Implementing a Classic: Zero-copy All-to-all Communication with MPI Datatypes[*]

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

Antoine Rougier
rougier@par.tuwien.ac.at

Sascha Hunold
hunold@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

We investigate the use of the derived datatype mechanism of MPI (the Message-Passing Interface) in the implementation of the classic all-to-all communication algorithm of Bruck et al. (1997). Through a series of improvements to the canonical implementation of the algorithm we gradually eliminate initial and final processor-local data reorganizations, culminating in a *zero-copy* version that contains no explicit, process-local data movement or copy operations: all necessary data movements are implied by MPI derived datatypes, and carried out as part of the communication operations. We furthermore show how the improved algorithm can be used to solve irregular all-to-all communication problems (that are not too irregular). The Bruck algorithm serves as a vehicle to demonstrate descriptive and performance advantages with MPI datatypes in the implementation of complex algorithms, and discuss shortcomings and inconveniences in the current MPI datatype mechanism. In particular, we use and implement three new derived datatypes (bounded vector, circular vector, and bucket) not in MPI that might be useful in other contexts. We also discuss the role of persistent collectives which are currently not found in MPI for amortizing type creation (and other) overheads, and implement a persistent variant of the `MPI_Alltoall` collective.

On two small systems we experimentally compare the algorithmic improvements to the Bruck et al. algorithm when implemented on top of MPI, showing the zero-copy version to perform significantly better than the initial, straightforward implementation. One of our variants has also been implemented inside `mvapich`, and we show it to perform better than the `mvapich` implementation of the Bruck et al. algorithm for the range of processes and problem sizes where it is enabled. The persistent version of `MPI_Alltoall` has no overhead and outperforms all other variants, and in particular improves upon the standard implementation by 50% to 15% across the full range of problem sizes considered.

[*]This work was co-funded by the European Commission through the EPiGRAM project (grant agreement no. 610598). This work was supported by Austrian FWF projects "Verifying self-consistent MPI performance guidelines" and "Improving reproducibility of experiments in parallel computing".

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent programming—*Parallel programming*; C.4 [**Performance of Systems**]: Measurement techniques; F.2.2 [**Analysis of algorithms**]: Nonnumerical algorithms and problems—*Routing*

Keywords

All-to-all collective communication; MPI; derived datatypes

1. INTRODUCTION

A now classical algorithm for regular all-to-all collective communication on fully connected, homogeneous communication networks with a trade-off between number of communication rounds (latency) and communicated data volume (bandwidth) was described in an influential paper on collective communication by Bruck et al. [1]. This algorithm is well-known in the MPI and collective communication communities, and since long implemented in several MPI libraries, e.g., `mpich`, OpenMPI, and vendor libraries [7, 9], where it is used for certain ranges of MPI processes and (smaller) problem sizes.

The communication pattern of the algorithm by Bruck et al. is inherently non-contiguous. The data elements that are sent in one communication round have been received in previous, but non-consecutive rounds, and it is therefore not possible to organize send and receive operations such that elements to be sent always form a contiguous sequence. When the algorithm is implemented, elements will either have to be communicated directly from/to non-contiguous segments of memory, or must be reorganized (packed) locally into contiguous communication buffers.

MPI, the Message-Passing Interface [6], makes it possible to delegate the handling of such non-contiguous data to the MPI library implementation. MPI's derived datatype mechanism [6, Chapter 4] facilitates description of arbitrary, non-contiguous data layouts as derived datatypes to be used subsequently in communication operations as handles to the data. The algorithm by Bruck et al. (originally implemented with explicit, hand-written packing and unpacking) is an exemplary candidate for the use of MPI derived datatypes. The advantage is a cleaner implementation that separates the algorithmic idea from data reorganization issues that are otherwise handled by customized (manual) packing and unpacking code. Depending on how well the MPI library implements the datatype mechanism and interacts with the communication system, a better performing implementation may be the added benefit (as we shall show).

In this paper we present several such implementations. The small theoretical improvement is the elimination of all process-local reordering steps of the original Bruck et al. algorithm [1], leading to a socalled *zero-copy implementation* in which there are no explicit local copy operations between any communication or intermediate buffers. Data for each communication round are described solely by MPI derived datatypes, and whether data reorganizations (copy-/pack/unpack) are necessary is fully an implementation issue of the MPI library and underlying communication system. The wider significance is the illustration of benefits by using derived datatypes in the implementation of complex algorithms, further in the analysis of shortcomings in the MPI derived datatype specification that lead to descriptive and performance obstacles. We discuss possible extensions that may be of value for a datatype-oriented programming style and thus could be considered in future developments of MPI or other message-passing interfaces. Some have been implemented here, so that their convenience and potential performance impact can be concretely discussed.

In particular, we have implemented the algorithm by Bruck et al. as originally presented [1] using a suitable (new) MPI datatype to implicitly pack and unpack the non-contiguous elements to be sent and received in each communication round (Basic Bruck), an improved algorithm that eliminates the final, post-communication, process local permutation using another (new) derived datatype (Modified Bruck), and finally a version that performs no explicit packing, unpacking or other process local reordering of data (Zero-copy Bruck). The latter variant uses structured, derived datatypes to select the non-consecutive data elements for each communication round from send-, receive- and intermediate buffers, respectively. These variations/improvements to the Bruck et al. algorithm have first been implemented on top of MPI, which allows a fair, differential assessment of the improvements over the original algorithm. The evaluation includes all overheads incurred by creation and destruction of the required MPI derived datatypes. We also measure this overhead in isolation by disabling the actual communication; especially for Zero-copy Bruck overheads are considerable and compromises the implementation for small problems. We also benchmark an implementation in `mvapich` of our modified variant and compare it to the `mvapich` implementation of the standard Bruck et al. algorithm (which uses an MPI indexed datatype). The zero-copy variant has also been implemented as a *persistent collective*, which binds all input parameters in a separate setup operation and thus allows to amortize the type creation and other (algorithm selection) overheads over a number of all-to-all communication operations. Persistent collectives are currently not part of MPI.

It is worth recalling that the Bruck et al. algorithm is designed on the assumption of a homogeneous, fully connected communication network, and that each processor communicates with (only) a logarithmic number of neighbors. Also, since each data element is forwarded a logarithmic number of times, the algorithm is competitive only for smaller problem sizes, which decrease slowly with the number of processes. The Bruck et al. algorithm is therefore only one among many all-to-all algorithms, and its range of concrete applicability depends on many factors. The actual performance benefits by using derived datatypes depends on the quality of the MPI library, and the protocol regimes used (small vs. eager vs. rendezvous; fixed buffers; pipelining). Other improvements (elimination of the post-communication permutation), however, are genuinely MPI independent.

2. THE BASIC ALGORITHM AND A FIRST IMPROVEMENT

We first recapitulate the all-to-all algorithm by Bruck et al. [1] which we for now term *Basic Bruck*. Let p be the number of MPI processes, each bound to a processor or core. The processes are numbered (*ranked*) from 0 to $p-1$. Each process has an individual data *element* to each other process, including an element to itself; elements are the units of communication and can represent larger data. The element from process i, $0 \leq i < p$, to process j, $0 \leq j < p$, is denoted $m_{i \to j}$. We consider first the *regular* all-to-all problem in which all elements have the same size; we denote this element size by n so that every process has to send and receive data of size $(p-1)n$ and possibly copy an element of size n locally.

Basic Bruck has three steps, the second of which involves communication. Each process has a p-element vector R where the elements received so far are stored, and from which the elements to send in the next communication round are also taken. Upon termination, $R[j]$ for process i shall store the element $m_{j \to i}$ from process j to process i. The p processes carry out the same operations with process i, $0 \leq i < p$, doing the following:

1. **Local** shift towards index 0 by i indices: set $R[j] = m_{i \to (i+j) \bmod p}$ for $j = 0, \ldots, p-1$.

2. **Global** communication step with $\lceil \log_2 p \rceil$ rounds. In round $k, 0 \leq k < \lceil \log_2 p \rceil$, all elements $R[j]$ where the kth bit of j is equal to one are sent to process $(i + 2^k) \bmod p$, which receives this element into $R[j]$.

3. **Local** reverse and shift towards index $p-1$ by $i+1$ indices: element $R[j]$ is moved to $R[(p-1-j+(i+1)) \bmod p] = R[(p-j+i) \bmod p]$ for each $j, 0 \leq j < p$.

The second step takes $\lceil \log_2 p \rceil$ communication rounds. In each round, up to $\lfloor p/2 \rfloor$ elements are sent and received, and must be handled as one contiguous message. Over all $\lceil \log_2 p \rceil$ rounds, $\lceil \log_2 p \rceil \lfloor p/2 \rfloor$ elements are sent and received per process. A straightforward all-to-all algorithm that sends each element directly to its destination process sends and receives exactly $p-1$ elements per process, but takes $p-1$ communication rounds to do this. In both cases, the trade-off between number of communication rounds and total volume of data communicated is best possible as shown in [1]. Correctness can be argued as follows. The first step puts the elements into R such that the element to be sent from process i to process $((i+j) \bmod p)$ is in $R[j]$. The

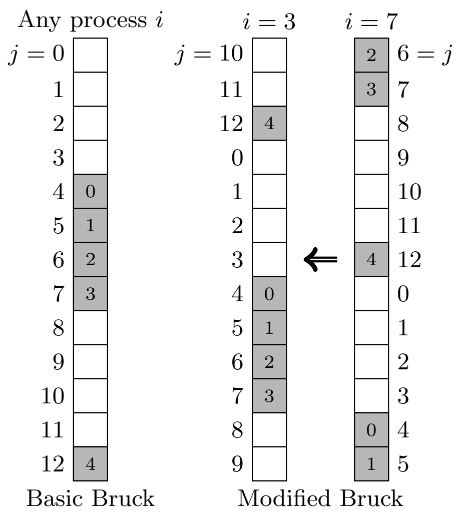

Figure 1: Elements sent in round $k = 2$ with $p = 13$ by Basic Bruck (left), and Modified Bruck for processes $i = 3$ and $i = 7$ (right). Process 7 sends its shaded elements in the indicated order to the shaded positions of process 3.

second communication step sends element $R[j]$ to its destination process $((i + j) \bmod p)$ by decomposing j into its unique sum of powers of two, $j = 2^{j_0} + 2^{j_1} + 2^{j_2} + \ldots$ with $j_0 < j_1 < j_2 < \ldots$ corresponding to the one-bits of j, and sending it via processes $(i+2^{j_0}) \bmod p$, $(i+2^{j_0}+2^{j_1}) \bmod p$, $(i+2^{j_0}+2^{j_1}+2^{j_2}) \bmod p, \ldots$. Thus, before Step 3, for each process i, $R[j] = m_{(i-j) \bmod p \to i}$. Step 3 accomplishes the permutation resulting in $R[j] = m_{j \to i}$ as desired.

Sending and receiving elements $R[j]$ in Step 2 explicitly or implicitly requires an intermediate buffer in addition to R. The elements for round k can be received into this buffer as a contiguous sequence, and copied into their right, non-contiguous positions in R after the receive operation has completed. When elements are sent out of R, an (implicit) pack operation is needed. Basic Bruck therefore entails a p-element copy operation for Step 1, an unpack and pack operation for each communication round in Step 2, and two full copy operations of $p - 1$ blocks via the intermediate buffer for Step 3. In summary:

THEOREM 1. *Basic Bruck solves the all-to-all problem for p processes and elements of size n in $\lceil \log_2 p \rceil$ communication rounds, with at most $\lceil \log_2 p \rceil \lfloor p/2 \rfloor n$ units of data sent and received per process, and at most $p+2\lceil \log_2 p \rceil \lfloor p/2 \rfloor + 2p$ local element copy/unpack/pack operations.*

Assuming a simple, linear cost communication model in which sending and receiving an element of size n takes $\alpha + \beta n$ time units, it follows that Basic Bruck et al. (and the following variants) can be better than a $p-1$-round, direct all-to-all algorithm when n is $O(\frac{\alpha}{\beta} 2/\log_2 p)$.

The third, costly process-local reordering step can be eliminated by maintaining the elements in R in a different order. The *Modified Bruck* algorithm has the following two steps in which each process i, $0 \leq i < p$, does the following:

1. **Local** reverse and shift towards index i: $R[(i+j) \bmod p] = m_{i \to (i-j) \bmod p}$ for $j = 0, \ldots, p-1$.

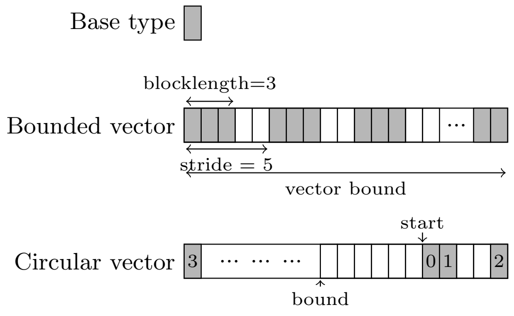

Figure 2: Type maps of bounded and circular vector datatypes for given base type. The bounded vector fits to Basic Bruck, circular vector to Modified Bruck.

2. **Global** communication step with $\lceil \log_2 p \rceil$ rounds. In round $k, 0 \leq k < \lceil \log_2 p \rceil$, all elements $R[(i+j) \bmod p]$ where the kth bit of j is equal to one are sent to process $((i - 2^k) \bmod p)$ which receives these elements into $R[(i - 2^k) \bmod p + j]$.

For both variations of the Bruck et al. algorithm it holds that elements $R[j]$, respectively $R[i + j]$, with the kth bit of j equal to one fall into consecutive sequences of 2^k elements. In Modified Bruck send and receive operations are in the opposite direction of Basic Bruck which ensures that the elements to send in each round are in increasing index order (modulo p). Correctness should be clear: process i initially stores the element $m_{i \to (i-j) \bmod p}$ in $R[(i + j) \bmod p]$, and elements are sent to their destinations following the power-of-two decomposition of j. Note that an element sent from index j by processor i is received at a lower index (modulo p) by the receiving processor; the cyclic shifting of the elements as done explicitly in the first and third step of Basic Bruck is now accomplished as part of the communication. Figure 1 illustrates which elements are communicated in communication round $k = 2$ for the two algorithms.

THEOREM 2. *Modified Bruck solves the all-to-all problem for p processes and elements of size n in $\lceil \log_2 p \rceil$ communication rounds, with at most $\lceil \log_2 p \rceil \lfloor p/2 \rfloor n$ units of data sent and received per process, and at most $p+2\lceil \log_2 p \rceil \lfloor p/2 \rfloor$ local element copy/unpack/pack operations.*

3. ELIMINATING EXPLICIT DATA MOVEMENTS WITH DERIVED DATATYPES

We now employ Basic and Modified Bruck to implement the regular, collective all-to-all communication operation of MPI. In an `MPI_Alltoall(sendbuf,sendcount,sendtype,-recvbuf,recvcount,recvtype,comm)` call, each MPI process stores the elements $m_{i \to j}$ to the other processes consecutively in `sendbuf`. Elements can be arbitrarily structured as described by the `sendcount` and `sendtype` arguments. The received elements will be stored in `recvbuf` and can likewise be arbitrarily structured, independently of the structure of the elements to be sent; however, all sent and all received elements have the same structure and size. MPI requires data in `sendbuf` to remain unchanged after the operation, but `recvbuf` can be used for the R vector, as long as the structure of the data to be received is respected.

Listing 1 Communication step of basic Bruck with bounded vector datatype in MPI.

```
for (k=1; k<size; k<<=1) {
  Type_create_vector_bounded((size-k)*recvcount,
                             k*recvcount,(k<<1)*recvcount,
                             recvtype,&recvblocktype);
  MPI_Type_commit(&recvblocktype);
  MPI_Pack_size(1,recvblocktype,comm,&packsize);

  sendrank = (rank+k)%size;
  recvrank = (rank-k+size)%size;
  MPI_Sendrecv((char*)recvbuf+k*recvcount*recvextent,
               1,recvblocktype,sendrank,BRUCK,
               interbuf,packsize,MPI_PACKED,recvrank,BRUCK,
               comm,MPI_STATUS_IGNORE);
  pos = 0;
  MPI_Unpack(interbuf,packsize,&pos,
             (char*)recvbuf+k*recvcount*recvextent,
             1,recvblocktype,comm);

  MPI_Type_free(&recvblocktype);
}
```

Listing 2 Communication step of Modified Bruck with circular vector datatype in MPI.

```
for (k=1; k<size; k<<=1) {
  sendrank = (rank-k+size)%size;
  recvrank = (rank+k)%size;
  Type_create_vector_circular(size*recvcount,
                              recvrank*recvcount,
                              (size-k)*recvcount,
                              k*recvcount,
                              (k<<1)*recvcount,
                              recvtype,&recvblocktype);
  MPI_Type_commit(&recvblocktype);
  MPI_Pack_size(1,recvblocktype,comm,&packsize);

  MPI_Sendrecv(recvbuf,1,recvblocktype,sendrank,BRUCK,
               interbuf,packsize,MPI_PACKED,recvrank,BRUCK,
               comm,MPI_STATUS_IGNORE);
  pos = 0;
  MPI_Unpack(interbuf,packsize,&pos,
             recvbuf,1,recvblocktype,comm);

  MPI_Type_free(&recvblocktype);
}
```

3.1 Basic Bruck with derived datatypes

In Basic Bruck the sequence of elements sent out of R in each round k is regularly structured: each element $R[j]$, where the kth bit of j is set, is sent. In round k, blocks of 2^k elements with a stride of 2^{k+1} elements, with the last block possibly having fewer than 2^k elements, are sent and received as illustrated in Figure 1 (left). In the original paper [1], hand-coded pack and unpack routines copy the $\lfloor p/2 \rfloor$ elements to be sent into a contiguous buffer, and the received $\lfloor p/2 \rfloor$ elements into their correct positions in R. For an MPI implementation, a natural approach is to use a derived datatype for each communication round to describe the strided pattern of elements to be sent and received. This could be described by an MPI vector datatype, except for the last block that may contain fewer elements. Instead, the layout is described as either a) an indexed type with a possibly smaller last block, as b) an indexed block type where each element is indexed separately, or as c) a structured type consisting of a vector followed by a contiguous type for the last block. The first and second alternative use extra arrays for index and block length information that is mostly redundant, and are thus wasteful both in storage, set-up and processing time. The third alternative is likely to be more efficient. We use it here for the implementation of a derived, derived datatype constructor for *bounded vectors* whose type map is illustrated in Figure 2. The difference from the bounded vector to the MPI vector is that a bound on the total number of (basetype) elements spanned by the datatype is given instead of the count of the number of blocks. Using the bounded vector, the communication step of Basic Bruck can readily be implemented as shown in Listing 1. For the experimental comparison (see Section 4) we consider the following two implementations:

1. Basic Bruck with indexed type (basicBruck-ix)
2. Basic Bruck with bounded vector (basicBruck)

We contend that the bounded vector layout could readily and efficiently be handled by MPI library internal datatype engines, and native support of this datatype would therefore save in both setup and processing overhead compared to our implementation via existing datatype constructors. We also contend that bounded vector patterns occur in other applications.

In the Basic Bruck outlined in Listing 1, elements are received as a contiguous sequence of `MPI_PACKED` type, and unpacked into the strided blocks of the given `recvbuf` using again the bounded vector datatype. This is a correct MPI solution to the problem of receiving typed data as a contiguous sequence of elements, and necessary since the type signature (see [6, Chapter 4, page 84]) of `recvtype` is not directly known. Note that the `packsize` of an MPI datatype may be larger than the actual size occupied by the elements of that type; also note that sequences of packed elements can only (legally) be accessed through the `MPI_Pack` and `MPI_Unpack` functions. An alternative, more elegant solution would be to provide each MPI (derived) datatype with a datatype corresponding to the signature of the type, that is a contiguous listing of the basic types. Such *signature types* could be used for receiving typed, consecutive sequences and for allocating intermediate buffers without losing type information.

The first and last step of Basic Bruck entail copying elements from `sendbuf` to `recvbuf` and putting the received elements in `recvbuf` into correct order. For Step 3, the MPI pack-unpack functionality suffices if the reordering is done via an intermediate buffer. For Step 1, where elements of `sendtype` have to be copied into a buffer of elements of a possibly different `recvtype`, a *datatyped memory copy* operation is called for. This operation, sometimes called transpacking [5, 8], is not in MPI. In the implementation, we use a process-local `MPI_Sendrecv` to transfer typed data from `sendbuf` to `recvbuf`. Other alternatives are possible, but MPI library support for a datatyped copy operation would be natural and easily more efficient than by-hand solutions.

3.2 Modified Bruck with derived datatypes

Modified Bruck also sends elements from `recvbuf` in a strided pattern, but in round $k, 0 \leq k < \lceil \log_2 p \rceil$, process i sends elements $R[i+j]$ for which the kth bit of j is one. This layout can be described as a strided vector of blocks of 2^k elements starting from offset 2^k, wrapping around at index p to index 0. In order to implement Modified Bruck we introduce another, new datatype constructor for *circular vectors*, the type map of which is also shown in Figure 2. A circular vector is specified by its total extent (in number of basic elements), a bound (as for the bounded vector) on the actual extent to be occupied within the total extent, a start offset, a number of elements per block and a stride between blocks.

With the circular vector, the implementation of Modified Bruck is straightforward, and Listing 2 shows the communication step. The circular vector constructor can be implemented using the existing MPI datatype constructors, but there is a number of tedious, special cases to take care of. A higher type creation overhead is to be expected. As with the bounded vector, we contend that the data accesses specified by this derived datatype could more efficiently be directly embodied in MPI library-internal datatype handling mechanisms. For performance comparison, we give two implementations of Modified Bruck:

1. Modified Bruck with indexed type (modBruck-ix)
2. Modified Bruck with circular vector (modBruck)

3.3 Zero-copy Bruck with derived datatypes

The previous implementations receive elements into an intermediate buffer which is then unpacked into the `recvbuf` before the next communication round. It would be desirable to eliminate this overhead. For instance, a received element which will have to be sent further on in a later communication round could remain in and be sent directly out of the intermediate buffer with no need for unpacking into the `recvbuf`. We now explain in more detail how to completely eliminate any explicit unpacking of the intermediate buffers. We call the resulting implementation *Zero-copy Bruck*; the term was used similarly for other applications in [4].

We use the same communication and storage pattern as in Modified Bruck. When a new element for $R[i+j]$ is received by process i in round k, the kth bit of j is set. In the same round, element $R[i+j]$ is sent. We store the elements to be sent and received alternatingly in the `recvbuf` and an intermediate buffer. If j has no further bits $k' > k$ equal to one, the element is at its destination process and should be received directly into position $i+j$ in `recvbuf`. In general, elements for which the number of set bits $k' > k$ in j is even will be received into `recvbuf`, and elements with an odd number of set bits $k' > k$ will be received into the intermediate buffer. Conversely, elements j with an even number of set bits $k' > k$, will be sent out of the intermediate buffer, elements j with an odd number of set bits after k out of the `recvbuf`. Finally, in each round k, the first element of each segment of 2^k elements is a "new" element, and taken from position $((i-j) \bmod p)$ of `sendbuf` (another way to see this is that such elements have no set bits $k' < k$ in j, and thus have not been received in any previous round); this eliminates the copy operation of Step 1 of Modified Bruck. To implement the alternation between intermediate, send and receive buffer, we need to be able to determine for each index j how many bits are set in j after position k. We do this by computing a table of set bits in $j, 0 \leq j < p$, and in round k mask out the bits below k. The number of set bits in all j can easily be (pre)computed in $O(p)$ time steps:

```
bits[0] = 0;
for (j=1; j<size; j++) bits[j] = bits[j>>1]+(j&0x1);
```

Listing 3 Zero-copy Bruck in MPI using structured send and receive types.

```
MPI_Type_get_extent(sendtype,&lb,&sendtotal);
MPI_Type_get_extent(recvtype,&lb,&recvtotal);
sendtotal *= sendcount; recvtotal *= recvcount;
MPI_Type_size(recvtype,&recvsize); recvsize *= recvcount;

unsigned int mask = 0xFFFFFFFF;
for (k=1; k<size; k<<=1) {
  b = 0; j = k;
  do { // bit j set
    sendrank = (rank-j+size)%size;
    recvrank = (rank+j)%size;

    if ((bits[j&mask]&0x1)==0x1) { // to recvbuf
      recvblocks[b] = recvcount;
      recvindex[b] =
        (MPI_Aint)((char*)recvbuf+recvrank*recvtotal);
      recvtypes[b] = recvtype;

      if ((j&mask)==j) { // from sendbuf
        sendblocks[b] = sendcount;
        sendindex[b] =
          (MPI_Aint)((char*)sendbuf+sendrank*sendtotal);
        sendtypes[b] = sendtype;
      } else { // from intermediate
        sendblocks[b] = recvsize;
        sendindex[b] = (MPI_Aint)(interbuf+j*recvsize);
        sendtypes[b] = MPI_BYTE;
      }
    } else { // to intermediate
      recvblocks[b] = recvsize;
      recvindex[b] = (MPI_Aint)(interbuf+j*recvsize);
      recvtypes[b] = MPI_BYTE;

      if ((j&mask)==j) { // from sendbuf
        sendblocks[b] = sendcount;
        sendindex[b] =
          (MPI_Aint)((char*)sendbuf+sendrank*sendtotal);
        sendtypes[b] = sendtype;
      } else { // from recv
        sendblocks[b] = recvcount;
        sendindex[b] =
          (MPI_Aint)((char*)recvbuf+recvrank*recvtotal);
        sendtypes[b] = recvtype;
      }
    }
    b++; // next element
    j++; if ((j&k)!=k) j += k;
  } while (j<size);

  MPI_Type_create_struct(b,sendblocks,sendindex,sendtypes,
                         &sendblocktype);
  MPI_Type_commit(&sendblocktype);
  MPI_Type_create_struct(b,recvblocks,recvindex,recvtypes,
                         &recvblocktype);
  MPI_Type_commit(&recvblocktype);

  sendrank = (rank-k+size)%size;
  recvrank = (rank+k)%size;
  MPI_Sendrecv(MPI_BOTTOM,1,sendblocktype,sendrank,BRUCK,
               MPI_BOTTOM,1,recvblocktype,recvrank,BRUCK,
               comm,MPI_STATUS_IGNORE);

  MPI_Type_free(&recvblocktype);
  MPI_Type_free(&sendblocktype);
  mask <<= 1;
}
```

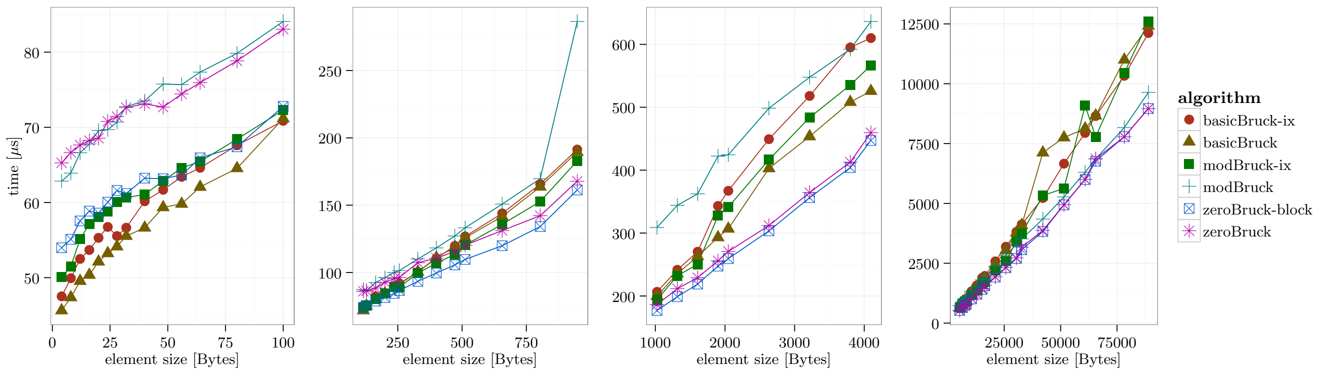

Figure 3: Running times of the six variants on InfiniBand cluster (Jupiter), $p = 36$, element size $n \in [4, 80000]$ Bytes.

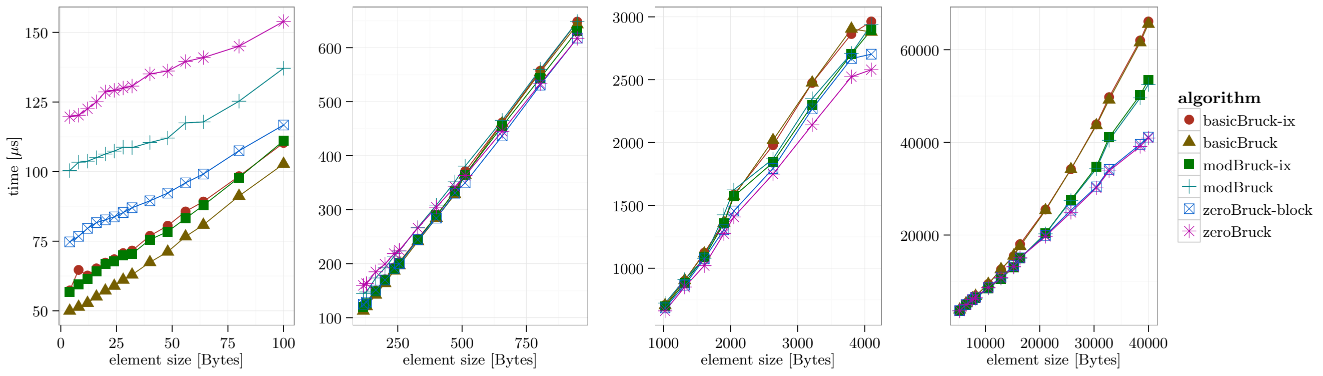

Figure 4: Running times of the six variants on shared-memory system (Mars), $p = 80$, element size $n \in [4, 40000]$ Bytes.

Listing 3 shows the full Zero-copy Bruck. For both the received and the sent blocks separate MPI derived datatypes are computed for each round, and is likely to have a non-negligible overhead. A possible improvement, not shown here, is to collapse consecutive elements into blocks of 2^k and $2^k - 1$ elements. An MPI library with sophisticated datatype preprocessing (*type normalization* [2]) might be able to do such improvements by itself, at the cost of an even higher type creation overhead.

Although structured (determined by the bits in each j), it is much less obvious that these patterns could be of general use, therefore we have not defined a new derived datatype constructor for the zero-copy implementation. Note that the element-wise analysis used to set up the datatypes does not hurt overall complexity, since there are $\lfloor p/2 \rfloor$ elements to be sent anyway.

A final point should be mentioned. Each element in the intermediate buffer is stored as a non-structured, contiguous sequence in order to avoid repeated overheads incurred by possibly structured element `sendtype` and `recvtype`. In the absence of the signature types mentioned in Section 3.1, these contiguous sequences are stored as `MPI_BYTE` sequences; this is not strictly correct since information about basetypes is lost. The `MPI_PACKED` type should have been used as subtype for the elements, but that would have made Listing 3 more confusing.

3.4 Persistent collectives

All three algorithm implementations, Basic Bruck, Modified Bruck and Zero-copy Bruck, create (and free) new, derived datatypes for each communication round. Basic and Modified Bruck use only one derived datatype per round, which is regular enough to be captured using (mostly) MPI vector and contiguous constructors (via the proposed, new type constructors for bounded and cyclic vectors). Hence, the overhead of setting up and using these types might be tolerable. For Zero-copy Bruck separate send and receive datatypes are created, both with a higher overhead by the index analysis and by the use of the MPI struct constructor. In all three cases it would be desirable to be able to amortize the type creation and destruction overheads over a number of `MPI_Alltoall` calls.

The overall structure of the derived datatypes is fully determined by the number of processes p, and could therefore potentially be reused from call to call. Datatypes are static, unchangeable objects in MPI, and since the types are created with `sendtype` and `recvtype` as basetypes and also depend on `sendcount` and `recvcount`, each `MPI_Alltoall` call either has to create (and free) these derived datatypes, or to maintain the created datatypes in a cache. An explicit means for the user to specify caching would be via a persistent version of `MPI_Alltoall`, in analogy with the persistent point-to-point communication operations [6, Section

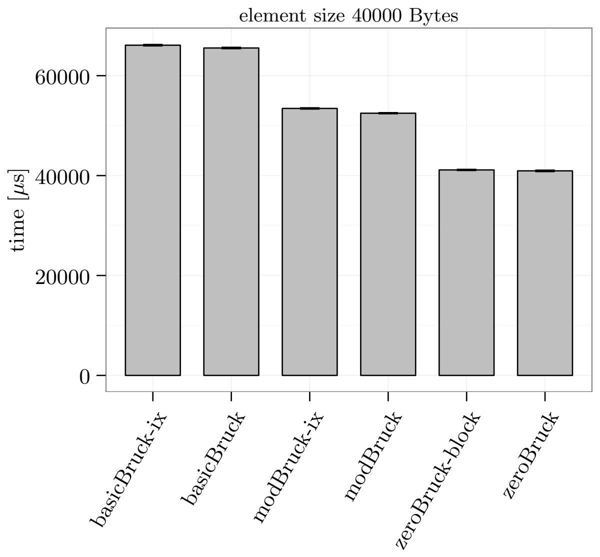

Figure 5: All versions with mpich on shared-memory system (Mars), $p = 80$, $n = 40000$ Bytes, 95% confidence intervals.

3.9]. An `MPI_Alltoall` *init* call would take the same arguments as `MPI_Alltoall` and bind these in a special request object also given as parameter to the call; in this object all required datatypes would be precomputed and stored. The corresponding `MPI_Alltoall` *start* operation simply consists of the $\lceil \log_2 p \rceil$-round loop performing the precomputed send and receive operations (that could themselves be persistent).

MPI currently does not specify *persistent collective operations*. To investigate the benefits of factoring out type creation overheads, we have implemented persistent all-to-all operations based on Zero-copy Bruck. As we discuss in the next section, persistence turns out even more beneficial for the irregular counterpart to `MPI_Alltoall`.

3.5 Irregular all-to-all communication

A basic assumption for the analysis of the Bruck algorithm is that elements have the same size. Nevertheless, the Bruck variations could be useful (and efficient) for the implementation of the irregular all-to-all collectives `MPI_Alltoallv` and `Alltoallw` for cases where the element sizes do not differ too much; for very irregular problems, employing an algorithm designed for the regular problem is far from optimal. Employing, for instance, Zero-copy Bruck to implement `MPI_Alltoallv` poses some interesting challenges that we discuss in the following.

A first difficulty is how to determine whether we are in a regular-enough case that Zero-copy Bruck should be used. Perhaps the user has this information, and could assert it to the MPI library; unfortunately, the `MPI_Alltoallv` interface does not provide an easy handle to convey such information. Because of this, the collective would have to do a global analysis to detect whether Zero-copy Bruck should be used. An `MPI_Allreduce` can be used to find the smallest and the largest element size, and based on this, the MPI processes can consistently select the desired algorithm. For regular-enough problems, the processes now allocate intermediate buffers with space enough for $p - 1$ elements of the maximum element size. Such buffers could conveniently be accessed using an MPI derived *bucket datatype* which divides an extent of memory into contiguous buckets of the same maximum size, but with a possibly different actual number of elements in each bucket. This datatype, also not in MPI, is the natural counterpart of the block-indexed datatype: instead of an array of indices and a fixed block size, the bucket type takes as arguments a maximum bucket size and an array of actual element counts for each bucket. Naturally, algorithm selection incurs overhead, which is particular hurtful for the smaller element sizes where the Bruck variations are efficient.

Another difficulty is that the MPI processes do not know in advance the actual sizes of the elements to be received in each of the $\lceil \log_2 p \rceil$ communication rounds. To handle this, each process in each communication round first receives and sends the sizes of the elements it is going to send and receive in that round (using, as in Basic Bruck, a bounded vector of integer counts). This "only" doubles the latency of the communication rounds since count vectors have at most $\lfloor p/2 \rfloor$ entries. With this information, correct, structured datatypes can be constructed, just as shown in Listing 3, using a bucket type for the 2^k-element blocks of differently sized elements. We note here that the trick of using the given `recvbuf` as intermediate buffer will not work, since this buffer may not have space for the possibly larger, intermediate elements. Instead, two intermediate buffers (of size $p - 1$ maximum elements) are allocated; the alternation described for Zero-copy Bruck still works, resulting in a zero-copy algorithm for not too irregular all-to-all communication. On the other hand, using an explicit, intermediate buffer instead of `recvbuf` may have advantages also for the regular problem, namely if `recvtype` is a complicated, structured type: in that case each intermediate element copied into the `recvbuf` is processed by the MPI datatype engine, and could incur an undesired overhead.

Using Zero-copy Bruck to implement `MPI_Alltoallv` incurs a two-element `MPI_Allreduce` overhead and doubles the number of send-receive operations. For small data, such an implementation of `MPI_Alltoallv` would be up to a factor of two slower than the Zero-copy Bruck implementation of `MPI_Alltoall`. However, the extra overhead depends only on p, and on the send-receive types and counts. A persistent version of `MPI_Alltoallv` would make it possible to fully isolate both the algorithm selection and the datatype creation overhead. For comparison, we have implemented such a version, although it has no counterpart in current MPI.

4. EXPERIMENTAL EVALUATION

As can be inferred from Section 3, our hypothesis is that Modified Bruck which saves an $O(pn)$ local reordering step will improve over Basic Bruck, and that Zero-copy Bruck which completely eliminates explicit, local copy operations will improve over Modified Bruck, all on the *assumption that the MPI derived datatype mechanism does not lead to excessive overheads.* To test these expectations, we have implemented the three variations which we term *basicBruck*, *modBruck* and *zeroBruck*, respectively. For additional comparisons *basicBruck-ix* uses an MPI indexed datatype instead of the bounded vector, *modBruck-ix* uses an indexed-block datatype instead of the circular vector, and *zeroBruck-*

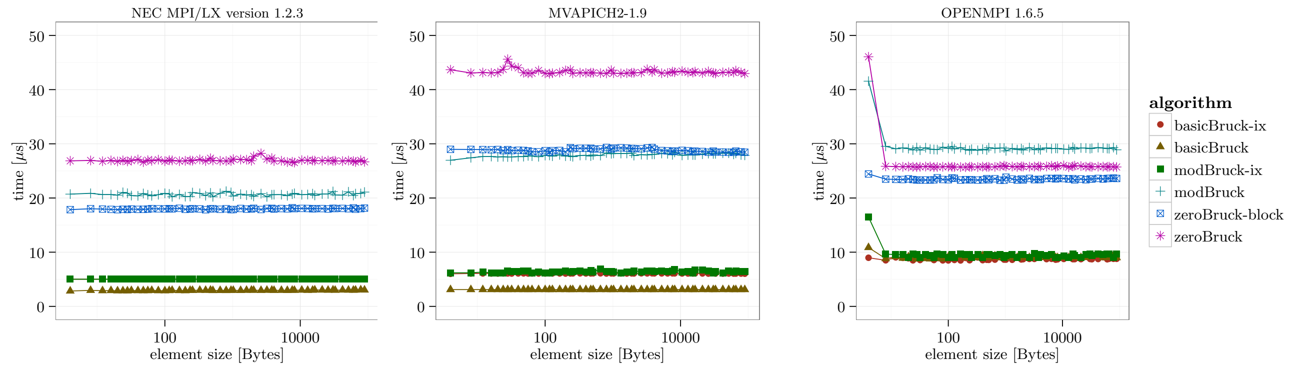

Figure 6: Type creation overheads with vendor MPI, `mvapich` and OpenMPI on InfiniBand cluster (Jupiter).

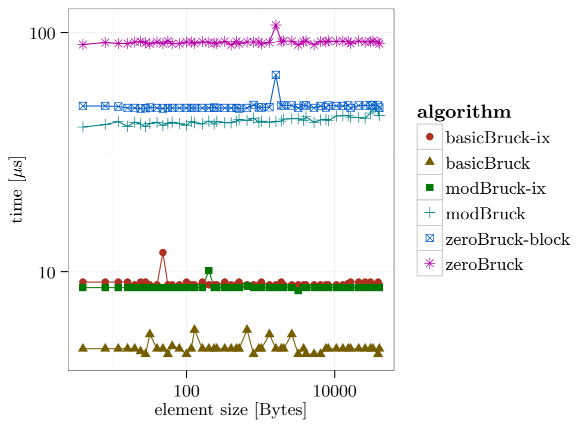

Figure 7: Type creation overheads with `mpich` on shared-memory system (Mars).

blocks creates structured types with fewer, but longer 2^k-element blocks. For each variant, we can disable actual communication in order to measure the derived datatype creation and freeing overheads. Zero-copy Bruck has also been given a persistent implementation, termed *persBruck*; we have also used the zero-copy algorithm to implement `MPI_Alltoallv`, both in non-persistent *zeroBruckV* and persistent *persBruckV* versions. All code is available from the authors upon request. Finally, we also implemented Modified Bruck directly in the `mvapich` library and compared it to the `mvapich` version. Interestingly, `mvapich` (R. Thakur, personal communication) already implements Basic Bruck using an indexed-block datatype; OpenMPI (George Bosilca, personal communication) instead used an indexed type to maintain larger 2^k-element blocks of elements for each round.

For our evaluation we currently have access to two small systems. The first is a 36-node, 576-core InfiniBand cluster (Jupiter) with two 8-core 2.3GHz AMD 6134 Opteron processors/node and a Mellanox MT4036 QDR switch. The second, an 80-core shared-memory system (Mars) based on Intel 10-core 2.0GHz Westmere-EX E7-8850 processors. Although the latter is a shared memory system, we use it as an approximation to a fully connected, homogeneous system. To have as far as possible homogeneous communication between processes, the measurements on the InfiniBand cluster have been done with one MPI process per node. On the Jupiter InfiniBand cluster we use the vendor MPI library, on the Mars shared-memory system we have used `mpich 3.0.4`.

In our experiments the basic datatype for send and receive buffers is `MPI_INT`, so element sizes are multiples of four bytes. Each single measurement was repeated at least 40 times. To compensate for system noise, sensitive measurements in the microseconds range were repeated 300 times for each element size. We applied Tukey's outlier filter (see, e.g., [3]: for the upper quartile (Q_3) of the sample all measurements larger than $Q3+1.5IQR$ are removed, where IQR denotes the interquartile range. We also computed the 95% confidence interval for each element size, but show confidence intervals only in the barcharts. When intervals do not overlap, the results are significantly different at a 95% confidence level.

4.1 Regular all-to-all communication

We first compare the performance of the six implementation variants for regular all-to-all communication on the two systems. Element sizes have been chosen in a larger interval (up to $n = 80000$ Bytes) than that in which the Bruck idea is better than a direct algorithm, in order to amplify the asymptotic differences between the improvements. The results are shown in Figures 3 and 4. We emphasize that the measured implementations are fully self-contained `MPI_Alltoall` algorithms and include all datatype manipulation overheads.

Especially on Mars, Zero-copy Bruck clearly performs asymptotically better than both Modified and Basic Bruck. This is shown in detail in Figure 5, where the improvement is by more than 30% for $n = 40000$ Bytes. For small element sizes the datatype overhead of the zero-copy variants is too large to make these implementations competitive. The best performing variant here is Basic Bruck implemented with the bounded vector type. Modified Bruck with circular vector is surprisingly bad: although a regular layout is described in terms of MPI vector and contiguous types, putting these together with an MPI structured type takes more time than describing the layout by an indexed type.

4.2 Type creation overhead

We explicitly measured the type creation (and destruction) overhead for all six variants by disabling all commu-

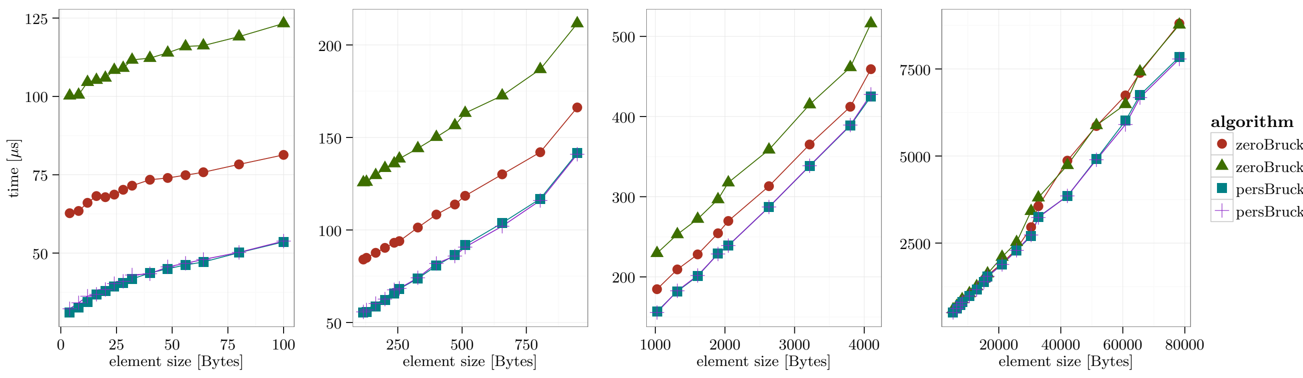

Figure 8: Persistent and irregular Zero-copy Bruck on InfiniBand cluster (Jupiter), $p = 36$, element size $n \in [4, 80000]$ Bytes.

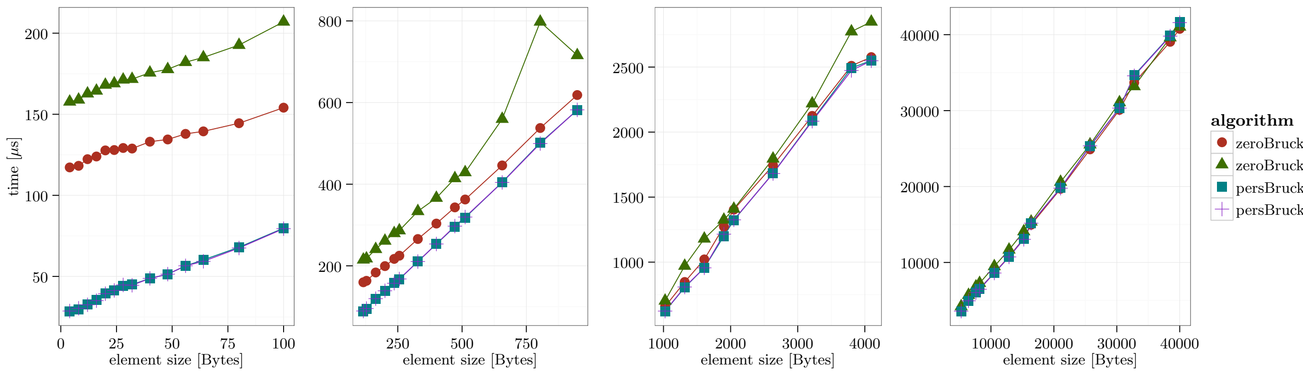

Figure 9: Persistent and irregular Zero-copy Bruck on shared-memory system (Mars), $p = 80$, element size $n \in [4, 40000]$ Bytes.

nication. The results are shown in Figure 6 and 7. The results confirm the speculations: the structured types for Zero-copy Bruck and the circular vector incur significantly higher overheads than the other types. On Jupiter we tried three different MPI libraries, and as Figure 6 shows there are important performance differences, with `mvapich` being particularly slow in the setup/destruction of the structured types for Zero-copy Bruck.

4.3 Persistent and irregular all-to-all communication

We analyze the performance of the persistent versions of Zero-copy Bruck for both `MPI_Alltoall` and `MPI_Alltoallv`. In the experimental setup we use `MPI_Alltoallv` to solve the same regular problem as `MPI_Alltoall`. This gives us an idea of the best-case overhead incurred by using the irregular collective operation. As explained, we expect more than a factor of two difference for small element sizes. This is clearly confirmed by Figures 8 and 9. The persistent versions have no extra overhead. For small element sizes, e.g., 64 Bytes, the persistent versions are more than a factor 2 (for regular all-to-all) and up to a factor 3 (for the irregular version) faster than the non-persistent counterparts. A nice property is that `MPI_Alltoall` and `MPI_Alltoallv` perform equivalently in their persistent variants: all the extra overhead in the irregular algorithm is taken care of in the `MPI_Alltoallv` *init* operation.

4.4 Incorporating into existing MPI library

Finally, we implemented Modified Bruck (in the version using an indexed type) inside the `mvapich` library; as mentioned `mpich` and `mvapich` have their own implementation of Basic Bruck, which is enabled for a small range of element sizes. The intention is to show that the savings of Modified Bruck indeed give a significantly better (in the statistical sense) implementation than Basic Bruck, also for the range of element sizes where this algorithm is normally employed. The results are shown in Figure 10 for n up to 300 Bytes. Although the difference is only a few percent, it is statistically significant as the barchart shows.

5. SUMMARY AND OUTLOOK

We discussed the use of MPI derived datatypes to implement and improve the all-to-all algorithm of Bruck et al. [1]. Without modifying existing MPI libraries, we showed that the expected improvements can also be achieved in practice, and that the overheads by using datatypes can in many cases be (surprisingly) tolerable with modern MPI libraries.

We made a number of observations for datatype-oriented programming in MPI:

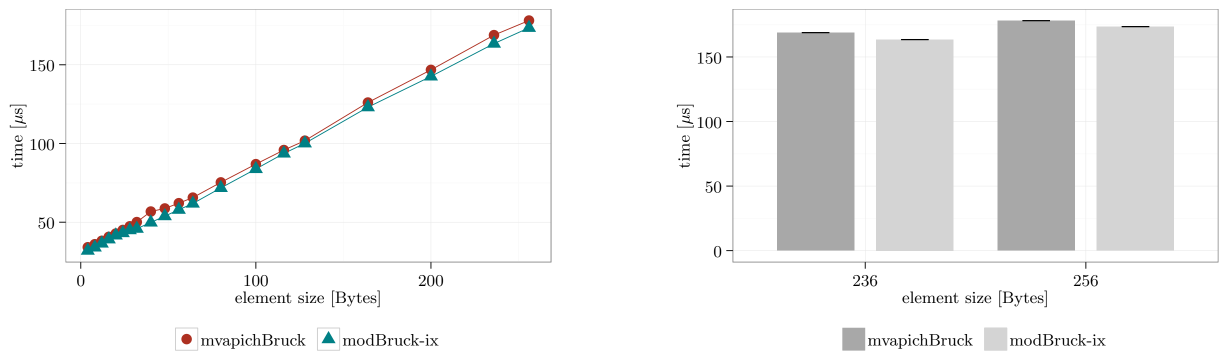

Figure 10: Bruck et al. implementation in `mvapich` versus Modified Bruck-ix in `mvapich` on shared-memory system (Mars), $p = 80$, for the interval $n \in [4, 300]$ Bytes.

- For correct handling of intermediate buffers holding elements of a structured type, having access to a *signature type* for (derived) datatypes would be convenient, and would alleviate the need for using `MPI_PACKED` and the often used, but incorrect resort to `MPI_BYTE`.
- A *datatyped copy* for process local copying of MPI structured data would likewise be convenient, and can likely be more efficient than work-arounds with process-local `MPI_Sendrecv` and/or `MPI_Pack`/`MPI_Unpack`; in analogy with the `MPI_Reduce_local` operation introduced in MPI 2.2.
- We proposed new, derived datatype constructors for *bounded* and *circular vectors*; both can more or less easily be implemented with the existing constructors, but a native implementation in the MPI library would likely be more efficient.
- We also discussed a *bucket datatype* constructor which is the natural complement to the MPI block-indexed type.
- Finally we experimentally introduced and evaluated *persistent all-to-all collectives*, and showed that persistence can effectively be used to hide overheads in collective operations.

The discussion in this paper is related to and continues the discussion of (a)symmetries between MPI collective operations and datatypes in [10]. Part of these results were described in the first author's invited talk at EuroMPI 2013.

We mention a small, open problem: Zero-copy Bruck allocates intermediate buffer for p elements, but each round receives at most $\lfloor p/2 \rfloor$ elements. Is it possible to do with only $\lfloor p/2 \rfloor$ element intermediate buffer space and still be zero-copy?

6. REFERENCES

[1] 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.

[2] 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.

[3] J. Hedderich and L. Sachs. *Angewandte Statistik*. Springer, 14 edition, 2012.

[4] 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.

[5] F. G. Mir and J. L. Träff. 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*, volume 5205 of *Lecture Notes in Computer Science*, pages 141–150. Springer, 2008.

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

[7] H. Ritzdorf and J. L. Träff. Collective operations in NEC's high-performance MPI libraries. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, page 100, 2006.

[8] R. B. Ross, R. Latham, W. Gropp, E. L. Lusk, and R. Thakur. Processing MPI datatypes outside MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 16th European PVM/MPI Users' Group Meeting*, volume 5759 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2009.

[9] R. Thakur, W. D. Gropp, and R. Rabenseifner. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications*, 19:49–66, 2005.

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