concept. In collective communication calls with varying counts (such as `MPI_Alltoallv`), MPI requires not just a single *count* parameter

but a separate count for each MPI process (aka rank). Additionally, displacements can be specified for each rank. To reflect these semantic changes, we have to extend the `mpi::DataBuffer` concept as previously defined (Fig. 4) and define *irregular* `mpi::DataBuffers`. These buffers are still based on the concept `std::ranges::contiguous` and have to expose MPI datatype information. Additionally, they also have to expose `size_v()` and `displacements()` functions, which return the respective counts and displacement data. For brevity, we refrain from providing an example definition of this additional concept. Again, simple lightweight adapters can be provided for standard library and similar custom containers. Similarly, scalar arguments (such as a single `int` or `bool` in a reduction) can be easily supported via lightweight adapters, which implicitly return `size` 1 and implement the data buffer concept (Fig. 5).

3.4   Ownership and Non-Blocking Communication

Using the data buffer concept instead of raw pointers already offers benefits such as an ergonomic interaction with standard library containers, improved type safety and the potential to prevent out-of-bounds accesses. Another important advantage of this object-oriented data handling is that it allows us to model a proper concept of ownership. In C++, if an object *owns* a resource—such as heap-allocated memory—the object is responsible for releasing this resource when it goes out of scope. This is usually achieved by placing the clean-up code in the object's destructor and the key idea behind the RAII idiom. A data buffer owning its underlying memory is therefore responsible for freeing this memory once it goes out of scope, thus preventing memory leaks. This plays well with C++'s move semantics. During move-construction or move-assignment, ownership of the underlying resource is transferred from the source to the destination object. Using this mechanism it is for example possible to move a (receive) data buffer to an MPI call as shown in Fig. 6. Then the MPI call receives data into the provided buffer, which it now owns. Once the communication is completed, it returns the updated receive buffer by value resulting in a clean and idiomatic pass through of the underlying memory resource. This data flow allows us to avoid the use of *out* parameters that are discouraged in C++ [26, F.20].

In terms of library interface design, this "pass through" of data can be achieved by considering the *value category* of data buffers passed to communication calls. We propose that a C++ interface should explicitly distinguish between arguments passed as r- or l-values: If a data buffer is an *r-value*, the data buffer is moved to the MPI operation, and returned to the caller as a return value of the function call. If a data buffer is an *l-value*, it is passed by reference to the operation, ownership is not transferred, and it is not returned. For an example see Fig. 6.

Further, (moving) ownership is particularly useful for enhancing the safety of non-blocking communication. In MPI, a call to a non-blocking communication operation *initiates* the operation but does not *complete* it. Instead, it returns a request handle, which the user can test for completion via `MPI_Test` or wait (blocking) on via `MPI_Wait`. This return of control between initiation and completion of the underlying call introduces a potential source of programming errors

since MPI semantics require that any buffer involved in a non-blocking operation remain unmodified by the user until the operation has completed.

A robust solution to this problem is to transfer ownership of data buffers involved in non-blocking communication calls to the C++ interface, which forward it to request object as show in Fig. 6. The request object is conceptually similar to `std::future` [3]: Users can only access the data upon completing the request via `wait` or `test`, which move the data back to the caller, either directly by value in the former case, or encapsulated in `std::optional` in the latter. This way, invalid accesses to data involved in non-blocking communication can be prevented entirely through library semantics. This idea was first introduced and implemented in the MPI wrapper KaMPIng [27].

```cpp
std::vector<int> recv_buf = {...};
recv_buf = comm.recv(std::move(recv_buf), ...); // ownership transferred to call
                                                // and back to caller
comm.recv(recv_buf, ...); // buffer only captured by reference, and nothing returned

// Non-blocking communication with ownership transfer
mpi::request<std::vector<int>> req = comm.irecv(std::move(recv_buf), ...);
recv_buf = req.wait(...);
std::optional<std::vector<int>> result = req.test(...);
```

Fig. 6. Transferring ownership through C++'s move semantics

3.5   Error Handling

MPI notifies users of errors by returning error codes from almost every function defined by the MPI standard. Although this is the most common way to handle errors in C, this does not fit well with modern C++. Most current MPI C++ interfaces either ignore errors completely or encapsulate the returned error code in an exception and throw it. This has a shortcoming: MPI makes no distinction between failures, which may be recoverable, such as insufficient buffer space or node failures (when using a ULFM-enabled MPI implementation [6]) and usage errors, such as providing invalid parameters, which cannot be resolved. The strategy of converting all returned errors to exceptions is opposed to the C++ Core Guidelines [26] which give the following suggestions on error handling: (E.2) "Throw an exception to signal that a function can't perform its assigned task," and (P.5) "Prefer compile-time checking to run-time checking." Guided by this, we propose the following strategy (implemented by KaMPIng [27] and B.MPI3 [7]):

First, usage errors, such as invalid parameter combinations or invalid types, are handled at compile time via `static_assert`. Since C++ template meta-programming is notorious for complex, hard-to-read compiler errors, we try to

---

[3] Using `std::future` to provide a safe interface for non-blocking communicating is not possible, as they are tied to asynchronous progress happening in the background, which the MPI standard does not guarantee.

ensure that compile-time assertions fail early and provide helpful human-readable error messages. Looking ahead, C++-26's enhancements to `static_assert` will enable us to supply constant-expression diagnostic messages, potentially leveraging `std::format`, so that compile-time checks can convey rich contextual information.

For invariants that can only be verified during execution, we use a layered assertion system: We verify invariants ranging from lightweight checks to assertions involving additional communication. These can be disabled level-by-level at compile-time, encouraging developers to use exhaustive checking while writing and testing code, yet permitting a lean, high-performance configuration for production builds. By integrating error-handling controls directly into the C++ interface—rather than relying on rebuilding the MPI library itself in a "debug" mode—we provide an ergonomic, flexible framework. If an assertion is enabled and fails, we call `MPI_Abort` because execution can not safely continue.

Errors that may be recoverable are signaled to the user. This can either be done by throwing an exception, or by returning `std::expected` from MPI procedures. C++-23's `std::expected<T, E>` offers a lightweight, value-based alternative to exceptions, by encapsulating either a valid `T` or an error `E` in one object. Using `std::expected` makes error handling explicit, avoids hidden blocking operations inside distant `catch` handlers, and allows for easy error chaining and propagation. Together with the return-by-value-based design proposed in Section 3.4, this extends the concept of error codes in MPI's C interface, by only providing access to the return value of a procedure in case of success.

4   Considerations Beyond a Preliminary C++ Interface

In the following, we discuss additional features which improve the usability of preliminary C++ MPI language bindings. These features are perhaps beyond the scope of initial standardization, since they introduce additional overhead on top of the current MPI specification, such as allocation or communication under the hood. Nevertheless, they make working with MPI from C++ easier and often safer. In the following, we discuss how to easily communicate data not directly compatible with MPI (Section 4.1), how to improve the handling of type lifetimes (Section 4.2) and to improve programmer productivity and reduce boilerplate code by offering sane defaults to programmers (Section 4.3).

4.1   Serialization

The data buffer concept defined in Section 3.3 provides a type-safe and easy-to-use way to use standard C++ containers and user-defined types with a regular memory layout and a direct mapping to an MPI data type in MPI communication. Commonly used complex C++ types, such as `std::list`, `std::unordered_map`, or polymorphic class hierarchies, do not provide contiguous storage or a well-defined memory representation. These are *dynamic types*: their size or layout may vary at runtime, they may require logic to (re)construct from memory (on both ends of the communication) and may involve intermediate allocations associated

with data copies. While such types cannot be communicated directly using the MPI type system, a user-defined *data buffer* encapsulating packed data data can be used. However, constructing this is cumbersome and error-prone.

Instead, we propose a general serialization mechanism: The solution is to serialize complex types into a contiguous byte buffer (e.g., a `std::vector<char>`), which are directly transmittable via MPI. The deserialization step on the receiving side reconstructs the original object from this representation.

Importantly, serialization libraries such as Boost.Serialization [21] or Cereal [12] let users define what data members of a type should be serialized; that is, what constitutes the object's logical state. The concrete binary layout or packing is handled automatically by the library and is treated as an implementation detail [4]. This cleanly separates the intent of serialization from its mechanics, which is especially useful for complex types. Constructing such a serialized buffer may itself involve MPI communication, for example to exchange message sizes ahead of receiving the actual payload. While this is beyond the scope of the MPI standard, it can be built naturally on top of the previously defined data buffer concept. A user-defined type wrapping the serialization logic and exposing the required interface (contiguous, sized, and typed buffer) integrates seamlessly with the rest of the system.

In summary, even though dynamic types require custom logic and cannot be described via static type traits, the buffer abstraction is general enough to accommodate them. Serialization turns arbitrary C++ objects into something communicable without burdening the MPI interface with additional complexity.

4.2   Safer Usage of Complex Types

The type classification and data buffer concept introduced in Sections 3.2 and 3.3 provide all necessary building blocks to work with complex types. Even if a type is not *directly* supported by MPI, for the large class of trivially copyable types, there still exists a type-safe, direct type mapping. This allows the user to semi-automatically construct a type, and pass it alongside a container to the communication call as a data buffer, as shown in Fig. 5.

But this has shortcomings: Committing an MPI data type is not free, since implementations may optimize the internal representation of a type when it is committed. So in a C++ interface, it is the user's responsibility to commit the type and free it when it is no longer needed (by destructing the type object using the RAII pattern). This introduces additional problems, since now the user has to keep the type around all the time, and the type correspondence between the container and the data type object is lost. To improve usability here, we suggest the idea of a *type pool* that allows to store committed data types and to perform a lookup based on type information.

Users then just construct a data buffer from a container of a trivially copyable type and a reference to the type pool and pass it to the communication call. The communication call can then perform a runtime lookup based on

---

[4] Implementations may even avoid byte packing and communicate in chunks.

`std::type_info::hash_code` of the containers value type to obtain the MPI data type used in communication.

There is one caveat to this approach: The lifetime of types in MPI is currently not clearly defined. In the session model, the lifetime of types is allowed to span across multiple sessions which suggests that the MPI type system exists orthogonally to MPI initialization. But it also mandates that types may only be created once a session or the world model is initialized, yet users are encouraged to commit types separately for each session.

This inconsistency appears to be unintentional but has not been resolved since sessions were introduced[5]. Fixing this will enable language bindings to MPI, not only limited to C++, to handle types more easily. For example, when type lifetime is tied to sessions, a C++ implementation can associate a session-local type pool with each session object, which can be queried for types without providing a pool explicitly. Types can then be freed automatically when the session is finalized.

4.3   Improving MPI Productivity with Sensible Defaults

Many libraries improve usability features beyond a high-level standardized interface: automatically resizing receive containers, computing missing counts or displacements in collectives with varying counts, or even performing additional communication to determine missing arguments. For example, Boost.MPI [13] and RWTH-MPI [9] offer various overloads for communication calls with default arguments, preventing users from writing a lot of boilerplate code.

KaMPIng [27] goes even further: It chooses an alternative approach inspired by *named parameters*, where parameters passed to a function can be named at the caller site and passed in arbitrary order (as seen in languages like Python).This allows to check for the presence of each parameter and to compute default values only if the respective parameter is omitted, without resorting to many overloads exploring the complete combinatorial explosion of parameters. To avoid run-time overhead, they rely on template meta-programming to only generate the code paths required for computing missing parameters at compile time.

While this is well beyond the scope of a low-level interface due to additionally introduced allocations or communications, concrete implementations building on this interface could introduce such improvements to make writing MPI code easier, less error-prone and more productive, by handling resources on behalf of the user.

5   Conclusions

Standardizing a modern C++ interface is an ambitious—and perhaps impractical—undertaking. Instead, we advocated here for the creation of language support guideline documents, similar in spirit to what this work aims to contribute.

While many desire a truly modern C++ interface for MPI, this paper has laid out the conceptual and technical challenges that need to be addressed before such an

---

[5] https://github.com/mpi-forum/mpi-issues/issues/733

interface can be standardized. Our goal was to clarify these challenges and provide a foundation of design principles that could guide future development and community discussion. To this end, we covered high-level considerations: a C++representation of the MPI object model (Section 3.1), data representation (Section 3.3) and ownership (Section 3.4), modeling and mapping types (Section 3.2), and idiomatic error handling (Section 3.5). We also discussed ideas beyond these core concepts, including serialization (Section 4.1) and usability improvements (Sections 4.2 and 4.3).

Based on these considerations, we identified specific aspects where the MPI standard requires clarification—most notably the lifecycle of user-defined data types, which remains under-specified in the context of sessions and makes automated resource management more difficult. We also support ongoing efforts to improve attribute support across the standard, particularly for sessions and requests, as this is essential for associating higher-level interface constructs with the lifetime of MPI objects (see Issue #664). Finally, while callback support already exists in the standard, not all callbacks currently support user-defined state. Enabling this consistently would improve the design and usability of language bindings (see Issue #839).

We hope that our discussion can serve as a blueprint for other MPI language communities. A prime example for language support is Vapaa [14], a standalone implementation of MPI's Fortran interface built on the C API, which strongly influenced the addition of an MPI Application Binary Interface in the MPI 5.0 standard [15]. Also, many of the mentioned concepts can be applied to or were inspired from the Rust MPI bindings `rsmpi` [24], since modern C++ and Rust share many conceptual similarities. For other languages, it might be helpful to derive their own, independent set of guidelines. This will enable MPI continue to evolve together with popular languages that define the future of high-performance computing.

References

1. Andrade, X., Pemmaraju, C.D., Kartsev, A., Xiao, J., Lindenberg, A., Rajpurohit, S., Tan, L.Z., Ogitsu, T., Correa, A.A.: Inq, a modern gpu-accelerated computational framework for (time-dependent) density functional theory. Journal of Chemical Theory and Computation **17**(12), 7447–7467 (Nov 2021). https://doi.org/10.1021/acs.jctc.1c00562, http://dx.doi.org/10.1021/acs.jctc.1c00562

2. Avans, C.N., Ciesko, J., Pearson, C., Suggs, E.D., Olivier, S.L., Skjellum, A.: Performance Insights into Supporting Kokkos Views in the Kokkos Comm MPI Library. In: 2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops). pp. 186–187 (2024). https://doi.org/10.1109/CLUSTERWorkshops61563.2024.00051, https://github.com/kokkos/kokkos-comm

3. Bauke, H.: MPL - a message passing library (2015), https://github.com/rabauke/mpl

4. Beni, M.S., Crisci, L., Cosenza, B.: EMPI: Enhanced Message Passing Interface in Modern C++. In: 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid). pp. 141–153 (2023). https://doi.org/10.1109/CCGrid57682.2023.00023

5. Bienz, A., Schafer, D., Skjellum, A.: MPI Advance : Open-Source Message Passing Optimizations (2023), https://arxiv.org/abs/2309.07337

6. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of mpi communication capability: Design and rationale. The International Journal of High Performance Computing Applications **27**(3), 244–254 (2013). https://doi.org/10.1177/1094342013488238, https://doi.org/10.1177/1094342013488238

7. Correa, A.A.: B-MPI3 (2018), https://github.com/LLNL/b-mpi3

8. cppreference.com: Constraints - cppreference.com (2025), https://en.cppreference.com/w/cpp/language/constraints.html, accessed: 2025-06-03

9. Demiralp, A.C., Martin, P., Sakic, N., Krüger, M., Gerrits, T.: A C++20 interface for MPI 4.0. CoRR **abs/2306.11840** (2023)

10. Ghosh, S., Alsobrooks, C., Rüfenacht, M., Skjellum, A., Bangalore, P.V., Lumsdaine, A.: Towards modern C++ language support for MPI. In: 2021 Workshop on Exascale MPI (ExaMPI). pp. 27–35. IEEE (2021)

11. Godoy, W.F., Hahn, S.E., Walsh, M.M., Fackler, P.W., Krogel, J.T., Doak, P.W., Kent, P.R., Correa, A.A., Luo, Y., Dewing, M.: Software stewardship and advancement of a high-performance computing scientific application: Qmcpack. Future Generation Computer Systems **163**, 107502 (Feb 2025). https://doi.org/10.1016/j.future.2024.107502, http://dx.doi.org/10.1016/j.future.2024.107502

12. Grant, W.S., Voorhies, R.: cereal – a C++11 library for serialization (2017), http://uscilab.github.io/cereal/

13. Gregor, D., Troyer, M.: Boost.MPI (2005–2007), https://www.boost.org/doc/libs/1_84_0/doc/html/mpi.html, version 1.84

14. Hammond, J.: Vapaa: A standalone implementation of the mpi fortran 2018 module (2023), github.com/jeffhammond/vapaa

15. Hammond, J., Dalcin, L., Schnetter, E., PéRache, M., Besnard, J.B., Brown, J., Gadeschi, G.B., Byrne, S., Schuchart, J., Zhou, H.: MPI application binary interface standardization. In: Proceedings of the 30th European MPI Users' Group Meeting. p. 1–12. EUROMPI '23, ACM (Sep 2023). https://doi.org/10.1145/3615318.3615319, http://dx.doi.org/10.1145/3615318.3615319

16. Message Passing Interface Forum: MPI: A message-passing interface standard (2009), https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

17. Message Passing Interface Forum: MPI: A message-passing interface standard, version 2.2. Specification, MPI Forum (September 2009), http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

18. Message Passing Interface Forum: Memory allocation kinds: A MPI side document (2024), https://www.mpi-forum.org/docs/sidedocs/mem-alloc10.pdf

19. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 5.0 (Jun 2025), https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf

20. Polukhin, A.: Boost.pfr (2016), https://www.boost.org/doc/libs/1_84_0/doc/html/boost_pfr.html

21. Ramey, R.: Boost.Serialization (2002–2009), https://www.boost.org/doc/libs/1_86_0/libs/serialization/doc/, version 1.86

22. Revzin, B., Childers, W., Dimov, P., Sutton, A., Vali, F., Vandevoorde, D., Katz, D.: Reflection for C++26. Proposal P2996R5 P2996R5, ISO/IEC JTC1/SC22/WG21 (2024), https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2996r5.html

23. Skjellum, A., Wooley, D.G., Lu, Z., Wolf, M., Bangalore, P.V., Lumsdaine, A., Squyres, J.M., McCandless, B.: Object-oriented analysis and design of the message passing interface. Concurrency and Computation: Practice and Experience **13**(4), 245–292 (2001). https://doi.org/https://doi.org/10.1002/cpe.556, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.556

24. Steinbusch, B., Gaspar, A., Brown, J.: rsmpi - MPI bindings for rust (2015), https://github.com/rsmpi/rsmpi

25. Stroustrup, B.: The design and evolution of C++. Addison-Wesley (1994)

26. Stroustrup, B., Sutter, H., et al.: C++ core guidelines (2024), https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html

27. Uhl, T.N., Schimek, M., Hübner, L., Hespe, D., Kurpicz, F., Seemaier, D., Stelz, C., Sanders, P.: Kamping: Flexible and (near) zero-overhead C++ bindings for MPI. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2024, Atlanta, GA, USA, November 17-22, 2024. p. 44. IEEE (2024). https://doi.org/10.1109/SC41406.2024.00050, https://dl.acm.org/doi/10.1109/SC41406.2024.00050