Library Development with MPI: Attributes, Request Objects, Group Communicator Creation, Local Reductions and Datatypes

Jesper Larsson Träff
traff@par.tuwien.ac.at
Faculty of Informatics
TU Wien
Vienna, Austria

Ioannis Vardas*
vardas@par.tuwien.ac.at
Faculty of Informatics
TU Wien
Vienna, Austria

ABSTRACT

A major design objective of MPI is to enable support for the construction of safe parallel libraries that can be used and mixed freely in complex applications. In this respect, MPI has been extremely successful; but may nevertheless lack elementary supporting functionality for some situations, and may have made design choices that are difficult to accommodate in certain libraries. We discuss several cases of library construction requiring different kinds of supporting MPI functionality, and propose concrete improvements for library implementations and future MPI versions to alleviate the problems that were encountered. Specifically, we pinpoint (performance) issues with MPI object attributes, caching and lookup, request objects, partly collective and non-blocking communicator creation, process local reductions, type correct process local copying, and user-defined datatypes.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Computing methodologies → Parallel programming languages**.

KEYWORDS

MPI, library building, attributes, request objects, collective operations, derived datatypes

ACM Reference Format:
Jesper Larsson Träff and Ioannis Vardas. 2023. Library Development with MPI: Attributes, Request Objects, Group Communicator Creation, Local Reductions and Datatypes. In *Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting (EUROMPI '23), September 11–13, 2023, Bristol, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3615318.3615323

1 INTRODUCTION

A major design objective of MPI, the *Message-Passing Interface* [15], is to enable support for the construction of safe parallel libraries that can be used and mixed freely in complex applications without any possibilities of unintended, destructive interference, neither concerning local objects and data nor concerning communication. Libraries may likewise implement new abstractions and operations that better fit the intended application domains, and hide complexities of MPI communication that are irrelevant to the application programmer. In that respect, MPI has been extremely successful, as witnessed by libraries for a plethora of different types of applications; see for instance the cases provided by Hoefler and Snir [11] in their discussion of library construction with MPI. One key MPI construct for safe communication libraries is the *communicator* (and *communication window*), which safely isolates communication between processes in different application libraries. However, much additional, surrounding functionality is needed for the construction of portable libraries that use and expose (or hide) various MPI objects, and much such support functionality is indeed provided by the MPI standard.

The list of widely used parallel libraries for different scientific domains built using MPI is long. Some examples of numerical (linear algebra) libraries include ScaLAPACK [5], Elemental [17], PETSc [2], and many, many more. Libraries for machine learning, such as PyTorch [16], can be built with MPI as the underlying communication library for distributed model training and applications. Such libraries usually hide MPI specifics by using the supporting functionality of MPI for library construction.

An instructive test of adequate library building support is to examine which parts of the MPI standard itself can be implemented efficiently and transparently to the user with full-fledged support and full safety guarantees, in terms of other, more basic MPI functionality, notably point-to-point and one-sided communication and certain book-keeping operations, e.g., for communicator and datatype management.

As discussed in the MPI standard [15, Section 7.7], the whole set of blocking, collective operations, including a hidden shadow communicator for isolating library specific (non-blocking) communication from the user's communication, can be implemented by point-to-point communication, the `MPI_Comm_dup` function, and the communicator attribute caching mechanism. This is concretely done in many libraries offering collective functionality, for instance in the library described in [22] that gives alternative, hierarchical implementations for all regular, blocking MPI collectives, and uses two hidden orthogonal communicators per process.

The MPI virtual topology functionality can likewise be implemented entirely and transparently on top of MPI, as an external library, again by relying heavily on the attribute caching functionality. The MPI standard itself prescribes that topology information

*This work was partially supported by the Austrian Science Fund (FWF): project P 31763-N31.

EUROMPI '23, September 11–13, 2023, Bristol, United Kingdom

ACM ISBN 979-8-4007-0913-5/23/09.
https://doi.org/10.1145/3615318.3615323

(Cartesian or distributed graph) is attached to the created communicators in the form of MPI attributes [15, Section 8.3]. It can therefore be expected that many production MPI libraries indeed build on their own implementation of the attribute caching mechanism. The blocking, neighborhood collectives that build on the topology functionality can likewise be implemented transparently and entirely as an external library, again relying on `MPI_Comm_dup` and the attribute caching mechanism. This was partly done in [24] which proposes a more economic use of the MPI interfaces and suggests and implements extensions to the expressive power of the (Cartesian) topology mechanism. Such proposals can, due to the strong MPI library building support, be explored as external libraries.

Non-blocking collective operations, as later specified in the MPI standard, were initially explored as external MPI libraries [9]. The implementations involve checking for and enforcing the completion of multiple outstanding communication requests, as well as deliberate posting of further, non-blocking communication operations, which makes a fully transparent, external library implementation of this functionality as now specified by the MPI standard difficult. See additional discussion and proposals in [10].

Libraries with non-blocking, collective semantics pose additional, interesting requirements to supporting MPI functionality. To ensure communication safety, hidden shadow communicators are convenient and can be attached to the library communicator as attributes as explained above. In order for all collective library calls to indeed satisfy the non-blocking requirements, all communicator creation functionality required to manage these hidden communicators must be non-blocking as well. Currently, the MPI standard defines a non-blocking `MPI_Comm_idup` operation. For more complex, communicator splitting operations like `MPI_Comm_split`, there are currently no non-blocking counterparts in MPI. A truly non-blocking version of the collective library described in [22] (two non-trivial communicators per process), for instance, would thus not be implementable with the current MPI standard.

Also initially explored outside the MPI standard was MPI supported functionality for collective IO. Much of this was implemented in a fully portable manner in the ROMIO library by relying on MPI functionality for library building and user-defined datatypes [18].

In the rest of this paper, we recount recent experience with building communication and tool libraries on top of MPI (and for MPI applications). We examine in more detail some of the requirements to the MPI support for building transparent and efficient parallel libraries, in particular in situations where we found that support from current MPI [15] is lacking.

2 ATTRIBUTES

Attributes make it possible to associate certain information, as key-value pairs, locally, with certain objects defined by MPI. The attribute caching mechanism is essential for libraries that allocate MPI objects that are possibly hidden from the library user, but need to survive between library operations that (re-)use these objects.

MPI makes it possible to use attributes with three types of MPI objects: Communicators, windows, and datatypes. Attributes for other MPI objects, e.g., `MPI_Request` objects not are supported, which may be a lack as will be discussed in Section 3. For these three types of objects, functions for generating a new key, attaching and detaching attribute values, and deleting a key are provided [15, Section 7.7]. Associated attribute values can be arbitrary (data structures) represented by a pointer to the attribute content, and are copied and deleted via user-defined call-back functions. In addition to the attributes, MPI defines special functionality to name objects of these three types by a string, e.g. `MPI_Comm_set_name`/`MPI_Comm_get_name` [15, Section 7.8]. With the flexibility of the attribute mechanism, this functionality seems superfluous. There are, however, some reasons not to handle naming by attributes as explicitly rationalized in the MPI standard [15, Section 7.8], and for these reasons, the MPI standard chose to provide both attributes and explicit naming.

A concrete example of communicator attribute usage is a recently developed library for collective operations for clustered, hierarchical systems [22] that was already mentioned in the introduction. This library provides regular collective operations for any communicator and requires a decomposition of the given communicator into a set of communicators for the compute nodes (node communicators), and a set of communicators between single processes of each compute node (lane communicators; for the actual details that are not important here, see [22]). This communicator decomposition into lane- and node-communicators is computed once and for all at the first call to a collective operation using `MPI_Comm_split_type` and `MPI_Comm_split`. Each process in the calling communicator will belong to two subcommunicators which it attaches to the calling communicator as an attribute. Any subsequent call to a library function of this library then looks up the required subcommunicators and proceeds with communication on these subcommunicators. For this to work, a new attribute key is created by the first call of a library function (in a static variable). For this particular library, an explicit library initialization call is therefore not needed, and the user can do this particular type of collective communication as if the functions were just some MPI collective operations. The two subcommunicators function as hidden shadow communicators, exactly as commonly used in the native implementations of the MPI collectives, but neither span all processes of the calling communicator. As mentioned in the MPI standard [15, Section 7.7], and demonstrated by this library, the (blocking) collective operations can be implemented entirely transparently as a library building on simpler MPI functionality.

For such a library to be used in production, where collective calls are frequent and latency-critical, attribute retrieval, in particular, must be fast. There seems to be no empirical performance studies available similar to the work of Balaji et al. [3], for instance, on the quality (speed) of the MPI attribute caching mechanism.

We have created a simple, synthetic benchmark to measure the cost of sequences of attribute operations: key creation, attribute setting and getting (retrieval), and final deletion. The benchmark does not exercise the key-value copy and delete call-back functions, and uses only the simple, built-in functions provided by MPI. The benchmark takes two parameters, the number n of attribute keys (and attributes) to be created, and m the number of attribute lookup operations. It measures attribute operation time over a number of repetitions. In each repetition, an MPI object of the type to be investigated (here: communicator) is created; the time for this is

not counted. The processes are then synchronized with an `MPI_Barrier` operation. Several n attribute keys are generated, and for each of these, an attribute value in the form of a pointer to a predefined table entry is attached. This is a total of $2n$ attribute operations. After that, m lookup sequences are performed. In each sequence, first the n attribute keys are looked up in order of creation, and then looked up again in decreasing order of creation. This amounts to a total of $2mn$ attribute operations. Finally, the n attributes are deleted, and the n keys are freed, again for a total of $2n$ attribute operations. A total of $2mn + 4n$ attribute operations are therefore performed per repetition.

We measure the total time per repetition, and we divide it by the counted number of attribute operations. In this, and all other benchmarks, we do five (5) initial "warm-up" repetitions that are not measured in order to let the MPI library become properly initialized. We record the time of the slowest process, and from this, we compute the average time over all repetitions, as it is the minimum, best time over all repetitions.

We conducted our experiments on single nodes of two HPC clusters, our own small *Hydra* cluster at TU Wien and *LUMI*[1] (Large Unified Modern Infrastructure) located at CSC in Finland. *Hydra* consists of 36 dual socket, 16-core Intel(R) Xeon(R) Gold 6130F compute nodes. The nodes are interconnected via Intel Omnipath with a bandwidth of 100 Gibt/ and are running Debian 11 GNU/Linux with kernel v5.10.0 as the OS. In *LUMI*, we used the CPU partition (*LUMI-C*) which consists of 1536 compute nodes, each of which is equipped with two AMD EPYC 7763 CPUs with 64 cores each running at 2.45 GHz for a total of 128 cores per node. *LUMI-C* compute nodes are connected via a single 200 Gibt / s interface to the Slingshot-11 interconnect and are running SUSE Linux Enterprise Server 15 SP4 with kernel v5.14.21. We used five different MPI libraries, namely OpenMPI 4.1.4, MPICH 4.0.2, MVAPICH2 2.3.7, IntelMPI 2021.8, and Cray MPICH 8.1.23. The latter four libraries are originally based on `mpich2`. We ran our experiment with Cray MPICH 8.1.23 on the *LUMI* supercomputer and used *Hydra* for the other four libraries.

The results with the five different MPI libraries with different numbers of processes p on a single node, different $n = 10, 100, 1000$ and $m = 1000$ and $r = 95$ repetitions are shown in Tables 1–5. Running times are in microseconds and measured with the `MPI_Wtime` function. The first column is the average time for the $r = 95$ repetitions, and the second column is the best time seen over the repetitions.

For all five libraries and both machines, there seems to be no (or only very little) dependence on the number of processes per node (and the number of processes overall, not documented here). This is as it should be since the attribute caching mechanism is a purely process-local facility. There is also very little difference between the average time and the best seen time over the $r = 95$ repetitions, which indicates that the facility is robust (for this experiment, the warmup repetitions do not seem to matter, results are stable). There are, however, significant differences in raw performance between the tested libraries.

The results for the MPICH 4.0.2, MVAPICH2 2.3.7 and IntelMPI 2021.8 libraries are shown in Tables 1, 2, and 3. The behavior is quite similar, which is perhaps not surprising, considering the common ancestry of these MPI libraries. The time per operation for a small number of $n = 10$ attributes is small, but increases linearly (and even more) with increasing n, and becomes more than a factor of 100 slower for $n = 1000$ attributes. This is not a desired behavior, and the operation time of more than 2 μs per operation could be a factor for low latency operations. Attribute lookup time should in the best case be constant, or depend only weakly on n (say, $O(\log n)$ by some efficient tree data structure).

The results for Cray MPICH 8.1.23 are shown in Table 4. This library is faster than the others, performing well for a small number

[1] https://lumi-supercomputer.eu

Table 1: Attribute operation time with MPICH 4.0.2.

p	n	m	r	Ops	Time/Op (μs)	
					Avg	Min
1	10	1000	95	20040	0.021	0.021
1	100	1000	95	200400	0.165	0.165
1	1000	1000	95	2004000	2.124	2.104
32	10	1000	95	20040	0.023	0.023
32	100	1000	95	200400	0.171	0.170
32	1000	1000	95	2004000	2.137	2.123

Table 2: Attribute operation time with MVAPICH2 2.3.7.

p	n	m	r	Ops	Time/Op (μs)	
					Avg	Min
1	10	1000	95	20040	0.026	0.025
1	100	1000	95	200400	0.156	0.155
1	1000	1000	95	2004000	2.063	2.048
32	10	1000	95	20040	0.026	0.026
32	100	1000	95	200400	0.164	0.163
32	1000	1000	95	2004000	2.091	2.083

Table 3: Attribute operation time with IntelMPI 2021.8.

p	n	m	r	Ops	Time/Op (μs)	
					Avg	Min
1	10	1000	95	20040	0.027	0.027
1	100	1000	95	200400	0.160	0.159
1	1000	1000	95	2004000	2.079	2.066
32	10	1000	95	20040	0.028	0.027
32	100	1000	95	200400	0.163	0.162
32	1000	1000	95	2004000	2.098	2.090

Table 4: Attribute operation time with Cray MPICH 8.1.23.

p	n	m	r	Ops	Time/Op (μs)	
					Avg	Min
1	10	1000	30	20040	0.012	0.012
1	100	1000	30	200400	0.073	0.072
1	1000	1000	30	2004000	0.765	0.748
128	10	1000	30	20040	0.062	0.024
128	100	1000	30	200400	0.119	0.108
128	1000	1000	30	2004000	0.878	0.857

Table 5: Attribute operation time with OpenMPI 4.1.4.

p	n	m	r	Ops	Time/Op (μs)	
					Avg	Min
1	10	1000	95	20040	0.047	0.046
1	100	1000	95	200400	0.046	0.046
1	1000	1000	95	2004000	0.053	0.053
32	10	1000	95	20040	0.048	0.047
32	100	1000	95	200400	0.047	0.046
32	1000	1000	95	2004000	0.054	0.054

of attribute keys, but again its performance decreases almost linearly with the number of keys. In absolute terms, the *LUMI* seems substantially faster (for this type of operations) than *Hydra*.

Finally, the results for the OpenMPI 4.1.4 library are shown in Table 5. In contrast to the four other libraries, the OpenMPI 4.1.4 library exhibits (almost) constant time per attribute operation, independent of n (and of p), as should be. However, the time for small n is about a factor of two higher, so the better, more scalable implementation punishes use-cases with small number of attributes.

3 REQUEST OBJECTS

A recently developed profiling library [25] profiles communication operations for the communicators on which they are invoked. The implementation of this library, which uses the old-fashioned MPI profiling interface posed two problems for non-blocking operations: Where to account for the time spent in `MPI_Wait` etc. operations, and how to associate this time with the corresponding non-blocking call (as far as this is possible and meaningful)? From the request object of the `MPI_Wait` etc. calls, we aim to associate the time spent on that call with the communicator (and even the specific operation) for which the request object was created.

The MPI (check-for-)completion operations do not take a communicator argument, only an `MPI_Request` object as an input parameter from which a corresponding `MPI_Status` object may be set. Neither `MPI_Request` nor `MPI_Status` objects give access to the communicator from which the object was created. To make the association of the request object with the communicator and operation possible, the ability to cache information from the creation call with the request object itself would be extremely valuable, but this is not possible with the current MPI.

The MPI standard does not give a rationale for why attributes are not defined for `MPI_Request` objects, nor why the mechanism is limited to only three types of MPI objects (communicators, windows, datatype). The first two of these are distributed objects, and the last is an object created by purely local operations, so non-locality or dependent behavior cannot be the argument. Process local MPI group objects (`MPI_Group`) also do not allow attributes.

What was needed for the concrete library [25] is a way to determine or extract the communicator (or window) that was used to instantiate a given `MPI_Request` object. We found no way to circumvent the problem with standardized MPI functionality, neither with generalized requests [15, Section 13.2], nor with the tool information interface [15, Section 15.3]. The generalized requests are useful for user-defined non-blocking operations but do not solve the problem of associating the `MPI_Request` with the communicator. The MPI tool information interface [15, Section 15.3] exposes performance information depending on the MPI implementation, however, through it, we were unable to store new information, therefore, it was not suited for our profiling library.

Our solution for the profiling library [25] was to manually mimic an attribute mechanism, and use an additional hash table with the request object handle as a key to store the communicator for which the request object should be associated. This works for the concrete case and the concrete MPI library implementations for which we have used the profiler, but is not a sound approach in general, since some MPI operations are allowed to give back a new request handle still representing the same request. This is potentially possible for operations that pass an `MPI_Request` object as a reference, e.g., `MPI_Start` to start a persistent operation on the request object.

For a more elegant and portable solution to such problems, we make either of two non-exclusive proposals: Either a specific function for `MPI_Request` objects that can return the communicator for which the request object was created, similar to `MPI_Comm_group` (with a similar function for windows), or an extension of the attribute mechanism to `MPI_Request` (and possibly other MPI) objects.

4 GROUP COMMUNICATOR CREATION

In the profiling library [25] mentioned in Section 3, we encountered another interesting problem, to which we did not find a good solution. The aim of the library is to profile (all) communication relative to the communicator on which they were performed. For that, we need to associate profiling information, e.g., a table indexed by operations with local time, number of calls and data volume, with each new communicator created by the profiled application, in a way that this information can be collected and retrieved at the end of the application.

Profiling information is collected per process and stored per process in a table per communicator. When the communicator is freed, at the latest at application termination, the profiling information from all processes is summarized per communicator. For this, we needed to generate a unique "name" (key or index) for each new generated communicator, which must be the same for all processes belonging to the same communicator. We devised an algorithm to generate such unique keys at communicator creation time by a collective call over the calling communicator. Our solution is explained in detail in [25]; a different solution for essentially the same problem was given previously in [7].

For the standard, blocking communicator creation operations like `MPI_Comm_dup`, `MPI_Comm_create`, and `MPI_Comm_split`, this is not a problem, since these operations are collective over all calling processes. Each process contributes information (rank, number of subcommunicators) to the generation of the new key(s) for the created subcommunicators, and the uniqueness of the new keys are then guaranteed by an `MPI_Allreduce` operation over the calling processes. These operations are by the way all blocking.

The operations `MPI_Comm_create_group` and `MPI_Comm_create_from_group` have a different semantics, and are problematic in this scheme. The `MPI_Comm_create_group` and `MPI_Comm_create_from_group` operations are collective over the groups given in the

call and *not* over all processes in the calling communicator; for MPI_Comm_create_from_group there is no calling communicator at all! These are exceptional MPI functionalities (in some sense violating basic principles of MPI, namely that new objects are created out of old ones of the same kind), and there is therefore no way to apply a scheme that is based on a global MPI_Allreduce operation. A non-blocking MPI_Iallreduce could be used in our key generation to semantically solve the problem for MPI_Comm_create_group, but would entail blocking behavior (delay) when processes in a subcommunicator created by a process group perform profiling operations before the processes not belonging to the process group have completed the MPI_Comm_create_group and the pending MPI_Iallreduce calls. This is incommensurate with a fast profiling library, and an unacceptable solution. Because of the special semantics of MPI_Comm_create_group and MPI_Comm_create_from_group, our profiling library [25] currently does not handle code with these MPI calls.

In retrospect, we could have circumvented the problems by relying more on what an MPI implementation already does. First, for all communicator creation operations, including the problematic MPI_Comm_create_group and MPI_Comm_create_from_group calls, some form of internal, unique context identification for each created communicator is already created by the MPI implementation [6]. Profiling information per communicator could, therefore, be attached to the newly created (in that sense unique) communicator as an attribute for each process locally. Each profiled call would need to look up the profiling attribute on the communicator, and update the local profile, which again motivates strongly why a quick attribute lookup mechanism is essential. Per communicator, the local profiles are summarized together as profiling communication for that communicator, which can be done by a collective MPI_Gather, MPI_Reduce or MPI_Allreduce operation, possibly with a (commutative, or non-commutative, see discussion in Section 5) user-defined reduction operation at the time when the communicator is freed by MPI_Comm_free. After the free operation, profiling information for this (sub)communicator would be available at some local root process of the communicator and could be transferred as an attribute to the MPI_COMM_WORLD communicator of that process with a proper name identifying this particular communicator. A convenient name to identify the (sub)communicator could easily be composed locally from the rank of the process in MPI_COMM_WORLD, the running number of the communicator created by that process, its size, the type of communicator-creating operation used and the size of the subcommunicator.

Although the MPI_Comm_free operation is technically collective, it is mostly implemented as a purely local, at least non-blocking operation, and is typically used in a very asynchronous fashion (if at all). Therefore, forcing blocking behavior through a blocking MPI_Reduce collective call could badly hurt application performance, and would not be a good solution for a fast, low-overhead profiling library (recall the discussion in Section 1). This is again a use-case for non-blocking collective operations. A non-blocking MPI_Ireduce operation would have to be initiated by MPI_Comm_free, giving rise to a request object that would have to be stored, and tested, by the process in the MPI_COMM_WORLD communicator. Upon completion, the profiling information for the subcommunicator would have to be transferred to MPI_COMM_WORLD.

5 LOCAL REDUCTIONS

Libraries implementing reduction-like collective operations can and do benefit from the MPI mechanism for locally applying MPI binary operators (MPI_Op) on MPI buffers. Operators are either the standard, predefined MPI operators (MPI_SUM, MPI_BAND, ...) or user-defined operators that are allowed to operate on structured data described by MPI user-defined datatypes. Local application of these operators was made possible with the MPI_Reduce_local operation which is absolutely essential for such library building [11].

To mimic the flair of the various collective reduction operations (MPI_Reduce, MPI_Allreduce, ...), the MPI_Reduce_local functionality unfortunately implements only a two-argument, destructive update functionality. This is both inconvenient and, in some cases inefficient, since it forces copy operations into temporary buffers, or temporary copy out of data in reduction buffers that should not be destroyed. Especially for non-commutative operators (user-defined operators may well be non-commutative) where the order, from-left or from-right, in which an operator is applied is crucial, it is easy to come into a situation where a result has to be computed in a temporary buffer and later copied back.

Here is a concrete example. A butterfly/hypercube algorithm for some reduction collective goes through a logarithmic number of iterations, in each of which each process receives a partial result from a neighboring process, found by flipping the kth bit of the process' rank for iteration $k, 0 \leq k < \log p$ (for p processes and p being a power of 2). The neighboring process may, by the bit flip, thus have a rank that is smaller than the process's rank (if the bit was 1), or larger than the process's rank (the bit was 0). In the code snippet below, each process maintains a partial result in its recvbuf. It sends this to its neighbor and receives a partial result in tempbuf from its neighbor. To compute a new partial result for the next iteration, the two partial results are "added" together using the given MPI operator op – in the right order, which is strictly necessary if op is non-commutative.

```
int neighbor = (rank^k); // flip bit k
MPI_Sendrecv(recvbuf,...,neighbor,
             tempbuf,...,neighbor,...,comm);
if (neighbor<rank) {
  MPI_Reduce_local(tempbuf,recvbuf,...,op);
} else { // neighbor>rank
  MPI_Reduce_local(recvbuf,tempbuf,...,op);
  // now typed MPI copy back to recvbuf needed
  MPICPY(tempbuf,recvbuf,...);
}
```

In the code, MPICPY denotes type correct copying of MPI buffers, for which there is some discussion in Section 6. This additional copy can roughly double the local time taken for a process, for some iterations, and lead to delays in the entire algorithm. For large buffers this can be significant. We note, though, that in the concrete butterfly code, the extra copies can be partly avoided by introducing a double buffering scheme with two temporary buffers together with a more complex control structure to keep track of the buffer containing the partial result for the next iteration. This may not be possible for other algorithms, and is still an unnecessary complication of the algorithm [23].

A more natural and convenient solution would be to introduce (or replace MPI_Reduce_local by) a three-argument local reduction operation, which could look as follows[2].

```
int NEW_Reduce_locals(const void* inbuf,
                      const void *argbuf,
                      void* inoutbuf, int count,
                      MPI_Datatype datatype,
                      MPI_Op op)
```

The NEW_Reduce_locals applies the operation given by the MPI built-in or user-defined operator op element-wise to the elements of inbuf and argbuf in that order with the result stored element-wise in inoutbuf, as explained for user-defined operations in [15, Section 6.9.5]. The inbuf, argbuf and inoutbuf (the two inputs as well as the result) have the same number of elements given by count and the same datatype given by datatype. If the MPI_IN_PLACE option is given for either inbuf or argbuf (or both), the corresponding input is taken from inoutbuf. The MPI_IN_PLACE option is not allowed for the inoutbuf argument. The inbuf and argbuf buffers are not required to be distinct but must be distinct from the inoutbuf argument. This definition follows the spirit of other MPI reduction and communication operations (no overlapping send- and receive-buffers). This is possible by using the MPI_IN_PLACE option.

In applications (libraries, typically) applying local reductions with MPI predefined operators, a specific order of the arguments may be required, and some argument may not be placed in the required input or output buffer. In such cases, the two-argument MPI_Reduce_local function necessitates extra copying of either or both arguments. The three-argument function alleviates such extra copying. A call to MPI_Reduce_local can always be replaced by a call to NEW_Reduce_locals with MPI_IN_PLACE as the second input argument. By this observation MPI_Reduce_local could be deprecated in favor of NEW_Reduce_locals.

The proposed three-argument local reduction operation is considerably more flexible than MPI_Reduce_local, as shown by the following five use-case examples: Let A, X, Y be the buffers provided for inoutbuf, inbuf, and argbuf, respectively. The following reduction-assignments can readily be implemented with NEW_Reduce_locals:

- A = X op Y (with MPI_Reduce_local this would require first copying Y into A)
- A = A op Y (if X is MPI_IN_PLACE; with MPI_Reduce_local this would require first reducing into Y destructively, and then copying the result Y into A)
- A = X op A (if Y is MPI_IN_PLACE; this has the same effect as MPI_Reduce_local)

and even

- A = X op X (if Y is the same as X)
- A = A op A (if both X and Y are MPI_IN_PLACE)

The latter use-cases may be somewhat artificial (the latter could be relevant for algorithms that employ some form of logarithmic round doubling), but come for free through the semantics allowing MPI_IN_PLACE arguments.

[2]There is an open MPI Forum ticket with this proposal by one of the authors.

To complete the proposal for NEW_Reduce_locals, it must be possible to define three-argument, user-defined operators. Following the MPI standard, a prototype for this could be:

```
typedef void MPI_User_function_three(
  void *in0, void *in1,
  void *inout, int *len, MPI_Datatype *datatype)
```

User-defined functions should implement reductions of the form

A = X op Y

where A is the inout buffer, and X and Y are the two in buffers, either of which may be the same as the inout buffer. Thus, in the user code for the reduction operator, the MPI_IN_PLACE option would not have to be used. The reduction function is written as a standard C assignment with an expression on two arguments.

Here is the butterfly example with the proposed, three-argument local reduction function, which alleviates the need for a typed copy operation.

```
int neighbor = (rank^k); // flip bit k
MPI_Sendrecv(recvbuf,...,neighbor,
             tempbuf,...,neighbor,...,comm);
if (neighbor<rank) {
  NEW_Reduce_locals(tempbuf,MPI_IN_PLACE,recvbuf,
                    ...,op);
} else { // neighbor>rank
  NEW_Reduce_locals(MPI_IN_PLACE,tempbuf,recvbuf,
                    ...,op);
}
```

It is curious and worth mentioning that the specification of MPI_Reduce_local does not say explicitly in which order the two arguments are used. This is explained in the section on user-defined operators [15, Section 6.9.5]. An explicit and precise specification of the order in which the arguments are applied by MPI_Reduce_local would benefit the MPI standard.

We have tried to realistically benchmark the cost of two- and three-argument local reduction operations in comparison to the MPI_Reduce_local operation. Our use-case is the following. An MPI process has data in a sendbuf, has received data into a tempbuf, and needs to combine these into a recvbuf. Each buffer has a count of n MPI_INT elements. The operation to be applied is element wise addition (MPI_SUM). With two-argument reduction functions like MPI_Reduce_local, we first copy sendbuf into recvbuf and then perform the reduction. This is done either with a handwritten, simple loop function, or with the MPI_Reduce_local operation. In the latter case, the copying is done in a type safe manner as will be discussed in Section 6; in the former case, with memcpy. These reductions are contrasted with a handwritten three-argument function directly on the three arguments sendbuf, tempbuf and recvbuf. The benchmark involves no communication.

We have benchmarked as explained in Section 2. We perform $r = 95$ repetitions with five (5) warmups (for this benchmark, the first few iterations were more expensive than the rest) with different element counts $n = 10, 100, \ldots, 1\,000\,000$. Results with OpenMPI 4.1.4 are shown in Table 6. For small counts $n = 10, 100$, the solution with MPI_Reduce_local has a surprisingly high latency, and is a factor of $5 - 8$ slower than either of the handwritten solutions.

Table 6: Local reductions with two- and three-argument local reduction operations with OpenMPI 4.1.4.

			Time/Op (μs)					
			2-args		3-args		MPI	
p	n	r	Avg	Min	Avg	Min	Avg	Min
1	10	95	0.054	0.050	0.043	0.042	0.470	0.336
1	100	95	0.067	0.065	0.059	0.057	0.495	0.353
1	1000	95	0.304	0.293	0.221	0.218	0.627	0.475
1	10000	95	4.058	4.018	2.780	2.683	3.362	3.327
1	100000	95	46.64	45.98	35.24	34.48	45.23	44.67
1	1000000	95	1088	1082	553	548	1049	1039
32	10	95	0.070	0.055	0.051	0.047	0.833	0.371
32	100	95	0.230	0.070	0.066	0.061	0.532	0.370
32	1000	95	0.413	0.319	0.228	0.223	0.890	0.526
32	10000	95	4.182	3.961	2.956	2.668	3.700	3.668
32	100000	95	49.38	48.33	40.38	39.09	47.59	46.78
32	1000000	95	4255	4158	3322	3280	4234	4178

For large counts, the handwritten and `MPI_Reduce_local` two-argument solutions are on par. Here, the three-argument solution is about $20-25\%$ faster, and also has an advantage for the small counts. The differences between $p = 1$ and $p = 32$ are considerable for large counts n due to limited, total memory bandwidth.

6 DATATYPES AND TYPED COPY OPERATIONS

As was discussed, usages of local reductions with the restricted two-argument `MPI_Reduce_local` function as in the butterfly example in Section 5 may easily make process local copying of MPI structured data (user-defined datatypes) necessary. In general, libraries that work on structured data (matrices, tensors, ...) and rely on the MPI derived datatype mechanism for describing such data, are likely to come into a situation where structured data have to be locally copied from one (user or temporary) buffer to another [2].

There is no explicit functionality for type correct local copying between buffers of structured data in the MPI standard. Users either use a `memcpy` operation, which is not type safe (respecting signature and semantics of the data being copied) and only efficient and correct for consecutive data buffers, or employ the standard trick of doing `MPI_Sendrecv` communication from the process to itself in the `MPI_COMM_SELF` communicator.

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,0,0,
             recvbuf,recvcount,recvtype,0,0,
             MPI_COMM_SELF,MPI_STATUS_IGNORE);
```

Although tedious to write, this is an acceptable and correct solution to the problem that provides enough information for the MPI library to do the typed copy efficiently. The designated `MPI_COMM_SELF` communicator indicates to a high-quality MPI library that the `MPI_Sendrecv` operation can be handled specially, by `memcpy` for consecutive datatypes of the same type, and otherwise by efficient packing, unpacking and transpacking [13, 14] routines, and/or other, special treatment.

6.1 Performance of MPI typed memory copy

We check how well current MPI libraries handle typed copy operations with the `MPI_Sendrecv-MPI_COMM_SELF` implementation by a simple benchmark, by which we implicitly express certain expectations on how a process local, type safe copy operation could be expected to perform. Performance expectations on the use of datatypes in communication operations were formalized much more extensively and benchmarked in [4, 8]. The investigation here focuses exclusively on the process-local performance for which there is potential for optimizations that do not apply in general.

Our benchmark locally copies different representations of $m \times n$ matrices of doubles (`MPI_DOUBLE`), where m and n are parameters to the benchmark. Each of p processes (also a parameter to the benchmark) performs the same local copy operations. With derived datatypes, we can specify different traversals of the given $m \times n$ matrix stored in row-wise C order. The type correct local copy operation can thus be used to transform the given matrix from one representation to another [1].

Using an `MPI_DOUBLE` datatype with an element count mn traverses the matrix in row-wise order. The local copy operation with this type as both `sendtype` and `recvtype` could thus be expected to perform comparably to the (not type safe) `memcpy` operation. A datatype describing a column order traversal of the $m \times n$ matrix can easily be set up with a vector datatype (block of 1 element and stride of n elements) and "tiling" by resizing. This is the "col" datatype in the following tables and discussion. Finally, a datatype traversing the matrix as if the first and the last row had been swapped is set up with an indexed-type constructor. This datatype is called "swap" in the following tables and discussion.

With derived datatypes defining different traversals of the $m \times n$ matrix, a local typed copy will amount to a transformation of the matrix, for instance, a transposition (copy from row order to column order) [1], or a swap of two rows (typed copy from row to "swap"). Further, non-trivial, more involved transformations, e.g., swapping both rows and columns in the same go, can be defined and executed automatically by an MPI typed copy mechanism.

The point for our benchmarking of typed copy via `MPI_Sendrecv` on `MPI_COMM_SELF` is that all of the defined datatype (traversals) of the matrix have the same extent (footprint) in memory, namely mn `MPI_DOUBLE` doubles. Therefore, when input and output types (send and receive types) are the same, it would be reasonable to expect the typed copy operation to perform similarly to `memcpy`. We benchmark and check for this expectation.

We benchmark the `memcpy` operation against all combinations of the defined derived datatypes. We notice that in the MPI libraries we tried, a row wise traversal datatype (two nested, contiguous MPI datatypes) performs the same as the mn `MPI_DOUBLE` case, and we therefore omit these results. This is a good sanity check of the handling of derived datatypes in an MPI library implementation.

We now show results for two combinations of m and n corresponding to a small and a large matrix. We try for $p = 1$ and $p = 32$ processes on *Hydra* to see the effect of concurrent copy operations on the same compute node.

Results with the MVAPICH2 2.3.7 library on the *Hydra* machine are given in Table 7. For the larger $200 \times 10\,000$ matrix, the results are good in the sense that the time for `memcpy` and typed copy of

Table 7: Local copy, untyped and typed, MVAPICH2 2.3.7.

Test	p	m	n	r	Time (μs)	
					Avg	Min
memcpy	1	20	10	95	0.058	0.000
double-double	1	20	10	95	0.502	0.477
double-col	1	20	10	95	1.601	1.431
col-double	1	20	10	95	1.631	1.431
col-col	1	20	10	95	2.585	2.384
double-swap	1	20	10	95	0.931	0.715
swap-double	1	20	10	95	0.964	0.715
swap-swap	1	20	10	95	1.403	1.192
col-swap	1	20	10	95	2.043	1.907
swap-col	1	20	10	95	1.995	1.907
memcpy	1	200	10000	95	3072	3005
double-double	1	200	10000	95	3083	3028
double-col	1	200	10000	95	8835	8795
col-double	1	200	10000	95	6541	6480
col-col	1	200	10000	95	21914	19603
double-swap	1	200	10000	95	5527	5485
swap-double	1	200	10000	95	5060	5004
swap-swap	1	200	10000	95	15145	12935
col-swap	1	200	10000	95	17108	17056
swap-col	1	200	10000	95	18880	18832
memcpy	32	20	10	95	0.284	0.238
double-double	32	20	10	95	0.781	0.715
double-col	32	20	10	95	1.697	1.431
col-double	32	20	10	95	1.822	1.669
col-col	32	20	10	95	2.705	2.623
double-swap	32	20	10	95	1.119	0.954
swap-double	32	20	10	95	1.195	0.954
swap-swap	32	20	10	95	1.501	1.431
col-swap	32	20	10	95	2.229	1.907
swap-col	32	20	10	95	2.133	1.907
memcpy	32	200	10000	95	7306	7227
double-double	32	200	10000	95	7307	7245
double-col	32	200	10000	95	19929	19857
col-double	32	200	10000	95	17877	17767
col-col	32	200	10000	95	43317	37297
double-swap	32	200	10000	95	16728	16664
swap-double	32	200	10000	95	16267	16186
swap-swap	32	200	10000	95	38035	33863
col-swap	32	200	10000	95	40233	39402
swap-col	32	200	10000	95	41452	41045

mn `MPI_DOUBLE` are the same. Also average and best seen times over the $r = 95$ repetitions are very close. There is, however, a significant, negative dependency on the number of processes p for which copy operations are done simultaneously with a slowdown of a factor of two to three from the $p = 1$ to the $p = 32$ case. For the non-contiguous accesses described by the two derived datatypes, the typed copy times increase by a large factor. The most expensive traversal is column-wise with the "col" datatype, and the time for copying from "col" to "col" about twice the time for copying `MPI_DOUBLE` from or to "col", and about a factor of 6 from the simple copying of *mn* `MPI_DOUBLE`. Traversal of the matrix where two rows have been swapped is also expensive, but a bit less so than the column wise traversal. For the "col-col" and the "swap-swap" order copy operations (same send and receive type in the typed copy operation), the best to hope for is that the MPI library realizes that only a consecutive block of doubles has to be copied, and therefore internally performs the operations by a `memcpy` equivalent. This is most clearly not being done with the MVAPICH2 2.3.7 library.

For the small 20×10 matrix, the results clearly show that a `memcpy` is not being used internally for the `MPI_COMM_SELF` communicator. The difference between a `memcpy` and the typed copy of $20\times10 = 200$ doubles is about a factor of 3 apart. Also here, the fact that all datatypes have the same total ("true") extent is not exploited by the library. This makes typed copy of small buffers via `MPI_Sendrecv` an expensive operation, relatively speaking.

Since we look for implementation differences between different MPI libraries, we give the results with the OpenMPI 4.1.4 library in Table 8. There are performance differences between the two libraries in absolute terms, with the OpenMPI 4.1.4 being somewhat slower; but the qualitative behavior and the qualitative differences between `memcpy` and typed memory copy remain the same. This is overall disappointing.

In summary, MPI libraries could improve their performance of process-local communication with the special `MPI_COMM_SELF` communicator to provide advanced, type conscientious users a handle for doing type correct local copy operations. This could have a noticeable performance impact for libraries that work with non-consecutive data of smaller size.

6.2 Other datatype issues

Library building for applications that use non-consecutive, structured data described with MPI datatypes can be tedious, since the MPI standard is (still) weak on supporting functionality for datatype introspection and navigation, as have been argued at length in [19, 20] and elsewhere. We here briefly recapitulate some of the criticism and proposals for either additional, structure-conscious libraries built with MPI, or extensions to the MPI standard.

Overall, the list of lacking functionality for MPI derived datatypes that would be convenient is long. The syntactic decoding functionality provided by MPI [15, Section 5.1.13] is tedious, inconvenient, and unsupportive, and can even lose relevant, structural information.

We return to the reduction library example, where the library implementer may have to write a reduction operator to handle, say, blocks of sparse matrices. By the function prototype for such user-defined operators, a pointer to the MPI datatype describing the structure of the user data is passed and it has to be interpreted correctly by the library code. Since the sparsity pattern may be unknown in advance to the library, the datatype would have to be interpreted in order to locate the individual matrix elements. For such usages, MPI provides little or no direct functionality. The following functionality would be convenient for the writer of such advanced libraries.

- Is the datatype a built-in MPI datatype? Since there is no such MPI predicate, the library developer would have to case analyze for all the different built-in datatypes. In the worst case with MPI 4.0 [15, Annex A.1], this means the 32 built-in

Table 8: Local copy, untyped and typed, OpenMPI 4.1.4.

Test	p	m	n	r	Time (μs) Avg	(Min.)
memcpy	1	20	10	95	0.061	0.061
double-double	1	20	10	95	0.342	0.327
double-col	1	20	10	95	1.485	1.454
col-double	1	20	10	95	1.561	1.532
col-col	1	20	10	95	2.701	2.656
double-swap	1	20	10	95	0.534	0.517
swap-double	1	20	10	95	0.533	0.515
swap-swap	1	20	10	95	0.719	0.691
col-swap	1	20	10	95	1.741	1.704
swap-col	1	20	10	95	1.675	1.642
memcpy	1	200	10000	95	3065	3015
double-double	1	200	10000	95	3084	2977
double-col	1	200	10000	95	15357	15247
col-double	1	200	10000	95	13004	12722
col-col	1	200	10000	95	35008	32200
double-swap	1	200	10000	95	5703	5658
swap-double	1	200	10000	95	5645	5598
swap-swap	1	200	10000	95	17964	14186
col-swap	1	200	10000	95	24095	23925
swap-col	1	200	10000	95	28730	28619
memcpy	32	20	10	95	0.084	0.069
double-double	32	20	10	95	0.641	0.358
double-col	32	20	10	95	2.003	1.545
col-double	32	20	10	95	2.108	1.633
col-col	32	20	10	95	3.389	2.828
double-swap	32	20	10	95	1.064	0.601
swap-double	32	20	10	95	1.080	0.620
swap-swap	32	20	10	95	1.211	0.836
col-swap	32	20	10	95	2.919	1.858
swap-col	32	20	10	95	2.271	1.804
memcpy	32	200	10000	95	7320	7285
double-double	32	200	10000	95	7360	7295
double-col	32	200	10000	95	23789	23673
col-double	32	200	10000	95	21949	21778
col-col	32	200	10000	95	51177	45451
double-swap	32	200	10000	95	14529	14456
swap-double	32	200	10000	95	14547	14431
swap-swap	32	200	10000	95	33577	28642
col-swap	32	200	10000	95	41431	41001
swap-col	32	200	10000	95	43298	42877

datatypes for the different C types, and the 6 special two-element types for the `MPI_MAXLOC`, `MPI_MAXLOC` operators, etc..

- Is the datatype contiguous? This is typically decided by comparing size and (true) extent of the datatype; but since extents can be changed, and since types can have overlapping entries (as allowed for datatypes for send buffers), the implementer has to be careful with such tests.
- Is the datatype non-overlapping with no more than one data element for each index, offset or displacement addressed by the datatype?
- Is the datatype homogeneous with only elements of one built-in datatype? What is the base built-in datatype?
- What is the total number of elements in the datatype?
- What is the type of the ith element?
- What is the offset/displacement of the ith element?
- What is the repetition count for the largest consecutive, homogeneous segment starting from element i?

Much functionality of this sort was discussed and implemented in the library in [19, 20]. This library used the old-fashioned MPI profiling interface (which is then "gone"), relied heavily on the attribute caching mechanism, and used the MPI decoding functionality for the first pass through a user-defined datatype to build its own representation of the type that is then cached with the datatype. A particular, unsolvable problem with this approach is that the decoding functionality, as specified in the standard [15, Section 5.1.13], may return new datatypes for the component datatypes of a complex, derived datatype. Thus, information on whether the datatype was created by reuse of the same datatype is lost. It is not possible with the provided MPI functionality to determine whether the user created a particular datatype compactly by reuse of components, or wastefully using new datatypes for all components.

MPI provides no functionality for semantic introspection into datatypes. There is for instance no functionality for checking whether two derived datatypes are equivalent or similar in some appropriate sense (there is a weak definition of datatype equivalence in [15, Section 2.4.3]). In contrast, there is such (limited) functionality for groups and communicators, namely the `MPI_Group_compare` and `MPI_Comm_compare` functions.

A proposal would be to introduce a stronger semantic concept of datatype equivalence, and provide an operation, say `NEW_Type_compare` for datatypes. Outcomes could, as for the group and communicator operations, be that the compared datatypes are indeed identical (same handle), equivalent, by describing the same layouts in the same order (identical type maps), similar (by having the same signature type which means the same order of elements of the same types, but possibly different displacements), or simply different.

The "col" and "swap" datatypes used in experiment in Section 6.1 are for instance similar to a sequence of mn `MPI_DOUBLE`, and since they are also contiguous, it was possible to copy between to "col" types by a `memcpy` operation. The `NEW_Type_compare` functionality could therefore be used to answer some of the questions listed above.

MPI lacks functionality for effectively using pipelining with derived datatypes, namely for selectively addressing a next segment of data described by a derived datatype as the next block to the pipeline. The only type-safe alternative is to pack the entire buffer described by the derived datatype into a consecutive buffer, and pipeline from this buffer (in a type-safe manner, and even this is difficult). This entails an overhead of a full copy of the data buffer. MPI library implementations provide such functionality internally for their efficient, pipelined algorithms, e.g., for collective operations, but such functionality is not exposed through standardized MPI operations. This makes efficient library writing for new, application-specific collective communication operations that may support or use user-defined datatypes very tedious.

7 SUMMARY

Based on concrete experience with parallel library building with MPI, we discussed desired, but missing or awkward MPI functionality, and outlined possible solutions. The main thrust of our discussion is an argument for *orthogonality*: If a feature, say attributes, is defined for some objects, it should be, as far as it is sensible, defined for all objects. Surely, some user or library writer will someday come up with a use-case for attributes for `MPI_Group`, or `MPI_Status` objects. If a semantic property, say non-blockingness, is defined for some operations in a class, it should be, as far as it makes sense, defined for all operations in the class. For writers of truly non-blocking libraries, non-blocking versions of all communicator-creating functions will eventually be needed.

More concretely, we

- discussed why non-blocking versions for all communicator creation functions make sense,
- motivated why attributes for `MPI_Request` objects would be desirable,
- urged for a three-argument, local reduction operation,
- argued for an MPI typed local copy operation, possibly implemented through a special case `MPI_Sendrecv` operation, and
- revisited arguments for more, and more semantically oriented datatype introspection functionality.

The discussions of the three-argument reduction operation and the type navigation and pipelining functionality are arguments for considering to expose more of the functionality internally used in MPI libraries to the application programmer.

We benchmarked the performance of the MPI attribute caching mechanism, the two-argument `MPI_Reduce_local` operation, and a type-safe local copy with common MPI libraries, and found room for improvement, to varying degrees, in all of them.

A valuable proof-of-concept (proof of completeness) and perhaps a valuable addition to the MPI standard itself is to write stand-alone, fully transparent library versions for collective communication, topology functionality, and MPI-IO. This would also provide a performance baseline, in the sense of performance guidelines [12, 21], for professional implementations of these crucial parts of MPI.

REFERENCES

[1] Enes Bajrović and Jesper Larsson Träff. 2011. Using MPI derived datatypes in numerical libraries. In *Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 6960)*. Springer, 29–38.

[2] Pavan Balaji, Darius Buntinas, Satish Balay, Barry F. Smith, Rajeev Thakur, and William Gropp. 2007. Nonuniformly Communicating Noncontiguous Data: A Case Study with PETSc and MPI. In *21th International Parallel and Distributed Processing Symposium (IPDPS)*. 1–10.

[3] Pavan Balaji, Anthony Chan, William Gropp, Rajeev Thakur, and Ewing L. Lusk. 2010. The Importance of Non-Data-Communication Overheads in MPI. *International Journal on High Performance Computing Applications* 24, 1 (2010), 5–15.

[4] Alexandra Carpen-Amarie, Sascha Hunold, and Jesper Larsson Träff. 2017. On Expected and Observed Communication Performance with MPI Derived Datatypes. *Parallel Computing* 69 (2017), 98–117.

[5] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clinton Whaley. 1996. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming* 5, 3 (1996), 173–184.

[6] James Dinan, David Goodell, William Gropp, Rajeev Thakur, and Pavan Balaji. [n. d.]. Efficient Multithreaded Context ID Allocation in MPI. In *Recent Advances in the Message Passing Interface. 19th European MPI Users' Group Meeting (EuroMPI) (Lecture Notes in Computer Science, Vol. 7490)*. 57–66.

[7] Markus Geimer, Marc-André Hermanns, Christian Siebert, Felix Wolf, and Brian J. N. Wylie. 2011. Scaling Performance Tool MPI Communicator Management. In *Recent Advances in the Message Passing Interface. 18th European MPI Users' Group Meeting (EuroMPI) (Lecture Notes in Computer Science, Vol. 6960)*. 178–187.

[8] William D. Gropp, Torsten Hoefler, Rajeev Thakur, and Jesper Larsson Träff. 2011. Performance expectations and guidelines for MPI derived datatypes: a first analysis. In *Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 6960)*. Springer, 150–159.

[9] Torsten Hoefler, Prabhanjan Kambadur, Richard L. Graham, Galen M. Shipman, and Andrew Lumsdaine. 2007. A Case for Standard Non-blocking Collective Operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI User's Group Meeting (Lecture Notes in Computer Science, Vol. 4757)*. Springer, 125–134.

[10] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. 2009. Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *International Conference on Parallel Processing (ICPP)*. 574–581.

[11] Torsten Hoefler and Marc Snir. 2011. Writing Parallel Libraries with MPI – Common Practice, Issues, and Extensions. In *Recent Advances in the Message Passing Interface. 18th European MPI Users' Group Meeting (EuroMPI)*. 345–355.

[12] Sascha Hunold, Alexandra Carpen-Amarie, Felix Donatus Lübbe, and Jesper Larsson Träff. 2016. Automatic Verification of Self-Consistent MPI Performance Guidelines. In *Euro-Par Parallel Processing (Lecture Notes in Computer Science, Vol. 9833)*. 433–446.

[13] Faisal Ghias Mir and Jesper Larsson Träff. 2008. Constructing MPI input-output Datatypes for efficient Transpacking. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 15th European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 5205)*. Springer, 141–150.

[14] Faisal Ghias Mir and Jesper Larsson Träff. 2009. Exploiting efficient Transpacking for One-sided Communication and MPI-IO. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 16th European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 5759)*. Springer, 154–163.

[15] MPI Forum. 2021. *MPI: A Message-Passing Interface Standard. Version 4.0.* www.mpi-forum.org.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 8024–8035.

[17] Jack Poulson, Bryan Marker, Robert van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Software* 39, 2 (2013).

[18] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. On Implementing MPI-IO Portably and with High Performance. In *6th Workshop on I/O in Parallel and Distributed Systems (IOPADS)*. 23–32.

[19] Jesper Larsson Träff. 2016. A Library for Advanced Datatype Programming. In *23rd European MPI Users' Group Meeting (EuroMPI)*. ACM, 98–107.

[20] Jesper Larsson Träff. 2020. Signature datatypes for type correct collective operations, revisited. In *27th European MPI Users' Group Meeting (EuroMPI/USA)*. ACM, 1:1–1:8.

[21] Jesper Larsson Träff, William D. Gropp, and Rajeev Thakur. 2010. Self-consistent MPI Performance Guidelines. *IEEE Transactions on Parallel and Distributed Systems* 21, 5 (2010), 698–709.

[22] Jesper Larsson Träff and Sascha Hunold. 2020. Decomposing MPI Collectives for Exploiting Multi-lane Communication. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 270–280.

[23] Jesper Larsson Träff, Sascha Hunold, Nikolaus Manes Funk, and Ioannis Vardas. 2023. Uniform Algorithms for Reduce-scatter and (most) other Collectives for MPI. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society.

[24] Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, and Daniel J. Holmes. 2021. MPI collective communication through a single set of interfaces: A case for orthogonality. *Parallel Computing* 107 (2021).

[25] Ioannis Vardas, Sascha Hunold, Jordy I. Ajanohoun, and Jesper Larsson Träff. 2022. `mpisee`: MPI Profiling for Communication and Communicator Structure. In *27th International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS), 36th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 520–529.