Improving MPI Language Support Through Custom Datatype Serialization

Jake Tronge
Los Alamos National Laboratory
Los Alamos, New Mexico, USA
jtronge@lanl.gov

Joseph Schuchart
Institute for Advanced Computational Science
Stonybrook, New York, USA
joseph.schuchart@stonybrook.edu

Lisandro Dalcin
King Abdullah University
of Science and Technology
Thuwal, Saudi Arabia
dalcinl@gmail.com

Howard Pritchard
Los Alamos National Laboratory
Los Alamos, New Mexico, USA
howardp@lanl.gov

***Abstract*—Exascale applications are being increasingly written in modern languages such as Python, Julia, C++, and Rust. The Message-Passing Interface (MPI), the de facto standard for parallel computing, only defines interfaces for C and Fortran, languages that are very different from these modern languages, often containing more complex types and representations incompatible with MPI. The existing derived datatype interface is widely used for older applications, but fails to work efficiently for types containing multiple pointers, requiring application-specific initialization, or serialization. Applications written in these languages can still use MPI, but at the cost of complicated address manipulation or high overhead. This work proposes a new datatype interface for MPI giving more control to the application over buffer packing and the wire representation. We built a prototype for this interface, demonstrating it with Rust, Python, and C++, highlighting key concerns of each language and showing the improvements provided.**

***Index Terms*—MPI, distributed datatypes, Data serialization, C++, Rust, Python, Julia**

I. Introduction

Applications written using modern languages, such as C++, Julia, Python, and Rust, typically make extensive use of data structures with varying characteristics and representations. MPI derived datatypes were designed to support data structures in C and Fortran, representing compound types with a sequence of predefined types and offsets from a base pointer; this representation works extremely well for existing C and Fortran code but fails to support many types in modern languages, especially those that require serialization, which is a very common feature [1], [2], [3], [4]. Serialization, or marshalling, can be implemented by manual packing, as is done in most implementations, but this can put an unnecessary burden on memory usage, especially for larger buffers, resulting in inefficient cache use as well as ignoring other possibilities for performance improvements. Some serialization libraries are able to extract regions of memory that can be directly sent and received, allowing for some level of zero-copy serialization. Current MPI datatypes, however, cannot take advantage of performance enhancements provided by these libraries without some loss in performance. We believe that creating new MPI datatypes designed to take advantage of modern serialization techniques can lead to enhanced performance for modern languages.

Current MPI derived datatypes are designed around the use of a type map, consisting of a sequence of predefined types and displacements [5], using various type creation functions, to enable construction of types using common patterns, such as struct-like types, homogeneous arrays, as well as types with varying offsets between elements. While this representation suffices for many of the constructs typically used in C and Fortran, it fails to support dynamic types and types that require serialization. Derived datatypes are constrained by the use of a single pointer and offsets from that for the entire type. Dynamic types may include pointers to memory regions on the heap of varying lengths, and for serialized types, a dynamic buffer holding the serialized representation of the type may need to be constructed on the fly. Existing derived datatypes could potentially be used to represent some of these types, but would require error-prone address manipulation and expensive datatype recreation for every unique buffer.

The alternative to using derived datatypes is to pack serialization buffers before sending as a primitive type, such as `MPI_BYTE`, which is the case for many existing MPI bindings [6], [7]. Performance problems arise with this method for extremely large messages and for large numbers of small messages. To serialize a large buffer, implementations are currently required to allocate an additional buffer that is often the same size, or larger than, the datatype buffer as stored in memory, doubling memory usage. Some implementations of MPI bindings [7] attempt to take advantage of large, contiguous memory regions extracted from serialized types by using multiple messages, at the cost of thread safety concerns and high latency overhead. Multiple high-level messages may use the same tag, thus requiring higher-level locking mechanisms to ensure messages can be sent across threads. Dynamic data structures present problems as well, particularly on the receive side, where incoming serialized buffer lengths cannot be known in advance; this forces the receive side to do some sort of probe or receive an extra message to determine the length and allocate buffers for the serialized

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00062

```
struct A {
  int a, b, c;
  // 4B gap
  double d;
  A(int a, int b, int c, double d)
  : a(a), b(b), c(c), d(d)
  { }
  virtual ~A() = default;
  // size of data members to pack
  constexpr static
  size_t pack_size = sizeof(int)*3
                   + sizeof(double);
};
```

Listing 1: Non-POD C++ type with 4 B gap and virtual function table.

message. These probes and extra messages can lead to serious performance degradation, especially when receiving multiple small messages. Integrating a custom datatype API into MPI with full support for these types would alleviate the correctness problems and address potential performance problems by allowing for improved network utilization.

Our major contributions in this paper are as follows:

- Design of a custom datatype API to support more complex types used in languages such as C++, Python, Julia, and Rust.
- Prototype implementation of the new API.
- Evaluation of the API and prototype using Python, C++, and Rust with various datatypes.

This paper is organized as follows. Section II gives important context about datatypes and language-related issues. Section III explains our proposed API. Section IV presents a prototype implementation and section V contains our evaluation. Section VI discusses the API, the implementation, and limitations. We then list related works in section VII and close in section VIII.

II. Background

In this section we will give a more detailed explanation about datatype representations, key concerns for modern programming languages, and the complications these present for MPI.

A. Non-POD datatypes

Listing 1 shows a datatype that is not a *plain old datatype* (POD) in C++. The introduction of a virtual member function inserts a virtual function table into the binary representation of the object. These *vtables* contain function pointers that are called at runtime and should not be copied to other processes, where the memory layout may be different. Tools such as `offsetof` are not supported for non-POD types and thus it is not possible to query the beginning of the public data of a non-POD type. The only safe way to send or receive such a type is through explicit serialization.

B. Dynamic Types

Not all objects in modern languages can be expressed as fixed-size types with a known number of memory regions. For example, in a list of vectors (`std::list<std::vector<int>>` in C++) each vector is a contiguous memory region that can be transferred by MPI individually. However, a list itself is a non-contiguous container. We thus need to first determine the number of memory regions in the list before collecting the sizes of the individual vectors and their base pointers. The serialization of such a list consists of storing the size of each vector. The deserialization of such a list consists of resizing each vector to be able to hold the data that will be written by MPI.

C. Serialization in Python and Other Languages

Python and, in particular, its serialization library Pickle [2] present several challenges for MPI datatypes. Pickle was originally designed as a format for storing Python objects on disk, but has grown beyond this in recent years to be used for RPC calls, network programming and more. In its simplest form a Python object can be fully serialized into one large buffer and then sent over some medium or simply stored on disk. Full serialization becomes a problem for large objects since it can potentially double memory usage of a program, just to store an intermediate serialized form. This is a problem not just in Python, but in other languages that provide similar serialization mechanisms.

To remedy the memory usage concerns of full serialization, extensions to Pickle [8] were designed to allow for out-of-band or zero-copy buffers. Instead of packing all of the data into one buffer, larger data fragments are returned in an array of zero-copy buffers pointing to memory regions that can be sent directly without requiring a new allocation. These out-of-band buffers can be used by `mpi4py` by sending multiple MPI messages.

Another important issue with Pickle and other serialization frameworks in general is that the receive side often cannot determine the expected serialized size of an object, or for that matter, the multiple sizes that would be involved with zero-copy buffers. mpi4py currently uses **MPI_Mprobe** on the receive side to determine the serialized size. For an array of zero-copy buffers a separate message with the buffer lengths is required [7].

D. Rust and RSMPI

We use the Rust programming language to implement our prototype. Rust is a systems-level programming languages designed with memory safety, performance, and modern language ergonomics in mind. RSMPI [9] is the current Rust library, or *crate*, exposing most MPI functions to Rust code by linking with major MPI implementations. RSMPI supports derived datatypes through procedural macros, which can be invoked at compile time on struct type definitions to automatically generate the MPI type creation calls. On first use of the type in a call, the derived datatype will be created and cached for later usage. This mechanism is used for a couple of our

```c
int MPI_Type_create_custom(
    MPI_Type_custom_state_function *statefn,
    MPI_Type_custom_state_free_function *freefn,
    MPI_Type_custom_query_function *queryfn,
    MPI_Type_custom_pack_function *packfn,
    MPI_Type_custom_unpack_function *unpackfn,
    MPI_Type_custom_region_count_function *
        region_countfn,
    MPI_Type_custom_region_function *regionfn,
    void *context,
    // Flag indicating in-order pack requirement
    int inorder,
    MPI_Datatype *type
);
```

Listing 2: Signature for the datatype create function of the custom API. This includes a pointer to a context that can be used when initializing pack or unpack operations.

benchmarks to show the performance of an underlying MPI implementation with derived datatypes.

III. Custom Serialization API

Our proposed custom serialization API gives applications the ability to directly pack or extract memory regions from message buffers. Using this API, applications can operate at a slightly lower-level than the existing MPI datatype API and can represent more complex types commonly found in modern languages. The custom serialization API can be used to create MPI datatypes with application-provided functions to pack non-contiguous data and the ability to expose contiguous memory regions to MPI. Many complex objects often include parts that must be packed, such as individual fields of non-contiguous struct data, and other fields, such as vectors and long contiguous arrays of primitive types, that can be sent efficiently as memory regions. Errors are propagated through return values: each callback returns either `MPI_SUCCESS` or an error value indicating a failure. Error handling is crucial for serialization libraries that can fail in the case of invalid data.

These datatypes are created with a new type creation function and a list of application callbacks, whose prototype is shown in Listing 2. Instead of fixed offsets and counts, MPI implementations must query the details of objects to be sent through the provided callbacks. Generally, use of this API can be broken up into two main stages: i) packing of data using the pack callback; and ii) querying of memory regions for components of the buffer that can be sent directly.

When a buffer with the custom datatype is accessed by some MPI operation, a state object local to that operation can be allocated using the `MPI_Type_custom_state_function` callback as shown in Listing 3. The state object is used to store buffer-specific information for any sequence of callbacks that may be invoked during sending or receiving of a custom type buffer. The state object is freed on completion of the point-to-point operation using the `freefn` callback. The state object may contain context information for complex types and is optional, i.e., may be ignored for simpler types.

The callbacks for packing and unpacking, shown in Listing 4, are inspired by the generic datatypes implemented

```c
typedef int (MPI_Type_custom_state_function)(
    // Context passed to create function
    void *context,
    // Buffer provided to MPI
    const void *src,
    // Count provided to MPI
    MPI_Count src_count,
    // Out: State to be passed into callbacks
    void **state
);
typedef int (MPI_Type_custom_state_free_function)
    (void *state);
```

Listing 3: State management functions. The optional state is used during pack and unpack operations to keep track of custom serialization data between calls.

```c
typedef int (MPI_Type_custom_query_function)(
    // State information
    void *state,
    // User-provided buffer (not packed)
    const void *buf,
    // Count passed to MPI
    MPI_Count count,
    // Expected bytes to be packed
    MPI_Count *packed_size
);
typedef int (MPI_Type_custom_pack_function)(
    // State information for packing
    void *state,
    // Pointer to custom object to be packed
    const void *buf,
    // Number of elements of custom type
    MPI_Count count,
    // Virtual offset into the packed buffer
    MPI_Count offset,
    // Destination buffer
    void *dst,
    // Size of destination buffer
    MPI_Count dst_size,
    // Out: Number of bytes used
    MPI_Count *used
);
typedef int (MPI_Type_custom_unpack_function)(
    // State information for unpacking
    void *state,
    // Pointer to object to unpack data into
    void *buf,
    // Number of objects to unpack
    MPI_Count count,
    // Virtual offset into the unpacked buffer
    MPI_Count offset,
    // Incoming buffer to be unpacked
    const void *src,
    // Size of current buffer to be unpacked
    MPI_Count src_size
);
```

Listing 4: Callback functions used in the custom serialization API for querying the total packed size, packing of data, and unpacking of data.

```
typedef int (
    MPI_Type_custom_region_count_function)(
    void *state,
    // Buffer pointer
    void *buf,
    // Number of elements in send buffer
    MPI_Count count,
    // Out: Number of memory regions
    MPI_Count *region_count
);
typedef int (MPI_Type_custom_region_function)(
    void *state,
    // Buffer pointer
    void *buf,
    // Number of elements in send buffer
    MPI_Count count,
    // Number of regions
    MPI_Count region_count,
    // Out: start of each region
    void *reg_bases[],
    // Out: length of each region
    MPI_Count reg_lens[],
    // Out: MPI types for each region
    MPI_Datatype reg_types[]
);
```

Listing 5: Memory region/iovec callback signatures.

within the Unified Communication Protocol (UCP) component of UCX [10], an optimized and widely used HPC networking library. The `MPI_Type_custom_query_function` callback is used to determine the total packed size of a buffer. The packed size can be used by the implementation to determine how best to break the packed buffer up into fragments, as well as make other optimization decisions. Packing is done fragment by fragment using the `MPI_Type_custom_pack_function` callback. This function takes a state object, an offset, information about a destination fragment buffer, and a used output count. The offset, as in the UCP API, represents a virtual offset in the entire packed buffer in bytes. The pack function is not required to fill the entire fragment buffer, since in many cases this may not line up with the size of a packed element of the datatype. The pack function may choose to only partially fill the buffer, postponing packing of subsequent data to the next call with a new fragment. On the receive side the `MPI_Type_custom_unpack_function` callback is invoked for each individual fragment buffer in order to reconstruct the buffer on the send side.

One complication of giving lower-level pack/unpack control to application code is that data fragments may be received out of order, depending on the underlying transport library. This, however, may be problematic for some applications relying on the same order of fragments. Such applications can set the `inorder` flag to true (see Listing 2). Our prototype implementation always provides in-order packing. This flag would inhibit potential out-of-order optimizations in advanced implementations.

After all data been packed, when the `packed_size` limit is reached, the memory region API is invoked to determine if any parts of the buffer can be sent directly without packing. Listing 5 shows the two callbacks used for memory region extraction. First, the `MPI_Type_custom_region_count_function` is used to determine the number of memory regions that can be sent for this buffer; then the `MPI_Type_custom_region_function` is passed multiple arrays, all of `region_count` length, that will be filled with corresponding memory region pointers, counts, and datatypes.

IV. Implementation

We implement our custom serialization API using a lightweight Rust-based prototype providing a simplified MPI API, reusing components from an earlier MPI prototype [11]. Internally, our prototype uses UCX [10], a framework that provides various levels of abstraction for HPC networks, protocols, and hardware. In particular, we utilized the UC-Protocols (UCP) component, which is designed to support MPI and PGAS implementations, as well as multiple datatypes that helped with our implementation here, including `UCP_DATATYPE_IOV`, `UCP_DATATYPE_GENERIC`, `UCP_DATATYPE_CONTIG`, and others. The prototype will attempt to use `UCP_DATATYPE_CONTIG` when a single contiguous buffer can be used for a message, while using `UCP_DATATYPE_IOV` for when multiple memory regions must be sent.

Our implementation is organized into three main libraries (in Rust referred to as *crates*): the highest layer exposes a simplified MPI API (**mpicd-capi**), the middle layer implements point-to-point management and other necessary logic (**mpicd**), and the bottom layer exposes the UCX API to Rust code (**mpicd-ucx-sys**).

The mpicd-capi layer passes user-provided buffers and datatypes to the mpicd layer using a Rust trait (not shown here due to space) that provides an implementation calling directly into the user-provided function callbacks. Rust traits can roughly be thought of as an interface with a set of methods to implement for individual types, with some unique differences that help with Rust generics and type definitions. When sending or receiving a message with a custom type the library internally uses the `UCP_DATATYPE_IOV` type, allowing for scatter-gather functionality. The packed data is the first element in the scatter-gather pointer (or iovec) list, following which the iovec array is filled with any memory region pointers that were provided by the corresponding callbacks.

V. Evaluation

We evaluate our benchmarks using both bandwidth and latency tests, some of which are partially based on the OSU Micro Benchmarks [12]. We present tests in Python, C++, and Rust. The results presented in this section were obtained using two Dell PowerEdge R7525 servers each with 2 AMD EPYC 7232P 8-Core sockets connected by Mellanox ConnectX5 Infiniband network interfaces with the ports configured for 100 Gbps. The servers were running RedHat 8.4. Compilers used were rustc 1.77.1 and gcc 8.5.0, along with UCX version 1.12.0 and Open MPI v5.1.0a1 (main).

```rust
#[repr(C)]
pub struct StructVec {
    a: i32,
    b: i32,
    c: i32,
    d: f64,
    data: [i32; 2048],
}
```

Listing 6: *struct-vec* type with some elements that will be packed (`a`, `b`, `c`, `d`) and a buffer (`data`) that can be sent with an iovec. Rust types `i32` and `f64` represent 32-bit integers and 64-bit floating point types respectively. Using `#[repr(C)]` forces this struct to use a C representation, causing a gap to be formed between `c` and `d`.

```rust
#[repr(C)]
pub struct StructSimple {
    a: i32,
    b: i32,
    c: i32,
    d: f64,
}
```

Listing 7: *struct-simple* type containing only elements that will be packed (`a`, `b`, `c`, `d`). As in Listing 6, this type also contains a gap between `c` and `d`.

A. Rust

For our Rust evaluation we focus on bandwidth and latency results for different datatypes and datatype representations/transfer methods. These benchmarks tie directly into the **mpicd** prototype implementation, utilizing special traits directly to implement the various datatypes (not shown due to space). The two main methods that will be used in all results shown here are: (1) **custom** where our custom packing functions and memory region-based code is used; and (2) **manual-pack** where code manually packs data into a buffer before sending as a byte stream. All tests show the average of four runs, with error bars included in graphs.

Our Rust evaluation is based on three types that are designed to stress different components of our custom interface, showing where some methods may perform better or worse than others:

- The *double-vector* type is expressed in Rust as `Vec<Vec<i32>>`, being roughly equivalent to the C++ type `vector<vector<int>>`.
- *struct-vector* is a struct type, defined as in Listing 6, including several scalar fields that will likely need to be packed for best performance, while also including an array field best sent directly with a memory region or iovec representation. We use a static array type in the struct-vector type in order for it to work with existing derived datatypes for a useful baseline metric. In practice, however, this array type will likely be a dynamic vector which cannot be represented by derived datatypes without complicated address manipulation and recreation of the datatype on each call. A dynamic vector will only work well with our new custom interface.
- *struct-simple* is another struct type, shown in Listing 7, and is exactly the same as struct-vector except that the vector has been removed; this type is designed to show the performance of packing alone.
- *struct-simple-no-gap*, shown in Listing 8 is similar to struct-simple, except that the type has no gap and thus should require no packing.

```rust
#[repr(C)]
pub struct StructSimpleNoGap {
    a: i32,
    b: i32,
    c: f64,
}
```

Listing 8: The *struct-simple-no-gap* type is similar to the types in Listing 7 and Listing 6, but without the third integer field, thus removing the gap in the C representation.

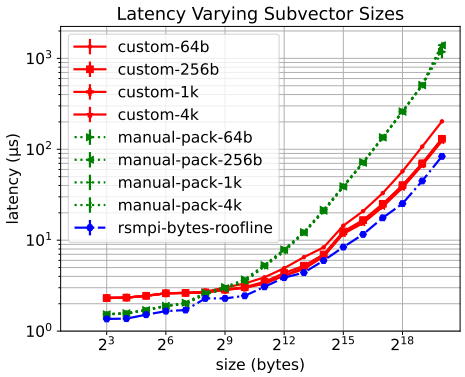

Fig. 1: Latency benchmark for the double vector type run while varying the subvector size on two nodes.

All of the Rust tests shown here will use some variant of the below methods for sending and receiving data:

- **custom** uses the custom packing API, utilizing packing and/or memory regions to send and receive data, depending on the type.
- **packed** manually packs the data into an allocated buffer before sending.
- **rsmpi** using RSMPI [9] linked with Open MPI v5.1.0a1 to act as a baseline; this method has partial support for the struct-simple, struct-simple-no-gap, and struct-vec types but doesn't support the double-vec type, where instead we show this test just sending bytes as an absolute baseline.

custom and **packed** both use our simple prototype MPI implementation.

Latency and bandwidth results for the double-vector type are shown in Figure 1 and Figure 2, respectively. The double-vector tests were performed with vectors of vectors of integers,

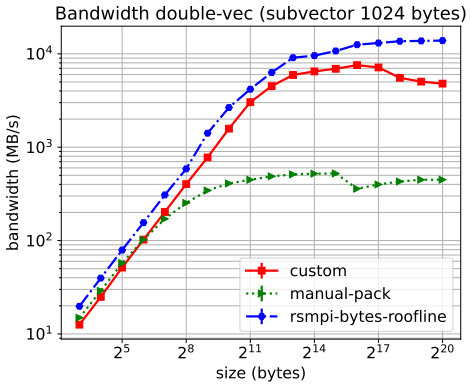

Fig. 2: Bandwidth benchmark for the double vector type run on two nodes. The subvector size was set to 1024 bytes.

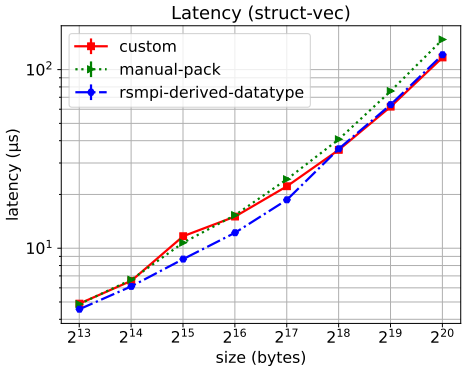

Fig. 3: Latency benchmark for the struct-vector type.

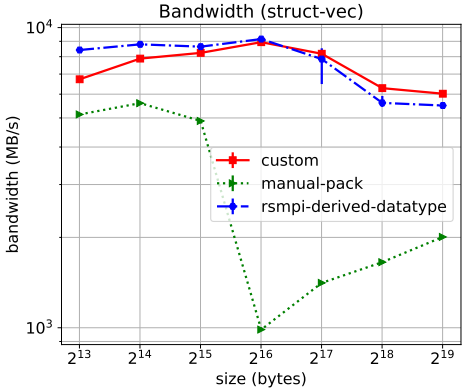

Fig. 4: Bandwidth benchmark for the struct-vector type.

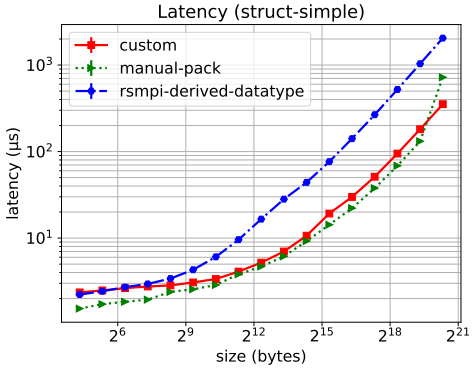

Fig. 5: Latency benchmark for the struct-simple type.

where each internal integer vector has uniform length, which we refer to here as the *sub-vector length*; for message sizes smaller than the sub-vector size, a single sub-vector of the message size is sent.

Figure 1 shows latency results while varying subvector sizes from 64 bytes to 4 KiB. At about a size of 2^9 bytes, the custom method begins to show better performance for larger subvector sizes (1-4 KiB) when compared with smaller sizes (64-256 bytes). The manual-pack tests after 2^9 bytes have the highest latency of all methods shown; these tests also show no negligible difference between the different subvector sizes.

As expected, the rsmpi-bytes-baseline has the lowest latency. The bandwidth results for the double-vector type were run with sub-vector sizes of 1024 bytes. Our custom method outperforms manual packing at larger data sizes due to the use of memory regions, since the double-vector type lends itself well to these types of transfers.

Figures 3 and 4 show results for the struct-vec benchmark. Since this type uses an array in the definition, we are able to use derived datatypes with rsmpi; if this array were instead a vector, then RSMPI and MPI in general would not support this type, whereas custom and manual-pack would be able to handle this type with the performance shown here, since we treat the type as if it contained a vector.

Figure 3 shows latency results for the struct-vector type. Latency is higher for custom until a size of 2^{18} bytes where both custom and rsmpi-derived-datatype show similar performance. Figure 4 shows bandwidth results for the struct-vector type. The sizes used here are evenly divisible by roughly 8K, which is the size of a packed single struct-vector element. Thus for size 32K there are four elements, for 64K eight elements, and so on.

Figures 5 and 7 show results for the struct-simple type. In Figure 5 we see that custom and manual-pack both have very low-latency in comparison with RSMPI. This is caused by the gap inside the structure, which the Open MPI type representation is not able to handle efficiently. In Figure 6 we ran the same test but with struct-simple-no-gap, showing that RSMPI, and therefore Open MPI, performs as expected when sending contiguous types. Figure 7 shows a similar trend where both custom and manual-pack achieve better performance at larger sizes. The dip shown with manual-pack at 2^{15} bytes can be attributed to the switchover from eager to rendezvous protocol within UCX, which doesn't affect custom since it uses the UCX iovec API internally.

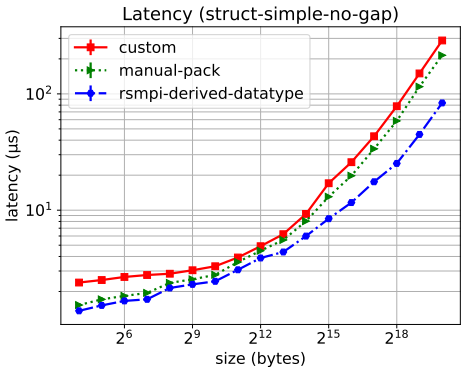

Fig. 6: Latency benchmark for the struct-simple-no-gap type.

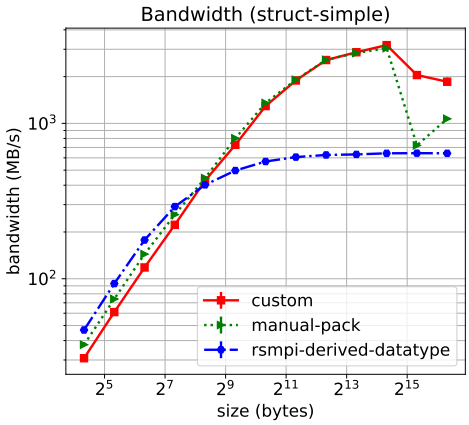

Fig. 7: Bandwidth benchmark for the struct-simple type.

B. Python

For Python we focus on a pingpong test measuring bandwidth using `mpi4py`. This involves measuring the bandwidth of two messages (a send-receive pair) multiple times and then averaging the results. We first determine a roofline bandwidth using preallocated raw memory buffers without any serialization process involved. Next, we determine effective bandwidth using Python's pickle serialization using three different strategies: i) basic pickle, where Python objects are serialized into/out of a single in-band contiguous byte stream communicated with a single pair of MPI messages, and ii) out-of-band pickle, where Python objects are serialized into an in-band contiguous byte stream and a sequence of zero-copy out-of-band memory buffers. The out-of-band serialization is communicated via two mechanisms: a) multiple pairs of MPI messages, and b) the custom datatype machinery proposed in this work, leading to a single pair of outer MPI messages with the MPI engine handling internally the pieces.

It should be noted that MPI communication of Python objects with pickle serialization always involve memory allocation on the receive side. For objects containing large memory buffers, these memory allocation steps on the receive side impact negatively on the effective achievable bandwidth. Our pingpong tests involve the communication of NumPy arrays. Pickle serialization of these Python objects always include, besides the contiguous array buffer, an additional small "header" with basic metadata (mostly shape, datatype, and byte order) required to reconstruct the object upon deserialization. For simple 1D arrays, this metadata header weighs around 120 bytes, thus being quite small in comparison to the array buffer sizes we use in our testing.

To make our testing representative of different scenarios, we consider two cases: 1) the communication of single NumPy arrays of a given size, and 2) the communication of complex user-defined Python object containing multiple 128-KiB NumPy arrays and adding up to a given total size; the effective bandwidth is shown on Figure 8 and Figure 9, respectively. When using basic in-band pickle, the small metadata headers and the large array buffers are all packed together in a single contiguous byte stream and then communicated with a single MPI message (lines with label *pickle-basic*). When using out-of-band pickle, only the small metadata headers are packed in-band in a single contiguous buffer; the large array buffers are handled out-of-band and have to be dealt with individually, i.e., either communicated with individual MPI messages (lines with label *pickle-oob*), or handled by our custom datatype engine as individual memory regions (lines labeled *pickle-oob-cdt*).

The trends seen in the figures show that for smaller aggregate message sizes, the basic pickle pack method yields similar performance to that for the two out-of-band methods. For the single NumPy array case, the performance of the two out-of-band methods is significantly better than the simple pickle method for message sizes 2^{18} bytes (256 KiB) and greater. For the complex object case containing multiple NumPy arrays, results are mixed for intermediate message sizes, but for the largest transfer sizes, the two out-of-band methods realize significantly better bandwidths than the basic pickle pack method. The out-of-band approaches cannot match the raw roofline performance, but this is expected; as mentioned previously, the required memory allocations on the receive side diminish the effective achievable bandwidth.

C. Datatype Benchmark

We implemented a subset of the benchmarks from the DDT-Bench benchmark suite [13], a collection of MPI datatypes representative of widely used MPI applications. Each benchmark performs ping-pong communication using various strategies: manual packing using C code, packing using MPI datatypes, and direct communication using MPI datatypes. For custom datatypes, we experimented with using both packing and memory regions (where sensible) and provide both data for comparison.

The chosen datatypes exhibit significantly different structures. For the sake of brevity, Table I provides a summary of the benchmark characteristics. For example, the manual pack function for LAMMPS consists of a single loop packing from six arrays while the manual pack functions in MILC and WRF

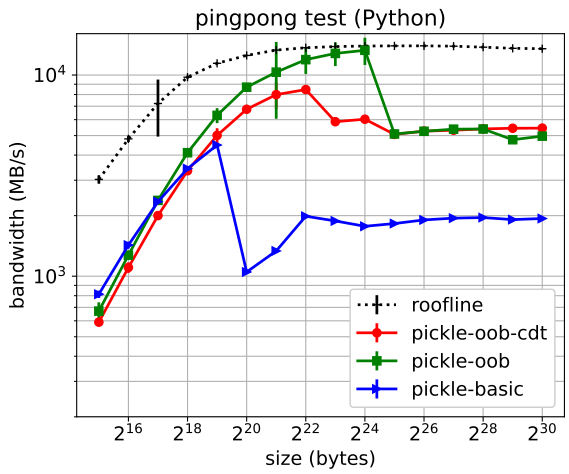

Fig. 8: Python pingpong test using single NumPy arrays.

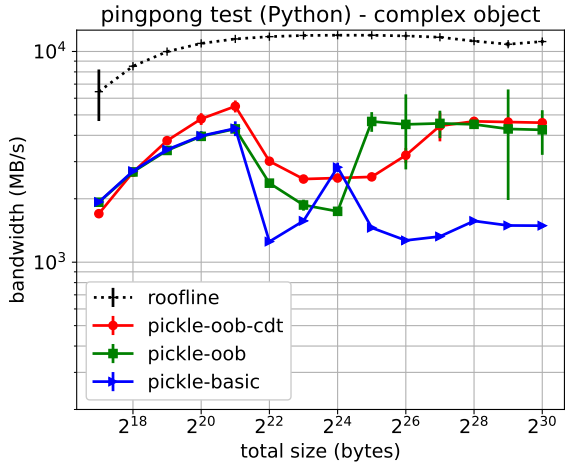

Fig. 9: Python pingpong test using a complex object composed of multiple NumPy arrays of 128 KiB buffers each that sum to the total size shown on the x-axis.

use loop nests of up to five with non-contiguous, non-unit stride iteration. For LAMMPS and WRF, the non-unit stride loop and non-contiguous loop nests, respectively, made the use of memory regions in the custom datatype impracticable.

We experimented with ways to partially pack data, i.e., the ability to return from the pack function before all data has been packed due to a limited size buffer and return later to pack into a new buffer. This has been straight-forward with the single loop in LAMMPS where the start of the loop can simply be computed from the offset passed to the pack/unpack callbacks. However, for nested loops, this quickly becomes intractable. We thus experimented with using C++ coroutines and specifically with `std::generator`. This interface allows us to suspend the coroutine in the middle of a loop nest and

TABLE I: Benchmark characteristics.

Benchmark	MPI Datatypes	Loop Structure	Memory Regions
LAMMPS	indexed, struct	single loop, 6 arrays (non-unit stride)	
MILC	strided vector	5 nested loops (non-unit stride)	✓
NAS_LU_x	contiguous	2 nested loops	✓
NAS_LU_y	strided vector	2 nested loops (non-contiguous)	✓
NAS_MG_*			✓
WRF_*_vec	struct of strided vectors	3/4/5 nested loops (non-contiguous)	

```
1  std::generator<MPI_Count>
2  pack_coro(PackInfoT *info) {
3    MPI_Count pos = 0, k, m, i, cnt;
4    MPI_Count dst_cnt = info->dst_cnt;
5    double *src = info->src;
6    double *dst = info->dst;
7    for (k = 1; k < DIM3; k++) {
8      for(m = 0; m < DIM1; /* inline */ ) {
9        /* pack as much as possible */
10       cnt = std::min(dst_cnt-pos, DIM1-m);
11       for (; m < cnt; ++m)
12         dst[pos++] = src[idx(m,k)];
13
14       if (pos == dst_cnt) {
15         /* dst full, suspend */
16         co_yield pos*sizeof(double);
17         /* we're back, reset state */
18         src = info->src;
19         dst = info->dst;
20         pos = 0;
21   } } }
22   co_return pos*sizeof(double);
23 }
24
25 int pack_cb(
26   void *state, const void *buf,
27   MPI_Count count, MPI_Count offs,
28   void *dst, MPI_Count dst_size,
29   MPI_Count *used)
30 {
31   PackInfoT *info = (PackInfoT*)state;
32   if (info->coro == nullptr)
33     info->coro = new pack_coro(info);
34   info->dst_cnt = dst_size/sizeof(double);
35   info->src = (double*)(intptr_t)src+offs;
36   info->dst = (double*)dst;
37   *used = *info->coro->begin();
38   return MPI_SUCCESS;
39 }
```

Listing 9: Packing NAS_LU_y using C++ coroutines.

return to the same position at a later time. Listing 9 provides the slightly simplified code of such a coroutine. The important part is the ability to suspend in the middle of the `m`-loop and resume without leaving the loop nest.

Unfortunately, we have seen issues with vectorization of loops in the Clang compiler inside coroutines. For the DDT-Bench experiments we thus resorted to full packing and a pack function that do not utilize coroutines. However, we consider this a deficiency that will eventually be resolved, making custom datatypes in MPI an interesting use-case for

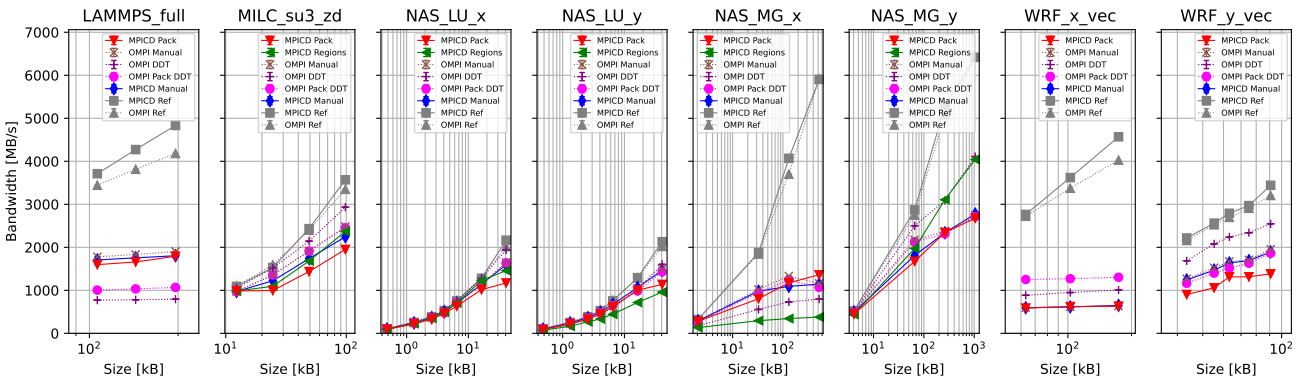

Fig. 10: DDTBench results for Open MPI and custom datatypes.

C++ coroutines.

The results are depicted in Figure 10. We compare the custom pack/unpack and memory regions against sending and receiving using the datatype engine in Open MPI, up-front packing and sending/receiving a contiguous buffer using either MPI datatypes or manual loops, as well as a reference ping-pong of the same size that does not require packing.

Figure 10 shows that the custom packing (red) provides competitive performance over MPI datatypes in some cases (LAMMPS, NAS_MG_x) while in some cases performance lags behind all other methods (MILC, WRF_y_vec). The latter cases require additional investigation.

In some cases, exposing the data as memory regions yields higher bandwidth than packing, i.e., for MILC, NAS_LU_x, and NAS_MG_y, where only a small number of regions is required. Conversely, when larger numbers of (smaller) regions are exposed the use of memory regions with the UCX scatter-gather API yields lower bandwidths (i.e., for NAS_LU_y and NAS_MG_x). We consider this a quality of implementation issue since an MPI implementation could pack the data into a contiguous buffer instead of calling the UCX API.

This evaluation shows that the custom datatype API can deliver competitive performance but we acknowledge that our implementation may require further optimization.

VI. Discussion

Our proposed API provides a new way for communicating objects that are not currently well supported by MPI, including complex types found in higher level languages. This allows for applications to easily represent dynamic types, such as those composed of multiple heap-allocated elements. The API also makes it easier to work with serialization libraries, such as Python's Pickle, which in some cases can provided out-of-band buffers to avoid the memory overhead inherent in serialization. The issues discussed here also go beyond implementing language bindings for newer languages. C, C++, and Fortran data structures containing dynamically sized arrays face similar shortcomings and would benefit from the custom datatype proposal. Today these arrays have to be sent separately, creating potential conflicts between threads sharing the message tag-space. Our proposed API combines these messages parts into a single MPI operation.

Our API does have some limitations to note. The first being that we require the receive side to know the exact length of individual components of a message, such as when sending multiple memory regions. In the Python implementation we work around this problem with a second message containing an array of the lengths of each region. Ideally, there should be some way to better handle this length information, perhaps by extending `MPI_Probe` and `MPI_Get_count`. Doing so would make it easier to avoid using multiple messages for multi-component buffers, which in turn helps eliminates the need for locking in multithreading situations since all data can be encapsulated in a single "atomic" MPI message.

One of the initial design questions we faced with this new API was whether to include existing MPI type information. Derived datatypes describe types by explicitly listing predefined types in type creation calls, the products of which can then be used in further calls to build up more complex types. MPI implementations then have complete control over how these types are transferred and reconstructed from incoming data. Our API gives more control to the application code, both allowing for more complicated datatypes and for the application to make performance decisions that may not be possible with the limited information known from derived datatypes. The problem then becomes whether or not we should integrate into the APIs this existing predefined type information, allowing for MPI implementations to make further decisions internally that could help with MPI reductions and other calls.

Language bindings today often have to work around MPI limitations by breaking language-level messages into multiple real MPI messages, whether for additional metadata or to avoid memory overhead. When multithreading is used, which is very common in modern languages, higher level thread safety controls need to be implemented around the MPI interfaces to ensure that messages being sent from multiple threads are not interleaved. This can involve locking per communicator and per tag, all of which can lead to significant overhead.

In addition, we anticipate some challenges to full stan-

dardization of the datatype API, including further refinement for collective call semantics and packing with accelerator device buffers. The interface may require a method for setting boundaries between minimum chunks of data to be processed by the callbacks, allowing for collective operations to function properly. Furthermore, packing and handling accelerator memory may require device kernels to run, as opposed to our host-based callbacks.

VII. Related Work

Several previous works have pointed out issues with using MPI and higher-level languages. Gregor et al. [14] give an early overview of this with respect to languages such as Java, Python, and others, noting that these languages often produce serialized objects without a fixed size. The *MPI_BLOB* datatype proposed in this paper corresponds to the *regions* concept presented earlier in Section III. Kambadur *et al.* presented an approach to improved C++ bindings for MPI that would support optional serialize/deserialize of transmitted objects [15]. The KaMPIng MPI C++ bindings implementation [16] makes use of the Boost PFR library to generate MPI types for user-provided structs at compile time. Carpenter *el al.* presented strategies for serialization/deserialization of objects within the context of Java MPI implementations [17]. They discussed various strategies for improving on Java's native *Object(Output/Input)Stream* methods to map better to MPI's datatype concept. Peng *et al.* proposed an MPI Stream concept for augmenting MPI's message passing paradigm with one more suitable for handling irregular communication patterns [18]. Other works in this area highlight safety concerns, especially the lack of message type validation [19], [11].

Existing derived datatypes, while supporting a wide range of types, have limitations that can cause problems for newer languages and more complex types. Several authors have investigated the use of compiler techniques to produce optimized serialize/deserialize functions for C and Fortran data structures already expressed in terms of MPI derived datatypes [20], [21], [22]. Träff et al. proposed a library for derived datatypes, noting and adding extensions for missing functionality [23].

There have been a few works focused on extracting information about MPI derived datatypes for use in external libraries or for validation purposes. MPICH recently implemented extensions that are designed to extract memory region information or iovecs from MPI datatypes [24], the opposite of our work, which attempts to create MPI datatypes from memory regions. Kimpe et al. [25] created a method for serializing datatype representations and provide an overview of the complications involved in determining MPI datatype equivalence.

Several works have attempted to use gather-scatter functionality of networks such as InfiniBand to improve performance of sending non-contiguous buffers, as well as attempting to achieve zero-copy communication [26], [27], [28].

Many newer languages, such as Python, Julia, and Rust, have binding libraries for MPI. mpi4py [7] implements support for Python, exposing most MPI calls. MPI.jl [6] provides MPI support for the Julia Language, borrowing ideas from mpi4py while also modeling the interface on the MPI C and C++ APIs. RSMPI [9] is the Rust binding library for MPI, attempting to provide better safety guarantees and a more comfortable interface for Rust programmers.

These languages also all provide some level of support for serialization. Python comes with the Pickle library [2] that allows for serialization of complex Python objects into a custom binary format. Julia includes serialization and deserialization methods that write or read from a stream of data [1]. Rust supports serialization through a variety of serialization libraries, with the Serde [4] crate being the current standard approach; Serde provides generic infrastructure code used by other libraries to implement multiple serialization formats. Serde was not used for our Rust benchmarks since we wanted to compare performance with manual packing and avoid any associated overhead with the serialization library itself. In practice an extended Rust MPI implementation supporting our new type interface may implement macros to automatically generate manual packing, instead of using Serde. Boost MPI[29] supports transmission of arbitrary C++ objects using the Boost serialization library. It supports a *is_mpi_datatype* trait which can be used to avoid extraneous copy overheads for objects of fixed size and offsets.

VIII. Conclusions

We have presented results using the proposed custom datatype API for several programming languages. Results are varied, but generally show that use of the proposed API is viable for transfer sizes where traditional serialize/deserialize approaches show poor performance. To more fully make effective use of the serialization packages avaialable with modern programming languages, in particular Python Pickle 5, additional MPI functionality needs to be developed. In particular, the generic challenge of efficiently receiving objects of undetermined size currently necessitates the use of multi-message protocols and inefficient message probing on the receive side. Mechanisms for delivering message meta-data to a receiver need to be developed if such protocols are to be avoided. More research is needed for implementing truly dynamic datatypes, where receiving objects of an undetermined size, or a list of sizes, may not be known before hand. This is important for both serialization and languages that have dynamic types that can be constructed at runtime. We also leave the integration with collective operations as future work, which we acknowledge as a requirement for standardization of our approach.

Acknowledgements

This research was supported partly by NSF awards #1931384 and #1931387. Jacob Tronge and Howard Pritchard acknowledge support by the National Nuclear Security Administration. Los Alamos National Laboratory is operated by Triad National Security, LLC for the U.S. Department of Energy under contract 89233218CNA000001. LA-UR-24-30217.

References

[1] Julia, "Julia," https://docs.julialang.org/en/v1/stdlib/Serialization/, 2024, accessed: 2024-5-9.

[2] Python, "Pickle," https://docs.python.org/3/library/pickle.html, 2024, accessed: 2024-5-9.

[3] D. Gregor and M. Troyer, "Boost. mpi," *MPI, November*, 2006.

[4] Serde, "Serde," https://github.com/serde-rs/serde, 2024, accessed: 2024-4-25.

[5] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf, 2023.

[6] S. Byrne, L. C. Wilcox, and V. Churavy, "Mpi.jl: Julia bindings for the message passing interface," *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, p. 68, 2021. [Online]. Available: https://doi.org/10.21105/jcon.00068

[7] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status update after 12 years of development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021.

[8] A. Pitrou, "PEP 574 – Pickle protocol 5 with out-of-band data," Python Software Foundation, PEP 574, 2018.

[9] RSMPI, "Rsmpi," https://github.com/rsmpi/rsmpi, 2024, accessed: 2024-5-9.

[10] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "Ucx: an open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[11] J. Tronge, H. Pritchard, and J. Brown, "Improving mpi safety for modern languages," in *Proceedings of the 30th European MPI Users' Group Meeting*, ser. EuroMPI '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3615318.3615328

[12] D. K. D. Panda, "Osu micro-benchmarks," https://mvapich.cse.ohio-state.edu/benchmarks/, 2024, accessed: 2024-05-16.

[13] T. Schneider, R. Gerstenberger, and T. Hoefler, "Micro-Applications for Communication Data Access Patterns and MPI Datatypes," in *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, vol. 7490. Springer, Sep. 2012, pp. 121–131.

[14] D. Gregor, J. Squyres, and A. Lumsdaine, "Mpi for high-level languages," 2008.

[15] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar, "Modernizing the c++ interface to mpi," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 266–274.

[16] D. Hespe, L. Hübner, F. Kurpicz, P. Sanders, M. Schimek, D. Seemaier, C. Stelz, and T. N. Uhl, "Kamping: Flexible and (near) zero-overhead c++ bindings for mpi," *arXiv preprint arXiv:2404.05610*, 2024.

[17] B. Carpenter, G. Fox, S. H. Ko, and S. Lim, "Object serialization for marshalling data in a java interface to mpi," in *Proceedings of the ACM 1999 conference on Java Grande*, 1999, pp. 66–71.

[18] I. B. Peng, S. Markidis, E. Laure, D. Holmes, and M. Bull, "A data streaming model in mpi," in *Proceedings of the 3rd Workshop on Exascale MPI*, ser. ExaMPI '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2831129.2831131

[19] T. Jammer, A. Hück, J.-P. Lehr, J. Protze, S. Schwitanski, and C. Bischof, "Towards a hybrid mpi correctness benchmark suite," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 46–56. [Online]. Available: https://doi.org/10.1145/3555819.3555853

[20] T. Schneider, F. Kjolstad, and T. Hoefler, "Mpi datatype processing using runtime compilation," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 19–24. [Online]. Available: https://doi.org/10.1145/2488551.2488552

[21] T. Prabhu and W. Gropp, "DAME: Runtime-compilation for data movement," *The International Journal of High Performance Computing Applications*, vol. 32, no. 5, pp. 760–774, 2018. [Online]. Available: https://doi.org/10.1177/1094342017695444

[22] Y. Li, J. Schuchart, and G. Bosilca, "Comprehensive Study for Just-In-Time Pack Functions in Open MPI," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 678–685. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPSW63119.2024.00130

[23] J. L. Träff, "A library for advanced datatype programming," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 98–107. [Online]. Available: https://doi.org/10.1145/2966884.2966904

[24] H. Zhou, K. Raffenetti, Y. Guo, T. Gillis, R. Latham, and R. Thakur, "Designing and prototyping extensions to mpi in mpich," 2024.

[25] D. Kimpe, D. Goodell, and R. Ross, "Mpi datatype marshalling: a case study in datatype equivalence," in *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 82–91.

[26] G. Santhanaraman, J. Wu, W. Huang, and D. K. Panda, "Designing zero-copy message passing interface derived datatype communication over infiniband: Alternative approaches and performance evaluation," *The International Journal of High Performance Computing Applications*, vol. 19, no. 2, pp. 129–142, 2005.

[27] A. Gainaru, R. L. Graham, A. Polyakov, and G. Shainer, "Using infiniband hardware gather-scatter capabilities to optimize mpi all-to-all," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 167–179. [Online]. Available: https://doi.org/10.1145/2966884.2966918

[28] S. Di Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler, "Network-accelerated non-contiguous memory transfers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356189

[29] D. Gregor and M. Troyer, "Boost mpi tutorial," https://www.boost.org/doc/libs/1_85_0/doc/html/mpi/tutorial.html, 2024.