Performance Expectations and Guidelines for MPI Derived Datatypes

William Gropp,[1] Torsten Hoefler,[1] Rajeev Thakur,[2] Jesper Larsson Träff[3]

[1] University of Illinois, Urbana, IL 61801, USA, {wgropp,htor}@illinois.edu
[2] Argonne National Laboratory, Argonne, IL 60439, USA, thakur@mcs.anl.gov
[3] University of Vienna, Austria, traff@par.univie.ac.at

Abstract. MPI's derived datatypes provide a powerful mechanism for concisely describing arbitrary, noncontiguous layouts of user data for use in MPI communication. This paper formulates *self-consistent performance guidelines* for derived datatypes. Such guidelines make performance expectations for derived datatypes explicit and suggest relevant optimizations to MPI implementers. We also identify self-consistent guidelines that are too strict to enforce, because they entail NP-hard optimization problems. Enforced self-consistent guidelines assure the user that certain manual datatype optimizations cannot lead to performance improvements, which in turn contributes to performance portability between MPI implementations that behave in accordance with the guidelines. We present results of tests with several MPI implementations, which indicate that many of them violate the guidelines.

1 Introduction

Self-consistent performance requirements for MPI are an invitation to MPI implementers to ensure consistent performance among interrelated functionalities. In addition to guarding against unpleasant performance surprises, such guidelines can support performance portability among MPI implementations: They avoid the need for hand optimizations to compensate for unsatisfactory performance of specific functions in specific contexts, systems, or MPI implementations, which could also be counterproductive on other systems, implementations, or circumstances. Self-consistent MPI performance guidelines can be construed as performance expectations for application programmers, recommendations for MPI implementers, or even requirements that would be desirable to fulfill.

Performance expectations for MPI communication functions were formulated in [11] and for MPI-IO in [3]. This paper proposes performance expectations and guidelines for the derived datatype mechanism in MPI. We identify a number of guidelines for the performance of the MPI datatype mechanism that an MPI implementation should meet so as to enable and encourage performance-portable programming. We also present the results of simple experiments to validate MPI implementations. Our measurement results for several implementations indicate that many of them violate the performance guidelines, which can lead to unpleasant surprises for users. This result should serve as an encouragement for

further research and implementation work on improving the handling of MPI derived datatypes.

1.1 Related Work

The derived datatype mechanism is one of the central concepts of the MPI standard. It separates communication operations from the structure of data being communicated [7, Chapter 4] and is vital for the MPI-IO specification for distributed file structures [7, Chapter 13]. The generality and expressive power of the derived datatype mechanism is one feature that sets MPI apart from other interfaces with similar intentions and scope. Describing complex, local data layouts by derived datatypes makes it possible for the MPI implementation to handle such structures by efficient packing and unpacking mechanisms that interact closely with (pipelined) communication algorithms or by exploiting available hardware support for noncontiguous data communication. Achieving similar or better effects by hand is tedious and in many cases non-portable performance wise. The ultimate goal of an efficient MPI implementation of the datatype mechanism is, in some loose sense, never to be worse than what the application programmer can achieve by hand packing/unpacking and communicating the packed buffers. This paper is an attempt toward defining this goal more precisely.

Providing efficient implementations of MPI datatypes has therefore been the focus of several groups [2, 4, 9, 10, 12], and much progress has been achieved, although there are still situations where datatype performance is less satisfactory as we discuss in Section 7. The use of MPI datatypes to provide better performance within applications has been explored in several studies, e.g., [1, 5, 6]. Benchmarks for datatypes focusing on the complexity of the different constructors were defined in [8]. We are not aware of any work directly addressing performance expectations and guidelines for MPI datatypes.

2 Derived Datatype Constructors

MPI derived datatypes can be thought of as concise descriptions of layouts of data in process memory. MPI derived datatypes are described in [7, Chapter 4], which the reader should consult for precise definitions (constructors, type signatures and maps). There are five main MPI functions for constructing new datatypes out of old ones. Let n be the value of the count argument supplied to the constructors. We omit all arguments that are not essential for the discussion.

1. MPI_Type_contiguous(n,T): n successive blocks of type T, denoted as contig(n,-T)
2. MPI_Type_vector(n,m,T): n strided blocks of m instances of type T, denoted as vector(n, m, T)
3. MPI_Type_create_indexed_block(n,m,T): n blocks of m instances of type T each with own displacement, denoted as index_block(n, m, T)

4. MPI_Type_indexed(n,m_n,T): n blocks of type T each with own count m_i ($0 \leq i < n$) and displacement, denoted as index(n, m_n, T). The total number of blocks is $\sum_{i=0}^{n} m_i$.
5. MPI_Type_create_struct(n, m_n, T_n): n blocks of types T_i each with own count m_i ($0 \leq i < n$) and displacement, denoted as struct(n, m_n, T_n)

The constructors can be applied recursively, so T can be a primitive, basic datatype or a previously constructed, derived datatype. In addition, there are convenience functions for creating datatypes representing subarrays and distributed arrays. Another special constructor makes it possible to change the extent of a (derived) datatype, which is important when using nested type constructors, see for instance [1].

A first benchmark measures the basic communication performance for strided layouts described by each of the five constructors. The benchmark can be parameterized in type T (here we use only the basic `MPI_DOUBLE` type), stride s and number of blocks n. Communication performance is measured by point-to-point ping-pong communication in order to be able to focus as far as possible on the datatype component.

Benchmark 1 *The* same strided layout *of a n repetitions of type T with stride s described by the five different type constructors. Communication time for the five types as a function of number of repetitions n.*

On a given architecture the layout of the data elements in memory eventually determines the performance of communication operations involving the derived datatype. Alignment of the basic datatypes might be good or bad, the basic datatypes may be blocked, or strided or otherwise regularly spaced which might be advantageous for some architectures, there might be special hardware that can exploit certain structures in the layout, etc. For these reasons it is not possible to pose *absolute* performance requirements on MPI operations involving datatypes. A natural user expectation, however, would be that hardware support, e.g., for strided memory access or communication, bulk transfers etc. be utilized wherever possible by the MPI library.

However, what can be done, and this is the key point, is to relate the many different ways that a *given* type map can be described by the derived datatype mechanism (e.g., as in Benchmark 1). A self-consistent MPI performance guideline for datatypes would state that the performance of an MPI communication operation with some datatype T describing the *given* (non-contiguous) layout should be no worse than the same operation with any other datatype T' that describes the *same* layout. Otherwise, the user could improve performance by possibly tedious and non-portable redefinitions of the datatype description of the application data.

Another user expectation which we discuss in more detail in Section 5 is that MPI operations with datatypes perform at least as well as manually packing the data into a contiguous buffer before the MPI operation.

Each of the type constructors describes a sequence of n blocks. The contiguous and vector types do so with constant extra information, the indexed

block requires one array for the displacements, the indexed one extra array for the block lengths, and the structure yet one more array for the datatypes of the blocks. We formalize this by associating the *penalties* $0, 0, 1n, 2n, 3n$, plus some constant $O(1)$ overhead, with the five constructors, accordingly. The total penalty of a datatype is defined recursively as the penalty of the top-level constructor times the penalty of the subtype, or, for the struct constructor, the sum of the penalties of the subtypes. The intuition is that in order to process a layout described by a datatype with some total penalty h, $\Omega(h)$ operations are required just to parse the type map. The strictest, self-consistent performance guideline then says that the performance of an MPI function with datatype T should be no worse than the performance with a datatype T' that has minimal total (considering possibly recursive type specifications) penalty.

3 Trivial Expectations

We will use the following notation to express performance expectations and guidelines: $\mathsf{MPI\_A}(n, T_{A'}) \preceq \mathsf{MPI\_B}(n, T_{B'})$ shall mean that MPI function A operating on n elements as described by datatype $T_{A'}$ is not slower than MPI function B with type $T_{B'}$ for almost all n, all other things, including in particular the type map of the datatypes $T_{A'}$ and $T_{B'}$, being equal.

Expectation (1) comes directly from the MPI standard which states that a call to a communication function with a count and a datatype argument is functionally equivalent to the same call where the count and the datatype have been encapsulated in a contiguous datatype [7, Section 4.1.11]. It would be sensible to expect that these two equivalent call forms would also perform similarly:

$$\mathsf{MPI\_A}(1, \mathsf{contig}(n, T)) \approx \mathsf{MPI\_A}(n, T) \tag{1}$$

This should hold for any type T. Exhaustive verification is of course not possible, but a simple benchmark will indicate whether the expectation is reasonably fulfilled.

Benchmark 2 *Six basetypes $T_0 = T$, $T_1 = \mathsf{contig}(k, T)$, $T_2 = \mathsf{vector}(k, T)$, $T_3 = \mathsf{index\_block}(k, T)$, $T_4 = \mathsf{index}(k, T)$, and $T_5 = \mathsf{struct}(k, T)$ with repetition count n, versus $T_0, \ldots, T_5$ encapsulated in a contiguous type with count n; for T_0 repetition count is kn so as to have the same number of element in all six cases. Communication performance with the two versions for the six types.*

This benchmark measures both sides of Equation 1. It should be extended with more irregular layouts, e.g., from the following benchmarks.

The five constructors are able to express more and more irregular layouts of data in memory, but at an increasing penalty (more parameters for displacements/indices, block lengths, and datatypes). For a given, regularly strided layout that can be expressed with all five constructors, it is therefore natural to

expect, for any function $f(n) \leq n$ (e.g., constant), that

$$\begin{aligned}\mathsf{MPI\_A}(n, \mathsf{contig}(f(n), T)) &\preceq \mathsf{MPI\_A}(n, \mathsf{vector}(f(n), T)) \\ &\preceq \mathsf{MPI\_A}(n, \mathsf{index\_block}(f(n), T)) \\ &\preceq \mathsf{MPI\_A}(n, \mathsf{index}(f(n), T)) \\ &\preceq \mathsf{MPI\_A}(n, \mathsf{struct}(f(n), T)) \end{aligned} \tag{2}$$

Guideline (2) says that if a given layout can be expressed with fewer parameters (less penalty), then for any MPI function A this should perform no worse, ideally better, than expressing this layout with a datatype constructor with higher penalty. It is a (trivial) self-consistent performance requirement: if the higher penalty datatype constructor would perform better in some context, the user could obtain this performance by manually rewriting his code to use the better performing constructor. With Benchmark 1 Expectation (2) can be checked for non-nested instances of the five constructors, and we discuss this in Section 7.

4 Non-trivial Guidelines

Non-trivial guidelines either constrain or impose requirements on an MPI implementation. Not all MPI libraries may fulfill them, but for performance portability reasons it is beneficial for implementations to adhere to them. This saves the user from the temptation to look for the best performing constructor, and let him focus instead on the most convenient, close-to-the-application-logic description.

The self-consistent principle would seem to require that MPI libraries do *type normalization* of any user-defined datatype to the "most efficient" representation that could be expressed by other datatype constructors. The MPI_Type_commit function is the point where MPI libraries can do such normalization. For instance, a $\mathsf{struct}(n, m_n, T)$ where all n blocks have the same basetype could trivially be converted into an indexed type which has penalty $2n$ instead of $3n$. Or an $\mathsf{index}(n, m_n, T)$ where all blocks have the same size could be converted into an $\mathsf{index\_block}(n, m, T)$, again with less penalty. If in addition the indices are regularly strided the $\mathsf{index\_block}(n, m, T)$ could be converted into a $\mathsf{vector}(n, m, T)$, now with constant penalty, and if the stride is equal to the block length, this could also be expressed as a $\mathsf{contig}(n, T)$. This is stated as guidelines/requirements of the form

$$\mathsf{MPI\_A}(n, \mathsf{struct}(n', m_{n'}, T_{n'})) \approx \mathsf{MPI\_A}(n, \mathsf{index}(n', m_{n'}, T)) \tag{3}$$

for indexed layouts where all indexed elements have the same basetype $T_i = T$. From such requirements it would follow that communication with a datatype T whose type map consists of consecutive, basic datatypes in increasing offset order should be no worse than communication with a basic datatype alone, that is

$$\mathsf{MPI\_A}(n) \approx \mathsf{MPI\_A}(n, T) \tag{4}$$

In general, self-consistency would require MPI implementations to solve the following problem.

Definition 1. *The* datatype normalization problem *is the problem of finding, for given layout, the derived datatype with the lowest penalty describing the same layout.*

At the top level use of a constructor, type normalization is easy and looks sensible. A simple scan through index, blocksize and type lists can easily discover whether a type constructor with high penalty can be expressed in terms of a more regular constructor with lower penalty. However, for nested types, normalization is not trivial. A datatype layout described by the MPI constructors can be described by a tree with repetition counts and displacement/type lists at the nodes, and there are many trees describing the same layout. Finding the one with least penalty is similar to hard optimization problems on trees, and the presence of repetition counts makes the problem particularly difficult. We conjecture that the type normalization problem is NP-hard. If this conjecture is true, it is not reasonable to *require* that an MPI implementation performs optimum type normalization in all cases.

The next benchmark is intended to test whether slightly non-trivial normalizations are performed. It is parameterized in a type T.

Benchmark 3 *a) A strided layout where the ith element is placed at position $is+(i \bmod 2)$ described with the MPI_Type_create_indexed_block constructor (cannot be normalized to a one level vector type) versus a two level vector of $n/2$ blocks of a two element vector with stride $s+1$ and extent $2s$. The first description has penalty n, the second penalty $O(1)$.*

b) A layout of two elements, a stride, three elements, a stride, and a single element is repeated $n/6$ times. This layout described with the MPI_Type_indexed constructor versus description as two elements followed by a vector of $n/3-1$ blocks of three elements, followed by a single element. The latter description has penalty $O(1)$, the former penalty $2n$.

Communication performance with the two versions of the layouts.

5 Packing

MPI provides functionality for packing any layout described by a derived datatype into a contiguous buffer. It is reasonable to expect that in communication functions this is done internally as necessary, such that first packing and then communicating the consecutive buffer does not make sense, performance wise. This is an example of a self-consistent performance requirement in which an MPI functionality (namely, any communication function with a non-contiguous

layout) is implemented (by the application programmer) in terms of other MPI functionality [11].

$$\mathsf{MPI\_A}(n,T) \preceq \mathsf{MPI\_Pack}(n,T,B) + \mathsf{MPI\_A}(B) \tag{5}$$

where B is the intermediate packed buffer.

Benchmark 4 *The previous benchmarks in two versions: communication with datatypes directly in the communication functions, and with a pack/unpack to/-from contiguous buffers before/after communication. Also pack time is measured stand alone.*

As n grows large, a reasonable MPI implementation should be able to do pipelining to overlap any internal packing that may be necessary with other operations. For very small data, explicit packing with MPI_Pack could make sense, but should make no difference.

Again, by self-consistency recursive application of pack should not lead to an improvement [11]. Pack for basetypes should be comparable to `memcpy`; otherwise, the user would be tempted to do this optimization by hand. This implies that packing by hand in the sequence implied by the datatype constructors will not make sense. Hand-packing can lead only to an improvement if non-trivial tricks or domain knowledge is exploited. This can be expressed as

$$\mathsf{MPI\_Pack}(n,T,B) \preceq \mathsf{Userpack}(n,T,B) \tag{6}$$

Note that user-provided code for pack and unpack operations range from very simple loops to complex, memory-hierarchy-aware codes using deep application knowledge. A natural user expectation is that the MPI operations perform at least as well as "simple" user code implemented by straightforward loops over and recursive decomposition of the datatype T.

Benchmark 5 *Packing time versus user packing time with a* simple *pack loop for the datatypes of the previous benchmarks.*

6 Datatype Preprocessing and Commit

It appears difficult to pose self-consistent or absolute performance requirements for the type constructors and the MPI_Type_commit function. For the constructors at least all parameter lists must be read (and unfortunately copied, because the user may change the buffers after the creation call), so the time is $\Omega(n)$ where n is the total size of parameters in the call, and possibly $\Omega(m)$ where m is the penalty of the constituent datatypes (here it probably suffices to go through the normalized subtypes). The MPI_Type_commit function may for trivial library implementations do nothing and take constant time otherwise an expectation may be that no more than linear time (in either penalty or total size of parameters) be taken.

Benchmark 6 *Type construction and commit times are measured for the datatypes of previous benchmarks.*

Library	Phase	User	Contig	Resized	Vector	BIdx	Idx	Struct
MPICH2	Pack	74	65	65	65	94	**180**	**262**
MPICH2	Send	486	459	463	**544**	480	460	457
Open MPI	Pack	74	66	66	66	**375**	**370**	**279**
Open MPI	Send	428	428	428	428	428	428	428
BG/P	Pack	386	148	148	**409**	149	149	149
BG/P	Send	238	238	238	238	238	238	238
POE	Pack	224	195	196	196	195	197	198
POE	Send	368	362	363	362	351	361	373

Table 1. Results for Stride-1 in μs, $n = 32768$. **Bold** entries indicate violation of a performance guideline.

7 Initial Experimental Results

We have implemented a first datatype expectation benchmark program incorporating some of Benchmarks 1-6. The benchmark creates datatypes for describing strided layouts of single basetype elements by means of the five basic constructors. A basic experiment compares the performance with a ping-pong benchmark. Likewise, packing by MPI_Pack can be performed. The benchmark also measures the construction time and the commit time.

We here present some of the benchmark results for communicating n MPI_DOUBLE values with stride 1 (contiguous) and stride 16 (vector) for different MPI implementations communicating in shared memory . We used Open MPI 1.4.3 and MPICH2 1.3.2p1 on a 1GHz Quad Core Opteron 270 HE system at Indiana University, IBM's BG/P MPI on Intrepid at Argonne National Laboratory, and POE MPI 5.1 on a 16 core POWER5+ system at the University of Illinois at Urbana-Champaign (we also have results for POE on POWER7 under Linux; they are qualitatively similar to the POWER5+ results and are omitted).

We compare a simple pack loop (**User**) with types constructed with MPI_Type_contiguous (**Contig**, only stride 1), MPI_Type_create_resized (**Resized**, the extent of the type is used to generate the correct stride), MPI_Type_vector (**Vector**), MPI_Type_indexed_block (**BIdx**), MPI_Type_indexed (**Idx**), and MPI_Type_struct (**Struct**). The combination Send/User means that the data is sent directly from the user buffer (this is only possible in the contiguous stride-1 case).

Table 1 shows the results for different specifications of stride-1 data access. This can be considered the simplest case (a Benchmark 0), and provides both a basis for comparing non-unit strides in Table 2 and for identifying which datatypes the MPI implementation simplifies to a more efficient internal representation. Our experiments show that, for stride-1 data, MPI_Pack is generally faster than a pack loop. We also observed that sending datatypes directly was generally faster than combining packing and sending manually. Our results show that almost all libraries fail to detect the contiguous data pattern reliably. Bold entries in Table 1 show where the self-consistency requirements are violated because the

Library	Phase	User	Resized	Vector	BIdx	Idx	Struct
MPICH2	Pack	592	1035	1034	**843**	**704**	**797**
MPICH2	Send	-	2917	3045	3015	3013	3036
Open MPI	Pack	600	1494	1490	**2773**	**2769**	**2717**
Open MPI	Send	-	3086	3060	**5281**	**5279**	**5269**
BG/P	Pack	2049	2116	2115	**2218**	**2292**	**2368**
BG/P	Send	-	6402	6402	6414	6414	6412
POE	Pack	563	631	623	**2056**	**2064**	**2072**
POE	Send	-	1658	1694	**6203**	**6263**	**6296**

Table 2. Results for Stride-16 in μs, $n = 32768$. **Bold** entries indicate violation of a performance guideline.

requirement of Equation (3) is not met. Note that these timing results have some uncertainty and small differences are not significant.

Table 2 shows the results for different specifications of stride-16 data access. Our experiments show that, for stride-16 data, MPI_Pack is often slower than a pack loop. One notable exception is MPICH2 where the pack performance is slightly better. We also observed that sending datatypes directly was generally faster than combining packing and sending manually. Our results show that almost all libraries fail to detect the vector pattern reliably. Bold entries in Table 2 show where the self-consistency requirements are violated because the requirement of Equation (3) is not met.

8 Conclusion

By identifying self-consistently motivated performance guidelines and performance expectations for the MPI derived datatype mechanism first steps were taken toward a benchmark for testing aspects of datatype performance. The datatype normalization problem was formalized in terms of penalties, and we conjecture that this problem is NP-hard. This limits the amount of type normalization that an MPI library can be expected to do, and therefore the user still needs to be careful how data layouts are described. Our experiments on a selection of platforms and MPI libraries showed unpleasant performance surprises, indicating for instance that very little type normalization is performed, even for cases where this would be trivially possible. The experiments also clearly showed large performance differences depending on the way a given layout is described, thus more normalization could well make sense in MPI implementations.

Acknowledgments

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract DE-AC02-06CH11357 and DE-FG02-08ER25835, and by the Blue Waters sustained-petascale

computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

References

1. Bajrović, E., Träff, J.L.: Using MPI derived datatypes in numerical libraries. In: Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting. Lecture Notes in Computer Science, Springer (2011), this volume
2. Byna, S., Sun, X.H., Thakur, R., Gropp, W.: Automatic memory optimizations for improving MPI derived datatype performance. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting. pp. 238–246. No. 4192 in Lecture Notes in Computer Science, Springer (2006)
3. Gropp, W.D., Kimpe, D., Ross, R., Thakur, R., Träff, J.L.: Self-consistent MPI-IO performance requirements and expectations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 15th European PVM/MPI Users' Group Meeting. Lecture Notes in Computer Science, vol. 5205, pp. 167–176. Springer (2008)
4. Gropp, W.D., Lusk, E., Swider, D.: Improving the performance of MPI derived datatypes. In: Proceedings of the Third MPI Developer's and User's Conference. pp. 25–30. MPI Software Technology Press (1999)
5. Hoefler, T., Gottlieb, S.: Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. In: Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting, Lecture Notes in Computer Science, vol. 6305, pp. 132–141. Springer (2010)
6. Lu, Q., Wu, J., Panda, D.K., Sadayappan, P.: Applying MPI derived datatypes to the NAS benchmarks: A case study. In: 33rd International Conference on Parallel Processing Workshops (ICPP 2004 Workshops). pp. 538–545. IEEE Computer Society (2004)
7. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4th 2009), `www.mpi-forum.org`
8. Reussner, R., Träff, J.L., Hunzelmann, G.: A benchmark for MPI derived datatypes. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting. Lecture Notes in Computer Science, vol. 1908, pp. 10–17. Springer (2000)
9. Ross, R., Miller, N., Gropp, W.D.: Implementing fast and reusable datatype processing. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting. Lecture Notes in Computer Science, vol. 2840, pp. 404–413. Springer (2003)
10. Santhanaraman, G., Wu, J., Huang, W., Panda, D.K.: Designing zero-copy message passing interface derived datatype communication over Infiniband: Alternative approaches and performance evaluation. International Journal on High Performance Computing Applications 19(2), 129–142 (2005)
11. Träff, J.L., Gropp, W.D., Thakur, R.: Self-consistent MPI performance guidelines. IEEE Transactions on Parallel and Distributed Systems 21(5), 698–709 (2010)
12. Träff, J.L., Hempel, R., Ritzdorf, H., Zimmermann, F.: Flattening on the fly: efficient handling of MPI derived datatypes. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting. Lecture Notes in Computer Science, vol. 1697, pp. 109–116. Springer (1999)