Zero-copy, Hierarchical Gather is not possible with MPI Datatypes and Collectives[*]

Jesper Larsson Träff
traff@par.tuwien.ac.at

Antoine Rougier
rougier@par.tuwien.ac.at

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

ABSTRACT

It is a desirable feature of MPI that application specific collective operations can be implemented efficiently in terms of other, more primitive operations of MPI. Recent MPI 3.0 functionality makes it possible to build portable libraries that are sensitive to and can exploit the hierarchical structure of, e.g., current shared-memory clusters, and thus in principle to implement efficient, hierarchy-aware collective operations effectively.

In this paper, which complements investigations on all-to-all communication, we show that current MPI 3.0 functionality is insufficient to implement simple, collective gather operations as found in the MPI standard in a hierarchical manner in terms of simpler gather operations, if we pose the extra constraint that no explicit data reorganization apart from what can be expressed with MPI derived datatypes is allowed in any stage of the implementation. Thus, we establish that *zero-copy*, hierarchical implementations of `MPI_Gather` are not possible in current MPI. We discuss ways to alleviate the problems, and sketch a possible solution.

1. INTRODUCTION

In this companion paper to [12] we show by example that it is *not* possible with current MPI functionality to give hierarchy sensitive, zero-copy implementations of the gather, allgather and scatter collective operations of MPI. A *zero-copy implementation* is an implementation with no explicit, process-local data reorganization operations; all data reorganization is performed implicitly by the (collective) communication operations, typically using MPI derived datatypes in the communication operations to specify the placement of the involved data elements. Zero-copy implementations for application specific (FFT and other) collective operations were considered in [1, 5]; zero-copy implementations of a well-known, logarithmic-round all-to-all algorithm [2] were recently presented and benchmarked in [13], which also discusses ways to eliminate or factor out the overheads incurred by the MPI type constructors, commit and type free operations, for instance by viewing specific collective calls as *persistent*.

It is a desirable completeness feature of MPI (or any other message-passing interface for that matter) that the collective MPI operations can be implemented efficiently in terms of other, more primitive operations. Completeness in this sense would make it plausible that application specific (collective) operations can also be implemented with the available MPI functionality. In particular, if *some* cases of a complex, collective functionality can be implemented in a "natural" way under certain likewise "natural" constraints, then *all* cases of this functionality should be implementable in the same, natural way. Collective operations whose specification require a specific data distribution are an example: if some cases of input/output data distributions can be handled by the mechanisms provided by MPI, it would be desirable that all allowed distributions can be handled by the same mechanisms. This would save the implementer from tedious case-by-case implementations, and possibly make implementations more efficient. Explicit data reorganization is an example: if it can in some cases be avoided, for instance by using the MPI derived datatype mechanism to describe where data are to be found and placed, it would be desirable for the implementation if the datatype mechanism would make it possible to avoid explicit data reorganization in all cases. With the MPI derived datatype mechanism, whatever data reorganization would (have to) take place, would effectively be delegated to the MPI library, which would be able to balance and/or hide data movement behind other communication operations.

Current parallel systems like clusters of shared-memory nodes mostly have a hierarchical communication structure, and good collective algorithms should take this into account. For this, MPI must expose, in a portable way, enough of the structure of the underlying system. Recent MPI 3.0 functionality [6] makes it possible to group together processes that belong to the same shared-memory node, which is sufficient for implementing simple hierarchical, collective algorithms (and was the reason this functionality was added to

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

EuroMPI/ASIA '14, September 09-12 2014, Kyoto, Japan
Copyright 2014 ACM 978-1-4503-2875-3/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642769.2642772.

the standard). In [12] we use this functionality to give a hierarchical implementation of the MPI_Alltoall collective, and show that this has the drawback that the non-scalable MPI_Alltoallw operation must be employed. On the other hand, MPI_Alltoallw is general enough that MPI derived datatypes can be used for the necessary, node-local data reorganizations, in that sense leading to a zero-copy, hierarchical implementation of MPI_Alltoall. However, by virtue of MPI_Alltoallw and the necessity of setting up arrays of different, non-tilable datatypes, the solution is potentially less scalable than a non-hierarchical MPI_Alltoall implementation, and entails overheads that compromise performance for small problem instances compared to a non-hierarchical implementation. We note that some of these overheads could be amortized by introducing *persistent collectives* (whether by explicit interfaces, or implicitly in the implementation), as discussed and shown in [13].

For the gather, and by implication, allgather and scatter collective operations, the situation is different, and in a sense even less satisfactory. If we insist that the hierarchical implementation of gather is only allowed to use other gather collectives as components, an implementation that is also zero-copy is not possible. The problem is caused by the limited expressivity of the MPI_Gather and MPI_Gatherv interfaces, see also the discussion in [10]. Although probably not surprising to the expert, we elaborate on this observation, and discuss some extensions to the datatype mechanism that in some cases would help. For the concrete gather problem, it seems that a good solution would require more expressive collective interfaces that goes beyond MPI. Introducing a general, typed gather operation (MPI_Gatherw in analogy with MPI_Alltoallw) could solve the concrete problem, but leads to other issues, and is not our recommendation.

We provide an experimental evaluation of some implementation alternatives. It is easy to implement the hierarchical algorithm in a way that is not zero-copy with a root local reordering step using, e.g., MPI_Pack and MPI_Unpack. In the case of a linear process distribution this step is not necessary. We compare the cost of the two cases of linear vs. randomly permuted communicator in order to estimate the costs incurred by the reordering step. However, with the current MPI 3.0 standard there is no way of comparing to a zero-copy, datatype only implementation: such an implementation is not possible.

2. HIERARCHICAL GATHER

To make our point, it is enough to consider the regular MPI_Gather collective. A standard, hierarchy sensitive implementation of the MPI gather operation is shown in Figure 1. The communicator comm, possibly spanning several shared-memory nodes, is split into communicators spanning the shared-memory nodes, called local, and a communicator with a local root process for each node, called bridge. The implementation performs concurrent, node-local gather operations into intermediate buffers, which are then used as input buffers for a gather operation over the nodes into the recvbuf of the root process.

Communicator splitting into local communicators can trivially be done with the new MPI 3.0 function MPI_Comm_split_type as was discussed (and benchmarked) in [12]. Splitting can be done once and for all, and the resulting local and bridge communicators cached with the comm communicator. Since the hierarchical scheme in Figure 1 is mostly good for small problems, it is actually important that the potentially expensive communicator splitting is separated from the rest of the algorithm. An innocent looking problem is to determine the rank of the node (in the bridge communicator) at which the root (in the original comm communicator) resides. The only, pure MPI solution we have to this problem is to enforce that the node of the root process in comm always receives a fixed rank (e.g., rank 0) in the bridge communicator. We achieve this by giving the local root of the root node a small key and all other local roots a large key in the split of comm into the bridge communicator. Unfortunately, this solution defies the caching idea: a different root rank can be given in each MPI_Gather(...root,comm); call, so the split has to be done in each gather call. Precomputing a mapping for ranks in comm to the node at which they are residing is unattractive for scalability reasons; this map will take linear space in the worst case. Furthermore, this will not solve the problem since we cannot ensure that the root process will always be a process in the bridge communicator, unless bridge is created dynamically at each gather call. To summarize: local roots in the local communicators must be able to determine the rank of the node in the bridge communicator at which the global root resides. Furthermore, for the local communicator containing the global root process, the local root should be the local rank of the global root in this communicator. The new MPI 3.0 functionality does not provide direct support for the local shared-memory communicators to determine the processes in other local communicators. Ensuring that the global root becomes local root in its corresponding local communicator is also only possible by a communicator splitting operation. It thus seems impossible to precompute shared-memory communicators for use in the rooted collective operations; at least the split into bridge communicator seems unavoidable per gather operation. Splits already performed could of course be cached with the communicator, making the rooted calls at least partly persistent. However, there are a large number of possible splits.

```
int MPI_Gather(sendbuf,...,recvbuf,...,root,comm) {
  MPI_Comm_rank(comm,&rank);
  // split comm into local and bridge communicators
  // for this and next level
  MPI_Comm_rank(local,&localrank);
  // choose root in this level communicator
  if (rank==root)
    localroot = localrank; else localroot = 0;
  // allocate local send buffer
  if (localrank==localroot)
    sendlocal = malloc(...);
  MPI_Gather(sendbuf,...,sendlocal,...,localroot,local);
  // Problem: choose bridgeroot in bridge communicator
  // bridgeroot is the rank of the node of the root process
  if (localrank==localroot)
    MPI_Gather(sendlocal,...,recvbuf,...,bridgeroot,next);
}
```

Figure 1: Paradigm hierarchical MPI_Gather implementation, useful for small problem sizes.

We use p for the total number of MPI processes in comm, N for the number of nodes (size of bridge communicator), and $n = p/N$ for the average number of processes per node (size of local communicator). We say that the communicator comm is *regular* if all local, shared-memory node communicators have the same size of n processes. The size of each data element to another process is m (for regular problems), and the *total problem size* is pm. Data elements can be arbitrarily structured as determined by the MPI type and count arguments. To make our point, it is enough to consider regular communicators.

The implementation in Figure 1 can be attractive for small problem sizes. In this case the `MPI_Gather` at each level might be implemented by a communication-round optimal, tree-based algorithm. For the number of communication rounds for the hierarchical algorithm, it holds that $\lceil \log N \rceil + \lceil \log n \rceil \leq \lceil \log p \rceil + 1$, so in number of rounds, the hierarchical algorithm is at most one round off from the round lower bound (see, e.g., [3] for an overview). Other than this, we make no claim that this hierarchical algorithm is best possible.

3. THE ELEMENT-ORDERING PROBLEM

We insist on no further operations on the gathered data other than those entailed by the two gather operations, in particular no explicit node-local data reorganizations. Intermediate and final data placements must be described by the datatype arguments to the two `MPI_Gather` operations.

By the generality of the communicator creation operations of MPI, even for a regular communicator the MPI processes can be arbitrarily distributed over the shared-memory nodes. Let the MPI process ranks of the given `comm` communicator be (semi-)sorted in node-order, and let $\pi(i)$ denote the rank of the ith process in this order. This is illustrated in Figure 2: each node hosts n processes of the `comm` communicator, and the nodes are ranked in some arbitrary order as determined by the split operation for the `bridge` communicator. The gather operation over the `bridge` communicator will collect the blocks of elements from the nodes in consecutive order determined by the node ordering. Such is the semantics of `MPI_Gather(...,bridge)` and only the ordering of the blocks (including possible, regular interblock tilings) can be influenced by the receive datatype argument. However, the semantics of `MPI_Gather(...,root,comm)` requires that the ith element be in position $\pi(i)$ times the extent of the receive datatype.

Now, clearly, an arbitrary permutation π cannot be expressed as a an intrablock injective mapping ρ of n-element blocks onto $\{0, \ldots p-1\}$, plus a block dependent offset o, that is $\pi(i) = \rho(i \bmod n) + o\lfloor i/n \rfloor$ is not possible for all permutations π. Therefore, with only the same datatype per block, it is not possible to achieve the required, global rank order at the root process. An explicit reordering of the received data as in Figure 2 is needed in the general case. Resorting to the irregular `MPI_Gatherv` operation does not help. With this collective, the placement of the blocks can be controlled, but the blocks still all have the same structure determined by the single receive type argument.

Thus, to implement hierarchical, zero-copy gather, allgather, or scatter operations more expressive MPI interfaces are needed which allow a specific datatype for each of the N blocks. The naive solution would be to introduce the corresponding `MPI_Gatherw`, `MPI_Allgatherw` and `MPI_-Scatterw` collectives. Apart from bloating the MPI standard, this has a very serious drawback. Namely, that since these operations would be able to express irregular communication problems (different amount of data between different processes) they would be implemented by corresponding irregular algorithms (or require a preprocessing to determine whether the concrete instance is regular or irregular); and not many good such algorithms are known. Irregular interfaces for `MPI_Gatherw` etc., if following the MPI specification, would not allow process local, consistent decisions on which (regular or irregular) algorithm to use, so implementations would in general have to be prepared for the worst case. In our concrete case, regular gather for regular communicators, for small problems, it is clear that regular, e.g., tree-based, algorithms should be used. More expressive collective interface would have to make it possible to convey this information to the library in some form. These problems were previously discussed in [10], with other examples. A different, more elegant and much less disruptive solution is outlined in Section 6.

We finally remark that using the `MPI_Alltoallw` operation to express the required, but missing `MPI_Gatherw` functionality is out of the question both for scalability reasons, and for the algorithmic reasons given above.

4. AN EXPERIMENTAL ANALYSIS

We have implemented a hierarchical gather algorithm following the paradigm of Figure 1. We assume that the root in `comm` is rank 0, so that the communicator splitting can be done separately. This solution works for regular communicators where the processes are consecutively assigned to the shared-memory nodes. For arbitrary, non-consecutive (but regular) communicators, the gather operation over the `bridge` communicator most gather into a contiguous, intermediate buffer, from which it should be possible to let an `MPI_Unpack` operation with an indexed-block type corresponding to the mapping π perform the reordering into the `recvbuf`. We compare the implementations to estimate how expensive the extra, explicit reordering operation may be. Results are shown in Figure 3.

The experiment is done on a small InfiniBand cluster with $N = 36$ nodes and $n = 16$ cores per node. The nodes consist of two 8-core 2.3GHz AMD 6134 Opteron processors/node, and are interconnected with a Mellanox MT4036 QDR InfiniBand switch. The total number of cores is $p = 576$.

While using `MPI_Unpack` seems the right way to do the permutation of the data elements from intermediate to `recvbuf`, this solution is (probably) not strictly correct. The unpack operation operates on a socalled packing unit of type `MPI_PACKED`. Thus, the `bridge` gather operation should gather into a buffer of this type and we therefore use `MPI_PACKED` as the receive type. The semantics of gather states that the gathered elements are stored contiguously in the receive buffer. On the other hand, the MPI 3.0 standard says [6, Chapter 4, p. 134] that “The concatenation of two packing units is not necessarily a packing unit”. It is therefore not clear that unpacking the whole receive buffer as single unit (as is necessary to accomplish the desired permutation) is correct. It is not clear whether the MPI standard actually intends packing units to be used with collective operations. We have therefore implemented an alternative solution where we receive into an intermediate buffer of `MPI_BYTE` type (also not a good solution, since type information is lost) and do a process local send-receive operation with the indexed-block type as receive type. Both solutions are benchmarked in Figure 3.

The vendor MPI has a problem with unpacking short and medium sized data up to about 1,2MByte of indexed-

$$\underbrace{m_{\pi(0)}m_{\pi(1)}\ldots m_{\pi(n-1)}}_{\text{Node }0}\underbrace{m_{\pi(n)}m_{\pi(n+1)}\ldots m_{\pi(2n-1)}}_{\text{Node }1}\cdots\underbrace{m_{\pi((N-1)n)}m_{\pi((N-1)n+1)}\ldots m_{\pi((N-1)n+n-1)}}_{\text{Node }N-1}$$

$$\Downarrow$$

$$m_0m_1m_2\ldots m_{p-2}m_{p-1}$$

Figure 2: The order in which the p gathered elements become available to the root process, with $\pi(i)$ denoting the global rank of the ith element. Each node contributes one block of elements, in node order. Since the communicator is regular, the node blocks have the same number of n elements.

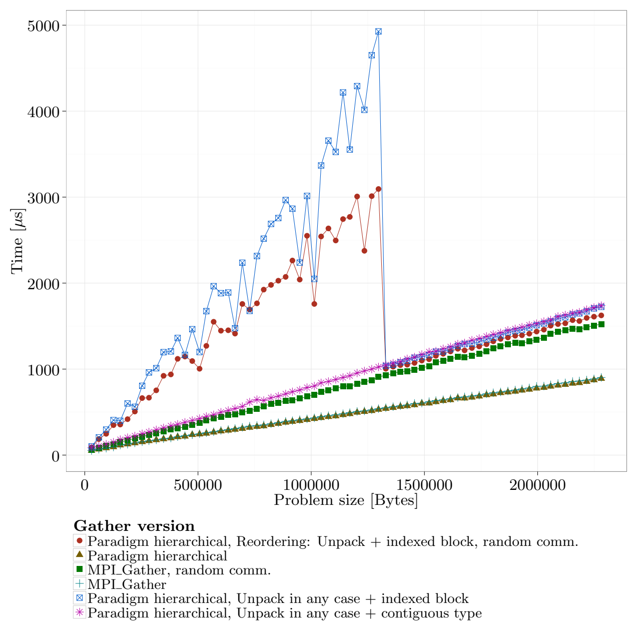

Figure 3: Hierarchical gather with ordered and random, regular communicator. For the random communicator the required, explicit unpack operation is implemented with either `MPI_Unpack` or with `MPI_Sendrecv`. For comparison, implementations with explicit unpack for the ordered case, both from contiguous and indexed-block type are also shown.

block type, as seen both for the random communicator, and for the ordered `MPI_COMM_WORLD` with forced unpack from an indexed-block type. The penalty of having to unpack from an intermediate buffer can be estimated by comparing `MPI_Gather` for the regular communicator with the hierarchical implementation with intermediate buffer and forced unpack. For this case, the hierarchical implementations perform similarly to `MPI_Gather` with a randomly permuted communicator. For ordered communicators, our hierarchical implementation is on par with vendor `MPI_Gather`, which is also the case for random communicators beyond the anomaly with `MPI_Unpack`.

5. NEW DATATYPES FOR MPI?

The ability of MPI to describe the location and order of access to the data that are used in communication operations by derived datatypes adds considerable power to the interface, as we (and many, many others) have shown here and in accompanying papers [1, 12, 13]. With the MPI derived

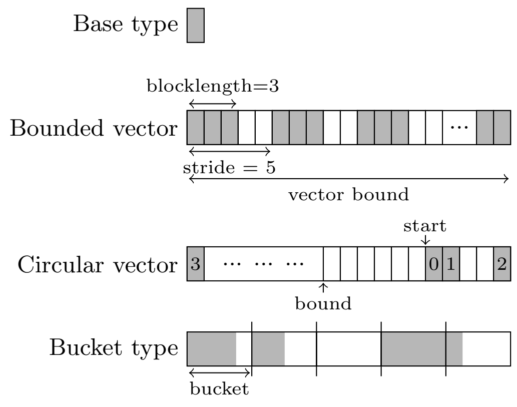

Figure 4: Type maps of bounded and circular vector datatypes, and the bucket type for given base type.

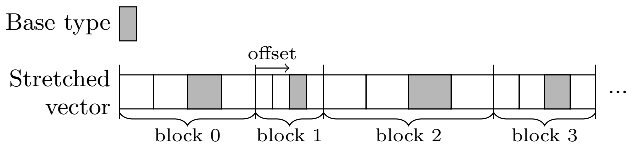

Figure 5: The type map of the stretched vector for $p \geq 4$ blocks. The constructor takes a blocksize $b[i]$ for each block and stretches the offsets for each block as indicated.

datatype functionality, any static layout can be described in an often very concise, space- and time-efficient manner. However, as we have identified, there are cases that cannot readily be expressed, or only expressed at an unnecessary cost. In this section we summarize some findings. We illustrate suggested, new datatype constructors by showing how new layouts are constructed from repetitions of given base types, as done in [8, 9].

Bounded and circular vectors (Figure 4) are layouts that slightly generalize the MPI vectors. They are natural and useful in the implementation of the all-to-all algorithm of Bruck et al. [2, 13]. They can quite easily be expressed in terms of existing MPI derived datatypes, with only a bit of case analysis to select the proper constructors (contiguous, vector, and struct). Especially the derived, bounded vector constructor showed also to be efficient enough to be used in individual communication operations. However, tailored datatype engine functionality could give even better performance and lower overhead. There is, however, no way in MPI to extend the datatype engine, and therefore derived, derived datatypes come at some cost. For the efficient processing of new data layouts, library-internal type normalization [4] could lead to efficient, internal representations, but entails other overheads [11] at type creation (commit) time.

The bucket type was also discussed in [13], and is now shown in Figure 4. It was useful in the implementation of the logarithmic round all-to-all algorithm for not-quite-regular problems. The bucket type is the natural complement to the indexed-block type that was introduced with MPI 2.1. For a fixed bucket size, it takes an array of block sizes giving the number of actual elements going into each of the buckets. The bucket type can of course easily be implemented with an indexed constructor, but that defies an important point. The index constructor requires (and MPI almost enforces storage of?) two arrays, whereas for the bucket type the array of displacements is redundant: each bucket starts a some multiple of the fixed bucket size. The bucket constructor is a more space-efficient representation of this kind of layout, which we think is common to some (bucket) algorithms.

In [12] a new problem came up. To solve the all-to-all problem in a hierarchical fashion for irregular communicators it was necessary to use `MPI_Alltoallw` for the all-to-all exchange over the nodes. A useful datatype for this has the type map illustrated in Figure 5. What was needed was a way of specifying and representing in a space efficient way p indexed types in one operation, where each type consists of n blocks of size $b[j], 0 \leq j < n$, and the jth block of type $i, 0 \leq i < p$ starts at displacement $ib[j] + \sum_{k=0}^{j-1} pb[k]$.

This socalled *stretched vector constructor* takes as input an array of p elements of block sizes. The ith type derived from this array is like an indexed type with the jth block starting at displacement $ib[j] + \sum_{k=0}^{j-1} pb[k]$. While each of the p stretched types can easily be expressed as an MPI indexed type, this explicit representation would take p arrays of size n, instead of the implicit representation as a stretched vector which takes only one p element block size array. The stretched vector would give the opportunity to create all p types required in one operation and with only one input array. However, this implicit representation of p type maps would require more significant changes to the MPI library internal datatype engine, and therefore comes with a nontrivial effort. We would like to investigate whether efficient type-processing implementations can indeed be given for this new kind of datatype constructors.

The most important suggested addition to the datatype functionality, is to associate a *signature type* [13] with each (derived) MPI type. This represents the sequence of basic datatypes in the type in the order determined by the constructors as a contiguous layout such that the size of this type would equal its extent. Signature types are suggested as a type-safe, more convenient way of dealing with contiguous, typed, intermediate buffers, and an alternative to typing with `MPI_PACKED`.

6. A SOLUTION: RE-INTERPRETING THE COLLECTIVE INTERFACES

A slight re-interpretation of the collective interfaces for the regular (and irregular) collective operations could actually solve the problems discussed, in a much more elegant and less intrusive manner than proposed in [10]. The problems were all caused by the MPI specification stating that the `recvcount` and `recvtype` arguments describe *only one element* of the all in all p elements to be gathered by the root process. This prevented us from using a global indexed-

```
int MPI_Gather(sendbuf,sendelement,sendtype,
               recvbuf,recvelement,recvtype,root,
               comm)
{
  MPI_Comm_rank(comm,&rank);
  // split comm into local and bridge communicators
  // for this and next level
  MPI_Comm_rank(local,&localrank);
  // choose root in this level communicator
  if (rank==root)
    localroot = localrank; else localroot = 0;
  // allocate local send buffer
  if (localrank==localroot) {
    sendlocal = malloc(...);
    Get_signature(sendtype,&sendsigtype);
  }
  MPI_Gather(sendbuf,sendelement,sendtype,
             sendlocal,sendelement,sendsigtype,
             localroot,local);
  // Problem: choose bridgeroot in bridge communicator
  // bridgeroot is the rank of the node of the root process
  if (localrank==localroot) {
    MPI_Comm_size(comm,&size);
    MPI_Type_create_indexed_block(size,1,
                                  nodesortedranks,recvtype,
                                  &roottype);
    MPI_Type_commit(&roottype);
    MPI_Comm_size(local,&localsize);
    MPI_Gather(sendlocal,localsize*sendelement,sendsigtype,
               recvbuf,localsize*recvelement,roottype,
               bridgeroot,bridge);
    MPI_Type_free(&roottype);
}
```

Figure 6: A hierarchical `MPI_Gather` implementation under re-interpreted collective interface semantics. The `Get_signature` function returns the signature type of the `sendtype` which is needed for the node local, intermediate, contiguous buffer.

block datatype to express the arbitrary permutation needed by the root to put the elements of the gathered blocks into rank order. We suggest to separate the number of elements to be gathered from the description of the structure of the elements. The gather interface would stay the same:

```
MPI_Gather(sendbuf,sendelement,sendtype,
           recvbuf,recvelement,recvtype,root,comm)
```

The re-interpreted semantics of this gather operation is that each process contributes `sendelement` (non-negative integer) basic elements, taken from the layout described by a large enough, contiguous sequence of `sendtype` datatypes. For gather usage where some number of elements of a basic type like, e.g., `MPI_DOUBLE` is to be sent, the user would see no change. If the `sendtype` is a complex, derived datatype, `sendelement` elements are taken in order from this layout. A standard MPI application with such datatypes would have to change the count argument by multiplying with the number of basic elements in the `sendtype`. Functionality to determine the number of basic elements in any MPI (derived) datatype would be convenient. For the receive buffer, `recvelement` elements are gathered from each process and put in the `recvbuf` with structure likewise determined by a large enough, contiguous sequence of the `recvtype` layout. The elements from rank i are put into the positions of the elements $i\texttt{recvelements}, i\texttt{recvelements} + 1, \ldots, (i +$

1)`recvelements`-1 of the contiguous repetition of the `recvtype` layout. As in current MPI, the semantic requirement is that `recvelement` equals `sendelement`, so one argument is actually redundant.

A fully worked out, hierarchical gather algorithm under this re-interpretation is shown in Figure 6. The indexed-block datatype needed for the permutation is given as receive type in the gather call on the `bridge` communicator, and the received blocks of `localsize*sendelement` elements per shared-memory node are put into the corresponding positions determined by the indexed-block type. The indexed-block type has p elements, and is thus by itself sufficiently large.

All regular MPI collectives can be reinterpreted in this fashion. We stress that applications using only counts of basic datatypes would actually see no change. Applications using composite types would have to multiply their count arguments with the number of basic types in the complex type, which can easily be provided for. Importantly, the new interpretation of the collective operations (for consistency, all communication operations should eventually be changed to operate on number of elements instead of on counts of a datatype!) would provide a whole new scope for advanced, datatype assisted programming, and would, we contend, make all issues raised in [1, 10, 12] elegantly solvable.

The proposal carries over to the irregular collectives as well, and would obviate the unpleasant `MPI_Alltoallw` collective altogether. As an example, an irregular gather operation would have the following specification:

```
MPI_Gatherv(sendbuf,sendelement,sendtype,
            recvbuf,recvelements[],recvtype,root,
            comm)
```

The `recvelements` vector, significant at the root process only, contains the number of elements to be gathered from each process in the call with the semantic requirement that `sendelement` is equal to `recvelements[rank]`. Structure is solely determined by a large enough, contiguous sequence of the `recvtype` layout; all indices are implicit in this type, thus, there a separate displacements array is not needed. MPI applications using the displacements in a non-trivial fashion would have to create an indexed type to be used as `recvtype` with blocklengths and displacements as in the `MPI_Gatherv` call. Irregular operations where processes contribute different number of elements, but where the result is stored in a contiguous buffer would be much simpler, though. No new datatype is needed, and the displacements array is effectively saved.

7. CONCLUDING REMARKS

By way of example, we showed that zero-copy implementations of certain kinds of collective operations are not possible with the current MPI specification. This is due mainly to lack of expressivity in the collective operation interfaces. We discussed ways of alleviating those problems, and use the examples analyzed here as starting points towards more complete and expressive collective operation interfaces.

A prospect that we have not discussed is for an MPI aware compiler to perform non-trivial optimizations across successive MPI operations. We think this could be greatly aided by the (semi-) static datatype arguments in the two gather-calls in Figure 1, see also [7].

8. REFERENCES

[1] E. Bajrović and J. L. Träff. Using MPI derived datatypes in numerical libraries. In *Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting*, volume 6960 of *Lecture Notes in Computer Science*, pages 29–38. Springer, 2011.

[2] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[3] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[4] W. D. Gropp, T. Hoefler, R. Thakur, and J. L. Träff. Performance expectations and guidelines for MPI derived datatypes: a first analysis. In *Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting*, volume 6960 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2011.

[5] T. Hoefler and S. Gottlieb. Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. In *Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting*, volume 6305 of *Lecture Notes in Computer Science*, pages 132–141. Springer, 2010.

[6] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.0*, September 21st 2012. www.mpi-forum.org.

[7] T. Schneider, F. Kjolstad, and T. Hoefler. MPI datatype processing using runtime compilation. In *Recent Advances in the Message Passing Interface, 20th European MPI Users's Group Meeting (EuroMPI)*, pages 19–24, 2013.

[8] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.

[9] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[10] J. L. Träff. Alternative, uniformly expressive and more scalable interfaces for collective communication in MPI. *Parallel Computing*, 38(1–2):26–36, 2012.

[11] J. L. Träff. Optimal MPI datatype normalization for vector and index-block types. In *Recent Advances in Message Passing Interface. 21st European MPI Users' Group Meeting*, 2014.

[12] J. L. Träff and A. Rougier. MPI collectives and datatypes for hierarchical all-to-all communication. In *Recent Advances in Message Passing Interface. 21st European MPI Users' Group Meeting*, 2014.

[13] J. L. Träff, A. Rougier, and S. Hunold. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *28th ACM International Conference on Supercomputing (ICS)*, pages 135–144, 2014.