High-Performance
Computing Center
Stuttgart

MPPI – Type safe C++ Datatypes for MPI

Mike Söhner, Christoph Niethammer

41st Workshop on Sustained Simulation Performance, Stuttgart, April 23rd 2026

Outline

- Background
- Motivation
- MPPI
- Evaluation
- Application example
- Outlook

Background – Point-to-Point Message Passing

```
int value;

if (rank == 0)
{
    value = 42;
    MPI_Send(&value, 1, MPI_INT, 1,
    0, MPI_COMM_WORLD);
}
```

Message Passing

```
int value;

if (rank == 1)
{
    MPI_Recv(&value, 1, MPI_INT, 0,
    0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

Background – Point-to-Point Message Passing

```
int value;

if (rank == 0)
{
    value = 42;
    MPI_Send(&value, 1, MPI_INT, 1,
    0, MPI_COMM_WORLD);
}
```

Message Passing

```
int value;

if (rank == 1)
{
    MPI_Recv(&value, 1, MPI_INT, 0,
    0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

Background – MPI Derived Datatypes

```
MyClass mc;

if (rank == 0)
{
    mc = {{4.2, 1.5}, 3.2, ‚a‘, 5};
    MPI_Send(&mc, 1, ?, 1,
    0, MPI_COMM_WORLD);
}
```

Message Passing

```
MyClass mc;

if (rank == 1)
{
    MPI_Recv(&mc, 1, ?, 0,
    0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n;
};
```

Motivation

- MPI Datatypes can be arbitrarily complex, but their creation is cumbersome.
- Three arrays of information are necessary:

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n;
};
```

```
MPI_Datatype T[] = {
        MPI_DOUBLE,
        MPI_DOUBLE,
        MPI_CHAR,
        MPI_INT
};

int B[] = { 2, 1, 1, 1 };

MPI_Aint D[] = {
        offsetof(MyClass, x),
        offsetof(MyClass, y),
        offsetof(MyClass, a),
        offsetof(MyClass, n)
};

MPI_Datatype mpi_dt_custom;
MPI_Type_create_struct(4, B, D, T, &mpi_dt_custom);
MPI_Type_commit(&mpi_dt_custom);
```

Motivation

- MPI Datatypes can be arbitrarily complex, but their creation is cumbersome.
- Three arrays of information are necessary:
  - Type information.

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n;
};
```

```
MPI_Datatype T[] = {
        MPI_DOUBLE,
        MPI_DOUBLE,
        MPI_CHAR,
        MPI_INT
};

int B[] = { 2, 1, 1, 1 };

MPI_Aint D[] = {
        offsetof(MyClass, x),
        offsetof(MyClass, y),
        offsetof(MyClass, a),
        offsetof(MyClass, n)
};

MPI_Datatype mpi_dt_custom;
MPI_Type_create_struct(4, B, D, T, &mpi_dt_custom);
MPI_Type_commit(&mpi_dt_custom);
```

Motivation

- MPI Datatypes can be arbitrarily complex, but their creation is cumbersome.
- Three arrays of information are necessary:
  - Type information.
  - Number of types.

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n;
};
```

```
MPI_Datatype T[] = {
        MPI_DOUBLE,
        MPI_DOUBLE,
        MPI_CHAR,
        MPI_INT
};

int B[] = { 2, 1, 1, 1 };

MPI_Aint D[] = {
        offsetof(MyClass, x),
        offsetof(MyClass, y),
        offsetof(MyClass, a),
        offsetof(MyClass, n)
};

MPI_Datatype mpi_dt_custom;
MPI_Type_create_struct(4, B, D, T, &mpi_dt_custom);
MPI_Type_commit(&mpi_dt_custom);
```

Motivation

- MPI Datatypes can be arbitrarily complex, but their creation is cumbersome.
- Three arrays of information are necessary:
  - Type information.
  - Number of types.
  - Setoff of types.

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n;
};
```

```
MPI_Datatype T[] = {
        MPI_DOUBLE,
        MPI_DOUBLE,
        MPI_CHAR,
        MPI_INT
};

int B[] = { 2, 1, 1, 1 };

MPI_Aint D[] = {
        offsetof(MyClass, x),
        offsetof(MyClass, y),
        offsetof(MyClass, a),
        offsetof(MyClass, n)
};

MPI_Datatype mpi_dt_custom;
MPI_Type_create_struct(4, B, D, T, &mpi_dt_custom);
MPI_Type_commit(&mpi_dt_custom);
```

Motivation

- MPI Datatypes can be arbitrarily complex, but their creation is cumbersome.
- Three arrays of information are necessary:
  - Type information.
  - Number of types.
  - Setoff of types.
- Data from a class can be omitted from sending.
- For example:

```
class MyClass {
        std::array<double, 2> x;
        double y;
        ~~char a;~~
        int n;
};
```

```
MPI_Datatype T[] = {
        MPI_DOUBLE,
        MPI_DOUBLE,
        ~~MPI_CHAR,~~
        MPI_INT
};

int B[] = { 2, 1, ~~1,~~ 1 };

MPI_Aint D[] = {
        offsetof(MyClass, x),
        offsetof(MyClass, y),
        ~~offsetof(MyClass, a),~~
        offsetof(MyClass, n)
};

MPI_Datatype mpi_dt_custom;
MPI_Type_create_struct(3, B, D, T, &mpi_dt_custom);
MPI_Type_commit(&mpi_dt_custom);
```

Our Contribution: MPPI

- Development of C++-based Datatype engine.
  - Replaces MPI Datatype engine.
- Simplified interface using C++26 Reflection.
- Compile-time optimizations using templates.
- Resolving of non-trivially-copyable data members.

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n;
};
```

```
constexpr mppi::Pattern<MyClass, "x", "y", "n"> pattern;
```

https://github.com/mikesoehner/MPPI/

Moving to C++ – Related Work

- C++ interfaces for MPI are not new.
- Previous works in that area:
  - Original C++ bindings by the MPI Standard. (Deprecated in MPI 3.0).
  - Boost (Supports MPI 1.0).
  - MPL (Header-only library utilizing C++17-based bindings).
  - KaMPIng (Modern (near) zero-overhead C++ bindings library for MPI).

Moving to C++ – Specialization with Concepts

H L R S

General send function for communicating a container.

```
template<typename Container>
void send(int dest, int tag, Container& container);
```

Same behaviour for vector and list?

Overloaded send functions for communicating a container.

```
template<typename T>
void send(int dest, int tag, std::vector<T>& container);
```

```
template<typename T>
void send(int dest, int tag, std::list<T>& container);
```

Moving to C++ – Specialization with Concepts

General send function for communicating a container.

```
template<typename Container>
void send(int dest, int tag, Container& container);
```

Same behaviour for vector and list?

Overloaded send functions for communicating a container.

```
template<typename T>
void send(int dest, int tag, std::vector<T>& container);
```

```
template<typename T>
void send(int dest, int tag, std::list<T>& container);
```

Select specialized function using Concepts

Specialized send function for consecutive-memory container.

```
template <typename Range>
            requires std::ranges::input_range<Range> &&
                     std::ranges::contiguous_range<Range>
void send(Destination dest, Tag tag, Range& range);
```

Moving to C++ – Specialization with Concepts

H L R S

General send function for communicating a container.

```
template<typename Container>
void send(int dest, int tag, Container& container);
```

Same behaviour for vector and list?

Overloaded send functions for communicating a container.

```
template<typename T>
void send(int dest, int tag, std::vector<T>& container);
```

```
template<typename T>
void send(int dest, int tag, std::list<T>& container);
```

Select specialized function using Concepts

Specialized send function for consecutive-memory container.

```
template <typename Range>
            requires std::ranges::input_range<Range> &&
                     std::ranges::contiguous_range<Range>
void send(Destination dest, Tag tag, Range& range);
```

Specialized send function for consecutive-memory container of trivially-copyable types.

```
template <typename Range>
   requires std::ranges::input_range<Range> &&
            std::ranges::contiguous_range<Range> &&
            std::is_trivially_copyable_v
                < typename std::ranges::range_value_t<Range> >
void send(Destination dest, Tag tag, Range& range);
```

Moving to C++ – Reflection

- Reflection is a feature voted in for C++26.
- Enables type introspection.
- Available via experimental compiler.

```
class MyClass {
        std::array<double, 2> x;
        double y;
        ~~char a;~~
        int n;
};
```

Basic Functions offered by Reflection:

```
nonstatic_data_members_of(^^MyClass);
type_of(^^MyClass);
size_of(^^MyClass);
alignment_of(^^MyClass);
offset_of(^^MyClass);
identifier_of(^^MyClass);
```

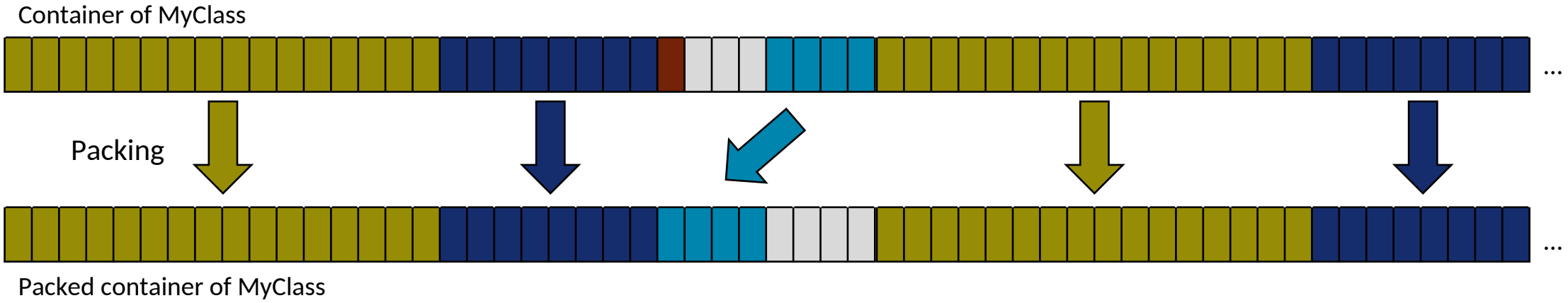
Necessary Information for Datatype engine:

```
for (member : nonstatic_data_members_of(^^MyClass))
{
        if ("identifier" == identifier_of(member)) {
                type_of(member);
                size_of(member);
                alignment_of(member);
                offset_of(member);}
}
```

Our MPPI interface using Reflection:

```
constexpr mppi::Pattern<MyClass, "x", "y", "n"> pattern;
```

MPPI: Template-Based Serialization




```
class MyClass {
        std::array<double, 2> x;
        double y;
        ~~char a;~~
        int n;
};
```

MPI

```
for(i : count)
        memcpy(dest, src + D[i],
        sizeof(T[i]) * B[i]);
```

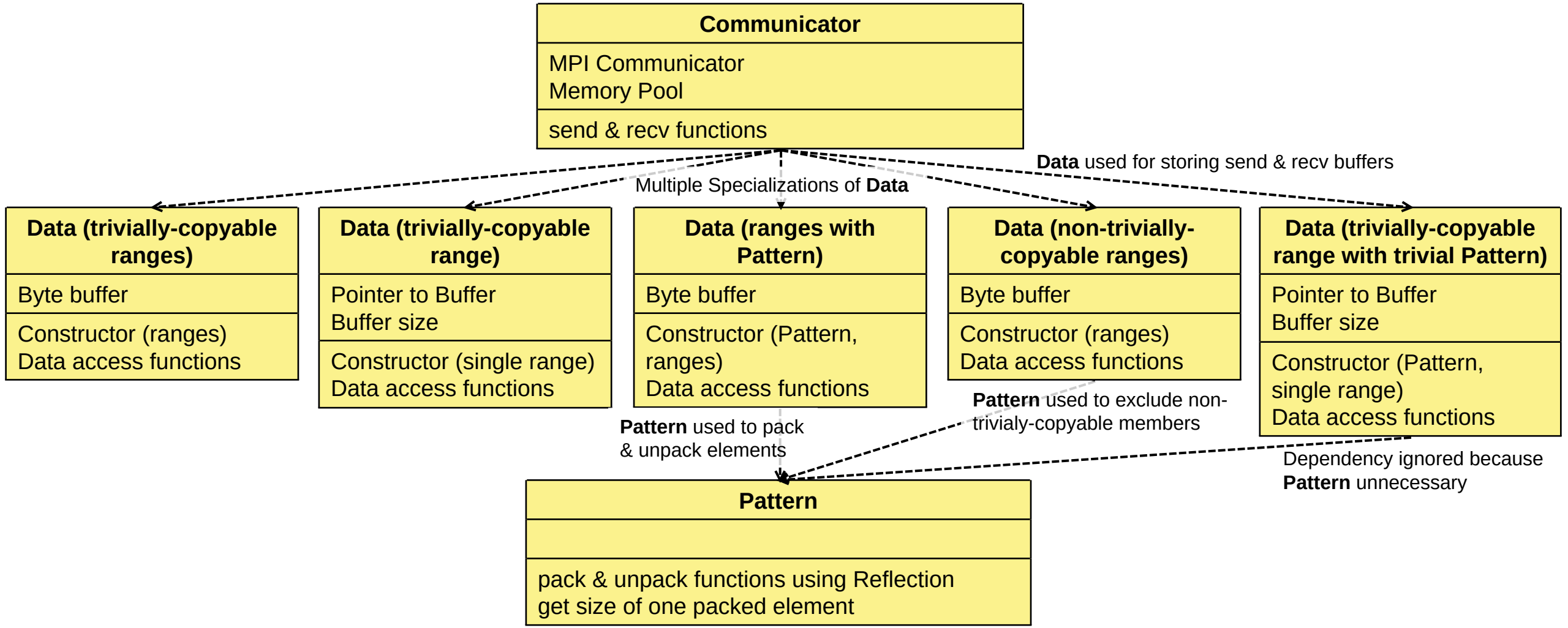
▶ Reads 3 ints per component in derived Datatype.

MPPI

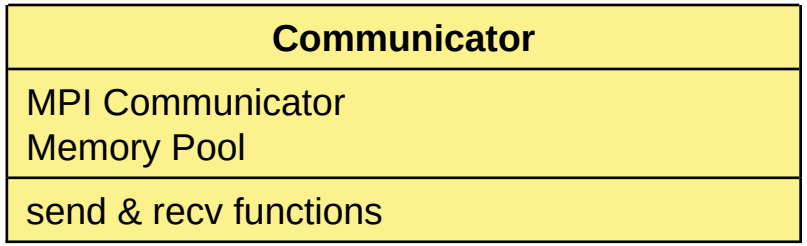
```
memcpy(dest, src + 0, 16);
memcpy(dest, src + 16, 8);
memcpy(dest, src + 28, 4);
```

▶ Information is inlined, no reads necessary.

Implementation Details



Implementation Details



Send and recv functions use range semantics.

```
std::vector<int> vec(10);

// Sending the entire vector
mppi_comm.send(mppi::Dest(1), mppi::Tag(99), vec);

// Sending only the first three elements of the list
mppi_comm.send(mppi::Dest(1), mppi::Tag(99), vec | std::ranges::views::take(3));

// Sending only elements with an even value
mppi_comm.send(mppi::Dest(1), mppi::Tag(99), vec | std::ranges::views::filter([](int i) { return 0 == i % 2; }));
```

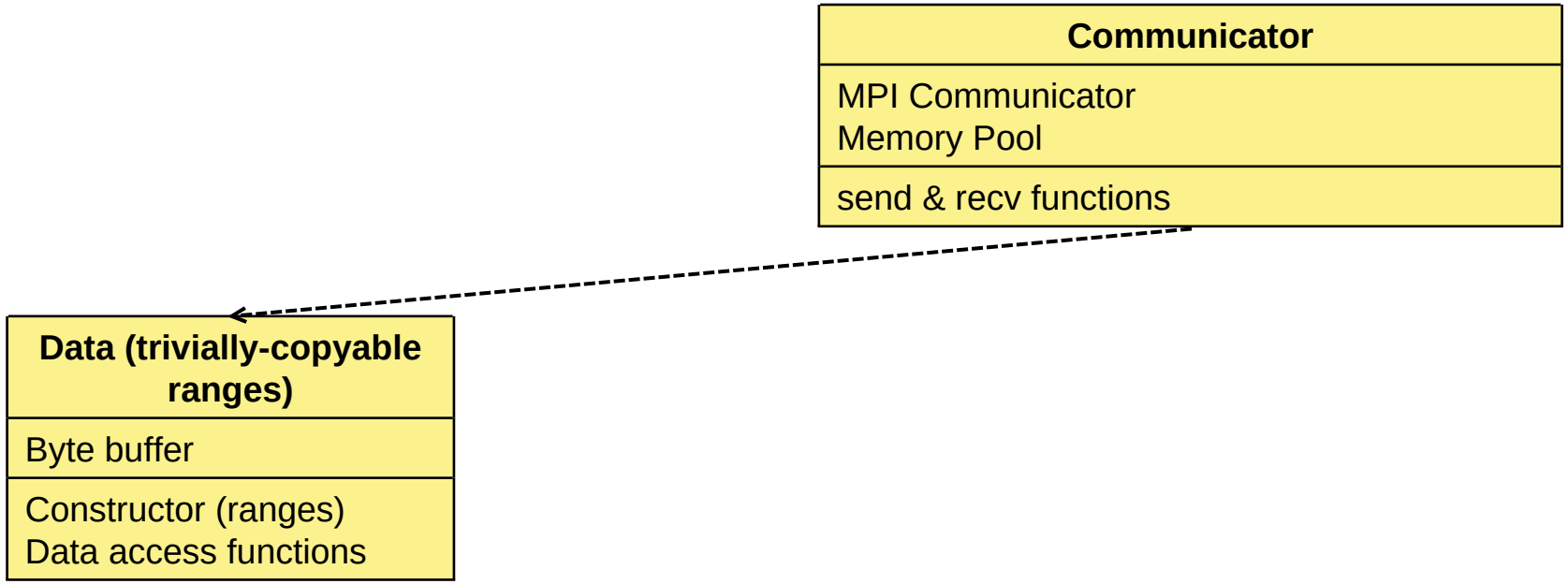
Implementation Details

Pattern creation requires the identifiers of wanted data members.

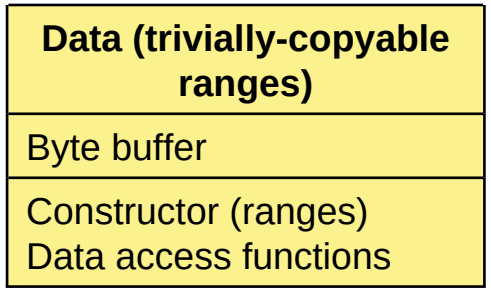
```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n; };


constexpr mppi::Pattern<MyClass, "x", "y", "n"> pattern;
```

Pattern
pack & unpack functions using Reflection get size of one packed element

Implementation Details



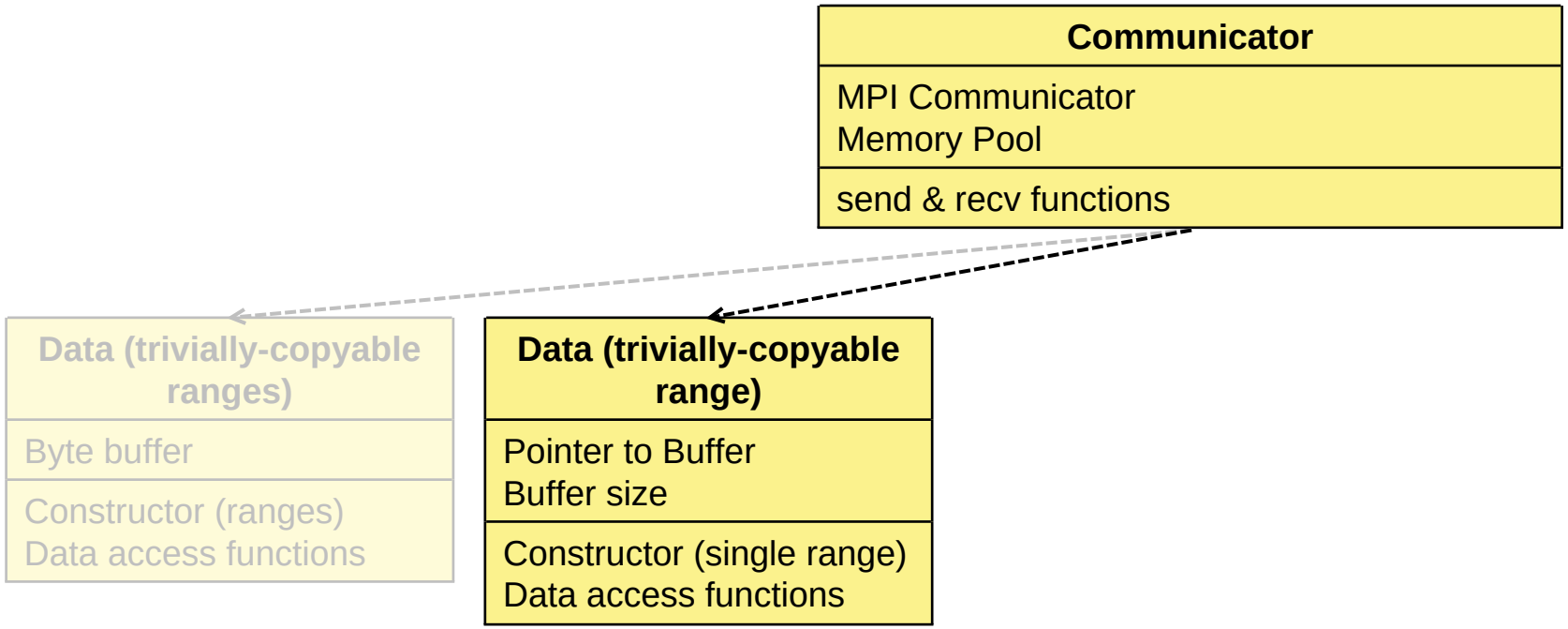
Implementation Details



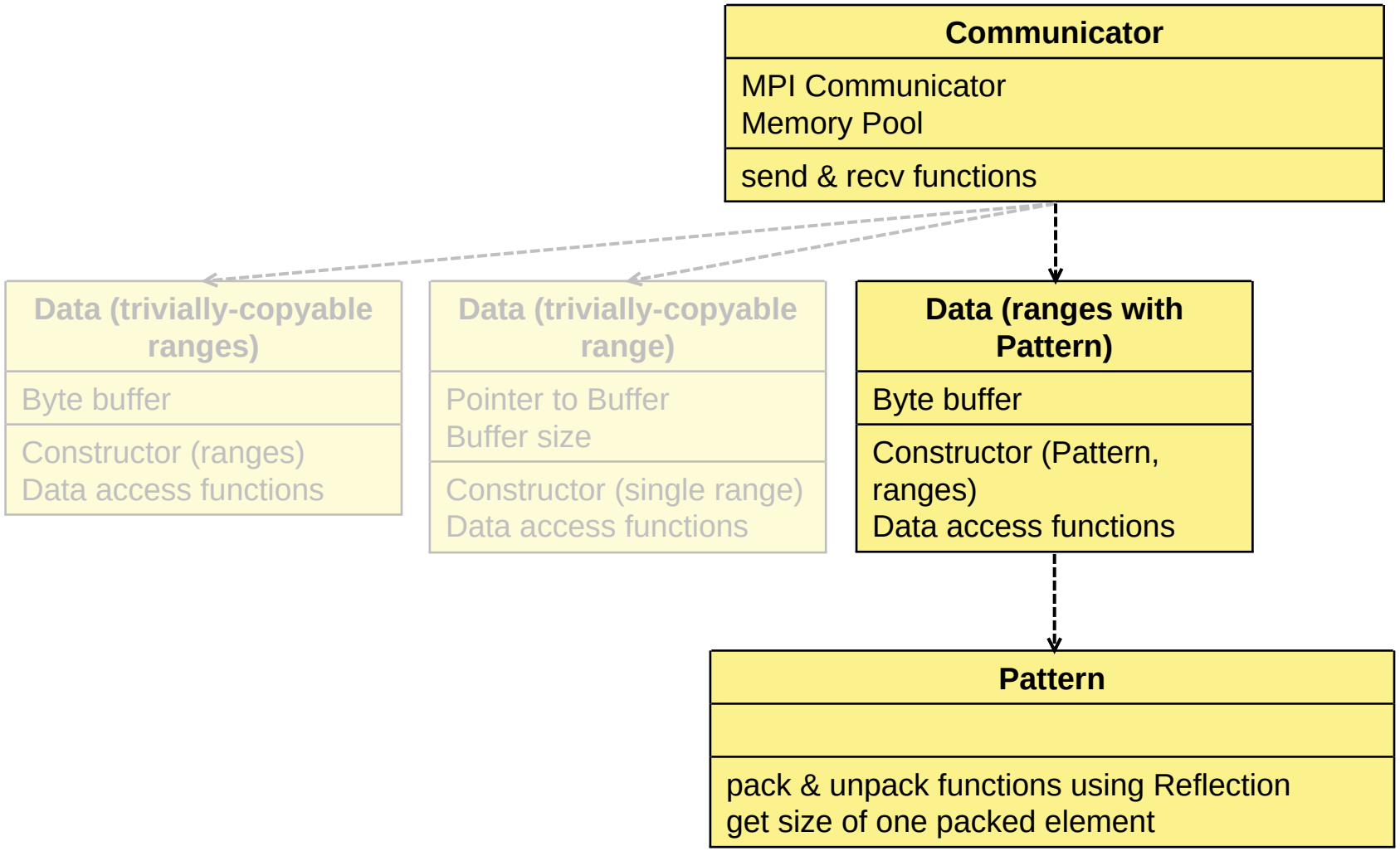
Used for one or multiple ranges of trivially-copyable elements.

```
std::vector<int> vec(10);
std::list<int> list(8);
mppi::Communicator mppi_comm;

if (mppi_comm.get_rank() == 0)
        mppi_comm.send(mppi::Destination(1), mppi::Tag(0), vec, list);
else
        mppi_comm.recv(mppi::Source(0), mppi::Tag(0), vec, list);
```

Implementation Details



Implementation Details



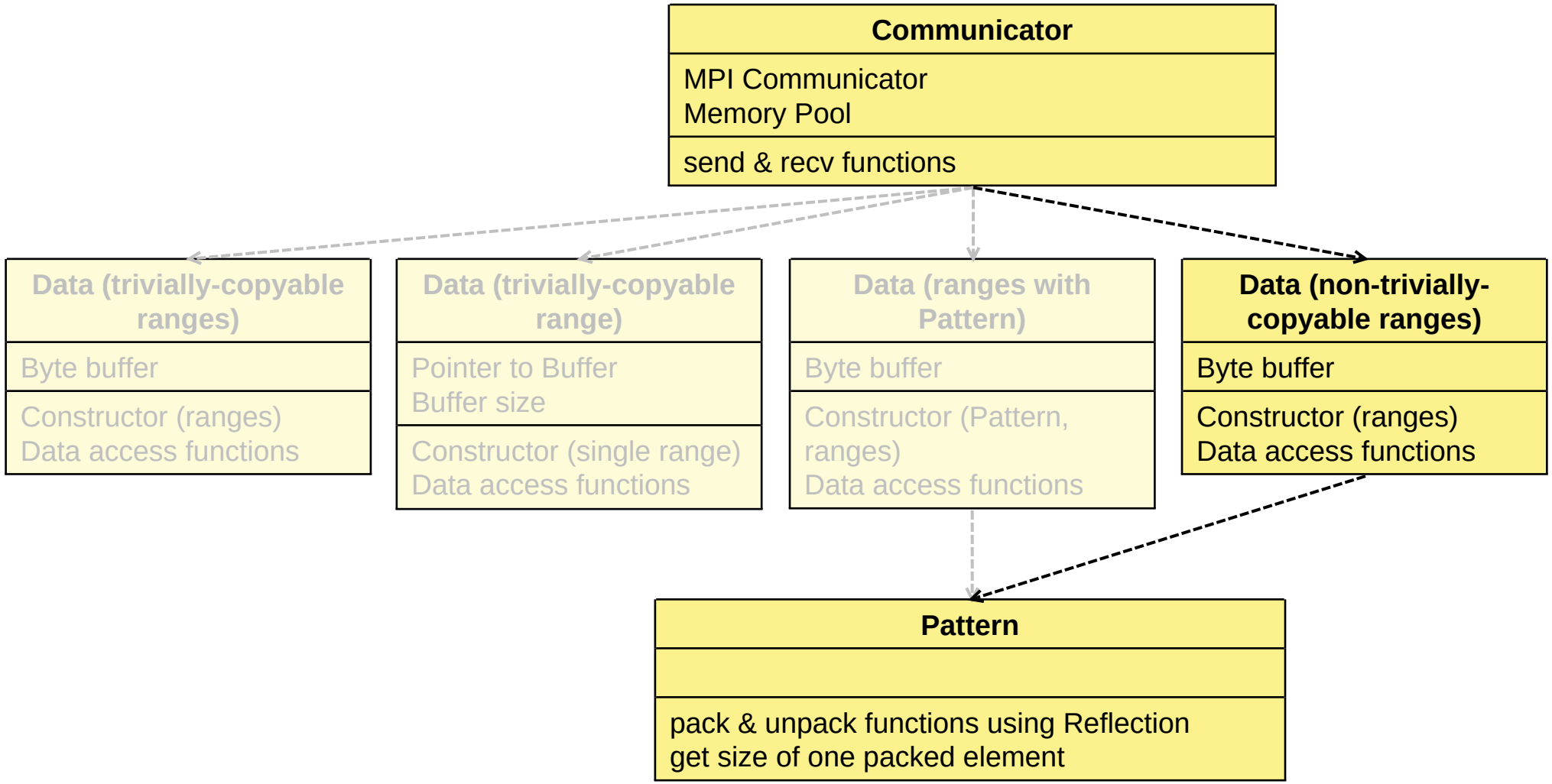
Implementation Details

Data (ranges with Pattern)
Byte buffer
Constructor (Pattern, ranges) Data access functions

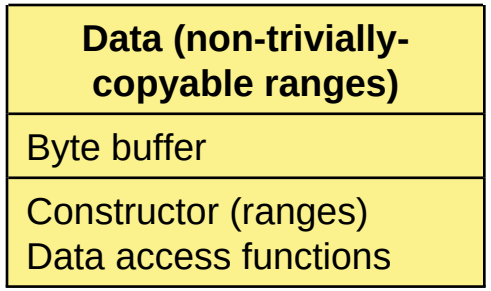
Used for one or multiple ranges of elements must be serialized using a „Datatype“.

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n; };

std::vector<MyClass> vec(4);

constexpr mppi::Pattern<MyClass, "x", "y", "n"> pattern;
if (mppi_comm.get_rank() == 0)
        mppi_comm.send(mppi::Destination(1), mppi::Tag(0), vec | mppi::pattern_view(pattern));
else
        mppi_comm.recv(mppi::Source(0), mppi::Tag(0), vec | mppi::pattern_view(pattern));
```

Implementation Details



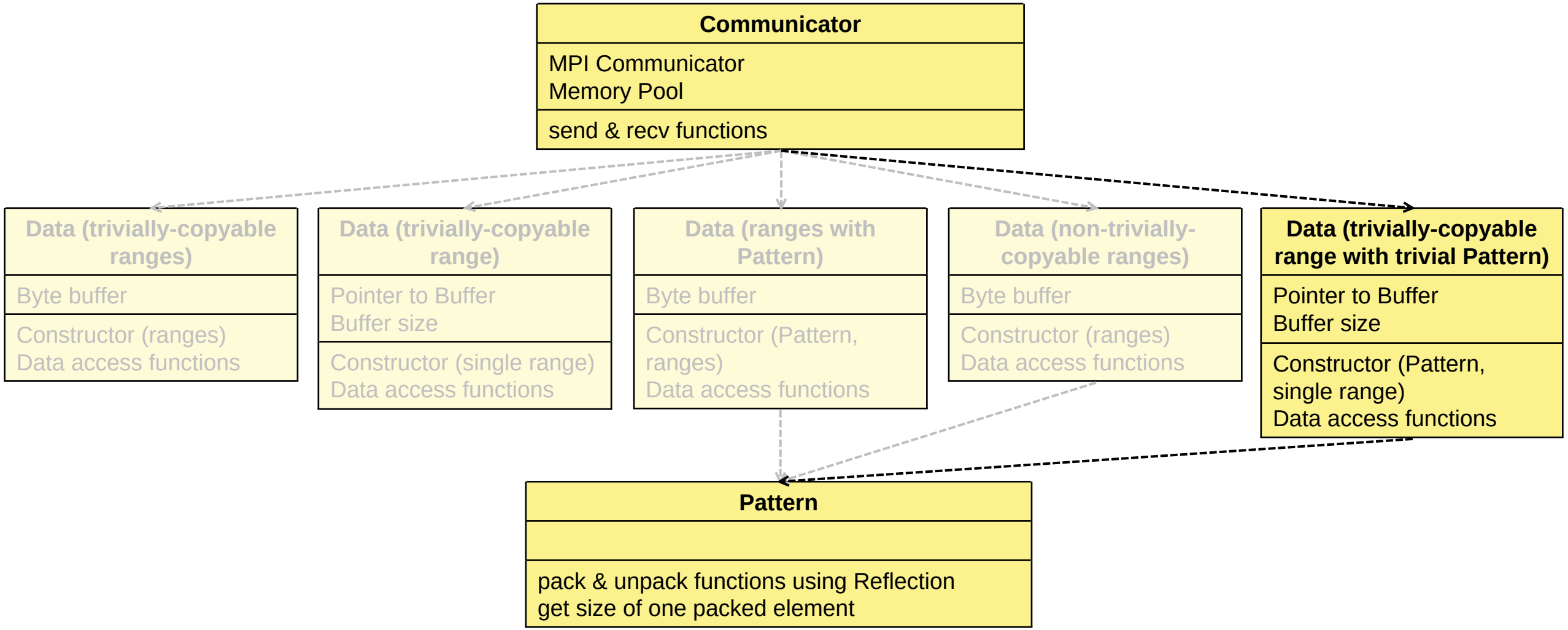
Implementation Details



Used for one or multiple ranges of non-trivially-copyable elements. For these an internal Pattern is created.

```
class MyClass {
        virtual ~MyClass() = default;
        std::array<double, 2> x;
        double y;
        char a;
        int n; };

std::vector<MyClass> vec;

if (mppi_comm.get_rank() == 0)
        mppi_comm.send(mppi::Destination(1), mppi::Tag(0), vec);
else
        mppi_comm.recv(mppi::Source(0), mppi::Tag(0), vec);
```

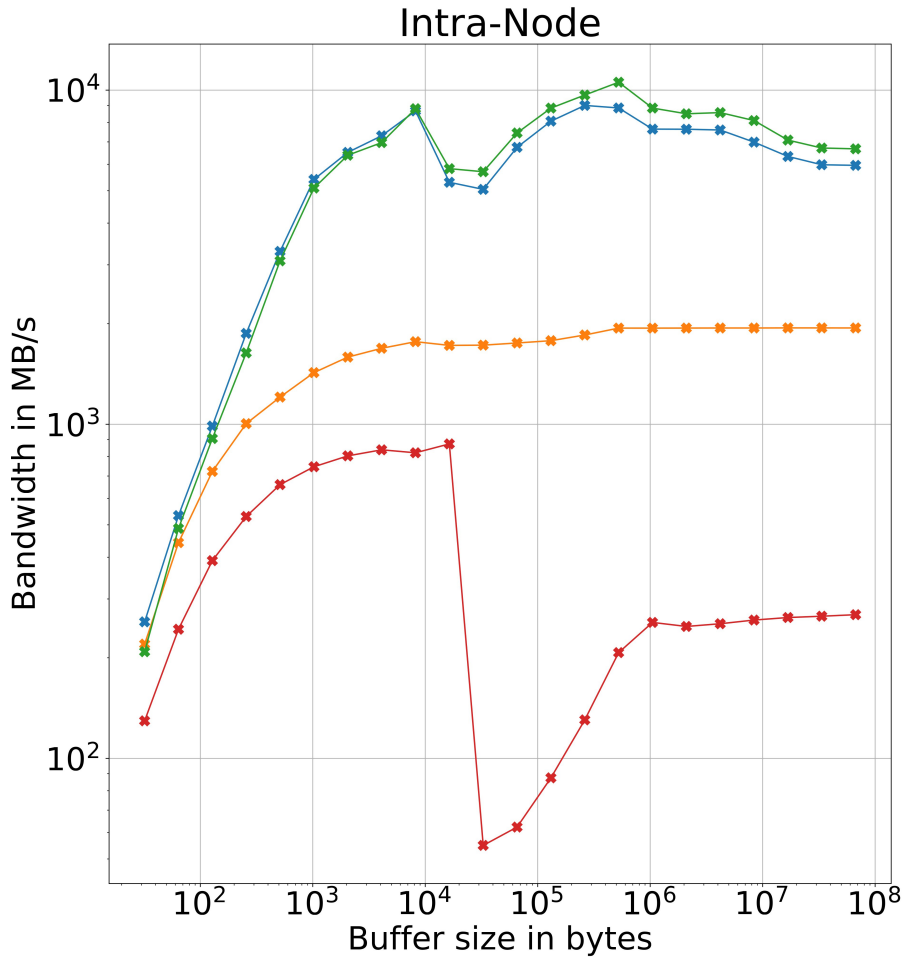
Implementation Details

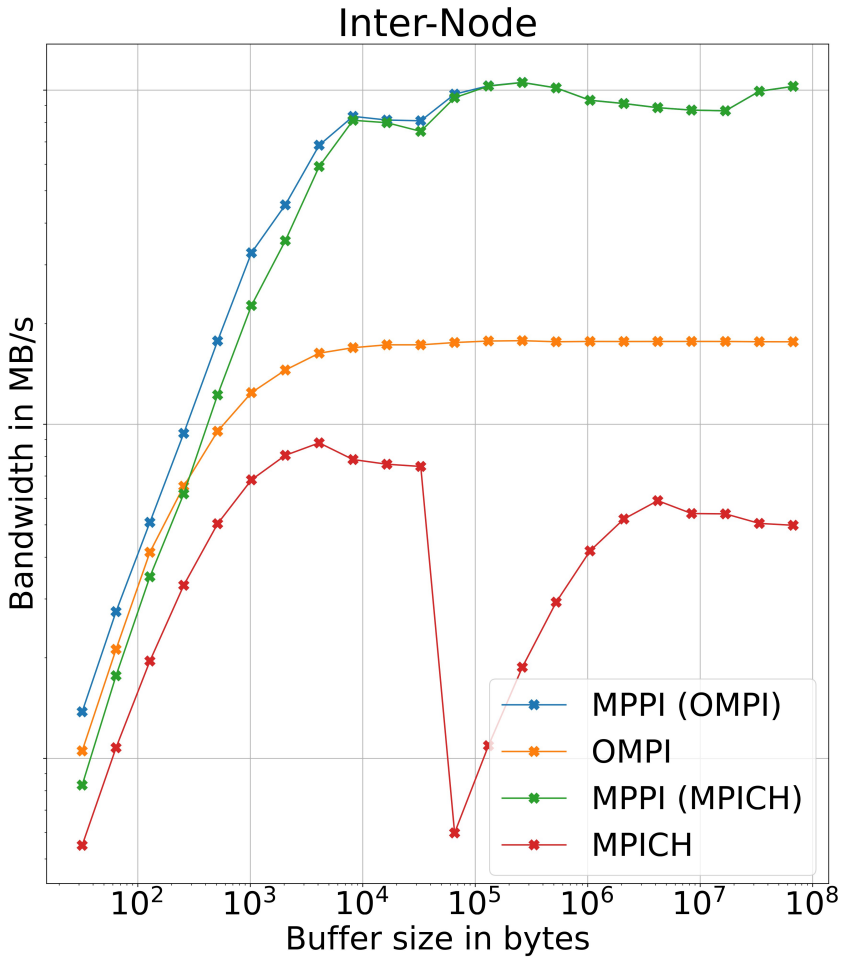


Evaluation – OSU-Inspired Benchmarks

- Bandwidth benchmark.
- Used Open MPI and MPICH.
- Used type:

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n;
};
```



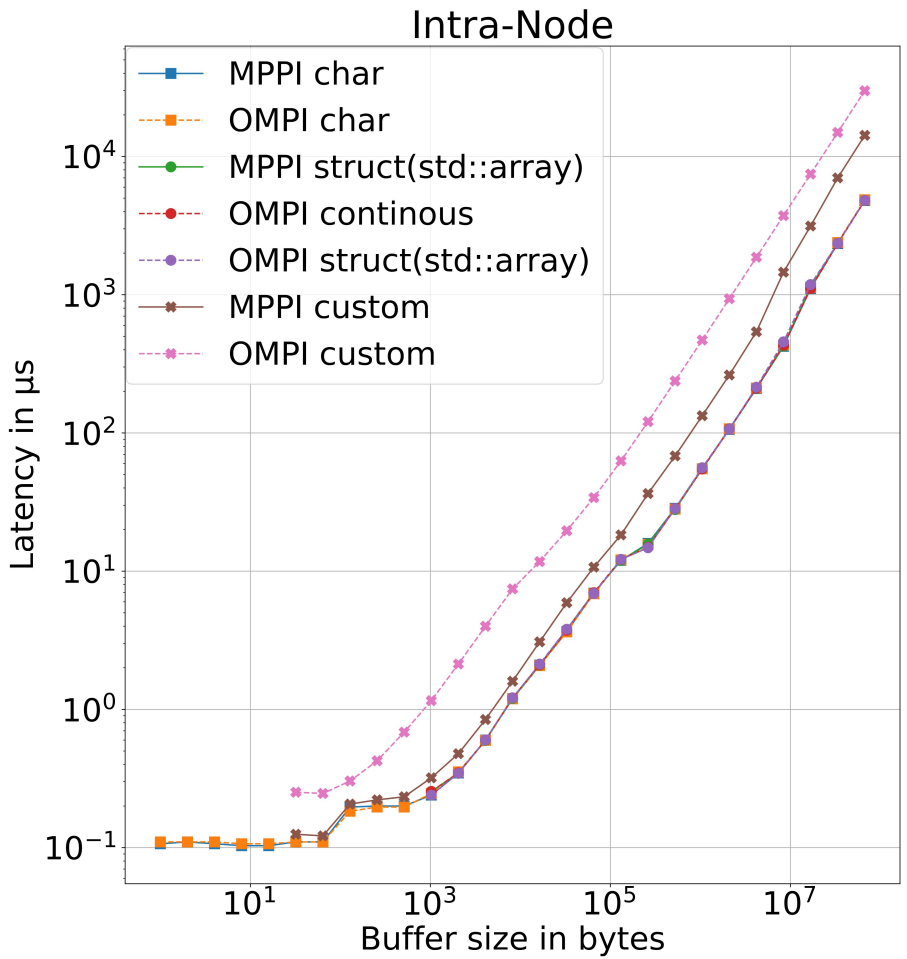


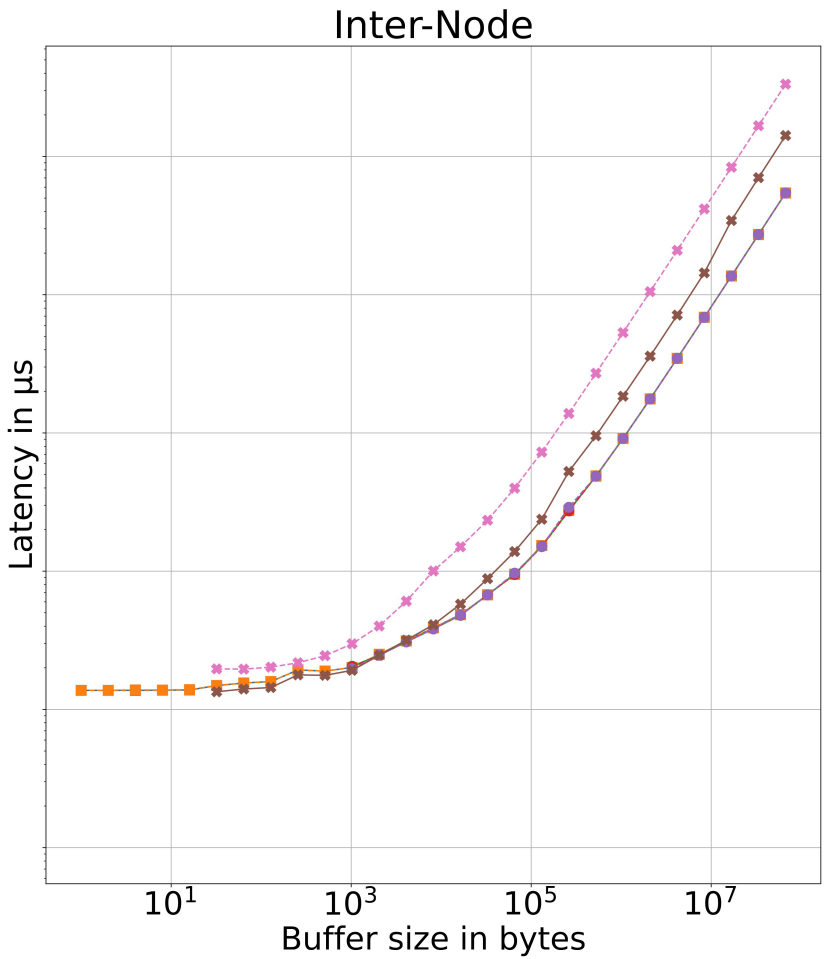
Evaluation – OSU-Inspired Benchmarks

- Latency benchmark.
- Used Open MPI.

- Used types:

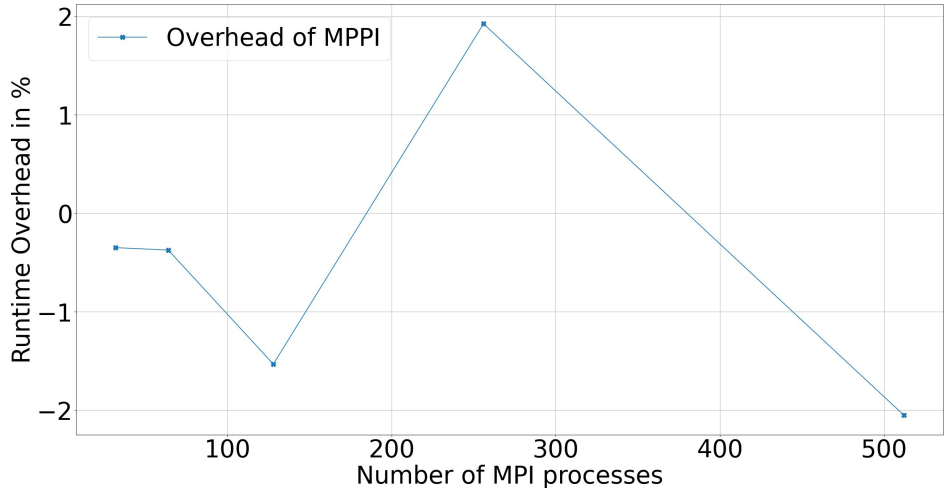
Type	Size
Char	1
Continous	1024
MPI struct(std::array)	1024
MPPI struct(std::array)	1024
Custom MyClass	32





Application example: ls1-MarDyn

- Communication buffer functionality with MPPI.
- Maintainability and readability increased.
- Overhead is negligible.




```
constexpr mppi::Pattern<Molecule, "_id", "_cid", "_r", "_v", "_q", "_L"> leaving_pattern;
constexpr mppi::Pattern<Molecule, "_id", "_cid", "_r", "_q"> halo_pattern;

std::vector<Molecule> leaving_molecules;
std::vector<Molecule> halo_molecules;
...

mppi_comm.send(mppi::Destination(dest), mppi::Tag(99),
                          leaving_molecules | mppi::pattern_view(leaving_pattern),
                          halo_molecules | mppi::pattern_view(halo_pattern));
```

Outlook

- Address collective communication.
- General multi-threading support.
- Support for utilization of GPUs or accelerators in general. For example to offload serialization.
- Optimization for overlapping communicating and serializing data.

Thank you for your attention!

https://github.com/mikesoehner/MPPI/

Implementation Details

Data (trivially-copyable range)
Pointer to Buffer Buffer size
Constructor (single range) Data access functions

Optimization for a single, consecutive range of trivially-copyable elements.

```
std::vector<int> vec(10);
mppi::Communicator mppi_comm;

if (mppi_comm.get_rank() == 0)
        mppi_comm.send(mppi::Destination(1), mppi::Tag(0), vec);
else
        mppi_comm.recv(mppi::Source(0), mppi::Tag(0), vec);
```

Implementation Details

Data (trivially-copyable range with trivial Pattern)
Pointer to Buffer Buffer size
Constructor (Pattern, single range) Data access functions

Optimization used for a single range of trivially-copyable elements given with a trivially-copyable pattern.

```
class MyClass {
        std::array<double, 2> x;
        double y;
        char a;
        int n; };
constexpr mppi::Pattern<MyClass, "x", "y", "a", "n"> pattern;
std::vector<MyClass> vec(4);
mppi::Communicator mppi_comm;
if (mppi_comm.get_rank() == 0)
        mppi_comm.send(mppi::Destination(1), mppi::Tag(0), pattern, vec);
else
        mppi_comm.recv(mppi::Source(0), mppi::Tag(0), pattern, vec);
```