# First Steps Towards Designing Modern Language Interfaces for MPI

## The Resurrection of the C++ Interface (?)

MPI Forum Meeting, Sep 30 2025, Charlotte


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

[1]Tennessee Technological University, Cookeville, Tennessee, USA
[2]Lawrence Livermore National Laboratory, Livermore, California, USA
[3]Pacific Northwest National Laboratory, Richland, Washington, USA
[4]Karlsruhe Institute of Technology, Karlsruhe, Germany
[5]Stony Brook University, Stony Brook, New York, USA

# The Rise and Fall of the C++ Bindings

```
void MPI::Comm::Allreduce(
  const void* sendbuf, void* recvbuf, int count,
  const MPI::Datatype& datatype, const MPI::Op& op
) const
```

- Bindings too close to C
- Did not keep up with the evolving standard
- And did not improve productivity

… and predated “modern” C++

# “C and Fortran have great interoperability, why bother with something else?”

A good (higher level) language interface can improve

- Productivity
- Performance
- Usability + Safety

# “Why not just write an open source library on top of the C interface?”

- Plenty of existing C++ bindings
  - Modern C++ moved way ahead of MPI
  - Ad-hoc design, instead of clear interface semantics
  - Either too close to the C-interface, or sacrificing performance for abstraction
  - GPU adoption is still lacking (MPI influenced NCCL, but we need rich device interfaces in MPI)
- Normative interface still useful (see VAPAA w.r.t Fortran, ExaMPI from TTECH) - allows modernization without sacrificing performance
  - But the C interface should not *limit* a C++ interface
- We must define *semantic concepts* to bridge MPI and idiomatic C++
  - A interface should adhere to the principles of a target language
- MPI features which are hard to use correctly from C++ will be used less
  - A good C++ interface has the potential to “sell” new MPI features better

# Core Concepts of a modern C++ Interface

For detailed discussion attend talk on Thursday

## Object Model

- Clear responsibility for freeing resources
- Automatic memory management via RAII
- Move semantic support

```c
MPI_Session session = MPI_SESSION_NULL;
MPI_Session_init(MPI_INFO_NULL,
                 MPI_ERRORS_RETURN, &session);
MPI_Group group = MPI_GROUP_NULL;
MPI_Group_from_session_pset(session,
                            "mpi://WORLD", &group);
MPI_Comm comm = MPI_COMM_NULL;
MPI_Comm_create_from_group(group,
                           "org.example",
                           MPI_INFO_NULL,
                           MPI_ERRORS_RETURN, &comm);
MPI_Group_free(&group);
// ...
MPI_Comm_free(&comm);
MPI_Session_finalize(&session);
```

```cpp
mpi::Session session;
mpi::Comm comm = session.group_from_pset("mpi://WORLD")
                        .create_comm("org.example");
```

## Type Taxonomy

- Automatic (static) type mapping at compile time
- Improve type safety
- Applicable if type fulfills `std::is_trivially_copyable<T>`

```cpp
struct MyType {
  int a;
  std::array<int, 3> b;
  double c;
  char d;
};
```

```cpp
template<>
struct type_traits<MyType> {
  static MPI_Datatype type() {
    MPI_Datatype type;
    // get type and disp for each member
    MPI_Type_create_struct(...);
    MPI_Type_create_resized(...);
    return type;
  }
};
```

# Core Concepts of a modern C++ Interface

For detailed discussion attend talk on Thursday

## Data Buffers as First-class Objects

```
template <typename T>
concept DataBuffer =
  std::ranges::contiguous_range<T> &&
  std::ranges::sized_range<T> && mpi::Typed<T>;
```

```
std::vector<int> v = {...};

comm.send(v, destination);

comm.send(std::span(v.data(), v.size(), destination);

comm.send(std::views::single(42), destination);

comm.send(v | with_type(MY_TYPE), destination);

thrust::device_vector<int> dv;
comm.send(dv | thrust_adaptor, destination);
```

- Enables value semantics
- Safety features for non-blocking for free

## Error Handling

- Catch usage errors at compile-time (if possible)
- No universally preferred way of handling errors in C++
- We are flexible: either exceptions or `std::expected`

# Work in Progress / Future Ideas

## Safe Semantics for One-Sided

```
auto win = mpi::Win<int>(size, comm); // will be freed when
                                       // going out of scope
int read_value = -1;
{ // remote access
  auto remote_lock = win.lock(remote_rank);
  remote_lock.put(index, single_value);
  remote_lock.put(index, buffer); // can be anything implementing
                                  // the data buffer concept 1
  // ...
  remote_lock.flush();
  // ...
  remote_lock.get(other_index,read_value);
} // MPI_Win_unlock called here

{ // local window access
  std::span<int> local_window = win.lock_local();
  local_window[0] = 42;
} // lock released here
```

## Breaking Orthogonality

```
std::vector<int> v = {...};

auto req = comm.isend(std::move(v),
           /*destination = */1,
           mpi::synchronous_mode);

v = req.wait();
```

- Modern languages may help to get rid of legacy design decisions
- Could be extended to a unified interface for persistent/non-blocking/blocking/streaming

## Other Languages

- Rust
- Julia
- (compiled Python)

# Open Question: How to get all this in the Standard?

- Full specification is too complex
- Implementation as “ground truth” only works for a normative interface on top of C
- We don’t want all languages in the standard

**Our Current Idea:** Language Guideline Side Document, “recipe” for implementers

# Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

PSAAP Funding in part is acknowledged from these NSF Grants OAC-2514054, CNS-2450093, CCF-2405142, and CCF-2412182 and the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966.

AAC work performed under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344, and supported by the Center for Non-Perturbative Studies of Functional Materials Under Non-Equilibrium Conditions (NPNEQ) funded by the Computational Materials Sciences Program of the US Department of Energy, Office of Science, Basic Energy Sciences, Materials Sciences and Engineering Division. This work was also performed under the auspices of the US Department of Energy's Pacific Northwest National Laboratory, operated by Battelle Memorial Institute under contract DE-AC05-76RL01830.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, or the U.S. Department of Energy's National Nuclear Security Administration.