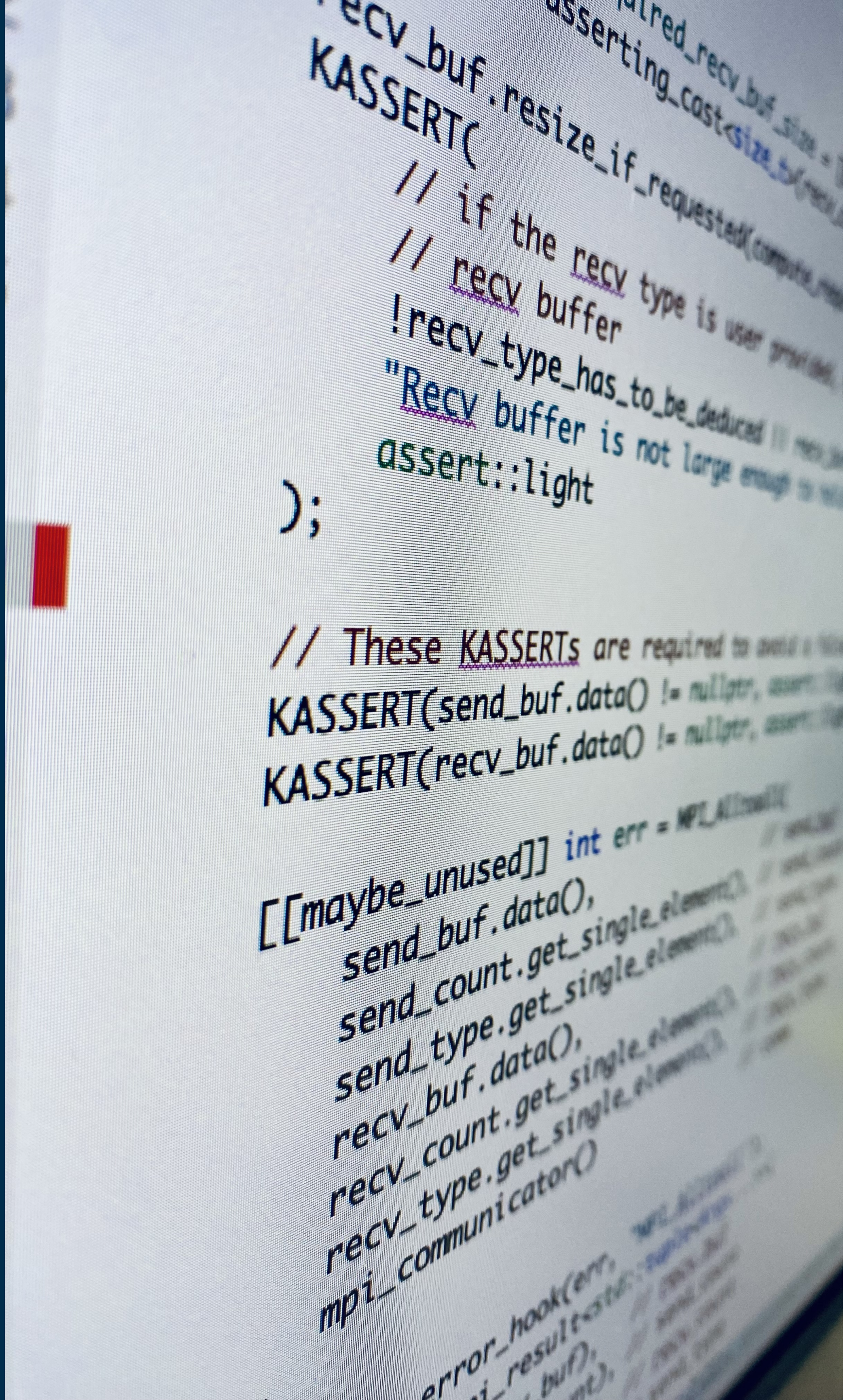
Concepts for designing modern C++ interfaces for MPI

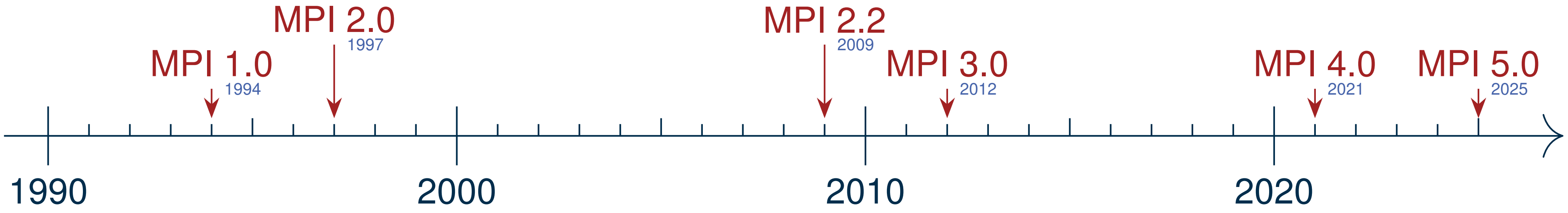
C. Nicole Avans[1]
Alfredo A. Correa[2]
Sayan Ghosh[3]
Matthias Schimek[4]
Joseph Schuchart[5]
Anthony Skjellum[1]
Evan D. Suggs[1]
Tim Niklas Uhl[4]

[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

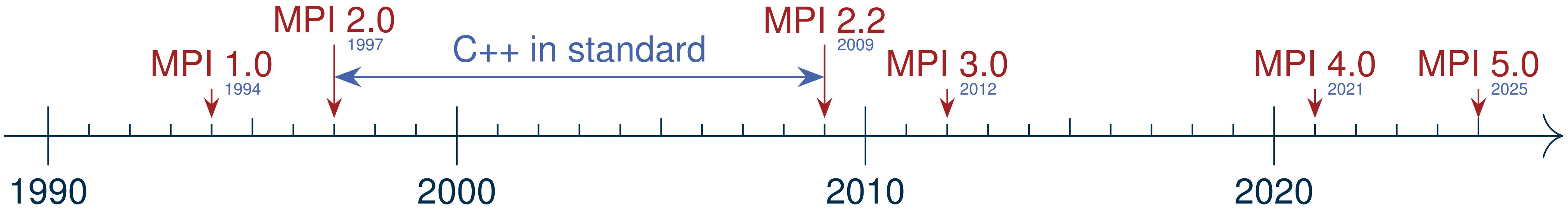
EuroMPI/USA'25, Charlotte, NC | October 2, 2025



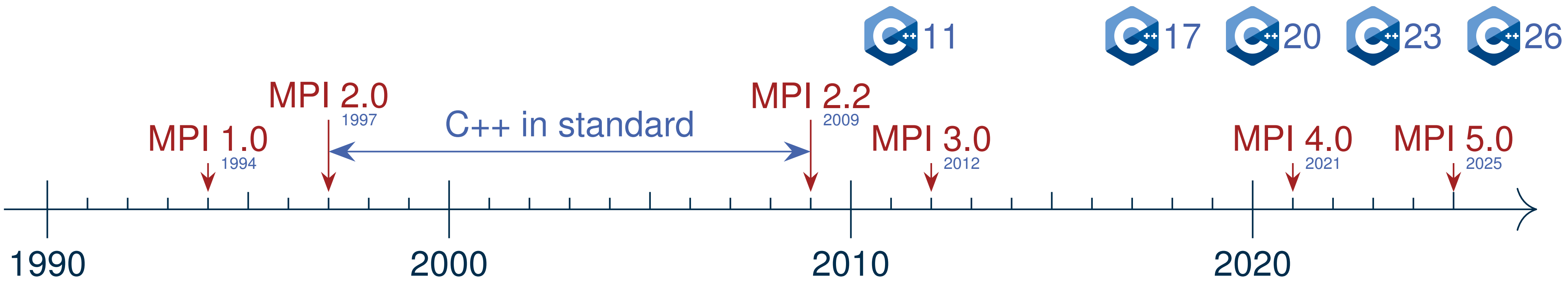
A Brief History of MPI and C++



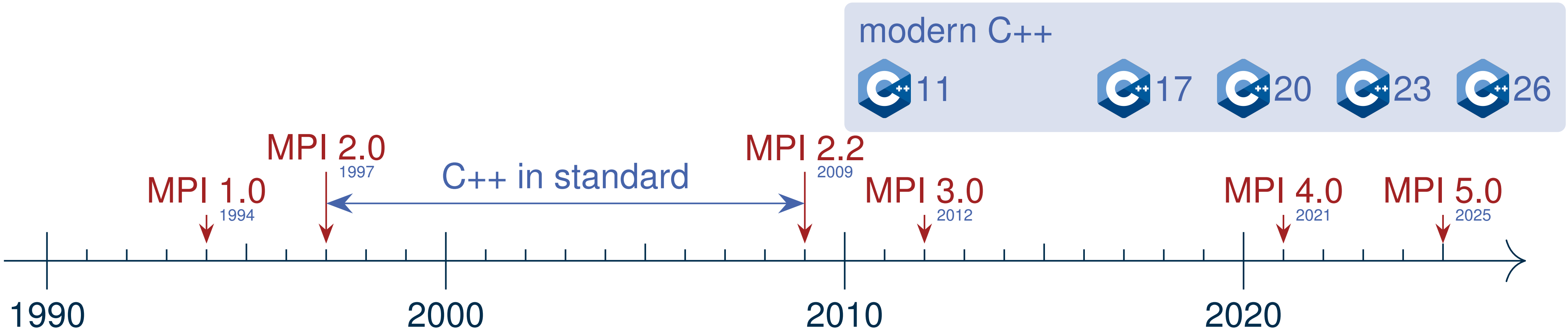
A Brief History of MPI and C++



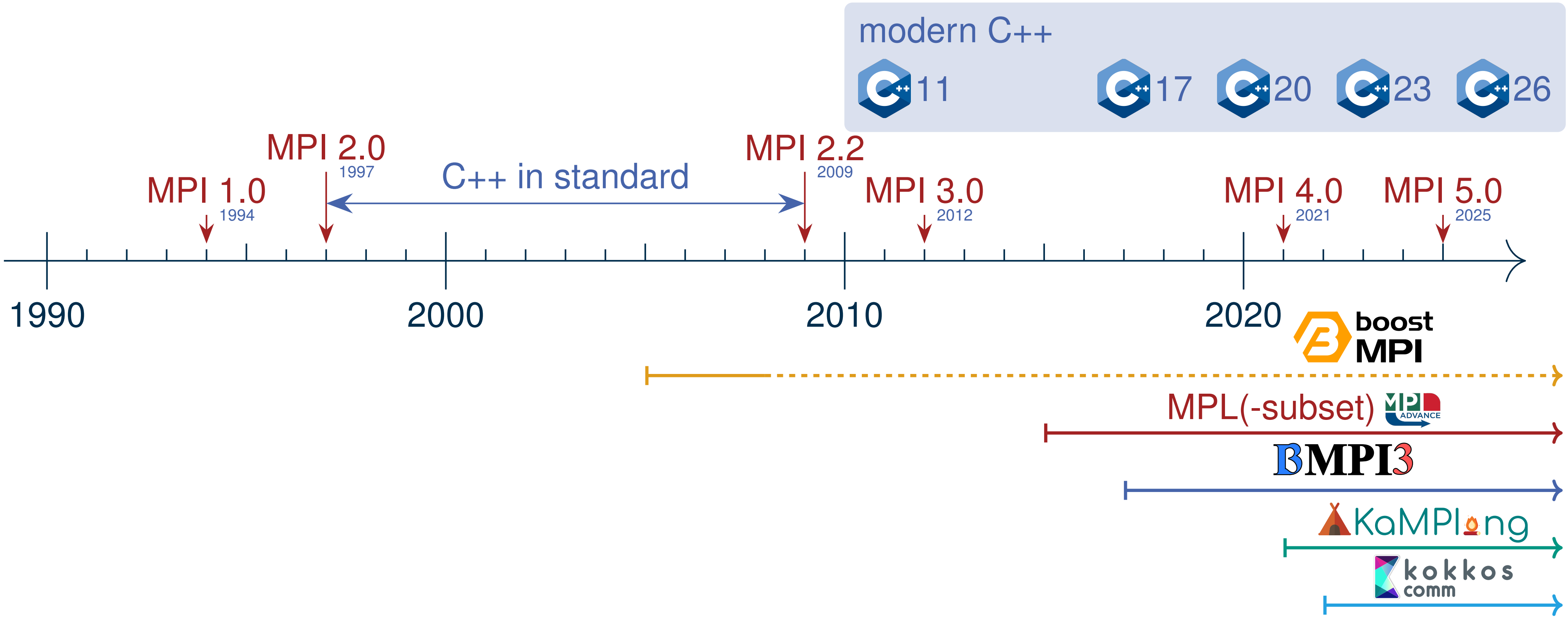
A Brief History of MPI and C++



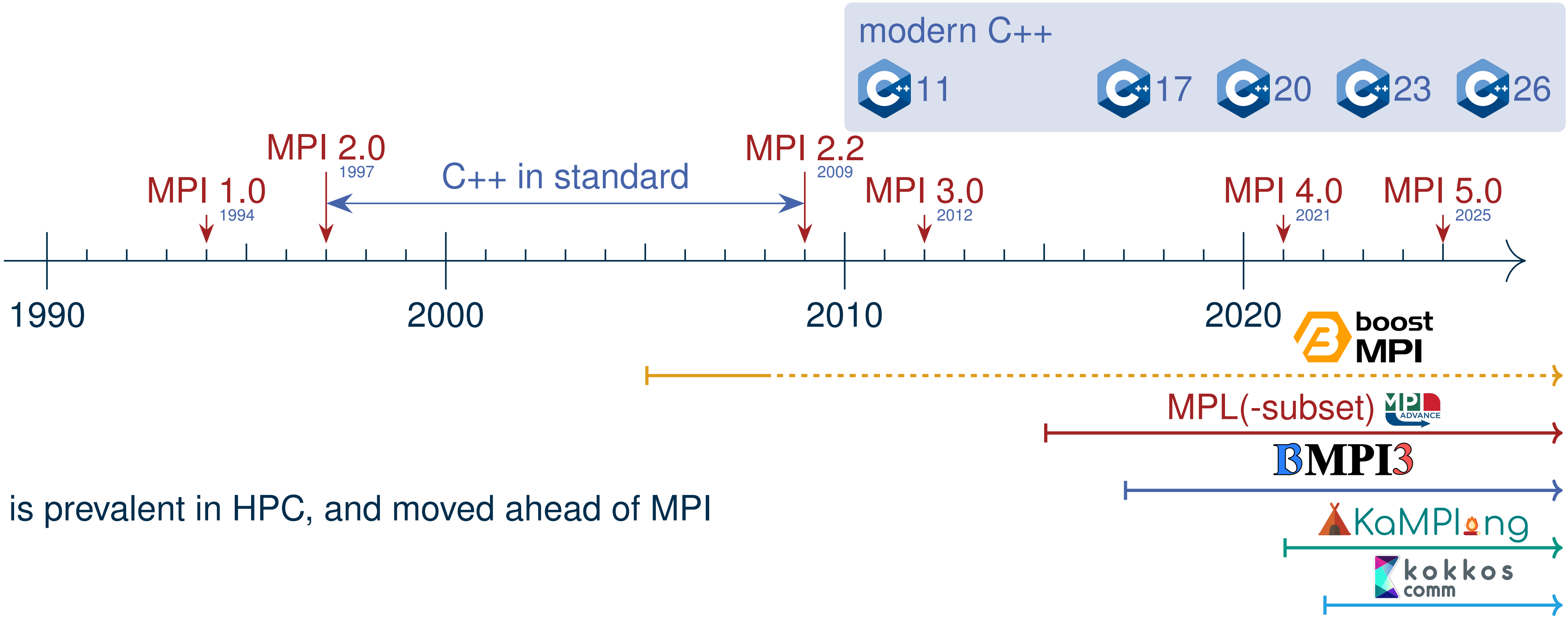
A Brief History of MPI and C++



A Brief History of MPI and C++

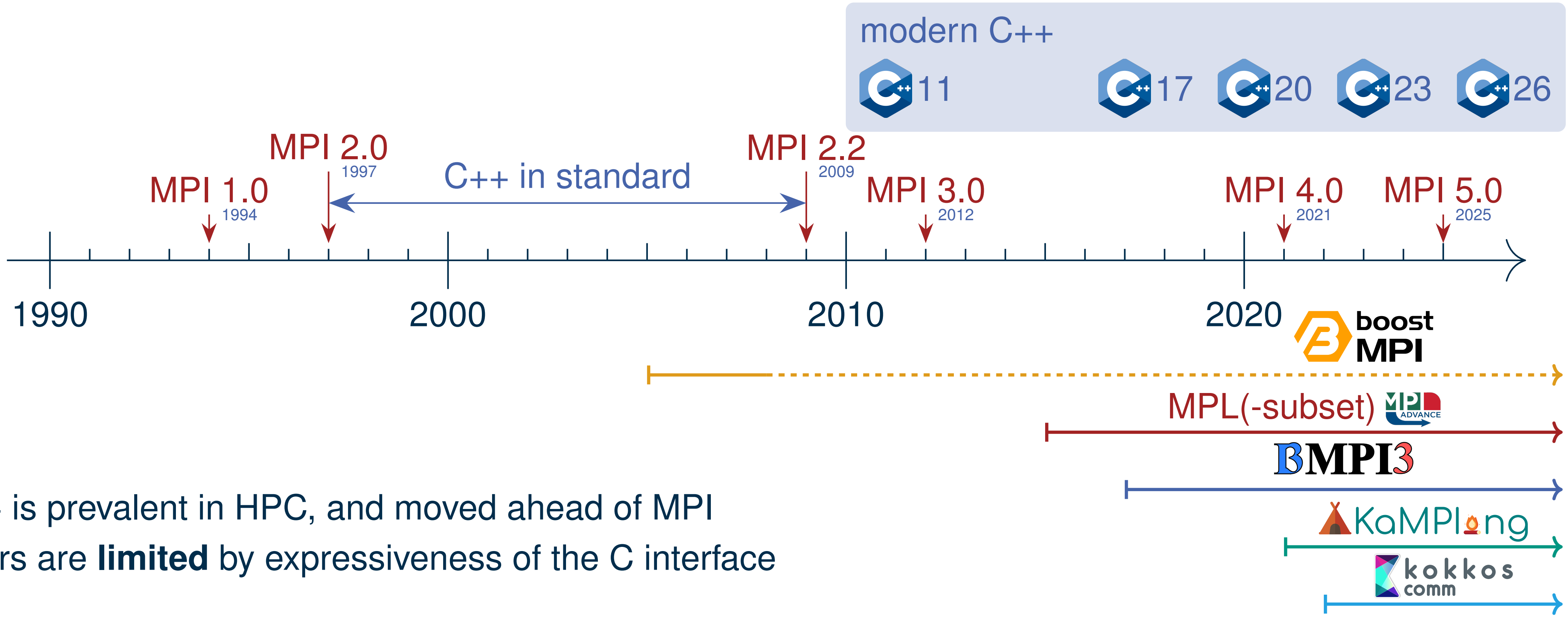


A Brief History of MPI and C++



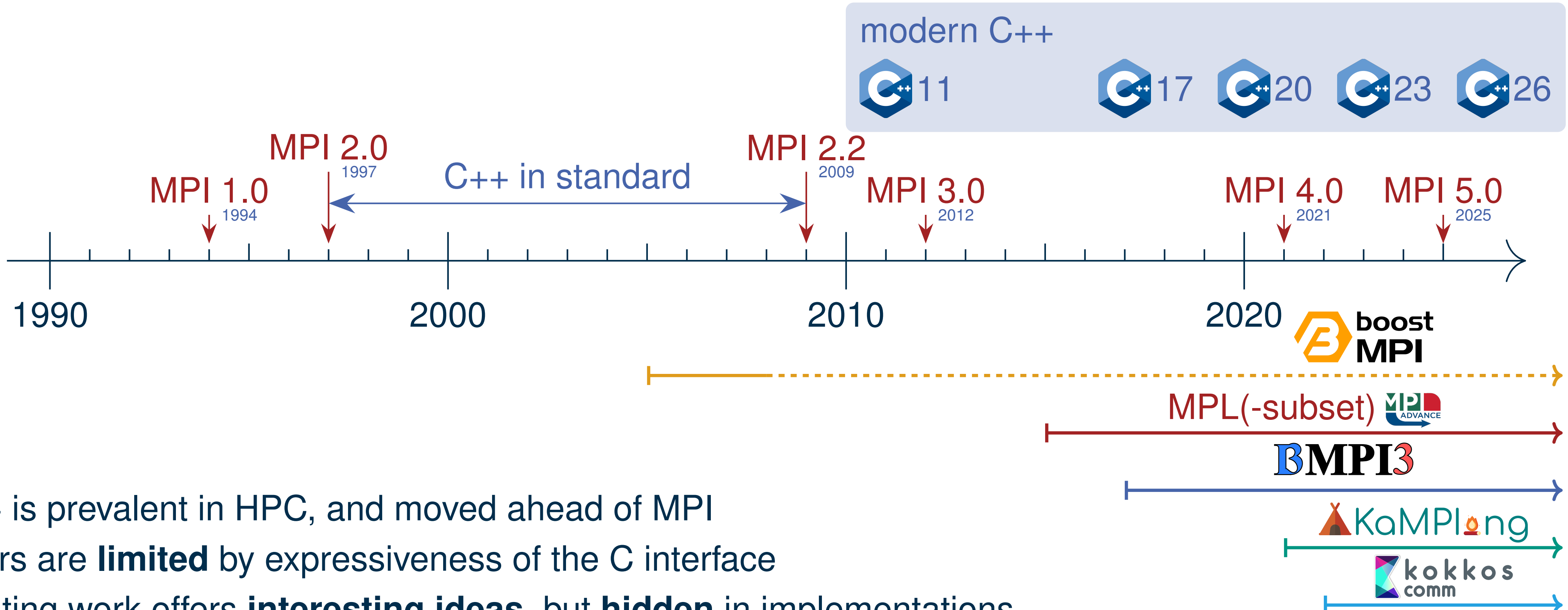
- C++ is prevalent in HPC, and moved ahead of MPI

A Brief History of MPI and C++



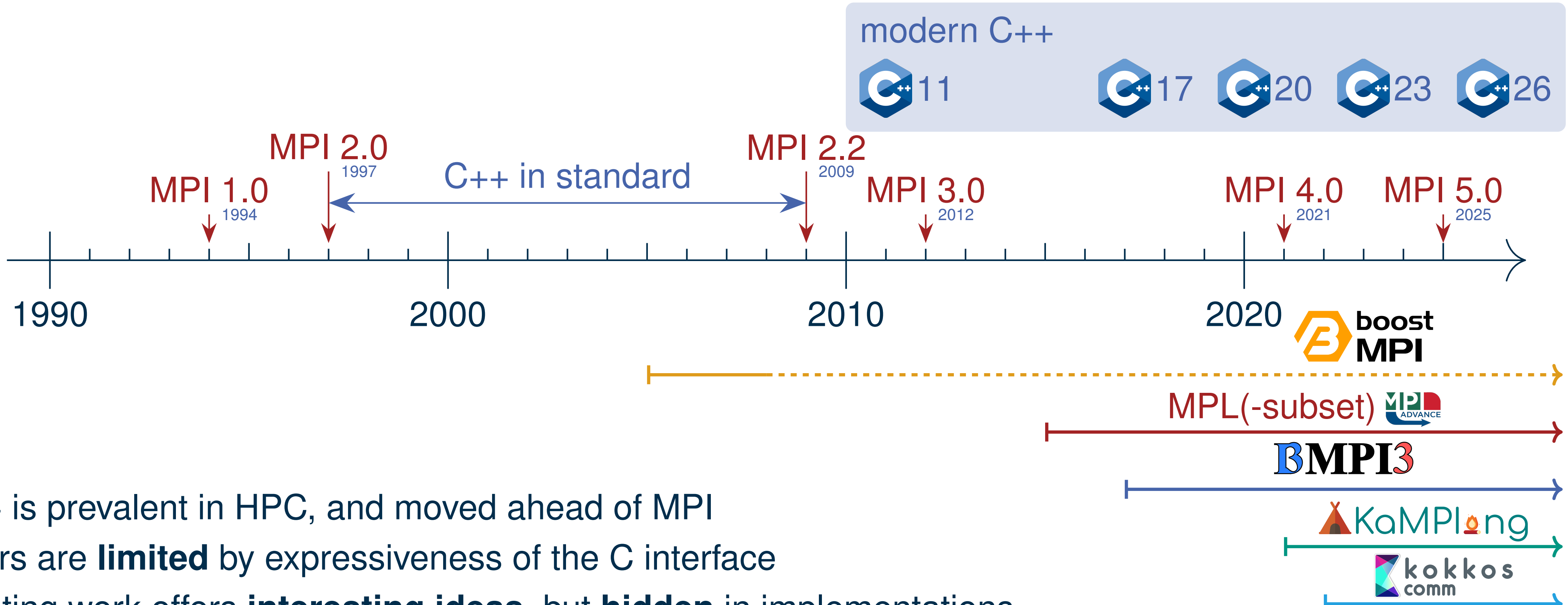
- C++ is prevalent in HPC, and moved ahead of MPI
- Users are **limited** by expressiveness of the C interface

A Brief History of MPI and C++



- C++ is prevalent in HPC, and moved ahead of MPI
- Users are **limited** by expressiveness of the C interface
- Existing work offers **interesting ideas**, but **hidden** in implementations

A Brief History of MPI and C++

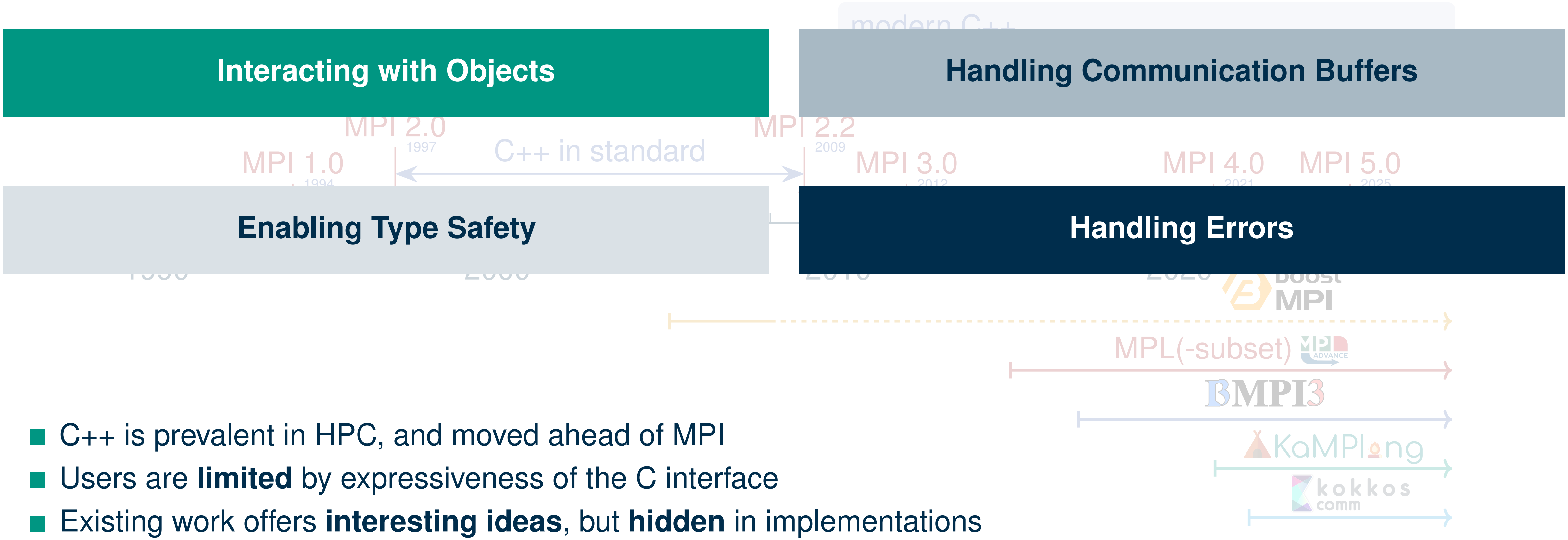


- C++ is prevalent in HPC, and moved ahead of MPI
- Users are **limited** by expressiveness of the C interface
- Existing work offers **interesting ideas**, but **hidden** in implementations
- **Our goal:** define **semantic concepts** to **bridge** MPI and idiomatic C++

A Brief History of MPI and C++

Interacting with Objects | **Handling Communication Buffers**

Enabling Type Safety | **Handling Errors**



- C++ is prevalent in HPC, and moved ahead of MPI
- Users are **limited** by expressiveness of the C interface
- Existing work offers **interesting ideas**, but **hidden** in implementations
- **Our goal:** define **semantic concepts** to **bridge** MPI and idiomatic C++

Modeling Objects

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, . . .
  - via **opaque handles** in C

Modeling Objects

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, ...
  - via **opaque handles** in C
- C++ idioms enable **value and ownership semantics**
  - via RAII (Resource Acquisition Is Initialization) ...
  - ... and move semantics

Modeling Objects

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, ...
  - via **opaque handles** in C
- C++ idioms enable **value and ownership semantics**
  - via RAII (Resource Acquisition Is Initialization) ...
  - ... and move semantics

```
class Comm {
public:
  Comm(Comm const &other, Group const &group) {
    MPI_Comm_create(other.handle(), group.handle(), &comm_);
  }
  Comm(Comm &&other) { ... }
  operator=(Comm && other) { ... }
  ~Comm() { MPI_Comm_free(&comm_); }
private:
  MPI_Comm comm_;
};
```

Modeling Objects

```cpp
class Comm {...};
```

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, ...
  - via **opaque handles** in C
- C++ idioms enable **value and ownership semantics**
  - via RAII (Resource Acquisition Is Initialization) ...
  - ... and move semantics

Modeling Objects

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, ...
  - via **opaque handles** in C
- C++ idioms enable **value and ownership semantics**
  - via RAII (Resource Acquisition Is Initialization) ...
  - ... and move semantics

```cpp
class Comm {...};

auto construct() {
  mpi::Comm comm{parent_comm, group};
  return comm;
}

void foobar(mpi::Comm comm) {
  // ...
  // comm gets freed at
  // end of scope
}

auto comm = construct();
foobar(std::move(comm));
```

Modeling Objects

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, . . .
  - via **opaque handles** in C
- C++ idioms enable **value and ownership semantics**
  - via RAII (Resource Acquisition Is Initialization) . . .
  - . . . and move semantics
  - but that sometimes conflicts with MPI's implicit global state

```
class Comm {...};

auto construct() {
  mpi::Comm comm{parent_comm, group};
  return comm;
}

void foobar(mpi::Comm comm) {
  // ...
  // comm gets freed at
  // end of scope
}

auto comm = construct();
foobar(std::move(comm));
```

Modeling Objects

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, ...
  - via **opaque handles** in C
- C++ idioms enable **value and ownership semantics**
  - via RAII (Resource Acquisition Is Initialization) ...
  - ... and move semantics
  - but that sometimes conflicts with MPI's implicit global state
- **session model** aligns well with object oriented design

```
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);
```

Modeling Objects

- MPI users interact with **objects**
  - communicators, data types, requests, groups, info objects, ...
  - via **opaque handles** in C
- C++ idioms enable **value and ownership semantics**
  - via RAII (Resource Acquisition Is Initialization) ...
  - ... and move semantics
  - but that sometimes conflicts with MPI's implicit global state
- **session model** aligns well with object oriented design

```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");
```

Modeling Data

Safe Type Mapping

`void`
`std::nullptr_t`

`float`
`double`
`long double`

`char`

`signed char`
`unsigned char`
`short int`
`unsigned short int`
`int`
`unsigned int`
`long int`
`unsigned long int`
`long long int`
`unsigned long long int`
`bool`
`char`

Modeling Data

Safe Type Mapping

fundamental types

`void`
`std::nullptr_t`

`float`
`double`
`long double`

`char`

`signed char`
`unsigned char`
`short int`
`unsigned short int`
`int`
`unsigned int`
`long int`
`unsigned long int`
`long long int`
`unsigned long long int`
`bool`
`char`

Modeling Data

Safe Type Mapping

MPI data types

fundamental types

`void`
`std::nullptr_t`

`float`
`double`
`long double`

`char`

`signed char`
`unsigned char`
`short int`
`unsigned short int`
`int`
`unsigned int`
`long int`
`unsigned long int`
`long long int`
`unsigned long long int`
`bool`
`char`

Modeling Data

Safe Type Mapping

Modeling Data

Safe Type Mapping

MPI data types

fundamental types

float
double
long double

char

signed char
unsigned char
short int
unsigned short int
int
unsigned int
long int
unsigned long int
long long int
unsigned long long int
bool
char

MPI_Type_contiguous
MPI_Type_vector
MPI_Type_create_hvector
MPI_Type_indexed
MPI_Type_create_hindexed
MPI_Type_create_struct
MPI_Type_create_subarray

Modeling Data

Safe Type Mapping

MPI data types

fundamental types

```
float
double
long double
```

```
char
```

```
signed char
unsigned char
short int
unsigned short int
int
unsigned int
long int
unsigned long int
long long int
unsigned long long int
bool
char
```

```
std::array<int, 3>
```

```
double[5]
```

```
enum class MyEnum {...};
```

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

```
MPI_Type_contiguous
MPI_Type_vector
MPI_Type_create_hvector
MPI_Type_indexed
MPI_Type_create_hindexed
MPI_Type_create_struct
MPI_Type_create_subarray
```

Modeling Data

Safe Type Mapping

MPI data types

fundamental types

```
template<typename T, size_t N>
struct type_traits<std::array<T, N>> {
  static MPI_Datatype type() {
    MPI_Datatype type;
    MPI_Type_contiguous(N,
      type_traits<T>::type(),
      &type);
    return type;
  }
};
```

```
float
double
long double
```

```
char
```

```
int
unsigned int
long int
unsigned long int
long long int
unsigned long long int
bool
char
```

```
std::array<int, 3>
```

```
double[5]
```

```
enum class MyEnum {...};
```

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

```
MPI_Type_contiguous
MPI_Type_vector
MPI_Type_create_hvector
MPI_Type_indexed
MPI_Type_create_hindexed
MPI_Type_create_struct
MPI_Type_create_subarray
```

Modeling Data

Safe Type Mapping

MPI data types

fundamental types

```
float
double
long double
```

```
char
```

```
int
unsigned int
long int
unsigned long int
long long int
unsigned long long int
bool
char
```

```
template<typename T, size_t N>
struct type_traits<std::array<T, N>> {
  static MPI_Datatype type() {
    MPI_Datatype type;
    MPI_Type_contiguous(N,
      type_traits<T>::type(),
      &type);
    return type;
  }
};
```

```
std::array<int, 3>
```

```
double[5]
```

```
enum class MyEnum {...};
```

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

```
MPI_Type_contiguous
MPI_Type_vector
MPI_Type_create_hvector
MPI_Type_indexed
```

```
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;
  }
}
```

Modeling Data

Safe Type Mapping

fundamental types

float
double
long double

char

int
unsigned int
long int
unsigned long int
long long int
unsigned long long int
bool
char

```
template<typename T, size_t N>
struct type_traits<std::array<T, N>> {
  static MPI_Datatype type() {
    MPI_Datatype type;
    MPI_Type_contiguous(N,
      type_traits<T>::type(),
      &type);
    return type;
  }
};
```

trivially copyable types

`std::array<int, 3>`

`double[5]`

`enum class MyEnum {...};`

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

MPI data types

MPI_Type_contiguous
MPI_Type_vector
MPI_Type_create_hvector
MPI_Type_indexed

```
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;
  }
}
```

Modeling Data

Safe Type Mapping

MPI data types

trivially copyable types

fundamental types

- `float`
- `double`
- `long double`

- `char`

- `signed char`
- `unsigned char`
- `short int`
- `unsigned short int`
- `int`
- `unsigned int`
- `long int`
- `unsigned long int`
- `long long int`
- `unsigned long long int`
- `bool`
- `char`

```
std::array<int, 3>
```

```
double[5]
```

```
enum class MyEnum {...};
```

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

- `MPI_Type_contiguous`
- `MPI_Type_vector`
- `MPI_Type_create_hvector`
- `MPI_Type_indexed`
- `MPI_Type_create_hindexed`
- `MPI_Type_create_struct`
- `MPI_Type_create_subarray`

Modeling Data

Handling Memory Buffers

```
std::vector<int> v = {...};
MPI_Sendrecv(v.data(), v.size(), MPI_INT, dest, ...);
```

Modeling Data

Handling Memory Buffers

```
template<typename T>
int sendrecv(std::vector<T> const& s, int dest, ...) {
  return MPI_Sendrecv(s.data(), s.size(), MPI_INT, dest, ...);
}
```

Modeling Data

Handling Memory Buffers

```
template<typename T>
int sendrecv(std::vector<T> const& s, int dest, ...) {
  return MPI_Sendrecv(s.data(), s.size(), MPI_INT, dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

Modeling Data

Handling Memory Buffers

```
template<typename T>
int sendrecv(std::vector<T> const& s, int dest, ...) {
  return MPI_Sendrecv(s.data(), s.size(), MPI_INT, dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

Modeling Data

Handling Memory Buffers

```
template<typename T>
int sendrecv(std::vector<T> const& s, int dest, ...) {
  return MPI_Sendrecv(s.data(), s.size(), MPI_INT, dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

send buffer

recv buffer

Modeling Data

Handling Memory Buffers

```
template<DataBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(s.data(), s.size(), MPI_INT, dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

send buffer

recv buffer

```
template <typename T>
concept DataBuffer =
```

Modeling Data

Handling Memory Buffers

```
template<DataBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), s.size(), MPI_INT, dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

send buffer

recv buffer

```
template <typename T>
concept DataBuffer =
  std::ranges::contiguous_range<T>;
```

Modeling Data

Handling Memory Buffers

```
template<DataBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::ranges::size(s), MPI_INT, dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

send buffer

recv buffer

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

Modeling Data

Handling Memory Buffers

```
template<DataBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::ranges::size(s), mpi::type(s), dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

send buffer

recv buffer

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

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::ranges::size(s), mpi::type(s), dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

send buffer

recv buffer

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

template <typename T>
concept SendBuffer =
  DataBuffer<T> &&
  std::ranges::input_range<T>;
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::ranges::size(s), mpi::type(s), dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

send buffer

recv buffer

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

template <typename T>
concept SendBuffer =
  DataBuffer<T> &&
  std::ranges::input_range<T>;

template <typename T>
concept RecvBuffer =
  DataBuffer<T> &&
  std::ranges::output_range<T,
       std::ranges::range_value_t<T>>;
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::ranges::size(s), mpi::type(s), dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype>;
};
```

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

template <typename T>
concept SendBuffer =
  DataBuffer<T> &&
  std::ranges::input_range<T>;

template <typename T>
concept RecvBuffer =
  DataBuffer<T> &&
  std::ranges::output_range<T,
       std::ranges::range_value_t<T>>;
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::ranges::size(s), mpi::type(s), dest, ...);
}
```

Problem: ad-hoc, limited to a fixed set of containers → **Solution:** define general concepts

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype>;
};

template <typename Buf>
  requires mpi::has_static_type<
             std::ranges::range_value_t<Buf>> ||
           has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>();
  }
}
```

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

template <typename T>
concept SendBuffer =
  DataBuffer<T> &&
  std::ranges::input_range<T>;

template <typename T>
concept RecvBuffer =
  DataBuffer<T> &&
  std::ranges::output_range<T,
      std::ranges::range_value_t<T>>;
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::r
}
```

Problem: ad-hoc, limited to a fixed set of containers

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype
};

template <typename Buf>
  requires mpi::has_static_type<
              std::ranges::range_value_t<Buf>> |
           has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::r
}
```

Problem: ad-hoc, limited to a fixed set of containers

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype
};

template <typename Buf>
  requires mpi::has_static_type<
              std::ranges::range_value_t<Buf>> |
           has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::r
}
```

Problem: ad-hoc, limited to a fixed set of containers

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype>
};

template <typename Buf>
  requires mpi::has_static_type<
             std::ranges::range_value_t<Buf>> |
           has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::r
}
```

Problem: ad-hoc, limited to a fixed set of containers

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype>
};

template <typename Buf>
  requires mpi::has_static_type<
             std::ranges::range_value_t<Buf>> ||
           has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::r
}
```

Problem: ad-hoc, limited to a fixed set of containers

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype
};

template <typename Buf>
  requires mpi::has_static_type<
              std::ranges::range_value_t<Buf>> |
           has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::r
}
```

Problem: ad-hoc, limited to a fixed set of containers

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype
};

template <typename Buf>
  requires mpi::has_static_type<
             std::ranges::range_value_t<Buf>> |
           has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
  return MPI_Sendrecv(std::ranges::data(s), std::r
}
```

Problem: ad-hoc, limited to a fixed set of containers

```
template <typename Buf>
concept Typed = requires(Buf buf) {
  { mpi::type(buf) } -> std::same_as<MPI_Datatype
};

template <typename Buf>
  requires mpi::has_static_type<
                std::ranges::range_value_t<Buf>> |
            has_type_member<Buf>
MPI_Datatype type(Buf &buf) {
  if constexpr (has_type_member<Buf>) {
    return buf.type();
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
```

Collectives with varying counts

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

// does not compile, send_buf
// needs members .sizev() and .displs()
comm.scatterv(send_buf, recv_buf, root);

// adapt data buffer with count information
comm.scatterv(
  mpi::vbuf {send_buf, scounts, sdisp},
  recv_buf,
  root
);
```

```
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Handling Memory Buffers

```
template<SendBuffer SendBuf>
int sendrecv(SendBuf&& s, int dest, ...) {
```

Collectives with varying counts

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

// does not compile, send_buf
// needs members .sizev() and .displs()
comm.scatterv(send_buf, recv_buf, root);

// adapt data buffer with count information
comm.scatterv(
  mpi::vbuf {send_buf, scounts, sdisp},
  recv_buf,
  root
);
```

```
  } else {
    type_traits<std::ranges::range_value_t<Buf>>(
  }
}
```

Examples

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

// send a vector directly
comm.send(v, dest);

// send something from a raw pointer
comm.send(std::span(buf, BUFSIZE), dest);

// send a single element
comm.send(std::views::single {42}, dest);

// send only the first 4 elements
comm.send(v | std::views::take(4), dest);

// send with custom type
comm.send(v | with_type(MY_MPI_TYPE), dest);

// send a kokkos view, internally repacking data
// or constructing an appropriate type
Kokkos::View<int*> kv (...);
comm.send(kokkos_adapter {kv}, dest);

// accelerator support, adapting .data()
thrust::device_vector<int> dv;
comm.send(thrust_adapter {kv}, dest);
```

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory

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

recv_buf = comm.recv(std::move(recv_buf), ...);
```

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory

```
std::vector<int> recv_buf = {...};
                   transfer ownership to library

recv_buf = comm.recv(std::move(recv_buf), ...);
```

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory

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

transfer ownership to library

```
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory

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

transfer ownership to library

```
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```
comm.recv(recv_buf, ...);
```

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory

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

transfer ownership to library

```
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```
comm.recv(recv_buf, ...);
```

captured by reference
$\rightarrow$ value not returned

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory
- **Memory safety** guarantees for non-blocking communication **for free**

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

transfer ownership to library

```
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```
comm.recv(recv_buf, ...);
```

captured by reference
→ value not returned

```
auto req = comm.irecv(          recv_buf , ...);

// ... other computations

                  req.wait(...);
```

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory
- **Memory safety** guarantees for non-blocking communication **for free**

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

transfer ownership to library

```
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```
comm.recv(recv_buf, ...);
```

captured by reference
→ value not returned

```
auto req = comm.irecv(          recv_buf , ...);

// ... other computations

                req.wait(...);
```

access to `recv_buf` prohibited

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory
- **Memory safety** guarantees for non-blocking communication **for free**

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

transfer ownership to library

```
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```
comm.recv(recv_buf, ...);
```

captured by reference
→ value not returned

```
auto req = comm.irecv(std::move(recv_buf), ...);

// ... other computations

                    req.wait(...);
```

access to
recv_buf
prohibited

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory
- **Memory safety** guarantees for non-blocking communication **for free**

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

transfer ownership to library

```
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```
comm.recv(recv_buf, ...);
```

captured by reference
→ value not returned

mpi::request<std::vector<int>>

```
auto req = comm.irecv(std::move(recv_buf), ...);

// ... other computations

               req.wait(...);
```

access to recv_buf prohibited

EUROMPI 25

Modeling Data

Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory
- **Memory safety** guarantees for non-blocking communication **for free**

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

transfer ownership to library

```cpp
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```cpp
comm.recv(recv_buf, ...);
```

captured by reference
→ value not returned

```cpp
mpi::request<std::vector<int>>
```

```cpp
auto req = comm.irecv(std::move(recv_buf), ...);

// ... other computations

recv_buf = req.wait(...);
```

access to recv_buf prohibited

return ownership

Modeling Data
Ownership

- Modeling data as **data buffers** allows for a better **ownership** model
- Clear **responsibility** for freeing memory
- **Memory safety** guarantees for non-blocking communication **for free**

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

transfer ownership to library

```cpp
recv_buf = comm.recv(std::move(recv_buf), ...);
```

return ownership to caller

```cpp
comm.recv(recv_buf, ...);
```

captured by reference
→ value not returned

```cpp
mpi::request<std::vector<int>>
```

```cpp
auto req = comm.irecv(std::move(recv_buf), ...);

// ... other computations

recv_buf = req.wait(...);
```

access to `recv_buf` prohibited

return ownership

```cpp
std::optional<std::vector<int>> result = req.test(...);
```

Handling Errors

- Meta-programming allows catching many (usage) errors at **compile-time**

Handling Errors

- Meta-programming allows catching many (usage) errors at **compile-time**
  - using `static_assert`
  - even more flexible with C++26

Handling Errors

- Meta-programming allows catching many (usage) errors at **compile-time**
  - using `static_assert`
  - even more flexible with C++26
- **but** no clear consensus on error handling – in C++ **and** MPI

Handling Errors

- Meta-programming allows catching many (usage) errors at **compile-time**
  - using `static_assert`
  - even more flexible with C++26
- **but** no clear consensus on error handling – in C++ **and** MPI

Errors via exceptions

```
try {
  std::vector<int> buf = comm.recv(std::vector<int>(recv_size), src, tag);
  // [...] computation
} catch (const mpi::exception& e) {
  std::cerr << "Recv failed: " << e.what() << "\n";
}
```

Handling Errors

- Meta-programming allows catching many (usage) errors at **compile-time**
  - using `static_assert`
  - even more flexible with C++26
- **but** no clear consensus on error handling – in C++ **and** MPI

Errors via exceptions

```
try {
  std::vector<int> buf = comm.recv(std::vector<int>(recv_size), src, tag);
  // [...] computation
} catch (const mpi::exception& e) {
  std::cerr << "Recv␣failed:␣" << e.what() << "\n";
}
```

Errors via `std::expected`

```
std::expected result = comm.recv(std::vector<int>(recv_size), src, tag);
if (result) {
  std::vector buf = std::move(*result);
  // [...] computation
} else {
  std::cerr << "Recv␣failed:␣"
            << result.error().message() << "\n";
  auto recovered_buffer = std::move(result).error().recv_buf();
}
```

Conclusion

We started deriving **language support guidelines** for C++

Core aspects

- Object model
- Data representation
- Ownership
- Modeling and mapping types
- Idiomatic error handling

Future work

- Build a reference implementation (+ extensions for accelerators, libraries, . . .)
- Concretize guidelines for non-blocking, one-sided, partitioned, . . . communication
- Improve MPI's language interoperability, without sacrificing a C++ performance path

Open Question: How to bring C++ back to the MPI standard?

Sincere thanks to everyone who contributed to the MPI Languages WG discussions.

Conclusion

We started deriving **language support guidelines** for C++

Core aspects

- Object model
- Data representation
- Ownership
- Modeling and mapping types
- Idiomatic error handling

Future work

- Build a reference implementation (+ extensions for accelerators, libraries, . . .)
- Concretize guidelines for non-blocking, one-sided, partitioned, . . . communication
- Improve MPI's language interoperability, without sacrificing a C++ performance path

Open Question: How to bring C++ back to the MPI standard?

Sincere thanks to everyone who contributed to the MPI Languages WG discussions.

Conclusion

We started deriving **language support guidelines** for C++

Core aspects

- Object model
- Data representation
- Ownership
- Modeling and mapping types
- Idiomatic error handling

Future work

- Build a reference implementation (+ extensions for accelerators, libraries, . . .)
- Concretize guidelines for non-blocking, one-sided, partitioned, . . . communication
- Improve MPI's language interoperability, without sacrificing a C++ performance path

Open Question: How to bring C++ back to the MPI standard?

Sincere thanks to everyone who contributed to the MPI Languages WG discussions.

Acknowledgments

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.