

# MPI Datatype Marshalling: A Case Study in Datatype Equivalence

Dries Kimpe<sup>1,2</sup>, David Goodell<sup>1</sup>, and Robert Ross<sup>1</sup>

<sup>1</sup> Argonne National Laboratory, Argonne, IL 60439

<sup>2</sup> University of Chicago, Chicago, IL 60637  
{dkimpe,goodell,ross}@mcs.anl.gov

**Abstract.** MPI datatypes are a convenient abstraction for manipulating complex data structures and are useful in a number of contexts. In some cases, these descriptions need to be preserved on disk or communicated between processes, such as when defining RMA windows. We propose an extension to MPI that enables marshalling and unmarshalling MPI datatypes in the spirit of `MPI_Pack/MPI_Unpack`. Issues in MPI datatype equivalence are discussed in detail and an implementation of the new interface outside of MPI is presented. The new marshalling interface provides a mechanism for serializing all aspects of an MPI datatype: the typemap, upper/lower bounds, name, contents/envelope information, and attributes.

## 1 Introduction

Since its inception, MPI has provided *datatypes* to describe the location of data in memory and files for communication and I/O. These datatypes are a flexible and powerful abstraction, capable of efficiently expressing extremely sophisticated data layouts. While MPI offers facilities to simply and efficiently transmit, store, and retrieve data described by these datatypes, however, it does not provide any direct mechanism to transmit the datatype description itself.

We originally set out to develop a library capable of *marshalling* MPI datatypes. We define marshalling to be the act of generating a representation of an MPI datatype that can be used to recreate an “equivalent” datatype later, possibly in another software context (such as another MPI process or a postprocessing tool). Such functionality is useful in many cases, such as the following:

- Message logging for fault-tolerance support
- Self-describing archival storage
- Type visualization tools
- Argument checking for collective function invocations
- Implementing “one-sided” communication, where the target process does not necessarily know the datatype that will be used
- Message or I/O tracing for replay in a tool or simulator.

When viewed in the abstract or from the perspective of a particular use case, datatype marshalling appears to be a well-defined problem with a number of direct solutions. As we considered the problem from several different angles, however, we consistently came up with different, sometimes incompatible, requirements. These requirements stem from the lack of a clear definition for “equivalent” MPI datatypes.

The rest of this paper is organized as follows. In Section 2 we discuss the thorny issue of MPI datatype equivalence. In Section 3 we present the design and implementation of our datatype marshalling library. In Section 4 we briefly evaluate the time and space performance of our implementation. In Section 5 we discuss related work, and our conclusions in Section 6.

## 2 MPI Datatype Equivalence

Pragmatically, two MPI datatypes might generally be considered equivalent when one can be substituted for another in MPI operations. However, datatypes are characterized by several independent dimensions that may constitute a concrete definition of equivalence.

The MPI standard [5] provides one definition for datatype equivalence (MPI-2.2 §2.4):

Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

Capturing the extent of the type is critical in cases where a count  $\geq 1$  is used. Section 4.1.7 states that `MPI_Type_create_resized` does the following:

Returns in `newtype` a handle to a new datatype that is identical to `oldtype`, except that the lower bound of this new datatype is set to be `lb`, and its upper bound is set to be `lb + extent`. Any previous `lb` and `ub` markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the `lb` and `extent` arguments.

If one sensibly interprets this as stating that `MPI_Type_create_resized`’s effect is to insert `MPI_LB` and `MPI_UB` markers into the typemap,<sup>3</sup> then the MPI standard definition provides an adequate definition for point-to-point, collective, one-sided (RMA), and I/O operations. If the typemaps match, then the MPI operations will access the same data items.<sup>4</sup>

---

<sup>3</sup> It is interesting that `MPI_LB` and `MPI_UB`, while deprecated for being error-prone to use, are extremely helpful in understanding the equivalence of datatypes.

<sup>4</sup> The implicit pad ( $\epsilon$ ) used in an MPI executable is intended to mimic the alignment behavior of the compiler used. This can vary based on architecture, compiler, and compiler flags, and it is not explicitly captured in the typemap.

---

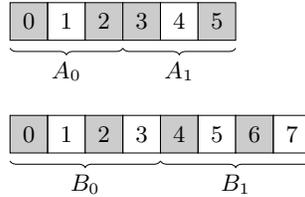
```

1 MPI_Aint lb, extent;
2 MPI_Datatype A, B;
3 MPI_Type_vector(2, 1, 2, MPI_BYTE, &A);
4 MPI_Type_get_extent(A, &lb, &extent);
5 MPI_Type_create_resized(A, 0, extent+1, &B);

```

---

**Listing 1.** MPI Code to Create MPI Datatypes  $A$  and  $B$



**Fig. 1.**  $A$  and  $B$  used in `MPI_Send` with `count = 2`. (shaded boxes indicate transmitted bytes)

In explanation, consider two types,  $A$  and  $B$ , with identical typemaps but differing extents,  $E_A$  and  $E_B$ . The code to create these types is shown in Listing 1. If `MPI_Send` is invoked with `count = 2` and alternately with  $A$  and  $B$ , different data will be sent (Figure 1). The typemap for  $B$  must incorporate the `MPI_UB` defined by the `resize`.

We note that no facility for comparing datatypes is provided in the MPI standard, as it is for comparing communicators (i.e., `MPI_Comm_compare`). This omission complicates the construction of external libraries that would marshal datatypes, as it is impossible to detect equivalent datatypes without dissecting the datatype via the `envelope` and `contents` calls.

In many contexts, several additional characteristics besides the typemap may determine the semantic equivalence of two types. These include the type names, construction sequence, and attribute values.

## 2.1 Type Name Equivalence

The name associated with an MPI datatype may be changed by the `MPI_Type_set_name` routine. This information is not considered in the MPI standard’s definition of type equivalence. For example, a type used to represent the layout of a dataset in a file may be named by that dataset’s name. In some cases, an application or library may not consider two types to be equivalent unless the types’ names are also equivalent.

## 2.2 Constructor Equivalence

The MPI-2 standard introduced two functions and a handful of constants that together provide a form of type introspection. The `MPI_Type_get_envelope` function

returns the “combiner” used to create the type. Combiner examples include `MPI_COMBINER_VECTOR`, `MPI_COMBINER_RESIZED`, and `MPI_COMBINER_NAMED`, corresponding to `MPI_Type_vector`, `MPI_Type_create_resized`, and a named predefined datatype such as `MPI_INT`. The complementary function, `MPI_Type_get_contents`, returns information sufficient to recreate the call to the combiner routine, such as the input datatypes, counts, and indices. The MPI standard requires that “the actual arguments used in the creation call for a datatype can be obtained,” including zero-count arguments. This requirement goes beyond the MPI standard’s definition of equivalence, as elements with a zero count do not appear in the typemap of the constructed datatype.

Other than a heavyweight, noncomposable scheme involving the MPI profiling interface (`PMPI_*` functions), the envelope and contents routines are the only available mechanisms for determining the make-up of an MPI datatype. Thus, any external scheme for marshalling datatypes will use this interface.

### 2.3 Attribute Equivalence

MPI provides attributes to allow applications and libraries to attach process-local information to communicator, window, and datatype objects. We limit our discussion of attributes to datatypes. An attribute attached to a datatype object is a (key,value) pair, with exactly one value for a given key. Keys are integer values, allocated in a process-local context via `MPI_Type_keyval_create` and deallocated by `MPI_Type_keyval_free`. Attribute values are void pointers<sup>5</sup> and can be queried/set/deleted with the `MPI_Type_{get,set,delete}_attr` functions.

Creating a keyval both reserves a key for later use and associates a set of function pointers with that key. The corresponding function pointer is invoked by the MPI implementation when types are copied (via `MPI_Type_dup`) or deleted (via `MPI_Type_free`) as well as when attributes themselves are explicitly replaced or deleted. These function pointers are responsible for copying underlying attribute values and cleaning up associated storage according to the semantics of that attribute’s usage.

For example, the MPITypes library [8] uses attributes to cache high-performance dataloop [9] representations of MPI datatypes on the datatypes themselves. This strategy allows the MPITypes library to avoid recomputing the dataloop representation on every use. The dataloop information could be stored externally, without the use of the attribute code, but the attribute system provides two advantages. First, if a type is duplicated via `MPI_Type_dup`, the dataloop representation can be trivially copied, or shared and reference counted. Second, the dataloop can be easily freed when the type is freed. Otherwise the MPITypes library has no easy means to identify when a type is no longer in use; hence, it must use an external caching scheme with bounded size and an eviction policy, or memory usage will grow without bound.

Two MPI datatypes that are equivalent modulo their attributes may or may not be semantically equivalent, depending on the particular usages of those at-

---

<sup>5</sup> Attribute values are address-sized integers (`KIND=MPI_ADDRESS_KIND`) in Fortran.

---

```

1 int MPIX_Type_marshal(const char *typerep, MPI_Datatype type,
2     void *outbuf, MPI_Aint outsize, MPI_Aint *num_written);
3 int MPIX_Type_marshal_size(const char *typerep, MPI_Datatype type,
4     MPI_Aint *size);
5 int MPIX_Type_unmarshal(const char *typerep, void *inbuf,
6     MPI_Aint insize, MPI_Datatype *type);

```

---

**Listing 2.** Marshalling and Unmarshalling Function Prototypes

tributes. Consider the case of marshalling a type,  $t_1$  followed by unmarshalling the obtained representation into a second type,  $t_2$ . If the marshalling system naïvely fails to preserve attributes during this round trip, any attributes, such as the data loop from the MPI Types example, must be recalculated for  $t_2$  when accessed later. If the attribute value is essential for correct operation and cannot be recalculated, erroneous program behavior may occur. Therefore, any complete MPI datatype marshalling solution should provide the capability to also marshal a datatype’s attributes. Section 3 details one approach to maintaining attributes despite serialization.

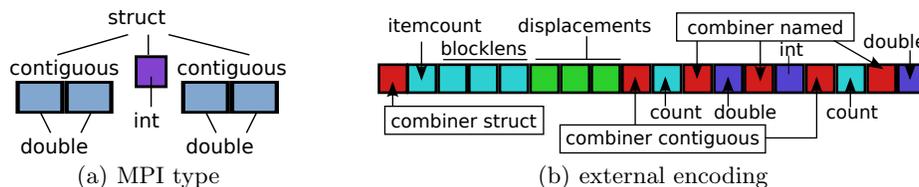
### 3 MPI Datatype Marshalling

Listing 2 shows the function prototypes of the marshalling and unmarshalling functions. We modeled our prototypes on those of the packing and unpacking functions (`MPI_Type_{un}pack`) defined by the MPI standard. `MPIX_Type_marshal_size` returns an upper bound for the space required to marshal the given type. `MPIX_Type_unmarshal` reconstructs the datatype. If the type passed to `MPIX_Type_marshal` was a named type, such as `MPI_INT`, the same named type will be returned when unmarshalling. The committed state of the returned datatype is undefined, and the user is responsible for freeing the type if it is not a built-in type.

The representation parameter allows the user to choose which encoding will be used to marshal the type definition. Our library currently defines three type representations: **internal**, **external**, and **compressed**.

A datatype marshalled by using “internal” representation can be unmarshalled only by a process of the same parallel program as the marshalling process. As such, datatypes using “internal” encoding cannot be stored on disk to be retrieved later. The main advantage of using “internal” encoding is that it enables MPI library specific optimizations. For example, an MPI implementation could use its internal type description as the “internal” encoding, avoiding repeated calls to the MPI type construction and introspection functions to marshal and unmarshal a datatype. In addition, any optimizations performed by `MPI_Type_commit` could be captured and stored as well, making sure these optimizations don’t need to be repeated for the unmarshalled type.

Similar to the “external32” data representation in MPI-IO, the “external” type representation has a well-defined layout ensuring the marshalled type can be unmarshalled by an MPI program using another MPI implementation. The



**Fig. 2.** External datatype encoding

“external” format is described in Section 3.1. The “compressed” format reduces the space consumed by a marshalled type at the expense of additional computation to marshal and unmarshal the type. The “compressed” type representation is described in Section 3.4.

### 3.1 External Type Representation

We chose to use the eXternal Data Representation standard (XDR) [10] to portably store a datatype description. The “external” type format follows a top-down model: the MPI datatype is first broken down into its combiner and associated data. For example, the type shown in Figure 2 has a top-level combiner of `MPI_COMBINER_CONTIGUOUS`. The combiner is converted into an integer by using a lookup table.<sup>6</sup> The integer is stored in XDR encoding. `MPI_Type_contiguous` has two more parameters: a base type and a count describing how many times the base type is to be repeated. The count is stored using XDR. Next, the process repeats but this time for the base type, essentially descending into the datatype. Since no cycles are possible in MPI datatypes, this process will eventually end at a leaf node of the datatype. As all derived datatypes are ultimately built from predefined types, this leaf node must be a named predefined type or unnamed Fortran predefined type. Named types are marshalled simply by storing the code for a `MPI_COMBINER_NAMED`, followed by an integer identifying the named type, ending the recursion.

In general, each MPI datatype constructor will be marshalled to a combiner and zero or more integers, addresses, and MPI datatypes. By defining a portable way to store the combiner, integers, and addresses (which is provided by XDR), and the set of named datatypes, using recursion, any MPI datatype can be portably marshalled and unmarshalled.

### 3.2 Marshalling Type Names

Marshalling type names is relatively straightforward. The name can easily be obtained using `MPI_Type_get_name`, and stored using the XDR representation for

<sup>6</sup> This conversion is done because the actual value of `MPI_COMBINER_CONTIGUOUS` is not specified by the MPI standard and thus might differ between MPI implementations. The same is true for the value of the named datatypes. Such link-time constants may not be used as labels in a C-language `switch` statement.

---

```

1  /* provides upper bound on buffer size */
2  typedef int MPIX_Type_marshal_attr_size_function(int keyval,
3      const char *canonical_name, const char *typerep,
4      MPI_Datatype type, MPI_Aint *size);
5  /* Marshals attribute value specified by keyval/canonical_name into
6     outbuf. Sets (*num_written)=0 if outsize isn't big enough. */
7  typedef int MPIX_Type_marshal_attr_function(int keyval,
8      const char *canonical_name, const char *typerep,
9      MPI_Datatype type, void *outbuf, MPI_Aint outsize,
10     MPI_Aint num_written);
11 /* responsible for setting attribute on type */
12 typedef int MPIX_Type_unmarshal_attr_function(
13     const char *canonical_name, const char *typerep,
14     MPI_Datatype type, void *inbuf, MPI_Aint insize);
15 int MPIX_Type_register_marshaled_keyval(int keyval,
16     const char *canonical_name,
17     MPIX_Type_marshal_attr_size_function *marshal_size_fn,
18     MPIX_Type_marshal_attr_function *marshal_fn,
19     MPIX_Type_unmarshal_attr_function *unmarshal_fn);

```

---

**Listing 3.** Attribute Marshalling Function Prototypes

character strings. When unmarshalling, any name found in the data stream is reattached to the type handle using `MPI_Type_set_name`.

### 3.3 Marshalling Attributes

As discussed in Section 2.3, supporting marshalling and unmarshalling of attributes is desirable and can often simplify libraries or optimize type handling. Marshalling and unmarshalling attributes is not straightforward, however, as there is no easy way to obtain the attributes associated with a datatype. The MPI standard does not provide a function capable of retrieving the set of attributes associated with a particular handle.

In addition, attributes have a user-defined meaning and cannot be portably interpreted by a library. `MPI_Type_dup`, which must copy attributes to the new handle, faces similar issues. The solution adopted by the standard requires that when a new keyval is registered, two function pointers are provided. One is called when an attribute needs to be copied, and one is called when an attribute needs to be destructed (for example, when the object it is associated with is freed).

For each keyval that needs to be marshalled, a call to `MPI_Type_register_marshaled_keyval` must be made. Since keyvals have limited process local scope, their actual value cannot be marshalled. Instead, the library associates each keyval with a *canonical name*. When unmarshalling the attribute, this mapping is used to retrieve the correct local keyval for the attribute.

Listing 3 presents the C binding function prototypes for our proposed attribute marshalling interface. It closely mirrors the datatype marshalling in-

terface; the corresponding function will be called by the datatype marshalling system when a type is marshalled or unmarshalled.

In order to store attributes, the “external” representation described in Section 3.1 is extended by following the type description with an integer indicating the number of attributes that follow in the data stream. Each attribute is stored by storing its canonical name, followed by the data provided by its `MPIX_Type_marshall_function`. Note that this data is not modified in any way by the library and is stored as opaque XDR data. Therefore, the user is responsible for serializing the attribute in a portable fashion.

### 3.4 Compression

A given base type might be used multiple times in constructing a derived datatype. For example, the type shown in Figure 2 contains multiple copies of a type composed out of two doubles. The marshalled representation of the complete type,  $C$ , should ideally contain only one copy. Unfortunately, there is no easy way to detect reuse of types. According to the MPI standard, `MPI_Type_get_contents` returns handles to datatypes that are *equivalent* to the datatypes used to construct the type. There is no guarantee that the returned handle will be the same as the handle used in constructing the datatype.

Therefore, our library relies on a non-type-specific compression function (zlib) to remove duplicate datatypes from the marshalled representation. Compression can be requested by passing “compressed” as the requested encoding to the marshalling functions.

## 4 Evaluation

We evaluated the marshalling and unmarshalling functions for a number of MPI datatypes, using both “external” and “compressed” representations. We timed 10,000 iterations for each operation and reported the mean time per iteration. Table 1 shows the results.

The first type evaluated (“named”) refers to any predefined MPI datatype. As each predefined type is treated equally by the library, the numbers listed are valid for any predefined type. The second type tested (“indexed”) is an `MPI_Type_indexed` type selecting three contiguous regions from a byte array. The third type is a complex derived datatype we captured from the HDF5 [3] storage library. This type was created to describe the on-disk access pattern used when accessing 5000 bytes of a dataset stored in the HDF5 file.<sup>7</sup> This particular type is 12 type constructors deep.

As expected, more complicated types take additional space and time to marshal. The named and indexed types are cheaply serialized in the “external” format. In the case of the complex type, the “compressed” format consumes  $\approx 8.6$  times less space but takes  $\approx 6.4$  times as long to marshal.

---

<sup>7</sup> The exact type can be found in the MPITypes distribution [7] as `test/very_deep.c`.

**Table 1.** Evaluation of marshalling and unmarshalling time and space consumption in the prototype implementation.

Type	External			Compressed		
	Size (B)	Time ( $\mu$ s)		Size (B)	Time ( $\mu$ s)	
		Marshal	Unmarshal		Marshal	Unmarshal
Named	8	< 1	< 1	14	76	1
Indexed	40	1	4	30	79	3
Complex	824	23	33	95	147	34

Our marshalling implementation is currently a prototype and has not been extensively optimized. We expect that marshalling and unmarshalling times could be reduced further with additional effort. The data presented in Table 1 are intended to provide a rough idea of marshalling performance.

## 5 Related Work

One focus of research in MPI datatypes has been the detection of mismatched datatypes passed to MPI communication functions. Gropp introduced the idea of using hashes for this purpose and defined a hashing function that maps the type signature into an integer tuple [2]. Langou et al. extended this idea, proposing alternative hashing schemes and examining the performance of these approaches [4]. Falzone et al used this approach in a library for detecting user errors in the use of MPI collective communication calls [1], building on a simpler scheme first presented by Träff et al. [11]. These hashes can be used, for instance, to reference a datatype in a local cache, allowing a remote entity to query if a datatype is already represented in the cache.

Another focus has been in the efficient processing of MPI datatypes. Ross et al. describe the implementation of datatype processing used in MPICH2 [9] and an external library for processing MPI datatypes [8]. The approach used is similar to the one first described by Träff et al. [12]. Recently, Mir and Träff studied *transpacking*, or moving data between datatype representations [6]. These approaches generally rely on a simplified, underlying datatype representation with a type signature identical to the original user datatype. When these representations are available, they allow marshalling of a simplified description of a datatype, if envelope and contents information is not needed.

The Hierarchical Data Format version 5 (HDF5) [3] provides functionality similar to the MPI datatypes (called *datasets* in HDF5), splitting the definition of a dataset into a *dataspace* that describes the organization of elements and a datatype that describes a single element, similar to an MPI struct. HDF5 stores these descriptions persistently in HDF5 files, but it does not present an interface for marshalling these descriptions to users.

## 6 Conclusions and Future Work

In this paper we have discussed the notion of MPI datatype equivalence, arguing that the definition put forth in the standard is appropriate only for a certain set of use cases. We have identified a number of other interpretations, and we have provided an API and a library of functions that enable marshalling of MPI datatypes in order to meet various levels of equivalence.

We intend to release this functionality for general use in the MPITypes library [8, 7]. We also plan to investigate using a combination of MPI attribute caching and datatype hashing techniques [4] to optimize the case when types are repeatedly serialized. Issues in the design and implementation of “internal” marshalling schemes also merit further study. We intend to propose this interface for the MPI-3 standardization process.

## Acknowledgments

This work was supported by the U.S. Department of Energy, under Contract DE-AC02-06CH11357.

## References

1. Falzone, C., Chan, A., Lusk, E., Gropp, W.: A portable method for finding user errors in the usage of MPI collective operations. *International Journal of High Performance Computing Applications* 21(2), 155–165 (2007)
2. Gropp, W.: Runtime checking of datatype signatures in MPI. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. LNCS, vol. 1908, pp. 160–167. Springer (2000)
3. HDF5. <http://hdf.ncsa.uiuc.edu/HDF5/>
4. Langou, J., Bosilca, G., Fagg, G., Dongarra, J.: Hash functions for datatype signatures in MPI. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. LNCS, vol. 3666, pp. 76–83. Springer (2005)
5. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 2.2 (September 2009), <http://www.mpi-forum.org/docs/docs.html>
6. Mir, F., Träff, J.: Constructing MPI input-output datatypes for efficient transpacking. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. LNCS, vol. 5205, pp. 141–150. Springer, Berlin (September 2008)
7. MPITypes library. <http://www.mcs.anl.gov/mpitypes/>
8. Ross, R., Latham, R., Gropp, W., Lusk, E., Thakur, R.: Processing MPI datatypes outside MPI. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. LNCS, vol. 5759, pp. 42–53. Springer (September 2009)
9. Ross, R., Miller, N., Gropp, W.: Implementing fast and reusable datatype processing. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. LNCS, vol. 2840, pp. 404–413. Springer (October 2003)
10. Srinivasan, R.: XDR: External data representation standard (1995)
11. Träff, J., Worrigen, J.: Verifying collective MPI calls. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. LNCS, vol. 3241, pp. 18–27. Springer (2004)
12. Träff, J.L., Hempel, R., Ritzdorf, H., Zimmermann, F.: Flattening on the fly: Efficient handling of MPI derived datatypes. In: *PVM/MPI 1999*. pp. 109–116 (1999)