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. This work first presents a novel datatype handling strategy for nested strided datatypes, which finds a middle ground between the specialized or generic handling in prior work. This work also shows that the performance characteristics of non-contiguous data handling can be modeled with empirical system measurements, and used to transparently improve MPI_Send/Recv latency. Finally, despite substantial attention to non-contiguous GPU data and CUDA-aware MPI implementations, good performance cannot be taken for granted. This work demonstrates its contributions through an MPI interposer library, TEMPI. TEMPI can be used with existing MPI deployments without system or application changes. 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 917× in a 3D halo exchange with 3072 processes.

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**.

KEYWORDS

MPI, CUDA, derived datatype, Summit, Spectrum MPI

ACM Reference Format:
Carl Pearson, Kun Wu, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. 2021. TEMPI: An Interposed MPI Library with a Canonical Representation of CUDA-aware Datatypes. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431379.3460645

1 INTRODUCTION

MPI Derived Datatypes [16] are an 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 [12], 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-process transfers of GPU-resident data.

Previous works have created specialized strategies for specific datatypes [15, 17], 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 this substantial attention to non-contiguous data and broad GPU deployment in high-performance distributed computing, fast handling of GPU datatypes may not be available in practice.

This work makes three contributions

- A novel strategy for converting nested strided MPI datatypes to a compact, canonical representation to enable fast GPU handling (Sec. 3).
- Low-overhead runtime selection of non-contiguous packing strategy for MPI_Send and MPI_Recv according to empirical system properties (Sec. 4).
- A portable implementation based off an MPI interposer library, tested with MVAPICH, Spectrum MPI, and Open MPI on POWER and x86 platforms, and evaluated in-depth against Spectrum MPI on Summit (Sec. 5).

ACM ISBN 978-1-4503-8217-5/21/06…$15.00
https://doi.org/10.1145/3431379.3460645

Primarily, this work 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 improvements include specialization for types with certain kinds of regularity [15, 17] or sophisticated strategies for 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 canonicalization phase generates a 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.

This work also 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 - depending on the properties of the non-contiguous data and the system, it may be preferable to pack data into GPU memory and use the underlying CUDA-aware mechanism for interprocess transfer. The model is evaluated in the context of the OLCF Summit system.

Finally, though CUDA-aware MPI implementations have varying degrees of fast non-contiguous data handling, such support cannot be taken for granted for a particular datatype or platform. The Summit computer at the Oak Ridge Leadership Computing Facility is one such system: it's Spectrum MPI implementation offers functional but extremely slow performance for most MPI datatypes. The final contribution of this work is to demonstrate that a portable interposed library can transparently deliver large derived-datatype performance improvements without system or 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 via a run-time algorithmic selection before it is passed to the underlying MPI implementation. TEMPI's interposer design makes it compatible with widely-deployed MPI implementations, but as a consequence it relies on the performance of the contiguous transfer primitives of the underlying MPI implementation. TEMPI is available at https://github.com/cwpearson/tempi.

TEMPI demonstrates a speedup of up to 242,000× for MPI_Pack and MPI_Unpack, 59,000× for MPI_Send, and 20,000× 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 5 describes the library-interposer method that makes the derived type modifications available without application modification. Section 4 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 consideres the following "strided" datatypes due to their applicability to stencil codes:

- "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*): *newtype* is 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_create_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 types may be composed in many ways to describe the same non-contiguous bytes. For example, consider the 3D object in Figure 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.

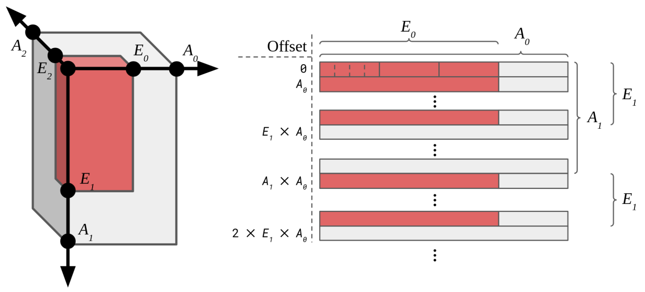


Figure 1: A 3D object with extent $E_0 \times E_1 \times E_2$ floats in an allocation $A_0 \times A_1 \times A_2$ bytes, and the corresponding linearized memory layout.

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_contiguous($E_0 \times 4$, MPI_BYTE, &row): "row" is $E_0 \times 4$ 1-byte elements.
- MPI_Type_vector(1, E_0, 1, MPI_FLOAT, &row)
- MPI_Type_vector(E_0, 4, 4, MPI_BYTE, &row)
- MPI_Type_create_hvector($E_0 \times 4$, 1, 1, MPI_BYTE, &row)
- MPI_Type_create_subarray(1, $\{A_0\}$, $\{E_0\}$, $\{0\}$, MPI_ORDER_C, MPI_FLOAT, &row)
- MPI_Type_create_subarray(1, $\{A_0 \times 4\}$, $\{E_0 \times 4\}$, $\{0\}$, 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:

- MPI_Type_vector(E_1, E_0, A_0, MPI_FLOAT, &plane)
- MPI_Type_vector(E_1, $E_0 \times 4$, A_0, MPI_BYTE, &plane)
- MPI_Type_create_subarray(2, $\{A_0, A_1\}$, $\{E_0, E_1\}$, $\{0, 0\}$, MPI_ORDER_C, MPI_FLOAT, &plane)
- MPI_Type_create_subarray(2, $\{A_0{\times}4, A_1\}$, $\{E_0{\times}4, E_1\}$, $\{0, 0\}$, MPI_ORDER_C, MPI_BYTE, &plane)

or alternatively, as an hvector of rows:

- MPI_Type_create_hvector(E_1, 1, A_0, row, &plane)

or for the subarray row types:

- MPI_Type_vector(E_1, 1, 1, row, &plane)
- MPI_Type_create_subarray(1, A_1, E_1, 0, MPI_ORDER_C, row, &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,

- MPI_Type_create_hvector(E_2, 1, $A_0 \times A_1$, plane, &cuboid)
- MPI_Type_create_subarray(2, $\{A_0, A_1, A_2\}$, $\{E_0, E_1, E_2\}$, $\{0, 0, 0\}$, MPI_ORDER_C, MPI_FLOAT, &cuboid)
- MPI_Type_create_subarray(2, $\{A_0{\times}4, A_1, A_2\}$, $\{E_0{\times}4, E_1, E_2\}$, $\{0, 0, 0\}$, MPI_ORDER_C, MPI_BYTE, &cuboid)

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

Consider such a representation of N non-contiguous blocks of M MPI_FLOATs. To support objects dispersed across large address ranges, the block offset and size would each be 8 bytes (64 bits) 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.

This work recognizes that structured data in many MPI applications do 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 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/transformation/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 a fully specified MPI derived datatype into a *Type* hierarchy, which represents a (possibly non-contiguous) set of bytes from a memory region. Each *Type* level has a field *data* of *TypeData*, which represents information about the level. Each *Type* also tracks zero or one child *Type* levels. The *Type* hierarchy and its children describe the MPI datatype, where the order of the hierarchy matches the hierarchy 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

Type translation is accomplished by converting each MPI datatype to a corresponding *DenseData* or *StreamData* node, and then recursively doing the same to its child before attaching them to the converted node. The recursive base case is when an MPI Named type is reached, which by definition has 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) is a special case of *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.

An MPI vector (MPI_Type_vector) or hvector (MPI_Type_create_hvector) are translated into two nested *StreamData*, a “parent” and “child”. The parent represents the repeated blocks, and the child the repeated elements within each block. Both offsets are 0. The child count is the vector blocklength, and the child stride is the extent of *oldtype*. The parent count is the vector count, and the parent stride is the child *stride* times the vector stride. For hvector the parent *stride* is given directly in the hvector *stride* argument and does not need to be computed.

An MPI subarray (MPI_Type_create_subarray) is a set of nested *StreamData* equivalent to the dimension of the subarray. MPI subarray arguments are provided inner-to-outer, which corresponds to a descendant-ancestor relationship in the *Type* tree. The count of dimension i is provided by the corresponding subarray *subsize*. The stride of dimension i is the product of the MPI extent of the subarray *oldtype* and the $i - 1$ preceding subarray sizes. The offset of each dimension is given in terms of elements and is converted to bytes for the *TypeData*.

3.2 Type Canonicalization

The construction of the *Type* hierarchy described in Section 3.1 yields a hierarchy of *StreamData* with a base of *DenseData*. Since each level of the datatype has a direct correspondence in the *Type* hierarchy, semantically equivalent datatypes may have different *Types*. In order to provide fast handling of equivalent types, these various representations are canonicalized.

Four 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. “Stream flattening” combines two *StreamData* that could be represented as one. “Sorting” ensures the *StreamData* have a unique order. The optimizations are applied repeatedly in turn, only terminating when neither optimization would modify the *Type* hierarchy. Algorithm 1 summarizes the overall simplification process.

3.2.1 *Dense Folding.*

Dense folding 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: simplification

```
Function simplify(ty):
    simplified ⟵ ty
    changed ⟵ TRUE
    while changed do
        changed ⟵ FALSE ∨ dense_folding (simplified)      ▹ in-place
        changed ⟵ changed ∨ stream_elision (simplified)   ▹ in-place
        changed ⟵ changed ∨ stream_flatten (simplified)   ▹ in-place
        changed ⟵ changed ∨ sort (simplified)             ▹ in-place
    end
    return ty
```

Algorithm 2 shows how the transformation is applied to a *Type*, and Figure 3 shows the transformation graphically. 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.

Algorithm 2: dense_folding from Alg. 1

```
Function dense_folding(ty):
    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
        changed ⟵ TRUE
        cData.off ⟵ cData.off + pData.off
        cData.extent ⟵ pData.count × pData.stride
        ty ⟵ child                           ▹ replace ty with child
    end
    return changed
```

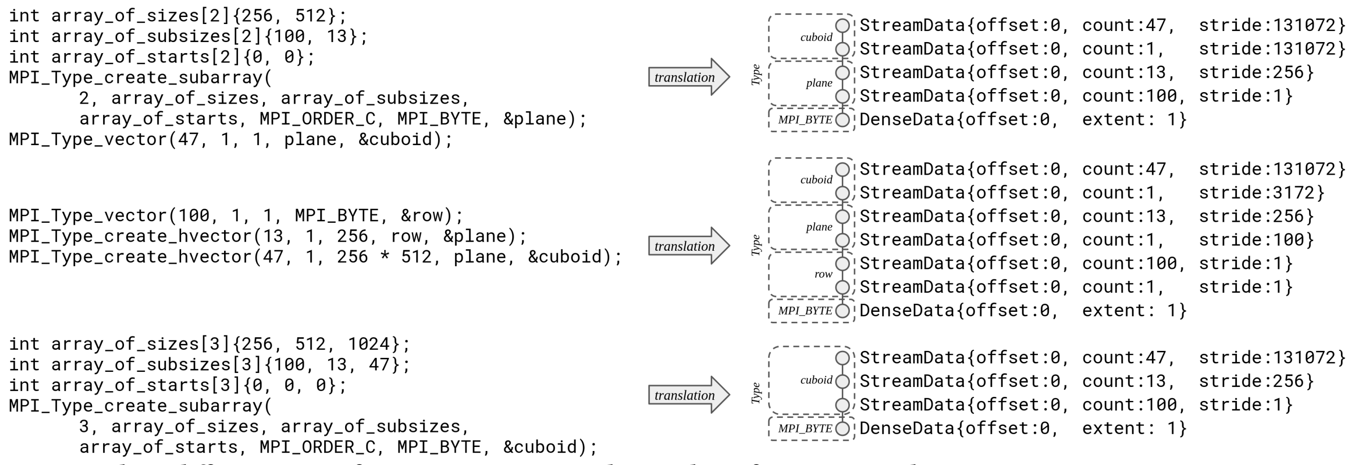
3.2.2 *Stream Elision.*

Stream elision canonicalizes a case where a stream has only a single element. 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* 1 dimension with *subsize* 1.

Algorithm 3 shows how the transformation is applied to a *Type*, and Figure 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.

3.2.3 *Stream Flattening.*

Stream flattening canonicalizes the case where a pair of nested streams could be represented as a single stream. Consider a child *StreamData* with a count A and a stride B. If the parent *StreamData* has a stride that is a product of A and



```
int array_of_sizes[2]{256, 512};
int array_of_subsizes[2]{100, 13};
int array_of_starts[2]{0, 0};
MPI_Type_create_subarray(
      2, array_of_sizes, array_of_subsizes,
      array_of_starts, MPI_ORDER_C, MPI_BYTE, &plane);
MPI_Type_vector(47, 1, 1, plane, &cuboid);
```

```
StreamData{offset:0, count:47,  stride:131072}
StreamData{offset:0, count:1,   stride:131072}
StreamData{offset:0, count:13,  stride:256}
StreamData{offset:0, count:100, stride:1}
DenseData{offset:0,  extent: 1}
```

```
MPI_Type_vector(100, 1, 1, MPI_BYTE, &row);
MPI_Type_create_hvector(13, 1, 256, row, &plane);
MPI_Type_create_hvector(47, 1, 256 * 512, plane, &cuboid);
```

```
StreamData{offset:0, count:47,  stride:131072}
StreamData{offset:0, count:1,   stride:3172}
StreamData{offset:0, count:13,  stride:256}
StreamData{offset:0, count:1,   stride:100}
StreamData{offset:0, count:100, stride:1}
StreamData{offset:0, count:1,   stride:1}
DenseData{offset:0,  extent: 1}
```

```
int array_of_sizes[3]{256, 512, 1024};
int array_of_subsizes[3]{100, 13, 47};
int array_of_starts[3]{0, 0, 0};
MPI_Type_create_subarray(
      3, array_of_sizes, array_of_subsizes,
      array_of_starts, MPI_ORDER_C, MPI_BYTE, &cuboid);
```

```
StreamData{offset:0, count:47,  stride:131072}
StreamData{offset:0, count:13,  stride:256}
StreamData{offset:0, count:100, stride:1}
DenseData{offset:0,  extent: 1}
```

Figure 2: Three different MPI C fragments to generate the 3D object from Fig. 1 with $A_0 = 256$, $A_1 = 512$, $A_2 = 1024$, $E_0 = 100$, $E_1 = 13$, and $E_2 = 47$. The right-hand side shows the corresponding *Type* IR after translation, with parent *TypeData* above child *TypeData*. Equivalent objects can be represented differently and require a later transformation pass.

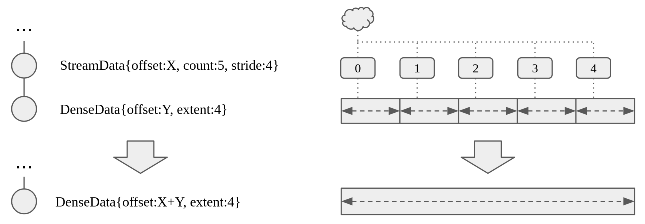

StreamData{offset:X, count:5, stride:4}
DenseData{offset:Y, extent:4}
DenseData{offset:X+Y, extent:4}
0
1
2
3
4

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.

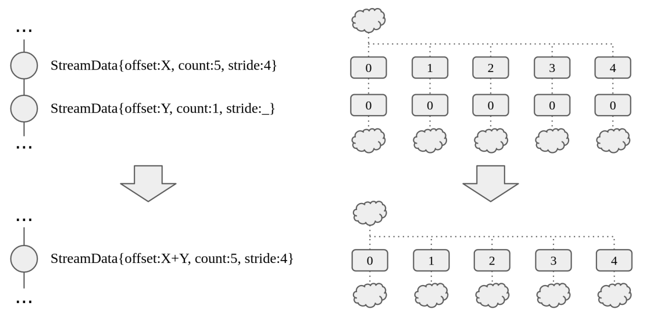

StreamData{offset:X, count:5, stride:4}
StreamData{offset:Y, count:1, stride:_}
StreamData{offset:X+Y, count:5, stride:4}
0
1
2
3
4

Figure 4: Example of stream elision. When a child *StreamData* has only a single element, it can be removed from the *Type* tree.

B, that means multiple children are separated by the child's stride. In such a case, the parent and child can be flattened into a single *StreamData* with a larger count.

Algorithm 4 shows how the transformation is applied to a *Type*, and Figure 5 shows an example. This operation has some overlap with stream elision. Stream elision handles the specific case when the child's count is 1, which lifts the restriction on the child and parent stride relationship in stream flattening.

Algorithm 3: stream_elision from Alg. 1

```
Function stream_elision(ty):
    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 1 == cData.count then
        changed ← TRUE
        ty.child ← child.children                         ▷ delete child
    end
    return changed
```

3.2.4 Sorting. *Sorting* canonicalizes the ordering of a pair of nested streams is arbitrary. For example, consider a 2D non-contiguous object. That object could be constructed as columns of rows of blocks, or rows of columns of blocks. To canonicalize this case, the *StreamData* hierarchy is sorted by stride, with the largest strides first in the hierarchy and the smaller strides last.

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 list *counts*: number of elements in the dimension
  - integer list *strides*: bytes between the start of each element in the dimension

The *start* field describes the offset of the first byte in the object from the beginning of the allocation. The *i*th entry of *counts*

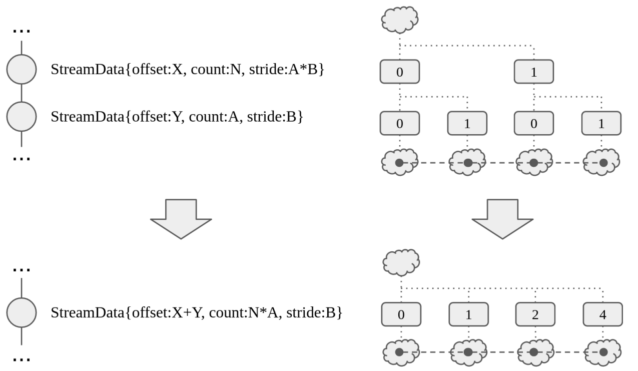

StreamData{offset:X, count:N, stride:A*B}
StreamData{offset:Y, count:A, stride:B}
StreamData{offset:X+Y, count:N*A, stride:B}
0
1
0
1
0
1
0
1
2
4

Figure 5: Example of stream flattening. When the parent stride allows repeated children to maintain a fixed stride between their elements, the parent and child can be flattened into a single stream.

Algorithm 4: stream_flatten from Alg. 1

```
Function stream_flatten(ty):
    changed ⟵ FALSE
    for child of ty do
        changed ⟵ changed ∨ stream_flatten(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 pData ⟵ ty.data
    StreamData cData ⟵ child.data
    if pData.stride == cData.count × cData.stride then
        changed ⟵ TRUE
        pData.count ⟵ pData.count × cData.count
        pData.stride ⟵ cData.stride
        pData.off ⟵ pData.off + cData.off
        ty.child ⟵ child.children                          ▷ delete child
    end
    return changed
```

and *strides* describes the number of repetitions of the previous dimension and the number of bytes separating each repetition, respectively.

Algorithm 5 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 ensures that structure, 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.

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 cudaStreamSynchronize. 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

Algorithm 5: conversion of *Type* to *StridedBlock*

```
Function strided_block(ty):
    datas ⟵ []
    cur ⟵ ty                                  ▷ Add all TypeData to an array
    while true do
        datas.append(cur)
        if cur.child == {} then
            break                             ▷ no children left
        else
            cur ⟵ cur.child
        end
    end
    StridedBlock sb                           ▷ to be returned
    for i = 0 to datas.size() do
        if i == 0 then
            if data is DenseData then
                sb.off ⟵ data.off
                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.off ⟵ sb.off + data.off
                sb.counts.append(data.count)
                sb.strides.append(data.stride)
            else
                return NULL                   ▷ Not strided
            end
        end
    end
    return sb
```

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 dimension is filled from X to Z by the smallest power of two that encompasses the corresponding extent, ultimately limited by a block limit of 1024 threads. The grid is then sized to cover the entire input object once the block size is determined.

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.

Many MPI functions that operate on datatypes accept a *count*, *incount*, or *outcount* parameter, describing how many objects are to be operated on in the buffer. Unlike other properties of the type, this value is not known until the MPI function is called and therefore is not included in the type optimization. The kernels handle this value dynamically either by increasing the grid Z dimension (for 2D), or by applying the entire kernel grid to each object in turn (3D).

By the end of this whole process, each MPI datatype has a corresponding kernel implementation with a specific W instantiation. No metadata is consumed in GPU memory - all object parameters are either encoded into the kernel binary (W) or passed as a scalar kernel argument.

4 MODELING MPI PRIMITIVES WITH DATATYPE ACCELERATION

The interposer design (Section 5) 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 (T_{device}, Eq. 1), the strided object is packed from the original GPU buffer into an intermediate GPU buffer ($T_{gpu-pack}$), then transferred to an intermediate buffer on the destination GPU with CUDA-aware MPI_Send/MPI_Recv ($T_{gpu-gpu}$), then unpacked into the strided destination object ($T_{gpu-unpack}$).

$$T_{device} = T_{gpu-pack} + T_{gpu-gpu} + T_{gpu-unpack} \tag{1}$$

In the "one-shot" packing method ($T_{oneshot}$, Eq. 2), the strided object is packed from the original GPU buffer into intermediate mapped CPU buffer ($T_{host-pack}$), transferred to an intermediate mapped buffer at the destination ($T_{cpu-cpu}$), then unpacked directly into GPU memory ($T_{host-unpack}$).

$$T_{oneshot} = T_{host-pack} + T_{cpu-cpu} + T_{host-unpack} \tag{2}$$

Finally, in the "staged" method (T_{staged}, Eq. 3) matches the device method, except the intermediate GPU buffer is transferred to a pinned buffer on the host (T_{h2d}), where it is transferred to the destination rank's CPU before being copied to the destination GPU (T_{h2d}). This method would only be faster than the device method if $T_{cpu-cpu} + T_{h2d} + T_{d2h} < T_{gpu-gpu}$.

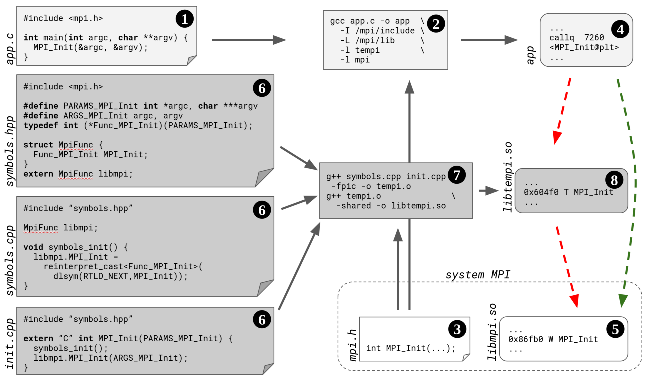
$$T_{staged} = T_{gpu-pack} + T_{d2h} + T_{cpu-cpu} + T_{h2d} + T_{gpu-unpack} \tag{3}$$

Wang et al. [17] introduces the one-shot and staged methods (using cudaMemcpy2DAsync instead of GPU kernels). They find that the staged method is preferable to one-shot. In contrast, 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 ($T_{cpu-cpu}$) 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 ($T_{gpu-gpu}$) 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 [3]. When datatypes are involved, there is additional complexity regarding efficiency of non-contiguous memory accesses served through device memory ($T_{gpu-pack}$) or over the CPU-GPU interconnect ($T_{cpu-pack}$). 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).

5 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,



```
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
}
```

```
gcc app.c -o app  \
  -I /mpi/include \
  -L /mpi/lib     \
  -l tempi        \
  -l mpi
```

```
...
callq  7260
<MPI_Init@plt>
...
```

```
#include <mpi.h>

#define PARAMS_MPI_Init int *argc, char ***argv
#define ARGS_MPI_Init argc, argv
typedef int (*Func_MPI_Init)(PARAMS_MPI_Init);

struct MpiFunc {
  Func_MPI_Init MPI_Init;
}
extern MpiFunc libmpi;
```

```
#include "symbols.hpp"

MpiFunc libmpi;

void symbols_init() {
  libmpi.MPI_Init =
    reinterpret_cast<Func_MPI_Init>(
      dlsym(RTLD_NEXT,MPI_Init));
}
```

```
#include "symbols.hpp"

extern "C" int MPI_Init(PARAMS_MPI_Init) {
  symbols_init();
  libmpi.MPI_Init(ARGS_MPI_Init);
}
```

```
g++ symbols.cpp init.cpp
 -fpic -o tempi.o
g++ tempi.o                \
  -shared -o libtempi.so
```

```
...
0x604f0 T MPI_Init
...
```

```
int MPI_Init(...);
```

```
...
0x86fb0 W MPI_Init
...
```

Figure 6: The application source file (❶) includes the MPI header file provided by the system (❸). It is compiled (❷) and linked with the system MPI implementation. When the binary (❹) is executed, symbols are resolved and the MPI code from the system MPI library is executed. The TEMPI source files (❻) are compiled (❼) into a dynamic library (❽) using the system MPI header. The application is compiled as normal except for TEMPI being inserted into the link order (❷), 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 (❽), allowing the TEMPI code to be executed. Any others will be resolved in the system implementation.

Fig. 6 shows a compiled MPI application (❶-❺) and the TEMPI interposer (❻-❽). The application source (❶) includes the system MPI headers (❹) and is compiled (❷) to produce a binary (❸). At run time, the operating system will resolve the symbols in the application binary according to the order of linked libraries, and MPI_Init is found in the system MPI implementation (❺).

TEMPI provides new MPI functionality for unmodified applications by exporting a partial implementation for the MPI interface. For example, init.cpp (❻) implements the MPI_Init function. The TEMPI source includes the system MPI header, and must be compiled (❼) with the same MPI as the target application so that the ABI matches. If the original application can be recompiled, the TEMPI library (❽) may be inserted into the link order before the system MPI library (❷). If not, the TEMPI library can be injected using LD_PRELOAD or similar mechanism (not shown).

Either way, the operating system will search for the MPI_Init symbol in the TEMPI library. As it is found there, that function will be called instead of the system MPI. Internally, TEMPI may ultimately call some system MPI function after introducing its own functionality. This is achieved through the dlsym function. Any parts of the MPI interface that TEMPI does not cover will fall back to the system MPI library automatically.

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.

6 RESULTS

The experiments are carried out on the OLCF Summit platform, using Spectrum MPI 10.3.1.2. Summit nodes have two IBM POWER 9 CPUS and six Nvidia V100 GPUs connected in two quadruplets by 100 GB/s bidirectional Nvlink 2 interconnects. Experiments were performed with NVCC 11.0.221, GCC 9.3.0, and GPU driver 418.116.00.

6.1 MPI_Type_commit

The type transformation and kernel selection process is executed when the application calls MPI_Type_commit. Fig. 7 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* functions to assemble the type description. Commit refers to calling MPI_Type_commit on that description. Different type configurations have different commit times as a different sequence of transformations is required to arrive at the canonical form. Overall, the transformation and kernel selection process slows down the create+commit process by 3.8× to 8.3× on Summit. This slowdown is a one-time cost during program startup and is small in magnitude.

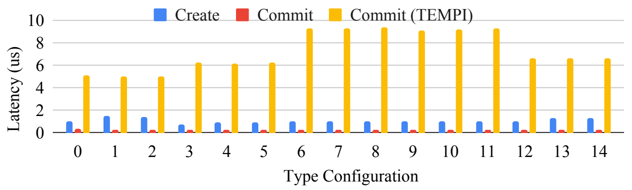


Figure 7: Time for MPI derived datatype creation and commit time for a variety of 3D objects described with subarray (0-2), hvector of vector (3-5), hvector of hvector of vector (6-11), and subarray of vector (12-14). The "create" component uses MPI_Type* and MPI_Type_create* family of MPI APIs to describe 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. TEMPI's performance varies due to different required canonicalization operations. TEMPI slows commit time substantially, but it still has a negligible impact on application run time.

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. 8 shows the pack bandwidth achieved for various 2D objects, described as a vector or subarray datatype.

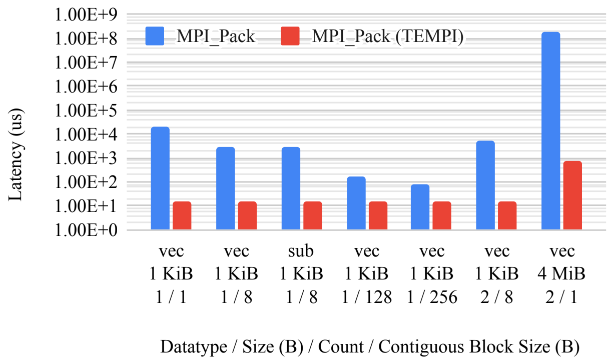


Figure 8: MPI_Pack performance for a variety of 2D objects described as a vector or subarray datatype. "Size" is the total object size, "count" is the number of objects packed, and "contiugous block size" is the number of contiguous bytes in each block of the object. The pitch of each contiguous block is 512 B. TEMPI provides fast handling regardless of MPI datatype construction or datatype count.

Spectrum MPI 10.3.1.2 provides 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. TEMPI delivers speedup of over 242,000 on Summit for the largest datatype. Across the experiments speedup varies from 5.7× to 242,000×. Generally TEMPI performs comparatively better when the contiguous regions are smaller or the total data 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. Regardless of datatype count, the MPI datatype used to provide the description, the size of the non-contiguous block, or the total size of the data, TEMPI is able to transparently provide enormous speedup.

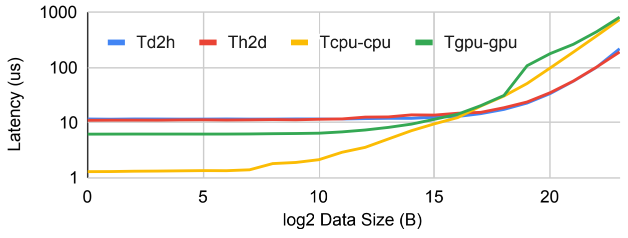
6.3 MPI_Send and MPI_Recv

MPI allows datatypes to be directly used with MPI_Send and other communication routines. Section 4 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.

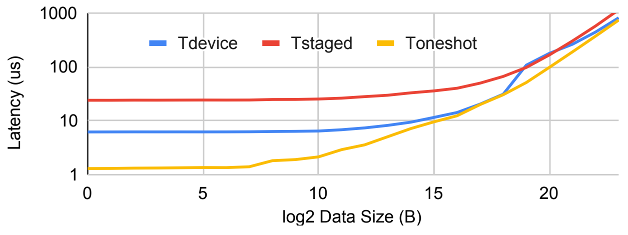
The performance model is analyzed with measured data of various primitives.

- $T_{cpu-cpu}$: MPI_Send/MPI_Recv on CPU buffer
- $T_{gpu-gpu}$: MPI_Send/MPI_Recv on GPU buffer
- T_{d2h}: cudaMemcpyAsync from device (GPU) to host (CPU) and cudaStreamSynchronize
- T_{h2d}: cudaMemcpyAsync from host to device and cudaStreamSynchronize

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.



(a) Measurements of T_{d2h}, T_{h2d}, $T_{cpu-cpu}$, and $T_{gpu-gpu}$ on Summit.



(b) Partial values of T_{device}, $T_{oneshot}$, and T_{staged}, (excluding pack time), using the values from (a).

Figure 9: Raw measurements and partial performance models (omit pack/unpack) for various data transfer methods on OLCF Summit.

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 from 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 that $T_{oneshot}$ 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 $T_{oneshot}$ or T_{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.

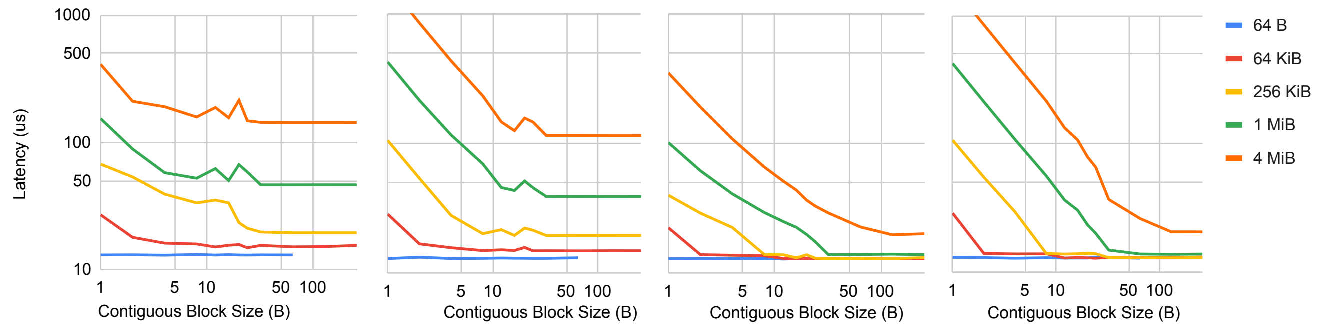
When a communication primitive is called, TEMPI uses the object size and parameters to query the performance model. TEMPI provides a binary that records system performance parameters to the file system. This binary should be run once before TEMPI is used in an application. Performance measurements are sparse by necessity. $T_{cpu-cpu}$ and $T_{gpu-gpu}$ are estimated through 1D interpolation of the object size, while $T_{cpu-pack}$, $T_{cpu-unpack}$, $T_{gpu-pack}$, and $T_{gpu-unpack}$ 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 to not require a redundant expensive interpolation.

Fig. 11 shows the application-visible performance of MPI_Send / 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 $T_{oneshot}$ or T_{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.

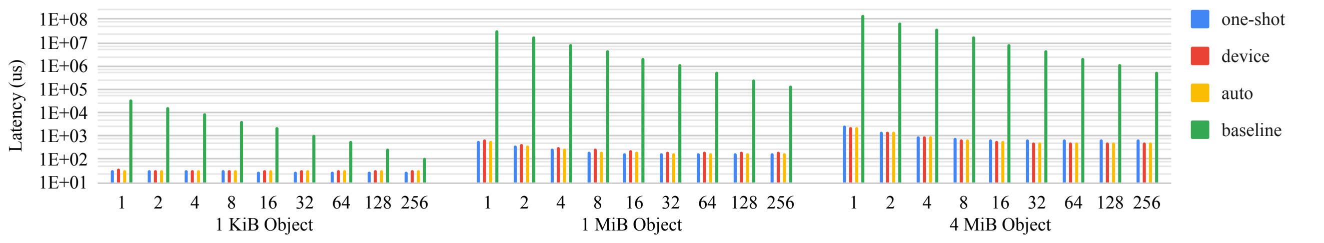
6.4 Case Study: 3D Stencil

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 $256^3 \times P$ gridpoints and P is the number of ranks. Each gridpoint has eight eoght-byte value, and the stencil radius is 3. This implementation is chosen to replicate the communication requirements of the Astaroth stellar simulation code [13]. 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_Alltoallv 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.

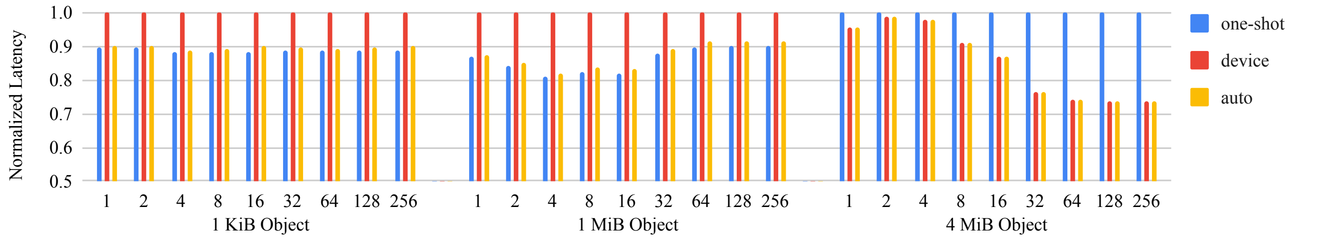


(a) One-shot pack. (b) One-shot unpack. (c) Device pack. (d) Device unpack.

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) 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.

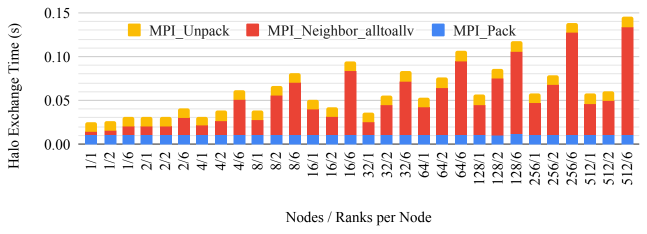
Figure 11: Time for an MPI_Send/MPI_Recv pair for 1KiB, 1MiB, and 4MiB 2D objects with contiguous blocks of various sizes. "Baseline" is the Summit platform without TEMPI. Each group of bars is labeled with the contiguous block size.

Fig. 12 shows TEMPI's halo exchange latency on Summit, broken down into pack, MPI_Alltoallv, 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×.

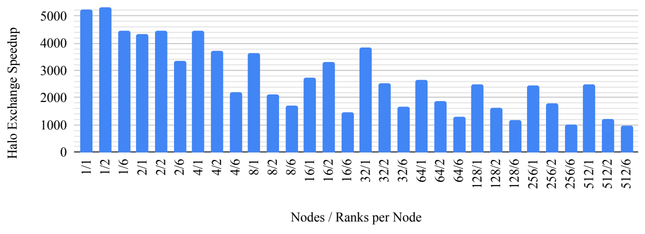
7 RELATED WORK

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 GPU 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, including pipelining and zero-copy transfer between GPUs. 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 was achieved.

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



(a) Performance of phases of halo exchange. Alltoallv 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.

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

Table 1: Selected microbenchmark results from related work. Provided for reference, as hardware changes hinder direct performance comparison.

Work	Platform	Non-contiguous Operation		Nominal Bandwidth Slowdown		
		Pack	Dis. Mem. Ping-Pong	Network	GPU Mem.	GPU Interconnect
[17]	C2050, QDR IB	25us (1KiB), 10ms (4MiB)	20ms (4MiB)	3.125	6.25	6.25
[15]	C2050, QDR IB	120us (1KiB)	*(none provided)*	3.125	6.25	6.25
[10]	C2050, QDR IB	10us (1 KiB)	70us (1KiB), 700us (256KiB)	3.125	6.25	6.25
[18]	K40, FDR IB	75us (512KiB), 150us (4 MiB)	7ms (4 MiB)	1.8	3.125	3.175
This	V100, EDR IB	13us (64 KiB), 21us (4 MiB)	60us (1KiB), 354us (1MiB), 888us (4MiB)	1	1	1

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 a kernel.

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.

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 integration with the underlying communication library, which TEMPI is unable to address due to its interposer model.

Yaksa [11] is a high-performance non-contiguous datatype engine being developed for MPICH. Like this work, it features an internal representation of non-contiguous data derived from MPI datatypes. Custom GPU kernels are synthesized for each Yaksa datatype that corresponds to the familiar MPI datatypes. When datatypes are nested, the result is traversed and pack operations are issued separately for each subtree for which there is a synthesized kernel.

Not all prior works include microbenchmarks comparable to the ones presented herein. Table 1 summarizes those that do. Substantial hardware changes make a direct comparison difficult, so normalized bandwidth numbers for various subsystems are provided for reference. TEMPI is competitive for both latency- and bandwidth-bound operations (small and large amounts of data, respectively).

8 FUTURE WORK

Several other MPI implementations, including MVAPICH (and especially MVAPICH-GDR), Open MPI, and MPICH, have varying degrees of support for fast GPU datatype handling. As these implementations are not available on Summit, a direct comparison was not possible. However, TEMPI's approach is complementary to what is adopted by those implementations. TEMPI could be used to prototype improvements or extensions to the currently-implemented datatype handling, just as it was for Spectrum MPI. Furthermore, prior work that has not been integrated with an existing MPI implementation could use TEMPI to evaluate and deployment to a large variety of existing systems and codes. TEMPI could also be extended to use an approach from prior works to handle indexed datatypes (including MPI_Type_struct) on Summit, or to evaluate the use of the GPU DMA engine for non-contiguous data (e.g. cudaMemcpy2D).

The performance modeling and runtime implementation choice is currently limited to MPI_Send and MPI_Recv pairs. Furthermore, the interposer library model hinders deeper modification of MPI primitives, such as pipelining packing and interprocess communication. Even simple asynchronous operations (MPI_Isend) introduce additional modeling and measurement challenges, as many communications can be active simultaneously. Similarly, this work does not address MPI collectives. 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.

9 CONCLUSION

Despite years of deployment of CUDA-aware MPI systems alongside research contributions for GPU datatype handling, fast MPI derived datatype handling is not available on all platforms. This work used a library-interposer approach for deploying fast datatype handling on OLCF Summit without requiring modification to the system or applications. It presented a novel approach for transforming datatypes describing common objects into a compact canonical form, backed by generic processing kernels. Empirical system measurements are used at run-time to optimize the packing method. 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 917× at 3072 processes. TEMPI is available at https://github.com/cwpearson/tempi.

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 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/10.1145/140901.140903

[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] Jahanzeb Maqbool Hashmi, Ching-Hsiang Chu, Sourav Chakraborty, Mohammadreza 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 Noncontiguous GPU-Resident Data. *IEEE Transactions on Parallel and Distributed Systems* 25, 10 (2014), 2627–2637. https://doi.org/10.1109/TPDS.2013.234

[11] Programming Models and Runtime Systems Team. 2021. *Yaksa.* https://github.com/pmodels/yaksa

[12] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza 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

[13] Johannes Pekkilä, Miikka S. Väisälä, Maarit J. Käpylä, Petri J. Käpylä, and Omer Anjum. 2017. Methods for compressible fluid simulation on GPUs using high-order finite differences. *Computer Physics Communications* 217 (2017), 11–22. https://doi.org/10.1016/j.cpc.2017.03.011

[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 vandeVaart, 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