To INT_MAX... and beyond! Exploring large-count support in MPI

Jeff R. Hammond
Parallel Computing Lab
Intel Corp.
jeff_hammond@acm.org

Andreas Schäfer
Friedrich-Alexander-Universität
Erlangen-Nürnberg
andreas.schaefer@fau.de

Rob Latham
Mathmatics and Computer Science Division
Argonne National Lab.
robl@mcs.anl.gov

***Abstract*—To describe a structured region of memory, the routines in the MPI standard use a (count, datatype) pair. The C specification for this convention uses an `int` type. Since C `int` types are nearly always 32 bits large and signed, counting more than 2^{31} elements poses a challenge. Instead of changing the existing MPI routines, and all consumers of those routines, the MPI Forum asserts users can build up large datatypes from smaller types. To evaluate this hypothesis and to provide a user-friendly solution to the large-count issue, we have developed BigMPI, a library on top of MPI that maps large-count MPI-like functions to MPI-3 standard features. BigMPI demonstrates a way to perform such a construction, reveals shortcomings of the MPI standard, and uncovers bugs in MPI implementations.**

I. Introduction

The Message Passing Interface [1], [2], [3], [4] defines a broad set of functionality for writing parallel programs, especially across distributed computing systems. Now more than 20 years old, MPI continues to be widely used and has met the challenges of post-petascale computing, including scaling out to millions of cores [5]. In order to scale up, in terms of problem size, one needs to be able to describe large working sets. The existing (count, datatype) pair works well, until the "count" exceeds the range of the native integer type (in the case of the C interface, `int`, which is 32 bits on most current platforms). We call this the "large-count" problem.

When drafting MPI-3 the MPI Forum took a minimalist approach large-count support [6]. The forum introduced a handful of MPI_Foo_x routines that provide a large-count equivalent of an existing MPI_Foo to make rudimentary large-count support possible. To be explicit, in this context, Foo is "Get_elements", "Type_size", "Type_get_extent", "Type_get_true_extent", "Status_set_elements", which is the minimal set of functions that must support large counts in order for one to be able to deal with derived datatypes that represent large counts. After lengthy deliberation, the Forum asserted that "just use datatypes" is a sufficient solution for users [7]. For example, one can describe 4 billion bytes as 1 billion 4-byte integers. Or, one could use contiguous MPI dataypes to describe 16 billion bytes as 1,000 16 million-byte chunks. For these simple examples, it is easy to envision a solution. It is only through implementing the proposed approach for all cases in MPI that one discovers the challenges hidden in such an assertion.

BigMPI provides a high-level library that attempts to support large counts. It was written to test the Forum's assertion that datatypes are sufficient for large-count support and to provide a drop-in library solution for applications requiring large-count support. In this context, "large-count" is any count that exceeds `INT_MAX`. BigMPI makes the smallest possible changes to the MPI standard routines to enable large counts, minimizing application changes.

BigMPI is designed for the common case where one has a 64-bit address space and is unable to do MPI communication on more than 2^{31} elements despite having sufficient memory to allocate such buffers. As systems with more than 2^{63} bytes (8192 PiB) of memory per node are unlikely to exist for the foreseeable future – the total system memory capacity for an exascale machine has been predicted to be 50-100 petabytes [8] – BigMPI makes no attempt to support the full range of `MPI_Count` (possibly a 128-bit integer) internally; rather it uses `size_t` and `MPI_Aint`, as these reflect the limit of the available memory rather than the theoretical filesystem size (as `MPI_Count` does).

II. Background

The MPI standard provides a wide range of communication functions that take a C `int` argument for the element count, thereby limiting this value to `INT_MAX` or less. This means that one cannot send, e.g. 3 billion bytes using the `MPI_BYTE` datatype, or a vector of 5 billion integers using the `MPI_INT` type, as two examples.

These limitations may seem academic in nature, as 2 billion `MPI_DOUBLE` equate to 16GB and one might think that applications may rarely ever need to transmit that much data, as there may be less memory available for the whole address space in which the MPI process is running. Two recent trends may render this limit increasingly impractical: first, growing core counts per CPU mean larger data portions per MPI process and second, Big Data applications may demand more memory per core than the traditional 2GB for computer simulations.

If the user code manually packs data, either for performance [9] or encoding reasons ([10] [11], then the MPI implementation may be given just an array of `MPI_BYTE`, which further reduces the maximum message size (e.g. 250 million for C `double`).

A natural workaround is to use MPI derived datatypes. While it is plausible that application developers will know typical data sizes and can thus intercept calls which may exceed the `INT_MAX` limit, another scenario is harder to solve: problem solving environments [12], [13] and computational

```
int MPIX_Send_x(const void *buf, MPI_Count count,
                MPI_Datatype dt, int dest,
                int tag, MPI_Comm comm)
{
    int rc = MPI_SUCCESS;
    if (likely (count <= INT_MAX )) {
        rc = MPI_Send(buf, (int)count, dt, dest, tag, comm);
    } else {
        MPI_Datatype newtype;
        MPIX_Type_contiguous_x(count, dt, &newtype);
        MPI_Type_commit(&newtype);
        rc = MPI_Send(buf, 1, newtype, dest, tag, comm);
        MPI_Type_free(&newtype);
    }
    return rc;
}
```

Fig. 1. The implementation of large-count Send, which serves as a template for many other MPI-3 routines.

libraries [14], [15] operate on data structures with user-defined dimensions. To ensure correctness, developers would need to safeguard all communication functions which operate on user data.

This paper focuses on the issues with the C interface and we use the well-known convention In_ILn_LPn_P to refer to the sizes of the C types `int`, `long`, and `void*`, respectively. For ILP32 systems, the largest buffer one can allocate is 2^{32} bytes (4GiB), while MPI can handle buffers of up to 2GiB; the factor of two difference is almost never a problem since 4GiB of `int` for example requires only a count of only 2^{30}. A problem emerges in IL32P64 and I32LP64 systems because it is possible to allocate more memory in a buffer than can be captured with an integer count and built-in datatype. For example, a vector of 3 billion floats requires 12 GB of memory but cannot be communicated using any communication routine using built-in datatypes.

III. Design

In this section, we describe the mapping from large-count variants of MPI-like communication functions to MPI-3 functions, which usually involves the creation of a large-count datatype, but possibly much more. BigMPI implements all variants of send and receive, blocking and nonblocking variants of the homogeneous collectives (bcast, gather, scatter, allgather, alltoall) and RMA (put, get, accumulate, get_accumulate) along the lines of the example for MPI_Send, shown in Figure 1. This class of routines provides the most commonly used MPI functionality, so for many codes the Forum has been proven correct. However, as we will see in Section III-A, not all parts of the MPI standard were so straightforward.

The critical function in all of the large-count implementations noted above is `MPIX_Type_contiguous_x`, which emits a single datatype that represents up to `SIZE_MAX` elements. This utility routine allows us to implement large-count support in a straightforward fashion since all instances of $(large\_count, type)$ are mapped to $(1, large\_type)$ by this function. An associated decoder function extracts the original $large\_count$ from a user-defined datatype; this function is employed within the user-defined reduction operations. Decoding a datatype is nontrivial even for such a simple case – we must call `MPI_Type_get_envelope` and `MPI_Type_get_contents` three times each just to unwind the result of `MPIX_Type_contiguous_x`. BigMPI is boon to the majority of application programmers that are unfamiliar with such features in the MPI standard by virtue of hiding these details.

```
int MPIX_Type_contiguous_x(MPI_Count count,
                           MPI_Datatype oldtype,
                           MPI_Datatype * newtype)
{
    assert(count<SIZE_MAX); /* has to fit into MPI_Aint */
    MPI_Count c = count/INT_MAX, r = count%INT_MAX;

    MPI_Datatype chunks, remainder;
    MPI_Type_vector(c, INT_MAX, INT_MAX, oldtype, &chunks);
    MPI_Type_contiguous(r, oldtype, &remainder);

    MPI_Aint lb /* unused */, extent;
    MPI_Type_get_extent(oldtype, &lb, &extent);

    MPI_Aint remdisp          = (MPI_Aint)c*INT_MAX*extent;
    int blklens[2]            = {1,1};
    MPI_Aint disps[2]         = {0,remdisp};
    MPI_Datatype types[2]     = {chunks,remainder};
    MPI_Type_create_struct(2,
        blklens, disps, types, newtype);

    MPI_Type_free(&chunks);
    MPI_Type_free(&remainder);

    return MPI_SUCCESS;
}
```

Fig. 2. Function for construction a large-count contiguous datatype. A vector type describes a series of adjacent chunks, and a struct type picks up any remaining data in case the count is not evenly divisible.

Other datatypes can be supported easily within BigMPI, but this is not a high priority because the primary goal is to solve the large-count problem for users that are not currently making use of derived datatypes. A user that employs derived datatypes in their code already is likely capable of implementing their own large-count support already. Nonetheless, the release version of BigMPI will support large-count equivalents of all of the existing datatype constructors.

A. Reductions

Large-count support for reductions poses a challenge, particularly in the nonblocking case. For the blocking case, it is straightforward to block a single large-count operation into multiple normal-count (count$< 2^{31}$) operations (we will refer to this as chunking); however, as it is not possible to return a single request object associated with more than one nonblocking operation, we cannot implement nonblocking reductions in this manner. Generalized requests - the MPI-standard way to implement non-blocking operations in a library - are not a viable alternative for reasons that have been documented in other work ([16]). For the blocking case, it is desirable to use chunking because many MPI implementations have optimized implementations of reductions for built-in reduction operations.

The MPI standard stipulates that built-in reduction operations can be used with built-in types in the case of reductions, which means that performing a reduction on a vector of N doubles using count=N and type=`MPI_DOUBLE` is compatible with `MPI_SUM`, whereas the same reduction performed using a contiguous datatype to represent the vector of doubles

requires a user-defined reduction operation. Thus, BigMPI creates user-defined operations corresponding to all the built-in reductions acting on contiguous datatypes. Inside of these reduction operations, the datatype is decoded and the reduction performed using multiple calls to `MPI_Reduce_local` and the appropriate built-in reduction operation. This is a general solution that works for both the blocking and nonblocking cases, at least for out-of-place reductions.

Unfortunately, user-defined reductions cannot support `MPI_IN_PLACE`. The user-defined reduce function interface (see below) does not expose the information required to do an arbitrary in-place reduction.

```
MPI_User_function(void* invec, void* inoutvec,
                  int *len, MPI_Datatype *datatype);
```

As user-defined reduce operations are the only way to implement large-count nonblocking reductions, *we identify this as the first example where MPI-3 lacks the necessary features to support large counts effectively, as the inefficiency associated with user-defined reductions and lack of support for in-place reductions has a substantial negative impact on users.*

B. Vector-argument collectives

Vector-argument collectives (henceforth v-collectives) are the generalization of e.g. `MPI_Scatter`, `MPI_Gather`, `MPI_Alltoall` when the count but not the datatype varies across processes. When datatypes are used to support large counts, all of these operations must be mapped to `MPI_Alltoallw` because each large count will be mapped to a different user-defined datatype, and `MPI_Alltoallw` is the only collective that supports a vector of datatypes. Using `MPI_Alltoallw` to implement, e.g., a large-count `MPI_Scatterv` is particularly inefficient because the former assumes inputs from every process, whereas the latter only uses the input from the root. However, the overhead of scanning a vector of counts where all but one is zero is almost certainly inconsequential compared to the cost of transmitting a buffer of 2^{31} bytes.

The v-collectives encounter a second, more subtle issue due to the mapping to `MPI_Alltoallw`. Because this function takes a vector of datatypes, the displacements into the input and output vectors are given in bytes, not element count, and the type of this offset is a C integer. This creates an overflow situation *even in the case where the input buffer is less than 2 GiB* because a vector of one billion alternating integers and floats may require an byte-offset in excess of 2^{31}. Thus, `MPI_Alltoallw` is not an acceptable solution for the large-count v-collectives because of the likelihood of overflowing in the displacement vector. The use of C integer instead of `MPI_Aint` for the displacement vector in the collective operations added prior to MPI-3 is a unfortunate oversight that cannot be rectified without breaking backwards compatibility.

Fortunately, the overflow issue with displacements in `MPI_Alltoallw` is resolved using the neighborhood collectives introduced in MPI-3, which do use `MPI_Aint` for displacements. On the other hand, neighborhood collectives require an appropriate communicator, which must be constructed prior to calling `MPI_Neighborhood_alltoallw`. BigMPI creates a distributed graph communicator using `MPI_Dist_graph_adjacent` on-the-fly for every invocation of the large-count v-collectives, which instead assumed to be insignificant overhead compared to the data movement entailed in such an operation.

Thus, the implementation of large-count v-collectives using `MPI_Neighborhood_alltoallw` requires two $O(n_{proc})$ setup steps: the first is to allocate and populate the vectors of send and receive counts, displacements and datatypes and the second is to create a distributed graph communicator. Figures III-B and III-B contain the implementation of these functions, which are included in their entirety to illustrate that, while the mapping from v-collectives to `MPI_Neighborhood_alltoallw` is feasible, is rather involved and in some cases quite unnatural. Creating the vector of datatypes requires $O(n_{proc})$ calls to `BigMPI_Type_contiguous_x`, which itself requires six MPI calls, although all of these are expected to be inexpensive.

An alternative approach to implementing large-count v-collectives is to map these to point-to-point operations, although this only works for blocking operations due to the inability to aggregate requests, as described above. Given that large-count v-collectives are well outside of the regime where latency-oriented optimizations like recursive-doubling are important, this is unlikely to have a significant impact on performance and it eliminates the need for some of the $O(n_{proc})$ setup steps. The MPI standard describes every collective in terms of its implementation in terms of send-recv calls; the point-to-point BigMPI implementation follows these recipes closely: (1) nonblocking receives are pre-posted by the root or all ranks as appropriate; (2) the root or all ranks then call nonblocking send; (3) all ranks then call Waitall. As the large-count BigMPI send-recv functions are used, there is no need for $O(n_{proc})$ vectors of datatypes, etc. – only a vector of `MPI_Request` objects for the nonblocking operations is required.

A third implementation of v-collectives is to use RMA (one-sided) that follows the same traffic pattern as the point-to-point implementation. In this case, a MPI window must be created associated with the source (target) buffers and `MPI_Get` (`MPI_Put`) operations used for moving data. The most appropriate synchronization mode for mapping collectives to RMA is `MPI_Win_fence`, although one could use passive target instead. In the event that a future version of the MPI standard introduces a nonblocking equivalent of `MPI_Win_fence` or `MPI_Win_unlock_all`, these could be used to implement nonblocking v-collectives in terms of RMA; at least within MPI-3, we are limited to the blocking case. The RMA implementation was prototyped in BigMPI but is not currently implemented. The current state of RMA implementations map one-sided operations to two-sided ones internally, thus we would expect to see no performance benefit from BigMPI's RMA approach. However, it is possible that noticeable performance improvements will be observed if RMA operations exploit RDMA hardware.

While not named as such, `MPI_Reduce_scatter` is a v-collective. BigMPI currently does not yet support this function but it is straightforward to implement in terms of `MPI_Reduce` and `MPI_Scatterv`, which will be the basis for the BigMPI implementation when it exists.

```
void BigMPI_Convert_vectors(int num,
                            int splat_old_count,
                            const MPI_Count oldcount,
                            const MPI_Count oldcounts[],
                            int splat_old_type,
                            const MPI_Datatype oldtype,
                            const MPI_Datatype oldtypes[],
                            int zero_new_displs,
                            const MPI_Aint olddispls[],
                            int newcounts[],
                            MPI_Datatype newtypes[],
                            MPI_Aint newdispls[])
{
    assert(splat_old_count || (oldcounts!=NULL));
    assert(splat_old_type  || (oldtypes!=NULL));
    assert(zero_new_displs || (olddispls!=NULL));

    MPI_Aint lb /* unused */, oldextent;
    if (splat_old_type) {
        MPI_Type_get_extent(oldtype, &lb, &oldextent);
    } else {
        /* !splat_old_type implies ALLTOALLW,
            which implies no displacement zeroing. */
        assert(!zero_new_displs);
    }

    for (int i=0; i<num; i++) {
        /* counts */
        newcounts[i] = 1;

        /* types */
        MPIX_Type_contiguous_x(oldcounts[i],
                       splat_old_type ? oldtype : oldtypes[i],
                       &newtypes[i]);
        MPI_Type_commit(&newtypes[i]);

        /* displacements */
        MPI_Aint newextent;
        /* If we are not splatting old type, it implies
         *  ALLTOALLW, which does not scale the
         * displacement by the type extent,
         * nor would we ever zero the displacements. */
        if (splat_old_type) {
            MPI_Type_get_extent(newtypes[i], &lb, &newextent);
            newdispls[i] = (zero_new_displs ? 0 :
                            olddispls[i]*oldextent/newextent);
        } else {
            newdispls[i] = olddispls[i];
        }
    }
    return;
}
```

Fig. 3. Function for populating the vector inputs for `MPI_Neighborhood_alltoallw` for the various v-collectives.

Unfortunately, there is no way to implement nonblocking v-collectives using the aforementioned approaches. In the case of the neighborhood collective implementation, we cannot free the vector temporaries holding the counts, displacements and datatypes until the operation has completed. If callback functions associated with request completion were present in the MPI standard (see Ref. [17] for a proposal of this), then it would be possible to free the temporary buffers using this callback. As noted already, since one cannot associate a single request with multiple nonblocking operations, the point-to-point implementation is not viable for the nonblocking v-collectives. Finally, all relevant forms of MPI RMA synchronization have blocking semantics and thus cannot be used to implement nonblocking collectives.

We identify nonblocking v-collectives as the second example where MPI-3 lacks the necessary features to support large counts.

```
int BigMPI_Create_graph_comm(MPI_Comm comm_old, int root,
                             MPI_Comm * comm_dist_graph)
{
    int rank, size;
    MPI_Comm_rank(comm_old, &rank);
    MPI_Comm_size(comm_old, &size);

    /* in the all case (root == -1), every rank is a
     * destination for every other rank;
     * otherwise, only the root is a destination. */
    int indeg  = (root == -1 || root==rank) ? size : 0;
    /* in the all case (root == -1), every rank is a
     * source for every other rank;
     * otherwise, all non-root processes are the
     * source for only one rank (the root). */
    int outdeg = (root == -1 || root==rank) ? size : 1;

    int * srcs = malloc(indegree*sizeof(int));
    assert(srcs!=NULL);
    int * dsts = malloc(outdegree*sizeof(int));
    assert(dsts!=NULL);

    for (int i=0; i<indegree; i++) {
        srcs[i] = i;
    }
    for (int i=0; i<outdegree; i++) {
        dsts[i] = (root == -1 || root==rank) ? i : root;
    }

    int empty = MPI_WEIGHTS_EMPTY;
    int unwtd = MPI_UNWEIGHTED;
    int rc = MPI_Dist_graph_create_adjacent(comm_old,
                indeg, srcs, indeg==0 ? empty : unwtd,
                outdeg, dsts, outdeg==0 ? empty : unwtd,
                MPI_INFO_NULL, 0 /* reorder */,
                comm_dist_graph);

    free(srcs);
    free(dsts);

    return rc;
}
```

Fig. 4. Function for constructing the distributed graph communicator that allows the mapping of both rooted (e.g. `MPI_Gatherv`) and non-rooted (e.g. `MPI_Allgatherv`) collectives to `MPI_Neighborhood_alltoallw`.

C. Neighborhood collectives

The implementation of large-count neighborhood collectives is straightforward using the approach noted above for mapping v-collectives to `MPI_Neighborhood_alltoallw`, except that we omit the creation of the distributed graph communicator. All aforementioned issues with the nonblocking cases still exist, as temporary vectors are still required for the mapping of $(large\_count, type)$ to $(1, large\_type)$ for all ranks. Thus, *we identify nonblocking neighborhood collectives as the third example where MPI-3 lacks the necessary features to support large counts.*

D. Interface

The BigMPI API follows the pattern of `MPI_Type_size(_x)`: all BigMPI functions are identical to their corresponding MPI ones except that they end with `_x` to indicate that the count arguments have the type `MPI_Count` instead of `int`. Following the MPICH convention, BigMPI functions use the MPIX namespace because they are not in the MPI standard. It is a trivial matter of preprocessing to support arbitrary namespacing in the library to make it more friendly to other implementers that may wish to support it as an extension in their library.

BigMPI has both a Cmake and an Autotools build system for compatibility with third-party tools that wish to configure BigMPI automatically. A generic programming environment composed of a C99 compiler and "count-safe" (i.e. one that supports large counts internally) implementation of MPI-3 is required by BigMPI.

E. Limitations

As previously stated, BigMPI does not actually support the full range of `MPI_Count`, but rather only the range of the address space (i.e. `size_t` and `MPI_Aint`), as buffers larger than the address space are rather difficult to allocate.

BigMPI only supports built-in datatypes. Code already using derived-datatypes should already be able to handle large counts without BigMPI. However, see Section IV-B for an example of HINDEXED not being sufficient.

Support for `MPI_IN_PLACE` is not implemented in some cases (e.g. where it is impossible) and implemented inefficiently (i.e. via a buffer copy) in others. Using `MPI_IN_PLACE` is discouraged at the present time although it is expected that it will be supported efficiently whenever possible in the release version of BigMPI.

BigMPI requires C99. Fifteen years is more than enough time for compiler implementors interested in supporting ISO languages to provide a C99 compiler.

The MPI-3 standard supports language bindings for C and Fortran – the latter via three different mechanisms (`mpif.h`, `use mpi` and `use mpi_f08`). Currently, BigMPI only provides a C interface, but a Fortran 2003 interface to the C API via `ISO_C_BINDING` is planned. It is expected that C++ programmers will be able to make use of the C interface and can implement wrappers consistent with their own style of C++ programming.

F. Performance optimizations

BigMPI is optimized for the case when count is smaller than 2^{31} with a `likely_if` macro to minimize the performance hit for the common case. The aim is for users to call the BigMPI routines directly, instead of inserting a branch for the large-count case themselves. It is assumed that branch mis-prediction is significantly less expensive than transferring gigabytes of data across the network.

While software overhead is expected to be insignificant compared to data movement in BigMPI, it is possible to reduce the overhead of `MPIX_Type_contiguous_x` by implementing it using the internal functions of the MPI implementation, which we have prototype within MPICH already (https://github.com/jeffhammond/mpich/tree/type_contiguous_x) and begun prototyping within Open MPI.

Additional optimizations included caching graph communicators or windows associated with v-collectives and searching count vectors for repetition to reduce the number of user-defined datatypes required. The former optimization was previously implemented in BigMPI but was removed because of the challenges associated with making it thread-safe and the goal to neither require a special initialization routine for BigMPI nor intercept MPI's own initialization routine via PMPI interposition.

In general, the goal of BigMPI is to provide a straightforward implementation of large-count support using a friendly library interface. The best way to develop an optimized implementation of large-count support is within an MPI implementation, whether that be through new functions in MPI-4 or non-standard extensions to MPI provided by a particular implementation. For example, it would be straightforward, albeit a substantial amount of work, to implement the large-count operations of the BigMPI interface within MPICH.

IV. Making MPI Implementations Large-count Clean

As mentioned earlier, the MPI Forum contended that a "count, type" tuple was sufficient to describe arbitrarily large types. As of mid 2013, few codes required large count functionality, and those that did had devised workarounds. BigMPI and the re-emergence of a certain class of I/O routines finally served as the motivation needed to audit the MPICH code. To illustrate the challenges in making any MPI implementation "large count" clean, we describe the changes needed for the MPICH datatype processing engine and the ROMIO I/O library. We also share our experiences with an unfortunate OS limitation.

A. MPICH Datatloop code

The MPICH code base prior to the 3.1 release contained widespread assumptions that an int-sized type would be sufficient to contain not only the size of a datatype but also the product of a count of the number of datatypes and the size of those types. Even before MPI-3, this assumption was false: the size of a million MPI_DOUBLE types exceeds 32 bits. An obvious first step would be to promote 'int' to 'MPI_Count' wherever it was used to hold a size. Out of concern for possibly conducting 128-bit math on a 64 bit platform (a rather poorly performing situation on the LP64 machines common in 2014), we instead used MPI_Aint. The MPI_Aint type, large enough to hold a count of bytes for a memory allocation, will be sufficient to describe the file and memory use cases we envision. To find all the locations in 8600 lines of code requiring promotion, the Clang compiler warning flag `-Wshorten-64-to-32` proved invaluable for this task. The compiler option has flagged many more locations in the MPICH code that remain in need of examination.

B. ROMIO type processing

Once we made it possible for MPICH to describe arbitrarily large datatypes, we needed to update the ROMIO layer to understand these new larger datatypes. ROMIO [18] was designed to be a portable implementation of MPI-2's I/O chapter. While in modern practice it is almost always part of an MPI implementation, it is possible to build a stand-alone ROMIO library. Thus, ROMIO strives to use only MPI library routines to process datatypes, and not reach into the internal datatype processing engine of the underlying MPI implementation.

The MPI-3 standard provides the large-count aware _x variants of MPI_Type_get_size, but ROMIO, like the MPICH dataloop code, used int types for the count. Here again, we

had to audit ROMIO for instances of storing $count * size$ into an integer, an operation that would result in the compiler truncating the result upon assignment.

Even some surprising regions of ROMIO needed updating. For one example, the two-phase collective buffering optimization will split up even large requests into "cb_buffer_size" chunks. However, there is still a preliminary step where ROMIO exchanges offset-length pairs among coordinating processes. ROMIO constructs an HINDEXED type to describe these pairs. HINDEXED's "lengths" array is defined as an int type. ROMIO borrowed the BigMPI ideas and implemented an HINDEXED datatype constructor that used an MPI_Count type for its length array.

C. UNIX system calls

Finally, after updating MPICH and ROMIO to accommodate large data transfers, we are left with one last problem: the system call layer. The write system call has the following prototype:

```
ssize_t write(int fd, const void *buf, size_t count);
```

where size_t is supposed to be big enough to hold "the size of an object" [19]. However, we must remember that the rule for write is that it may write "up to count bytes". In practice, short writes to a file are not seen – until the count of bytes approaches 2^{31}. On Linux, we observed the write system call outputting at most $2^{31} - 4096$ bytes no matter how many bytes were requested, necessitating the introduction of retry logic. On Darwin and BSD, the story is even worse: if 2^{31} bytes are passed down to the read or write system call, the call will return an error. We now cap the size of a transfer to INT_MAX and issue multiple system calls until all bytes have been transfered. The lesson for implementors is clear: operating on large amounts of data has seen very little test coverage throughout the software stack.

V. Results

The primary experiment involved in this project was the mapping of large-count BigMPI functions to MPI-3 ones, which was described in §III. However, it is worthwhile to measure the overhead associated with layering BigMPI on top of MPI-3, particularly for v-collectives. Additionally, as user-defined reductions are not amenable to numerous optimizations normally found in high-performance MPI implementations, that may lead to significant performance degradation in some cases. The final version of this paper will evaluate these two issues using two modern HPC architectures: Cray XC30 and InfiniBand clusters. In order to make an apples-to-applies comparison, a count of 2^{30} will be used. This is sufficiently large that the data transfer cost should be dominant if the overheads are to be considered irrelevant. Similarly, this is sufficiently large as to observe a substantial difference between an optimized reduction and the user-defined one, should a difference exist. For debugging purposes, BigMPI allows the user to specify any large-count cutoff, not just `INT_MAX`, so these experiments require no development.

VI. Suggestions for MPI-4

1) Reductions: Whether or not one can apply a built-in reduce operation to a simple (e.g. contiguous and homogeneous) user-defined datatype is a fundamental inconsistency in the MPI standard, as accumulate functions permit this while reductions do not. Tickets 34 [20] and 338 [21] propose to reconcile reductions and accumulate by generalizing reductions to include the features of accumulate (but not the converse, as that would entail support for active-messages via RMA). Both BigMPI and the popular numerical library PETSc wish to leverage "accumulate-style behavior" in reductions, i.e. the built-in operations can work on user-defined datatypes in an element-wise basis.

Ticket 339 [22] ("User-defined op with derived datatypes yields space-inefficient reduce") is related to the problem with `MPI_IN_PLACE` with user-defined reductions. A more general interface for user-defined reduction operations that supports both in-place and pipelined reductions would be of great value to BigMPI.

While creating a large-count contiguous datatype seems like a simple thing to do, the näive implementation encounters overflow issues without explicit casting and is thus error-prone. In any case, the implementation of this feature on top of MPI requires six MPI functions, whereas the internal implementation would be almost trivial, as it would merely set the internal count on the datatype – a field that will not overflow if the implementation is count-safe. Adopting ticket 423 [23] ("add MPI_Type_contiguous_x") will reduce user difficulty when dealing with large counts. As is evidenced by BigMPI and the prototyped implementation within MPICH, the change is straightforward to implement.

When applying BigMPI's large-count strategy to the v-collectives, the (counts[], type) description has to be mapped to (newcount[], newtypes[]), and that in turn requires the w-variants. Ticket 430 [24] ("large-count v-collectives") would provide a large-count v-collective and would avoid the need for big temporary memory allocations. It also solves the problem associated with `int` displacements in `MPI_Alltoallv`, which lead to an overflow issue even if each process sends less than 2^{31} elements. For example, a parallel FFT on 12GB of C `float` will overflow because the value of the displacements for approximately one-third of the processes exceed 2^{31}.

Finally, the implementation of non-blocking collectives using point-to-point – which is the most straightforward solution in many cases – requires improved generalized requests. Ticket 457 [25] ("expose progress in generalized requests") is an older proposal to address well-known issues with generalized requests that has long been solved in MPICH but is not standardized.

Note that we do not propose to add large-count versions of all MPI communication routines, as was suggested but ultimately rejected during MPI-3 discussions. Many of the most popular MPI functions work just fine with the datatypes solution and the addition of `MPI_Type_contiguous_x` would make it almost trivial for users to realize large-count support in applications. Where we have proposed a set of new functions – large-count v-collectives – it is because the overhead of emulating this support on top of MPI-3 is $O(n_{proc})$ and the semantic mismatch is profound (e.g. large-count

`MPI_Scatterv` as `MPI_Neighborhood_alltoallw` is unnatural).

VII. Related Work

As noted in §I and §VI, there have been efforts within the MPI Forum to address count-safety issues in the MPI standard. Both MPICH and OpenMPI have made significant strides towards count-safety at the implementation level. MPICH currently passes all of the large-count tests in its own test suite, although these tests may not exercise all possible code paths. We are not aware of any other efforts to implement a high-level library on top of MPI-3 that supports large-count usage in the manner that BigMPI does.

A. OpenSHMEM

OpenSHMEM 1.0 [26] conscientiously uses `size_t` for counts and `ptrdiff_t` for offsets throughout, hence is a count-safe API. As there are numerous implementations of OpenSHMEM, it is not possible to evaluate the count-safety of all of them. However, when a count-safe API like DMAPP [27] is used, this is more likely than if the implementation is required to map from 64-bit counts to 32-bit counts internally.

B. GASNet

GASNet uses `size_t` and is thus count-safe. We have not attempted to evaluate the count-safety of GASNet implementations, as there are numerous conduits, each of which might have large-count issues due to platform-specific low-level APIs and bugs in system software.

C. GA/ARMCI

Both the Global Arrays [28] and ARMCI [29] interfaces use native integer types in both C and Fortran to represent element counts, and in the case of ARMCI Put and Get, the count is in terms of bytes, not elements. Thus, both models have the same (or worse) large-count issues as MPI-3.

VIII. Conclusions and Future Work

In a time where 64 bit systems are widespread, but C integer types remain 32 bits, describing large memory or file requests will more frequently require the special handling BigMPI provides. The exercise has also revealed several difficulties in the standard. We have described fundamental issues with nonblocking collective operations (reductions and both vector and neighborhood collectives) that cannot be overcome using MPI-3 features. The MPI Forum issued a challenge to consumers of MPI: "prove to us that derived datatypes are insufficient". We believe this burden has been met and suggest the following features be added to MPI in order to make holistic large-count support a reality.

We intend to drive the aforementioned tickets within the MPI Forum in order to make complete large-count possible and efficient. These features will be prototyped within MPICH and exploited by BigMPI to provide that they are both necessary and sufficient. A second area where ongoing development work is required is large-count tests that can be used to validate the count-safety of MPI-3 implementations. Finally, we will attempt to write a set of large-count tests for OpenSHMEM and GASNet. The large-count tests of OpenSHMEM also serve as large-count tests for MPI-3, by virtue of OSHMPI [30].

Acknowledgments

This research used computing resources at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] MPI Forum, "MPI: A message-passing interface standard," University of Tennessee, Knoxville, Tech. Rep. UT-CS-94-230, 1994.

[2] ——, "MPI-2: Extensions to the message-passing interface," University of Tennessee, Knoxville, Tech. Rep., 1996.

[3] ——, "MPI: A message-passing interface standard. Version 2.2." Sep. 2009.

[4] ——, "MPI: A message-passing interface standard. Version 3.0." Nov. 2012.

[5] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "Mpi on millions of cores," *Parallel Processing Letters*, vol. 21, no. 01, pp. 45–60, 2011.

[6] F. Tillier, "Support for large counts using derived datatypes," 2011. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/265

[7] J. Squyres, "New things in MPI-3: MPI_Count," 2011. [Online]. Available: http://blogs.cisco.com/performance/new-things-in-mpi-3-mpi_count/

[8] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science–VECPAR 2010*. Springer, 2011, pp. 1–25.

[9] J. Jenkins, J. Dinan, P. Balaji, N. F. Samatova, and R. Thakur, "Enabling fast, noncontiguous gpu data movement in hybrid mpi+ gpu environments," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 468–476.

[10] D. Gregor and M. Troyer, "Boost.MPI Library http://www.boost.org," 2014, [accessed 2014-09-04]. [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/mpi.html

[11] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *Proceedings of SC2003: High Performance Networking and Computing*, ser. SC '03. Phoenix, AZ: IEEE Computer Society Press, November 2003, pp. 39–. [Online]. Available: http://www.sc-conference.org/sc2003/paperpdfs/pap258.pdf

[12] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen, "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus," in *SC '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Denver, USA, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=582086

[13] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4, Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct700301q

[14] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers," in *SC '11: Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, 2011.

[15] A. Schfer and D. Fey, "LibGeoDecomp: A Grid-Enabled Library for Geometric Decomposition Codes," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer, 2008, pp. 285–294.

[16] R. Latham, W. Gropp, R. Ross, and R. Thakur, "Extending the MPI-2 Generalized Request Interface," *Lecture Notes in Computer Science*, pp. 223–232, October 2007, (EuroPVM/MPI 2007). [Online]. Available: http://www.springerlink.com/content/y332095819261422

[17] T. Höfler, "Add a callback function if a request completes," 2008. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/26

[18] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, May 1999, pp. 23–32. [Online]. Available: http://www.mcs.anl.gov/~thakur/papers/mpio-impl.ps

[19] IEEE, *2004 (ISO/IEC) [IEEE/ANSI Std 1003.1, 2004 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. New York, NY USA: IEEE, 2004.

[20] G. Bosilca, "Extend predefined mpi_op's to user defined datatypes composed of a single, predefined type," 2008. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/34

[21] D. Goodell and J. Dinan, "MPI_Accumulate-style behavior for predefined reduction operations," 2012. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/338

[22] J. Dinan, "User-defined op with derived datatypes yields space-inefficient reduce," 2012. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/339

[23] J. Hammond, "Add MPI_Type_contiguous_x," 2014. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/423

[24] ——, "Large-count v-collectives," 2014. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/430

[25] J. Träff and T. Höfler, "Exposing progress in generalized requests," 2007. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/457

[26] J. A. Kuehn, B. Chapman, A. R. Curtis, R. Mauricio, S. Pophale, R. Nanjegowda, A. Banerjee, K. Feind, S. W. Poole, and L. Smith, "OpenSHMEM application programming interface, v1.0 final," Oak Ridge National Laboratory (ORNL), Tech. Rep., 2012.

[27] "Using the GNI and DMAPP APIs," http://docs.cray.com/books/S-2446-5002/S-2446-5002.pdf, Cray, Tech. Rep. S-2446-5002, 2013. [Online]. Available: http://docs.cray.com/books/S-2446-5002/S-2446-5002.pdf

[28] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: a portable "shared-memory" programming model for distributed memory computers," in *Proc. ACM/IEEE Conference Supercomputing (SC '94)*, 1994, pp. 340–349.

[29] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," *Lecture Notes in Computer Science*, vol. 1586, pp. 533–546, 1999. [Online]. Available: citeseer.ist.psu.edu/nieplocha99armci.html

[30] J. R. Hammond, S. Ghosh, and B. M. Chapman, "Implementing OpenSHMEM using MPI-3 one-sided communication," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*. Springer, 2014, pp. 44–58.