TEMPI: An Interposed MPI Library with a Canonical Representation of CUDA-aware Datatypes

Carl Pearson
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA
pearson@illinois.edu

Kun Wu
University of Illinois
Urbana-Champaign
Urbana, Illinois, USA
kunwu2@illinois.edu

I-Hsin Chung
IBM T. J. Watson Research
Yorktown Heights, New York, USA
ihchung@us.ibm.com

Jinjun Xiong
IBM T. J. Watson Research
Yorktown Heights, New York, USA
jinjun@us.ibm.com

Wen-Mei Hwu
Nvidia Research
Champaign, Illinois, USA
whwu@nvidia.com

ABSTRACT

MPI derived datatypes are an abstraction that simplifies handling of non-contiguous data in MPI applications. These datatypes are recursively constructed at runtime from primitive Named Types defined in the MPI standard. More recently, the development and deployment of CUDA-aware MPI implementations has encouraged the transition of distributed high-performance MPI codes to use GPUs. Such implementations allow MPI functions to directly operate on GPU buffers, easing integration of GPU compute into MPI codes. Despite substantial attention to CUDA-aware MPI implementations, they continue to offer crippling poor GPU performance when manipulating derived datatypes on GPUs. This work presents a new MPI library, TEMPI, to address this issue. TEMPI first introduces a common datatype to represent equivalent MPI derived datatypes. TEMPI can be used as an interposed library on existing MPI deployments without system or application changes. Furthermore, this work presents a performance model of GPU derived datatype handling, demonstrating that previously preferred “one-shot” methods are not always fastest. Ultimately, the interposed-library model of this work demonstrates MPI_Pack speedup of up to 242,000× and MPI_Send speedup of up to 59,000× compared to the MPI implementation deployed on a leadership-class supercomputer. This yields speedup of more than 1000× in a 3D halo exchange at 192 ranks.

CCS CONCEPTS

- Software and its engineering → Software libraries and repositories; Message passing.
- Computing methodologies → Massively parallel algorithms; Distributed algorithms; Theory of computation → Massively parallel algorithms.

1 INTRODUCTION

MPI derived datatypes [16, p. 83] are a useful abstraction for describing the layout of non-contiguous data in memory. They allow MPI functions to operate on such data without intermediate handling by the user application, especially packing the data into a contiguous buffer before transfer. As GPUs have become a dominant high-performance computing accelerator, MPI implementations such as OpenMPI [6], MVAPICH [13], Spectrum MPI [9] and MPICH [7] have become “CUDA-aware”. In such implementations MPI can directly operate on CUDA device allocations to streamline application development and potentially accelerate inter-rank transfers of GPU-resident data.

Previous works have created specialized strategies for each datatype [15], contributed solutions to handling arbitrary datatypes on GPUs [10], integrated datatype handling with the communication layer [8, 18], and tackled the latency of GPU operations [5]. Despite broad GPU deployment in distributed computing and substantial work in datatype handling, high-performance handling of GPU datatypes remains a rare feature. MVAPICH-GDR features some fast handling of GPU datatypes, but requires [12] the unrelated Mellanox OpenFabrics Enterprise Distribution, which is tied to specific network hardware not present on all systems.

Therefore the challenge of deployment appears to be fundamental and unaddressed by efforts so far. Consequently, one contribution of this work is to demonstrate that an interposed library can transparently deliver large derived datatype performance improvements without application modification. The Topology Experiments for MPI (TEMPI) library implements this work and has been tested with OpenMPI 4.0.5, MVAPICH 2.3.4, and Spectrum MPI 10.3.1.2. It transparently converts non-contiguous GPU-resident types to contiguous data before it is passed to the underlying MPI implementation. TEMPI is available at https://github.com/cwpearson/tempi.
This work also presents a new approach to handling strided MPI datatypes. Prior work (Section 7) recognizes that MPI datatypes can be generalized to a list of contiguous blocks defined by offsets and sizes. Further optimizations include specialization for types with certain kinds of regularity [15, 17] or handling arbitrary datatypes [4, 10]. This work draws a middle ground by observing that compositions of contiguous, vector, hvector, and subarray types are all special cases of an object suitable for compact representation. A translation phase converts the datatype into an in-memory representation, a translation phase generates a canonical simplified representation, and a parameterized kernel is selected to pack and unpack the data transparently. This affords wide coverage of structured non-contiguous data without performance fragility of specialized kernels or large metadata sizes for arbitrary datatype handling.

Thirdly, this work presents a performance model for understanding the impact of packing strategies, and shows it can be used at runtime to improve performance of datatype handling. Prior work has largely focused on “one-shot” methods where non-contiguous GPU data is transferred directly into contiguous host memory through the CUDA “zero-copy” mechanism. This model shows that the one-shot method may not be preferable for non-contiguous data with large strides. The model is evaluated in the context of the OLCF Summit system. Prior work has also investigated deeper integration with MPI or data transfer performance details [4, 5, 8, 18]. TEMPI’s interposer design makes it compatible with all widely-deployed MPI implementations, but it cannot modify the underlying data-transfer primitives in this way. TEMPI’s interposer design means underlying transfer primitives cannot be modified, but compatibility with all widespread MPI deployments is retained.

TEMPI demonstrates a speedup of up to $242,000 \times$ for MPI_Pack and MPI_Unpack, $59,000 \times$ for MPI_Send, and $20,000 \times$ for a 3D stencil halo exchange on the OLCF Summit system, which does not natively support fast datatype operations on GPU.

This paper is organized in the following way: Section 2 introduces MPI derived datatypes, their composition, and the need for compact representation. Section 3 describes how TEMPI transforms datatypes and selects the kernel. Section 4 describes the library-interposer method that makes the derived type modifications available without application modification. Section 5 describes how datatype-accelerated MPI primitives can be created without system MPI support. Section 6 describes the microbenchmark and 3D stencil results. Section 8 describes future work for the library. Section 7 describes related work. Finally, Section 9 concludes.

2 COMPOSITION AND REPRESENTATION OF DERIVED DATATYPES

Many MPI datatypes can be composed to describe multi-dimensional strided objects. This work considers the following subset due to their natural application to describing such objects.

- “predefined” or “named” [16, §3.2.2]: these are the base MPI datatypes (MPI_BYTE, MPI_FLOAT, etc) that correspond to various C or Fortran types.

- “contiguous” [16, §4.1.2]: these describe “replication of a datatype in contiguous locations.” MPI_Type_contiguous(n, oldtype, newtype) = newtype is n contiguous repetitions of oldtype.

- “vector/hvector” [16, §4.1.2]: these describe “replication of a datatype into...equally spaced blocks.” MPI_Type_vector(c, l, s, oldtype, newtype) = a vector of c blocks, each block is l contiguous repetitions of oldtype and the beginning of each block is separated by s contiguous repetitions of oldtype. For hvector, s is given in bytes instead.

- “subarray” [16, §4.1.3]: these describe “n-dimensional subarray of an n-dimensional array.” MPI_Type_subarray(n, [sizes], [subsizes], [offsets] order, oldtype, newtype) = newtype is an n-dimensional subarray of an oldtype array with extent sizes. The subarray is of extent subsizes at offset offsets. order controls C or FORTRAN ordering.

These may be composed in many ways to describe the same non-contiguous byte. For example, consider the 3D object in Fig. 1, which can be visualized as a three-dimensional sub-object of an enclosing three-dimensional object, where the sub-object shares an origin with the enclosing object and each element of the object is a single-precision floating-point number (an MPI_FLOAT), consuming four bytes.

![Figure 1: A 3D object with extent $E_0 \times E_1 \times E_2$ floats in an allocation $A_0 \times A_1 \times A_2$ bytes, and the corresponding linearized memory layout.](image)

Each 1D row of the object ($E_0 \times 4$ contiguous bytes) to be described in many ways; a non-exhaustive list follows (meanings of function parameters described in the bulleted list above):

- MPI_Type_contiguous($E_0$, MPI_FLOAT, &row): “row” comprises a contiguous replication of $E_0$ single-precision floating-point (4-byte) elements.

- MPI_Type_vector(1, $E_0$, 1, MPI_FLOAT, &row): “row” is $E_0 \times 1$-byte elements.

- MPI_Type_create_hvector($E_0 \times 4$, MPI_BYTE, &row): “row” is $E_0 \times 4$-byte elements.

- MPI_Type_create_subarray(1, {A0}, {E0}, {0}, MPI_ORDER_C, MPI_FLOAT, &row)

- MPI_Type_create_subarray(1, {A0}, {E0 \times 4}, {0}, MPI_ORDER_C, MPI_BYTE, &row)

These are equivalent for describing a single row, but are not entirely interchangeable since their extents vary. This distinction is
relevant for certain compositions of these types (e.g., below), or when multiple types are manipulated at once. A 2D plane ($E_1$ rows, offset by $A_0$ bytes between the beginning of each row) can be constructed directly from named types:

- $\text{MPI Type vector}(E_1, E_0, A_0, \text{MPI FLOAT}, \&\text{plane})$
- $\text{MPI Type vector}(E_1, E_0 \times 4, A_0, \text{MPI BYTE}, \&\text{plane})$
- $\text{MPI Type create subarray}(2, \{A_0, A_1\}, \{E_0, E_1\}, \{0, 0\}, \text{MPI ORDER_C, MPI FLOAT, \&\text{plane}})$
- $\text{MPI Type create subarray}(2, \{A_0 \times 4, A_1\}, \{E_0 \times 4, E_1\}, \{0, 0\}, \text{MPI ORDER_C, MPI BYTE, \&\text{plane}})$

Alternatively, as an hvector of rows:

- $\text{MPI Type create hvector}(E_1, 1, A_0, \text{row, \&\text{plane}})$
- or for the subarray row types:
  - $\text{MPI Type vector}(E_1, 1, 1, \text{row, \&\text{plane}})$
  - $\text{MPI Type create subarray}(1, A_1, E_1, 0, \text{MPI ORDER_C, row, \&\text{plane}})$

Similarly planes comprise a cuboid ($E_2$ planes, offset by $A_0 \times A_1$ bytes between the beginning of each plane). For example, the cuboid can be constructed:

- $\text{MPI Type create hvector}(E_2, 1, A_0 \times A_1, \text{plane, \&\text{cuboid}})$
- $\text{MPI Type create subarray}(2, \{A_0, A_1, A_2\}, \{E_0, E_1, E_2\}, \{0, 0, 0\}, \text{MPI ORDER_C, MPI FLOAT, \&\text{cuboid}})$
- $\text{MPI Type create subarray}(2, \{A_0 \times 4, A_1, A_2\}, \{E_0 \times 4, E_1, E_2\}, \{0, 0, 0\}, \text{MPI ORDER_C, MPI BYTE, \&\text{cuboid}})$

At its most basic, a datatype can be considered to be a list of contiguous blocks, each defined by an offset and a size. Indeed, many prior works use such a representation in the general case [4, 8, 15, 17, 18], occasionally with additional optimization [10]. The weakness of this approach is that representing datatype may consume as much GPU memory as the datatype itself.

Consider such a type describing $N$ non-contiguous blocks of $M$ MPI FLOATs. To support objects dispersed across large address ranges, the block offset and size would be 8 bytes each, yielding at least $16 \times N$ bytes to represent $M \times N \times 4$ bytes of data. If $M$ is relatively small (common for any higher-dimension object) the representation will consume similar memory to the data itself, limiting the space left for the application.

Many works adopt specialized kernels to handle certain common datatypes [15, 17, 18]. These naturally lend themselves to specific compact representations, e.g., an MPI vector of any size as only a block length, block count, and stride. Unfortunately, the combinatorial explosion of equivalent representations renders the strategy of specialized kernels infeasible in general; Section 6 shows MVAPICH exhibiting this problem.

This work recognizes that structured data in many MPI applications does not require a generic and costly representation. Compositions of strided datatypes can adequately cover many cases and are amenable to a common compact representation despite the variety of equivalent constructions. Such a representation uses a negligible amount of GPU memory for each datatype, and minimizes the maintenance burden as a small number of generic packing kernels can cover many datatypes. Section 3 describes how this is achieved.

3 MPI DERIVED DATATYPE HANDLING

TEMPI provides a transparent translation (Section 4) from non-contiguous to contiguous data between the application and the MPI implementation. A packing strategy is created for each datatype the application calls MPI_Type_commit on. When later operations use that datatype, TEMPI first packs the non-contiguous data into a contiguous buffer before passing it on to MPI (and unpacks the data before returning it to the application). This necessarily places the packing and unpacking operations on the critical path. This section describes how the packing strategy is selected.

TEMPI uses a three-phase translation/translation/kernel selection approach to convert distinct-but-equivalent MPI datatypes to a common format. The MPI_Type_commit(datatype) function delineates the boundary between when an application constructs a datatype and when that type may be used with the rest of the MPI functions. The MPI standard advises that “the system may compile at commit time an internal representation for the datatype and select the most convenient transfer mechanism.” [16, p. 110]. In line with that advice, TEMPI’s phases are implemented within the MPI_Type_commit function and cached for later use in MPI functions:

1. Translation to an internal representation (IR) (Section 3.1)
2. Transformation to a compact representation (Section 3.2)
3. Kernel selection and on-device representation (Section 3.3)

3.1 Type Translation

The first phase of the datatype handling process is to convert the MPI derived datatype into a Type. Each Type represents a (possibly non-contiguous) set of bytes from a memory region. Each Type has a field data of TypeData, which represents information about the node and is also used to discriminate the kind of node. Each Type also tracks zero or more of its Type children. The root of the Type tree and its children describe the MPI datatype, the structure of the tree matches the pattern of the constructed MPI datatype. The IR currently includes two kinds of TypeData: DenseData for contiguous bytes, and StreamData for strided patterns of a single child Type. DenseData plays the same role as a named type in MPI: it represents a sequence of contiguous bytes and has no children.

1. DenseData
   a. integer offset, the number of bytes between the lower bound and the first byte of the Type
   b. integer extent, the number of contiguous bytes in the Type
2. StreamData, a strided sequence of elements of the child type
   a. integer offset, as DenseData
   b. integer stride, the number of bytes between elements
   c. integer count, the number of elements in the stream

This is done by converting each MPI datatype to a corresponding DenseData or StreamData node, and then recursively doing the same to its children before attaching them to the converted node. The recursive base case is when an MPI Named type is reached, which by definition have no children. Fig. 2 shows three different MPI C snippets to create the 3D object described in Fig. 1.

An MPI named type (MPI_INT, etc), is translated into a DenseData with the extent field equal to the extent of the named type, and offset 0. A named type is not a derived type, so it has no children.

An MPI contiguous type (MPI_Type_contiguous, etc), is a special case StreamData where the stride matches the size of the element. It is not DenseData as oldtype may not be dense. Offset is 0, stride equal to the extent of the oldtype argument, and count equal to the count argument.
3.2 Type Transformation

The construction of the Type tree described in Section 3.1 yields trees of StreamData with a leaf node of DenseData. In order to provide fast handling of equivalent types, these various representations need to be canonicalized and optimized.

Two transformations are used to canonicalize the Type tree. Dense Folding collapses DenseData into a parent StreamData. Stream Elision removes a StreamData representing a stream of one element. The optimizations are applied repeatedly in turn, only terminating when neither optimization modifies the Type. Algorithm 1 shows the overall simplification process.

3.2.1 Dense Folding. The first optimization is driven by the observation that stride of a StreamData may match the extent of a child DenseData. Such a configuration represents a stream of repeated contiguous dense elements. In that case, the DenseData extent can be "folded" up into the StreamData, and the pair can be represented as a single DenseData node. This scenario may arise when an MPI vector, subarray, or contiguous type is used to describe a contiguous region larger than any MPI named type.

| Algorithm 1: simplify |
|-----------------------|
| Function simplify(ty): |
| simplified ← ty |
| changed ← TRUE |
| while changed do |
| changed ← FALSE ∨ (dense_folding(simplified) ∨ in-place) |
| changed ← changed ∨ (stream_elision(simplified) ∨ in-place) |
| end |
| return ty |

Figure 3: Example of Dense Folding. When the extent of a DenseData matches the stride of a parent StreamData, the parent/child combination can be replaced with a single larger DenseData.

Algorithm 2 shows how the transformation is applied to a Type, and Fig. 3 shows an example. The transformation is applied to each Type node of the Type tree in a depth-first order. At each node, the transformation only applies if the node (ty) is a StreamData kind and the node’s child (child) is a DenseData. If the parent’s stride matches the child’s extent, the parent is replaced with a larger DenseData node that represents the entire contiguous stream. The child’s offset is increased to include any offset the parent had.

3.2.2 Stream Elision. Consider ty, a StreamData with a child StreamData whose count count is one. In such a case, child is a single
element and can be elided. This construction arises in the case of an MPI vector with blocklength one, or a subarray dimension with subsize one.

Algorithm 3 shows how the transformation is applied to a Type, and Fig. 4 shows an example. Like with dense folding, stream elision is applied separately to each Type node in a depth-first order. After that, if both the type ty and its child child are StreamData, then if the child has count of 1, the child is replaced with its own children.

![Algorithm 2: dense_folding from Alg. 1](image1)

**Algorithm 2: dense_folding from Alg. 1**

Function dense_folding(ty):

```plaintext
changed ← FALSE
for child of ty do
    changed ∨ dense_folding(child) ⇒ fold from bottom up
end
if ty.data is not StreamData then
    return changed
end
Type child = ty.children[0]
if child.data is not DenseData then
    return changed
end
StreamData cData ← child.data
StreamData pData ← ty.data
if cData.extent = pData.stride then
    cData.offset ← cData.offset + pData.offset
    cData.extent ← pData.count × pData.stride
    ty ← child ⇒ replace ty with child
end
return changed
```

![Algorithm 3: stream_elision from Alg. 1](image2)

**Algorithm 3: stream_elision from Alg. 1**

Function stream_elision(ty):

```plaintext
changed ← FALSE
for child of ty do
    changed ∨ changed ∨ stream_elision(child) ⇒ bottom up
end
if ty.data is not StreamData then
    return changed
end
Type child = ty.child
if child.data is not StreamData then
    return changed
end
StreamData cData ← child.data
if i = child.count then
    changed ← TRUE
    ty.child ← child.children ⇒ delete child
end
return changed
```

The first byte is start bytes from some arbitrary starting point. Each dimension describes count repetitions of the previous dimension, separated by stride bytes.

Algorithm 4 describes the conversion from Type to StridedBlock. This is only possible if the bottom is a DenseData and every other object is a StreamData. The process in Section 3.2 will apply the conversion if it is possible. The DenseData describes the first dimension, which will have stride 1 and count equal to the extent of the DenseData. Each higher dimension directly corresponds to the StreamData. The offset of each dimension is accumulated into the single offset of the StridedBlock.

![Algorithm 4: conversion of Type to StridedBlock](image3)

**Algorithm 4: conversion of Type to StridedBlock**

Function strided_block(ty):

```plaintext
datas ← []
while true do
    datas.append(cur) ⇒ Add all TypeData to an array
    cur ← ty
    datas.append(cur) ⇒ no children left
end
if cur.child = {} then
    break
else
    cur ← cur.child
end
```

```plaintext
StridedBlock sb ⇒ to be returned
for i = 0 to datas.size() do
    if i = 0 then
        if data is DenseData then
            sb.offset ← data.offset
            sb.counts.append(data.extent)
            sb.strides.append(1) ⇒ DenseData stride is 1
        else
            return NULL ⇒ Not strided
        end
    else
        if data is StreamData then
            sb.offset ← sb.offset + data.offset
            sb.counts.append(data.count)
            sb.strides.append(data.stride)
        else
            return NULL ⇒ Not strided
        end
    end
end
return sb
```

3.3 Kernel Selection

Once the type is canonicalized, it is converted into a StridedBlock structure. The StridedBlock structure is semantically similar to an MPI subarray and is used only to select the kernel implementation.

- StridedBlock
  - integer start: byte offset between the lower bound and the first element
  - integer[] counts: number of elements in the dimension
  - integer[] strides: bytes between the start of each element in the dimension
Once the Type is converted into a StridedBlock, the next task is to choose a method for fast packing and unpacking on the GPU. If the StridedBlock is 1D (contiguous), we issue a single cudaMemcpyAsync to move the data into the destination buffer, followed by a cudaMemcpyStreamSynchronize. This is similar to the implementation in MVAPICH, OpenMPI, and Spectrum MPI. If the StridedBlock is 2D we select a kernel that maps the X-dimension of the thread index into the count[0] and the Y-dimension to count[1]. If the StridedBlock is 3D, we map the X dimension to the count[0], Y dimension to the count[1], and Z dimension to the count[2]. Higher dimensional objects can follow the same general pattern, with additional outer loops for each dimension.

Each kernel is specialized to a word size $W$, which is the largest GPU-native type that is both aligned to the object and is a factor of count[0]. The X-dimension collaboratively loads count[0] contiguous bytes that make up each block using elements of size $W$.

In addition to a datatype, Pack and Unpack accept an incount/outcount parameter, which specify how many of the datatype are present in the input buffer. This could be viewed as an additional nested stride on the type, except that it is not known until the Pack/Unpack call and therefore is not included in the type optimization. Therefore, the kernels all handle this value dynamically.

Each kernel dimension is filled from X to Z by the largest power of two that encompasses the structure described above, ultimately limited by a block limit of 1024 threads. The grid is sized to cover the entire input object once the block size is chosen.

By the end of this whole process, each MPI datatype is mapped to one of two kernel implementations parameterized by $W$. No object metadata is stored on the GPU - the kernel parameters are scalar values from the StridedBlock object.

4 LIBRARY ARCHITECTURE

The Topology Experiments for MPI library is designed to make MPI modifications available to research and production code without relying on updates to the system MPI implementation. For reference, Fig. 5 shows a compiled MPI application and the TEMPI interposer. The application source file includes the system MPI header file and is compiled into a dynamic library using the system MPI header. The application is compiled as normal except for TEMPI being inserted into the link order, or an unmodified application can be used with the LD_PRELOAD mechanism. When the application is executed, any symbols defined by the TEMPI library will be resolved there, allowing the TEMPI code to be executed. Any others will be resolved in the system implementation.

A transparent transformation from non-contiguous application data to contiguous data provided to MPI introduces some engineering challenges not discussed in detail in this work. Generally, the performance modeling (described later) as well as CUDA APIs to provide streams, pinned host buffers, devices buffers and events introduce too much latency. TEMPI uses a caching layer to accelerate repeated requests for the same resources, which is common in iterative applications. This allows otherwise expensive resources and decisions to be provided tens or hundreds of nanoseconds amortized time, instead of microseconds to milliseconds.

5 MODELING MPI PRIMITIVES WITH DATATYPE ACCELERATION

The interposer design requires that interprocess communication is handled by the underlying system MPI. Therefore, integration of datatype handling with underlying communication is restricted to packing and unpacking non-contiguous data into contiguous buffers, upon which system MPI primitives are invoked.

In the “device” packing method (Eq. 1), the strided object is packed from the original GPU buffer into an intermediate GPU buffer, then transferred to an intermediate buffer on the destination GPU with CUDA-aware MPI_Send, then unpacked into the strided destination object (Eq. 2).

$$T_{device} = T_{gpu-pack} + T_{gpu-gpu} + T_{gpu-unpack}$$
In the “one-shot” packing method (\(T_{\text{onest}}\), Eq. 2), the strided object was packed from the original GPU buffer into intermediate mapped CPU buffer (\(T_{\text{host} \rightarrow \text{pack}}\)), transferred to an intermediate mapped buffer at the destination (\(T_{\text{cpu} \rightarrow \text{cpu}}\)), then unpacked directly into GPU memory (\(T_{\text{host} \rightarrow \text{unpack}}\)).

\[
T_{\text{onest}} = T_{\text{host} \rightarrow \text{pack}} + T_{\text{cpu} \rightarrow \text{cpu}} + T_{\text{host} \rightarrow \text{unpack}}
\] (2)

Finally, in the “staged” method (\(T_{\text{stage}}\), Eq. 3) matches the device method, except the intermediate CPU buffer is transferred to a pinned buffer on the host (\(T_{\text{host} \rightarrow \text{pack}}\)), where it is transferred to the destination rank’s CPU before being copied to the destination GPU (\(T_{\text{cpu} \rightarrow \text{cpu}}\)). This method would only be faster than the device method if \(T_{\text{cpu} \rightarrow \text{cpu}} < T_{\text{host} \rightarrow \text{pack}}\).

\[
T_{\text{stage}} = T_{\text{gpu} \rightarrow \text{pack}} + T_{\text{cpu} \rightarrow \text{cpu}} + T_{\text{cpu} \rightarrow \text{cpu}} + T_{\text{gpu} \rightarrow \text{pack}}
\] (3)

Wang et al. [17] introduces the one-shot and staged methods (using cudaMempy2DAsync instead of GPU kernels). They find that the staged method is preferable to one-shot. On the contrary the other works described in Section 7 prefer the one-shot method with various GPU kernels.

Modeling the performance of each is a challenge in its own right. Inter-node message latency (\(T_{\text{cpu} \rightarrow \text{cpu}}\)) is commonly modeled as a latency term plus a bandwidth term [1], possibly refined into short, eager, and rendezvous regimes. Inter-node GPU message latency (\(T_{\text{gpu} \rightarrow \text{gpu}}\)) further complicates the model with GPU-CPU bandwidth, GPU control latency, direct communication between GPU and NIC (Nvidia’s “GPUDirect”), and pipelining of large messages [5]. When datatype descriptions are involved, there is additional complexity regarding efficiency of non-contiguous memory accesses served through device memory (\(T_{\text{gpu} \rightarrow \text{pack}}\)) or over the CPU-GPU interconnect (\(T_{\text{cpu} \rightarrow \text{pack}}\)). The interposer design places TEMPI at the mercy of the performance characteristics of the underlying system, so this work sidesteps these concerns by measuring the relevant performance directly and using them at runtime to choose the packing method (Section 6.3).

6 RESULTS
The experiments are carried out on three MPI implementations spanning two hardware platforms summarized in Table 1. All multi-node performance is evaluated using Spectrum MPI on OLCF Summit. For non-communication operations, performance is also evaluated on the single-node openmpi and mvapich platforms. The mvapich platform does not use MVAPICH-GDR, which integrates some prior work by Chu et al. [4, 5], but requires [12] Mellanox networking hardware and drivers despite their irrelevance to datatype handling.

| Name       | OLCF Summit | openmpi  | mvapich |
|------------|-------------|----------|---------|
| MPI        | Spectrum MPI 10.3.1.2 | OpenMPI 4.0.5 | MVAPICH 2.3.4 |
| CPU        | IBM POWER 9 | AMD Ryzen 7 3700x | NVIDIA V100 |
| GPU        | NVIDIA V100 | NVIDIA GTX 1070 | NVIDIA V100 |
| nvcc       | 11.0.221    | 11.1.105  | 11.0.221 |
| gcc        | 9.3.0       | 10.2      | 9.3.0    |
| GPU Driver | 418.116.00  | 455.32.00 | 418.116.00 |

Figure 6: Time for MPI derived datatype creation and commit time for equivalent 3D objects described with subarray (1), hvector of vector (2), hvector of hvector of vector (2,3), and subarray of vector (4). The “create” component shows MPI_Type and MPI_Type_create family of MPI APIs to describe the type. The “commit” component is how much time is consumed in MPI_Type_commit. The trimean of 30000 executions of each phase is reported. Create time is unchanged (TEMPI does nothing) and is reported for comparison. In MPI_Type_commit, TEMPI does the same operations regardless of the MPI implementation, but performance varies due to performance of the calls that provide information about MPI types. TEMPI slows commit time substantially, but it still has a negligible impact on application run time.

6.1 MPI_Type_commit
The type transformation and kernel selection process is executed when the application calls MPI_Type_commit. Fig. 6 shows the run-time impact of creating MPI derived types, broken down into two phases. Creation refers to using the MPI_Type and MPI_Type_create family of MPI APIs to assemble the type description. Commit refers to calling MPI_Type_commit on that description. TEMPI does the same operations in each instance; however, it relies on the performance of the MPI_Type_get_envelope, MPI_Type_get_extent, MPI_Type_size, and MPI_Type_get_contents functions. The different implementations will have different performance for those routines, and therefore the “commit” component takes variable amounts of time. Within a particular implementation, different type configurations have different commit times as a different sequence of optimizations is run to arrive at the canonical form. Overall, the transformation and kernel selection process slows down the create+commit process by 2.1× to 5.5× vs. mvapich, 3.5× to 6.8× vs. openmpi, and 4.2× to 11.6× vs. Summit. This slowdown is a one-time cost during program startup and is small in magnitude.

6.2 MPI_Pack and MPI_Unpack
Once a type has been committed, it can be used in a communication routine. The simplest examination of such a routine is MPI_Pack, where a buffer is “sent” into another buffer in the same process. When GPU buffers are passed to the inbuf and outbuf parameters of MPI_Pack/Unpack, TEMPI uses the selected GPU kernel to complete the packing.

Fig. 7 shows the pack bandwidth achieved for various 2D objects, described as a vector or subarray datatype. Spectrum MPI 10.3.1.2, MVAPICH 2.3.4 and OpenMPI 4.0.5 all support a baseline derived datatype handling approach where each
contiguous portion of the derived datatype is copied into a contiguous buffer through cudaMemcpyAsync (or similar function). This approach becomes faster as the contiguous block is longer (amortizing overhead), and slower with more contiguous blocks comprise the datatype. MVAPICH also features optimized handling through specialized packing kernels for certain datatypes. TEMPI achieves a speedup of over 242,000 on Summit for the largest datatype. TEMPI nearly matches MVAPICH’s performance for the single vector type (“vec / 1 KiB / 1 / 8”), but generalizes that high performance to an equivalent subarray (“sub / 1 KiB / 1 / 8”) and multiple vector types (“vec / 1 KiB / 2 / 8”). Across the experiment speedup varies from 0.98× to 242,000×. Generally TEMPI performs comparatively better when the contiguous regions are smaller or the total data size is larger. In the first case, more memory copies are replaced by a single kernel, and in the second case, the GPU resources are better utilized by the kernel.

Fig. 8 shows the fragile MVPAICH performance in more detail. MVAPICH is fast for vectors, but slow for equivalent subarray types. Similarly, MVAPICH’s fast vector handling is disabled when multiple objects are packed at once. TEMPI features equivalent high performance in all scenarios.

6.3 MPI_Send and MPI_Recv

MPI allows datatypes to be directly used with MPI_Send and other communication routines. Section 5 introduced three ways of building MPI communication primitives with fast datatype support when the system MPI does not have it. This model identifies when the one-shot or the staged methods are preferable. All experiments are limited to the Summit platform, the only evaluation platform with multiple GPUs and multiple nodes.

The performance model is analyzed with measured data of various primitives.

- \( T_{cpu-cpu} \) : MPI_Send/MPI_Recv on CPU buffer
- \( T_{gpu-gpu} \) : MPI_Send/MPI_Recv on GPU buffer
- \( T_{2gh} \) : cudaMemcpyAsync from device (GPU) to host (CPU) and cudaMemcpyAsync from host to device and cudaMemcpyAsync from host to device and cudaMemcpyAsync

The MPI operations are measured through a ping-pong between two ranks, and the reported time is half of the total ping-pong time. The two ranks are on separate nodes. The CUDA operations are recorded using wall-time around the first and last calls, which reflect when control leaves and returns to the application.

Fig. 9a shows the results of the four operations for various data sizes. CUDA-aware MPI transfers show a latency floor of approximately 6 µs, compared to 1.3 µs transfers from pinned system memory.

Fig. 9b shows the measurements in Eqs. 1, 2, 3 while holding \( T_{gpu-pack/unpack} \) and \( T_{cpu-pack/unpack} \) to zero (i.e., \( T_{oneshot} = T_{cpu-cpu} \) and \( T_{device} = T_{gpu-gpu} \)). There is no region where \( T_{staged} \) is faster than \( T_{device} \), and it will be disregarded for further discussion. Whether \( T_{device} \) or \( T_{oneshot} \) is faster will depend on the relative pack/unpack performance of the two methods. As \( T_{device} \) has pack/unpack occur in the faster device memory it may be faster than \( T_{oneshot} \) for various transfer sizes.

To complete the model, Fig. 10 shows the measured latency of pack and unpack operations for “one-shot” (\( T_{cpu-pack}, T_{cpu-unpack} \)) and “device” (\( T_{gpu-pack}, T_{gpu-unpack} \)). The recorded time includes all of the operations described in Section 3.3, i.e. selecting appropriate grid dimensions, executing the kernel, and synchronizing after execution.

Pack/unpack latency depends on both the object size and the size of the contiguous blocks in the object. Larger objects are faster as GPU resources are more fully utilized. Larger contiguous blocks tend to be faster as accesses become more coalesced and make better use of memory and interconnect transactions. One-shot performance is maximized at 32 B contiguous blocks and in-device performance at 128 B. The unpack operation is slower than the pack.
due to non-contiguous writes (instead of non-contiguous reads in the pack operation).

Therefore, whether $T_{\text{oneshot}}$ or $T_{\text{device}}$ is faster depends on both the object size and the length of the contiguous blocks that make up that object. Qualitatively, the one-shot method is faster when objects are smaller, as the packing kernels are limited by launch and synchronization overhead and the CPU-CPU transfers are faster than GPU-GPU. It is also faster when objects are more contiguous, where the zero-copy accesses over the interconnect make good use of the interconnect bandwidth.

When a communication primitive is called, TEMPI uses the object size and parameters to query the performance model. TEMPI provides a binary that records system performance parameters to the file system. This binary should be run once before TEMPI is used in an application. Performance measurements are sparse by necessity. $T_{\text{cpu-cpu}}$ and $T_{\text{gpu-gpu}}$ are estimated through 1D interpolation of the object size, while $T_{\text{cpu-pack}}$, $T_{\text{cpu-unpack}}$, $T_{\text{gpu-pack}}$, and $T_{\text{gpu-unpack}}$ from a 2D interpolation from the stride and block length of the datatype. These modeling functions are “pure”, and their results are cached so that future invocations using the same parameters do not require a redundant expensive interpolation.

Fig. 11 shows the application-visible performance of MPI_Send/MPITP_Send and MPI_Recv compared to the baseline Spectrum MPI 10.3.1.2. Fig. 11a shows that the vast majority of the speedup comes from the datatype handling ("baseline" vs. "one-shot"/"device"). Since $T_{\text{oneshot}}$ or $T_{\text{device}}$ may be faster depending on the arguments passed to MPI_Send, TEMPI uses the performance model and system measurements to estimate which method will have lower latency. Fig. 11b shows that the automatic model-based selection is accurate enough to reliably choose the faster of the one-shot or device methods. In the 1 KiB object some small model slowdown is observed as TEMPI must dynamically query the performance model to make its method selection.

This slowdown is present at all sizes, but not as visible at the larger object sizes. Over these tests, model selection added 277 ns of latency. The latency floor is around 30 µs, of which 26 µs can be directly attributed to the pack/unpack kernels on the sending and receiving side. The rest of time is consumed by looking up the cached datatype handler and checking to see if the user-provided buffers are GPU-resident. Speedup between the baseline and TEMPI’s automatic selection is up to 59000× for large objects with small blocks.

### 6.4 Case Study: 3D Stencil

The enormous datatype handling performance has a commensurate impact on application performance. Here we consider a 3D stencil code, where the total stencil region is a cube of $512^3 \times P$ gridpoints and $P$ is the number of ranks. Each gridpoint is a single four-byte value, and the stencil radius is 2. Each halo region is defined in a separate MPI derived datatype and packed into the single buffer using MPI_Pack. Halo exchange is implemented as an MPI_Alltoall on that single buffer. Then, the receive buffer is unpacked. The stencil kernel is a standard 26 point, yielding 26 neighbors for each rank with periodic domain boundaries. This means each rank engages in 26 MPI_Pack and 26 MPI_Unpack operations on a variety of different 3D strided datatypes. Reported times are the maximum time consumed for each phase across all ranks.

Fig. 12 shows TEMPI’s halo exchange latency on Summit, broken down into pack, MPI_Alltoall, and unpack, as well as the overall halo exchange speedup. The different pack and unpack latencies reflects their different kernel performance (Section 6.2). The speedup is smallest for larger number of ranks, as the communication takes up a relatively larger amount of the total iteration time compared to the pack and unpack operations. Speedup at 192 ranks is 1050×.

### 7 RELATED WORK

TEMPI distinguishes itself from prior work in three ways. First, TEMPI can be used today without waiting for MPI implementers or HPC system administrators. Second, while prior work uses GPUs kernels to accelerate datatype operations, TEMPI is the first work that shows transformations on structured datatypes for canonicalization (as opposed to handling specific cases, or reducing everything to a list of offsets and lengths). This provides wide datatype coverage, tiny GPU memory consumption for metadata, and fast generic kernels. Third, prior work examines how to integrate data type handling into MPI more deeply. As TEMPI uses a library-interposer interface on top of MPI, it is not able to make those low-level changes. Despite that, enormous performance improvement is possible.

The MPITypes library [14] is one of the first attempts to generalize datatype handling outside of MPI. It provides several functions for flattening and copying datatypes, and a framework for extending those operations. TEMPI tries to maintain the structured information of types so the MPITypes operations are not directly applicable.

Wang et al. [17] describes an early approach in MVAPICH2. Several options are considered, ultimately selecting a pipelined version of the “staged” method that uses cudaMemcpy2D instead of...
Figure 10: Pack/unpack latency using the “one-shot” and “device” strategies for 64 B - 4 MiB objects. For smaller contiguous regions, performance is reduced due to low memory or interconnect efficiency for non-coalesced accesses. For larger objects, performance increases as GPU resources are better utilized.

(a) One-shot pack.  (b) One-shot unpack.  (c) Device pack.  (d) Device unpack.

Figure 11: Time for an MPI_Send/MPI_Recv pair for 1KiB, 1MiB, and 4MiB 2D objects with contiguous blocks of various sizes. “Baseline” is the Summit platform without TEMPI. Each group of bars is labeled with the contiguous block size.

(a) MPI_Send / MPI_Recv latency for the one-shot, device, model-based automatic selection, and Summit MPI baseline. The vast majority of the performance improvement comes from the datatype handling, before the one-shot or device method is selected.

(b) Normalized latency of the one-shot, device, and model-based selection based on the measured system parameters. The model-based automatic selection reliably chooses the faster method with minimal overhead.

Figure 12: Performance of 3D stencil halo exchange operation using TEMPI vs. SpectrumMPI 10.3.1.2.

(a) Performance of phases of halo exchange. Alltoally time is higher for more ranks and more nodes. Pack/Unpack time is constant, as the amount of data moved per rank is unchanged.

(b) Speedup of entire halo exchange. Speedup is lower for larger number of ranks as datatype handling is a smaller portion of the total time.
no optimizations are present for nested datatypes, or operations on more than one datatype.

Jenkins et al. [10] provide fast handling of arbitrary MPI datatypes on the GPU. Nested types are represented by a tree structure that must be traversed by each GPU thread using division, modulo, and binary search operations. They restrict inter-node communication to the one-shot method, which this work finds is not always preferable.

Shi et al. [15] also explicitly breaks the problem into transformation and kernel selection phase. Hand defines specific kernels for handling vector, hvector, subarray, and indexed block types. For other datatypes, it transforms a variety of datatypes into a blocklist, for which it has a specific kernel implementation. Instead, TEMPI recognizes that nested, strided types reduce to (essentially) a subarray, and explicitly designs a transformation and optimal packing kernel to cover all of those scenarios.

Wei et al. [18] describe a fork of OpenMPI that integrates derived datatype handling both on the GPU itself as well as communication between nodes. The datatype is ultimately represented as a list of blocks, and blocks are partitioned among separate kernels with pipelined communication. It also identifies that full GPU resources for handling non-contiguous data are not needed to saturate the communication links. This fork has remained unmerged and not publicly available.

Chu et al. [4] recognize that one of the challenges of all prior work is the latency of kernel launches. Like prior work, it also represents the datatypes as a list of displacements and lengths. Similar to this work, it defines extraction, conversion, and caching steps, and uses one-shot packing and unpacking. Unlike TEMPI, they do not recognize when the one-shot packing to the host is slower to inefficiency of packing and unpacking over the interconnect. Chu et al. [5] identify that a major cost of transfer is the launch of the packing kernels. They develop an engine that is able to merge various packing requests into a single kernel launch. TEMPI addresses the packing kernel launch cost by issuing a single kernel for multiple copies of the same MPI datatype, but cannot fuse further than that. These two works appear to have been integrated into MVAPICH2-GDR. MVAPICH does not include handling for generic datatypes, but does have more primitive handling of specific datatypes.

Hashmi et al. [8] describe a zero-copy-based data movement system for MPI datatypes. They include kernels where a warp is responsible for a contiguous block in a block list. They also describe a variety of integrations with the underlying communication library, which TEMPI is unable to address due to its interposer model.

8 FUTURE WORK

This work only addresses the MPI_Send primitive. TEMPI implements datatype handling for MPI_Isend as well and the performance impact is broadly the same. MPI_Isend introduces additional modeling and measurement challenges. Bienz et al. [2] have introduced some performance modeling techniques that could be extended to evaluate the impact of improved datatype handling in that context.

Similarly, this work does not address MPI collectives. An initial implementation would transparently pack the datatypes into a contiguous buffer on each rank before invoking an underlying collective. Due to the vast improvement in datatype handling performance, the authors expect a similarly large improvement in MPI collectives.

The interposer library model hinders deeper modification of MPI primitives. For example, pipelining the pack operation through a sequence of kernel calls and messages. Consequently, the performance model does not address such a scenario.

The proposed type analysis infrastructure cannot handle MPI indexed types. It may be the case that applications sometimes use indexed types to describe strided types, in which case an analysis at MPI_Type_commit time could traverse the indxed type and recover the strided nature. MPI struct provides an additional challenge, as it can package multiple child datatypes together. For these types, it may ultimately be better to select a more general strategy introduced in previous work.

Current approaches favor kernels for data handling over using the DMA engine, e.g. cudaMemcpy2D. These functions have more restrictions on data alignment, but may be useful when they apply. They do not occupy GPU SM resources, and therefore would have a smaller impact on application performance.

While the techniques in this paper are demonstrated in the context of the TEMPI interposer library for maximum compatibility, they can be introduced into any existing MPI implementation. TEMPI is open-source and provided under the permissive Boost software license.

9 CONCLUSION

Despite years of deployment of CUDA-aware MPI systems alongside research contributions for GPU datatype handling, MPI derived datatype handling is not generally available. This work presents a library-interposer approach for deploying new code into an existing MPI implementation without modifying an application. That approach is used to integrate novel GPU derived datatype handling into OpenMPI, MVAPICH, and Spectrum MPI. This approach transforms datatypes describing common objects into a compact canonical form, backed by generic processing kernels. The work is evaluated by injecting the new datatype handling code into a deployed MPI implementation on a leadership-class supercomputer. MPI_Pack performance on datatypes was sped up by up to 242,000×, MPI_Send up to 59,000× and a 3D stencil halo exchange up to 1000× at 192 ranks.

ACKNOWLEDGMENTS

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy Contract No. DE-AC05-00OR22725. This work utilizes resources supported by the National Science Foundation’s Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign. The authors would also like to thank Dawei Mu, Omer Anjum, and Mert Hidayetoglu.
REFERENCES

[1] Amotz Bar-Noy and Shlomo Kipnis. 1992. Designing broadcasting algorithms in the postal model for message-passing systems. In Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures. 13–22. https://doi.org/10.1145/144991.144993

[2] Amanda Bienz, William D. Gropp, and Luke N. Olson. 2018. Improving Performance Models for Irregular Point-to-Point Communication. In Proceedings of the 25th European MPI Users’ Group Meeting (Barcelona, Spain) (EuroMPI’18). Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3236367.3236368

[3] Amanda Bienz, Luke N. Olson, William D. Gropp, and Shelby Lockhart. 2020. Modeling Data Movement Performance on Heterogeneous Architectures. arXiv:2010.10378 [cs.DC]

[4] C. Chu, J. M. Hashmi, K. S. Khorassani, H. Subramoni, and D. K. Panda. 2019. High-Performance Adaptive MPI Derived Datatype Communication for Modern Multi-GPU Systems. In 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC). 267–276. https://doi.org/10.1109/HiPC.2019.00041

[5] C. H. Chu, K. S. Khorassani, Q. Zhou, H. Subramoni, and D. K. Panda. 2020. Dynamic Kernel Fusion for Bulk Non-contiguous Data Transfer on GPU Clusters. In 2020 IEEE International Conference on Cluster Computing (CLUSTER). 130–141. https://doi.org/10.1109/CLUSTER49012.2020.00023

[6] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. 2006. Open MPI: A Flexible High-Performance MPI. In Parallel Processing and Applied Mathematics, Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–239.

[7] William Gropp. 2002. MPICH2: A New Start for MPI Implementations. In Proceedings of the 9th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer-Verlag, Berlin, Heidelberg, 7.

[8] Jahangir Mozbood Hashmi, Ching-Hsiang Chu, Sourav Chakraborty, Mohamadreza Bayatpour, Hari Subramoni, and Dhabaleswar K. Panda. 2020. FALCON-X: Zero-copy MPI derived datatype processing on modern CPU and GPU architectures. J. Parallel and Distrib. Comput. (2020).

[9] IBM. 2016. IBM Spectrum MPI Version 10 Release 1 User’s Guide. Technical Report.

[10] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur. 2014. Processing MPI Derived Datatypes on Non-contiguous GPU-Resident Data. IEEE Transactions on Parallel and Distributed Systems 25, 10 (2014), 2627–2637. https://doi.org/10.1109/TPDS.2013.234

[11] mvapich. 2020. MVAPICH2 Changelog. http://mvapich.cse.ohio-state.edu/static/media/mvapich/MV2_CHANGELOG-2.3.4.txt

[12] MVAPICH. 2020. MVAPICH20GDR 2.3.5 - Technical Report.

[13] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohamadreza Bayatpour. 2020. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. Journal of Computational Science (2020), 101208. https://doi.org/10.1016/j.jocs.2020.101208

[14] Robert Ross, Robert Latham, William Gropp, Ewing Lusk, and Rajeev Thakur. 2009. Processing MPI Datatypes Outside MPI. In Recent Advances in Parallel Virtual Machine and Message Passing Interface. Matti Ropo, Jan Westerholm, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 42–53.

[15] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, and D. K. Panda. 2014. HAND: A Hybrid Approach to Accelerate Non-contiguous Data Movement Using MPI Datatypes on GPU Clusters. In 2014 43rd International Conference on Parallel Processing. 221–230. https://doi.org/10.1109/ICPP.2014.31

[16] MPI standards committee. 2015. MPI: A Message-Passing Interface Standard Version 3.1 - Technical Report.

[17] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda. 2011. Optimized Non-contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2. In 2011 IEEE International Conference on Cluster Computing. 308–316. https://doi.org/10.1109/CLUSTER.2011.42

[18] Wei Wu, George Bosilca, Rolf van de Vaart, Sylvain Jeaugey, and Jack Dongarra. 2016. GPU-Aware Non-Contiguous Data Movement in Open MPI. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (Kyoto, Japan) (HPDC ’16). Association for Computing Machinery, New York, NY, USA, 231–242. https://doi.org/10.1145/2907294.2907317