On Expected and Observed Communication Performance with MPI Derived Datatypes[☆],[☆☆]

Alexandra Carpen-Amarie, Sascha Hunold, Jesper Larsson Träff[*]

TU Wien, Faculty of Informatics, Institute of Information Systems
Research Group Parallel Computing
Favoritenstrasse 16/184-5, 1040 Vienna, Austria

Abstract

We are interested in the cost of communicating simple, common, non-contiguous data layouts in various scenarios using the MPI derived datatype mechanism. Our aim is twofold. First, we provide a framework for studying communication performance for non-contiguous data layouts described with MPI derived datatypes in comparison to baseline performance with the same amount of contiguously stored data. Second, we explicate natural expectations on derived datatype communication performance that any MPI library implementation should arguably fulfill. These expectations are stated semi-formally as *MPI datatype performance guidelines*.

Using our framework, we examine several MPI libraries on two different systems. Our findings are in many ways surprising and disappointing. First, using derived datatypes as intended by the MPI standard sometimes performs worse than the semantically equivalent packing and unpacking with the corresponding MPI functionality followed by contiguous communication. Second, communication performance with a single, contiguous datatype can be significantly worse than a repetition of its constituent datatype. Third, the heuristics that are typically employed by MPI libraries at type-commit time turn out to be insufficient to enforce the performance guidelines, showing room for better algorithms and heuristics for representing and processing derived datatypes in MPI libraries. In particular, we show cases where all MPI type constructors are necessary to achieve the expected performance.

Our findings provide useful information to MPI library implementers, and hints to application programmers on good use of derived datatypes. Improved MPI libraries can be validated using our framework and approach.

Keywords: MPI (Message-Passing Interface), Derived datatypes, Non-contiguous communication, Benchmarking, Performance guidelines

1. Introduction

The derived or user-defined datatype mechanism is a powerful, integral feature of MPI (the Message-Passing Interface [2]) that enables communication of structured, possibly non-contiguous and non-homogeneous (having different constituent basic types) application data with any of the MPI communication operations,

[☆]This work was supported by the Austrian FWF project "Verifying self-consistent MPI performance guidelines" (P25530), and co-funded by the European Commission through the EPiGRAM project (grant agreement no. 610598).

[☆☆]The paper is a revised, extended version of an eponymous paper presented at the 23rd European MPI Users' Group Meeting (EuroMPI), Edinburgh, September 2016 [1].

[*]Corresponding author

Email addresses: carpenamarie@par.tuwien.ac.at (Alexandra Carpen-Amarie), hunold@par.tuwien.ac.at (Sascha Hunold), traff@par.tuwien.ac.at (Jesper Larsson Träff) without the need for tedious, explicit, possibly time- and space-consuming manual packing between intermediate communication buffers [2, 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 several studies [3, 4, 5]. We extend and complement this research using a different approach. Non-contiguously stored data have to be sent and received in a certain, agreed-upon order. This serialization can be, and in applications often is [6], 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. The promise of a good MPI library implementation is that the descriptive approach will perform at least as good as or better than the manual (in time and/or space).

There are three interrelated factors determining the performance of data serialization with the derived datatype mechanism:

Factor 1 How expensive is serialization *per se* for given, non-contiguous, (non-)regular data layouts?

Factor 2 How well does a specific MPI library handle serialization with derived datatypes? Does performance depend on the type of communication?

Factor 3 How do different derived datatype descriptions of the *same layout* affect serialization performance?

The first factor has to do with the data layout and access patterns themselves, specifically how well given patterns and layouts fit the memory (cache) system and can utilize system capabilities (vectorization, prefetching) and features of the communication system (strided communication). Knowing this would establish the best possible performance baselines, against which to judge the performance of MPI communication with structured data. However, because of the essential dependence on system capabilities and access patterns, it does not seem possible to state independent expectations *a priori* on the costs of processing and communicating structured data. As performance baseline, we instead compare to communication of the same amount of consecutively stored elements of the basic datatype(s).

The second factor is in addition determined by the MPI implementation for serializing and communicating 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, but relative performance expectations. We explain and examine such expectations in the paper.

The third factor is solely an issue with the quality of the MPI library. With the given MPI datatype constructors [2, Chapter 4], it is easy to see that any given data layout can be described in an infinite number of (mostly trivial and irrelevant) ways. However, for many application layouts there are often competing, non-contrived ways of describing them. We can compare the communication performance with such different descriptions. Since MPI requires derived datatypes to be committed to the library, 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 in more detail why this is a reasonable expectation, and discuss why it is difficult to fulfill.

We discuss benchmarking of the MPI derived datatype mechanism in an attempt to characterize both the "raw performance" of communication with structured data in MPI (Factors 1 and 2), as well as to develop means for verifying the expected performance of certain uses of the derived datatype mechanism. We focus on four different (meta) performance guidelines, previously discussed by Gropp *et al.* [3], but give more precise formulations and benchmark 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 [4, 5]. The MPI packing and unpacking functionality has often been compared against derived datatypes, e.g., in the study of Schulz *et al.* which use derived datatypes for piggybacking small headers on larger messages [7].

The capability for transparently communicating structured data is a strong, widely applicable, and rather unique feature of MPI. It is therefore important to ensure good and consistent derived datatype communication performance, and to know how current MPI libraries behave. The 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 to improve the communication performance with derived datatypes [8, 9, 10, 11, 12, 13]. 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 [9, 12, 13], which is important for efficient pipelining and overlapping of data accesses and communication. Other developments focused on exploiting memory hierarchy [8] and communication capabilities for strided, non-contiguous data communication [14].

Many of the experiments in this paper are concerned with Factor 3. The expectation is that MPI libraries (at the `MPI_Type_commit` operation) compute a good, internal representation of the user-specified datatype, which has been termed *type normalization* [3]. In this paper, we put more emphasis on showing that strong datatype normalization can have performance advantages. It has recently been shown that optimal type normalization of derived datatypes into tree-structured representations can be done in polynomial time [15, 16, 17], but at high cost if all MPI derived datatype constructors are permitted. The latter two papers show that normalization costs are moderate if the `MPI_Type_create_struct` constructor is left out. However, as our last examples show, in order to get the expected performance for certain layouts this constructor is required, even for homogeneous layouts consisting of data items of the same basic datatype.

The paper is structured in two main parts. In Section 2, the focus is mostly on Factors 1 and 2, where the communication performance for simple layouts described in simple ways is compared against the performance with the same amount of contiguous data. In Section 3, we formalize relative performance expectations as MPI performance guidelines [18], and use them to structure the experiments. We focus on different descriptions of the same simple layouts as used in the first part, and on the performance of derived datatypes versus packing and unpacking with the `MPI_Pack` and `MPI_Unpack` operations. Further, extensive, complementary experimental results can be found in our previous work [1, 19].

2. Characterizing Datatype Performance

We first 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. Our primary focus is on the differences in communication performance caused by different non-contiguous data layouts (Factor 1), and not on the way that MPI handles such layouts (Factor 2). However, these concerns cannot be separated. We describe our non-contiguous layouts with MPI derived datatypes, and therefore cannot distinguish whether overheads are due to the layouts themselves on the given systems, or to the way the MPI derived datatype mechanism is implemented. In particular, we do not examine any "best possible" way of communicating non-contiguous data layouts. The reasons for this are twofold. First, as explained it is not at all obvious what the best possible way to handle communication of given non-contiguous layouts on different, given systems actually is. Second, the MPI derived datatype mechanism provides capabilities for optimization that are difficult to exploit at the user-level independently of MPI. For instance, MPI communication makes it possible to pipeline large non-contiguous buffers by partial packing [9, 12, 13] and/or to exploit hardware capabilities for non-contiguous data communication.

To establish a baseline performance, we consider non-contiguous, (ir)regularly strided data layouts with a given serialization order with some given number of elements, N, of some predefined, basic datatype corresponding to a programming language type, and measure the communication performance for different n. Each of our layouts can be described as small building blocks of some constant number, k, of elements, and we first use the "simplest possible" MPI derived datatype descriptions of these layout building blocks. Each complete, n element layout is then handled as n/k successive, contiguous (in the MPI sense) repetitions (the count argument in the MPI operations) of these building blocks. The baseline performance delivered by an MPI library is the time for communicating n contiguous elements of the same basic type. We describe the basic layouts in Section 2.2.

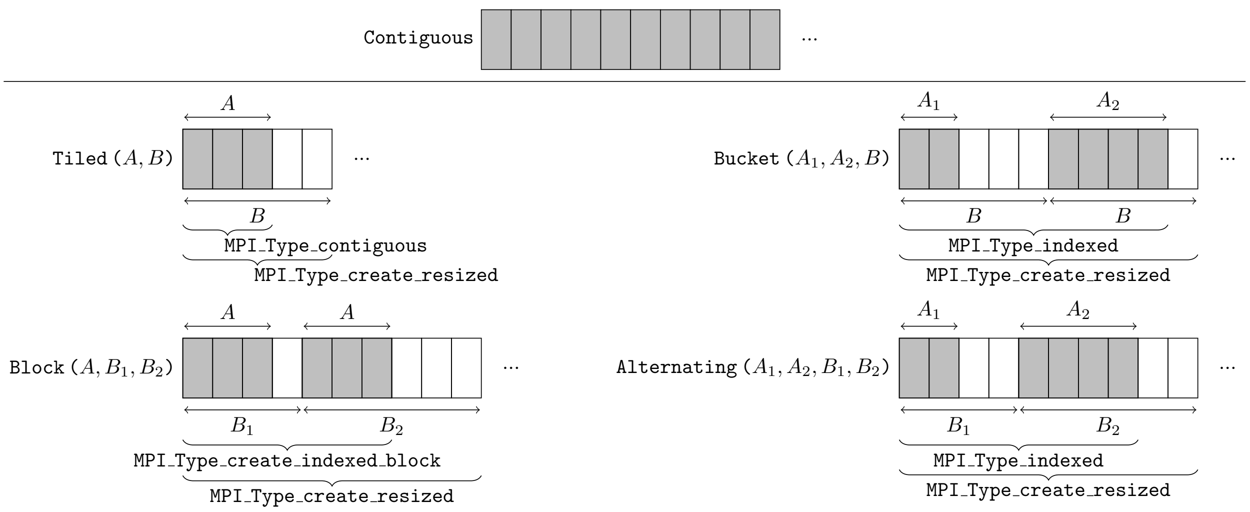


Figure 1: Building blocks for the four data layouts with units sizes $A = 3, A_1 = 2, A_2 = 4$ and strides $B = 5, B_1 = 4, B_2 = 6$. Serialization of the basetype elements is from left to right. The building blocks are described in a "simplest possible" way with the MPI datatype constructors as shown.

2.1. Communication Patterns

Derived datatypes can be used with all MPI communication operations, but may perform 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. The communication pattern is *ping-pong*: One MPI process sends data (*ping*) to another process which sends the received data back (*pong*) to the first process; only two processes in the communicator are actively involved. The same data layout description is used on both sides, that is both processes uses the same derived datatype.
2. The *asymmetric (rooted) collective* `MPI_Bcast` on p processes. A chosen root (process 0 in the communicator) broadcasts its structured data to the other processes which receive the data as the same chosen data layout. In other words, all processes use the same derived datatype.
3. The *symmetric (non-rooted) collective* `MPI_Allgather` on p processes. All processes contribute a buffer with n elements of the chosen layout, and all processes gather the p contributed data into a p times larger buffer. Again, all processes use the same derived datatype for both sent and received elements.

We do not benchmark one-sided communication performance with structured data. One reason for this 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. We also do not examine derived datatypes in the context of non-blocking communication operations, neither for point-to-point nor for collective communication. Non-blocking (and also one-sided communication) operations are methodologically much more difficult to benchmark, but do provide MPI library implementations with possibilities to hide some of the costs of handling non-contiguous data. It would be relevant, though, to extend our patterns to cover both one-sided and non-blocking communication.

2.2. Basic Data Layouts

We consider four non-contiguous data layouts that can all be described as contiguous repetitions of small building blocks of k elements of a chosen, basic datatype. Each of these building blocks is described with a corresponding MPI derived datatype constructor. We say that these building block descriptions are *static* since they do not depend on the total number of elements n to be communicated, but only on

Table 1: Fixed unit and stride choices for maximum blocksize C, unit size A with $1 \leq A \leq C$ (cf. Figure 1).

Layout	`Tiled` (A, B)	`Block` (A, B_1, B_2)	`Bucket` (A_1, A_2, B)	`Alternating` (A_1, A_2, B_1, B_2)
Parameters	$B = 3C$	$B_1 = 2C$	$A_1 = A$	$A_1 = A$
		$B_2 = 4C$	$A_2 = 3A$	$A_2 = 3A$
			$B = 6C$	$B_1 = 3C$
				$B_2 = 9C$

the structure of the building blocks and the number of elements k in the blocks (in Section 3 we consider *dynamic* descriptions that depend on n). The four building blocks consist of contiguous units of basic type elements with some strides between the units. We use A, A_1 and A_2 for the number of elements in the units, and B, B_1 and B_2 for the strides of the units. The names of the building blocks that we use in the following are chosen to indicate the structure of the corresponding n element layouts.

`Tiled` (A, B): A contiguous unit of A elements with a stride of B elements, with $B > A$. The building block is constructed using `MPI_Type_contiguous` with a count of A and a call to `MPI_Type_create_resized` to obtain extent B. The building block has $k = A$ elements and an extent of B elements. The `Tiled` layout described by repeating this building block is a regularly strided sequence of A element units in strides of B elements.

`Block` (A, B_1, B_2): Two contiguous units of A elements with alternating strides B_1 and B_2, with $B_1 \neq B_2$, and $B_1, B_2 > A$. The building block is constructed using `MPI_Type_create_indexed_block` and `MPI_Type_create_resized`. It has $k = 2A$ elements and an extent of $B_1 + B_2$ elements. The corresponding `Block` layout consists of repeated units of A elements, alternatingly with strides of B_1 and B_2 elements.

`Bucket` (A_1, A_2, B): Two alternating, contiguous units of A_1 and A_2 elements, with a regular stride of B elements, with $A_1 \neq A_2$ and $B > A_1, A_2$. This building block is constructed with `MPI_Type_indexed`, and has $k = A_1 + A_2$ elements and an extent of $2B$ elements. The `Bucket` layout is a sequence of regularly strided buckets, alternatingly with A_1 and A_2 elements.

`Alternating` (A_1, A_2, B_1, B_2): Two alternating, contiguous units of A_1 and A_2 elements with strides B_1 and B_2, respectively, with $A_1 \neq A_2$, $B_1 \neq B_2$ and $B_1, B_2 > A_1, A_2$. The building block is described with `MPI_Type_indexed`, and has $k = A_1 + A_2$ elements and an extent of $B_1 + B_2$ elements. The `Alternating` layout described by this building block is a sequence of alternating units with A_1 and A_2 elements, at alternating strides of B_1 and B_2 elements.

The four building blocks are illustrated in Figure 1. The building blocks can be constructed over any of the predefined, basic MPI datatypes. In order to compare same against same in our experiments, we always choose the strides B, B_1, B_2 such that the four described n element layouts have the same, total MPI extent. The restrictions on the strides, e.g., $B_1 \neq B_2$, ensure that the building blocks describe distinct layouts. For each building block, we use the most concise, specific MPI datatype constructors to describe the block. 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. This assumption can be tested by using different basic datatypes. The assumption could be violated because of the different semantics of basic datatypes (doubles, integers, characters), which an MPI library might handle differently.

2.3. Experimental Methodology and Setup

In all our benchmark patterns (ping-pong and collective), we compare different datatypes and/or layouts for the same total communication volume. We give the communication volume as the number of Bytes m for some chosen number of elements n of the chosen, basic MPI datatype. We measure only communication performance, and in all cases *exclude* the time to set up and commit the derived datatypes. For the MPI libraries and systems we have had access to, experiments have shown the communication performance to be

Table 2: Hardware and software used in the experiments.

Machine Name	*Jupiter*	*JUQUEEN* (BlueGene/Q)
Hardware	36 × Dual AMD Opteron 6134 @ 2.3 GHz 16 cores per node InfiniBand QDR MT26428	28 672 × IBM PowerPC A2 @ 1.6 GHz 16 cores per node 5D Torus interconnect, 40 GBps, 2.5 µs latency
MPI Libraries	NEC MPI-1.3.1, MVAPICH2-2.2, OpenMPI-2.0.1	IBM MPI (based on MPICH2 version 1.5)
Compiler	gcc 4.4.7, gcc 4.9.2 (flags `-O3`)	`IBM XL C/C++ for Linux, V12.1` (`-O2 -qarch=qp -qtune=qp`)

independent of the chosen basic datatype. All results in the following are therefore for the basic datatype `MPI_INT`.

We have considered two types of unit and stride parameter choices for the building blocks. In the *fixed* variant, a maximum unit size of C elements is chosen, corresponding to some relevant system property, e.g., cacheline size. The unit sizes A, A_1, A_2 can be chosen freely with $1 \leq A, A_1, A_2 \leq C$, and the strides are chosen as multiples of C such that the conditions on the building blocks are fulfilled. In the *scaling* variant, an arbitrary, positive unit size A is chosen, and the stride parameters B, B_1, B_2 (as well as unit sizes A_1 and A_2 for the bucket and alternating layouts) are chosen as a function of this A such that the building block conditions are fulfilled.

In this paper, all our experiments use the fixed variant with $C = 16$ `MPI_INT` elements, corresponding to a cacheline size of 64 B (assuming that `MPI_INT` is 4 B). We present results only for $A = 1$ and $A = C$, corresponding to the extreme cases with only one element per cacheline, and with full cachelines. Cacheline utilization may be an important factor determining the performance of communicating non-contiguous data layouts (Factor 1). For the choice of C and A, the remainder parameters A_1, A_2, B, B_1, B_2 are chosen as shown in Table 1. With this choice, all n element layouts have the same total MPI extent.

Extensive results for two scaling building block variants can be found in our previous work [1, 19]. Although the absolute times are different, the qualitative findings are similar for both fixed and scaling building blocks.

Here, we present results for medium and very large communication buffers, $m = 256$ KiB and $m = 25.6$ MB. We have also experimented with small buffer sizes of $m = 256$ B and larger, but these results have shown no qualitative differences or new effects; some of these results can be found in our previous work [1, 19].

2.4. Systems and MPI Libraries

The experiments have been conducted on two different systems with different MPI libraries as summarized in Table 2. The first testbed *Jupiter* is a 36 node Linux cluster, where each node is equipped with two AMD Opteron 6134 processors. The nodes are interconnected with an InfiniBand QDR network. On this machine, we have benchmarked the datatype performance of three MPI libraries, namely NEC MPI-1.3.1, MVAPICH2-2.2 and OpenMPI-2.0.1. The benchmarks 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 new effects with gcc 4.9.2. Our second testbed is a BlueGene/Q machine called *JUQUEEN*, which consists of 28 672 IBM PowerPC A2 nodes interconnected through a 5D Torus network. We relied on the IBM XL compiler suite provided on *JUQUEEN* to compile our benchmarks.

2.5. Benchmarking Communication Patterns

We now explain our benchmarking procedures, in particular, which times have been measured.

When measuring ping-pong latency, we synchronize the MPI processes with an `MPI_Barrier` call before each individual ping-pong measurement. The measured ping-pong latency is then defined as the maximum of the local run-times of the two involved processes. The ping-pong measurement is repeated *nrep* times within one `mpirun` call. We repeat the ping-pong test over $r = 5$ calls to `mpirun` on *Jupiter* and $r = 3$ calls to `mpirun` on *JUQUEEN*, as `mpirun` has also been found to be a factor in such experiments [20].

Since the latency of a ping-pong becomes relatively long for the larger message sizes in our experiments, we cannot afford to execute many repetitions (large *nrep*) for every single experiment. In fact, the variance of the run-time for larger message sizes is relatively small. We therefore choose the number of repetitions per test case depending on the datasize m:

on *Jupiter* on *JUQUEEN*

$$nrep = \begin{cases} 100 & \text{if } m \leq 32\,\text{kB}, \\ 50 & \text{if } 32\,\text{kB} < m \leq 320\,\text{kB}, \\ 20 & \text{if } m > 320\,\text{kB}. \end{cases} \qquad nrep = \begin{cases} 50 & \text{if } m \leq 32\,\text{kB}, \\ 20 & \text{if } 32\,\text{kB} < m \leq 320\,\text{kB}, \\ 5 & \text{if } m > 320\,\text{kB}. \end{cases}$$

These values of *nrep* were fixed after analyzing some initial experiments, in which the measurement noise on the *JUQUEEN* was comparably smaller.

For each single datatype experiment, we obtain r datasets, each containing *nrep* measurements. For each `mpirun`, we compute the median of the *nrep* run-times. We then calculate the mean, minimum, and maximum values over these r median run-times. These values are used in the plots, i.e., the error bars in the bar graphs denote the minimum and maximum of the r median run-times.

In this paper, we show only results for the ping-pong pattern. Many results for the collective patterns can be found in our previous work [1, 19]. The ping-pong tests suffice, since qualitative datatype behavior does not seem to change when going to the collective patterns; but certain effects become more pronounced due to the higher communication volume and datatype processing effort.

Since whether communication is via shared memory or via the communication network may influence the datatype performance considerably (Factor 1), in cases where it matters we present ping-pong results both for MPI processes on the same shared-memory node and on two different nodes. On the *JUQUEEN* system there was hardly any difference between the two cases, and for this system we only present the results for MPI processes on different nodes. We always pin the MPI processes to cores in a round-robin fashion. Thus, the first two cores of the same CPU are used for the shared-memory experiments, to which we will refer as the *same node* configurations. When measuring inter-node communication, i.e., the *two nodes* configuration, each process is pinned to core zero on each node.

2.6. Experimental Results

We now summarize our findings that characterize the costs of communicating simple, structured data layouts in comparison to communicating the same amount of contiguous data.

2.6.1. Expectation Test 1

This is our basic experiment to measure the raw communication performance with the simple, non-contiguous layouts of Figure 1. We use the unit sizes $A = 1$ and $A = 16$ (and $C = 16$), and two different message sizes, $m = 256\,\text{KiB}$ and $m = 25.6\,\text{MB}$, with a corresponding number of elements n. We compare to the baseline performance of ping-pong with a consecutive buffer of n `MPI_INT` elements.

Expectations. We expect communication with the non-contiguous layouts to be slower than using the contiguous buffer. We expect this difference to become smaller for the large unit size $A = 16$ (full cacheline) and for larger m. We expect communication with the regular `Tiled(`A, B`)` building block to be faster than with the other, more irregular blocks, but do not know in advance how large the difference will be, neither whether there will be differences between these other layouts. We do not know whether there will be differences between the MPI libraries and systems, neither in absolute terms nor with respect to the relative behavior between the layouts.

Results. The results on the two systems are shown in Figure 2 and Figure 3. Results are qualitatively similar between the two machines and the four different MPI libraries. The `Tiled` layout indeed exhibits the best performance among the four non-contiguous layouts, and for NEC MPI-1.3.1 it is very close to the performance with data in a contiguous buffer (even for the large message size). For $A = 1$, this is somewhat surprising due to the poor spatial cache-locality of only one `MPI_INT` per cacheline. The `Block`

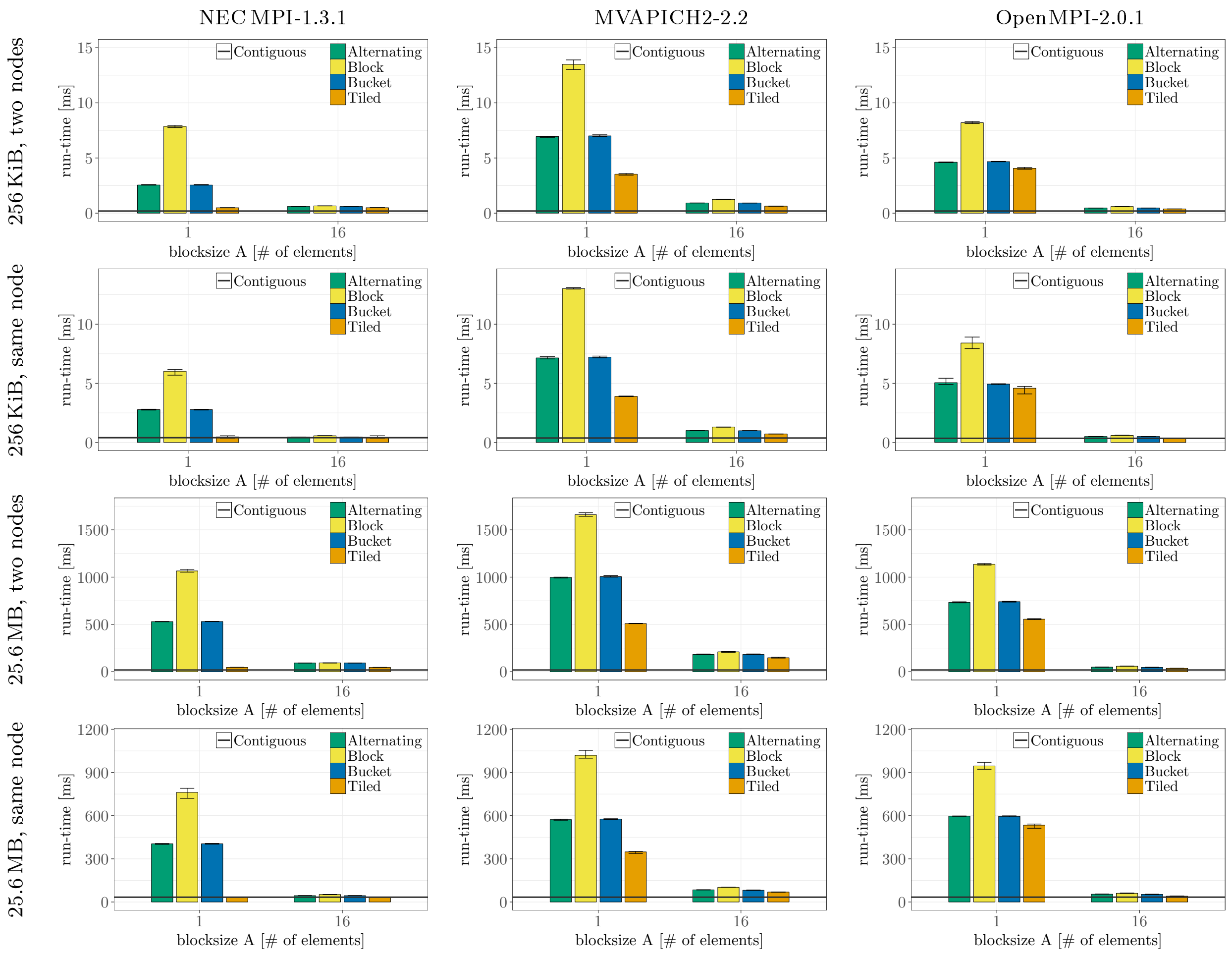


Figure 2: Exp. Test 1 – Basic data layouts vs. contiguous data, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

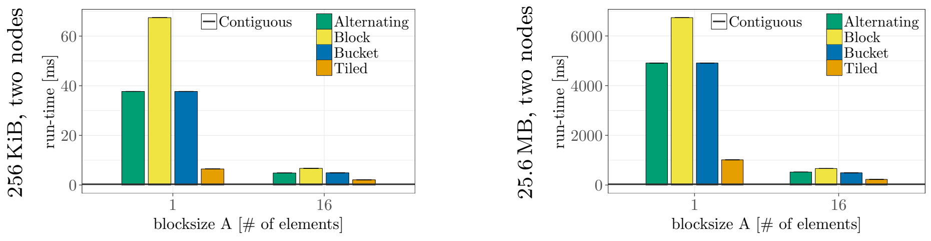


Figure 3: Exp. Test 1 – Basic data layouts vs. contiguous data, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

is the most expensive layout in all cases. On the *Jupiter* system, the three MPI libraries have comparable baseline (contiguous) performance, but differ considerably in absolute times (with MVAPICH2-2.2 being particularly slow). The differences between the layouts are prominent for the small unit size parameter $A = 1$ and largely disappear for $A = 16$. In this experiment, as in all following experiments, there is a large difference between layouts with full cachelines ($A = 16$) and with only one element per cacheline ($A = 1$), the former being faster by often large factors. For all libraries, there is also a noticeable difference between process configurations with the MPI processes on the same node and on two different nodes; shared-memory communication and datatype processing is faster. While the performance difference to the contiguous baseline for the irregular layouts `Alternating`, `Block`, and `Bucket` is only a few factors for the small message size, this difference becomes very considerable as the message size increases, up to factors of 60 for NEC MPI-1.3.1 and OpenMPI-2.0.1, and 90 for MVAPICH2-2.2 (for $A = 1$). This cannot be due solely to the poor cache-locality, since the `Tiled(`A, B`)` layout performs well for the NEC MPI-1.3.1 library.

3. Performance Guidelines for Derived Datatypes

We now focus on the performance that can be achieved by using different derived datatypes to describe the same data layout (Factors 2 and 3). We first formulate more precisely what it would be desirable and defensible to expect, and benchmark with the aim of verifying or falsifying these expectations. Our expectations take the form of self-consistent performance guidelines [18]. We also discuss more realistic expectations based on knowledge of how MPI libraries commonly perform improvements to derived datatype descriptions.

An MPI performance guideline for an MPI operation or an MPI usage states that the operation or usage should not be slower than certain other, equivalent MPI ways of performing the same operation, for the same problem size and under the same circumstances. The argument is that if the MPI operation is slower than an equivalent alternative that implements the same functionality, then the operation could be replaced with a performance advantage with this alternative implementation. This is clearly something that an application programmer should not have to do, but should have instead been taken care of by the MPI library. 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 "sane". Likewise, the verification of a set of guidelines can give valuable hints to the application programmer on how to best use features of the given MPI implementation on the system at hand. If some feature does not perform according to the expectation formalized in the guideline, the guideline tells how to possibly improve the application, and where the MPI library implementation is in need of improvement.

As discussed, it seems impossible to say anything *a priori* about the communication performance for different data layouts. But it is possible to formulate expectations about how the *same* layout is handled when it is described with different datatype constructors and different MPI operations.

The MPI standard states that any communication operation with a derived datatype t and repetition count argument c, $c > 1$ behaves as if the operation was passed a single repetition of a derived, contiguous datatype $\mathsf{contig}(c, t)$ that describes the c contiguous repetitions of the layout t [2, Section 4.1.11]. A first datatype performance guideline therefore states that

$$\mathsf{MPI\_}X(1, \mathsf{contig}(c, t)) \quad \preceq \quad \mathsf{MPI\_}X(c, t) \tag{GL1}$$

meaning that no MPI communication operation X in some fixed context should be slower with a contiguous datatype over basetype t than when called directly with c repetitions of t. If some operation X would be slower with the $\mathsf{contig}(c, t)$ datatype, the user could have done better by not constructing the contiguous type in the first place. On the other hand, when the contiguous datatype is committed, the MPI library has more information available than when committing only t (namely a global view of the full layout via the repetition count c), and this can possibly be exploited to make the left-hand side of Guideline (GL1) faster. We will see an example at the end of this section where the repetition count reveals structure in the full layout that can indeed be exploited for a more efficient description. The performance guideline therefore does not stipulate comparable performance between the two sides (despite what the MPI standard may suggest). This guideline (and the following ones) excludes the time for setting up and committing the derived datatypes t and $\mathsf{contig}(c,t)$, and commit time is not measured in our benchmarks.

The next two 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 the `MPI_Pack` and `MPI_Unpack` operations [2, Section 4.2]. Indeed, this expectation is the whole point of the derived datatype mechanism, and any violation is a serious problem of an MPI library.

For any MPI sending operation X, we would expect the performance of the left-hand side to be at least as good as the performance of the right-hand side (all other things being equal) in

$$\begin{aligned}\mathsf{MPI\_}X(c,t) \quad &\preceq \quad \texttt{MPI\_Pack}(c,t) + \\ &\quad\;\; \mathsf{MPI\_}X(1,\mathsf{packed}(c,t)) \end{aligned} \tag{GL2}$$

where $\mathsf{packed}(c,t)$ denotes the special packing unit of data generated by the `MPI_Pack` operation (which does not have to be committed). Similarly for an MPI receiving operation Y:

$$\begin{aligned}\mathsf{MPI\_}Y(c,t) \quad &\preceq \quad \mathsf{MPI\_}Y(1,\mathsf{packed}(c,t)) + \\ &\quad\;\; \texttt{MPI\_Unpack}(c,t) \end{aligned} \tag{GL3}$$

In a good MPI library, we would expect many cases where the left-hand side performs significantly better than the right-hand side. In particular, the right-hand sides have the disadvantages of (1) requiring an extra buffer for the intermediate, contiguous packing unit, (2) preventing direct communication of large contiguous parts of the datatype, and (3) 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 to explicitly pack and unpack non-contiguous user data. 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, $\mathsf{contig}(1,t)$ describes the same layout as t itself for any datatype t, and so does therefore $\mathsf{contig}(1,\mathsf{contig}(1,\ldots,\mathsf{contig}(1,t)))$. For any given data layout, each MPI library will have layout descriptions that lead to a best communication performance in a given context. The `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 *datatype normalization* [3, 17], and we call this best, alternative representation of a layout described by datatype t its *normalized form* $\mathsf{normal}(t)$. The expectation is that an MPI library will indeed attempt to find a good normalized form at `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 *datatype normalization* performance guideline:

$$\mathsf{MPI\_}X(c,t) \quad \preceq \quad \mathsf{MPI\_}X(c,\mathsf{normal}(t)) \tag{GL4}$$

That is, we would like to expect the performance of a communication operation X with committed datatype t to be no worse than what can be achieved with any externally normalized description of the layout. The user may not readily be able to see what the best way to describe a layout in a given situation is, but in many cases she can give a good guess, and Guideline (GL4) states that we would expect the `MPI_Type_commit` operation to do at least as well. Since $\mathsf{normal}(t)$ per definition will perform at least as well as t itself in operation X, and since any two datatype descriptions of the same layout will have the same normalized form, the guideline states the strong expectation that different datatype descriptions of the same layout, after being committed to the MPI library, will perform similarly. Instead of coming up with a specific normalized form for some datatype t, we can evaluate the guideline by examining the performance of different descriptions of the same layout.

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 applied by MPI libraries can be found in papers by Kjolstad *et al.* [21, 22]. As we will see in the following, there are natural layout descriptions that are *not* normalized by these heuristics, leading to severe violations of the guideline.

Regardless of what an MPI library does in its `MPI_Type_commit` operation, it is reasonable to expect that a description of a given layout with a more specific datatype constructor (taking fewer, simpler arguments, e.g., `MPI_Type_vector`) in any MPI communication operation context performs no worse than a description of the layout with a more general, complex constructor (e.g., `MPI_Type_create_struct`). If that would not be the case, the user (or library implementer) could simply use the more general constructor, and there would be no reason for bothering with the more specific constructor. Depending on the regularity of the layout and the required constructors, this gives rise to a set of of performance guidelines that, when fulfilled, would assure the user that the most specific, "most natural" constructor for each application layout can be used with no performance disadvantage. Such guidelines were described previously by Gropp *et al.* [3]. Specifically, for a regularly strided layout with unit size A and stride B that can be described with the `MPI_Type_vector` constructor, it can be expected that

$$\begin{aligned}\mathsf{MPI\_}X(c, \texttt{MPI\_Type\_vector}(b, A, B, t) &\preceq \mathsf{MPI\_}X(c, \texttt{MPI\_Type\_create\_indexed\_block}(b, A, [B], t))\\ &\preceq \mathsf{MPI\_}X(c, \texttt{MPI\_Type\_indexed}(b, (A), [B], t))\\ &\preceq \mathsf{MPI\_}X(c, \texttt{MPI\_Type\_create\_struct}(b, (A), [B], (t))) \quad \text{(GL5)}\end{aligned}$$

where (A) denotes an array of size b containing the same unit size A, $[B]$ an array of increasing indices with stride B, and (t) an array of the datatype t as required by the MPI constructors. With our datatypes described below we focus on the stronger Guideline (GL4) that subsumes these guidelines.

3.1. Communication Patterns

In our experiments, we use the same three communication patterns operations as in Section 2. In order to verify Guidelines (GL2) and (GL3), we extend the benchmarks with `MPI_Pack` and `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.* [5]): Ping side: `MPI_Pack` followed by `MPI_Send` followed by `MPI_Recv` followed by `MPI_Unpack`. Pong side: `MPI_Recv` followed by `MPI_Unpack` followed by `MPI_Pack` followed by `MPI_Send`.
2. Asymmetric (rooted) collective, e.g., `MPI_Bcast` on p processes. Root: `MPI_Pack` followed by `MPI_Bcast`. Non-roots: `MPI_Bcast` followed by `MPI_Unpack`.
3. Symmetric (non-rooted) collective, e.g., `MPI_Allgather` on p processes. All processes call `MPI_Pack`, followed by `MPI_Allgather`, then all processes perform p successive `MPI_Unpack` operations on the received, packed blocks.

In the `MPI_Allgather` pattern, the successive unpacking of the received blocks is necessary, since the catenation of packing units is not a packing unit [2, Section 4.2], so even if the received p packed blocks do form a contiguous piece of memory, it is not correct to unpack it with only one `MPI_Unpack` operation.

3.2. Experimental Results

The structure of our 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 the four simple layouts of Section 2. Our basetype is `MPI_INT`, and the layout parameters and data sizes are chosen as in the previous section. Also, since the qualitative behavior between the three communication patterns is similar, we present here only the ping-pong results. Experiments have been done on the same systems and with the same MPI libraries. We report mostly for the two node configuration, and give the results for the same node configuration in the cases where there are qualitative differences.

3.2.1. Expectation Test 2

For Guidelines (GL2) and (GL3), we compare the performance of the benchmark with datatype communication against the benchmark with explicit pack and unpack operations.

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

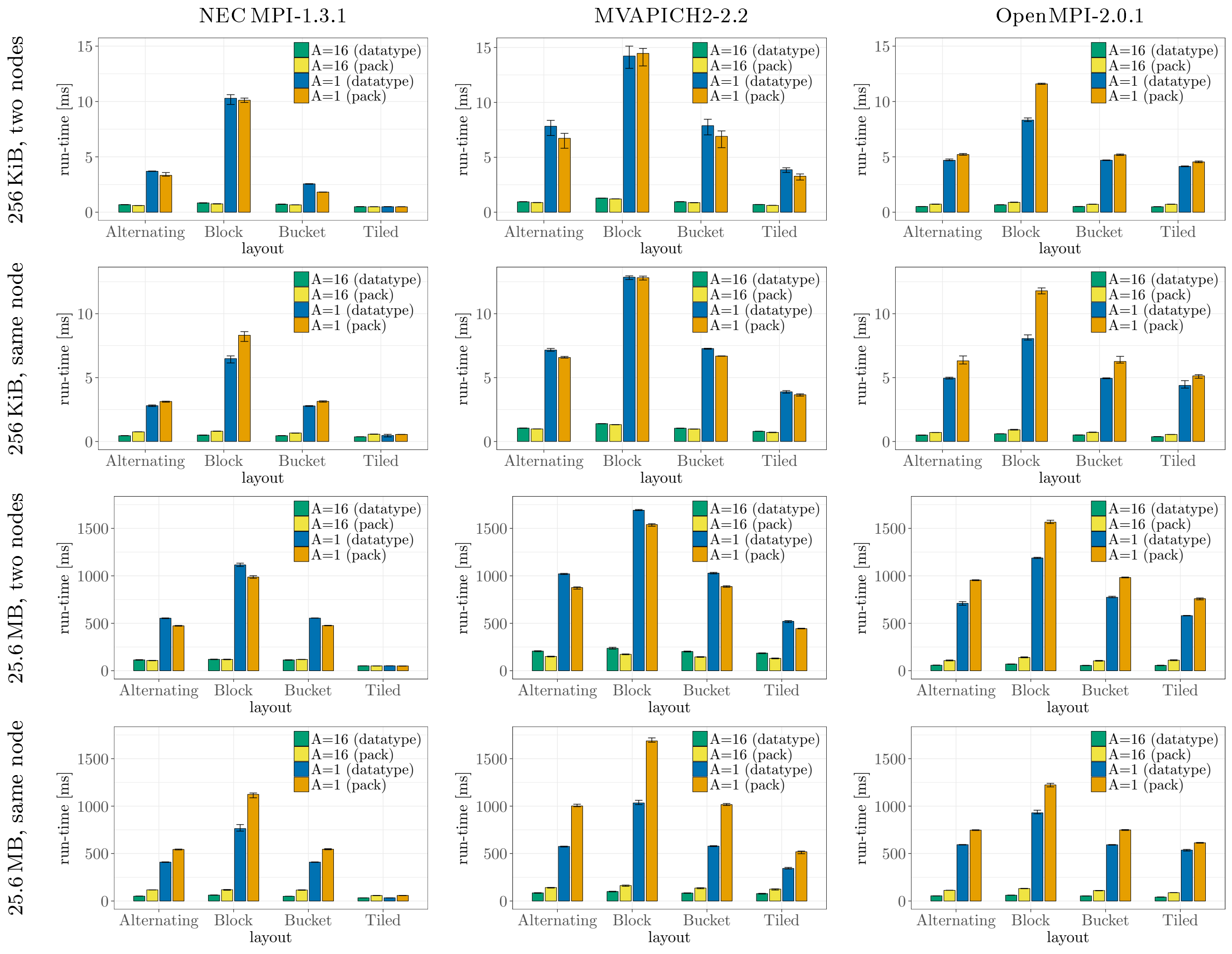


Figure 4: Exp. Test 2 – Basic data layouts vs. explicit data packing/unpacking, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

Results. Much to our surprise, we found many cases where the guidelines are either violated or where there is no advantage of communicating directly with the derived datatype instead of via explicit packing and unpacking. The results for the two machines are shown in Figure 4 and Figure 5. For processes on different nodes, the MVAPICH2-2.2 library violates the guidelines for all layouts when using the large message size, and there is no performance benefit of using datatypes instead of explicit pack-unpack for 256 KiB. Similarly, the NEC MPI-1.3.1 library violates the guidelines for the `Alternating`, `Block`, and `Bucket` layouts with a message size of 25.6 MB and $A = 1$ element per cacheline size $C = 16$. OpenMPI-2.0.1 is the only library that shows a considerable advantage when performing communications directly with the datatype for all configurations, even though, in absolute terms, NEC MPI-1.3.1 performs better in most cases. Datatype performance over explicit pack-unpack is disappointing for the IBM MPI library, where there is virtually no difference between the two cases for any layout, regardless of whether $A = 1$ or $A = 16$.

3.2.2. Expectation Test 3

As a sanity check for Guideline (GL1), we create a contiguous n/k-element datatype $\texttt{contig}(n/k, t)$ with the `MPI_Type_contiguous` constructor for each of the k-element building blocks t described in Section 2.

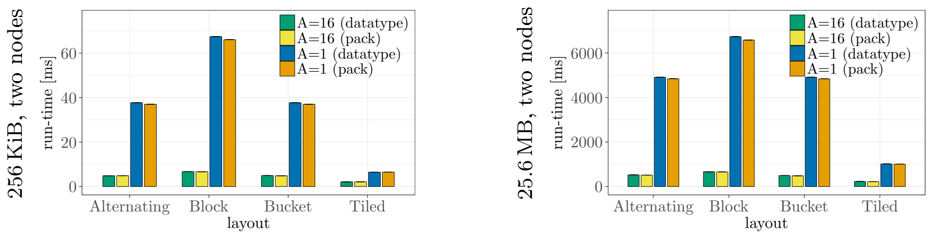


Figure 5: Exp. Test 2 – Basic data layouts vs. explicit data packing/unpacking, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

These contiguous types are our first examples of *dynamic* derived datatypes that can only be set up when the number of elements n to be communicated is known. We have compared the performance of the two datatype descriptions of the four layouts against each other in the three communication patterns, but report results for ping-pong only.

Type description. The concrete `Contiguous-subtype` is shown in Table 3, Expectation Test 3 with, e.g., `Tiled` (A, B) as subtype. The `Tiled` (A, B) subtype consists of blocks of A elements, and in the communication patterns n/A such units are communicated. In contrast, the `Contiguous-subtype` contains all n/A blocks in a single type, so all n elements are communicated using a count of one with this datatype.

Expectation. Since there is no large scale structure in these simple layouts that can be revealed to the MPI library by describing the n/k repetitions as a contiguous datatype, we expect no difference in performance between the reference and compared layouts.

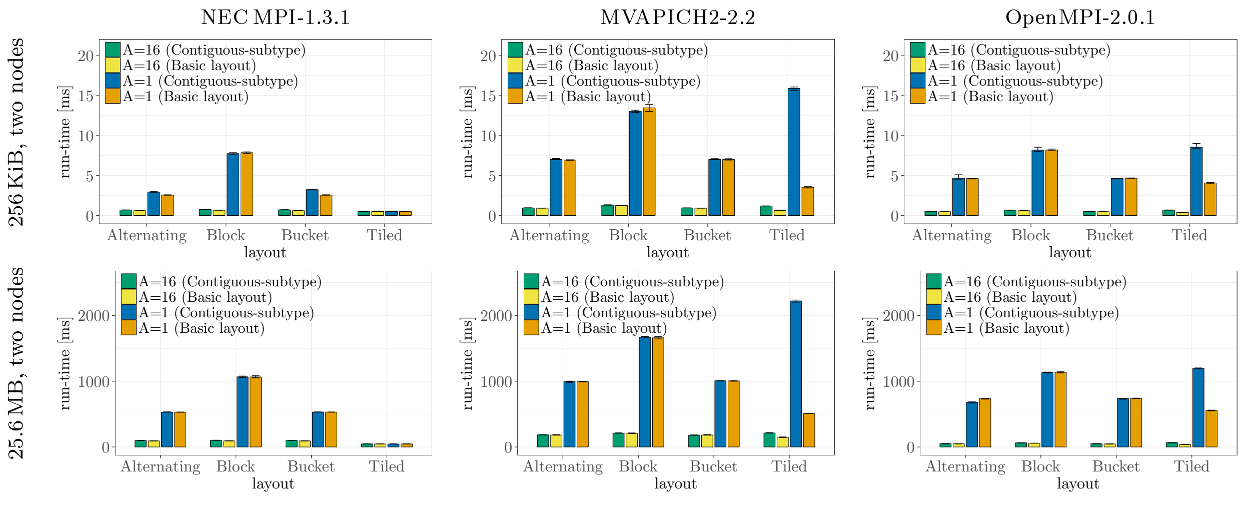
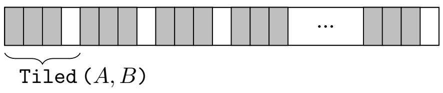
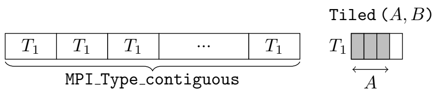
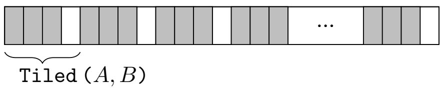
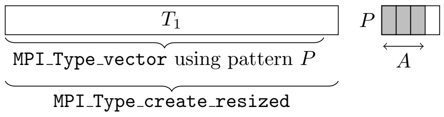
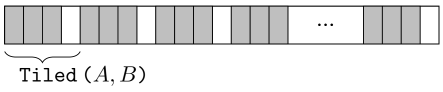
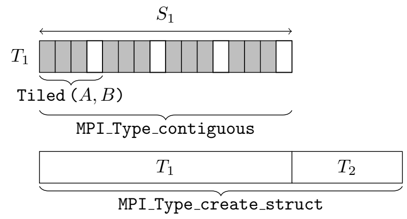
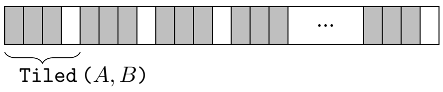
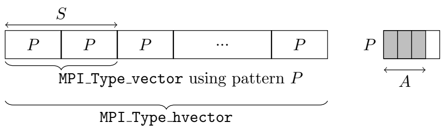
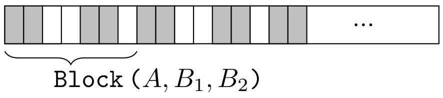
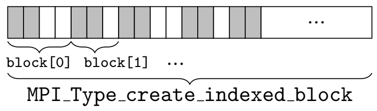
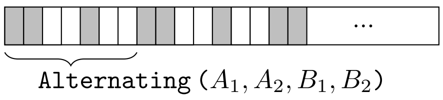
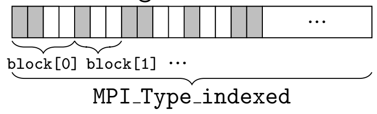
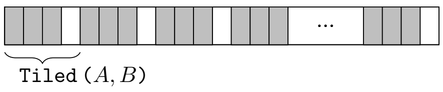
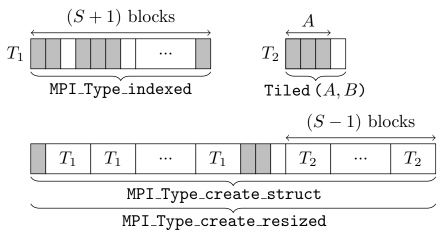


Figure 6: Exp. Test 3 – Basic layouts vs. `Contiguous-subtype`, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

Results. Results are shown in Figure 6 and Figure 7. Surprisingly, the MVAPICH2-2.2 and OpenMPI-2.0.1 libraries violate the guideline for the `Tiled` layout (for $A = 1$), with `Contiguous-subtype` being a factor of two to four slower with `Tiled` (A, B) as subtype. The NEC MPI-1.3.1 library behaves as expected in this test for all four layout building blocks, with no difference between the two different descriptions of the layouts. Also on *JUQUEEN*, the `Tiled` layout severely violates the guideline, both for $A = 1$ and for $A = 16$, and regardless of the message size and node configuration. The other layouts behave as expected.

Table 3: Overview of expectation tests for the derived datatype performance guidelines.

Exp. Test	Reference Layout(s)	Compared Layout(s)
2	`Tiled`, `Block` `Bucket`, `Alternating`	same layouts, but using `MPI_Pack` and `MPI_Unpack`
3	`Tiled`, `Block` `Bucket`, `Alternating` 	`Contiguous` with subtypes: `Tiled` (A, B), `Block` (A, B_1, B_2) `Bucket` (A_1, A_2, B), `Alternating` (A_1, A_2, B_1, B_2)  T_1 T_1 T_1 ··· T_1 MPI_Type_contiguous Tiled (A, B) T_1 A
4	`Tiled` 	`Tiled-vector`  T_1 MPI_Type_vector using pattern P MPI_Type_create_resized P A
5	`Tiled` 	`Tiled-struct`  S_1 T_1 Tiled (A, B) MPI_Type_contiguous T_1 T_2 MPI_Type_create_struct
6	`Tiled` 	`Vector-tiled`  S P P P ··· P MPI_Type_vector using pattern P MPI_Type_hvector P A
7	`Block` 	`Block-indexed`  ··· block[0] block[1] ··· MPI_Type_create_indexed_block
8	`Alternating` 	`Alternating-indexed` 
9	`Tiled` 	`Tiled-struct-indexed`  $(S+1)$ blocks T_1 MPI_Type_indexed A T_2 Tiled (A, B) $(S-1)$ blocks T_1 T_1 ··· T_1 T_2 ··· T_2 MPI_Type_create_struct MPI_Type_create_resized

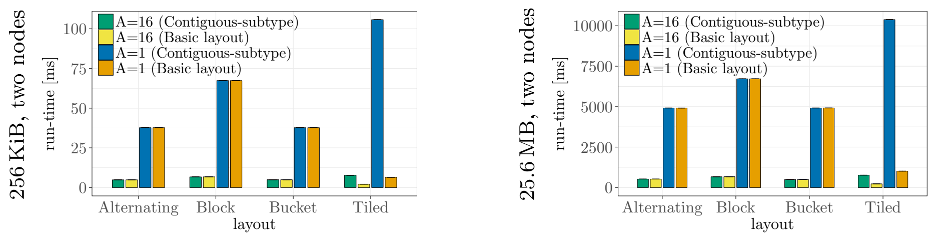


Figure 7: Exp. Test 3 – Basic layouts vs. `Contiguous-subtype`, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

3.2.3. Expectation Test 4

The remaining expectation tests are concerned with verifying Guideline (GL4). In each of these tests, we give two different descriptions of the same data layout and measure the communication performance. We use the four basic layouts for which we now know the baseline performance when described with the corresponding four building blocks. We compare these against (slightly) more complex descriptions of the layouts using different constructors and combinations of constructors. Performance Guideline (GL4) states that we should expect no significant performance difference between these different descriptions, and we are interested to see how far MPI libraries fulfill this expectation. As explained in Section 1, finding the best performing normal form for a given derived datatype description is a difficult problem, for which MPI libraries use only heuristics. We therefore do actually *not* expect that libraries will always fulfill Guideline (GL4), and so the tests will reveal the costs of different ways of describing the layouts with alternative uses of the MPI datatype constructors. In particular, we are interested in descriptions via nested constructors, and in descriptions via constructors that take (long) explicit arrays of displacements, block counts, and subtypes. The datatype descriptions used in our expectation tests are summarized in Table 3.

Our first experiment for Guideline (GL4) is a sanity check where we would expect no performance differences between two natural descriptions of a tiled layout. We describe the `Tiled` layout as a vector of n/A contiguous blocks of A elements with stride B. This is the “most natural” way in MPI to describe a long, regularly strided layout, and is accomplished with the `MPI_Type_vector` constructor. To get the same extent of the vector type as the `Tiled` layout, we resize the extent of the vector to n/A times the extent of `Tiled`(A, B). As in the previous expectation test, the `Tiled-vector` datatype is a *dynamic* derived datatype that can only be set up when the number of elements n to be communicated is known. As in all our experiments, we do not include the datatype setup time in the measured run-time.

Type description. The two contrasted datatype layout descriptions `Tiled`(A, B) and `Tiled-vector` are shown in Table 3, Expectation Test 4.

Expectation. We expect the performance of the two descriptions to match. The MPI internal representations of the two descriptions are likely to be similar, and concrete offsets for accessing the elements in the layout can be computed easily by the datatype engine given these representations. Since `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 in Figure 8, for the NEC MPI-1.3.1 library, the `Tiled-vector` performs much worse for $A = 1$ element per block (and slightly worse for $A = 16$) than simply repeating the `Tiled`(A, B) block. This is a surprising finding of a problem in this particular library. For the other libraries, the performance of the two descriptions is on par, with the `Tiled-vector` sometimes being slightly faster. On the *JUQUEEN* the two descriptions are completely on par as can be seen in Figure 9.

3.2.4. Expectation Test 5

Our next experiment uses the regularly strided `Tiled` layout, for which we now know the baseline communication performance when using the `Tiled`(A, B) building block. Using `MPI_Type_create_struct`, we describe this pattern as a larger block comprised of several `Tiled`(A, B) subtypes. The test examines

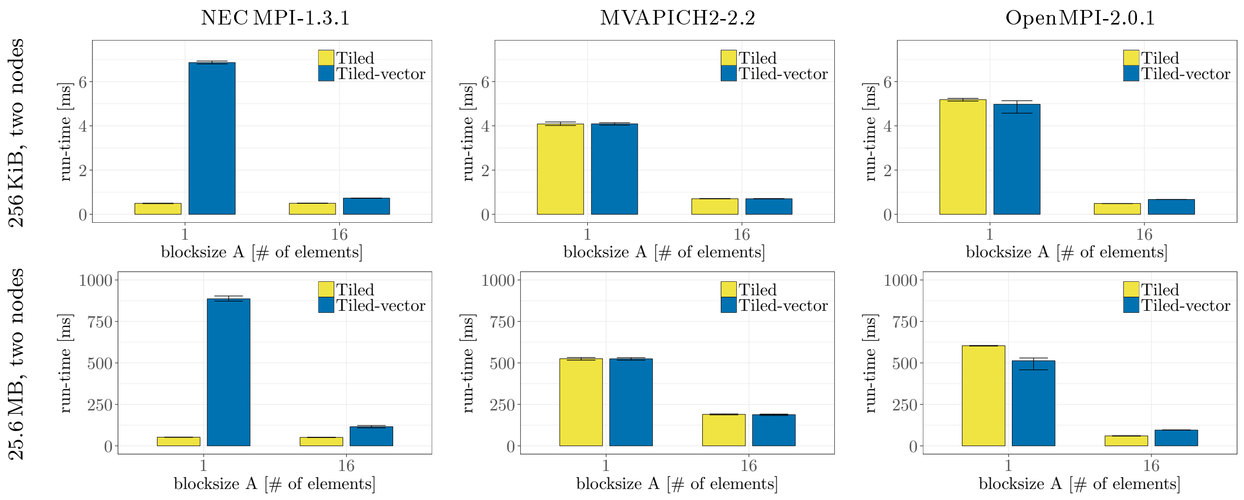


Figure 8: Exp. Test 4 – `Tiled` vs. `Tiled-vector`, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

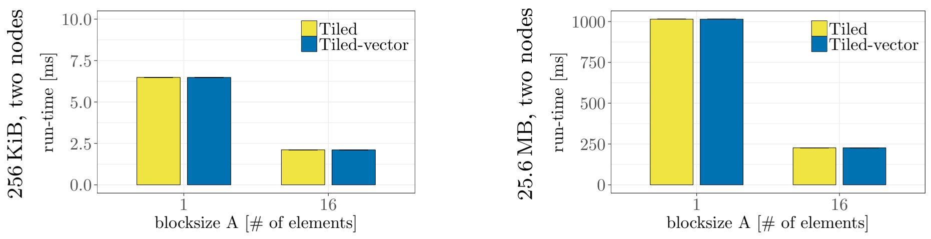


Figure 9: Exp. Test 4 – `Tiled` vs. `Tiled-vector`, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

whether the MPI libraries are able to discover simple regularities in layouts that are described with the `MPI_Type_create_struct` constructor.

Type description. The `Tiled-struct` datatype captures a larger, tiled layout block as a concatenation of two smaller, contiguously strided layouts of S_1 and S_2 tiled blocks put together with an `MPI_Type_create_-struct` constructor. The description is illustrated in Table 3, Expectation Test 5. 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 do two experiments, an "easy" description where the two types have the same number of elements, $S_1 = S_2 = 1$, and a possibly more difficult (for `MPI_Type_commit`) case with larger, different $S_1 = 2, S_2 = 3$.

Expectation. As explained, since the user can easily and with good performance describe the `Tiled` layout with `Tiled`(A, B) building blocks, we would like to expect that the MPI library can similarly detect from the `Tiled-struct` description that the underlying pattern is indeed just a simple, tiled pattern. That would require detecting that both sub-layouts of the `MPI_Type_create_struct` are tiled (possibly with different repetition counts, the case where $S_1 = S_2$ might be easier) and have the same stride and basetype. However, the heuristics used by MPI libraries at `MPI_Type_commit` time usually do not attempt to unify different subtypes of structured MPI types [21, 22]. Therefore, we might well see cases where the `Tiled-struct` description performs worse than the reference layout description.

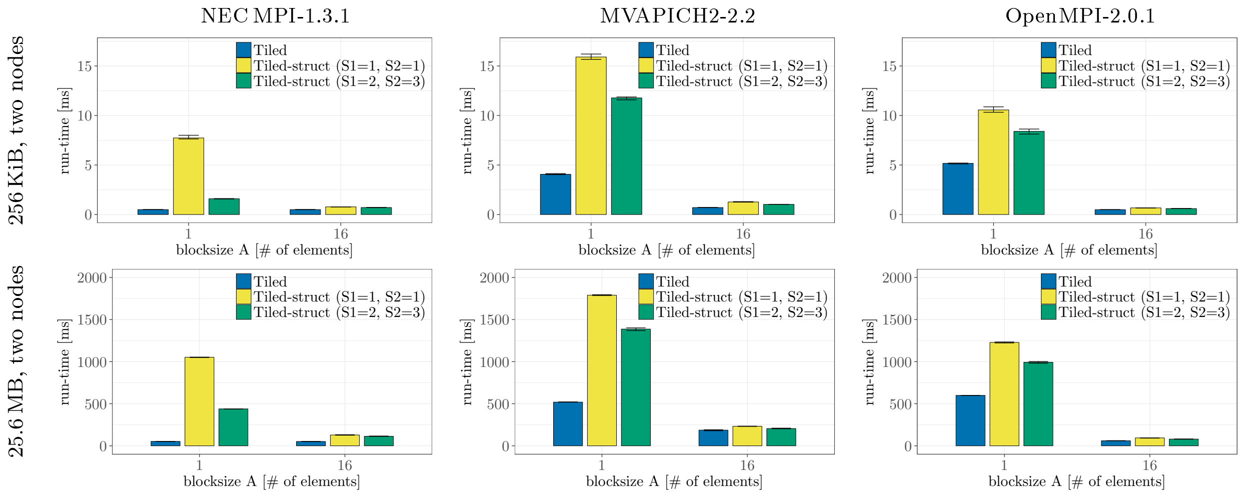


Figure 10: Exp. Test 5 – `Tiled` vs. `Tiled-struct`, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

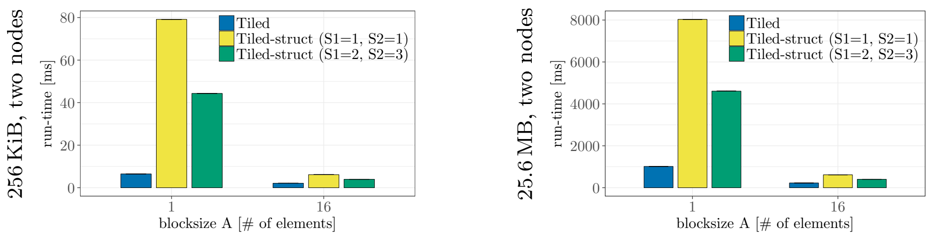


Figure 11: Exp. Test 5 – `Tiled` vs. `Tiled-struct`, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

Results. As can be seen from the results in Figure 10 and Figure 11, none of the MPI libraries discover the underlying `Tiled` pattern and normalize the more complex `Tiled-struct` description. The penalty for using the complex description is the highest when there is only $A = 1$ element per block, and when the blocks are small, $S_1 = S_2 = 1$. For the MVAPICH2-2.2 library, the difference is more than a factor of three, and for the NEC MPI-1.3.1 library even higher, due to the good handling of the `Tiled` (A, B) blocks by this library. The IBM MPI exhibits an even higher penalty of up to a factor of 12.

3.2.5. Expectation Test 6

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

Type description. The setup of the nested vector `Vector-tiled` versus the basic layout `Tiled` (A, B) is illustrated in Table 3, Expectation Test 6. We perform experiments with $S = 2$ and $S = 10$ blocks in the innermost vector.

Expectation. We expect that the MPI libraries 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, and thus recognize that the nested vector datatype description is that of a simple `Tiled` layout. The performance of the two descriptions should therefore be similar.

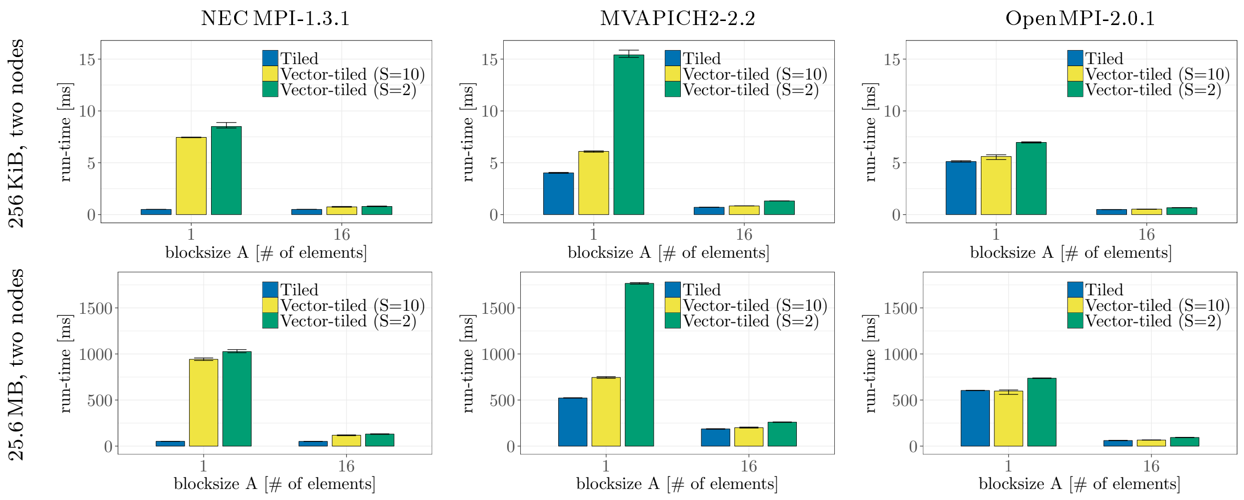


Figure 12: Exp. Test 6 – `Tiled` vs. `Vector-tiled`, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

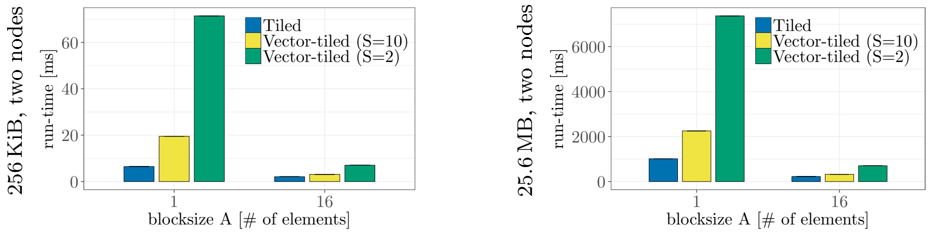


Figure 13: Exp. Test 6 – `Tiled` vs. `Vector-tiled`, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

Results. Much to our surprise, only the OpenMPI-2.0.1 seems able to normalize the nested vector into a flat, single vector or strided representation. This is seen in Figure 12 and Figure 13. For the other libraries, the case with the small $S = 2$ inner blocksize performs especially bad, most notably for $A = 1$ elements.

The penalty over the simple repetition of `Tiled` (A, B) is a factor of 3 for MVAPICH2-2.2, but it reaches a factor of 19 for NEC MPI-1.3.1. The same behavior can be observed on the *JUQUEEN* machine, where the communication performance with the `Vector-tiled` layout is up to 10 times worse than when using `Tiled` (A, B) for $A = 1$, regardless of the message size.

3.2.6. Expectation Test 7

The next two experiments are concerned with indexed descriptions of the more irregular `Block` and `Alternating` layouts. In the first experiment, we compare the repeated `Block` (A, B_1, B_2) description to an explicit description of the `Block` layout by explicitly listing the displacements and number of elements in all n/k blocks in the n element layouts. This dynamic datatype is set up using the `MPI_Type_create_-indexed_block` constructor. The purpose of this experiment and the next is to probe the penalty of having the MPI library traverse long, explicit lists of displacements and block sizes when processing the data layout descriptions.

Type description. The `Block-indexed` layout describes a `Block` layout with given A, B_1, and B_2 using the `MPI_Type_create_indexed_block` constructor with an array of n/A displacements and a blocksize of A. Since the block strides alternate between B_1 and B_2, the block displacements can easily be computed. This is illustrated in Table 3, Expectation Test 7.

Expectation. A good MPI library should normalize both cases to the same internal datatype representation with good performance, although it is not obvious what the best such representation might be. A reasonable expectation is that beyond some number of elements, the large array of displacements in the `Block-indexed` datatype will become expensive to traverse, and that simple repetitions of the small, irregular non-contiguous `Block` (A, B_1, B_2) pattern will perform better.

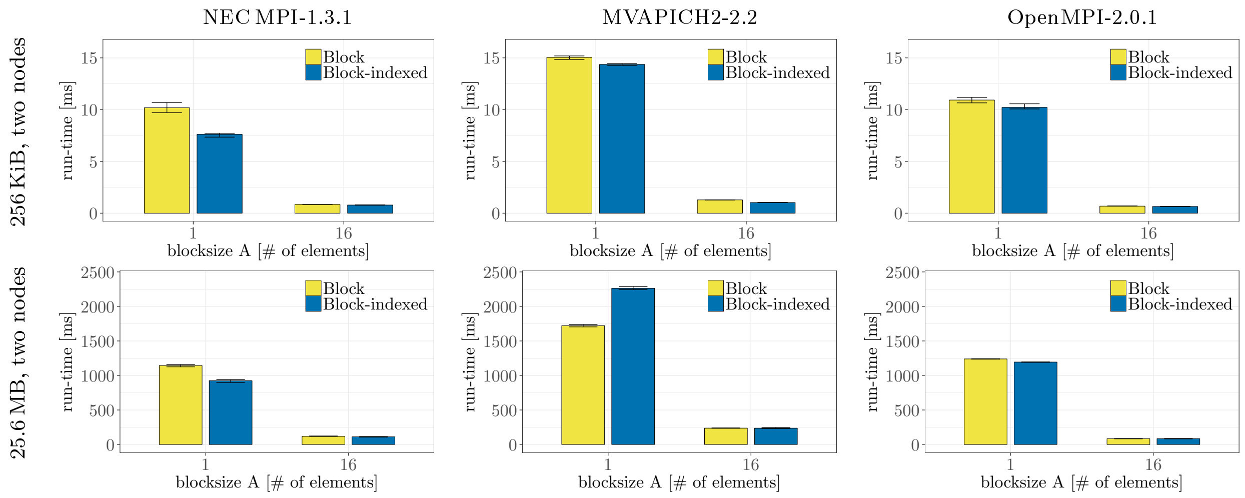


Figure 14: Exp. Test 7 – `Block` vs. `Block-indexed`, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

Results. The outcomes of these experiments are different for the different MPI libraries. As seen in Figure 14, the MVAPICH2-2.2 library behaves most closely to our expectations, with the `Block-indexed` description being slower for the large message size. The absolute performance of this library is on the other hand worse than the two other MPI libraries on *Jupiter*. The other libraries behave differently, especially with the IBM MPI it seems much better to use an `MPI_Type_create_indexed_block` description even when the index lists are very long, as can be seen in Figure 15. This may be due to normalization into some particularly efficient internal representation, which unfortunately is not done for the `Block` (A, B_1, B_2)

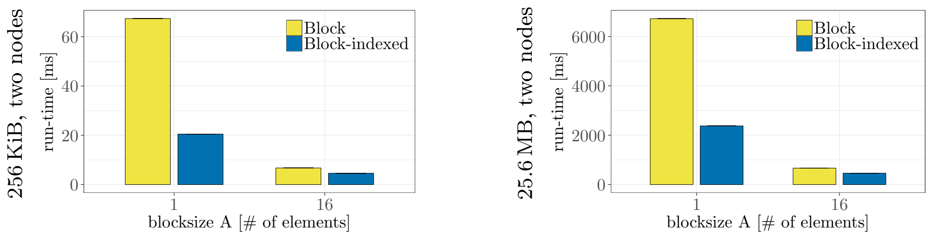


Figure 15: Exp. Test 7 – `Block` vs. `Block-indexed`, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

description. Such normalization might have been possible with the `Block` (A, B_1, B_2) embedded in a contiguous `Contiguous-subtype` type, but as can be seen in Expectation Test 3 (Figure 7), the IBM MPI does not perform any normalization for this type either.

3.2.7. Expectation Test 8

This experiment is similar to the previous one. Here, two descriptions of the "most irregular" of the four basic layouts, namely `Alternating` are contrasted.

Type description. The `Alternating-indexed` datatype describes an `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 alternating blocksizes of A_1 and A_2. The data layout is illustrated in Table 3, Expectation Test 8.

Expectation. As for the previous experiment, it is not obvious which of the two descriptions will perform better. Experimental results will give insight on whether there is a penalty for large arrays of displacements and blocksizes in the `MPI_Type_indexed` constructor.

Results. For the large message size, in contrast to the previous experiment, the libraries on *Jupiter* behave more as expected, as can be seen in Figure 16, with the `Alternating-indexed` being slower than the repeated `Alternating` (A_1, A_2, B_1, B_2) block datatype. The NEC MPI-1.3.1 library has a severe problem for the same node configuration, with the `Alternating-indexed` datatype being up to a factor 40 slower than when communicating with the `Alternating` (A_1, A_2, B_1, B_2) datatype for $A = 1$. This performance gap increases with the message size and correlates with the difference between the number of L3 cache misses obtained when communicating with the two datatypes. On *JUQUEEN*, the behavior is the other way around and similar to the previous experiment (see Figure 15), with the `Alternating-indexed` description being faster that the repeated `Alternating` (A_1, A_2, B_1, B_2) type. This could indicate that `MPI_Type_indexed` and `MPI_Type_create_indexed_block` constructors are treated in the same way by the IBM MPI library.

3.2.8. Expectation Test 9

In our final experiment we present a completely different view of the simple `Tiled` layout with A elements and stride $B = 3C$ (as in the previous experiments, with fixed $C = 16$). For this view, we have to assume that $A > 1$, and present our results for $A = 2$ and $A = 16$. The tiled layout is viewed as an initial, contiguous unit of one element, followed by a middle part of $n - A$ elements, followed by a last, contiguous unit of $A - 1$ elements, for a total of n elements. The middle part is again an almost regularly tiled layout, consisting of first a unit of $A - 1$ elements with a stride of $B - 1$, followed by a repetition of regularly strided units of A elements with stride B, followed by a last unit with only one element. The middle part can be described using the `MPI_Type_indexed` constructor with n/A blocks, or, as we do here, as a repetition of smaller indexed subtypes with fixed number of blocks. The layout description is illustrated in Table 3, Expectation Test 9. The parameter S determines the number of blocks and elements in the indexed type making up the middle part. This is $S + 1$ and SA, respectively. We call this involved description of the tiled layout `Tiled-struct-indexed`.

The `Tiled-struct-indexed` layout description illustrates a number of important points, and poses non-trivial challenges to MPI type normalization heuristics. First, the almost regularly tiled middle part can

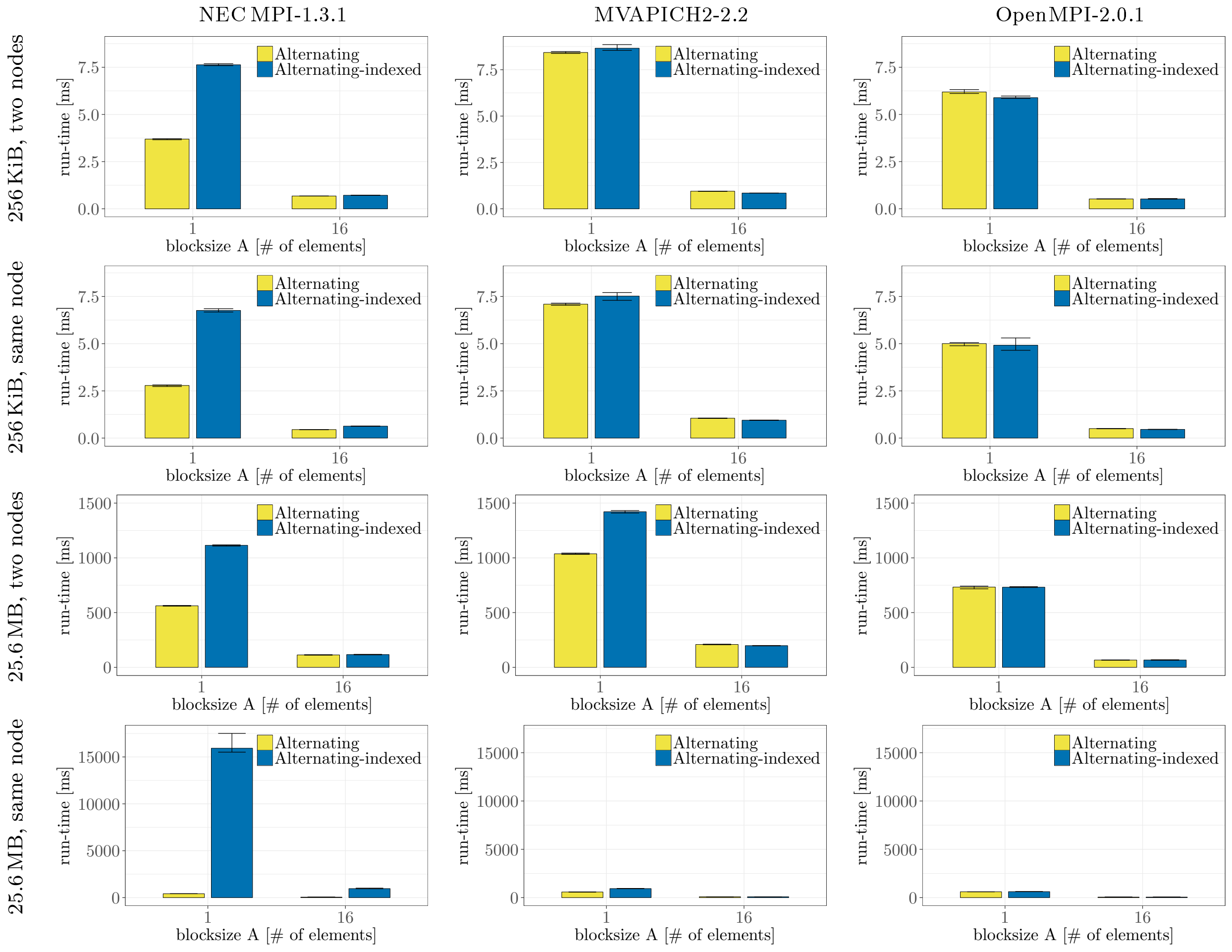


Figure 16: Exp. Test 8 – `Alternating` vs. `Alternating-indexed`, element datatype: `MPI_INT`, ping-pong, *Jupiter*.

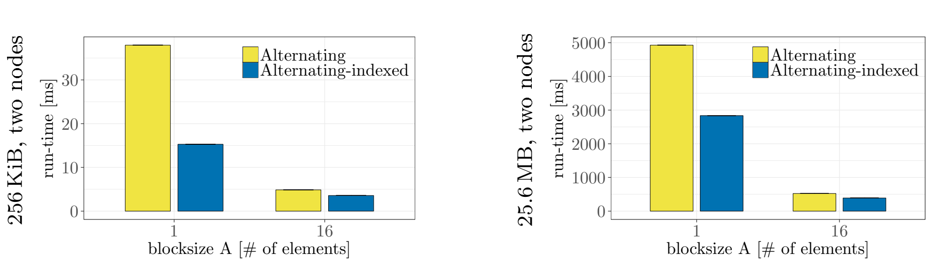


Figure 17: Exp. Test 8 – `Alternating` vs. `Alternating-indexed`, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

be described either as a) a large `MPI_Type_indexed` constructed datatype, as b) a repetition of smaller `MPI_-Type_indexed` constructed datatypes, or, with the `MPI_Type_create_struct` constructor, as c) small first and last units with $A-1$ and one elements, respectively, and a repeated `Tiled` (A, B) datatype in the middle. The latter description, which, due to the large, regular middle part, might lead to better communication performance, can only be expressed with the `MPI_Type_create_struct` constructor, even though the layout itself is homogeneous (using only one and the same element datatype). Concise descriptions of such layouts thus require a constructor like `MPI_Type_create_struct`. Good type normalization must be able to detect the tiled structure in the repetitions of the `MPI_Type_indexed` constructed datatypes in the middle part of the `Tiled-struct-indexed` datatype. This regularity is exposed only when it is known that the indexed datatype blocks are repeated a large number of times. This is why, in Guideline (GL1), a $\mathsf{contig}(c, t)$ datatype may perform better than giving only datatype t and the count argument c to the communication operation: A large count in the datatype can expose regularities that can possibly be exploited. Finally, type normalization must be able to detect that when the almost tiled middle part appears in the context of the smaller first and last parts, the whole complex datatype is in fact just a description of the simple, regularly `Tiled` layout.

Type description. The `Tiled-struct-indexed` layout description is parameterized in S giving the number of blocks, $S+1$, in the indexed datatype making up the middle part. To make it possible to compare directly with the repetition of `Tiled` (A, B) with a given number of n elements, the outer `MPI_Type_create_struct` constructor takes four blocks, namely a contiguous block of one element, a repetition of the indexed datatype with $S+1$ blocks, a block of $A-1$ elements, and a repetition of $S-1$ `Tiled` (A, B) blocks. We have created versions of `Tiled-struct-indexed` with $A = 2$ and $A = 16$ elements, and with $S = 1$ and $S = 8$, respectively. We verify that this complex datatype actually describes the `Tiled` layout using the type map functionality discussed in [23].

Expectation. As stated by Guideline (GL4), the `Tiled-struct-indexed` layout description should perform similarly to the best representation of `Tiled` (A, B). However, commonly used MPI type normalization heuristics cannot detect that the underlying layout is regularly tiled, and our actual, more realistic expectation is that the `Tiled-struct-indexed` will perform worse. The interesting outcome of the experiment is then rather how much worse the complex description is compared to the natural description as a repetition of `Tiled` (A, B) blocks, which effect (if any) the number $S+1$ of blocks in the middle part datatype has, and whether different MPI libraries differ in their handling of such complex descriptions.

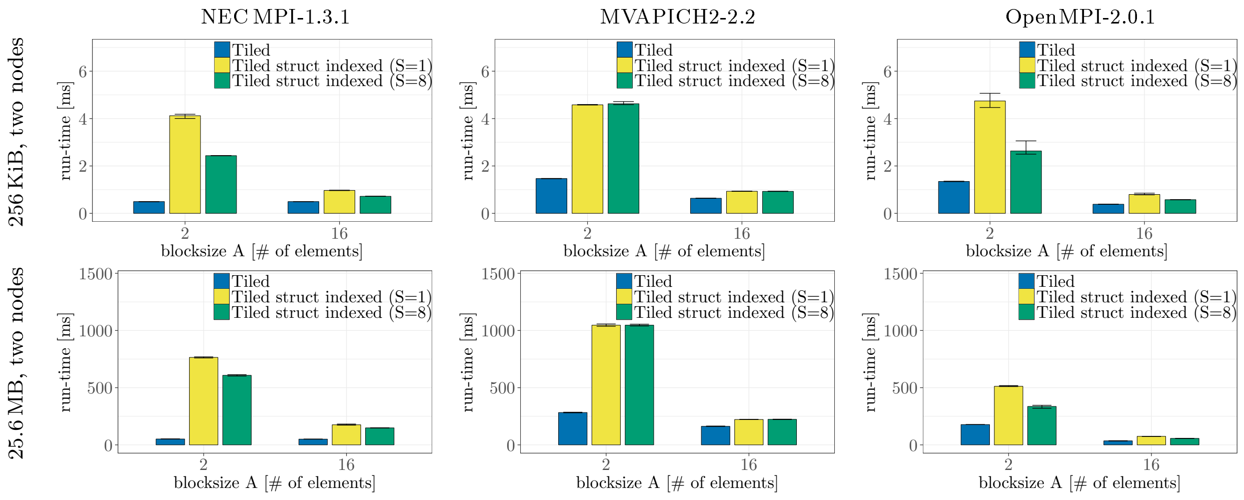
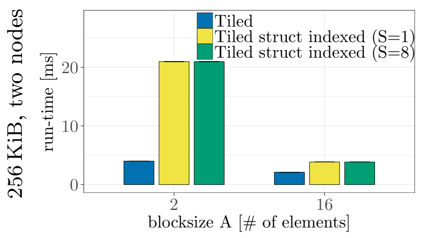


Figure 18: Exp. Test 9 – `Tiled` vs. `Tiled-struct-indexed`, element datatype: `MPI_INT`, ping-pong, *Jupiter*.



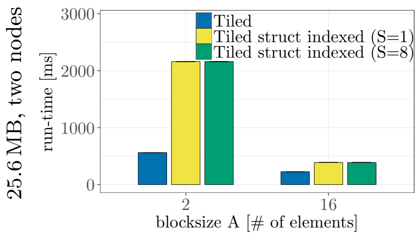


Figure 19: Exp. Test 9 – `Tiled` vs. `Tiled-struct-indexed`, element datatype: `MPI_INT`, ping-pong, IBM MPI, *JUQUEEN*.

Results. The results, shown in Figure 18 and Figure 19, compare the two `Tiled-struct-indexed` datatypes with $S = 1$ and $S = 8$ and unit sizes $A = 2$ and $A = 16$ against the simple `Tiled` (A, B) description of the tiled layout. On both machines, the complex description is significantly slower than the simple description, with a factor of about two to four (for $A = 2$). With the NEC MPI-1.3.1 and OpenMPI-2.0.1, a small middle part subtype $S = 1$ performs worse than the larger indexed subtype with $S = 8$, whereas for MVAPICH2-2.2 and IBM MPI the two layout descriptions perform identically. The results clearly show that none of the MPI libraries are able to find a good internal representation for the complex datatype descriptions.

4. Summary and Discussion

We presented a methodology to assess the quantitative and qualitative behavior of MPI communication of structured, non-contiguous data with MPI derived datatypes. Our benchmark consists of three communication patterns, and four basic, parameterizable data layouts. Describing these layouts as small building block datatypes allows us to assess the MPI data access and communication costs for different communication patterns and process configurations. The four derived datatype performance guidelines make it possible to relate the measured performance between different uses of derived datatypes and between different datatype descriptions of the layouts. A violated performance guideline points to undesirable behavior of an MPI library. Our benchmark contains a number of different descriptions of the four basic layouts, using different combinations of MPI datatype constructors in order to investigate the MPI library handling of, e.g., nested uses of the constructors and the various indexed and struct MPI datatype constructors. For each of these uses we formulate performance expectations based on the corresponding performance guideline.

We performed a large number of experiments on two different systems and four different MPI libraries, but reported here only results for the ping-pong communication pattern, although extensively. Although such results are only transitory, they are nevertheless revealing and in many cases point to concrete problems with the MPI libraries that should be addressed. Our framework can be used routinely on new library versions and perhaps guard the application programmer against surprises with the derived datatype performance. For instance, it was unexpected that Guidelines (GL2) and (GL3) would be violated, but we found many cases where they were severely compromised. In such cases, the recommendation to the application programmer to use datatypes is hard to justify. Sometimes, the simplest expectations concerning the use of the `MPI_Type_contiguous` constructor, captured in Guideline (GL1), were severely violated, and also simple uses of nested vectors were not handled well at all by the libraries (except for OpenMPI-2.0.1). We also observe that the communication performance with (non-trivial) derived datatypes is quite different between the libraries. For example, MVAPICH2-2.2 does not handle derived datatypes as efficiently as the other libraries on the *Jupiter* system.

Our experiments around Guideline (GL4) show that the way a given layout is described as a derived datatype matters a lot. Put differently, the heuristics employed by common MPI libraries in `MPI_Type_commit` are insufficient to find good internal datatype representations such that the guideline holds. It is worthwhile to improve this, since an application programmer needs a very good intuition to select a good derived datatype description in a given situation. Since the good derived datatype may be different between different systems and libraries, this is not portable. Simple rules of thumb are not enough: Our findings sometimes contradicted our own intuitions and expectations, e.g., indexed constructors with very long arrays of indices were not as harmful as we thought. Some of the experiments show that datatype descriptions of small blocks can be too localized to make a sufficiently good normalization possible. We constructed cases where both the (large) repetition count and the block datatype are needed to reveal a regular overall structure of the layout. Committing a contiguous type with the repetition count and block datatype gives the MPI library the possibility to discover such regularities and compute a more efficient, internal datatype representation, as is expressed in Guideline (GL1). Our experiments showed that libraries do not do this.

Our framework is not concerned with the time used in `MPI_Type_commit`, and datatype setup time was not measured. For the usefulness of the derived datatype mechanism, set up time is of course crucial. It would therefore useful to estimate the amortization point of `MPI_Type_commit`, that is, how often and how much must be communicated in order to offset the type set up costs. There is a tradeoff here: Optimal type normalization (under a simple cost model accounting for space consumption) is time consuming [15]. The MPI standard currently does not offer any means of influencing what should be done by `MPI_Type_commit`, as has often been pointed out [17, 22]. Also, the MPI standard does not give any means of accessing the internal representation of derived datatypes.

Acknowledgments

We thank Bernd Mohr (Julich Supercomputing Centre) for providing us access to a IBM BlueGene/Q.

References

[1] A. Carpen-Amarie, S. Hunold, J. L. Träff, On the expected and observed communication performance with MPI derived datatypes, in: Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI), ACM, 2016, pp. 108–120.

[2] MPI Forum, MPI: A Message-Passing Interface Standard. Version 3.1, `www.mpi-forum.org` (June 4th 2015).

[3] W. D. Gropp, T. Hoefler, R. Thakur, J. L. Träff, Performance expectations and guidelines for MPI derived datatypes: a first analysis, in: Proceedings of the 18th European MPI Users' Group Meeting (EuroMPI), Vol. 6960 of Lecture Notes in Computer Science, Springer, 2011, pp. 150–159.

[4] R. Reussner, J. L. Träff, G. Hunzelmann, A benchmark for MPI derived datatypes, in: Proceedings of the 7th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Vol. 1908 of Lecture Notes in Computer Science, Springer, 2000, pp. 10–17.

[5] T. Schneider, R. Gerstenberger, T. Hoefler, Application-oriented ping-pong benchmarking: how to assess the real communication overheads, Computing 96 (4) (2014) 279–292.

[6] T. Hoefler, S. Gottlieb, Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes, in: Proceedings of the 17th European MPI Users' Group Meeting (EuroMPI), Vol. 6305 of Lecture Notes in Computer Science, Springer, 2010, pp. 132–141.

[7] M. Schulz, G. Bronevetsky, B. R. de Supinski, On the performance of transparent MPI piggyback messages, in: Proceedings of the 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Vol. 5205 of Lecture Notes in Computer Science, Springer, 2008, pp. 194–201.

[8] S. Byna, W. D. Gropp, X.-H. Sun, R. Thakur, Improving the performance of MPI derived datatypes by optimizing memory-access cost, in: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), IEEE Computer Society, 2003, pp. 412–419.

[9] T. Prabhu, W. Gropp, DAME: A runtime-compiled engine for derived datatypes, in: Proceedings of the 22nd European MPI Users' Group Meeting (EuroMPI), ACM, 2015, pp. 4:1–4:10. doi:10.1145/2802658.2802659.

[10] R. Ross, N. Miller, W. D. Gropp, Implementing fast and reusable datatype processing, in: Proceedings of the 10th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Vol. 2840 of Lecture Notes in Computer Science, Springer, 2003, pp. 404–413.

[11] R. B. Ross, R. Latham, W. Gropp, E. L. Lusk, R. Thakur, Processing MPI datatypes outside MPI, in: Proceedings of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Vol. 5759 of Lecture Notes in Computer Science, Springer, 2009, pp. 42–53.

[12] T. Schneider, F. Kjolstad, T. Hoefler, MPI datatype processing using runtime compilation, in: Proceedings of the 20th European MPI Users's Group Meeting (EuroMPI), ACM, 2013, pp. 19–24.

[13] J. L. Träff, R. Hempel, H. Ritzdorf, F. Zimmermann, Flattening on the fly: efficient handling of MPI derived datatypes, in: Proceedings of the 6th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Vol. 1697 of Lecture Notes in Computer Science, Springer, 1999, pp. 109–116.

[14] J. Wu, P. Wyckoff, D. K. Panda, High performance implementation of MPI derived datatype communication over InfiniBand, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2004, p. 14.

[15] R. Ganian, M. Kalany, S. Szeider, J. L. Träff, Polynomial-time construction of optimal MPI derived datatype trees, in: Proceedings of the 30th International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2016, pp. 638–647.

[16] M. Kalany, J. L. Träff, Efficient, optimal MPI datatype reconstruction for vector and index types, in: Proceedings of the 22nd European MPI Users' Group Meeting (EuroMPI), ACM, 2015, pp. 5:1–5:10. doi:10.1145/2802658.2802671.

[17] J. L. Träff, Optimal MPI datatype normalization for vector and index-block types, in: Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA), ACM, 2014, pp. 33–38.

[18] J. L. Träff, W. D. Gropp, R. Thakur, Self-consistent MPI performance guidelines, IEEE Transactions on Parallel and Distributed Systems 21 (5) (2010) 698–709.

[19] A. Carpen-Amarie, S. Hunold, J. L. Träff, MPI derived datatypes: Performance expectations and status quo, CoRR abs/1607.00178.

[20] S. Hunold, A. Carpen-Amarie, Reproducible MPI benchmarking is still not as easy as you think, IEEE Transactions on Parallel & Distributed Systems 27 (12) (2016) 3617–3630. doi:10.1109/TPDS.2016.2539167.

[21] F. Kjolstad, T. Hoefler, M. Snir, A transformation to convert packing code to compact datatypes for efficient zero-copy data transfer, Tech. rep., University of Illinois at Urbana-Champain, retrieved from `http://hdl.handle.net/2142/26452`, last visited on 03/01/2016 (2011).

[22] F. Kjolstad, T. Hoefler, M. Snir, Automatic datatype generation and optimization, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, 2012, pp. 327–328.

[23] J. L. Träff, A library for advanced datatype programming, in: Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI), ACM, 2016, pp. 98–107.