Implementing Fast and Reusable Datatype Processing

Robert Ross, Neill Miller, and William Gropp

Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne IL 60439, USA,
{rross, neillm, gropp}@mcs.anl.gov

Abstract. Methods for describing structured data are a key aid in application development. The MPI standard defines a system for creating "MPI Types" at run time and using these types when passing messages, performing RMA operations, and accessing data in files. Similar capabilities are available in other middleware as well. Unfortunately many implementations perform poorly when processing these structured data types. This leads application developers to avoid these components entirely, instead performing any necessary processing of data by hand.
In this paper we describe an internal representation of types that aids in maintaining the highest possible performance during processing. The performance of this system, used in the MPICH2 implementation, is compared to well-written manual processing routines and other available MPI implementations. We show that performance for most tested types is comparable to manual processing. Finally we point to additional opportunities for optimization and other software where this implementation could be leveraged.

1 Introduction

Many middleware packages now provide mechanisms for building *datatypes*, descriptions of structured data, and using these types in other operations, such as passing messages, performing remote memory access (RMA), and I/O. These systems typically allow regularity of structured data to be described, leading to concise descriptions of sometimes complicated layouts.

The problem with many implementations of these systems is that they perform poorly [9]. Because of this application programmers are forced to avoid the systems altogether and instead perform this processing manually in the application code. A common instance of this is manually *packing* structured data (placing noncontiguous data into a contiguous region for efficiently sending in a message) and then manually copying the data back into structured form on the other side.

Obviously no implementors providing mechanisms for structured data description and manipulation intend these systems to unusably slow. Further, it is not the case that the MPI datatype specification precludes high performance implementations. Several groups have investigated possibilities for improving MPI datatype processing performance [11, 5] with some success.

This work describes the implementation of a generic datatype processing system and its use in the context of a portable, high-performance MPI implementation, MPICH2. The goal of this work is to provide a high-performance implementation of datatype processing that will allow application programmers to leverage the power of datatypes without sacrificing performance. While we will show the use of this system in the context of MPI datatypes, the implementation is built in such a way that it can be leveraged in other environments as well by providing a simple but complete representation for structured types, a mechanism for efficiently performing arbitrary operations on data types (not just packing and unpacking), and support for partial processing of types. In Section 2 we discuss the design of our datatype processing system. In Section 3 we examine the performance of our system using a number of synthetic benchmarks. In Section 4 we discuss related work in the area of datatype processing, and in Section 5 we present conclusions and point to future areas of study.

2 Design

Previous work presented a taxonomy of MPI types and a methodology for representing these in a concise way [5]. It further discussed the use of an explicit stack-based approach for processing that avoids recursive calls seen in simple implementations. However, only preliminary work was done in this direction at that time. This effort builds on this previous effort, implementing many of the ideas and extending and generalizing the approach for use in additional roles. There are three characteristics of this implementation that we will consider in more detail:

- simplified type representation (over MPI types)
- support for partial processing of types
- separation of type parsing from action to perform on data

We will discuss each of these in turn in upcoming sections.

2.1 Representing Types: Dataloops

We describe types by combining a concise set of descriptors that we call *dataloops*. Dataloops can be of five types: contig, vector, blockindexed, indexed, and struct [5]. These five types allow us to capture the maximum amount of regularity possible, keeping our representation concise. At the same time these are sufficient to describe the entire range of MPI types. Simplifying the set of descriptors aids greatly in implementing support for fast datatype processing because it reduces the number of cases that our processing code must handle. Further, we maintain the type's extent in this representation (a general concept) while eliminating any future need for the MPI-specific LB and UB values. This simplification has the added benefit of allowing us to process resized types with no additional overhead in our representation.

For MPI we create the dataloop representation of the type within the MPI type creation calls (e.g. `MPI_Type_vector`), building on the dataloop representation of the input type. We also take this opportunity to perform optimizations based on the input type and new constructor, such as coalescing of adjacent regions in indexed types. Figure 1 shows the dataloop representation of the type used in the Flash I/O datatype test described in Section 3. It consists of a sequence of vector dataloops. We will denote the bottom dataloop the *leaf* dataloop; optimizing processing of leaf dataloops is key to performance, and we will discuss how we accomplish this momentarily.

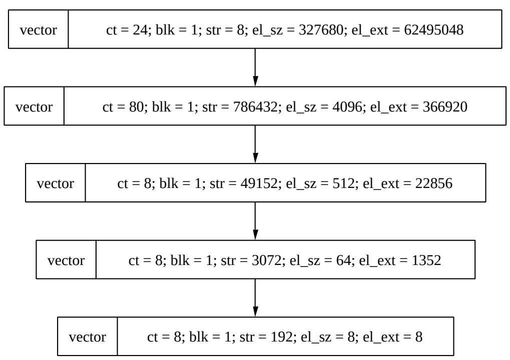

Fig. 1. Dataloop representation of Flash test type

2.2 Partial Processing: Segments

In many cases it is necessary to break up processing of a type into a number of steps. For example, when sending a message we may need to copy data into a contiguous buffer for sending. If the message is large, we may have to break the data into chunks in order to limit our use of available memory. We call this action of operating on data in chunks *partial processing*.

For the purpose of partial processing, we need some structure to maintain state between calls to datatype processing routines. We call this structure a `segment`. Segments are allocated before processing and freed afterwards. Segments contain the stack used to process the type and state describing the last position processed. With this information processing can be broken into multiple steps or performed out-of-order.

2.3 Actions on Data

So far we haven't discussed actions that might be performed as we are processing a type other than alluding to copying data to and from contiguous buffers. However, there are actually a variety of actions that one might perform.

For MPICH2 the use of type processing occurs in three locations. The most obvious of these is the `MPI_Pack` and `MPI_Unpack` routines that allow users to pack a type into a contiguous buffer. These routines are implemented trivially with the approach described above and do not require partial processing. The second use is in the point-to-point messaging code itself. In this code we must in some cases pack or unpack from contiguous buffers as well. However, these buffers are of limited size in order to bound the memory requirements of the implementation, so partial processing is needed. In other cases we can leverage the `readv` and `writev` calls to avoid the need for data copy. In these cases we instead convert the type to a list of $(offset, length)$ pairs to be passed to these calls. In this case too we must partial process, as these calls will only accept a limited number of these pairs. Finally the ROMIO [10] MPI-IO component of MPICH2 requires similar $(offset, length)$ pairs for use with noncontiguous file views. The sizes of these types do not match the sizes of the types for `readv` and `writev` calls on all platforms, so separate routines are required.

From this it is obvious that just in the context of MPI there are a number of potential uses of this component and operations that might be performed. We thus separate the code that understands how to process the type from the *action* that will be performed on pieces of the type. For a given action to perform on types, we need functions (or possibly macros for performance reasons) that understand each of the leaf dataloop types. By providing code that can process entire leaf dataloops we avoid processing the type as a collection of contiguous pieces, maintaining performance.

2.4 Implementation Details

Currently our system is implemented using a core "loop manipulation" function which calls action-specific functions to process leaf dataloops. For each action a set of functions are implemented that understand contiguous, vector, indexed, block indexed, and struct loops. The ability to process entire leaf dataloops in this manner leads directly to the performance seen in the following section. We have investigated conversion of these routines into macros in order to eliminate function call overhead, but at this time this overhead does not appear to be significant. The current implementation supports only homogenous systems.

Optimizations of the loop representation are currently applied at two points. First, when types are built we perform optimizations such as conversion of struct types into indexed types (for homogenous systems) and coalescing of contiguous indexed regions. Second, at the time the segment is created (the first step in a `MPI_Pack`) we examine the type and the count used in the segment. This provides an opportunity to optimize for cases such as a count of a contiguous type, converting this into a larger contiguous type or a vector (depending on the extent of the base type). We will see the results of this in the Struct Array and Struct Vector tests in the following section.

In addition to loop optimizations we are able to preload our stack at segment creation time. This is possible because of conversion of structs into indexed loops; there is never more than one leaf dataloop in our homogeneous type representation.

3 Benchmarking

When choosing benchmarks for this work, we first examined the SKaMPI benchmark and datatype testing performed with this tool [7, 8]. While this tool did seem appropriate for testing of type processing within a single MPI implementation, it does so in the context of point to point message passing or collectives. Because we wanted to look at a number of different implementations, and were concerned solely with type processing, we desired tests that isolated datatype processing. We implemented a collection of synthetic tests for this purpose. These tests compare the `MPI_Pack` and `MPI_Unpack` routines to hand-coded routines that manually pack and unpack data, effectively isolating type processing from other aspects of the MPI implementation. Each test begins by allocating memory, initializing the data region, and creating a MPI type describing the region. Next a set of iterations are performed using `MPI_Pack` and `MPI_Unpack` in order to get a rough estimate of the time of runs. Using this data we then calculate a number of iterations to time, and execute those iterations. The process is repeated for our manual packing and unpacking routines.

The *Contig* and *Struct Array* set of tests both test performance operating on contiguous data. The Contig tests are simple contiguous cases using `MPI_INT`, `MPI_FLOAT`, and `MPI_DOUBLE` types. A contiguous type of 1048576 elements is created, and a count of 1 is passed to the `MPI_Pack` and `MPI_Unpack` routines. We expect in these tests that most MPI implementations will perform competitively with the manual routines. The Struct Array test creates a single struct type, and an array of 64K of these are manipulated. The structure consists of two integers, followed by a 64-byte char array, two doubles, and a float. The structure is defined to be packed, so there are no gaps between elements or between structures in the array. This provides an opportunity for implementations to optimize, and we would expect performance to be virtually identical to the Contig test.

The *Vector* and *Struct Vector* tests both examine performance when operating on a vector of 1048576 basic types with a stride of 2 types (i.e. accessing every other type). In the Vector tests, we build a vector type and pack and unpack using a count of 1. In the Struct Vector tests we first build a struct type using `MPI_LB` and `MPI_UB` to increase the extent, then we pack and unpack with a count of 1048576. This is a useful method of operating on strided data when the number of elements might change from call to call. Examining the relative performance of these two tests allows us to see the importance of optimizations that are applied after the type is created based on the count passed to MPI calls.

The *Indexed* set of tests use an indexed type with a fixed, regular pattern. Every block in the indexed type consists of a single element (of type `MPI_INT`, `MPI_FLOAT`, or `MPI_DOUBLE`, depending on the particular test run.) There are 1048576 such blocks in the type. As shown in Figure 2, some pieces in the pattern are adjacent, allowing for some underlying optimization of the region repre-

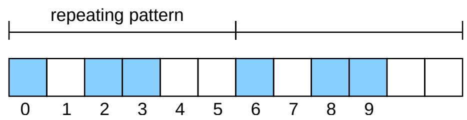

Fig. 2. Pattern of elements in Indexed tests

sentation. A particularly clever implementation could refactor this as a vector of an indexed base type, but we do not expect any current implementations to discover this, and ours does not. These tests showcase the importance of handling indexed leaf dataloops efficiently.

t

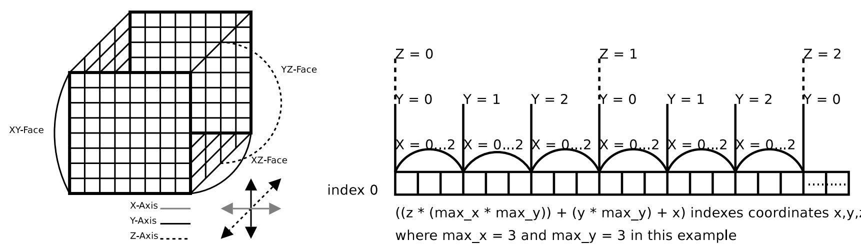

Fig. 3. Data layout in 3D Face tests

The *3D Face* set of tests pull entire faces off of a 3D cube of elements (Figure 3, described in Appendix E of [4]). Element types are varied between `MPI_INT`, `MPI_FLOAT`, and `MPI_DOUBLE` types. The 3D cube is 256 elements on a side. We separate the manipulation of sides of the cube in order to observe the performance impact of locality and contiguity. The XY side of the cube has the the greatest locality, while the YZ side of the cube has the least locality.

The *Flash I/O* test examines performance when operating on the data kept in core by the Flash Astrophysics application. The Flash code is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [3].

Figure 4 depicts the in-memory representation of data used by Flash. The data consists of 80 three-dimensional blocks of data. Each block consists of a 8x8x8 block of elements surrounded by a guard cell region four elements deep on each side. Each element consists of 24 variables, each a `MPI_DOUBLE`. For postprocessing reasons the Flash code writes data out by variable, while variables are interleaved in memory during computation. In the Flash I/O test we describe the data in terms of this by-variable organization used in writing checkpoints. This leads to a very noncontiguous access pattern across memory. Further, this t

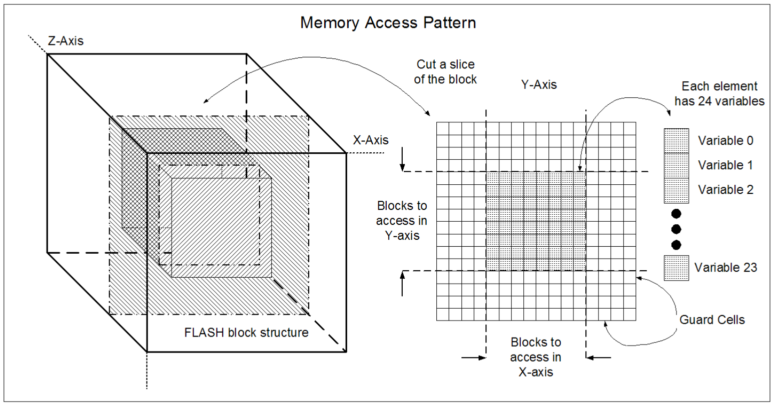

Fig. 4. Data layout in Flash test

is the most deeply nested type tested in this suite, showcasing the need for nonrecursive approaches to processing.

3.1 Performance Results

Measurements on the IA32 platform were performed on a dual-processor 2.0 GHz Xeon system with 1 GByte of main memory. The machine has 512 Kbytes of L2 cache. Results for the Stream benchmark [6] show a copy rate of 1230.77 Mbytes/sec on this machine. This gives us an upper bound for main memory manipulation, although in specific cases we will see cache effects.

Table 1 shows the performance of our synthetic benchmarks for manual packing, MPICH2, MPICH1, and the LAM MPI implementations. Also listed are the size and extent of the data region manipulated (in MBytes). Results with `MPI_INT` types were removed; they were virtually identical to `MPI_FLOAT` results. LAM 6.5.9, MPICH 1.2.5-1a, and a CVS version of MPICH2 (as of May 7, 2003) were used in testing. The `CFLAGS` used to compile test programs and MPI implementations were "-O6 -DNDEBUG -fomit-frame-pointer -ffast-math -fexpensive-optimizations."

Performance for the Contig tests are as expected. The data extent is large enough that caching isn't a factor, and in all cases performance is very close to the peak identified by the Stream benchmark. The Struct Array test shows similar results for all but LAM. This indicates that the LAM implementation does not detect that this is really a large contiguous region, resulting in significant performance degradation.

The Vector and Struct Vector tests show that the same pattern can be processed very differently depending on how it is described to the datatype processing system. Our implementation detects the vector pattern in the Struct Vector

Table 1. Comparison of processing performance (MB/sec)

Test	Manual	MPICH2	MPICH1	LAM	size (MB)	extent (MB)
Contig (FLOAT)	1156.37	1124.04	1136.48	1002.38	4.00	4.00
Contig (DOUBLE)	1132.26	1126.22	1125.05	1010.81	8.00	8.00
Struct Array	1055.02	1131.39	1131.28	512.72	5.75	5.75
Vector (FLOAT)	754.37	753.81	744.42	491.31	4.00	8.00
Vector (DOUBLE)	747.98	743.88	744.81	632.77	8.00	16.00
Struct Vector (FLOAT)	746.04	750.76	36.57	141.60	4.00	8.00
Struct Vector (DOUBLE)	747.31	743.70	72.81	252.34	8.00	16.00
Indexed (FLOAT)	654.35	401.26	82.79	122.85	2.00	4.00
Indexed (DOUBLE)	696.59	530.29	161.52	204.43	4.00	8.00
3D, XY Face (FLOAT)	1807.91	1798.52	1754.45	1139.04	0.25	0.25
3D, XZ Face (FLOAT)	1244.52	1237.68	1210.53	992.80	0.25	63.75
3D, YZ Face (FLOAT)	111.85	112.06	112.15	64.22	0.25	63.99
3D, XY Face (DOUBLE)	1149.84	1133.86	1132.43	1011.11	0.50	0.50
3D, XZ Face (DOUBLE)	1213.10	1201.54	1157.93	969.46	0.50	127.50
3D, YZ Face (DOUBLE)	206.41	206.39	201.82	103.24	0.50	127.99
Flash I/O (DOUBLE)	245.60	212.55	215.80	159.63	7.50	59.60

tests at segment creation time, converting the loop into a vector and processing in the same way. We see that this type of optimization is not applied in the other two implementations, leading to poor performance.

The Indexed tests showcase the importance of handling indexed leaf dataloops well. Prior to the addition of an indexed leaf processing function in the MPICH2 implementation, performance was approximately 25% of the current value. With the inclusion of action-specific functions that handle indexed dataloops we are able to attain 60% of the manual processing rate at the smaller data type sizes and 76% for `MPI_DOUBLE`, while the other implementations lag behind significantly. We intend to spend additional time examining the code path for this case, as we would expect to more closely match manual packing performance.

The 3D Face test using `MPI_FLOAT` types shows two interesting effects. First, because in the XY case the extent of the data is such that it all fits in cache, performance actually exceeds Stream benchmark performance for this case. In the YZ case data elements are strided but laid out in groups of 256; this is more than adequate to maintain high performance. In the last (YZ) case we see the effect of very strided data; we are only accessing one type for every 256 in a row, and performance is less than 10% of peak for manual routines and all the tested MPI implementations.

The Flash I/O type processing test shows that while we are able to maintain the performance of a manual packing implementation, performance overall is quite bad. Just as with the YZ 3D Face test, this type, with its many nested loops, provides opportunities for optimization that we do not currently exploit.

As such, this type will serve as a test case for application of additional optimizations.

4 Related Work

The work by Träff et al. on datatypes is in many ways similar to this approach [11]. They also consider derived types as a tree. However, their leaf nodes are always basic (primitive) types, and they allow branches to occur at indexed types, while we maintain a single child in these cases. Further they have rules for each type of MPI constructor, rather than converting to a more general, yet simpler, set of component types. At the same time, they leverage some loop reordering optimizations that are not used in this work.

5 Conclusions and Future Work

This work presents a concise abstraction for representing types coupled with a well-engineered algorithm for processing this representation. With this system we are able to maintain a high percentage of manual processing performance under a wide variety of situations. This system is integrated into MPICH2, ensuring that many scientists will have the opportunity to leverage this work.

Besides comparisons on additional architectures, the next step in this work is to move beyond matching manual processing performance. Techniques such as loop reordering [11] and optimizing based on memory access characteristics [1] offer opportunities for improved performance and should match well to the dataloop representation used in our system. If we think of dataloops as a representation of a program for processing types, runtime code generation is another avenue for performance gains.

Finally we are looking at other applications of this component in scientific computing software. For example, we are currently incorporating this work into PVFS [2] as a type processing component. By passing serialized dataloops as I/O descriptions we obtain a concise I/O request and can leverage this high performance type processing code for processing at the server. Similarly this component could be used to replace the datatype processing system in HDF5 that was the source of performance problems in a previous study [9].

References

1. S. Byna, W. Gropp, X. Sun, and R. Thakur. Improving the performance of mpi derived datatypes by optimizing memory-access cost. Technical Report Preprint ANL/MCS-P1045-0403, Mathematics and Computer Science Division, Argonne National Laboratory, April 2003.
2. P. Carns, W. Ligon, R. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

3. B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.
4. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.
5. W. Gropp, E. Lusk, and D. Swider. Improving the performance of MPI derived datatypes. In Anthony Skjellum, Purushotham V. Bangalore, and Yoginder S. Dandass, editors, *Proceedings of the Third MPI Developer's and User's Conference*, pages 25–30. MPI Software Technology Press, 1999.
6. J. McCalpin. Sustainable memory bandwidth in current high performance computers. Technical report, Advanced Systems Division, Silicon Graphics, Inc., Revised to October 12, 1995.
7. R. Reussner, P. Sanders, L. Prechelt, and M Müller. SKaMPI: A detailed, accurate MPI benchmark. In Vassuk Alexandrov and Jack Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer, 1998.
8. R. Reussner, J. Träff, and G. Hunzelmann. A benchmark for MPI derived datatypes. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 10–17, September 2000.
9. R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A case study in application I/O on linux clusters. In *Proceedings of SC2001*, November 2001.
10. R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.
11. J. Träff, R. Hempel, H. Ritzdoff, and F. Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In J. J. Dongarra, E. Luque, and Tomas Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 1697 in Lecture Notes in Computer Science, pages 109–116, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. Springer-Verlag.