Introduction:

Motivation:

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. Thus, 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 side 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. Reducing the time in pack/unpack becomes the focus to increase performance.

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

[a demonstration of point-to-point pipelining]

While improving pack/unpack, we noticed an inconsistent performance in the current datatype engine when performing pack/unpack on the same memory layouts but are defined using different methods. Thus, we compared the current datatype engine with a trivial datatype that is consisted with an array of iovecs (a starting address and a length).

Although vector type, a regular patterned memory layout, is the most common datatype, irregular patterned datatypes are also widely used by applications. Due to the irregular memory accesses, most hardware prefetch will have a hard time gaining performance, thus, using software prefetch could be one of the solutions to improve bandwidth.

Thus, our motivations to improve Open MPI datatype engine are the following: 1) Improve pack/unpack 2) Investigate the inconsistency in which Open MPI Datatype Engine by comparing with a consistent and trivial memory layout representation 3) Exploit prefetch usage

Datatypes:

We use three types of datatype type that are used by most application.

Vector datatypes:

The most common datatypes used across all application is the vector datatypes. Thus, testing and optimizing the vector datatypes is crucial for better performance.

In our benchmarks, we use *vector(512, 1, 8, double, &ddt)* as one of our datatypes. We vary the stride to create several more datatypes and see how sparsity will impact the performance.

Structure datatypes:

A close up of a newspaper

Description automatically generated

Figure#

A picture containing text

Description automatically generated

Figure#1

The Open MPI datatype engine saves memory by putting a loop around data that needs to be repeat, so there is no need to expand the description.

Both figures in #1 describe the same memory layout as shown in Figure #2.

[figure of memory layout drawing]

However, the first is described by using

[description]

Second description is describe by using

[description]

Matrix datatypes:

In our benchmark, we simulate two types of matrix datatype, a diagonal matrix and a sparse irregular pattern matrix. We increase the size of the matrix, nxn as n is the width of the matrix, for each data point.

In order to simulate a sparse matrix, we use rand() function to generate small element in each row and limit the size of the element to be less than 1/25 of the n. We use seed to make sure that the matrix we generate will be the same across tests.

Text, application

Description automatically generatedText

Description automatically generated

The above pictures are the functions to declare diagonal and sparse matrix.

Open MPI Datatype Benchmark:

Under test/datatype directory in Open MPI directory, there is a benchmark solely for single node bandwidth performance. The performance in this paper, pack and unpack, is ran 200 times for each data point and is been reported from this benchmark.

ICON:

ICON is a European weather prediction application.

Open MPI Datatype Engine:

Intro:

The current Open MPI datatype engine uses minimal space to describe a memory layout (derived datatype) defined by the user.

An arbitrary datatype description will look like one of the descriptions in Figure #1 as the top part of the figure is the description given by the user and the bottom part is the optimization description generated by the datatype engine during commit time.

As picture # shows, an example of an arbitrary datatype is consisted of three elements,

1. Loop\_start
2. Data
3. Loop\_end

In which, 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;

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 3, within the datatype, there’s one element describes as a full vector type. Using such description, datatype engine has fully minimized the memory to store the datatype.

Data elements in current datatype engine are made exactly like the vector type on the user layer with one extra displacement attribute. 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;

It uses stack approach, where the stack acts as a bookkeeping for the datatype engine. As figure #3 explains what a stack is consisted of and what each attribute is used for.

* 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 pack/unpack function will examine each element inside datatype description and update stack while moving along.

Pros:

Since Open MPI datatype engine uses minimal space to store datatype description, that means there is more space for data to get into the cache with fewer evict and change cache line with the memory, thus resulting a more efficient use of the cache.

When datatype description is only composed with data elements, walking through the data description is very efficient for pack/unpack functions.

Cons:

From our experiment, which we will discuss in the performance section, we found that optimization plays a crucial step for pack/unpack performance. As two different ways that describes the same memory layouts could result in two different internal datatype description. Pack/unpack with two different description results in vastly different performance. The minimal memory that stores description comes back to haunt us, because optimization doesn’t always give an optimal description. This inconsistency in description will cause up to 5X performance difference.

Iovec Datatype Representation:

Intro:

A trivial, flattened datatype description is a perfect comparison to the current datatype description since the advantages and disadvantages are obvious. Thus, comparing the performance between the iovec datatype representation and current Open MPI datatype description will give us a clear understanding of what the current datatype description is lacking.

Iovec datatype representation is created at datatype optimization (commit). Iovec expands all the memory layout and describe each element using an iovec (an address and a length).

Struct iovec {

Void \*iov\_base;

Size\_t iov\_len;

};

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 become one single contiguous element. Reducing number of memcpy function calls is crucial to improve performance and, if the optimization is done correctly, Open MPI datatype description should expect to have exact same number of memcpy as iovec representation during pack/unpack.

Pros:

Unlike current Open MPI datatype engine, iovec datatype representation performs stable when same memory layout is defined using different methods because the description is always come out the same. Iovec datatype representations excels at datatypes that are filled with short length elements along with large gaps, typically a sparse matrix. Reason for that is that pack/unpack for iovec representation does not have as many branches as the current pack/unpack function does.

Cons:

Huge memory storage kept in cache cause less space for moving actually data, thus resulting in low performance for simple regular memory layouts.

Prefetch:

We try to compensate the large size of iovec representation with software prefetch. Since the datatype description is been flattened, adding prefetch in iovec is much easier than adding in master. Because of the complexity in current datatype description, adding prefetch requires a bookkeeping in traversing the description ahead of doing memcpy.

We use *\_\_builtin\_prefetch* function in our pack/unpack implementation. Behind the function is the prefetch hints that are described in the *Intel User Manual*. The prefetch hints gives memory an idea where to get data next and bring data to what cache level. In our benchmark, we found that bring data to all cache levels performs the best. But that can vary from application to application.

Since it takes time between issuing prefetch and data arriving in the cache, we accumulate prefetches at the start to create that time for data to arrive. We calculate the physical addresses using iovec description and issue prefetch based on the addresses calculated. We only issue one prefetch on each element within the iovec description no matter how long the element is, because we assume the hardware prefetch will kick in for large contiguous data, since software prefetch will always perform worse than hardware prefetch in this area.

After the initial prefetches, we issue only one prefetch before each memcpy, since there are numbers of prefetched addresses memcpy has not done yet. We refer to the number of addresses memcpy has not done as prefetch distance.

We found a particular prefetch distance that performs the best in our experiment. But it could vary from system to system based on the memory clock speed.

Prefetch:

We use the Intel prefetch intrinsic instruction to apply prefetch. There are few parameters we can test with

1. Locality of the data (which cache level to bring to)
   1. Non temporal
   2. 1, bring only to L3
   3. 2, bring to L3 and L2
   4. 3, bring to all cache levels
2. Read or write
3. Physical address of the data

While it takes hundreds of cycles for data to reach cache from memory, we have to do something else during this gap in order to fully take advantage of the prefetch. Luckily, the datatype engine does a lot of bookkeeping for the convertor to do pipelining. Before we reconfigured the code, bookkeeping is usually done after each memcpy. Now, we move bookkeeping in between prefetch and memcpy.

Since Open MPI datatype representation is a very compact form, it requires massive work to traverse the datatype and to apply prefetch. However, with the flattened datatype representation, iovec representation, it is much easier to traverse the datatype and apply prefetch. We applied prefetch in two ways

1. naively prefetch one element (iovec) ahead of memcpy
2. stack a few prefetches and then do memcpy on these elements

From our observation, prefetch in pack doesn’t benefit as much as the prefetch in unpack. We saw as much as 3X bandwidth improvement in unpack in a number of cases.

System setup:

Saturn:

ICON:

Performance:

We start calculating the theoretical performance for non-contiguous datatype. We first examine the vector(512, 1, 8, double) datatype. Since we take one double out of every cache line in this test, the theoretical bandwidth equals to 8/64 of the contiguous datatype.

Graphical user interface, chart

Description automatically generated

As we observed in the graph above, the performance for this particular datatype is indeed as good as the theoretical bandwidth.

A picture containing diagram

Description automatically generatedChart

Description automatically generatedGraphical user interface

Description automatically generatedChart

Description automatically generatedChart, line chart, histogram

Description automatically generatedChart, histogram

Description automatically generatedGraphical user interface, chart, line chart

Description automatically generated

We then vary the stride. We observe that

Chart

Description automatically generated

The performance in the two graphs above uses the datatype we showed in structure datatype. Both datatype describes the same memory layout, but the performance for master is vastly different, while iovec representation performs stable.

The reason behind this vastly different performance is because the commit time optimization did not fully optimize the datatype description. As shown in the structure datatype section, the first description optimized to three elements where the vector datatype inside the datatype has the last element combine to the first element in the vector. The second optimized version did not do as exactly, thus resulting in one more memcpy in every vector element.

TLB miss:

We examined there is a huge performance drop with unpack. Our guess to this phenomenon is that the hardware is doing page walk, which the TLB table does not have the translation between physical addresses and virtual addresses and it looks up in and updates the page table. We determined this might be the most time-consuming procedure for memory to take.

[performance graph for tlb miss]

From the picture #tlb, the tlb misses increase exponentially when the buffer and data size grow. Even though this phenomenon is observed on both pack and unpack sides, the performance for pack does not waiver at all.

One of the explanations is that, Intel Xeon CPUs are using write-allocate cache, which means that data are physically brought into cache, wrote, and flush back to the memory. Unlike pack function, brought non-contiguous data into cache and flush contiguous packed buffer back into memory, which fully utilize the write back bandwidth, unpack takes significant bandwidth loss due to its non-contiguous memory flush.

\*\*\*\*\*\*\*\*\*\*\*\*

One of the solutions suggested by *Empirical study of the Intel Xeon Phi CPU* is to use streaming store, which does not bring data into cache. Data will be stored in the buffer inside memory controller and sent to its destination, thus avoiding unnecessary data movement and cache pollution.

\*\*

In fact, when looking at all the graphs combined (stride 2 4 8 64 128), bandwidth for pack can always reach the same level, about 2200 MB/s. But the peak of the bandwidth for unpack keeps dropping when gap between data elements is increased. Such linear decay of performance has us suspect the prefetch for pack and unpack may perform differently, such that prefetch for pack may operate on cache line level, bring cache line that only has data in it, while prefetch for unpack could bring a couple more lines into the cache since it has no idea why the data is been read. That’s why we implemented a naïve prefetch strategy to test if the prefetch in unpack is indeed messed up.

Prefetch performance:

Table

Description automatically generatedTable

Description automatically generated

We tested prefetch will a range of parameters

1. locality (0, 1, 2, 3)
2. number of prefetches stacked (1-36)

From our result, prefetch in writing data is much more effective than accessing data. More specifically, pack access non-contiguous data and put them into a contiguous buffer, flushing this contiguous buffer will fully utilize the bandwidth. However, in unpack, access contiguous data and put them into a non-contiguous buffer greatly decreases the performance since we are now flushing a percentage of the cache line into the memory and the bandwidth becomes the percentage of the theoretical peak.

Stack a few prefetches with locality 1, bring data only to L3 cache, proves to be the best case. Since data takes about 700 hundred cycles for data to travel from memory to L3 cache, and it takes far less cycles from L3 to L2 and from L3 to L1, usually less than 100 cycles. This is why saving hundreds of cycles could improve unpack performance by 3X.

One of the reason pack doesn’t show any bandwidth improvement is because, pack fully utilizes the memory channel by flushing the contiguous packed buffer. While flushing in unpack requires physical address to be present in the cache since write-allocate cache is the policy.

Because the graph is presented in log scale in packed buffer size, the fact that prefetch does not work in small buffer can be easily ignored. From the first graph, the buffer size is much smaller and since datatype is much sparser. Thus, applying prefetch becomes very situational when it comes to the size of input buffer.

ICON Application:

Graphical user interface, text, application

Description automatically generated

We tested the previous version on ICON application, which runs on 24 nodes and does halo exchange among the nodes.

We observed that iovec datatype representation performs a little worse comparing to the current master’s datatype engine. Since iovec representation acquires more memory to hold the memory layout, it takes more cache from data movement and calculation, thus the performance drop. When prefetch is applied in unpack, the run time is decreased by 10% on data exchange and 2% on total application time.

Future work:

The current datatype engine at this point could not fully optimize the datatype description to combine elements that are contiguous. The result of having one more memcpy results in far less performance than the iovec datatype description. We could combine both descriptions and determine which description to use based on memory layout.

The current datatype description could be further optimized by combining elements. However, it is not as simple since the datatype could have the first and last element