MPI Derived Datatypes: Performance and Portability Issues

Qingqing Xiong
Boston University
Boston, MA
qx@bu.edu

Purushotham V. Bangalore
University of Alabama at Birmingham
Birmingham, AL
puri@uab.edu

Anthony Skjellum
University of Tennessee at Chattanooga
Chattanooga, TN
tony-skjellum@utc.edu

Martin Herbordt
Boston University
Boston, MA
herbordt@bu.edu

ABSTRACT

This paper addresses performance-portability and overall performance issues when derived datatypes are used with four MPI implementations: Open MPI, MPICH, MVAPICH2, and Intel MPI. These comparisons are particularly relevant today since most vendor implementations are now based on Open MPI or MPICH rather than on vendor proprietary code as was more prevalent in the past. Our findings are that, within a single MPI implementation, there are significant differences in performance as a function of it reasonable encodings of derived datatypes as supported by the MPI standard. While this finding may not be surprising, it is important to understand how fundamental vs. arbitrary choices made in early implementation impact the use of derived datatypes to date.

A more significant finding is that one cannot reliably choose a single derived datatype format and expect uniform performance portability among these four implementations. That is, the best-performing path under one of the MPI code bases is not the same as the best-performing path under another. Users have to be prepared to recode for a different formulation to move efficiently among MPICH, MVAPICH2, Intel MPI, and Open MPI. This lack of uniformity presents a significant gap in MPI's fundamental purpose of offering performance portability.

Specific examination of internal implementation details indicates why performance is different among the implementations. Proposed solutions to this problem include i) revamping datatypes; ii) providing a common, underlying datatype standard used by multiple MPI implementations; and iii) exploring new ways to describe derived datatypes that are optimizable by modern networks and faster than MPI implementations' software-based marshaling and unmarshaling.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**;

EuroMPI '18, September 23–26, 2018, Barcelona, Spain

ACM ISBN 978-1-4503-6492-8/18/09…$15.00
https://doi.org/10.1145/3236367.3236378

KEYWORDS

data marshaling, performance-portability, communication performance, derived datatypes, MPI

ACM Reference Format:
Qingqing Xiong, Purushotham V. Bangalore, Anthony Skjellum, and Martin Herbordt. 2018. MPI Derived Datatypes: Performance and Portability Issues. In *25th European MPI Users' Group Meeting (EuroMPI '18), September 23–26, 2018, Barcelona, Spain.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3236367.3236378

1 INTRODUCTION

HPC operating environments are becoming more and more diverse. One HPC system can be equipped with one or multiple MPI implementations, and different implementations can be developed for diverse goals. For each system, the underlying communication protocols (e.g., InfiniBand, Ethernet) supported by MPI are also varied. The design space is further expanded by the freedom of description provided by an MPI implementation; specifically, for derived datatypes, one memory layout can be represented with various derived datatype formats. All these differences not only indicate the importance and popularity of MPI but also bring potential performance issues.

One issue is with performance-portability: the performance of the same derived datatype might not be portable to different MPI implementations, or between different pairs of processes in a single implementation depending on system topology and data-transfer paths. Another issue is with derived datatype normalization: the same layout represented with different derived datatypes may or may not have the same performance. These hidden performance behaviors might affect the overall performance of an application greatly, especially when moving code from one implementation to another, when the programmer might need to recode to get the performance improvement out of the optimized code. That obstacle can lead application programmers to do all their own data marshaling, which, in high-quality implementations and where persistent operations are used, could be done more efficiently by the implementation.

In this paper, we perform a series of tests on two HPC systems, with various derived datatypes using MPICH, Open MPI, MVAPICH2, and Intel MPI under various communication protocols. We explore performance-portability and normalization issues under such a design space and show that the performance of the same

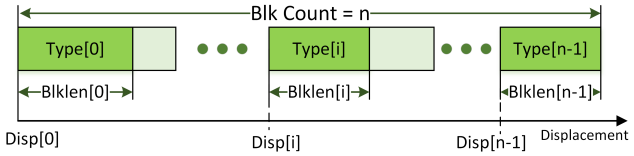


Figure 1: A memory layout can be represented as n blocks (Blk Count), where the i^{th} block has a length stored in Blklen[i]. Each element of the displacement array represents the offset of the corresponding block to the starting address of this memory layout.

derived datatype can vary by up to 3x between different MPI implementations. We specifically look into MPICH and Open MPI source code to determine possible causes of the performance gap between them and provide suggestions to improve performance of the "slower" one. On top of the suggestions, we demonstrate a vectorization approach to provide further performance improvement.

The performance gains one can obtain from software optimization is limited. Hardware offload that takes care of datatypes explicitly offers another level of potential optimization and can potentially relieve CPUs from datatype processing. As adding FGPAs to HPC clusters has become a growing trend, we revise hardware approaches and propose a prototype FPGA offload design to resolve the performance issues of derived datatypes.

The remainder of this paper is organized as follows. First, we cover background on derived datatypes (§ 2). Next, we discuss our approach to understanding derived datatypes' performance (§ 3). Then, we describe our experimental setup, the benchmark and case study we used, and results and analysis (§ 4). We close with our suggested approaches to solving performance issues of derived datatypes (§ 5) and offer conclusions (§ 7).

2 BACKGROUND ON MPI DERIVED DATATYPES

In this section, we first give an abstract introduction about how derived datatypes work and how they are used. Then, we outline the performance and portability issues that we focus on.

2.1 How Derived Datatypes Work and How They Are Used

MPI provides derived datatypes to describe local memory layouts concisely, particularly when the memory layout is complex and non-contiguous[1]. With derived datatypes, programmers can avoid tedious packing and unpacking of data using the `MPI_Pack()` and `MPI_Unpack()` APIs or manual copying. MPI communication functions take derived datatypes as arguments and enable data packing and unpacking by interacting with the communication stack closely; this arrangement exposes the potential performance benefits gained from segment-and-pipeline, hardware offload, etc.

[1] Note that there are five major MPI functions for constructing derived datatypes [5]. These assume that any memory layout can be represented by many blocks (shown in Fig. 1) and a displacement array, a block length array, and a datatype array. When the described memory layout has some level of regularity, any/all of the arrays can be a constant value/constant values (the displacement array will be regular). Different derived datatype constructors provide different levels of freedom for describing such a memory layout.

As a particular example, `MPI_Type_contiguous` represents an array of elements with built-in or predefined datatypes located contiguously in the memory. It is usually encapsulated by other datatypes such as `MPI_Type_vector`, which is used to describe a series of blocks of equal size and spaced by a constant stride (e.g., to represent a column of a row-major matrix). To describe irregular memory layouts where the block length is a constant but the displacements are irregular, `MPI_Type_indexed`, `MPI_create_struct`, or `MPI_Type_create_indexed_block` is used. `MPI_Type_create_resized` can be used to attach extra extend space to a base datatype. There are also several other functions for creating datatypes that describe subarrays and distributed arrays.

Communication functions such as `MPI_Send()` take committed derived datatypes, the starting address of the data, and the count as the arguments and perform data packing before the data goes to the network. Data unpacking is performed after data arrives from the network. These packing and unpacking procedures are also called *data marshaling and demarshaling*.

2.2 Problem Statement: Performance and Portability Issues

The performance of derived datatypes has drawn a lot of attention [22, 23]. Various approaches have been adopted in MPI implementations for improving derived datatype transfer performance; however, based on our results and some other recent profiling results [3], we find that the same derived datatype might perform differently across MPI implementations. The same memory layout can be represented using many derived datatypes but with totally different performance behavior, which comprises the derived datatype normalization problem. These behaviors might have complex causes, including that different implementations focus on different aspects of performance issues (e.g., MVAPICH2 exploits NIC functionalities). All the causes together lead to the derived datatype performance gap between implementations.

With the reliance of most MPI implementations on a few MPI open source implementations, performance-portability as well as absolute MPI application performance is threatened. In this paper, we address this issue while calling implementers' attention to further resolving the performance and portability issues of other MPI APIs. By resolving these problems in key open source implementations, leverage into several commercial implementations can be achieved over time as well, particularly if open source MPI exposes interfaces internally to support better hardware offload for forthcoming system architectures.

3 APPROACH TO PERFORMANCE UNDERSTANDING

What is the performance of derived datatypes? The answer is nontrivial for two reasons: (i) an MPI implementation is usually multi-layered from the user API level to the abstraction and use of the underlying communication protocol, and (ii) MPI derived datatypes, including construction and marshaling, might be implemented in multiple layers of an MPI implementation or even in the hardware (NIC), making it difficult to measure derived datatype performance separately. Therefore, looking at different parts of the performance

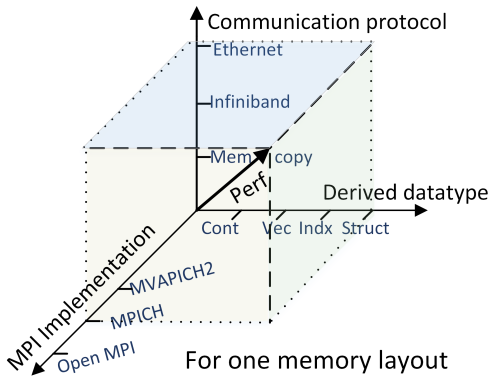


Figure 2: The multi-dimensional approach to studying MPI derived datatypes' performance. For one specific memory layout, depending on the communication protocol, the MPI implementation, and the derived datatype format, the performance can vary and can be represented as the shown vector (Perf). We remove the dimension of different memory layout for a clear view.

and different levels of the implementation will net different answers. In this paper, we take all of these considerations into account by defining derived datatype performance as the performance of `MPI_Xfer` (data transfer functions, e.g., `MPI_Send()`) or MPI I/O functions that use derived datatypes. Here, we specifically look at the case with `MPI_Xfer` calls.

3.1 Multi-Dimensional MPI Derived Datatype Performance Study

Our study of MPI derived datatype performance is multi-dimensional (as depicted in Fig. 2). Communication protocols and memory layouts in applications are always diverse, so we performed a "design space exploration" of four dimensions: the communication protocol between ranks, different derived datatypes to represent one memory layout, different MPI implementations, and different types of memory layouts. We then studied the performance under such assumptions.

To understand the performance of derived datatypes, their behavior across implementations, and different underlying interconnection between MPI processes (ranks in `MPI_COMM_WORLD`), we performed a series of experiments with various datatypes using MPICH, Open MPI, Intel MPI, and MVAPICH2. We explored performance portability across implementations when representing different memory layouts, or the performance of different MPI implementations when representing the same memory layout with the same constructors. We built on the benchmark concept offered by Carpen-Amarie et al. [3] to create four derived datatypes for four memory layouts systematically, and then we performed ping-pong sends and receives (blocking calls) to test each datatype.

Benchmarks alone, however, only give a partial picture and must be substantially augmented with other evaluation modalities. To see the behavior of datatypes in real-world use cases, we selected the 4D matrix edge transfer case used widely in stencil computations. For the specific memory layout, we looked at different derived datatype formats' performance across MPI implementations. Since one memory layout can be represented with one or more derived datatypes, each datatype might have different performance. To normalize the datatype representations is one goal of many researchers; therefore, it is worthwhile to consider the performance of each of the representations when using different APIs. Here, we are more interested in the performance portability issue, so we focused on analyzing the performance among MPI implementations; however, our tests could still reflect the normalization issue of derived datatypes.

The underlying communication protocol can also be different across platforms. For instance, when an MPI application only runs on one node, the communication protocol between MPI processes can be memory-copy-based; when an application runs across multiple nodes, the protocol can be over InfiniBand or TCP/IP on Ethernet, etc. Some HPC systems are even equipped with multiple types of protocols. We should not assume that the same implementation would have the same behavior with different underlying communication protocols. Therefore, we studied the derived datatypes with various underlying communication protocols and observed the resulting performance outcomes.

4 EXPERIMENTS PERFORMED AND ANALYSIS

In this section, we discuss our experimental setup, the benchmarks we created to test the performance portability of MPI libraries, and the 4D matrix edge transfer use case. For each of the tests, we analyze the results and offer our observations.

4.1 Experimental Setup

We experimented on two systems. The first is the Boston University Shared Computing Cluster (SCC) [24]. The SCC system currently includes over 14,500 CPU cores and over 4.2 petabytes of storage for research data. We ran our tests on a subset of the SCC. There are two networks: one is 1Gbps Ethernet, and the other is via a central QDR InfiniBand switch. Each node has 48GB of memory and two sockets, each with a six-core 3.07 GHz Intel(R) Xeon(R) X5675 processor. We experimented with three MPI implementations: MPICH-3.2, Open MPI-1.6.4, and MVAPICH2-2.2, which are all compiled with gcc 4.4.7 -O2.

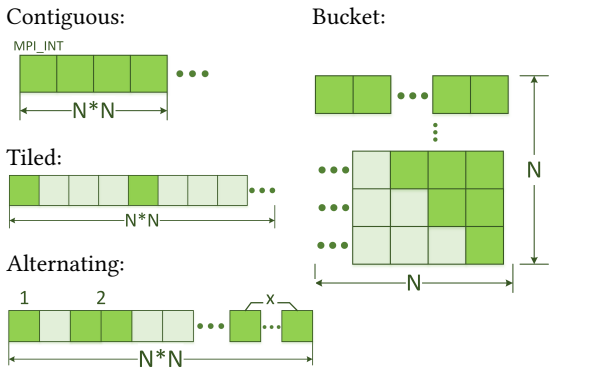
The second system (Cheaha) is equipped with the Mellanox ConnectX-5 InfiniBand adapter cards [25]. The CPU on Cheaha nodes is Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz (14 cores) with 256GB RAM connected with ConnectX-5 EDR InfiniBand cards to a 36-port EDR switch. There are three MPI implementations on this system: Open MPI-2.1.2 compiled with gcc-6.4.0 -O2, MVAPICH2-2.2b compiled with gcc-4.9.3 -O2, and Intel MPI Library 2017 Update 1 built with gcc 6.3.0.

We will now demonstrate the results on both systems.

4.2 Benchmarks and Case Study

We explored the multi-dimensional design space exploration with two tests: one benchmark test and one case study. The benchmark uses several memory layouts, and the case study demonstrates one memory layout. The two tests can support each other interactively. We show the two tests along with the result analysis in this section.

Table 1: Derived datatypes configurations.



4.2.1 Benchmark Performance Portability Across MPI Implementations. We used a "ping-pong" benchmark to compare the performance of three MPI implementations when describing the same memory layout in the same way. The message size varies at the outer loop. For each message size, the ping-pong blocking send and receive between two ranks happens 1,000 times. We show the average time per iteration in our results.

We instrumented four types of memory layouts shown in Table 1, naming each layout based on Carpen-Amarie et al. [3]. Carpen-Amarie et al. strategically choose five categories of memory layouts depending on the regularity of two characteristic arrays, the displacement and block length arrays. Taking the type array into account, there can be more than five categories of memory layouts. We used the same idea to categorize and name memory layouts, but we chose four categories and generalize the creation by avoiding nested derived datatypes. We also created memory layouts based on real-world cases.

In Table 1, *Tile* represents regular displacement (constant stride) and constant block length; we set the stride to 4 (the unit is MPI_INT) and block length to 1 as the case of sending one column of a 2D array where the row width is 4. *Bucket* represents the case where the block length varies in blocks and the stride is constant; we use this memory layout to demonstrate the upper triangle of a 2D matrix. *Alternating* represents a memory layout with irregular displacements and block lengths. We configured it based on the one described by Wu et al. [27]: the block length varies from one integer to x integers. The gap (empty blocks in the plot) between two blocks equals the size of the first block. The N in Table 1 varies from 10 to 500.

Contiguous and *Tile* are constructed using `MPI_Type_contiguous` and `MPI_Type_vector`, respectively. *Bucket* and *Alternating* are both constructed using `MPI_Type_indexed`. We focus here on comparing the same derived datatype in different MPI implementations. A datatype's configuration, such as the memory footprint, is not necessarily the same for all types.

In this benchmark, the communication calls used by two ranks are `MPI_Send()` and `MPI_Recv()`. We measured the average ping-pong time between the two ranks out of the 5,000 iterations (1,000 iterations per run), running first on SCC nodes. We mapped two ranks on the same CPU node and got the results shown in Fig. 3. When we mapped two ranks on two CPU nodes, we looked at the cases of communication via InfiniBand and Ethernet. To demonstrate the case that two nodes communicate via QDR InfiniBand, we compared Open MPI with MVAPICH2. We show the results in Fig. 4. We compared MPICH and Open MPI (using the `-mca` option to bind Open MPI to Ethernet) when two nodes communicate via Ethernet. The results are shown in Fig. 5. In the three resulting figures, we show latency over the packed data buffer (transferred message) size.

From the results, we make the following observations:

- For the Contiguous case, Open MPI performs the worst when the data communication is intra-node and inter-node via InfiniBand, but it performs better than MPICH when communication is via Ethernet. This phenomenon holds for the Tiled type except that the latency curves alternate when data size is smaller than 128KB when via Ethernet.
- MVAPICH2 outperforms Open MPI via InfiniBand and for intra-node communication on the Contiguous and Tiled types. Except for the Tiled type via InfiniBand, the performance gap between MVAPICH2 and Open MPI increases with message size.
- Each of the three libraries shows similar performance between the Bucket and Alternating types. MVAPICH2 performs the worst when doing intra-node communication; Open MPI and MPICH perform similarly when within one node; Open MPI and MVAPICH2 perform similarly over InfiniBand.
- In the case of Ethernet, when packed data size is smaller than about 128KB, the performance of Open MPI and MPICH alternate. MPICH can outperform Open MPI by 27% at a data size of 100KB; however, when the data size is larger than around 128KB, Open MPI outperforms MPICH in all four datatypes. This result is possibly because of the performance difference between the communication devices used in the two libraries. At larger data sizes, when the communication latency dominates, the performance of marshaling and demarshaling in libraries becomes hard to distinguish.
- We notice a breakpoint in MPICH's intra-node performance plots, which is the result of the 64KB threshold set for the rendezvous protocol.

Cheaha is equipped with ConnectX-5 EDR InfiniBand cards, where the network bandwidth is higher and the latency is lower than that of SCC. We added the experiments with Intel MPI, and tested the intra-node case and the InfiniBand case with Cheaha, and show the results in Fig. 6. We demonstrate three types except the Alternating type since it performs similarly to the Bucket type.

From the results, we make the following observations:

- The relation between the performance of MVAPICH2 and Open MPI partially holds on the Cheaha (e.g., for the Intra-node Tiled and Bucket, and the inter-node Contiguous and Tiled, MVAPICH2 outperforms Open MPI).
- Intel MPI performs the best via InfiniBand, except for Bucket type, where Open MPI outperforms Intel MPI by 18% at a message size of 400KB. Specifically, Intel MPI outperforms Open MPI by up to 3x for Tiled type.
- With the higher-speed InfiniBand card, the performance difference between the inter-node and intra-node cases is

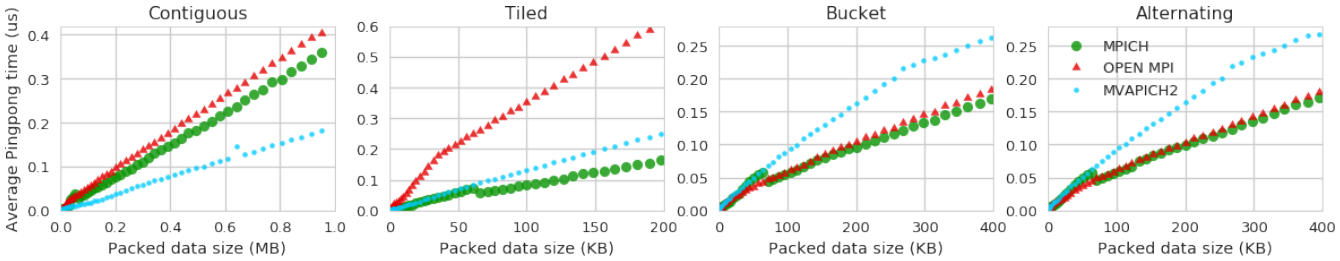


Figure 3: Derived datatype performance for intra-node communication on SCC.

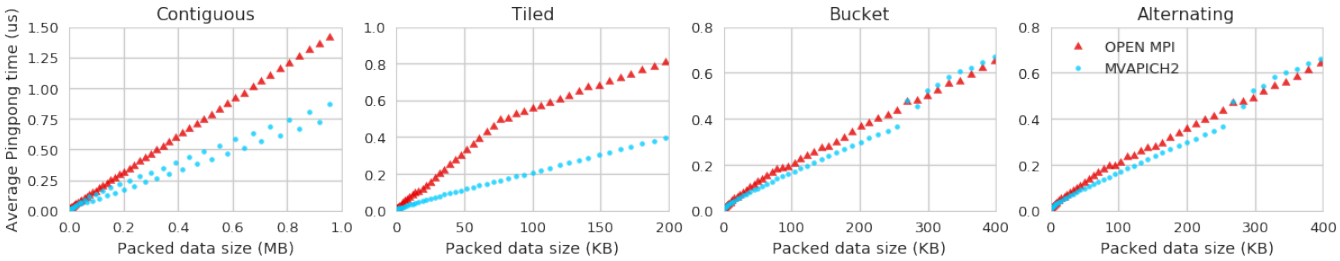


Figure 4: Derived datatype performance for inter-node communication via QDR InfiniBand on SCC.

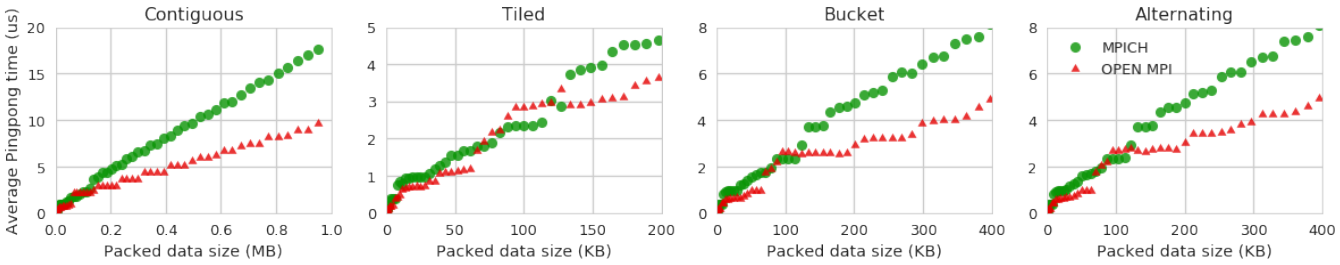


Figure 5: Derived datatype performance for inter-node communication via TCP/IP over Ethernet on SCC.

smaller than that of the results on SCC. However, for different datatypes, the performance relation across implementations is still alternating.

Overall, from our observations of the results on two different HPC systems, we offer the following conclusions:

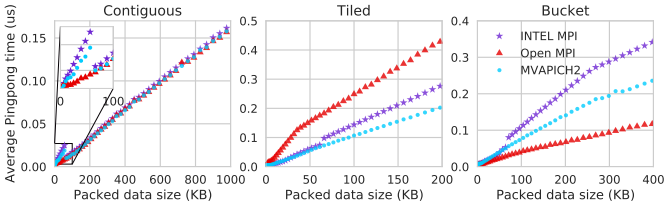
(1) Results are sufficient to support our claim that datatypes' performance is not completely portable across MPI implementations. Each datatype has different performance in the four implementations.
(2) As we expected, the datatype performance relation among implementations changes when using different underlying communication protocols.

4.2.2 4D Matrix Edge Transfer. In high-performance scientific computing where stencils and spatial decomposition exist, data are usually stored in multi-dimensional matrices. The innermost three dimensions typically represent the x,y,z coordinates in the 3D space, and the outer dimensions can be used to represent different types of variables (such as time or a timelike parameter). We present the case of 4D matrices, where the fourth dimension stands for two types of variables: double and float. The memory layout can simply be thought of as two contiguously stored 3D arrays shown in Fig. 7.

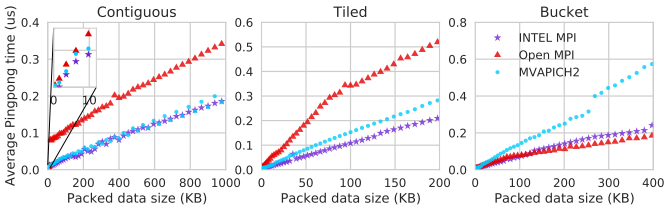
In the edge exchange process of stencil computing (we refer to Minighost of Mantevo Suite [7]), each rank packs the edge data and transfers the edge to the neighbor rank. Upon receiving the edge from other ranks, the neighbor rank unpacks the data into its memory space. The edge in our experiment is set to 5 elements wide in the first dimension, whereas the dimensional width (N) of one 3D matrix varies from 10 to 500 elements. The edge can be represented with derived datatypes or via manual copying. The edge exchange process in our benchmark happens only between two ranks, where communication consists of only blocking send and receive.

We use four different derived datatype constructors to describe the edge memory layout: struct, indexed, vector, and cont_of_resized datatypes. The construction of each of the four types follows:

◇ *Struct*: Constructed using one `MPI_Type_create_struct` constructor. The length of the type, displacement, and bloklen arrays are the block number, $5 * N * N * 2$. The first half of



(a) Intra-node communication.



(b) Inter-node communication via EDR InfiniBand.

Figure 6: Derived datatype performance benchmark results on Cheaha (equipped with EDR InfiniBand).

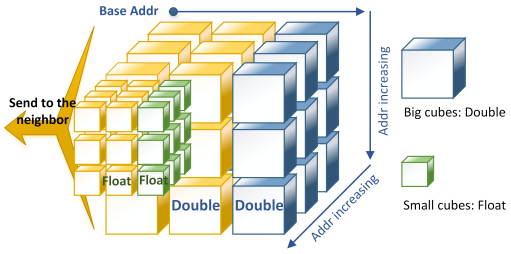


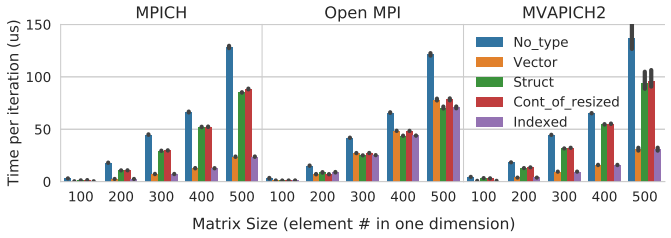
Figure 7: Example memory layout of the 4D matrix edge transfer benchmark. The fourth dimension of the 4D matrix is the data type of the elements (double or float). The yellow parts are the edge to be sent to the neighbor.

the type array stores `MPI_DOUBLE`, and the second half stores `MPI_FLOAT`.

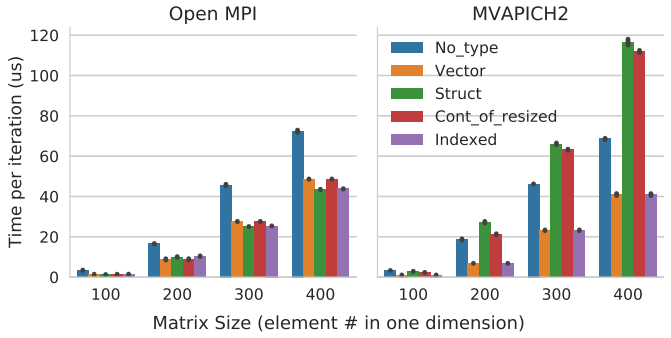
- ◇ *Indexed*: Constructed using two `MPI_Type_indexed` constructors, one for the matrix with double type and one for the matrix with float type.
- ◇ *Vector*: Constructed using two `MPI_Type_vector` constructors for the same purpose as the Indexed type.
- ◇ *Cont_of_resized*: Constructed for the edge of each 3D matrix, using one `MPI_Type_contiguous` to represent the five contiguous elements, `MPI_Type_create_resized` to add the extend, and another `MPI_Type_contiguous` to represent the edge of the whole matrix.

We refer to manually packing and unpacking as *No_type*, where the user copies the edge data to/from two contiguous buffers before sending/after receiving. Out of the five cases, only the Struct uses one constructor (and thus one pair of send and receive calls for performing the edge exchange); the other cases use two pairs of send and receive calls.

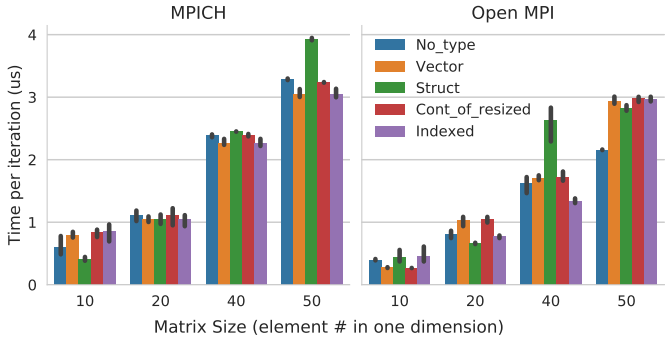
We ran the benchmark for the three implementations with three types of communication protocols each for 1,000 iterations (and



(a) Performance results of intra-node communication.



(b) Performance results of inter-node communication via QDR InfiniBand.



(c) Performance results of inter-node communication via Ethernet.

Figure 8: 4D matrix edge transfer results on SCC system. In the figure, Matrix size is the dimensional element number of the matrices.

performed 5 runs). Verification of the exchange results is done outside the timing section.

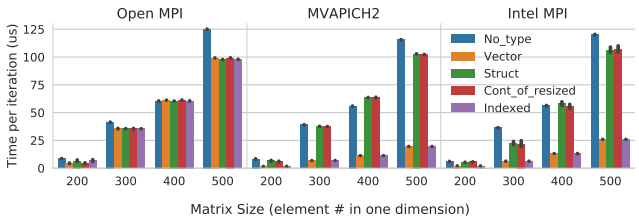
We ran on SCC nodes; the average results per iteration are shown in the bar graphs in Fig. 8. The error bars indicate the five different runs. We show the results of matrix size up to 500 in Fig. 8a, matrix size up to 400 in Fig. 8b, and matrix size up to 50 in Fig. 8c to capture the interesting details.

From these results, we make the following observations:

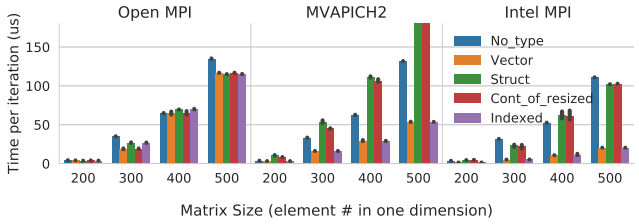
- The manual copying baseline (No_type) performs the same across the three implementations in the cases of intra-node communication and InfiniBand. In the case of Ethernet, Open MPI outperforms MPICH by around 33% on average for No_type; this phenomenon is also shown by the benchmark in § 4.2.1.

- Open MPI performs the worst among the three implementations in Vector and Indexed cases for intra-node communication, as well as when communicating over InfiniBand. For communication over Ethernet, we show small matrix size where the Ethernet latency is not dominant; the results of Vector in Open MPI and MPICH alternate, which is consistent with the benchmark results.
- For intra-node communication, derived datatypes outperform the manual copying baseline in all three implementations. For inter-node communication, the three implementations perform worse than the baseline (except for Open MPI via QDR InfiniBand) when using one or two datatypes (e.g., MVAPICH2 is 71% slower than the baseline at matrix size of 400 when using the Struct and Cont_of_resized datatypes via InfiniBand).
- The three implementations provide the most similar performance on Struct and Cont_of_resized for intra-node communication, among the four forms of derived datatypes.

We performed the same experiment on the Cheaha system. To capture the interesting details, we show the results of matrix size from 200 to 500 in Fig. 9. From the results, we observe several similar



(a) Performance results of intra-node communication.



(b) Performance results of inter-node communication with EDR InfiniBand.

Figure 9: 4D matrix edge transfer on Cheaha system (equipped with EDR InfiniBand).

performance trends as the ones on SCC. We also observe that

- Vector and Indexed types of MVAPICH2 can outperform manual packing and unpacking by up to 80% during intra-node communication,
- Open MPI provides similar performance when representing the same memory layout using four derived datatypes, and
- Among the three implementations, on average, Intel MPI provides the best performance for Struct and Cont_of_resized.

We offer two more conclusions from the results in addition to the two conclusions from the benchmark results:

(3) On the same system, when representing one memory layout, the profiled Open MPI, MVAPICH2, Intel MPI, and MPICH implementations provide similar performance without using datatypes (except via Ethernet) but provide different performance with derived datatypes.
(4) In all of our experiments, for all four implementations, using derived datatypes can outperform manual packing and unpacking by up to 80%. However, some derived datatype representations have negative impact on performance (71% slower than manual packing and unpacking).

From this case study, we can also see that derived datatype normalization is achieved differently in different libraries (e.g., profiled Open MPI achieved better than MVAPICH2), which further demonstrates the performance portability issue.

5 POTENTIAL SOLUTIONS TO SLOW AND UNPORTABLE DATATYPE PERFORMANCE

We discuss solutions for existing MPI implementations here. The presented solutions cover (i) improving the specific library that performs the worst in some derived datatypes and (ii) providing better performance to derived datatypes.

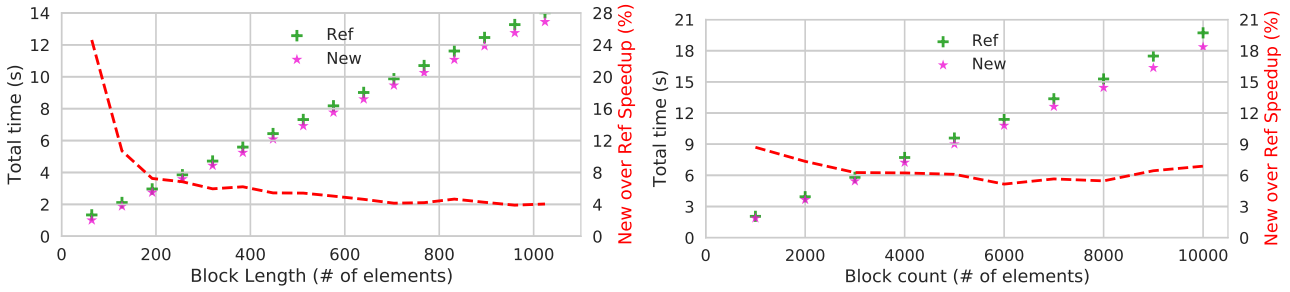
5.1 Software-Based Solutions

We demonstrate in § 4 the performance difference among MPI implementations when using derived datatypes. Specifically, in intra-node cases, Open MPI has the worst performance in processing the Tiled type (benchmark) and the Vector type (case study), which are both constructed using `MPI_Type_vector`. The reasons for poor performance are always complex, but, to determine the obvious reasons in this case, we choose to look into the source code of the vector type. Here we compare the source code of Open MPI with that of MPICH and show the possible reasons for Open MPI's poor performance with vector types.

Both MPICH and Open MPI segment large data into smaller chunks for packing and sending. Specifically, in the case we look at, Open MPI uses a packing function for data before sending (`opal_generic_simple_pack`). This function iterates element by element through the data to be packed. Contiguous, non-contiguous, built-in, and derived datatypes are all handled by the same function.

MPICH uses different functions for different datatypes for packing, and each function is specialized to one datatype (e.g., Vector has segment_vector_m2m, Indexed has Segment_indexed_m2m, and Contiguous has Segment_contig_m2m. Segment_vector_m2m uses an inline-macro `MPIDI_COPY_TO_VEC` to iterate through the segment to be packed). In each iteration, `MPIDI_COPY_TO_VEC` looks at one block of the current segment and packs several contiguous elements within this block to a buffer. The outer loop count is therefore smaller than the case in Open MPI. This macro performs manual loop unrolling.

Manual or compiler-loop-unrolling within each datatype processing function can improve the datatype's performance. To achieve further performance improvement on top what MPICH provides, we can better exploit the efficiency of modern CPU architecture (e.g., SSE, AVX, and AVX2); vectorized instructions can be used. However, compilers have limitations and may not detect some optimizable parts in the code even with the O3/Ofast option(s). In



(a) Test 1: stride = 1040 `MPI_INT` and block count = 1,000.

(b) Test 2: stride = 1040 `MPI_INT` and block length = 128 `MPI_INT`.

Figure 10: Performance comparison between Ref (MPICH-3.2.1 with -O3) and New (MPICH-3.2.1 with -O3 and AVX intrinsics in data packing) on intra-node ping-pong test for 10K iterations.

2013, Schneider et al. [19] showed that using SSE and AVX instructions can improve the performance of some derived datatypes. They demonstrate an online approach using the LLVM framework to produce vectorized machine code for each datatype at commit time. They use 128-bit store (SSE2) because of platform limitations and demonstrate only for Intel X86 architecture[2].

In this paper, we briefly demonstrate the performance gain of using 256-bit AVX load and store for only marshaling (no demarshaling) vector datatypes. We experimented on one SCC node with 2 fourteen-core 2.4 GHz Intel Xeon E5-2680v4 processors in two sockets. The details of the experiments and the performance results are shown in Fig. 10. We compiled with gcc 7.2.0 -O3 for both the baseline (Ref) and our optimized version (New). On top of MPICH's manual loop unrolling, we replace the inner loop element-by-element store with Intel 256-bit intrinsics. With vector instructions, we achieve a 1.24x speedup over using only the -O3 option in the best case (Block Length is 64 in Fig. 10a). For Test 2, where the block length is a constant, the performance gain achieved from each block is around 6%—thus in total 6% for the whole memory layout.

Different CPU architectures have different vector instructions. For example, ARM has NEON [1] and MIPS has MSA [13]. Manually replacing code with intrinsics can achieve performance, and, with the help of LLVM, the vectorization process can be automated. Thus, we can further achieve performance gain from using vector instructions on different architectures such as X86, ARM, MIPS, and Power PC.

The performance gain achieved from using the current vector instructions, however, is limited. Derived datatypes that have low vectorization levels can benefit only minimally from using current vector instructions. We also suggest adding new vector instructions such as stride store/load for data marshaling purposes.

5.2 Hardware/Hybrid-Based Solution

Offloading datatypes can potentially reduce the performance gap among different implementations and can completely/partially relieve CPUs from marshaling and demarshaling. There are NIC offload solutions provided in both researches and industrial products, which are covered as related work in § 6.

[2]With flexible installation criteria for open source MPI's, picking the best vectorization primitives for a given platform should not be a future barrier.

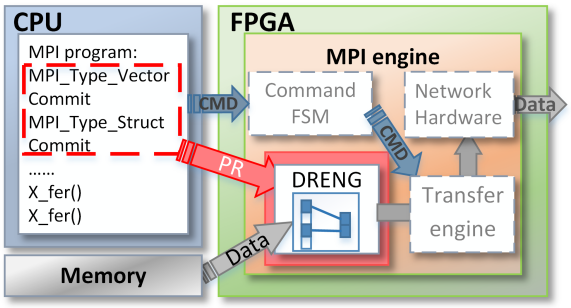


Figure 11: CPU-FPGA system with derived datatype offload. DRENG is the proposed Data reorganization engine.

In recent years, as adding FPGAs to HPC clusters has become a growing trend, FPGAs have emerged as strong candidates for MPI offload [20, 29]. As part of the program to offload MPI functionality, we propose accelerating MPI derived datatypes and the data marshaling process using pre-existing FPGAs in the cluster. We demonstrate an FPGA design in Fig. 11.

FPGAs provide reconfigurability and partial reconfigurability (PR), which allow the configuration of hardware support for various datatypes. For each MPI application, the derived datatypes used can be known as a priori. We create the hardware design of the derived datatype offload as the datatype reorganization engine (DRENG), and configure the FPGA together with the MPI communication functions offload (see an example design by the authors of this paper [29]). When the DRENG is designed as a PR region, for a new MPI program, the DRENG for new datatypes can be reconfigured with PR, whereas the rest of the MPI hardware remains the same.

We offer an example here to demonstrate the potential benefits that FPGAs can bring to datatype processing. The packing process can be interpreted as hardware wiring: wires are what connects the input buffer and the output contiguous buffer. Therefore, the DRENG can process data marshaling with a high throughput via pipelining. Once the output buffer reaches a threshold, the buffered data will be transferred to the network hardware; meanwhile, the DRENG is packing the rest of the data. This design is an example of how to use FPGAs for high-throughput data marshaling. We believe that more efficient designs can be delivered by making full use of the flexibility of the FPGAs.

6 RELATED WORK

Since the emergence of derived datatypes, the study of each datatype's performance becomes active every few years. In the early 2000s, researchers benchmarked derived datatypes and modified mini-applications to study the performance [12, 15]; datatype performance portability had not yet been found as an issue. Later studies praised derived datatypes for helping express distributed data structures more conveniently and for providing performance advantages over the cases without derived datatypes [2, 9, 28]. However, the results have not completely sustained after years of technology improvement.

A recent study by Carpen-Amarie et al. [3] demonstrated a series of new observations of the performance and portability of derived datatypes that opposes earlier observations. They find that the performance of non-trivial derived datatypes is different between NEC MPI and MVAPICH2, and they also mention the datatype normalization issue. We have quite similar observations with Carpen-Amarie et al. in some senses, which encouraged us to further study more aspects of MPI derived datatypes.

As opposed to the work of Carpen-Amarie et al., we experimented with four MPI implementations: Intel MPI, MVAPICH2, Open MPI, and MPICH. We focused mainly on the performance portability issue across implementations and experimented with various communication methods (intra-node memory-copy, Ethernet, InfiniBand). We not only benchmarked various datatypes but also investigated their impacts on real-world cases. Moreover, we provide general solutions to the performance and portability issues of datatypes.

There has been a substantial amount of research on optimizing derived datatypes. Work from the late 1990s and early 2000s [6, 17, 23] has been mostly introduced into MPI implementations. Offline generating and processing datatypes were proposed [10, 14, 16], but those studies either encountered offline processing overhead or modified the user's code or memory layout, which might affect correctness. Datatype normalization (the MPI implementation picks the most efficient representation of a memory layout) has been introduced but remains costly[3, 4]. Schneider et al. [19] propose an online optimization approach using the LLVM framework to produce vectorized machine code for each datatype at commit time.

Offloading approaches have also been investigated. InfiniBand provides read scatter, write gather, and the User-mode Memory Registration (UMR) functionalities that can be used for marshaling non-contiguous data [11, 18, 27]; these functionalities are used with RDMA operations. Researchers have also explored adding specialized hardware to the NIC for data marshaling [21] to improve performance. In GPU cluster domain, there is also work about partially offloading derived datatypes' processing to GPUs [8, 26] when performing inter-GPU communication with MPI.

7 CONCLUSION

In this paper, we addressed the performance portability and overall performance issues of derived datatypes in four MPI implementations: MPICH, Open MPI, MVAPICH2, and Intel MPI. We performed a series of experiments to test the derived datatype performance. From our results, we found that derived datatypes' performance can vary by 3x across MPI implementations and that each datatype has different performance in the four implementations. Moreover, the datatype performance relation among implementations changes when using different underlying communication protocols. We also observed that derived datatype normalization is achieved differently in different libraries, which further demonstrates the performance portability issue.

We demonstrated a software approach to first speed up the "slower" one's performance and then use vector instructions for further performance improvement. In addition, we showed our proposed FPGA hardware design as the hardware offload of MPI derived datatypes and discussed potential performance benefits.

ACKNOWLEDGMENTS

Support is acknowledged from the National Science Foundation under Grants Nos. CCF-1562659, CCF-1562306, CCF-1618303, CCF-1617690, CCF-1822191, CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] ARM. 2018. NEON. https://developer.arm.com/technologies/neon [Online; accessed 21-May-2018].
[2] Enes Bajrović and Jesper Larsson Träff. 2011. Using MPI derived datatypes in numerical libraries. In *Proceedings of European MPI Users' Group Meeting (EuroMPI)*, Vol. 6960 LNCS. 29–38.
[3] Alexandra Carpen-Amarie, Sascha Hunold, and Jesper Larsson Träff. 2017. On expected and observed communication performance with MPI derived datatypes. *Parallel Comput.* 69 (2017), 98–117.
[4] Robert Ganian, Martin Kalany, Stefan Szeider, and Jesper Larsson Träff. 2016. Polynomial-Time Construction of Optimal MPI Derived Datatype Trees. In *Proceedings of IPDPS*. 638–647. https://doi.org/10.1109/IPDPS.2016.13
[5] William Gropp, Torsten Hoefler, Rajeev Thakur, and Jesper Larsson Träff. 2011. Performance expectations and guidelines for MPI derived datatypes. In *Proceedings of European MPI Users' Group Meeting (EuroMPI)*, Vol. 6960 LNCS. 150–159.
[6] William Gropp, Ewing Lusk, and Debbie Swider. 1999. Improving the performance of MPI derived datatypes. In *Proceedings of the Third MPI Developer's and User's Conference*. 25–30.
[7] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).
[8] John Jenkins, James Dinan, Pavan Balaji, Nagiza F. Samatova, and Rajeev Thakur. 2012. Enabling fast, noncontiguous GPU data movement in hybrid MPI+GPU environments. In *Proceedings of CLUSTER*. 468–476. https://doi.org/10.1109/CLUSTER.2012.72
[9] Dries Kimpe, David Goodell, and Robert Ross. 2010. MPI datatype marshalling: A case study in datatype equivalence. In *Proceedings of European MPI Users' Group Meeting (EuroMPI)*, Vol. 6305 LNCS. 82–91.
[10] Fredrik Kjolstad, Torsten Hoefler, and Marc Snir. 2012. Automatic datatype generation and optimization. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*. 327.
[11] Mingzhe Li, Hari Subramoni, Khaled Hamidouche, Xiaoyi Lu, and Dhabaleswar K. Panda. 2015. High performance MPI datatype support with user-mode memory registration: Challenges, designs, and benefits. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER)*, Vol. 2015-Octob. 226–235.
[12] Qingda Lu, Jiesheng Wu, Dhabaleswar Panda, and P. Sadayappan. 2004. Applying MPI derived datatypes to the NAS benchmarks: A case study. *Proceedings of the International Conference on Parallel Processing Workshops* (2004), 538–545.
[13] MIPS Technologies, Inc. 2018. MIPS ® SIMD Architecture. https://www.mips.com/products/architectures/ase/simd/ [Online; accessed 21-May-2018].
[14] Ben Perry and Martin Swany. 2010. Improving MPI communication via data type fission. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*. 352.
[15] Ralf Reussner, Jesper Larsson Träff, and Gunnar Hunzelmann. 2000. A Benchmark for MPI Derived Datatypes. In *Proceedings of Euro PVM/MPI*, Vol. 1908. 10–17.
[16] Robert Ross, Robert Latham, William Gropp, Ewing Lusk, and Rajeev Thakur. 2009. Processing MPI datatypes outside MPI. In *Proceedings of Euro PVM/MPI*, Vol. 5759 LNCS. 42–53.

[17] Robert Ross, Neill Miller, and William Gropp. 2003. Implementing Fast and Reusable Datatype Processing. In *Proceedings of Euro PVM/MPI*. 404–413.

[18] Gopalakrishnan Santhanaraman, Jiesheng Wu, and Dhabaleswar K Panda. 2004. Zero-Copy MPI Derived Datatype Communication over InfiniBand. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2004), 47–56.

[19] Timo Schneider, Fredrik Kjolstad, and Torsten Hoefler. 2013. MPI Datatype Processing using Runtime Compilation. In *Proceedings of European MPI Users' Group Meeting (EuroMPI)*. 19–24.

[20] Josh Stern, Qingqing Xiong, Jiayi Sheng, Anthony Skjellum, and Martin C. Herbordt. 2017. Accelerating MPI_Reduce with FPGAs in the Network. In *Proc Workshop on Exascale MPI*.

[21] Noboru Tanabe, Hirotaka Hakozaki, Hiroshi Ando, Yasunori Dohi, Zhengzhe Luo, and Hironori Nakajo. 2010. An enhancer of memory and network for applications with large-capacity data and non-continuous data accessing. *Journal of Supercomputing* 51, 3 (March 2010), 279–309.

[22] Noboru Tanabe and Hironori Nakajo. 2008. Introduction to Acceleration for MPI derived datatypes using an enhancer of memory and network. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*. 17–23.

[23] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. 1999. Flattening on the fly: Efficient handling of MPI derived datatypes. In *Proceedings of the European PVM/MPI Users' Group Meeting (Euro PVM/MPI)*, Vol. 1697. 109–116.

[24] Boston University. 2018. Shared Computing Cluster (SCC). https://www.bu.edu/tech/support/research/computing-resources/scc.

[25] University of Alabama at Birmingham. 2018. Research Computing. https://docs.uabgrid.uab.edu/wiki/Resources.

[26] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Xiangyong Ouyang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *Proceedings of CLUSTER*. 308–316. https://doi.org/10.1109/CLUSTER.2011.42

[27] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. 2004. High performance implementation of MPI derived datatype communication over InfiniBand. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, Vol. 00. 14–23.

[28] Yongwen Wu, Junqiang Song, Kaijun Ren, and Xiaoyong Li. 2015. MPI derived datatypes and data communication analysis in meteorological applications. In *Proceedings of IEEE International Conference on Smart City(SmartCity)*. 536–541. https://doi.org/10.1109/SmartCity.2015.125

[29] Qingqing Xiong, Anthony Skjellum, and Martin C. Herbordt. 2018. Accelerating MPI Message Matching Through FPGA Offload. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*.