On the Expected and Observed Communication Performance with MPI Derived Datatypes

Alexandra Carpen-Amarie
Sascha Hunold
Jesper Larsson Träff
carpenamarie@par.tuwien.ac.at
hunold@par.tuwien.ac.at
traff@par.tuwien.ac.at
TU Wien
Faculty of Informatics, Institute of Information Systems
Research Group Parallel Computing
Favoritenstrasse 16/184-5, 1040 Vienna, Austria

ABSTRACT
We examine natural expectations on communication performance using MPI derived datatypes in comparison to the baseline, “raw” performance of communicating simple, non-contiguous data layouts. We show that common MPI libraries sometimes violate these datatype performance expectations, and discuss reasons why this happens, but also show cases where MPI libraries perform well. Our findings are in many ways surprising and disappointing. First, the performance of derived datatypes is sometimes worse than the semantically equivalent packing and unpacking using the corresponding MPI functionality. Second, the communication performance equivalence stated in the MPI standard between a single contiguous datatype and the repetition of its constituent datatype does not hold universally. Third, the heuristics that are typically employed by MPI libraries at type-commit time are insufficient to enforce natural performance guidelines, and better type normalization heuristics may have a significant performance impact. We show cases where all the MPI type constructors are necessary to achieve the expected performance for certain data layouts. We describe our benchmarking approach to verify the datatype performance guidelines, and present extensive verification results for different MPI libraries.

1. INTRODUCTION

The derived or user-defined datatype mechanism is a powerful, integral feature of MPI that enables communication of possibly structured, non-contiguous, and non-homogeneous (with different constituent basic types) application data with any of the MPI communication operations, without the need for tedious, explicit, possibly time- and space-consuming manual packing between intermediate communication buffers [10, Chapter 4]. Characterizing the expected and actual performance of MPI communication with structured, non-contiguous data is a difficult problem that has been addressed in many studies [4, 12, 15]. We extend and complement this research using a different approach. MPI derived datatypes can be viewed as a mechanism for serializing the access to non-contiguous data layouts. Data elements stored non-contiguously in memory have to be sent or received in a certain order. Serialization can be, and in applications often is [5], handled manually by packing and unpacking the data via contiguous, intermediate buffers of elements of basic datatypes in the desired order, upon which MPI communication operations are then performed. Alternatively, the given non-contiguous data layout and access order can be described by a derived datatype, and the serialization handled transparently by the MPI library implementation. There are three interrelated issues determining the performance of the data serialization and the derived datatype mechanism:

**Issue 1** How expensive is it *per se* to access and serialize data stored in certain (non-)regular patterns in memory?

**Issue 2** How well do specific MPI libraries handle the serialization using derived datatypes? Does the performance depend on the type of communication operation?

**Issue 3** How do different derived datatype descriptions of the same layout affect serialization cost?

The first issue has to do with the data layout itself, and access performance is dependent on both the specific data layout, as well as on the memory system and other factors of the underlying system, and on how well the serialization can be implemented to exploit such capabilities (cache, vectorization, prefetching). Because of this essential dependence on both system capabilities and on what is possible for each particular access pattern, it does not seem possible to state system-independent expectations or guidelines *a priori* on the costs of processing and communicating structured data. Nevertheless, it is enlightening for users to have means to measure the difference in communication performance with differently structured data layouts.
The second issue focuses on the quality of the MPI implementation for accessing structured layouts. The MPI standard itself does not prescribe how the datatype mechanism has to be implemented. It does, however, interrelate communication and datatype constructors in a way that makes it possible to formulate and check concrete expectations on the performance of the derived datatype mechanism. We explain and examine such expectations in the paper.

The third issue is solely related to the quality of the MPI library. With the given MPI datatype constructors [10, Chapter 4], it is easy to see that the same layout can be described in an infinite number of ways (almost all of which are trivial and irrelevant). However, for a given application layout there are often competing, non-contrived ways of describing it. We can compare the communication performance with such different descriptions. It might be sensible to expect that an MPI library ensures that performance is more or less the same, no matter how the user chooses to describe the given layout. We will argue why this is a reasonable expectation, and discuss why it cannot be (easily) fulfilled.

We discuss benchmarking of the MPI derived datatype mechanism in an attempt to characterize both the “raw performance” of communication with structured data (Issue 1), as well as to develop means for verifying the expected performance of certain uses of the derived datatype mechanism. We focus on three different (meta) performance guidelines, previously discussed by Gropp et al. [4], but give more precise formulations and implementations here. We then use our benchmarks to evaluate concrete MPI libraries and systems. Our benchmarks are synthetic, but parameterized to make it possible to investigate patterns that are relevant for applications. Most of the patterns and derived datatype descriptions that are considered here are natural and deliberately quite simple. Other synthetic patterns, in part derived from applications, have been used in other studies [12, 15]. Schulz et al. used derived datatypes for piggybacking small headers on larger messages, and contrast the performance achieved with derived datatypes against uses of the MPI pack/unpack functionality [17]. A trivial comparison of derived datatype and pack/unpack performance for simple 2- and 3-dimensional stencil operations with standard derived datatypes was done by Wu et al. [22].

We believe that the capability of transparently communicating structured data is a strong (and rather unique) feature of MPI. It is therefore important to ensure good and consistent performance of communication with derived datatypes. The larger purpose of this study is to prevent unrealistic performance expectations, but also to make developers and application programmers aware of concrete performance problems in given MPI libraries and systems. Much work has been done over the past decades in improving the communication performance with derived datatypes [1, 11, 13, 14, 16, 20]. For instance, it has been shown, in many different variations, that piece-wise packing of structured layouts described by derived datatypes can be performed efficiently [11, 16, 20]. This is important for efficient pipelining and overlapping of data accesses and communication. Likewise, some developments focused on exploiting the memory hierarchy [1] and the communication capabilities for strided, non-contiguous data communication [21].

Many of the experiments in this paper are concerned with Issue 1. The expectation is that MPI libraries (at the MPI_-

2. CHARACTERIZING DATATYPE PERFORMANCE

We first attempt to estimate the additional overhead (if any) in communicating non-contiguous data by comparing to the time for communicating the same amount of data from a contiguous memory buffer. In other words, the focus is on the differences in communication performance caused by different types of (non-)regular layouts, and not on the way that MPI handles such layouts. However, these concerns cannot be completely separated. We use MPI derived datatypes to describe our non-contiguous layouts, and thus do not attempt to estimate derived datatype overheads in any absolute way by comparing to any “best possible” way of copying non-contiguous data layouts between contiguous buffers (“packing and unpacking by hand”). The reasons for this are twofold. First, it is not at all obvious what the best possible way to manually pack some complex non-contiguous layout into a contiguous buffer for some specific system is. Second, such a comparison is not necessarily fair, since the derived datatype mechanism makes it possible to interact with the communication system, for instance by pipelining large non-contiguous buffers by partial packing [11, 16, 20] and/or by exploiting hardware capabilities for non-contiguous data communication. Such optimizations that are possible for the MPI implementation via the MPI derived datatype mechanism are difficult to perform at the application level.

To establish a baseline performance, we consider non-contiguous, blockwise, strided data layouts with a given serialization order of a given number of \( n \) elements (of some predefined, basic datatype corresponding to a programming language type), and measure the communication performance for different \( n \). We consider what we think are the simplest such layouts, and use what we think are the MPI implementation friendliest (non-nested) derived datatypes to describe fixed blocks of \( k \) elements, such that the complete layouts are described by \( n/k \) successive, contiguous repetitions (count
At contiguous smaller units of elements; we use constructors used to describe them. All layouts consist of consecutive blocks in Figure 1, and state below the MPI datatype constructors used to describe them. Any layout consists of parameterized, blockwise blocks in Figure 1, and state below the MPI datatype constructors used to describe them. Any layout consists of parameterized, blockwise

2.1 Communication Patterns

Derived datatypes can be used with all types of MPI communication operations, but may behave differently in different contexts. We therefore benchmark with three different types of communication operations in order to get an idea of whether this is the case. The \( n \) elements are communicated either from a contiguous buffer, or as a non-contiguous layout described by a derived datatype as outlined above. We use the following communication operations and patterns:

1. **Point-to-point** communication with blocking **MPI_Send** and **MPI_Recv** operations.
2. The **asymmetric (rooted) collective** **MPI_Bcast** on \( p \) processes.
3. The **symmetric (non-rooted) collective** **MPI_Bcast** on \( p \) processes.

We do not benchmark one-sided communication performance with structured data. One reason is that with the one-sided communication model, descriptions of derived datatypes may have to be transferred between processes, and MPI libraries may differ too much in the way this is handled.

2.2 Basic, Static Datatype Layouts

We first experiment with the parameterized, blockwise layouts described below. These layouts are static, by which we mean that derived datatypes of \( k \) base type elements are set up in advance and used for the whole sequence of experiments. When a total of \( n \) elements, stored according to either of these layouts, is to be communicated, the count argument in the communication operations is adjusted down to \( n/k \). The layouts thus consist of regularly strided, but structured, non-consecutive **blocks**. We illustrate these types of blocks in Figure 1, and state below the MPI datatype constructors used to describe them. All layouts consist of contiguous smaller units of elements; we use \( A \) for the number of elements in a unit. Units are strided with some stride \( B \), and mostly we require \( B > A \). In either of the layouts the number of elements in a unit may vary (different \( A \) values), or the strides of the units may vary (different \( B \) values), or both. We use an \((AB)\) notation for this. We use \( A \) (respectively \( B \)) when the blocksize (respectively stride) is fixed over the units, and \( A \) (respectively \( B \)) when the blocksize (respectively stride) varies between the units. Our basic layouts are as follows.

**Contiguous**: is a contiguous buffer of elements described by a predefined, basic MPI datatype (no derived datatype).

**Tiled \((AB)\)**: is a contiguous unit of \( A \) elements repeated with a stride of \( B \) elements, requirement \( B > A \) (the case \( B = A \) would be a contiguous layout). In MPI, the datatype is constructed using **MPI_Type_contiguous** with a count of \( A \) and a call to **MPI_Type_create_resized** to obtain the extent \( B \). A block in this case has \( k = A \) elements and an extent of \( B \) elements.

**Block \((AB)\)**: consists of two contiguous units of \( A \) elements with alternating strides \( B_1 \) and \( B_2 \), requirement \( B_1 \neq B_2 \), and \( B_1 \), \( B_2 > A \) (otherwise the layout would be as above). The description in MPI is done using **MPI_Type_create_indexed_block** and **MPI_Type_create_resized**. This block has \( k = 2A \) elements and an extent of \( B_1 + B_2 \) elements.

**Bucket \((\bar{AB})\)**: consists of two alternating, contiguous units of \( A_1 \) and \( A_2 \) elements, with strides \( B_1 \) and \( B_2 \), respectively. In MPI, the datatype is described with **MPI_Type_indexed**. This block has \( k = A_1 + A_2 \) elements and an extent of \( B_1 + B_2 \) elements.

All blocks can be defined over arbitrary predefined, basic MPI datatypes. It seems natural to assume that communication performance, regardless of the type of communication, should not depend on which basic type is used, but only on the amount of data communicated. However, with MPI this assumption is problematic, since different basic types have different semantics (doubles, integers, characters), and MPI may have to handle different basic types differently. In most systems and situations, this will not be the case, but measurements with different basetypes have to be performed.

2.3 Benchmarking Setup

In the following, we give an overview of the hardware and software setup used for our experiments.

2.3.1 System and MPI Libraries

The experiments have been conducted on a 36 node Linux cluster called Jupiter, where each node is equipped with two Opteron 6134 processors (see Table 1). The nodes are interconnected using an InfiniBand QDR network. We have benchmarked the datatype performance for three MPI libraries, namely NEC-MPI-1.3.1, MVAPICH2-2.1 and OpenMPI-1.10.1; in the paper we show results for the former two (see our technical report for the full set of results [2]).
Table 1: Hardware and software used in the experiments.

<table>
<thead>
<tr>
<th>machine</th>
<th>36 × Dual Opteron 6134 @ 2.3GHz</th>
<th>InfiniBand QDR MT26428</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Jupiter</td>
<td></td>
</tr>
<tr>
<td>MPI libraries</td>
<td>NEC MPI-1.3.1, MVAPICH2-2.1, OpenMPI-1.10.1</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc 4.4.7, gcc 4.9.2 (flags -O3)</td>
<td></td>
</tr>
</tbody>
</table>

The benchmarks, for which the results are shown in the paper, have been compiled using gcc 4.4.7. We have examined the datatype performance after compiling with gcc 4.9.2, to check whether the compiler version is an experimental factor. However, we have not seen any effects by using gcc 4.9.2.

2.3.2 Benchmarking Communication Patterns

We now explain how the benchmarking of the different communication patterns was done, in particular, which times have been measured. In each benchmark (Ping-pong or collective), we compare different datatypes for the same total communication volume. In addition, the measured run-times do not include the datatype setup times, and thus, they represent the communication times (latencies) only.

For the Ping-pong experiments, we first synchronize the two involved MPI processes with an MPI_Barrier. Then, a message is sent from one process, received by the other and returned using MPI_Send and MPI_Recv operations, and each process measures the time taken for the two operations. The time to perform a Ping-pong is then computed as the maximum over the local run-times of both processes. For each mpirun, we compute the median of the nrep run-times. Then, we calculate the mean, minimum, and maximum values over these r median run-times. These values will be used in the plots, i.e., the error bars in the bar graphs denote the minimum and maximum of the r median run-times.

2.4 Experimental Results

We now summarize our findings that characterize the costs of communicating simple, structured data in comparison to communicating the same amount of contiguous data. We have used the basic datatype MPI_INT as the element base-type. We have experimented with two variants for each of the basic layouts, which are summarized in Table 2. All layouts in both variants are defined using the unit size parameter A, and the values of the A’s and B’s are chosen such that all Variant 1 layouts have the same total extent of \( n + 2n/A \), and all Variant 2 layouts have the same total extent of 3n.

As this gives rise to a very large amount of experimental data, we only include a few exemplary results in the paper. A detailed list of all our experimental results can be found in our technical report [2].

We describe all experiments by stating the (derived) datatypes used, the reference (baseline) layout against which we evaluate, our expectations (hypotheses) on the performance, and then show and comment on the results.

2.4.1 Expectation Test 1

This is our basic experiment to measure the “raw” performance of the simple, non-contiguous layouts of Figure 1. We experiment with different message sizes (fixed number of elements in the layouts) and vary the blocksize parameter A. We use both variants Variant 1 and Variant 2 for determining the remaining parameters in the layouts.

Table 2: Basic layout variants (cf. Figure 1).

<table>
<thead>
<tr>
<th>Layout</th>
<th>Variant 1</th>
<th>Variant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiled (AB)</td>
<td>B = A + 2</td>
<td>B = 3A</td>
</tr>
<tr>
<td>Block (AB)</td>
<td>A1 = A + 1</td>
<td>B1 = 2A</td>
</tr>
<tr>
<td>Bucket (AB)</td>
<td>A1 = A - 1</td>
<td>B1 = A/2</td>
</tr>
<tr>
<td>Alternating (AB)</td>
<td>A1 = A - 1</td>
<td>B1 = A/2</td>
</tr>
</tbody>
</table>

Each datatype experiment with a collective call has been measured for \( r = 5 \) mpiruns.

2.3.3 Data Processing

When conducting a single datatype experiment, we obtain \( r \) datasets, each containing \( nrep \) measurements. For each mpirun, we compute the median of the \( nrep \) run-times.
Type description.

We compare the datatype layouts depicted in Figure 1.

Expectations.

We expect all communication operations with the non-contiguous layouts to be slower than using the Contiguous layout. We expect this difference to become smaller when increasing the blocksize $A$. It is interesting to find out how large the difference to the contiguous baseline performance is, how the performance is changing between the layouts, how it depends on the type of communication, and whether there are differences between the MPI libraries.

Results.

The tiled layout indeed gives the best performance among the four non-contiguous layouts for all three communication patterns. The differences between the layouts are the largest for small values of the blocksize parameter $A$. For all libraries, there is a large difference between process configurations when all MPI processes are on the same node and when they are on different nodes (not shown here, see our technical report [2]). For the NEC MPI-1.3.1 library, the performance with non-contiguous data, especially Tiled ($AB$), for processes on the same node is close to the raw performance with contiguous data. The libraries show a large difference in the way they handle non-contiguous data. While the raw performance with contiguous data is comparable among libraries, there is about a factor of two (and more) difference for the non-contiguous layouts, see Figure 2. The NEC MPI-1.3.1 library performs best, as it handles non-contiguous layouts with a tolerable overhead.

2.4.2 Expectation Test 2

MPI provides predefined, basic datatypes corresponding to the basic C and Fortran programming language types. These basic types have different semantic content, and communication performance may differ for data of different basic types. Knowing when this is the case is a valuable information to the application programmer. In particular, we investigate how consecutive buffers consisting of different semantic units perform in comparison to raw, uninterpreted bytes (described by MPI_BYTE).

![Graphs showing performance comparison between different datatypes and layouts.](image)

Figure 2: Basic layouts comparison for each communication pattern, element datatype: MPI_INT, $m = 2.56$ MB, $32 \times 1$ processes ($2 \times 1$ for Ping-pong), Variant 1.

Type description.

The unit of the Tiled-heterogeneous ($ABT$) layout consists of different basetypes $T_1, T_2, \ldots$ with blocksize $A$ and stride $B$, where the stride of each block is given in units of the corresponding basetype. The layout is shown in Figure 3. The unit can be described in MPI using MPI_Type_create_struct. It is required that $B \geq A$, where equality is allowed. In our experiments, we use a contiguous layout consisting of MPI_CHAR, MPI_INT, MPI_DOUBLE, MPI_SHORT, with $B = A$ and with $A$ varying between 2 and 200.

**Expectation.**

Unless the underlying system indeed requires a different handling of different basic datatypes, we would expect that a contiguous, dense layout of different basic types performs as well as the corresponding amount of unstructured bytes in the MPI communication operations. Since the MPI_Type_create_struct constructor respects alignment constraints
for the basic MPI datatypes (in a sense, these are semantic constraints), there might be differences when calling \texttt{MPI\_Type\_create\_struct} to set up the \textit{Tiled-heterogeneous} layout leads to a non-contiguous layout. This may degrade communication performance.

**Results.**

Our results confirm the hypothesis. The cases with a large difference between the reference and the compared layouts can be explained by alignment constraints that cause the \textit{Tiled-heterogeneous} layout to become larger than expected, and non-contiguous. This is the same for all three libraries. As the results are not surprising, the corresponding figures are omitted.

2.5 Summary

Our basic layouts can be used to gain insight about the performance when communicating non-contiguous data. We used only two blocksize and stride variants (see Table 2), and observed that the qualitative performance differences are similar. It may thus not be required to check a very large number of other variants. For this reason, we only use the \textbf{Variant 1} layouts in the next section. It is noteworthy that there are surprisingly large performance differences in handling structured data between the MPI libraries.

3. PERFORMANCE EXPECTATIONS

We now investigate relative performance gains (or the opposite) by using MPI derived datatypes, that is, the second and third issue raised in Section 1. We first formulate more precisely what it is reasonable to expect, and benchmark with the aim of verifying or falsifying these expectations. Our expectations take the form of self-consistent performance guidelines [19].

An MPI performance guideline states that a certain MPI operation for some given problem size \(n\) should not be slower than some equivalent MPI way of performing the same operation on the same problem size (with all other things being equal). If the MPI operation is slower than the composition of other MPI constructs implementing the same functionality, the operation could be replaced. This is clearly something that an application programmer should not have to do. Verifying such guidelines that interrelate different operations and features of the MPI standard provides a strong means of verifying that a given MPI implementation is “same”. The verification of a set of guidelines can give valuable hints to the application programmer on how to use features of the MPI standard best.

For MPI datatypes, it seems impossible to say anything absolute about the communication performance for different layouts. But it may well be possible to formulate expectations about how the \textit{same} layout is handled when it is described with different datatype constructors and different MPI operations.

A first guideline, which is directly derived from the MPI standard [10, Section 4.1.11], states that

\[
\text{MPI}_X(c, t) \preceq \text{MPI}_X(\text{contig}(c), t) \quad \text{(GL1)}
\]

excluding the time for setting up and committing the contiguous type on the right-hand side, i.e., an MPI communication operation \(X\) should have a similar latency when transferring either \(c\) elements of type \(t\) or one derived datatype of a contiguous block of \(c\) elements. If either side of the equation would be faster than the other, the application programmer could easily switch between them. Thus, it can be expected that an MPI implementation delivers similar performance for both equation sides. As we will see, the argument is not correct, since the \texttt{MPI\_Type\_commit} operation can perform global optimizations on the contiguous representation of the layout that may lead to a better performance than possible with \(c\) repetitions of the block described by \(t\).

Since this cannot be controlled (or queried) by the application, the two sides of Guideline (GL1) may actually be doing different things.

The next guidelines state that whatever implicit packing and unpacking of non-contiguous data (that may be necessary inside an MPI communication operation) is performed at least as efficiently as explicitly packing and unpacking the whole communication buffer before and after the communication operation using \texttt{MPI\_Pack} and \texttt{MPI\_Unpack} [10, Section 4.2]. In a good MPI library, we would expect many cases when the left-hand side performs significantly better than the right-hand side. Thus:

\[
\text{MPI}_X(c, t) \preceq \text{MPI}_X(\text{contig}(c), t) + \text{MPI}_X(\text{packed}(c), t) \quad \text{(GL2)}
\]

for an MPI sending operation \(X\), meaning that the performance of the left-hand side is expected to be at least as good as the performance of the right-hand side (all other things being equal). Similarly for an MPI receiving operation \(Y\):

\[
\text{MPI}_Y(c, t) \preceq \text{MPI}_Y(\text{contig}(c), t) + \text{MPI}_Y(\text{packed}(c), t) \quad \text{(GL3)}
\]

The right-hand sides have the disadvantages (1) of requiring an extra buffer for the intermediate, contiguous packing unit, (2) of preventing direct communication of large contiguous parts of the datatype and (3) of preventing pipelining of packing and unpacking in the communication operations (as well as all other dynamic optimizations, and optimizations that exploit communication hardware support). Therefore it should not be recommended. We would expect that MPI libraries trivially fulfill these guidelines with equality, and would hope to find relevant cases where the left-hand sides are much faster than the right-hand sides.

Any data layout can be described in an infinite number of ways with the available MPI datatype constructors. This is easy to see, for instance \texttt{contig}(1,t) describes the same layout as \(t\) itself for any datatype \(t\). For any given data layout, each MPI library will have layout descriptions that lead to the best communication performance. The \texttt{MPI\_Type\_commit} operation provides a handle for the MPI library to transform the datatype given by the user into a better (if possible), internal description. This process is called \textit{datatype normalization} [4], and we call this best, alternative rep-
representation of a layout described by datatype \( t \) its \textit{normalized form} \( \text{normal}(t) \). The expectation is that an MPI library will indeed attempt to find a good normalized form at \texttt{MPI_Type_commit} time (if not, the user could do better by deriving the normalized form by himself and setting up the datatype in that way), which is formalized as the following \textit{datatype normalization} performance guideline:

\[
\text{MPI}_X(c, t) \preceq \text{MPI}_X(c, \text{normal}(t)) \quad (\text{GL}4)
\]

That is, we expect the performance of a communication operation \( X \) with datatype \( t \) to be no worse than what can be achieved with the best, normalized description of the layout. The guideline is tricky, since the user may not readily be able to see what is the best way to describe a layout in a given situation. But in many cases he can give a good guess, and the guideline states that we would expect the \texttt{MPI_Type_commit} operation to do as well.

The normalization heuristics typically applied by MPI libraries replace more general type constructors (struct) with more specific ones (index or index block), collapse nested constructors, and identify large contiguous segments, where such replacements are applicable. Explicit descriptions of common type normalization heuristics can be found in [8, 9, 11, 13, 16]. As we will see in the following, there are natural layout descriptions that are \textit{not} normalized by these heuristics, leading to severe violations of the guideline.

### 3.1 Communication Patterns

In our experiments, we will use the same three types of communication operations as in Section 2. In order to verify Guidelines (GL2) and (GL3), we extend the benchmarks with \texttt{MPI_Pack} and \texttt{MPI_Unpack} operations to achieve the same semantics as when datatype arguments were used directly in the communication calls:

1. Ping-pong (cf. Schneider et al. [15]): Ping side: \texttt{MPI_Pack} followed by \texttt{MPI_Send} followed by \texttt{MPI_Recv} followed by \texttt{MPI_Unpack}. Pong side: \texttt{MPI_Recv} followed by \texttt{MPI_Pack} followed by \texttt{MPI_Send}.
2. Asymmetric (rooted) collective, e.g., \texttt{MPI_Bcast} on \( p \) processes. Root: \texttt{MPI_Pack} followed by \texttt{MPI_Bcast}. Non-roots: \texttt{MPI_Bcast} followed by \texttt{MPI_Unpack}.
3. Symmetric (non-rooted) collective, e.g., \texttt{MPI_Allgather} on \( p \) processes. All processes call \texttt{MPI_Pack}, followed by \texttt{MPI_Allgather}, then all processes perform \( p \) successive \texttt{MPI_Unpack} operations on the received, packed blocks.

In the \texttt{MPI_Allgather} pattern, the successive unpacking of the received blocks is necessary, since the concatenation of packing units is not a packing unit [10, Section 4.2], so even if the received packed blocks do form a contiguous piece of memory, it is not correct to unpack it with only one \texttt{MPI_Unpack} operation.

### 3.2 Experimental Results

The structure of experiments is guided by the guidelines, and we state for each experiment what our expectations (hypotheses) are, and comment on whether the results support or falsify them. As baseline we use in most cases the simple layouts of Section 2. We report results only for the \texttt{MPI_INT} basetype and the \textit{Variant 1} basic layouts.

#### 3.2.1 Expectation Test 3

For Guidelines (GL2) and (GL3), we first use the layouts of Section 2 with the same values for \( A \) and \( B \) and compare the benchmark performance with datatype communication against the performance with explicit pack and unpack operations.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Reference Layouts</th>
<th>Compared Layouts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{Tiled (AB)}, \texttt{Block (AB)}</td>
<td>\texttt{Bucket (AB)}, \texttt{Alternating (AB)}</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{blocksize} A &= 2, 10, 10000 \\
\text{datasize} m &= 64,000 \text{Bytes to 2,560,000 Bytes} \\
\text{comm. patterns} &= \text{MPI_Allgather, MPI_Bcast, Ping-pong} \\
\# \text{ of processes} &= 32 \times 1, 1 \times 16 (2 \times 1, 1 \times 2 \text{ for Ping-pong})
\end{align*}

#### Expectation

We do not expect any MPI library to significantly violate the guidelines. Guidelines (GL2) and (GL3), but with these simple layouts we hope to see cases where an MPI library performs significantly better with datatypes than with explicit packing and unpacking.

#### Results

Much to our surprise, we found many cases where the guidelines are severely violated. For processes on different nodes, both the NEC MPI-1.3.1 and MVAPICH2-2.1 libraries violate the guidelines for all layouts, see Figure 4 for two examples. Especially with MVAPICH2-2.1, the violations are severe and amount to factors of two or more. With MPI processes on the same node, in many cases (with NEC MPI-1.3.1), the MPI derived datatypes performed better than explicit packing and unpacking before and after communication. Examples with the \texttt{MPI_Bcast} pattern are shown in Figure 5.

#### 3.2.2 Expectation Test 4

As a sanity check for Guideline (GL1), we create a contiguous \( n/k \)-element datatype with the \texttt{MPI_Type_contiguous} constructor for each of the \( k \)-element datatypes of Section 2. We compare the performance of the two datatypes against each other for different communication patterns.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Reference Layouts</th>
<th>Compared Layouts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{Tiled (AB)}, \texttt{Block (AB)}</td>
<td>\texttt{Contiguous-subtype with subtypes: Tiled (AB), Block (AB)}</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{blocksize} A &= 2, 10, 100, 1000, 1024, 10000 \\
\text{datasize} m &= 2000 \text{ Bytes, 2,560,000 Bytes} \\
\text{comm. patterns} &= \text{Ping-pong} \\
\# \text{ of processes} &= 2 \times 1, 1 \times 2
\end{align*}

#### Type description

The concrete \textit{Contiguous-subtype} is shown in Figure 6 with \texttt{Tiled (AB)} as subtype. The \texttt{Tiled (AB)} subtype consists of units of \( k = A \) elements, and in the communication patterns, \( n/k \) such units are communicated. In contrast, the \textit{Contiguous-subtype} contains all \( n/k \) units in a single type, so all \( n \) elements are communicated using a count of one with this datatype.
We expect no difference in performance between the reference and compared layouts.

**Results.**

Again, surprisingly, the MVAPICH2-2.1 (and also OpenMPI-1.10.1, not shown here) library violates the guideline for the Tiled (AB) type, with Contiguous-subtype being a factor of three slower with Tiled (AB) as subtype, see Figure 7. For the other basic types Block (AB), Bucket (AB), and Alternating (AB), no significant performance difference between the left-hand and the right-hand side of Guideline (GL1) was detected.

### 3.2.3 Expectation Test 5

The remaining expectation tests are concerned with verifying Guideline (GLA). In all these tests, we give different descriptions of the same layout and study the communication performance. We formulate our expectations (hypotheses) on the assumption that a strong type normalization is not performed by the MPI libraries.

Our first experiment uses the regularly sized Tiled layout, for which we now know the baseline communication performance. We describe this pattern as a larger block comprised of several Tiled (AB) subtypes.

### Experiment.

<table>
<thead>
<tr>
<th>Reference Layout</th>
<th>Tiled (AB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compared Layout</td>
<td>Tiled-struct</td>
</tr>
<tr>
<td>blocksize A</td>
<td>2, 10, 100, 1000, 1024, 10000</td>
</tr>
<tr>
<td>stride B</td>
<td>A + 2</td>
</tr>
</tbody>
</table>
| repetition counts | (a) $S_1 = 1$, $S_2 = 1$  
                 | (b) $S_1 = 2$, $S_2 = 3$ |
| datatype         | 2000 Bytes, 2560000 Bytes |
| comm. patterns   | Ping-pong |
| # of processes   | $2 \times 1, 1 \times 2$ |

**Type description.**

The Tiled-struct layout is a concatenation described with MPI_Type_create_struct of two smaller, contiguously strided layouts of $S_1$ and $S_2$ tiled blocks. The description is illustrated in Figure 8. Each Tiled sub-layout has the same blocksize $A$ and stride $B$. The number of elements in the structure is $(S_1 + S_2)A$. We also allow the degenerated (fully contiguous) case of $A = B$. However, we only test the performance of the structured layout using $A < B$, with $A$ and $B$ chosen as in Variant 1 (cf. Table 2).

**Expectation.**

As explained, since the user can easily describe the Tiled layout with the MPI_Type_contiguous and MPI_Type_create_struct constructor, we would like to expect that the MPI library can detect from the Tiled-struct description that the underlying pattern is a simple, tiled pattern. That would require detecting that both sub-layouts of the MPI_Type_create_struct are indeed tiled (although with different repetition counts) and have the same basetype. However, the heuristics used by MPI libraries at MPI_Type_commit time usually work differently [8, 9]. Therefore, we actually expect to see cases where the Tiled-struct description performs worse than the reference layout.

**Results.**

The results in Figure 9, especially for MVAPICH2-2.1, show a large performance difference for $A = 2$. Even in the case of $S_1 = S_2$, where the two subtypes are identically set up, the Tiled-struct performs several factors worse. This shows that the normalization heuristics in the MPI libraries are insufficient to identify the complex description of the simple, tiled layout and to normalize accordingly.

### 3.2.4 Expectation Test 6

The next experiment for Guideline (GLA) is also a sanity check, where we would expect no differences between two equally simple, natural descriptions of a tiled layout. We now describe the Tiled (AB) layout as a vector of $n/A$ contiguous blocks of $A$ elements with stride $B$. This is the "most
accessing the elements in the layout can be computed easily by the datatype engine given these representations. Since the MPI_Type_vector is a commonly used datatype constructor, it may even have been specially optimized, such that the description as Tiled-vector might be slightly advantageous.

Results.

As shown on the left-hand side of Figure 11, the Tiled-vector performs much worse than repeating the Tiled vector block, for the NEC MPI-1.3.1 (and also for OpenMPI-1.10.1) library and small values of $A$. This is again surprising. However, the comparison of the absolute performance between NEC MPI-1.3.1 and MVAPICH2-2.1 shows that the bad performance of Tiled-vector is relative, in absolute terms it is on par with the performance in MVAPICH2-2.1 for both descriptions of the layout. These findings also illustrate that performance guidelines can only ensure consistency in an MPI library. Guideline verification needs to be complemented with benchmarking against hard baselines.

3.2.5 Expectation Test 7

Our next description of the Tiled layout is done using a nested vector. We describe a larger block of a constant $S$ number of units of $A$ elements and stride $B$ with the MPI_Type_vector constructor. On this datatype, we build a dynamic vector of $n/(SA)$ blocks with a stride of $SB$ elements. In order to express the stride correctly, this vector has to be constructed with the MPI_Type_hvector constructor.

Table: Reference Layout vs. Tiled (AB)

<table>
<thead>
<tr>
<th>Reference Layout</th>
<th>Tiled (AB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocksize $A$</td>
<td>2, 10, 100, 1000, 1024, 10000</td>
</tr>
<tr>
<td>stride $B$</td>
<td>$A + 2$</td>
</tr>
<tr>
<td>data size $m$</td>
<td>2000 Bytes, 2,560,000 Bytes</td>
</tr>
<tr>
<td>comm. patterns</td>
<td>Ping-pong</td>
</tr>
<tr>
<td># of processes</td>
<td>$2 \times 1, 1 \times 2$</td>
</tr>
</tbody>
</table>
Experiment.

<table>
<thead>
<tr>
<th>Reference Layout</th>
<th>Tiled (AB)</th>
<th>Vector-tiled</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocksize A</td>
<td>2, 10, 100, 1000, 1024, 10000</td>
<td></td>
</tr>
<tr>
<td>stride B</td>
<td>A + 2</td>
<td></td>
</tr>
<tr>
<td>datasize m</td>
<td>2000 Bytes, 2 560 000 Bytes</td>
<td></td>
</tr>
<tr>
<td>comm. patterns</td>
<td>Ping-pong</td>
<td></td>
</tr>
<tr>
<td># of processes</td>
<td>2 × 1, 1 × 2</td>
<td></td>
</tr>
</tbody>
</table>

Type description.

The setup of the nested vector Vector-tiled versus the basic layout Tiled (AB) is illustrated in Figure 12.

Expectation.

We expect that an MPI library will detect that the stride for the outer vector is equal to c times the stride of the inner vector, such that the layout can also be described by a non-nested vector constructor. MPI_Type_commit should perform the transformation. The performance of the two descriptions should therefore be similar, regardless of the communication pattern.

Results.

(a) NEC MPI-1.3.1

(b) MVAPICH2-2.1

Figure 13: Tiled (AB) vs. Vector-tiled, element datatype: MPI_INT, m = 2.56 MB, 2 × 1 processes, Ping-pong.

To our surprise, apparently none of the MPI libraries normalizes the two-level nested vector description into a better layout. For a small unit of size A = 2, the communication time with Vector-tiled compared to the communication time with the simple Tiled (AB) description differ by factors from 2 to 4, as shown in Figure 13.

3.2.6 Expectation Test 8

In this experiment, we look at different, explicit descriptions of the more irregular layout Block (AB), by explicitly listing the displacements and number of elements in all n/k blocks in the n element layouts. The purpose of this experiment is to investigate the relative penalty of having to traverse long, explicit lists of displacements, versus implicit, computed displacements.

Experiment.

<table>
<thead>
<tr>
<th>Reference Layout</th>
<th>Alternating (AB)</th>
<th>Alternating-indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocksize A</td>
<td>2, 10, 100, 1000, 1024, 10000</td>
<td></td>
</tr>
<tr>
<td>blocksize A₁, A₂</td>
<td>A₁ = A + 1, A₂ = A + 3</td>
<td></td>
</tr>
<tr>
<td>stride B</td>
<td>B₁ = A + 1, B₂ = A + 3</td>
<td></td>
</tr>
<tr>
<td>datasize m</td>
<td>3200 Bytes, 2 560 000 Bytes</td>
<td></td>
</tr>
<tr>
<td>comm. patterns</td>
<td>Ping-pong</td>
<td></td>
</tr>
<tr>
<td># of processes</td>
<td>2 × 1, 1 × 2</td>
<td></td>
</tr>
</tbody>
</table>

Type description.

The Block-indexed layout uses the Block (AB) layout with given A, B₁, and B₂, described with the MPI_Type_create_indexed_block constructor with n/A indices and blocksize A. The block displacements can easily be computed. This is illustrated in Figure 14.

Figure 14: Static Block (AB) (top) vs. dynamic Block-indexed (bottom) description of the Block (AB) layout.

Expectation.

An MPI library should normalize both cases to the same internal datatype representation with good performance. It is doubtful that anything like that will happen. More importantly, it is not obvious which of the two descriptions is better. A reasonable expectation is that beyond some number of elements, the large array of displacements (indices) in the Block-indexed datatype will become expensive to traverse, and that simple repetitions of the small, irregular non-contiguous Block pattern will perform better.

Results.

For small blockizes of A, the Block description is worse, especially for the NEC MPI-1.3.1 library. Otherwise the performance of the two descriptions looks similar. Nevertheless, the absolute performance of the NEC MPI-1.3.1 library is still better. We do not show the results here (the figures are available in our report [2]).

3.2.7 Expectation Test 9

This experiment is similar to the previous one. Here, two descriptions of the “most irregular” of the four basic layouts, namely Alternating (AB) are contrasted.
Type description.
The **Alternating-indexed** datatype is based on the Alternating layout with given $A_1$, $A_2$, $B_1$, and $B_2$, described with the MPI Type indexed constructor with $n/(A_1 + A_2)$ indices and block sizes of $A_1$ and $A_2$. The data layout is illustrated in Figure 15.

![Type description](image)

**Figure 15:** Static Alternating $(AB)$ (top) vs. dynamic Alternating-indexed (bottom) description of the Alternating $(AB)$ layout.

Expectation.
As for the previous experiment, it is not obvious which of the two descriptions will perform better (under the pessimistic assumption that no normalization takes place). Experimental results will give insight on whether there is a penalty for large lists of displacements in the MPI Type indexed constructor.

Results.
For larger values of $A$, the performance of the two descriptions looks similar. For small values of $A$, the results are the opposite of the previous experiment. Especially for the NEC MPI-1.3.1, the Alternating-indexed description performs worse than the Alternating $(AB)$ description. Results are not shown here (see our technical report [2]).

### 3.2.8 Expectation Test 10

In our final test with the basic layouts, we look at an Alternating $(AB)$ pattern where the stride $B_2$ of the second unit is equal to the number of elements $A_2$ in the unit. This pattern can be described as a repetition of small datatypes describing fixed blocks. It can alternatively be formulated as a layout comprised of (1) a first, small unit of $A_1$ elements, (2) a large, regularly strided middle part, and (3) a last, small unit of $A_2$ elements. We expect the second description to perform better, and we want to check this hypothesis.

**Experiment.**

<table>
<thead>
<tr>
<th>Compared Layouts</th>
<th>Alternating-repeated</th>
<th>Alternating-struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocksize $A$</td>
<td>2, 10, 100, 1000, 1024, 10000</td>
<td></td>
</tr>
<tr>
<td>unit block sizes $A_1$, $A_2$</td>
<td>$A_1 = A - 1$, $A_2 = A + 1$</td>
<td></td>
</tr>
<tr>
<td>stride $B$</td>
<td>$A + 1$</td>
<td></td>
</tr>
<tr>
<td>datasize $m$</td>
<td>3200 Bytes, 2560/6000 Bytes</td>
<td></td>
</tr>
<tr>
<td>comm. patterns</td>
<td>Ping-pong</td>
<td></td>
</tr>
<tr>
<td># of processes</td>
<td>$2 \times 1, 1 \times 2$</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 17:** Alternating-repeated vs. Alternating-struct. element datatype: MPI INT, $m = 2.56$ MB, $2 \times 1$ processes, Ping-pong.

For small values of $A$, the Alternating-repeated way of communicating the pattern is indeed slower for all tested MPI libraries; see Figure 17. This shows that a stronger type normalization (on the fly) than currently performed by MPI libraries is needed to approach the baseline performance. It also shows, and this is important, that the MPI Type create struct constructor is needed for the best description even of homogeneous layouts where all elements have the same basic type.
3.2.9 Expectation Test I1

For our final set of experiments, we use another layout. Given an \( (n + 1 - A) \times A \) matrix, we want to communicate together the first row of \( A \) elements and the (remainder of the) first column of \( n - A \) elements, for a total of \( n \) elements. For examining Guideline (GL4), we again compare natural ways of describing this layout, and compare the measured communication times. A similar example was used by Gañan et al. [3].

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Compared Layouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocksize ( A )</td>
<td>2, 10, 100, 128, 512, 1000, 1024, 5000, 10000</td>
</tr>
<tr>
<td># of elements ( n )</td>
<td>100, 10240, 1000, 1024, 5000, 10000</td>
</tr>
<tr>
<td>comm. patterns</td>
<td>Ping-pong</td>
</tr>
<tr>
<td># of processes</td>
<td>2 × 1, 1 × 2</td>
</tr>
</tbody>
</table>

Type description.
A row-column layout (submatrix of \( (n + 1 - A) \times A \) matrix), consisting of \( A \) consecutive elements followed by \( n - A \) elements in a strided layout with stride \( A \), can be described either by

1. using \( \text{MPI Type create indexed block} \) with \( n \) indices (RowCol-fully-indexed),
2. using \( \text{MPI Type create indexed} \) with \( 1 + n - A \) indices (RowCol-contiguous-and-indexed), or
3. using \( \text{MPI Type create struct} \) consisting of a contiguous subtype of \( A \) elements, followed by a vector of \( n - A \) blocks of one element with stride \( A \) (RowCol-struct).

The layout and the three possible descriptions as derived datatypes are shown in Figure 18.

Figure 18: From left to right: RowCol-fully-indexed, RowCol-contiguous-and-indexed, RowCol-struct.

Expectation.
The latter representation is the most compact, and expectedly best performing. This layout description also illustrates that the full power of the \( \text{MPI Type create struct} \) constructor is needed, even for homogeneous layouts of elements of the same basic type. As we do not expect the MPI libraries to perform a normalization into an efficient data representation, our hypothesis is that the RowCol-struct datatype will perform better than the two other types, and that RowCol-contiguous-and-indexed may perform better than RowCol-fully-indexed as the blocksize \( A \) goes up.

Figure 19: (a) NECMPI-1.3.1, (b) MVAPICH2-2.1

Results.
The results in Figure 19 confirm that for NECMPI-1.3.1 the compact description RowCol-struct gives the best performance, closely followed by RowCol-fully-indexed. For the MVAPICH2-2.1, the performance is worse (compared to NECMPI-1.3.1 as baseline) for all three descriptions, except for large values of \( A \). The results for OpenMPI-1.10.1 (available in [2]) are as expected, with the compact RowCol-struct description being close to a factor of two faster than the other two (except for the large \( A \) values).

4. SUMMARY AND OUTLOOK

We performed a large number of experiments to explore the performance of communication with differently structured, non-contiguous data layouts described by MPI derived datatypes. We focused on simple tiled layouts parameterized by element counts and strides, and structured the experiments as a set of expectation tests using MPI performance guidelines.

The results were revealing and in many cases surprising and disappointing. For instance, it was unexpected that Guidelines (GL2) and (GL3) would be violated, but we found many cases (for all libraries) where these guidelines were severely compromised. In such cases, the recommendation to use datatypes is hard to justify. It is definitely important to look into the reasons and improve the situation. We also observe that the communication performance with (non-trivial) derived datatypes is quite different between the libraries. For example, the current version of MVAPICH does not handle derived datatypes as efficiently as the other libraries.

In addition, the simplest expectations concerning the use of the \( \text{MPI Type contiguous} \) constructor, captured in Guideline (GL1), were sometimes violated. We believe that these violations can and should be repaired.

Our experiments around Guideline (GL4) first and foremost show that the way a given layout is described as a derived datatype matters a lot. Or put differently, the heuristics employed by common MPI libraries in \( \text{MPI Type commit} \) are insufficient to find good internal datatype representations. It is worthwhile to improve the situation, since an application programmer currently needs a good intuition to select an efficient derived datatype description. Simple rules of thumb are not enough: our findings sometimes contradicted our own intuitions and expectations. Furthermore, some experiments show that datatype descriptions may be too localized to make a sufficiently good normalization pos-
sible, namely, that both the repetition count and the data-
type are needed for computing the normalized type descrip-
tion. However, normalization on the fly in each communi-
cation call is not an option for a high-performance MPI li-
brary, especially since (optimal) normalization may be very
expensive [3]. Therefore, the performance equivalence of
the seemingly innocent Guideline (GL1) cannot hold when
MPI_Type_commit may normalize the contiguous type (right-
hand side of the guideline). MPI might need more query
functionality for the application programmer to explore and
guide the normalization.

5. ACKNOWLEDGMENTS

We thank Antoine Rougier and Felix Donatus Lübbe (TU
Wien) for helping us with some of the experiments.

6. REFERENCES

Improving the performance of MPI derived datatypes
by optimizing memory-access cost. In CLUSTER,
derived datatypes: Performance expectations and
Polynomial-time construction of optimal MPI derived
datatype trees. In IPDPS. IEEE Computer Society,
2016.
Performance expectations and guidelines for MPI
derived datatypes: a first analysis. In EuroMPI,
algorithms for fast fourier transform and conjugate
gradient using MPI datatypes. In EuroPVM/MPI,
Reproducible MPI micro-benchmarking isn’t as easy
as you think. In EuroMPI/ASIA, pages 69–76. ACM,
2014.
datatype reconstruction for vector and index types. In
to convert packing code to compact datatypes for
efficient zero-copy data transfer. Technical report,
University of Illinois at Urbana-Champaign, 2011.
Retrieved from http://hdl.handle.net/2142/26452, last
visited on 03/01/2016.
datatype generation and optimization. In PPoPP,
runtime-compiled engine for derived datatypes. In
EuroMPI, 2015.
benchmark for MPI derived datatypes. In
fast and reusable datatype processing. In
[14] R. B. Ross, R. Latham, W. Gropp, E. L. Lusk, and
R. Thakur. Processing MPI datatypes outside MPI. In
Application-oriented ping-pong benchmarking: how to
assess the real communication overheads. Computing,
datatype processing using runtime compilation. In
the performance of transparent MPI piggyback
vector and index-block types. In EuroMPI/ASIA,
Self-consistent MPI performance guidelines. IEEE
[20] J. L. Träff, R. Hempel, H. Ritzdorf, and
F. Zimmermann. Flattening on the fly: efficient
handling of MPI derived datatypes. In
performance implementation of MPI derived datatype
communication over InfiniBand. In IPDPS, page 14,
2004.
[22] Y. Wu, J. Song, K. Ren, and X. Li. MPI derived
datatypes and data communication analysis in
meteorological applications. In 2015 IEEE
International Conference on Smart
City/SocialCom/SustainCom (SmartCity), pages
536–541, 2015.