**Introduction**

**Related Work**

**Background**

***Open MPI Datatype Engine:***

Datatypes in Open MPI are consisted of basic datatypes, such as int, double, float, etc. When putting basic datatypes together, the combination is called derived datatype. The current Open MPI datatype engine will optimize the storage space for a derived datatype during commit time. First, let’s examine how a datatype looks like in Open MPI.

A close-up of a document

Description automatically generated with low confidence

Figure #1

Using Open MPI datatype feature, a datatype description can be exported using function *ompi\_datatype\_dump (MPI\_Datatype ddt).* An arbitrary datatype description will look like the description in Figure #1. 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.

As figure #1 shows, an example of an arbitrary datatype in Open MPI is consisted of three elements, 1) Loop\_start, 2) Data and 3) Loop\_end. Loop\_start and Loop\_end indicates what data elements and how many times the elements will be repeated. Each element in the datatype is described as follow:

Struct ddt\_elem\_desc{

Ddt\_elem\_id\_description common;

Uint32\_t blocklen;

Size\_t count;

Ptrdiff\_t extent;

Ptrdiff\_t disp;

};

Typedef struct ddt\_elem\_desc ddt\_elem\_desc\_t;

Text

Description automatically generated with low confidence

Figure #2

At first glance, it seems it is excessive to describe one single element with extent and count, but this is an optimal description when the given datatype is a regular pattern memory layout. Any regular pattern datatype, vector-like patterns, can be described using one ddt\_elem\_desc\_t, just like the example from Figure #2. Within the datatype, when there’s one element describes as a full vector type, in this case, 9 counts of block length of 40 bytes with extent of 44 bytes, using such datatype description, datatype engine has fully minimized the memory to store the datatype.

If looking closely, the datatype in Figure #2 is the same as Figure #1. Since both are defined using different methods, the genetics of the datatype (memory layouts) are the same. The difference of the optimized description comes from the optimization process during commit time.

***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. Although the optimization process does well in most cases, we did find scenarios, like the ones above, result in different optimized descriptions.

// reason of optimization being different and paper to cite optimization could use n^3 time

Before we examine the performance of the Open MPI datatype engine, we should first closely look at how datatype description is used in pack/unpack functions.

***Pack/Unpack Functions***

There are several pack/unpack functions built inside Open MPI to deal with different systems. We choose and examine the most common one, *opal\_generic\_simple\_pack\_function* and *opal\_generic\_simple\_unpack\_function*. Both functions have the exact same routine but with opposite data flows.

Struct dt\_stack\_t {

Int32\_t index;

Int16\_t type;

Int16\_t padding;

Size\_t count;

Ptrdiff\_t disp;

};

Typedef struct dt\_stack\_t dt\_stack\_t;

In Open MPI, in order to maintain the leisure of achieving pipelining, it 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. Along with these information, it also uses stack to keep the positioning within the user buffer. As figure #3 explains what a stack is consisted of.

* Index: correspond to ddt\_elem\_desc\_t, keeps track of which element in the datatype description the convertor is at
* Count: number of the whole datatype left to be done
* Disp: the current displacement in respect to the start of 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 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 first process, 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

Since network latency is hidden and pipelining with small segments wouldn’t fully utilize the bandwidth of the network, with the right pipeline size, the bottleneck for the communication lies within pack/unpack, thus, increasing the pack/unpack performance is what we are interested in.

Knowing there is a bottleneck for every datatype in turns of how fast the non-contiguous data can be put together to form the contiguous buffer, we can easily calculate the ideal/maximum bandwidth using the sparsity (data size / datatype extent) with the theoretical peak bandwidth of the system as for a given datatype:

Theoretical peak bandwidth = data size / datatype extent \* theoretical peak bandwidth

While improving pack/unpack, we did observe that inconsistent in datatype optimization also leads to inconsistent in pack/unpack performance. Thus, we also aim to revamp datatype description and optimization process to have consistent performance across all scenarios.

**IOVEC Datatype Representation**

Our first goal is to find a datatype representation that could result in consistent optimization form and pack/unpack performance. A flattened, easy-to-maintain datatype description that is consisted of an array of IOVECs is the perfect choice.

IOVEC datatype representation is created during commit time optimization. Each IOVEC corresponding to a data space and the IOVECs, as one datatype, contain all the memory locations and sizes.

During the commit optimization, IOVEC representation is created by traversing the Open MPI datatype description. It will merge any two or more IOVECs that can be combined to into one single contiguous element. Since less MEMCPYs equals to better performance, while current datatype optimization could not, IOVEC representation always guarantees the minimal number of MEMCPYs.

Because IOVEC representation is flattened and expands storage space based on the number of non-contiguous elements, using current pack/unpack methods (going through description X times), the performance would easily take a toll when IOVEC description is huge in memory and pack/unpack has to access the description over and over. Such implementation isn’t cache optimal since the description could take out a chunk of cache and leave less for the actual data. Thus, we also revamp pack function to accommodate IOVEC description.

**IOVEC Memory Access Rearrangements (MARs)**

Our goal to revamp pack function is to minimize the IOVEC description access. By doing so, we would free up cache that is required to keep IOVEC description in reach. In the meantime, we could also maximize the cache usage. Given today’s memory hierarchy, the time for data to travel from one cache level to another is almost nothing comparing to the time from main memory to highest level of cache. We proposed Memory Access Rearrangements (MARs) method in replace to today’s datatype engine’s sequential packing order.

In a typical MARs operation, instead of packing elements by elements in datatype description, MARs only look at one element (E) at a time and pack all X (user defined) times of E and then move on to the next element in the description.

Timeline

Description automatically generated//change to trash\_tlb ddt

Figure #4

Figure #4 demonstrate how MARs comparing to today’s packing sequence. Giving a datatype that has two elements, a double starts at displacement 0 and a double starts at displacement 16 with extent of 64 bytes. With the packed buffer being the same, the number represents the packing order. Such access rearrangements have two major benefits, 1) it greatly reduces burden on cache to have constant access to datatype, it only need to access datatype once, 2) most datatypes with user given high count number have irregular memory access pattern, hardware could not optimize with hardware prefetch, however, with MARs, the irregular access pattern is altered into multiple regular access patterns and the number of multiple access patterns is correlated to the number of elements in datatype description.

There is a hidden and need-to-be-exploit benefit of using MARs. Because of the access pattern MARs provide, all the other elements within the same granularity have been brought into cache before MARs move on to them.

Chart, line chart

Description automatically generated

Figure #5

To visualize this phenomenon, after each MEMCPY in pack function, we call rdtscp and replace the destination memory in MEMCPY with the timestamp. Figure #5 shows how the timestamp increments with each MEMCPY. For current pack function in Open MPI, the linear time incrementation corresponds to the sequential access of data. Since each access brought the next 3 elements into cache, after two accesses with pack, the next three accesses are already in cache, thus, much faster accesses than the first two. With MARs, the accesses for all X first elements are not in cache and are 8 pages apart, thus, much longer accesses for the first X accesses, but after that all the data has been brought into cache and creates faster accesses. We could fully utilize the benefit of using MARs by increasing the number of datatypes in each segment.

However, infinitely increasing the segment sizes could lead to huge performance loss.

Chart, line chart

Description automatically generated

Figure #6

As Figure #6 shows, when disregarding the segment size, the access time for each element with MARs is far from optimal. This is because of the size limitation on cache. When MARs keep reaching deep into the main memory, there is only limited number of cache lines the cache can hold. After that, the cache starts to evict data back into main memory. When MARs move on the next element in description, it will be evicted and will take time to be fetched back into cache again. Such back and forth movement will cause huge disruption and delay in performance. To solve this issue, we purpose to use segmentation or pipeline within MARS.

**MARs Segmentation/Pipeline**

Because of the access pattern from MARs, every first access in datatype results in bringing more data within the granularity into the cache. The idea to use segmentation/pipeline on MARs is trying to keep every possible reuse of data inside cache and to keep data eviction at minimum.

Chart, line chart

Description automatically generated

Figure #7

Figure #7 is an example that MARs perform 20% better in a large count scenario with correct segment sizes. However, the segmentation/pipeline sizes can vary from datatype to datatype and from platform to platform. After observing a few of datatype performance using MARs, we suspect the best performances using MARs segmentation/pipeline 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 are yet to find a perfect calculation/equation to maximize all performance. We will present performance with a few segmentation/pipeline sizes in performance section.

**Performance**

***IOVEC vs. Current Datatype Description***

//bad optimization from current datatype engine

//comparing vs. iovec

Chart, line chart

Description automatically generated

***IOVEC Gather vs. IOVEC vs. Current Datatype Description***

***IOVEC Gather Pipeline vs. IOVEC vs. Current Datatype Description***

**Conclusion**