# 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 communication performance. In this paper, we focus on Open MPI datatype engine, which uses pipelining 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 current 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. Although we see 1.2X and sometimes 3.2X performance from IOVEC, the performance is mainly based on the instruction count and current Open MPI datatype description could outperform IOVEC for optimized datatypes. By introducing MARs, we have minimized instruction count. After applying pipeline/segmentation strategy on MARs, we have seen a steady #X to #X performance benefit.

# Introduction

Exacale-computing is closer than ever before because of the rapid evolution of the hardware. While Moore’s Law [19] 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)[1] has always been the standard communication model in all parallel applications. Several key features in MPI provide user convenience while maintaining a high performance. 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 consisted 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 [3]. Users are more likely to pack non-contiguous data themselves. This clearly defeats the purpose of having a datatype engine. Thus, we use user manual pack as a standard to improve upon current datatype engine.

Numerous researches have gone to reduce/erase the overhead for datatype engine during communication, such as User-mode Memory Registration (UMR) [10], zero-copy [8][9] approach by using highly efficient InfiniBand [4]. It is far easier to achieve better performance by simply applying modern hardware capabilities, for example, applying hardware specific vector extensions [11][12][13]. 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 [21], we decided to revamp the datatype description which results in trivial and stable optimal form.

We took the challenge to revamp the datatype description and the pack function to reach as close as to theoretical peak communication speed. Based on our work, we have concluded that the bottleneck for datatype engine is limited mainly by the amount of instructions CPU issues/completed. And by introducing various datatype descriptions and pack functions, we could minimize the instructions issued/completed. With that, we have seen a stable 1.2X to 1.5X performance boost and, in some cases, a 3.2X.

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.

# Related Work

Improving MPI datatype performance has long been an effort to both improve communication performance and user convenience. However, because of the high overhead datatype engines come with [1][2], it has prevented scientific applications from adopting MPI datatype, even though efforts have tried to revamp datatype description [16][17][18]. 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 [3].

A lot of research efforts have gone into improving communication using datatype engine and utilizing underlying hardware capabilities. With interconnects such as Infiniband (IB)[4], 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)[7][8][9]. 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) [10], which can read/write multiple non-contiguous blocks of data through RDMA using Scatter-Gather-Lists (SGL). With similar concept, zero-copy strategy [5][6][8][9] 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 [14][15]. While smart NICs isn’t as common, utilizing Intel’s Advanced Vector Extensions (AVX) [11][12] and Arm’s Scalable Vector Extension (SVE) [12][13] is able to improve the time-to-solution in predefined MPI reduction operations. The same vectorization would also help the datatype engine to increase efficiency and close the gap to peak performance.

In our work, we took a step further to make a trivial solution to work on all platforms.

# Background

## Open MPI Datatype Engine

Using Open MPI datatype feature, a datatype description can be exported using function *ompi\_datatype\_dump (MPI\_Datatype ddt).* Figure #1 is an example of a typical datatype in Open MPI. The top half of the figure 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 better data transfer performance.

A close-up of a document

Description automatically generated with low confidence

Figure #1

As figure #1 shows, an example of a typical datatype in Open MPI is consisted of three elements, 1) Loop\_start, 2) Data and 3) Loop\_end. Loop\_start and Loop\_end indicate what and how many times elements in datatype description will be repeated. While each element is defined like a vector type. Thus, Open MPI Datatype ensures optimal performance when datatype is a regular patterned memory layout.

Text

Description automatically generated with low confidence

Figure #2

Such datatype description ensures minimal storage space for some datatypes, however, if a datatype is neither regular nor available for optimize, the performance could suffer since current datatype description uses extra memory to describe element.

There is also optimization inconsistency. Figure #1 and Figure #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 utilized 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.

## 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. However, Open MPI datatype engine statically groups the first occurrence of repeated elements. It will not optimize datatypes based on storage or memory. Different datatype description will also mean different performance.

Because it would take polynomial time to find the optimal form for datatype [21], we have decided to revamp datatype description which will always result in the same optimal form during optimization.

## Pack/Unpack Functions

There are several pack/unpack functions built inside Open MPI to deal with different scenarios. *opal\_generic\_simple\_pack\_function* and *opal\_generic\_simple\_unpack\_function* are the most common ones. Both functions have the exact same routine but with opposite data flows. *opal\_generic\_simple\_pack\_function* puts non-contiguous data into a single contiguous buffer, and *opal\_generic\_simple\_unpack\_function* does the opposite.

Open MPI optimized communication performance using pipelining. Thus, 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 that needs to be packed/unpacked, total data size that needs to be moved, and etc. It also uses stack to keep the positioning 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.

# Motivations

Open MPI datatype engine supports communication layer for data handling. It alleviates the workload from user to hard-code data movements and ensures a high 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) + Pack Overhead(Tpo) + Network Latency(L) + Unpack Overhead(Tuo) + Unpack(TU)

Because the network latency is scaled linearly in respect to the size of contiguous buffer, Open MPI uses pipelining 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

Smaller Pack(Tp) + Smaller Pack Overhead(Tpo) + Smaller Network Latency(L) + Smaller Unpack Overhead(Tuo) + Smaller Unpack(TU)

A screenshot of a computer

Description automatically generated with medium confidence

Because network latency is hidden and pipelining with small segments wouldn’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.

We concluded that the performance for pack functions is mostly based on the number of instructions and we will reason with graphs and numbers in performance section. While all pack functions are a loop around MEMCPY, more loops will result in more MEMCPYs and more instructions, thus, less performance. Because some datatypes would cause master to optimize into a less “optimized” form, which will cause datatype engine to do more loops and MEMCPYs, we need a datatype description that could always be reduced into the optimal form. In addition, we are looking to reduce even more instructions by rearranging the pack sequence in pack function to get more performance.

# Implementations

## IOVEC Datatype Representation

Our first goal is to find a datatype representation that could result in consistent datatype optimization and stable pack/unpack performance. An array of IOVECs is an optimal 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 of the data and the length of that data. 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).

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. From our experiments, we found that there are two possible looping technique when packing with master. The first one is looping around the entire datatype description, and second one is looping around partial repeating datatype description. Since the master uses vector-like attributes to describe every element, it could use a small loop to loop around elements which the count is larger than one. 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. Going through one big loop around description will require a few “bookkeeping”. 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.

## IOVEC Memory Access Rearrangements (MARs)

Since excess instructions are generated from looping the datatype description, we could further reduce the number of instructions by rearranging the pack sequence. Figure # is an example of how MARs is done. Instead of looping through every element in the datatype description multiple times, MARs pack every element for multiple times and move on to the next element.

Text

Description automatically generated

MARs succeed in reducing the number of instructions, however, the performance will suffer when user/packed buffer expand beyond cache. Because the access pattern MARs do, it 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 size.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

We used 6\_1\_1 datatype as an example in Figure # since 7\_1 datatype might not fully show the full characteristic of the method. Figure # A only packs 8 datatypes while Figure # B packs 2048 datatypes. There are small “boosts” in A for both master and MARs. These “boosts” 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 “boosts.” In large sizes, Figure # B, the resulting “boosts” are even greater since all the data has been pushed into the next level cache or even the main memory. Thus, the overhead of reaching the starting address of the next element is much larger comparing to smaller sizes.

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 the faster access of the data could be the result of hardware prefetch.

## MARs Segmentation/Pipeline

With large data size that reaches beyond cache size, MARs would always 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.

Chart, line chart

Description automatically generated

Figure #7

Figure #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 and from platform to platform. 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 to maximize performance for all datatypes. We will present performance with a few segmentation/pipeline sizes in performance section.

# Benchmark Datatypes

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

Description automatically generated

Figure # visualizes the 7 datatypes. They all have the same data size for one datatype. Starting 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 extent, 512 bytes, so that the length of user 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 the master’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.

# Experiment Setup

All our experiments were conducted on a single node which is consists of Intel Xeon CPU E5-2650 v3 with 20 physical cores. Each CPU 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 current master, version 5.0.0.

# Performance

## Inconsistent Optimization

From previous datatype example in section 3, 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 Figure #, 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.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

## IOVEC vs. MARs vs. Current Master with PAPI[22]

Chart

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart, scatter chart

Description automatically generated

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

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

We calculated the instructions per 8-byte copy and instructions of “bookkeeping” per loop (looping the datatype description) to see where master and 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 loop of description and what an 8-byte copy will take. We can calculate the performance difference just by using the numbers from table above. 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, it only takes two times of instructions of bookkeeping. Calculating the instructions per packing datatype:

Master = 3.3x8 + 25.27x3 = 102.21 ins/ddt

IOVEC = 8.4x8 + 9.71x2 = 86.62 ins/ddt

While master has calculated 1.18X more instructions, it has 0.75X the bandwidth comparing to IOVEC. Since bookkeeping instructions happen for every element in master’s datatype description, master will always have one more bookkeeping loop since the description always ends with Loop\_end. As the number of elements grows, the overhead for master could increase if the data size does not grow.

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 more elements. Since master optimize the regular pattern, the double in each cache line after the first element, there are always three elements in the description. The extreme example is the last datatype where there is one double in each cache line. Master’s datatype description has been optimized into only two elements, the data element and Loop\_end element. Calculating the instructions for each method:

Master = 3.3x8 + 25.27x2 = 76.94 ins/ddt

IOVEC = 8.4x8 + 9.71x8 = 144.88 ins/ddt

For this datatype, master has only 0.53X of instructions comparing to IOVEC while having 2.6X more bandwidth.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart

Description automatically generatedChart

Description automatically generated

The calculations showed the number of instructions could affect performance and it is proven from Figure #. When access patterns are the same, increasing the number of instructions will cause decrease in performance. 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 # A, 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 #, 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.

## MARs Segmentation/Pipeline

Figure # and Figure # present how MARs with segmentation/pipeline work. The vertical lines indicate where each cache level ends. The bandwidth speed 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 1.29X to 2X performance comparing to non-pipelined version. Although MARs with pipeline might not perform as well when datatype could be optimized, there is potential to kept performance at peak for data sizes ranging from L2 to L3.

# Conclusion

With our IOVEC datatype description, we have successfully removed the datatype optimization instability. One of the next steps is to autotune the datatype engine to be able to choose master or IOVEC method based on what’s the percentage of the datatype has been optimized. Adding MARs could also boost the performance for multi-element datatypes that could not be optimized. And 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.