MARs: Memory Access Rearrangements in Open MPI Datatype Engine

***Abstract*—Datatype engine in MPI libraries supports communication layer by handling non-contiguous data transfers. It uses primitive datatypes (integer, float, and etc.) as building blocks for derived datatype. The idea of having datatype engine is not only to facilitate the creation of complex datatypes but to also have high efficiency. In this paper, we focus on Open MPI datatype engine, which uses pipeline technique to hide communication overhead. With communication overhead being well hidden, improving the performance of pack/unpack functions is the goal. We focused on the cons in Open MPI datatype engine by revamping the datatype description using IOVEC and introducing memory access rearrangements (MARs). Using IOVEC datatype description, we concluded that the instruction count is what limiting current datatype engine and the performance has linear correlation to the instruction count. Although we see 1.2X and sometimes 3.2X performance from IOVEC, the performance is mainly based on the instruction count and Open MPI datatype description could outperform IOVEC for compacted datatypes. By introducing MARs, we have minimized instruction count. After applying pipeline/segmentation strategy on MARs, we have also seen up to 2.3X performance.**

***Index Terms*—HPC, MPI, Open MPI, Datatype**

I. Introduction

Exascale-computing is closer than ever before because of the rapid evolution of the hardware. While Moores Law [10] still in tack, we will continue to see a steady increase in computing power. With more computing power, we have seen an unprecedented time in data flow. However, the memory capability did not improve as rapidly and has limited scientific applications to unlock HPC systems full potential. With various programming model in HPC community, Message Passing Interface (MPI) [13] has always been the standard communication model in all parallel applications. Several key features in MPI provides portability while maintaining efficiency. One of them is the datatype engine.

Datatype Engine supports communication layer by handling non-contiguous data transfer. Datatype in datatype engine acts as a blueprint for the memory layout. All datatypes are composed of basic datatypes, such as integer, double, float, etc. The combination of basic datatypes is called the derived datatype. In situations where non-contiguous transfer is not supported, datatype engine would put non-contiguous data into a contiguous buffer and send it through the network. On the receiver side, the datatype engine would do the opposite. However, datatype engine isn't as popular and used as often comparing to other MPI features, mainly due to its poor performance. Research has suggested that pack and unpack could take up to 90% of all communication time [14]. Users are more likely to pack non-contiguous data themselves. This clearly defeats the purpose of having a datatype engine.

Numerous researches have gone to reduce/erase the overhead for datatype engine during communication, such as User-mode Memory Registration (UMR) [9], zero-copy [6] [7] [12] [19] approach by using highly efficient InfiniBand [1]. It is far easier to achieve better performance by simply applying modern hardware capabilities, for example, applying hardware specific vector extensions [21] [22] [23]. However, there are specific hardware requirements for all these approaches.

Through our work, we have also found that the optimization process for datatype description could fail when user describes the datatype in various ways, and the resulting datatype would not have the optimal performance during pack. Because finding the optimal datatype description would take polynomial time [8], we decided to revamp the datatype description to solve datatype optimization inconsistency.

We took the challenge to revamp the datatype description and the pack function to allow all scientific applications to run on all platforms. Through our work, we have concluded that the number of instructions is the main correlation to the datatype engine performance when memory access sequence is kept the same. The number of instructions could also affect performance greatly when memory access sequence is altered. Essentially, more instructions equal less performance. Through our experiments, we have seen a 1.2X to 1.5X performance boost and, in some cases, a 3.2X.

II. Related Work

Improving MPI datatype performance has long been an effort to improve communication performance. However, because of the high overhead datatype engines come with [15] [11], it has prevented scientific applications from adopting MPI datatype, even though efforts have tried to revamp datatype description [18] [5] [20]. In a typical point-to-point communication, up to 90% of the overhead goes into packing non-contiguous data into a contiguous buffer on the sender side and unpacking contiguous buffer into non-contiguous data on the receiver side [14].

A lot of research efforts have gone into improving communication using datatype engine and utilizing underlying hardware capabilities. With interconnects such as Infiniband (IB) [1], communications are able to take advantage of efficient network features and researches have shown the benefit of using gather/scatter and Remote Direct Memory Access (RDMA) [17] [19] [12]. It is an efficient alternative to remove the pack and unpack from communication and

directly write/read blocks of data from sender/receiver. To further utilize the capability of the interconnect, Mellanox purposed User-mode Memory Registration (UMR) [9], which can read/write multiple non-contiguous blocks of data through RDMA using Scatter-Gather-Lists (SGL). With similar concept, zero-copy strategy [6] [7] [12] [19] is also introduced to erase the overhead of having an extra copy of data from packing/unpacking. These concepts could efficiently support MPI communication, but they do require systems to equip modern day interconnect.

While hardware is gaining more performance and capabilities, some researches have suggested to offload communication and computation to other parts of the hardware such as the latest smart NIC, Bluefield [16] [2]. While smart NICs aren't as common, utilizing Intels Advanced Vector Extensions (AVX) and Arms Scalable Vector Extension (SVE) is able to improve the time-to-solution in predefined MPI reduction operations [21] [22] [23]. The same vectorization would also help the datatype engine to increase efficiency and close the gap to peak performance.

III. Background

A. Open MPI [4] Datatype Engine

```
-------G---[---][---]    OPAL_LOOP_S 10 times the next 6 elements extent 44
--C---P-D--[---][---]      OPAL_INT4 count 1 disp 0x0 (0) blen 1 extent 4 (size 4)
--C---------[---][---]    OPAL_LOOP_S 3 times the next 3 elements extent 12
--C---P-D--[---][---]      OPAL_INT4 count 1 disp 0x8 (8) blen 2 extent 8 (size 8)
--C---P-D--[---][---]    OPAL_FLOAT4 count 1 disp 0x10 (16) blen 1 extent 4 (size 4)
--C---------[---][---]    OPAL_LOOP_E prev 3 elements first elem displacement 8 size of data 12
-------G---[---][---]    OPAL_LOOP_E prev 6 elements first elem displacement 0 size of data 40
-------G---[---][---]    OPAL_LOOP_E prev 7 elements first elem displacement 0 size of data 400
Optimized description
-------G---[---][---]    OPAL_LOOP_S 10 times the next 3 elements extent 44
-cC---P-DB-[---][---]      OPAL_INT4 count 1 disp 0x0 (0) blen 1 extent 4 (size 4)
-cC---P-DB-[---][---]     OPAL_UINT1 count 1 disp 0x8 (8) blen 36 extent 36 (size 36)
-------G---[---][---]    OPAL_LOOP_E prev 3 elements first elem displacement 0 size of data 40
-------G---[---][---]    OPAL_LOOP_E prev 4 elements first elem displacement 0 size of data 400
```

Fig. 1: Indexed Datatype Description

```
--C---P-D--[---][---]     OPAL_UINT1 count 1 disp 0x0 (0) blen 4 extent 4 (size 4)
--C---P-D--[---][---]     OPAL_UINT1 count 9 disp 0x8 (8) blen 40 extent 44 (size 360)
--C---P-D--[---][---]     OPAL_UINT1 count 1 disp 0x194 (404) blen 36 extent 36 (size 36)
-------G---[---][---]    OPAL_LOOP_E prev 3 elements first elem displacement 0 size of data 400
Optimized description
-cC---P-DB-[---][---]     OPAL_UINT1 count 1 disp 0x0 (0) blen 4 extent 4 (size 4)
-cC---P-DB-[---][---]     OPAL_UINT1 count 9 disp 0x8 (8) blen 40 extent 44 (size 360)
-cC---P-DB-[---][---]     OPAL_UINT1 count 1 disp 0x194 (404) blen 36 extent 36 (size 36)
-------G---[---][---]    OPAL_LOOP_E prev 3 elements first elem displacement 0 size of data 400
```

Fig. 2: Optimized Indexed Datatype Description

Using Open MPI datatype feature, a datatype description can be exported. Fig. 1 is an example of a typical datatype in Open MPI. The top half of the Fig. 1 is the description given by the user and the bottom part is the optimization description generated by the datatype engine. Typically, MPI will optimize a datatype description during commit time to ensure efficient data transfer performance.

As Fig. 1 shows, an example of a typical datatype in Open MPI is consisted of three elements,

1) Loop_start
2) Data
3) Loop_end

Loop_start and Loop_end indicate what and how many times elements in datatype description will be repeated. And an additional Loop_end is added as the last element for all datatype to indicate the end of the datatype description.

Open MPI datatype describes each element using a vector-like (MPI_Type_vector(count, blocklength, stride)) structure. It ensures optimal performance when datatype is a regular patterned memory layout. Any regular pattern datatype can be compacted using one data element in Open MPI.

However, with the commit time optimization, we have found inconsistency in optimized datatype description. Fig. 1 and Fig. 2 as shown are the same memory layouts. However, the optimized descriptions are not the same, because they are defined using different methods (MPI datatype functions).

Open MPI datatype engine also utilizes pipeline strategy to hide communication overhead behind pack/unpack. It uses stack approach to keep track the start and end points for pack/unpack functions.

B. Commit Time Optimization

Optimization during commit time is a typical place for MPI to minimize the storage space and rearrange the description to help datatype engine handle data movements faster and more efficiently. In Open MPI, optimization will

1) combine datatype elements that are sequential in pack/unpack order and
2) group datatype elements that repeats the same pattern using Loop_start and Loop_end, if elements are exactly the same type and block length, datatype engine will use count in element description to group elements

However, Open MPI datatype engine statically groups the first occurrence of repeated elements. Thus, giving room for inconsistency. Different datatype descriptions will also result in different performance.

Because it would take polynomial time to find the optimal form for datatype [8], we have decided to revamp datatype description which will always result in the same optimized datatype description.

C. Pack/Unpack Functions

There are several pack/unpack functions built inside Open MPI to deal with different scenarios. We focused on the most common one which deals with local pack/unpack. The pack function puts non-contiguous data into a contiguous buffer and unpack does the opposite.

Open MPI optimizes communication performance using pipeline strategy. The datatype engine uses a structure called convertor to keep track of the position. The convertor contains information such as datatype description, count of datatype needs to be packed/unpacked, total data size that needs to be moved, and etc. It also uses stack to record the position within the user buffer.

The convertor in pack/unpack function will go through the datatype description element by element. As it moves along the description, it will keep checking if it has reached the pipeline size. If it does, it will record the position in stack and wait till next pack/unpack pipeline is being called. Since user could tell Open MPI to pack/unpack X number of datatype,

the convertor will also go through the datatype description X number of times.

IV. Motivations

Open MPI datatype engine supports communication layer by handling non-contiguous data. It alleviates the workload from user to hard-code data movements and ensures efficient and stable performance. A typical MPI application with point-to-point communication starts with a process packs non-contiguous data into a contiguous buffer and push it through the network. On the other side, after receiving the contiguous buffer from the sender side, it unpacks the contiguous buffer into non-contiguous memory layout. We are interested in speeding up this whole process. The total communication time can be calculated as:

$$Pack(Tp) + NetworkLatency(L) + Unpack(TU) \quad (1)$$

Because the network latency is scaled linearly in respect to the size of contiguous buffer, Open MPI uses pipelining as in Fig. 3 during communication to further reduce the communication time by overlapping pack/unpack with communication, thus, hiding network latency behind pack/unpack. And the total time for pipelining is calculated as

$$\begin{aligned} N \times (SmallerPack(Tp)) + \\ SmallerUnpack(TU)) + \\ SmallerNetworkLatency(L/N) \end{aligned} \quad (2)$$

Because network latency is hidden and pipelining with small segments won't fully utilize the bandwidth of the network. The bottleneck for the communication becomes the performance of pack and unpack functions when the right size of the pipeline is determined. Thus, improving the performance of pack/unpack functions becomes the final step.

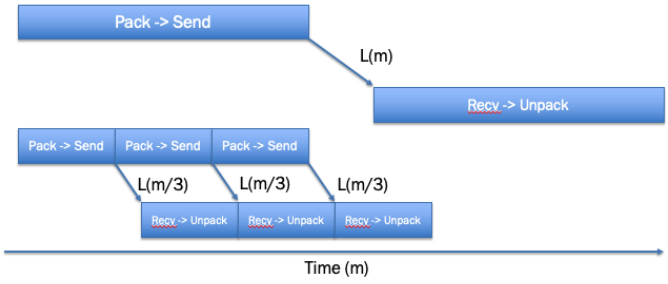


Fig. 3: Pipeline example

V. Benchmark Datatypes & Experiment Setup

After tested with numerous datatypes, we chose 7 datatypes that could both show how datatype engine could be affected by datatype optimization and how instruction count and cache would affect performance.

Fig. 4 visualizes the 7 datatypes we used for our experiments. First datatype starts with seven doubles on the first cache line and one double on the second cache line. We keep splitting one double from first element (the seven doubles) to the next vacant cache line until the last datatype has one double in every cache line. All 7 datatypes have the same data size (64 bytes) and extent (512 bytes) so that the length of user buffer and the length of packed buffer will be the same.

7_1 datatype is the only one that cannot be optimized into a compact form during commit time, while other datatype can be optimized by changing the count in the second element's attribute as several one-doubles with the same extent can be grouped into one element in Open MPI's datatype representation. On the other hand, the IOVEC datatype representation could not make a compact datatype representation since there's no count in element attribute. Thus, the number of datatype elements in the datatype representation for IOVEC is the number of non-contiguous elements.

All our experiments were conducted on a single node which is consists of Intel Xeon CPU E5-2650 v3 @ 2.30GHz. It has 32KB and 256KB of dedicated L1 and L2 caches respectively. It also has a 25600KB shared L3 cache. We based our Open MPI with version 5.0.0 and will refer to it as ”master” throughout this paper.

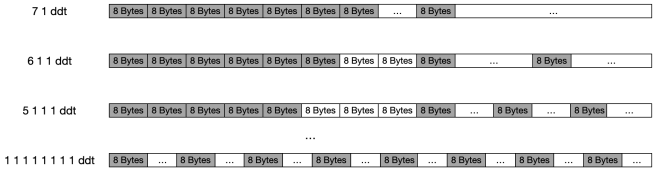


Fig. 4: Datatype Representations

VI. Implementations

A. IOVEC Datatype Representation

We first find a datatype representation that could result in consistent datatype optimization and stable pack/unpack performance. An array of IOVECs is an good choice since it is flattened and has a consistent optimization form. IOVEC datatype representation is created during commit time optimization. Each IOVEC corresponding to a starting address and length of that element. Because IOVEC description is a flattened representation of the datatype, pack/unpack simply goes through the description element by element and repeat this process with X count (given by the user). Between master and IOVEC, IOVEC guarantees fewest ”steps” to parse the datatype since it's a loop around MEMCPY, but we have also made sure IOVEC pack will complement pipeline strategy.

During the commit optimization, IOVEC representation is created by traversing the Open MPI datatype description. First, an IOVEC is created for each data access. Then any contiguous IOVECs will be merged to become one IOVEC. IOVEC description guarantees minimal MEMCPYs for all datatypes, in which current datatype description fails to do so in some scenarios.

Although IOVEC description is straight forward, the performance could suffer due to the flatness of the datatype representation. Since the master uses vector-like attributes to describe every element, it could use a small loop to loop around element when the count is larger than one to avoid ”bookkeeping” process. Because Open MPI uses pipeline to hide communication overhead, the datatype engine in Open MPI has to have the leisure to be able to start and stop at any point in datatype description. Parsing datatype description will require a few ”bookkeeping” when switching element. However, going through partial repeating description does not require ”bookkeeping” which results in fewer instructions. Thus, current master will perform better in cases where partial or all datatype repeats.

B. IOVEC Memory Access Rearrangements (MARs)

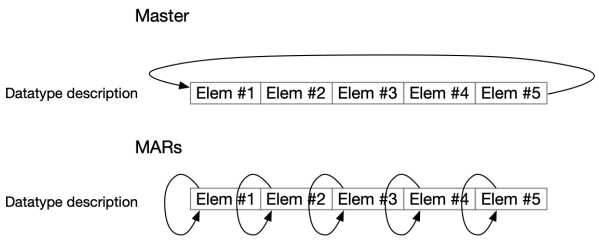
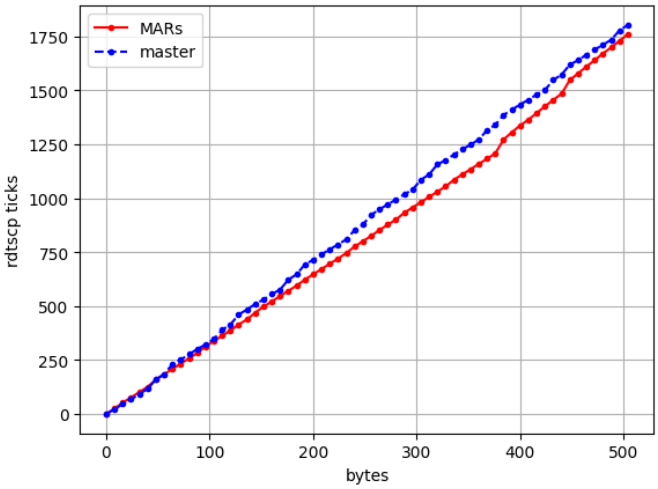


Fig. 5: MARs concept

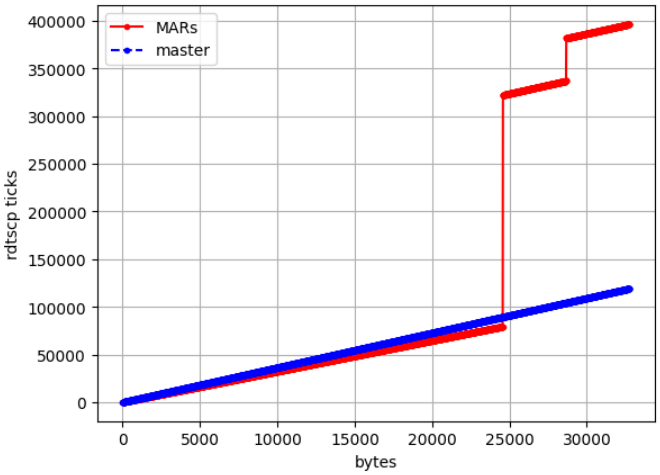
Through our experiment results, we have found that more instructions are issued from parsing datatype description (switching elements) than doing the element which count is larger than 1. We could further reduce the number of instructions by rearranging the pack sequence. Fig. 5 is an example of how MARs is implemented. Instead of looping through datatype description multiple times, MARs pack every element for multiple times so that it only parse the datatype description once.

MARs succeed in reducing the number of instructions, however, the performance will suffer due to increasing cache misses. Because of the access pattern, MARs will keep packing the same element until the last one, even when it need to evict unused data. The constant eviction of unused data will cause huge overhead. We record ticks after every MEMCPY to see how this effect changes with different data sizes.

We used 6_1_1 datatype as an example in Fig. 6 since 7_1 datatype might not fully show all characteristics of the method. Fig. 6a A only packs 8 count of datatype while Fig. 6b packs 2048 count of datatype. There are small ”jumps” in Fig. 6 for both master and MARs. These ”jumps” are the results when datatype engine ”switch” to the next element. MARs only ”switch” to the next element once, and since there are only three elements in the datatype, there are only two ”jumps” because first one does not get recorded. In large sizes, 6b, the resulting ”jumps” are even greater since all the data has been pushed into the next cache level or even the main memory.



(a) 6_1_1 datatype count 8



(b) 6_1_1 datatype count 512

Fig. 6: Master vs. MARs RDTSCP Ticks

The slope of access for MARs is a little bit flatter since MARs has rearrange random access of a datatype into multiple regular access and there is no ”bookkeeping” in between each MEMCPY.

C. MARs Segmentation/Pipeline

With large data size that reaches beyond L1 cache size, MARs would start to evict and replace unused data from cache. Thus, in order to get rid of expensive eviction which results in repetitive data fetching, pipelining/segmenting the user buffer for pack could reduce unnecessary data eviction. One simple method is to segment user buffer by the number of datatypes.

Fig. 7 is an example that MARs perform better in a large count scenario with pipeline. However, the segmentation/pipeline sizes can vary from datatype to datatype by data size and from platform to platform by cache level sizes. After observing a few datatypes' performance using MARs pipeline, we suspect the best performances is correlated to number of factors:

1) Size of the cache in each level
2) Number of physical pages in each cache level
3) Size of the data in datatype
4) Number of cache lines occupied by the data
5) Extent of the datatype
6) The gap sizes between elements
7) Element offset in regarding to the start of datatype
8) Number of layers in cache hierarchy
9) Associativity of the cache

These are the factors that we suspect that needed to be taken into consideration. However, we have yet to find a perfect calculation/equation to maximize all performance for all datatypes. We will present performance with a few segmentation/pipeline sizes in performance section.

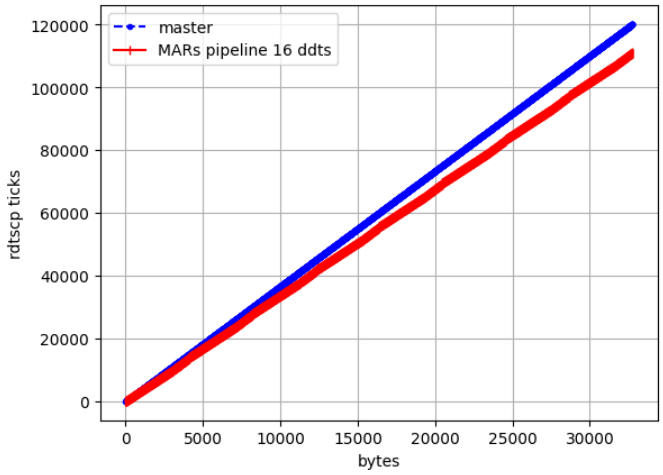


Fig. 7: 6_1_1 datatype count 512 pipeline size 16 ddts

VII. Performance

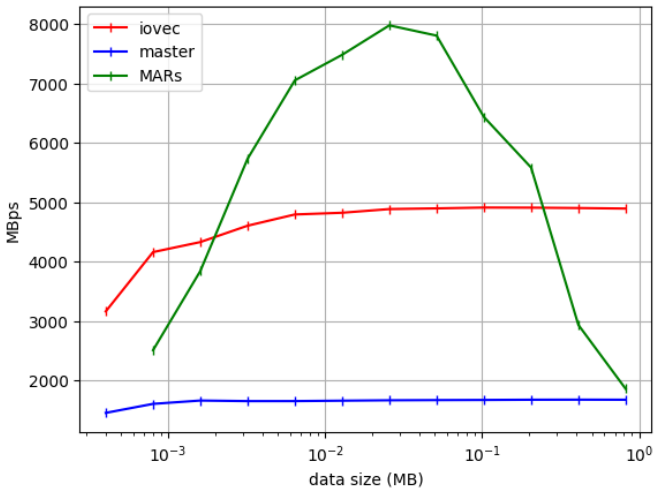
A. Inconsistent Optimization

From previous datatype examples in Fig. 1 and Fig. 2, indexed gap and optimized indexed gap, the indexed gap datatype results in 9 more MEMCPYs than optimized indexed gap datatype. We put these two datatypes into Open MPI datatype benchmark. From Fig. 8, by using current master's packing function, the better optimization description results in 3.2X the performance than the wrongly optimized version and 1.2X the performance than the IOVEC description. We will explain how master outperform IOVEC in the next section.

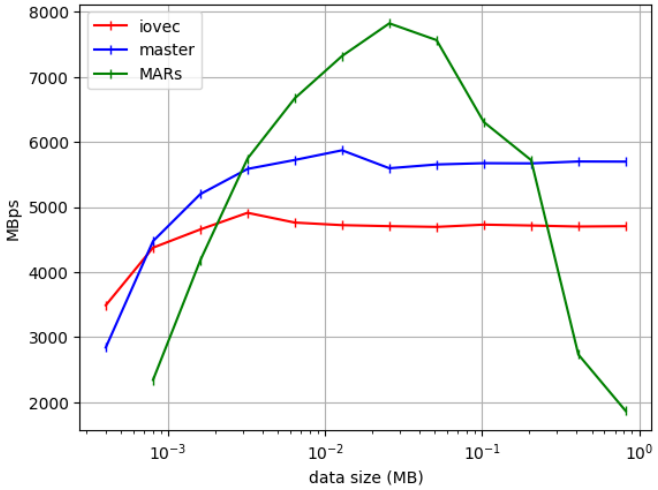
B. IOVEC vs. MARs vs. Current Master with PAPI [3]

When comparing master and IOVEC, the sole cause for performance is the number of instructions. Since master and IOVEC has the exact same memory access pattern and same description access, whichever has fewer steps ("bookkeeping", record position for communication pipeline) in between data movement has the better performance.

We calculated the instructions per 8-byte copy and instructions of "bookkeeping" per loop (looping the datatype description element by element) to see where master and



(a) Indexed Gap Datatype



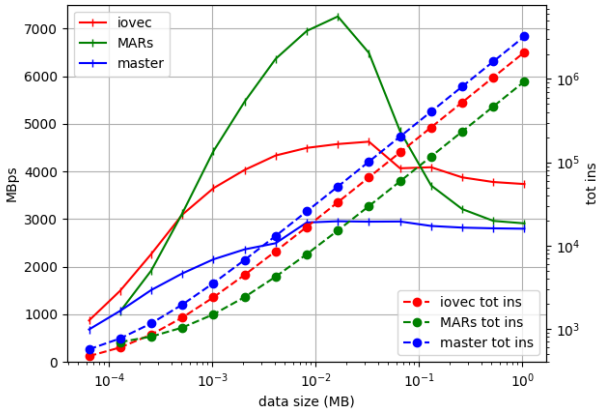
(b) Optimized Indexed Gap Datatype

Fig. 8: Datatype Optimization Issue

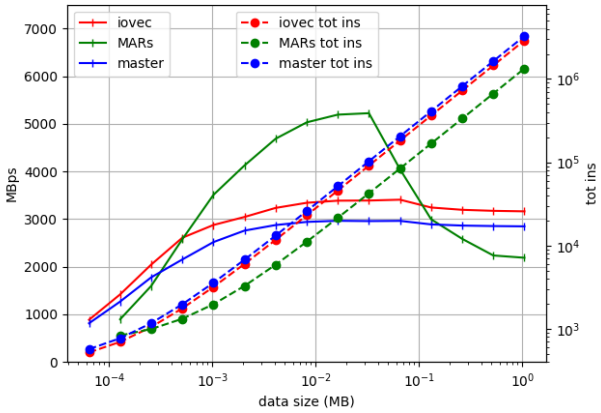
TABLE I: Instruction count table

Datatype 7_1	Instructions/8-byte copy	Instructions of "bookkeeping"/loop
Master	3.3	25.27
IOVEC	8.4	9.71

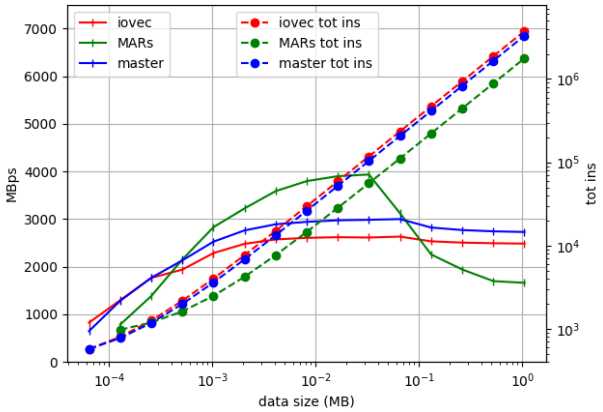
IOVEC differ. We took total instructions with data movement and subtracted with total instructions without data movement to calculate the number of instructions for data movement. IOVEC and master differ in how they move data, the IOVEC uses MEMCPY while the master uses assignments. Because datatype 7_1 could not be optimized, the calculated number of instructions is exactly what one switching of element in description and what an 8-byte copy will take. We can calculate the performance difference just by using the numbers from Table I. Since there are two data elements and a Loop_end element in master, it takes three times of instructions of "bookkeeping". While IOVEC only has two data elements,



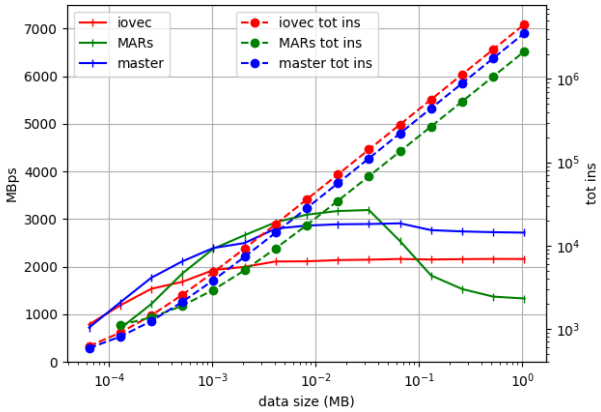
(a) 7_1 Datatype



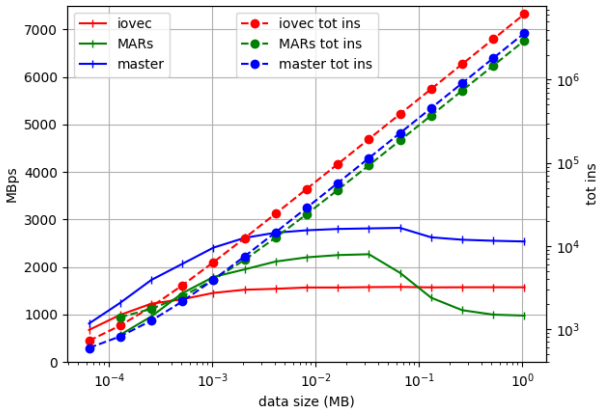
(b) 6_1_1 Datatype



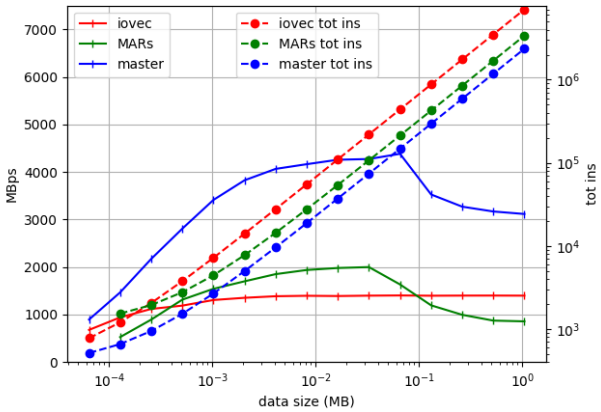
(c) 5_1_1_1 Datatype



(d) 4_1_1_1_1 Datatype



(e) 2_1_1_1_1_1_1 Datatype



(f) 1_1_1_1_1_1_1_1 Datatype

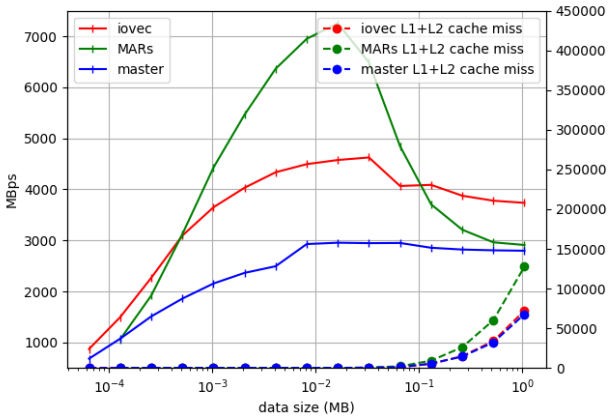
Fig. 9: Master vs. IOVEC vs. MARs, Performance vs. Instruction count

it only takes two times of instructions of ”bookkeeping”.

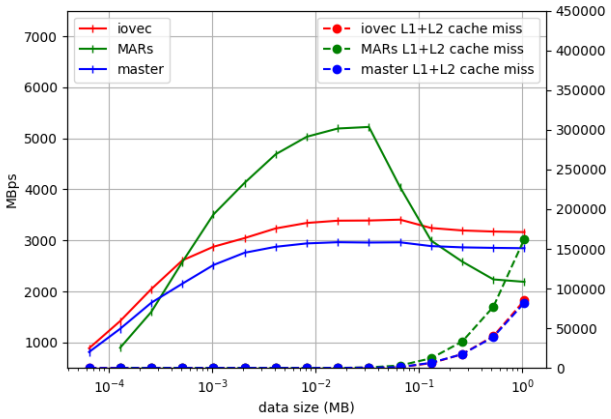
$$Master = 3.3 \times 8 + 25.27 \times 3 = 102.21\, ins/ddt \quad (3)$$

$$IOVEC = 8.4 \times 8 + 9.71 \times 2 = 86.62\, ins/ddt \quad (4)$$

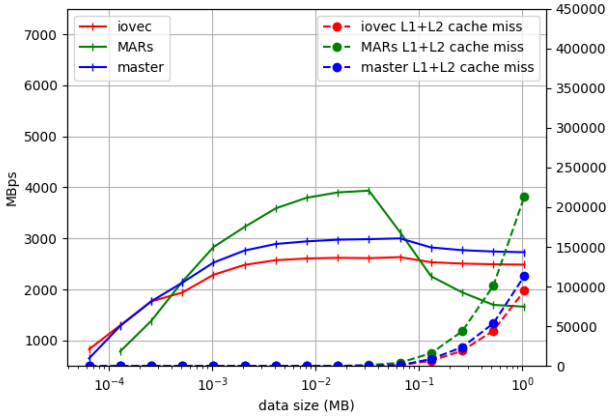
While master is calculated to have 1.18X more instructions by using equation 3 and equation 4, it has 0.75X the bandwidth



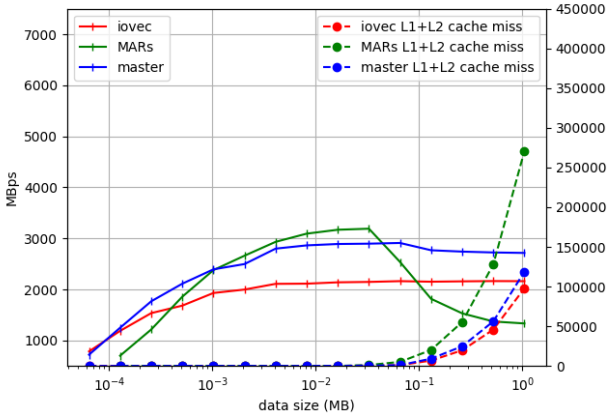
(a) 7_1 Datatype



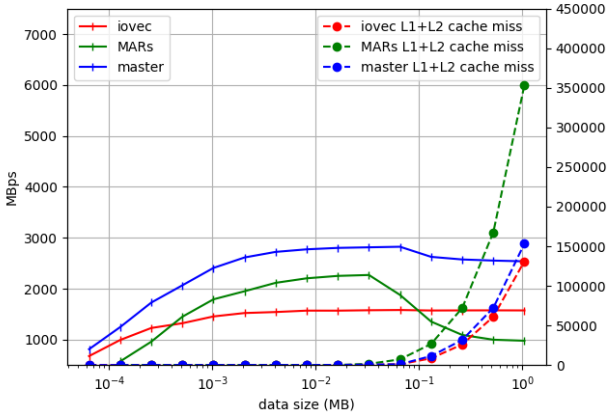
(b) 6_1_1 Datatype



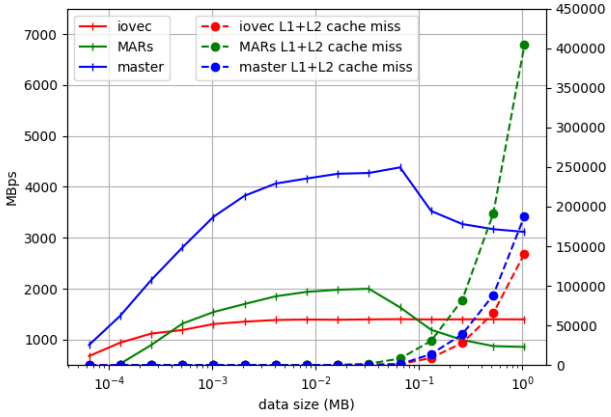
(c) 5_1_1_1 Datatype



(d) 4_1_1_1_1 Datatype



(e) 2_1_1_1_1_1_1 Datatype



(f) 1_1_1_1_1_1_1_1 Datatype

Fig. 10: Master vs. IOVEC vs. MARs, Performance vs. Cache Miss

comparing to IOVEC in benchmark. Since ”bookkeeping” instructions happen for every element in masters datatype description, master will always have one more ”bookkeeping” loop since the description always ends with Loop_end.

The IOVEC performance drops as more doubles are split into the next vacant cache line because there are more elements in the description, thus, IOVEC pack function has to take more ”bookkeeping” when going through the description. Since master compacts the regular patterned element (one double in each cache line), there are always three elements (first element,

second compacted element and Loop_end) in the description. The extreme example is the last datatype where there is one double in each cache line. Masters datatype description has been optimized into only two elements, the data element and Loop_end element. Calculating the instructions for each method:

$$Master = 3.3 \times 8 + 25.27 \times 2 = 76.94\, ins/ddt \quad (5)$$

$$IOVEC = 8.4 \times 8 + 9.71 \times 8 = 144.88\, ins/ddt \quad (6)$$

Using equation 5 and equation 6, we could calculate instructions and performance difference, master has only 0.53X of instructions comparing to IOVEC while having 2.6X more bandwidth. The calculations showed the number of instructions could affect performance and it is proven from Fig. 9. The performance for pack can be calculated using the table I as pack performance has linear correlation to the number of instructions when datatype description cannot be compacted.

Although the number of instructions could also affect performance greatly, there are other factors when access pattern changes.

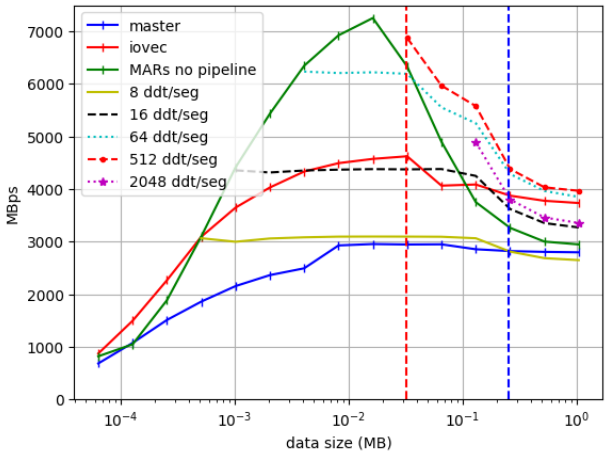
MARs are designed to have the lowest instructions when datatype could not be optimized since it only parses datatype description once, the cost for "bookkeeping" is extremely low. Due to the access pattern, MARs could reach far into the memory which will cause unused data to be evicted. From Fig. 10a, even though cache misses have overwhelmed MARs performance, it still manages to have higher bandwidth the master for large sizes. Because 7_1 only occupied two cache lines, the number of cache misses will be lower comparing to datatype that occupies more cache lines. From Fig. 10, as datatype starts to occupy more cache lines, the number of cache misses skyrockets which overwhelms the performance. To further optimize MARs, pipeline method is introduced to preserve both the low instruction count and to minimize the cache misses.

C. MARs Segmentation/Pipeline

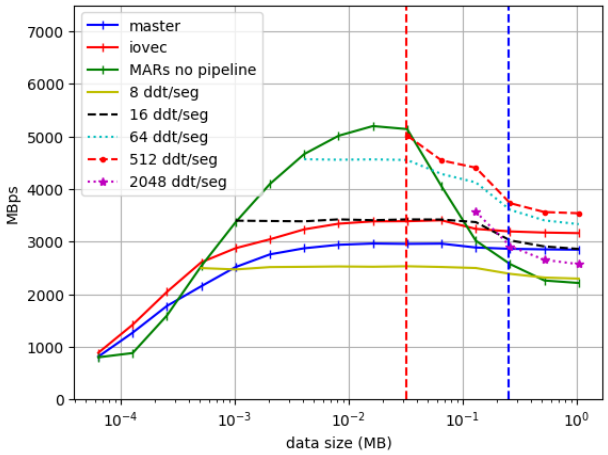
Fig. 11 present how MARs with segmentation/pipeline work. The vertical lines indicate where each cache level ends. The performance for MARs peaks within L1 cache size, then it keeps decreasing during L2 and it will drop to a plateau in L3. MARs outperform all other packing methods when the data size is kept within L1 cache because all data would be able to stay in cache without being evicted. And given MARs have the lowest instruction count, MARs would top performance during L1. Because we have not yet found a solution to calculate the best pipeline/segment size, we picked 8, 16, 64, 512 and 2048 datatypes per segment for the benchmark. Pipeline versions could have up 2.6X performance comparing to non-pipelined version. Although MARs with pipeline might not perform as well as master when datatype could be compacted, there is potential to keep performance at peak for data sizes ranging from L2 to L3.

VIII. Conclusion

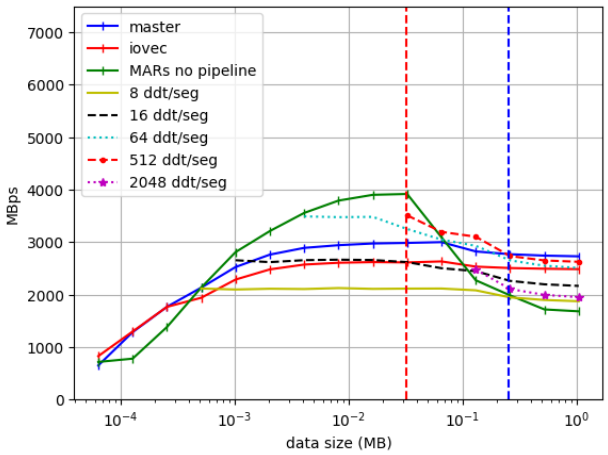
With our IOVEC datatype description, we have guaranteed the minimum calls to MEMCPY and minimum instructions when datatype cannot be compacted. One of the next steps is to autotune the datatype engine to be able to choose master or IOVEC method based on the instruction table. Adding MARs could also boost the performance for multi-element datatypes. With IOVEC implementation, we have found that parsing datatype description is not free. While MARs have avoided parsing datatype description successfully, we need to find the perfect pipeline strategy to bring out the best performance for all datatypes.



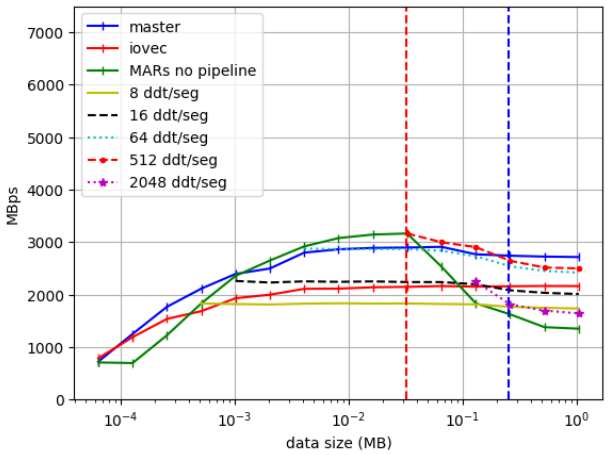
(a) 7_1 Datatype



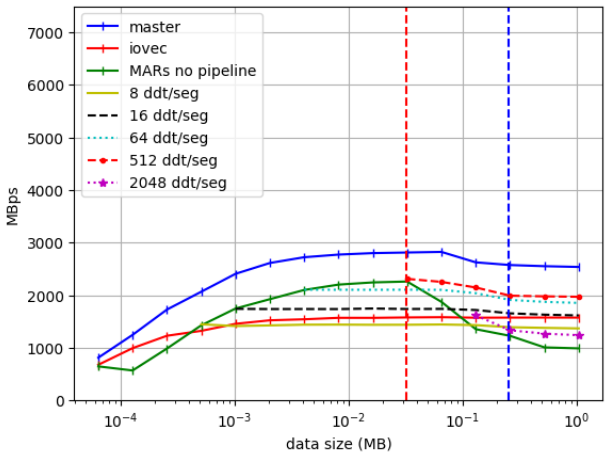
(b) 6_1_1 Datatype



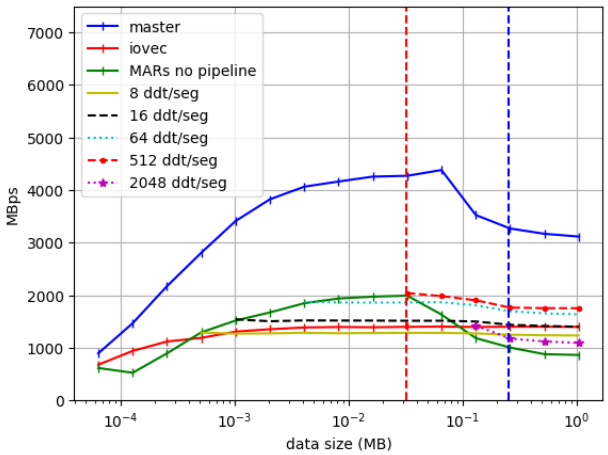
(c) 5_1_1_1 Datatype



(d) 4_1_1_1_1 Datatype



(e) 2_1_1_1_1_1_1 Datatype



(f) 1_1_1_1_1_1_1_1 Datatype

Fig. 11: Master vs. IOVEC vs. MARs pipeline

REFERENCES

[1] InfiniBand Trade Association et al. Infinibandtm architecture specification. *http://www. infinibandta. org*, 2004.

[2] Mohammadreza Bayatpour, Nick Sarkauskas, Hari Subramoni, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. Bluesmpi: Efficient mpi non-blocking alltoall offloading designs on modern bluefield smart nics. In *International Conference on High Performance Computing*, pages 18–37. Springer, 2021.

[3] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *The international journal of high performance computing applications*, 14(3):189–204, 2000.

[4] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[5] William Gropp, Ewing Lusk, and Deborah Swider. Improving the performance of mpi derived datatypes. In *Proceedings of the Third MPI Developers and Users Conference*, pages 25–30. Citeseer, 1999.

[6] Jahanzeb Maqbool Hashmi, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. Falcon: Efficient designs for zero-copy mpi datatype processing on emerging architectures. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 355–364. IEEE, 2019.

[7] Jahanzeb Maqbool Hashmi, Ching-Hsiang Chu, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. Falcon-x: Zero-copy mpi derived datatype processing on modern cpu and gpu architectures. *Journal of Parallel and Distributed Computing*, 144:1–13, 2020.

[8] Martin Kalany and Jesper Larsson Träff. Efficient, optimal mpi datatype reconstruction for vector and index types. In *Proceedings of the 22nd European MPI Users' Group Meeting*, pages 1–10, 2015.

[9] Mingzhe Li, Hari Subramoni, Khaled Hamidouche, Xiaoyi Lu, and Dhabaleswar K Panda. High performance mpi datatype support with user-mode memory registration: Challenges, designs, and benefits. In *2015 IEEE International Conference on Cluster Computing*, pages 226–235. IEEE, 2015.

[10] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[11] Robert Ross, Daniel Nurmi, Albert Cheng, and Michael Zingale. A case study in application i/o on linux clusters. In *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 59–59. IEEE, 2001.

[12] Gopalakrishnan Santhanaraman, Jiesheng Wu, and Dhabaleswar K Panda. Zero-copy mpi derived datatype communication over infiniband. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 47–56. Springer, 2004.

[13] E Schikuta. Message-passing-interface-forum: Mpi: A message-passing interface standard. *Techn. Ber., University of Tennessee, Knoxville, Tennesee*, page 30, 1994.

[14] Timo Schneider, Robert Gerstenberger, and Torsten Hoefler. Micro-applications for communication data access patterns and mpi datatypes. In *European MPI Users' Group Meeting*, pages 121–131. Springer, 2012.

[15] Xian-He Sun et al. Improving the performance of mpi derived datatypes by optimizing memory-access cost. In *2003 Proceedings IEEE International Conference on Cluster Computing*, pages 412–419. IEEE, 2003.

[16] Kaushik Kandadi Suresh, Bharath Ramesh, Chen Chun Chen, Seyedeh Mahdieh Ghazimirsaeed, Mohammadreza Bayatpour, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. Layout-aware hardware-assisted designs for derived data types in mpi. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 302–311. IEEE, 2021.

[17] Monika ten Bruggencate and Duncan Roweth. Dmapp-an api for one-sided program models on baker systems. In *Cray User Group Conference*, 2010.

[18] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: Efficient handling of mpi derived datatypes. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 109–116. Springer, 1999.

[19] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. High performance implementation of mpi derived datatype communication over infiniband. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 14. IEEE, 2004.

[20] Qingqing Xiong, Purushotham V Bangalore, Anthony Skjellum, and Martin Herbordt. Mpi derived datatypes: Performance and portability issues. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–10, 2018.

[21] Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. Using advanced vector extensions avx-512 for mpi reductions. In *27th European MPI Users' Group Meeting*, pages 1–10, 2020.

[22] Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. Using long vector extensions for mpi reductions. *Parallel Computing*, 109:102871, 2022.

[23] Dong Zhong, Pavel Shamis, Qinglei Cao, George Bosilca, Shinji Sumimoto, Kenichi Miura, and Jack Dongarra. Using arm scalable vector extension to optimize open mpi. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 222–231. IEEE, 2020.