Introduction:

(increasing transistors -> increasing cores -> data transfer -> memory wall)

(efficient prefetch data with regular pattern, compiler)

(talk about contiguous vs. non-contiguous, where this paper mainly focuses on non-contiguous. Contiguous has a hook function to deal with already)

(talked about cache system, and do haven’t tried no cache system)

(fetch vs. threw awat)

With the doubt whether Moore’s law is still alive or not, the computation ability for HPC system is at an all-time high. Memory wall is now the performance ceiling for most applications. MPI (message passing interface) for the past decade has become the facto for data communication.

Background on MPI Derived Datatypes:

(easier for user, avoid tedious packing loop written by user)

(derived datatype vs. primitive datatype, construction)

(describe datatype)(commit time optimization)(use datatype)

Related work:

Motivation:

(talk about send/recv pipeline, faster pack/unpack result in faster point-to-point, collective)

Maximizing the bandwidth for communication is one of the sole purposes for MPI. And derived datatype is one of the ways to relieve programmer’s burden on coding while delivers great performance.

In this paper, our work is mainly used in point-to-point communication. The current OpenMPI uses pipelining to achieve high bandwidth.

From our standpoint, derived datatypes can be categorized into two groups, regular and irregular shaped/patterned. The current implementation of derived datatype for OpenMPI is as optimized as possible for regular patterned datatype, e.g. vector type, as the current version uses vector-like structure to define datatypes. While the modern hardware can do a fantastic job prefetching the data with regular patterns, but not for irregular shaped/patterned datatype.

While we let hardware does its prefetch, the maximum bandwidth in theory is calculated as follows:

Max. Bandwidth = cache line prefetched / cache line threw away

From our tests, OpenMPI derived datatype is not doing as good as the regular ones since hardware has a hard time doing prefetch for irregular patterns. Besides hardware, the original derived datatype implementation has already introduced using different pack/unpack functions based on datatype and current environment. But OpenMPI has not split pack/unpack to as many optimized functions as other MPI implementations.

As Qingqing Xiong et al. {#} discusses the performance differences among OpenMPI, intelMPI, mpich and mvapich2, OpenMPI mainly does pack/unpack through only one function (opal\_generic\_simple\_pack), while the others, such as MPICH, uses multiple pack/unpack calls to specialize based on datatype.

The urge to increase the performance for derived datatype let us to categorize datatype patterns. When PMPI and function hooks for pack/unpack strategies were introduced, the decision-making procedure is added to the critical path before any pack/unpack starts. We added a datatype pack/unpack hybrid strategy into the critical path. Iovec datatype representation will be decided whether or not to generate at commit time. And whether or not to use iovec depends on if iovec representation is generated.

Implementation:

Current OpenMPI datatype internal representation:

struct ddt\_elem\_desc {

ddt\_elem\_id\_description common;

uint32\_t blocklen;

size\_t count;

ptrdiff\_t extent;

ptrdiff\_t disp;

}

The current OpenMPI datatype engine uses a common(a predefined datatype, e.g. char, int, float, etc.), blocklen, count, extent and displacement to describe each element within the datatype. However, during commit time, if there is a regular pattern, the current representation will ignore displacement and will see datatype as a vector type since there will be no irregularity. As a result, the current version is optimal for regular patterns.

For irregular patterns, the datatype engine will define each element using ddt\_elem\_desc structure, while adding displacement in respect to the first element of the datatype.

In case datatype repeats a portion of itself, the datatype engine will then add a loop start to the front and loop end to the back. Loop start includes information such as the how many times to repeat, what element to repeat and the extent of the loop.

iovec implementation:

struct iovec {

void \*iov\_base;

size\_t iov\_len;

}

Iovec datatype representation consists with an array of iovec structures, which contain the starting addresses and the corresponding sizes. Each iovec represents an element within the datatype. One complete iovec datatype represents the complete memory layout and contains the unrolled version of any repeated portion of the datatype.

The iovec representation is generated during commit time after the original OpenMPI internal datatype representation has been optimized.

Switch for OpenMPI datatype and iovec, OpenMPI commit time optimization:

(minimize the memcpy by combining)

OpenMPI optimizes internal datatype representation at commit time, where it stores both the original version and the optimized version. For original version, the optimization will combine elements that are contiguous together(??).

For both original and optimized version, the datatype stores the maximum number of elements in the description array and the number of used elements in the description array. When optimization does happen, e.g. combined some elements together, the number for used elements will be reduced. From all the tests we conducted, we categorize datatype to be regular when the number is less than or equal to 3, and irregular otherwise.

Experiment setup:

A:

(spec for Saturn node A)

B: benchmark description

OpenMPI saves construction of datatype description in MPI layer, we write this information to a file and we can reconstruct the datatype by reading the file. Using this method, we extracted every datatype committed during commit time in ICON benchmark and reconstructed the same datatypes into OpenMPI datatype benchmark. We then do pack/unpack operation on one node, intra-node and inter-node communication using point-to-point communication.

From the ICON benchmark, we extracted 589 datatypes and we examined every pack/unpack and communication performance between iovec datatype representation and latest master. We also examined the performance with OpenMPI benchmark’s datatypes.

Inside OpenMPI datatype benchmark, we create a total number of 10 datatypes that we think would be common for most applications.

1. Contiguous
2. Indexed gap
3. Optimized indexed gap
4. Constant indexed gap
5. Optimized constant indexed gap
6. Struct constant gap resized
7. Diagonal matrix
8. Upper triangle matrix
9. Lower triangle matrix
10. Randomized indexed type

Randomized datatype: We use rand() function to generate block length for each cache line.

block length \* unit size <= cache line size

And we generate a datatype with extent that equals to a page size. We keep the seed the same to make sure we generate the same datatype across the tests.

As we categorize datatype into two groups, we worked with ICON(ICOsahedral Nonhydrostatic) benchmark developed by European weather forecast. We see a dramatic performance increase for data exchange (60% decrease of time), resulting about 9% decrease in time for the total application.

Performance analysis:

(fewer branches, identical cache miss)

Future work:

Conclusion: