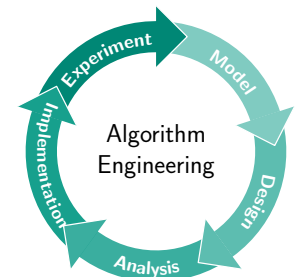


Concepts in Practice: C++ MPI Bindings for the HPC Ecosystem

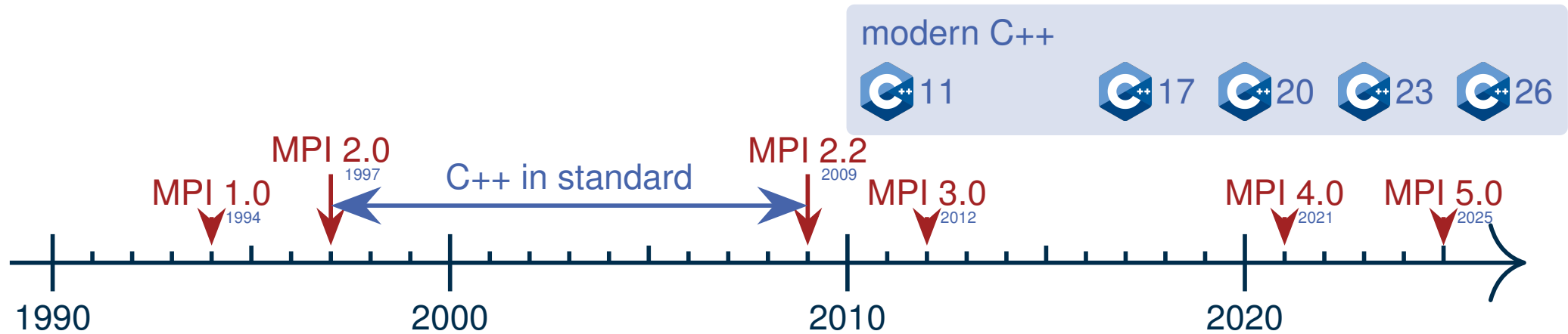
From a Standardizable Core to a Composable Interface

Tim Niklas Uhl uhl@kit.edu
Matthias Schimek
Daniel Brommer

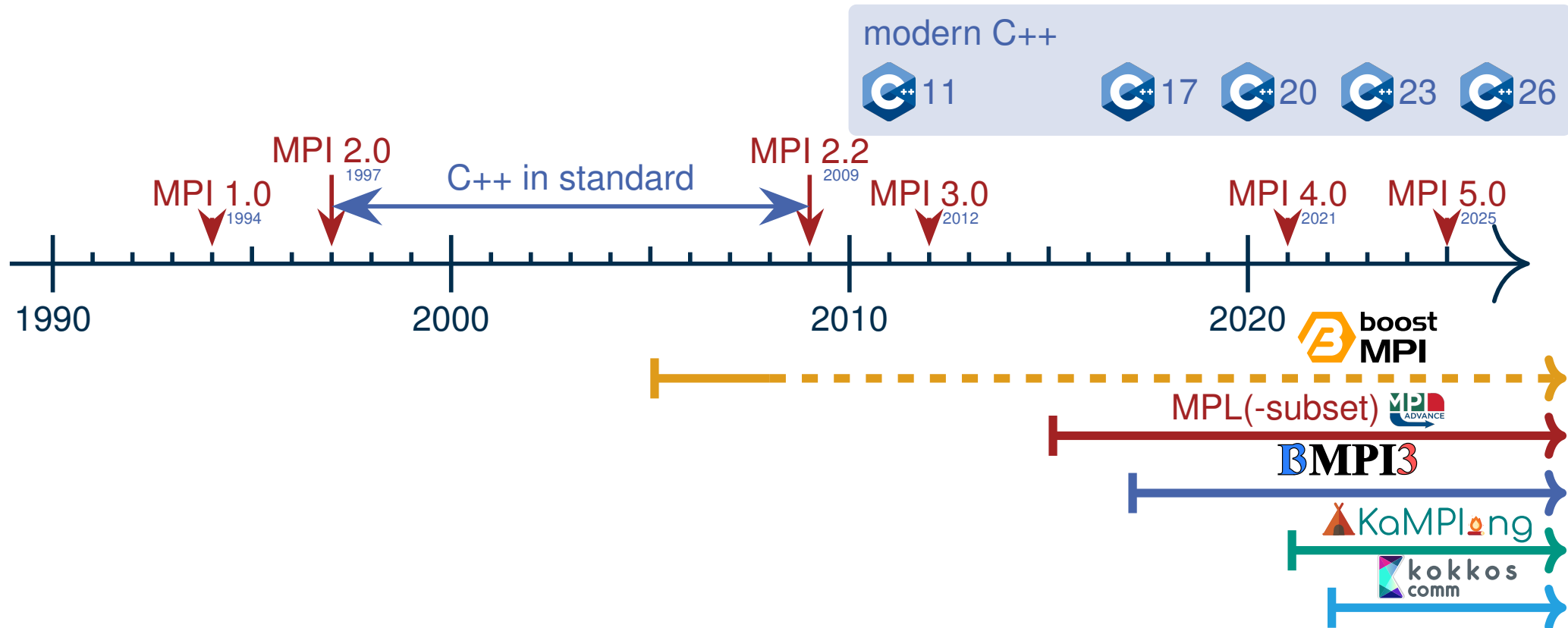
EuroMPI'26 | October 7, 2026



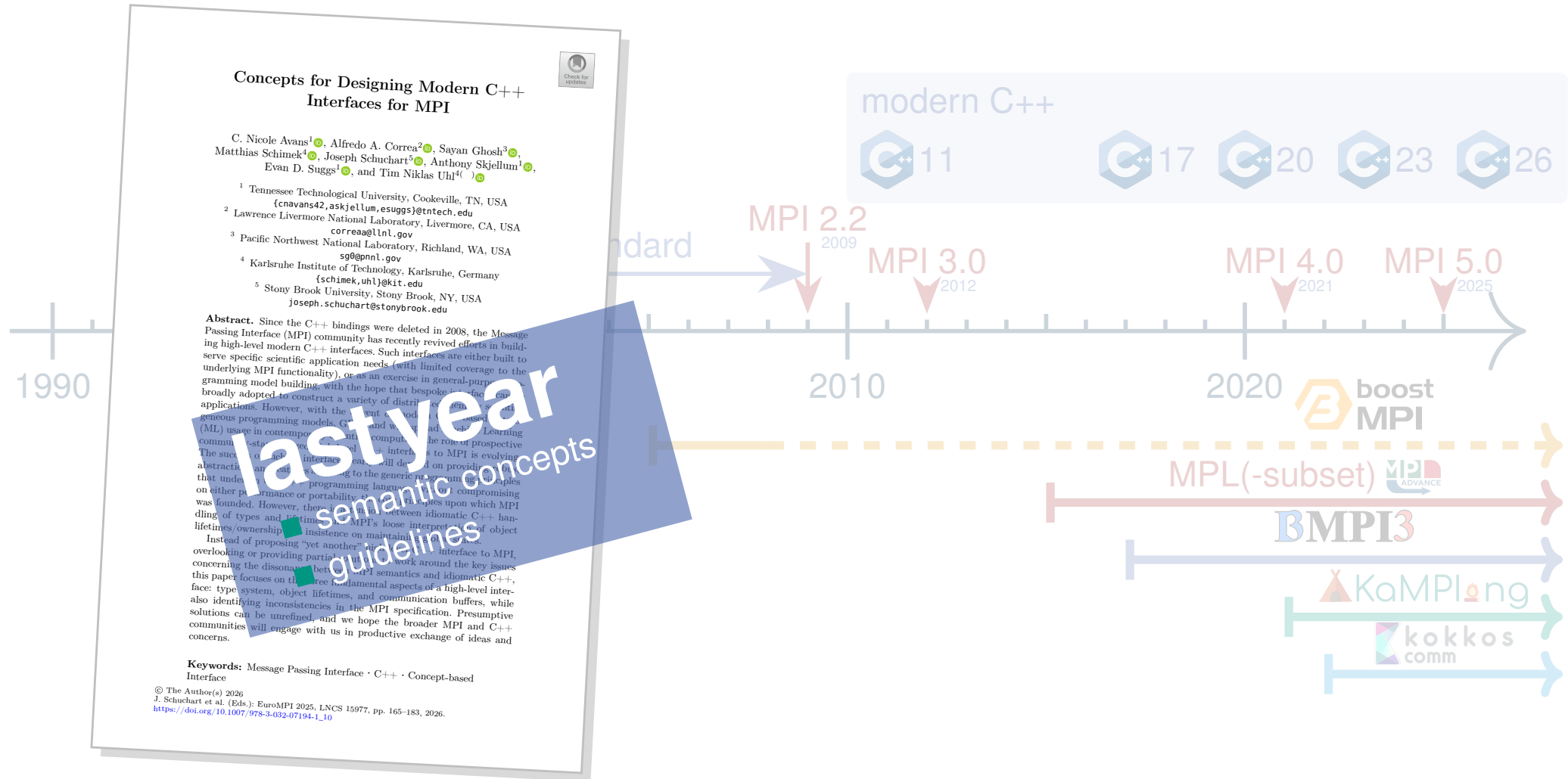
A Brief History of MPI and C++



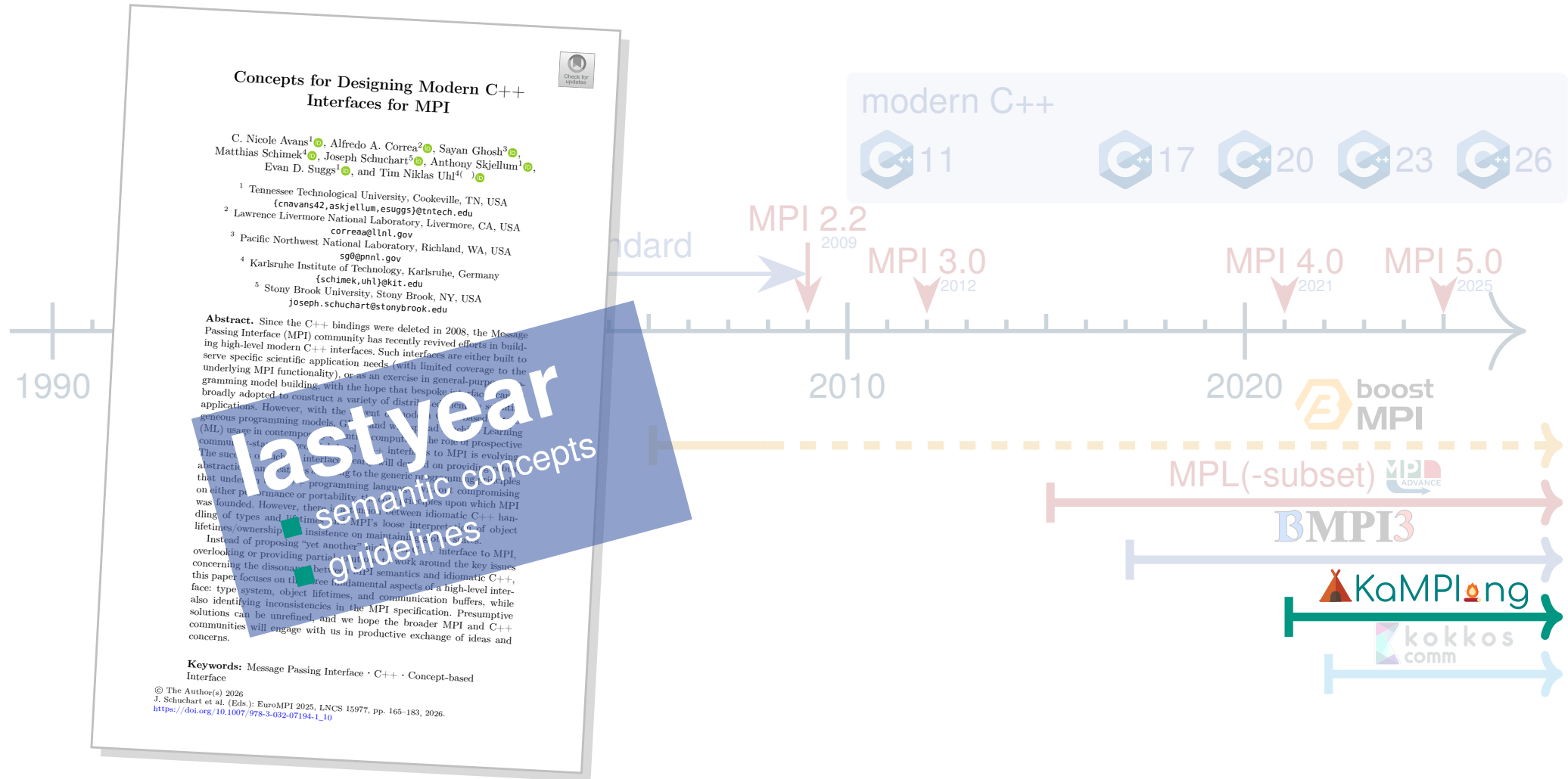
A Brief History of MPI and C++



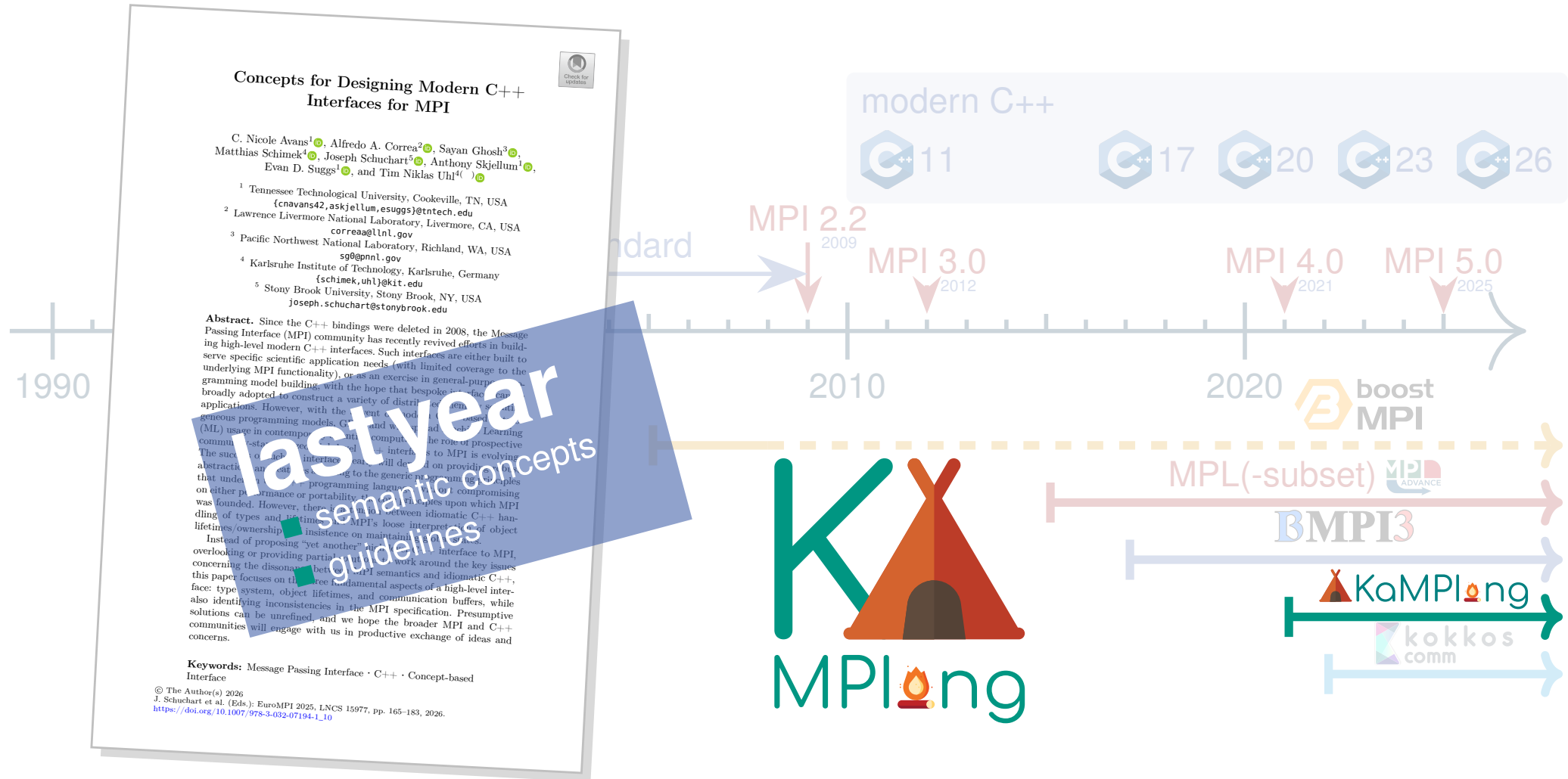
A Brief History of MPI and C++



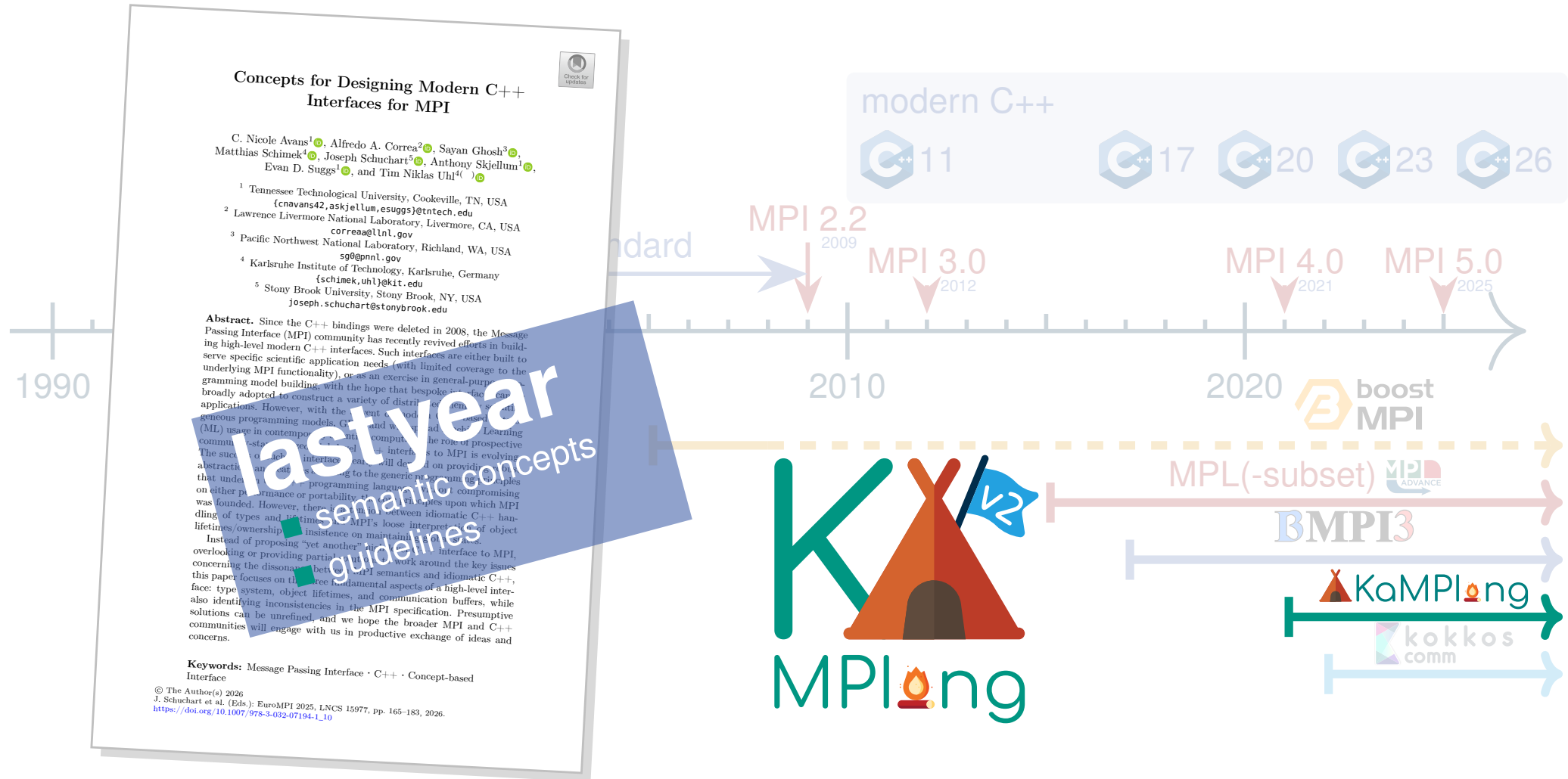
A Brief History of MPI and C++



A Brief History of MPI and C++



A Brief History of MPI and C++



Standardizable Core, Convenience on Top



C++ **interface** for all MPI procedures, built on **concepts** for buffers and objects — minimal, standardizable



language-independent semantics of **buffers** (address, count, datatype) and **objects** (communicators, requests, ...)

Standardizable Core, Convenience on Top



views

composable buffer adapters: types, counts, ...

usable directly
with `mpi::core`

interface

resizing, count deduction,
safe non-blocking

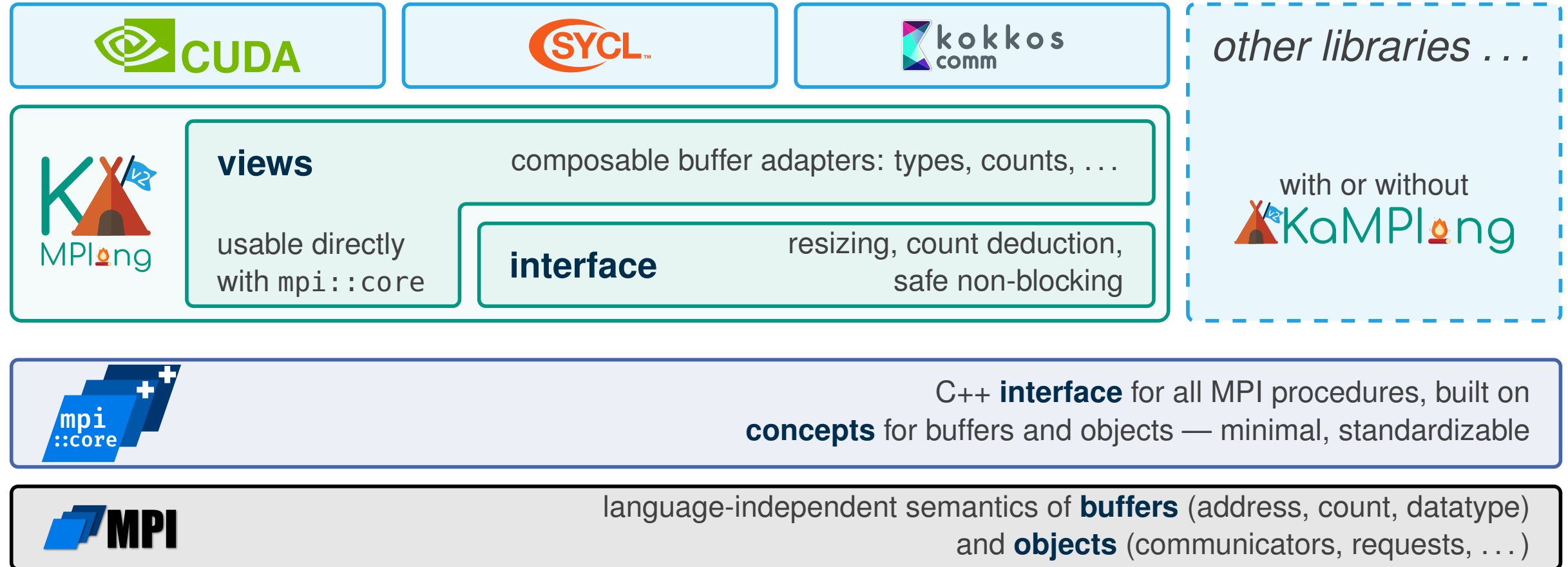


C++ **interface** for all MPI procedures, built on
concepts for buffers and objects — minimal, standardizable

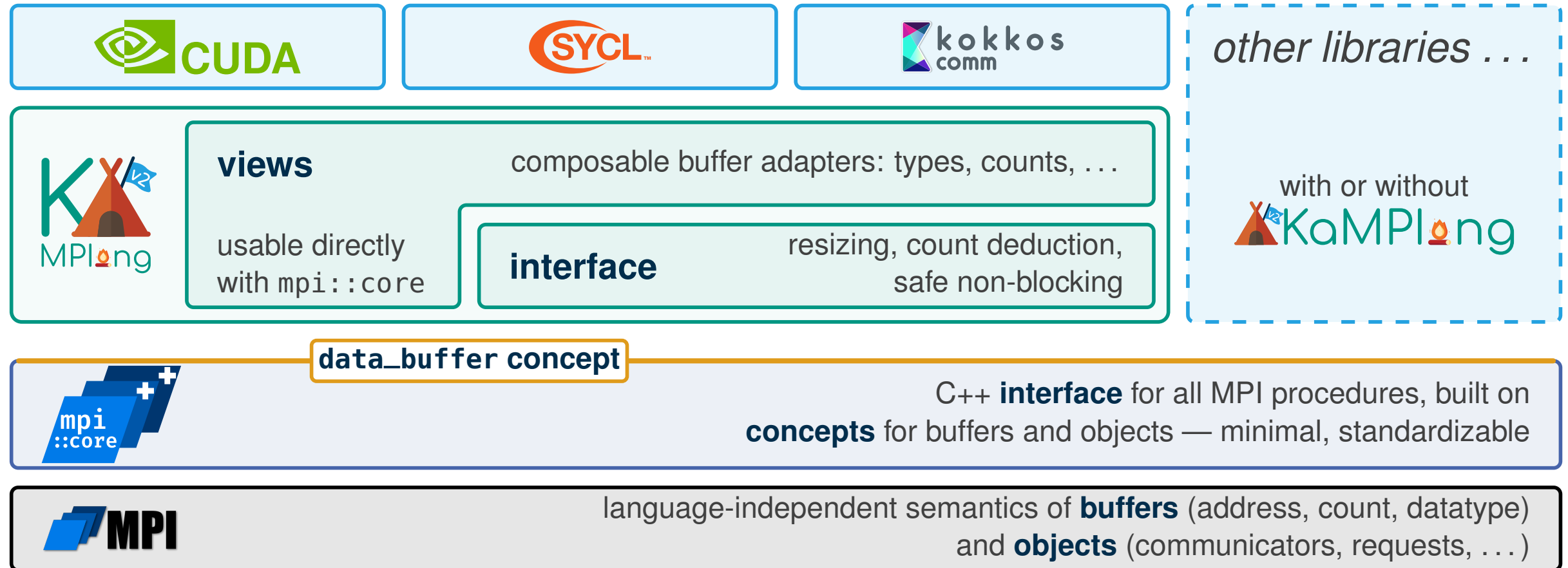


language-independent semantics of **buffers** (address, count, datatype)
and **objects** (communicators, requests, ...)

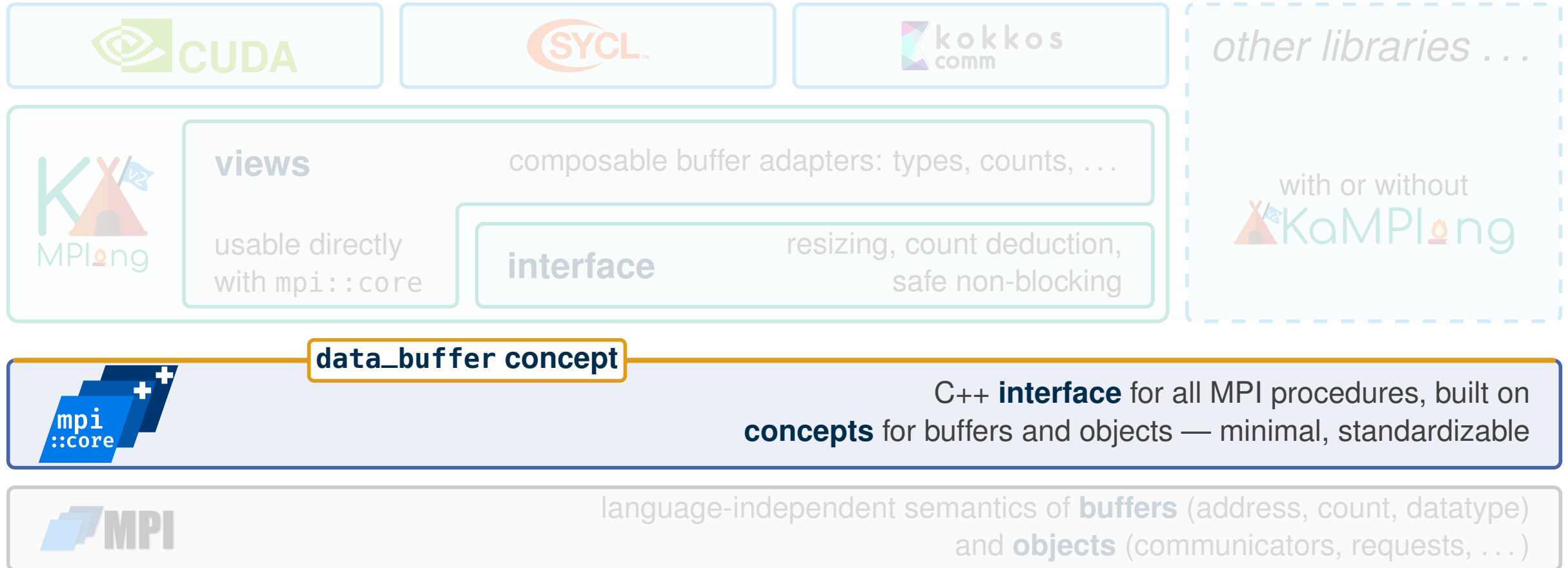
Standardizable Core, Convenience on Top



Standardizable Core, Convenience on Top



Standardizable Core, Convenience on Top



Data Buffers as a C++ Concept

3.2.2 Message Data

The `send buffer` specified by the `MPI_SEND` procedure consists of `count` successive entries of the type indicated by `datatype` starting with the entry at address `buf`. Note that we specify

Buffers in C

```
MPI_Send(  
    const void *buf, int count,  
    MPI_Datatype datatype, int dest, int tag,  
    MPI_Comm comm)
```

Data Buffers as a C++ Concept

3.2.2 Message Data

The `send buffer` specified by the `MPI_SEND` procedure consists of `count` successive entries of the type indicated by `datatype` starting with the entry at address `buf`. Note that we specify

Buffers in C

```
MPI_Send(  
    const void *buf, int count,  
    MPI_Datatype datatype, int dest, int tag,  
    MPI_Comm comm)
```

```
template <typename T>  
concept data_buffer = requires(T&& t, T const& ct) {  
    { mpi::ptr(t) } -> ptr_to_object;  
    { mpi::count(ct) } -> integer_like;  
    { mpi::type(ct) } -> std::convertible_to<MPI_Datatype>;  
};
```



Building the Interface

```
template <mpi::send_buffer SBuf,  
          mpi::convertible_to_mpi_handle<MPI_Comm> Comm>  
void send(SBuf&& sbuf, int dest, int tag, Comm const& comm) {  
    int err = MPI_Send(  
        mpi::ptr(sbuf), mpi::count(sbuf), mpi::type(sbuf),  
        dest, tag, mpi::handle(comm));  
    if (err != MPI_SUCCESS) throw mpi::mpi_error{err};  
}
```



Building the Interface

```
template <mpi::send_buffer SBuf,  
          mpi::convertible_to_mpi_handle<MPI_Comm> Comm>  
void send(SBuf&& sbuf, int dest, int tag, Comm const& comm) {  
    int err = MPI_Send(  
        mpi::ptr(sbuf), mpi::count(sbuf), mpi::type(sbuf),  
        dest, tag, mpi::handle(comm));  
    if (err != MPI_SUCCESS) throw mpi::mpi_error{err};  
}
```



Building the Interface

```
template <mpi::send_buffer SBuf,  
          mpi::convertible_to_mpi_handle<MPI_Comm> Comm>  
void send(SBuf&& sbuf, int dest, int tag, Comm const& comm) {  
    int err = MPI_Send(  
        mpi::ptr(sbuf), mpi::count(sbuf), mpi::type(sbuf),  
        dest, tag, mpi::handle(comm));  
    if (err != MPI_SUCCESS) throw mpi::mpi_error{err};  
}
```

any handle-like type: MPI_Comm, non-owning mpi::comm_view,
owning mpi::comm, or your own wrapper (via handle_traits)



Building the Interface

```
template <mpi::send_buffer SBuf,  
          mpi::convertible_to_mpi_handle<MPI_Comm> Comm>  
void send(SBuf&& sbuf, int dest, int tag, Comm const& comm) {  
    int err = MPI_Send(  
        mpi::ptr(sbuf), mpi::count(sbuf), mpi::type(sbuf),  
        dest, tag, mpi::handle(comm));  
    if (err != MPI_SUCCESS) throw mpi::mpi_error{err};  
}
```

Usage

```
std::vector<double> v = ...;  
  
mpi::send(v, dest, tag, MPI_COMM_WORLD);
```



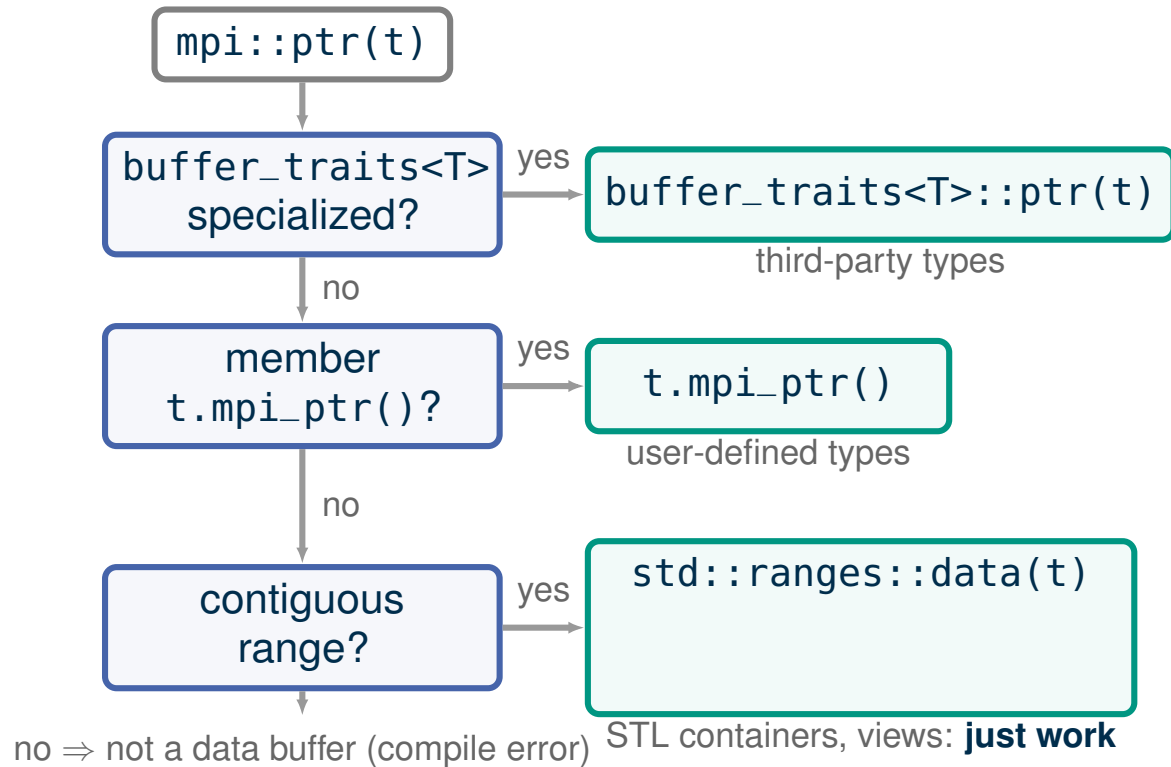
Building the Interface

```
template <mpi::send_buffer SBuf,  
          mpi::convertible_to_mpi_handle<MPI_Comm> Comm>  
void send(SBuf&& sbuf, int dest, int tag, Comm const& comm) {  
    int err = MPI_Send(  
        mpi::ptr(sbuf), mpi::count(sbuf), mpi::type(sbuf),  
        dest, tag, mpi::handle(comm));  
    if (err != MPI_SUCCESS) throw mpi::mpi_error{err};  
}
```

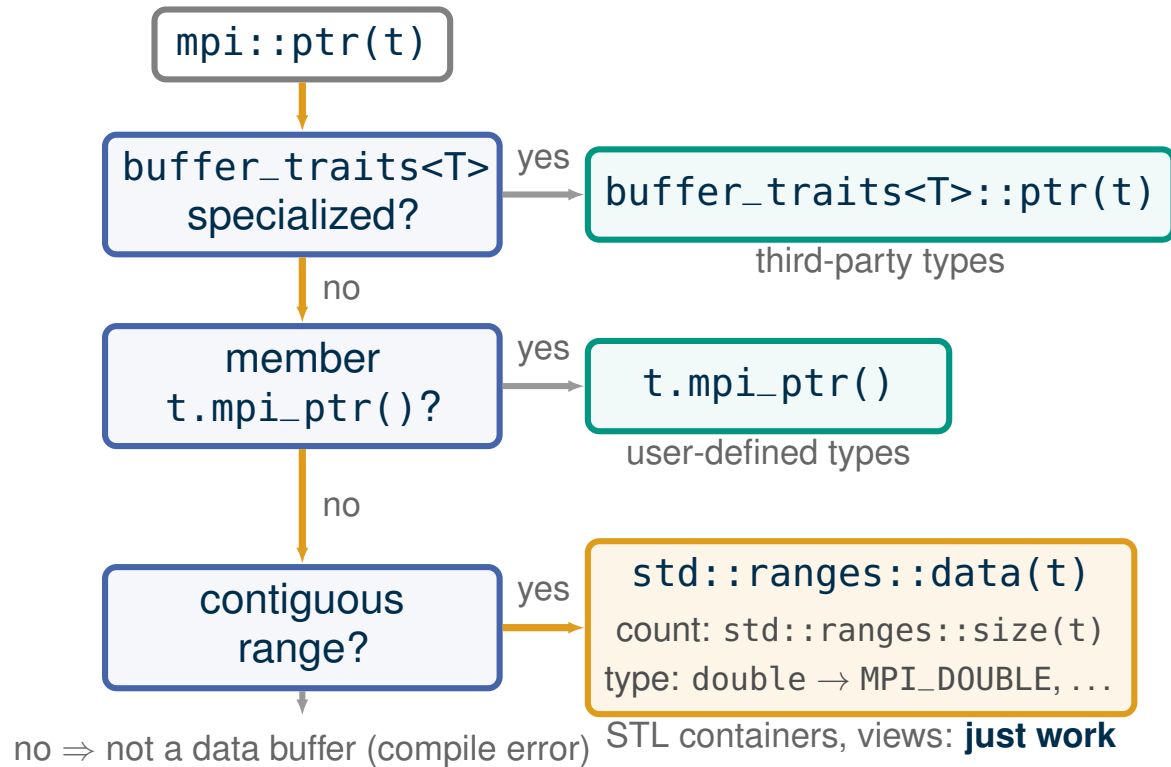
Usage

```
std::vector<double> v = ...;  
mpi::comm_view world{MPI_COMM_WORLD};  
mpi::comm my_comm = world.dup();  
mpi::send(v, dest, tag, my_comm);
```

Three-Level Buffer Dispatch



Three-Level Buffer Dispatch

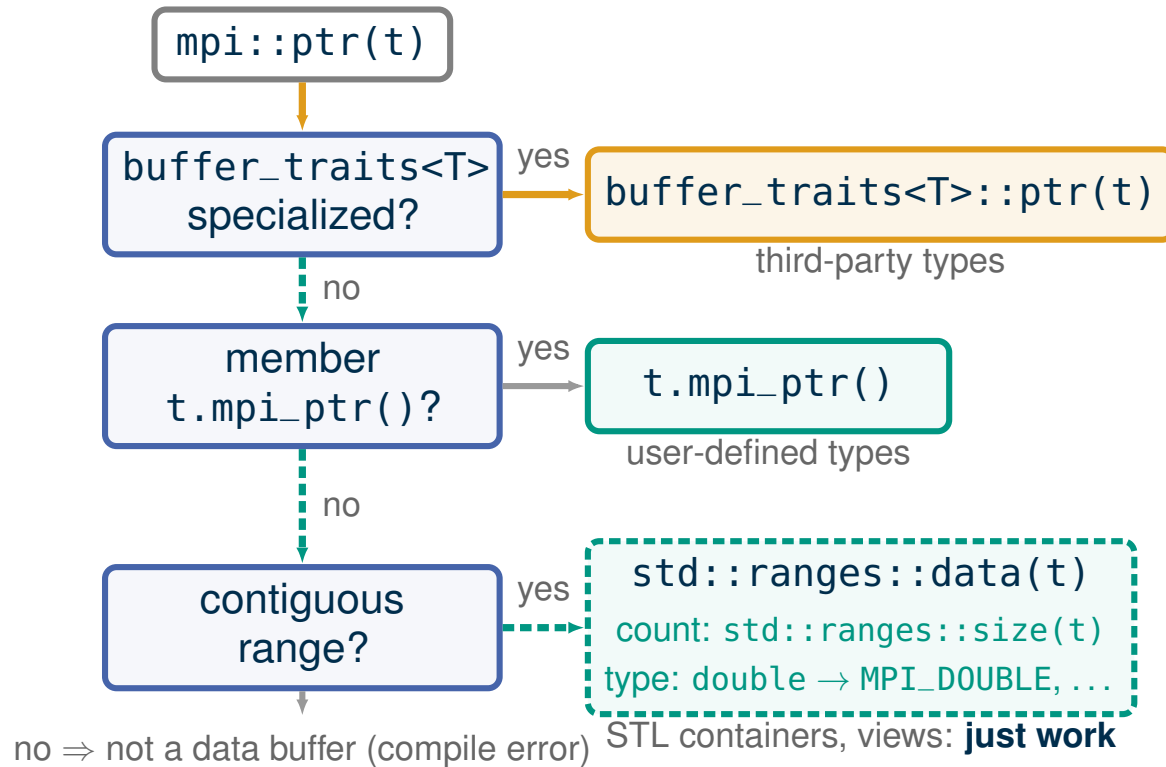


```
std::vector<double> v = ...;
```

```
mpi::send(v, dest, tag, comm);
```

```
// ptr, count, type:  
// all derived from the range
```

Three-Level Buffer Dispatch



```

template <typename T>
struct mpi::buffer_traits<
    thrust::device_vector<T>> {
    static T* ptr(
        thrust::device_vector<T>& v)
    { return v.data().get(); }
};
thrust::device_vector<double> v = ...;

```

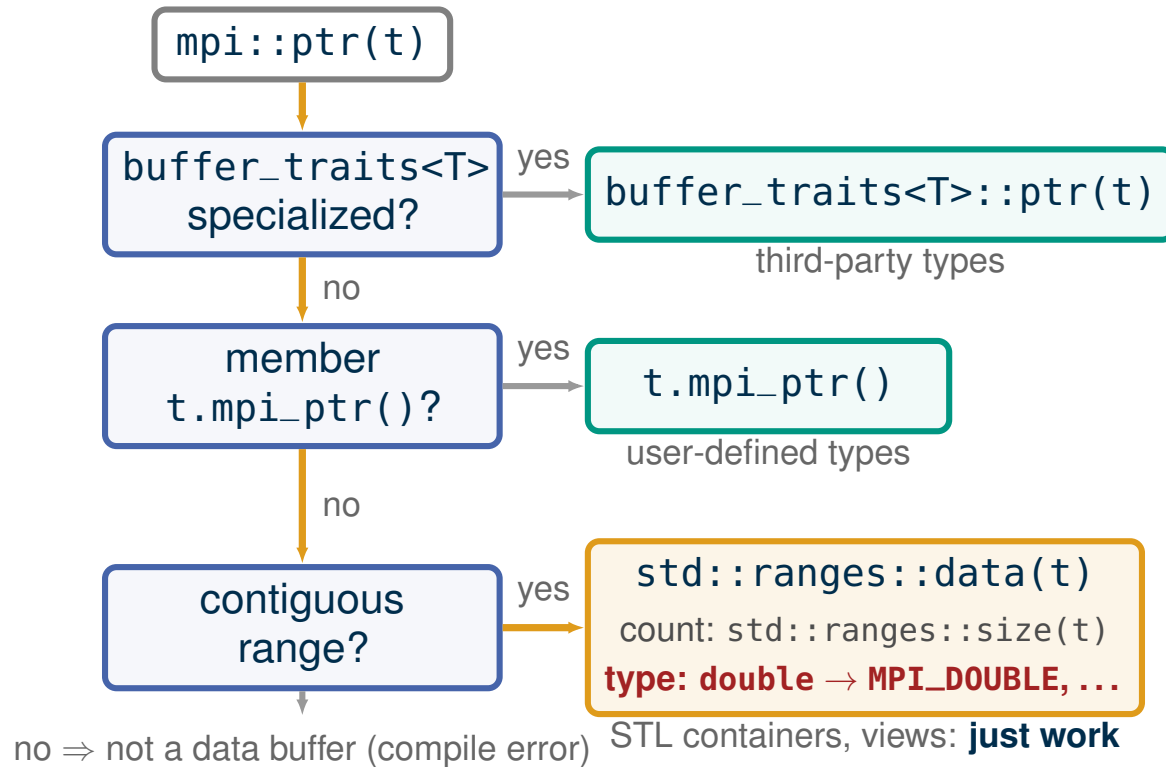
```
mpi::send(v, dest, tag, comm);
```

```

// count(), type(): no specialization
// -> range fallback

```

Three-Level Buffer Dispatch

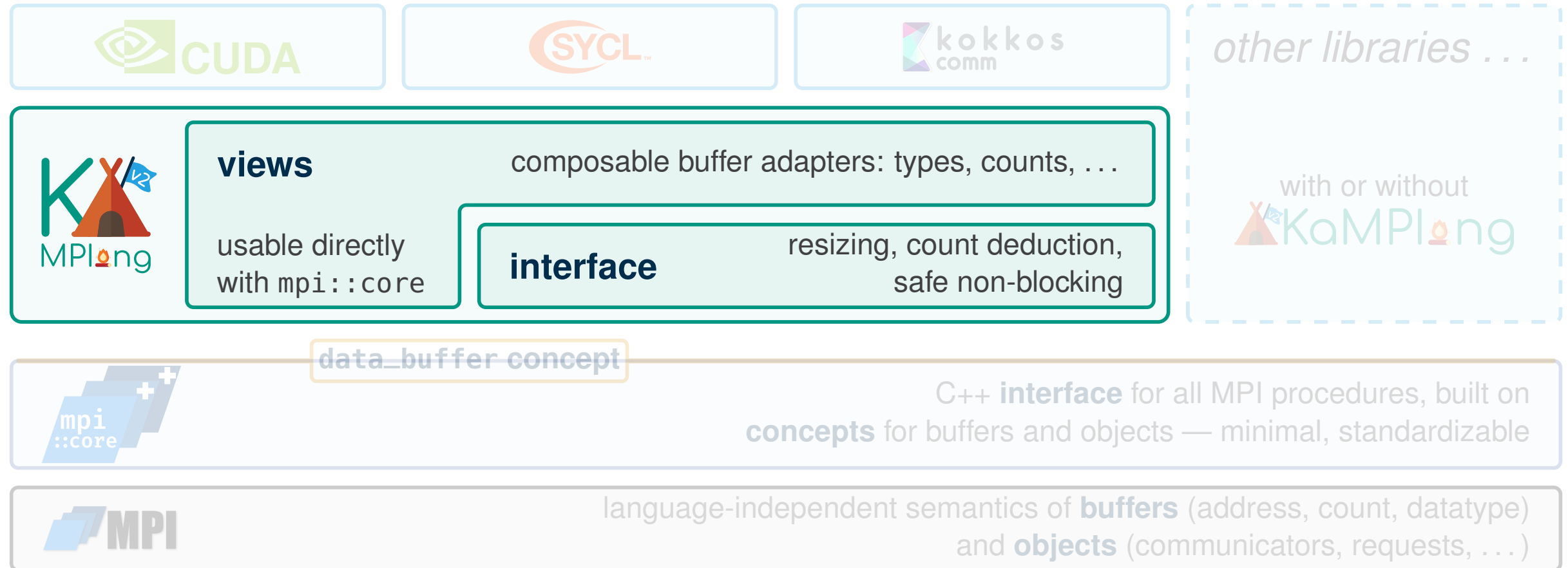


```
struct MyStruct { int x; double y; };
std::vector<MyStruct> v = ...;
```

```
mpi::send(v, dest, tag, comm);
```

```
error: no matching function for call to
'send(std::vector<MyStruct>&, int&, int&, mpi_communicator_t*&)'
• template argument deduction/substitution failed:
• constraints not satisfied
• required for the satisfaction of 'has_mpi_type<T>'
• required for the satisfaction of 'data_buffer<T>'
• required for the satisfaction of 'send_buffer<SBuf>'
• the required expression 'mpi::type(t)' is invalid
```

Standardizable Core, Convenience on Top





Composable Buffer Adapters

```
namespace views = kamping::v2::views;  
std::vector<double> data = /* ... */;  
  
// standard views satisfy the core concepts  
mpi::send(data | std::views::take(n),  
          dest, tag, comm);
```



Composable Buffer Adapters

```
namespace views = kamping::v2::views;
std::vector<double> data = /* ... */;

// standard views satisfy the core concepts
mpi::send(data | std::views::take(n),
          dest, tag, comm);

// attach/override MPI metadata,
// compose with std views
std::vector<MyStruct> s = /* ... */;
MPI_Datatype my_type; // MPI_Type_create_struct(...)
mpi::send(s | std::views::take(n)
          | views::with_type(my_type),
          dest, tag, comm);
```



Composable Buffer Adapters

```
namespace views = kamping::v2::views;
std::vector<double> data = /* ... */;

// standard views satisfy the core concepts
mpi::send(data | std::views::take(n),
          dest, tag, comm);

// attach/override MPI metadata,
// compose with std views
std::vector<MyStruct> s = /* ... */;
MPI_Datatype my_type; // MPI_Type_create_struct(...)
mpi::send(s | std::views::take(n)
          | views::with_type(my_type),
          dest, tag, comm);
```

```
// derived datatypes: layout reflected,
// committed on first use, freed with the pool
struct MyStruct { int x; double y; };
template <>
struct mpi::types::mpi_type_traits<MyStruct>
    : mpi::types::struct_type<MyStruct> {};

std::vector<MyStruct> v = /* ... */;
kamping::v2::type_pool pool;
mpi::send(v | views::with_auto_pool(pool),
          dest, tag, comm);
```



Automatic Counts and Resizing

```
std::vector<double> data = /* ... */;
std::vector<double> rdata;

std::vector<int> counts(comm.size()), displs(comm.size());
mpi::allgather(std::views::single(static_cast<int>(data.size())), counts, comm);
std::exclusive_scan(counts.begin(), counts.end(), displs.begin(), 0);
rdata.resize(counts.back() + displs.back());
mpi::allgatherv(data, rdata | views::with_counts(counts)
                | views::with_displs(displs), comm);
```



Automatic Counts and Resizing

```
std::vector<double> data = /* ... */;  
std::vector<double> rdata;
```

```
std::vector<int> counts(comm.size()), displs(comm.size());  
mpi::allgather(std::views::single(static_cast<int>(data.size()))  
               , counts, comm);  
std::exclusive_scan(counts.begin(), counts.end(), displs.begin(), 0);
```

```
mpi::allgatherv(data, rdata | views::with_counts(counts)  
               | views::with_displs(displs)  
               | views::resize_v, comm);
```



Automatic Counts and Resizing

```
std::vector<double> data = /* ... */;  
std::vector<double> rdata;
```

```
std::vector<int> counts(comm.size());  
mpi::allgather(std::views::single(static_cast<int>(data.size()))), counts, comm);
```

```
mpi::allgatherv(data, rdata | views::with_counts(counts)  
                | views::auto_displs()  
                | views::resize_v, comm);
```



Automatic Counts and Resizing

```
std::vector<double> data = /* ... */;  
std::vector<double> rdata;
```

```
mpi::allgatherv(data, rdata | views::auto_counts()  
| views::auto_displs()  
| views::resize_v, comm);
```



Automatic Counts and Resizing

```
std::vector<double> data = /* ... */;  
std::vector<double> rdata;
```

```
kamping::v2::allgatherv(data, rdata | views::auto_counts()  
                             | views::auto_displs()  
                             | views::resize_v, comm);
```



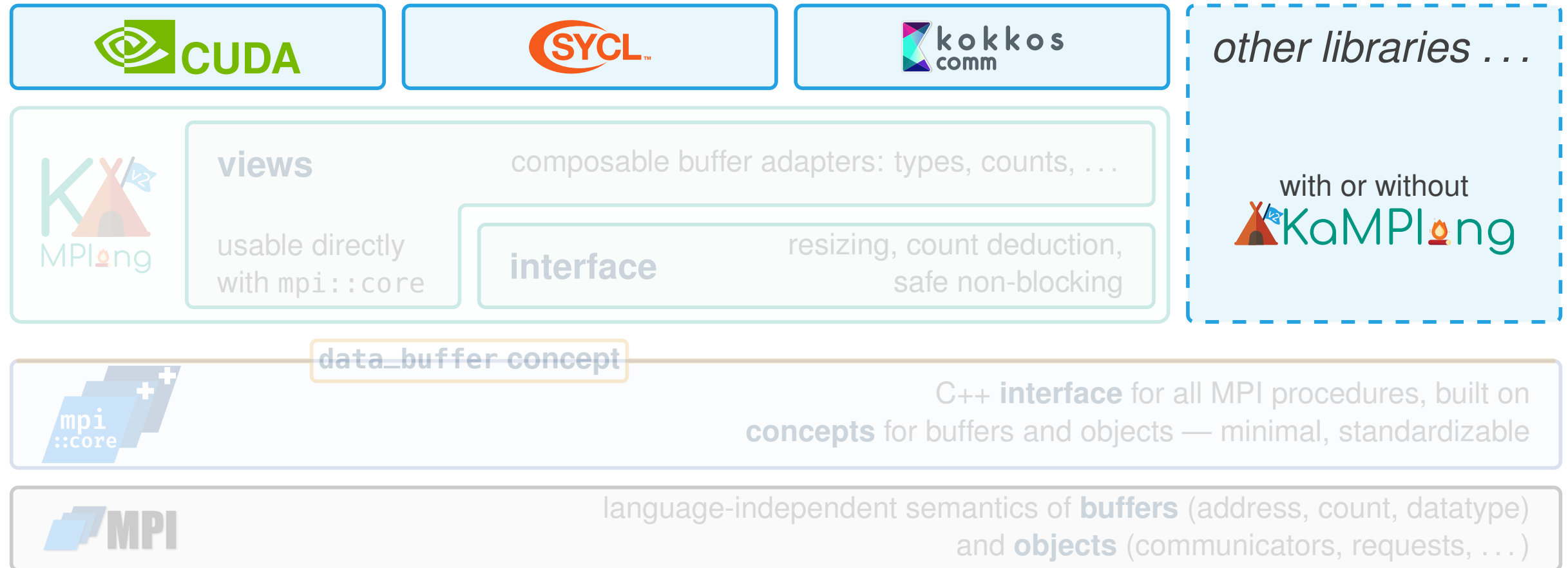
Automatic Counts and Resizing

```
std::vector<double> data = /* ... */;  
std::vector<double> rdata;
```

```
kamping::v2::allgatherv(data, rdata | views::auto_counts()  
                           | views::auto_displs()  
                           | views::resize_v, comm);
```

```
auto allgatherv(auto&& sbuf, auto&& rbuf, auto comm) {  
    infer(comm_op::allgatherv{}, sbuf, rbuf, comm);  
    mpi::allgatherv(sbuf, rbuf, comm);  
    return result{fwd(sbuf), fwd(rbuf)};  
}
```

Standardizable Core, Convenience on Top



Bridging to Heterogeneous Ecosystems



```
// raw device memory: wrap in std::span
int* d; cudaMalloc(&d, n * sizeof(int));
mpi::send(std::span{d, n}, dst, tag, comm);
```

Bridging to Heterogeneous Ecosystems



```
// raw device memory: wrap in std::span
int* d; cudaMalloc(&d, n * sizeof(int));
mpi::send(std::span{d, n}, dst, tag, comm);

// Thrust device vector
thrust::device_vector<int> v(n);
mpi::send(v, dst, tag, comm);
```

Bridging to Heterogeneous Ecosystems



```
// raw device memory: wrap in std::span
int* d; cudaMalloc(&d, n * sizeof(int));
mpi::send(std::span{d, n}, dst, tag, comm);

// Thrust device vector
thrust::device_vector<int> v(n);
mpi::send(v, dst, tag, comm);

// buffer_traits only needs ptr(),
// rest is automatic
template <typename T, typename A>
using dv = thrust::device_vector<T, A>;
template <typename T, typename A>
struct mpi::buffer_traits<dv<T, A>> {
    static T* ptr(dv<T, A>& v)
    { return v.data().get(); }
    static T const* ptr(dv<T, A> const& v)
    { return v.data().get(); }
};
```

Bridging to Heterogeneous Ecosystems



```
// raw device memory: wrap in std::span
int* d; cudaMalloc(&d, n * sizeof(int));
mpi::send(std::span{d, n}, dst, tag, comm);

// Thrust device vector
thrust::device_vector<int> v(n);
mpi::send(v, dst, tag, comm);

// buffer_traits only needs ptr(),
// rest is automatic
template <typename T, typename A>
using dv = thrust::device_vector<T, A>;
template <typename T, typename A>
struct mpi::buffer_traits<dv<T, A>> {
    static T* ptr(dv<T, A>& v)
    { return v.data().get(); }
    static T const* ptr(dv<T, A> const& v)
    { return v.data().get(); }
};
```



```
// USM: vector with USM allocator
using alloc_t = sycl::usm_allocator<
    int, sycl::usm::alloc::shared>;
std::vector<int, alloc_t> v(n, alloc_t{q});
mpi::send(v, dst, tag, comm);
```

Bridging to Heterogeneous Ecosystems



```
// raw device memory: wrap in std::span
int* d; cudaMalloc(&d, n * sizeof(int));
mpi::send(std::span{d, n}, dst, tag, comm);

// Thrust device vector
thrust::device_vector<int> v(n);
mpi::send(v, dst, tag, comm);

// buffer_traits only needs ptr(),
// rest is automatic
template <typename T, typename A>
using dv = thrust::device_vector<T, A>;
template <typename T, typename A>
struct mpi::buffer_traits<dv<T, A>> {
    static T* ptr(dv<T, A>& v)
    { return v.data().get(); }
    static T const* ptr(dv<T, A> const& v)
    { return v.data().get(); }
};
```



```
// USM: vector with USM allocator
using alloc_t = sycl::usm_allocator<
    int, sycl::usm::alloc::shared>;
std::vector<int, alloc_t> v(n, alloc_t{q});
mpi::send(v, dst, tag, comm);

// buffer: send from a host task
sycl::buffer<int> buf(...);
q.submit([&](sycl::handler& h) {
    sycl::accessor acc{buf, h};
    h.host_task(
        [=](sycl::interop_handle ih) {
            mpi::send(acc | views::sycl(ih),
                      dest, tag, comm);
        });
});
```

Bridging to Heterogeneous Ecosystems



```
// raw device memory: wrap in std::span
int* d; cudaMalloc(&d, n * sizeof(int));
mpi::send(std::span{d, n}, dst, tag, comm);

// Thrust device vector
thrust::device_vector<int> v(n);
mpi::send(v, dst, tag, comm);

// buffer_traits only needs ptr(),
// rest is automatic
template <typename T, typename A>
using dv = thrust::device_vector<T, A>;
template <typename T, typename A>
struct mpi::buffer_traits<dv<T, A>> {
    static T* ptr(dv<T, A>& v)
    { return v.data().get(); }
    static T const* ptr(dv<T, A> const& v)
    { return v.data().get(); }
};
```



```
// USM: vector with USM allocator
using alloc_t = sycl::usm_allocator<
    int, sycl::usm::alloc::shared>;
std::vector<int, alloc_t> v(n, alloc_t{q});
mpi::send(v, dst, tag, comm);

// buffer: send from a host task
sycl::buffer<int> buf(...);
q.submit([&](sycl::handler& h) {
    sycl::accessor acc{buf, h};
    h.host_task(
        [=](sycl::interop_handle ih) {
            mpi::send(acc | views::sycl(ih),
                dest, tag, comm);
        });
});
```



```
namespace views = kamping::v2::views;
Kokkos::View<int*> sv(n);
mpi::send(sv | views::kokkos,
    dest, tag, comm);
Kokkos::View<int*> rv(0);
kamping::v2::recv(
    rv | views::kokkos | views::resize,
    src, tag, comm);
```

Bridging to Heterogeneous Ecosystems



```
// raw device memory: wrap in std::span
int* d; cudaMalloc(&d, n * sizeof(int));
mpi::send(std::span{d, n}, dst, tag, comm);

// Thrust device vector
thrust::device_vector<int> v(n);
mpi::send(v, dst, tag, comm);

// buffer_traits only needs ptr(),
// rest is automatic
template <typename T, typename A>
using dv = thrust::device_vector<T, A>;
template <typename T, typename A>
struct mpi::buffer_traits<dv<T, A>> {
    static T* ptr(dv<T, A>& v)
    { return v.data().get(); }
    static T const* ptr(dv<T, A> const& v)
    { return v.data().get(); }
};
```



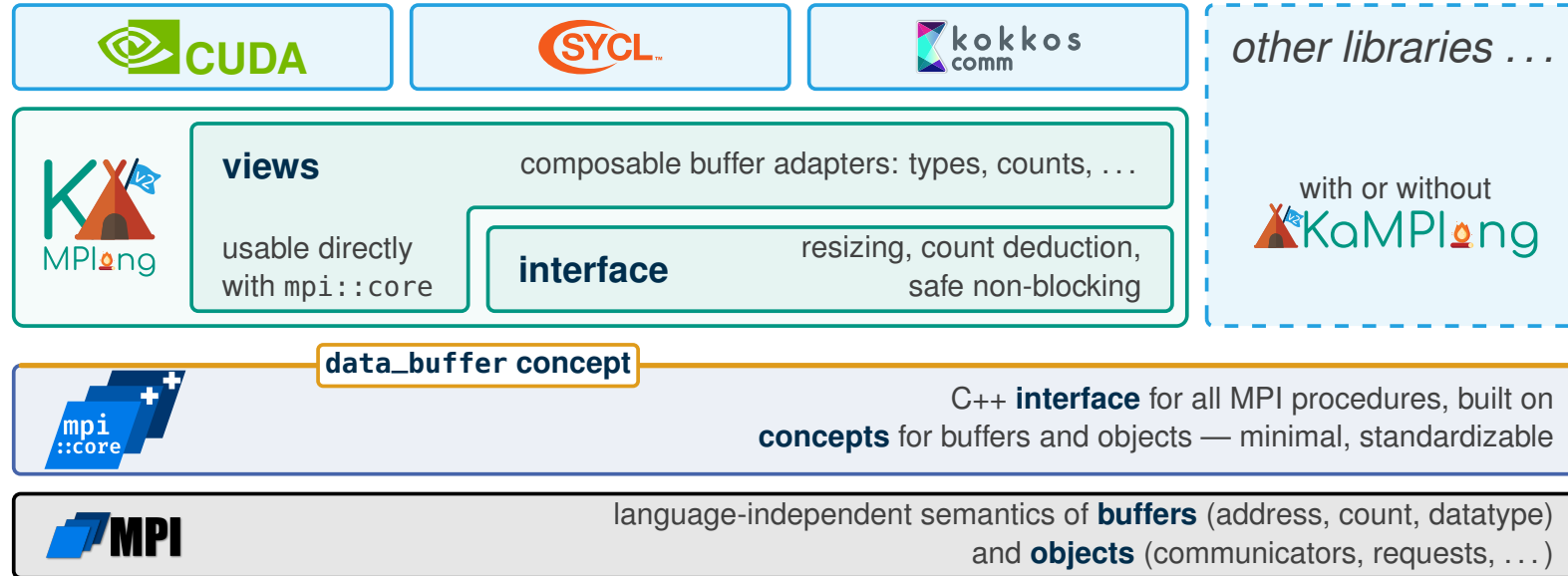
```
// USM: vector with USM allocator
using alloc_t = sycl::usm_allocator<
    int, sycl::usm::alloc::shared>;
std::vector<int, alloc_t> v(n, alloc_t{q});
mpi::send(v, dst, tag, comm);

// buffer: send from a host task
sycl::buffer<int> buf(...);
q.submit([&](sycl::handler& h) {
    sycl::accessor acc{buf, h};
    h.host_task(
        [=](sycl::interop_handle ih) {
            mpi::send(acc | views::sycl(ih),
                dest, tag, comm);
        });
});
```



```
namespace views = kamping::v2::views;
Kokkos::View<int*> sv(n);
mpi::send(sv | views::kokkos,
    dest, tag, comm);
Kokkos::View<int*> rv(0);
kamping::v2::recv(
    rv | views::kokkos | views::resize,
    src, tag, comm);
```

Conclusion



kamping-site/kamping-v2

Future work

- one-sided, partitioned, ... communication
- **find users**
- align buffer protocols across languages (`rsmpi`, `MPI.jl`, `mpi4py`, ...)
- build the dSTL

PS: I am finishing my PhD soon — if you know somebody who knows somebody who might have a position ... let's talk!



Building the Interface: Variable-Size Collectives

```
template <mpi::send_buffer SBuf, mpi::recv_buffer_v RBuf,  
          mpi::convertible_to_mpi_handle<MPI_Comm> Comm>  
void allgatherv(SBuf&& sbuf, RBuf&& rbuf, Comm const& comm) {  
    int err = MPI_Allgatherv(  
        mpi::ptr(sbuf), mpi::count(sbuf), mpi::type(sbuf),  
        mpi::ptr(rbuf),  
        std::ranges::data(mpi::counts(rbuf)),  
        std::ranges::data(mpi::displs(rbuf)),  
        mpi::type(rbuf), mpi::handle(comm));  
    if (err != MPI_SUCCESS) throw mpi::mpi_error{err};  
}
```

```
template <typename T>  
concept count_range =  
    std::ranges::contiguous_range<T> &&  
    std::same_as<int>, std::remove_cvref_t<  
        std::ranges::range_value_t<T>>>;
```

```
template <typename T>  
concept data_buffer_v = data_buffer<T> &&  
    requires(T const& t) {  
        { mpi::counts(t) } -> count_range;  
        { mpi::displs(t) } -> count_range;  
    };
```

Usage

```
std::vector<double> sdata = ..., rdata = ...;  
std::vector<int> rcounts = ..., rdispls = ...;  
mpi::mpi_span_v rbuf{rdata, rcounts, rdispls};  
mpi::allgatherv(sdata, rbuf, MPI_COMM_WORLD);
```



Safe Non-Blocking Operations

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

```
auto req = kamping::v2::irecv(          recv_buf , ...);
```

```
// ... other computations
```

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



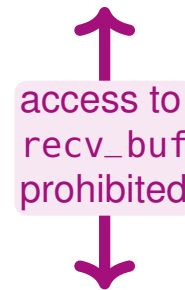
Safe Non-Blocking Operations

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

```
auto req = kamping::v2::irecv(recv_buf , ...);
```

```
// ... other computations
```

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





Safe Non-Blocking Operations

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

transfer ownership to library

```
auto req = kamping::v2::irecv(std::move(recv_buf), ...);
```

```
// ... other computations
```

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

↑
access to
recv_buf
prohibited
↓



Safe Non-Blocking Operations

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

transfer ownership to library

```
auto req = kamping::v2::irecv(std::move(recv_buf), ...);
```

```
    \- kamping::v2::iresult<std::vector<int>>
```

```
// ... other computations
```

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

↑
access to
recv_buf
prohibited
↓



Safe Non-Blocking Operations

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

transfer ownership to library

```
auto req = kamping::v2::irecv(std::move(recv_buf), ...);
```

```
    \- kamping::v2::iresult<std::vector<int>>
```

```
// ... other computations
```

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

return ownership

↑
access to
recv_buf
prohibited
↓



Safe Non-Blocking Operations

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

transfer ownership to library

```
auto req = kamping::v2::irecv(std::move(recv_buf), ...);
```

```
    kamping::v2::iresult<std::vector<int>>
```

```
// ... other computations
```

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

return ownership

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

↑
access to
recv_buf
prohibited
↓