To INT_MAX...and beyond!

Exploring large-count support in MPI

Jeff Hammond
Extreme Scalability Group
Parallel Computing Lab
Intel Corporation

Andreas Schäfer
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Rob Latham
Math and Computer Science Division
Argonne National Laboratory

Legal Disclaimer & Optimization Notice

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

The large-count problem

- A count in MPI refers the number of elements of the specified type, e.g. Send(buf,count,FLOAT,…).
- In_ILn_LPn_P to refer to the sizes of the C types *int*, *long*, and *void**, respectively.
- Circa MPI-1, most systems were ILP32 or at least had less than 2 GiB of memory per node.
- On IL32P64 and I32LP64 systems, one can allocate more than 2 GiB and thus potentially have buffers with more elements than can be represented as an 32-bit integer (such as C *int*).
- Let's not talk about Fortran…

Example 1

```
size_t n = INT_MAX+(size_t)42;
char * stuff = malloc(n);
if (myrank==0) memset(stuff, 7, n);
MPI_Bcast(stuff, n, MPI_CHAR, 0, mycomm);
```

Q: Assuming your compiler let's you pass a size_t into an int, what will happen?

A: You will not get what you want.

Example 2

```
int n = INT_MAX/2;
double * stuff = malloc(n);
if (myrank==0) memset_double(stuff, 42.0, n);
MPI_Bcast(stuff, n, MPI_DOUBLE, 0, mycomm);
```

The count does not overflow, but if the implementation converts all communication to bytes internally, then there will be an internal overflow.

This is the *other* large-count problem.

What MPI-3 offers

- MPI_Foo_x(MPI_Count count) provides a large-count equivalent to existing MPI_Foo(int count):
    - MPI_Get_elements_x
    - MPI_Type_size_x
    - MPI_Type_get_extent_x
    - MPI_Type_get_true_extent_x
    - MPI_Status_set_elements_x
- The MPI Forum decided that these routines, in conjunction with intelligent use of MPI datatypes, was sufficient for large-count support.
- Our library doesn't use these functions…

What is this paper about?

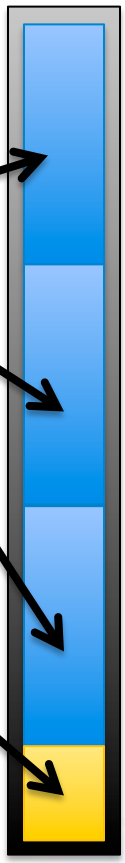
- *Evaluate* the Forum's assertion that a handful of utility routines and user-defined datatypes are sufficient for large-count support.
- *Implement* a high-level library on top of MPI (called BigMPI) that makes it possible to enable large-count support in applications with minimal source changes.
- Otherwise *demonstrate* how large-count support can be achieved with MPI-3 features.
- *Investigate* count-safety of MPI *implementations*.
- Suggest improvements to the *MPI standard* (MPI-4) related to large-counts.

The BigMPI pattern

```
int BigMPI_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;
        BigMPI_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;
}
```

```
int BigMPI_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;
}
```



BigMPI Design

- Only focused on IL32P64 and I32LP64 systems with less than 2^{64} bytes of memory.
- Focus on large-count buffers of built-in datatypes: BigMPI_Type_contiguous_x is used throughout to turn (large_count,built_in_type) into (1,large_count_type).
- Assume that all library overhead is negligible compared to moving >2 GiB of data.
- All-or-nothing w.r.t. large-counts; no specialization if only some counts are large.
- Avoid things that require init/finalize…

Things that are easy

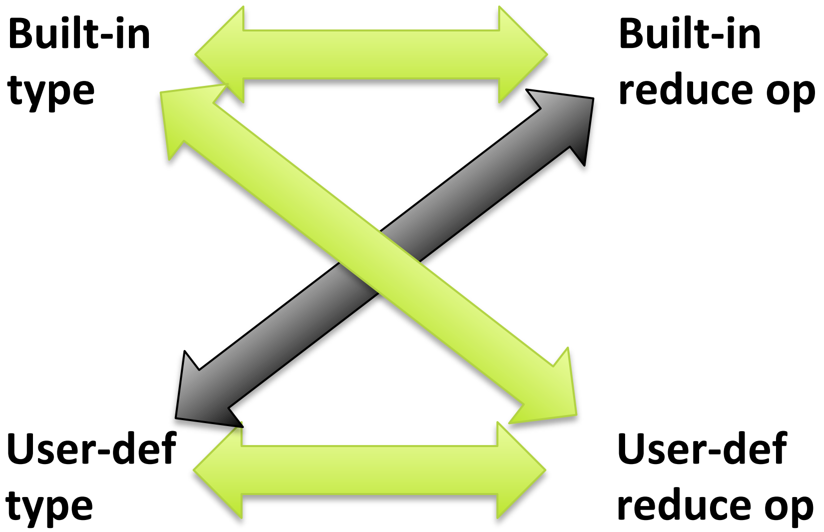
- Point-to-point (two-sided and one-sided) are trivial using the aforementioned template.
- Scalar-argument data-moving collectives – Bcast, Scatter, Gather, Allgather, Alltoall – are similarly trivial.

These are the most common MPI operations and thus for some applications, the large-count problem is minor.

`BigMPI_Type_contiguous_x` wasn't trivial to get right the first time…

Reductions are a problem

- Blocking collectives can be broken into **multiple operations** with normal counts.
- Blocking operations can use a large-count type and associated **user-defined reduce operation**.
- Implementations do not optimize user-defined reductions…



This situation is only true for reductions! RMA allows the black arrow as long as the user-def type is homogeneous.

In-place reductions

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

It is impossible to use MPI_IN_PLACE within a user-defined reduction!

There are other issues with this function signature…

Nonblocking reductions

- A reduction of e.g. 2^{32} doubles is something one would like to overlap with computation…
- Can't break into multiple messages and return a single request. Defining special request object and test/wait operations within BigMPI is gross.
- No way to free the large-count type or reduce op when the reduction is finished due to lack of callbacks in request completion.
- MPI generalized requests are terrible; MPICH generalized requests are good but non-portable.

Large-count in user-defined reductions

```
int BigMPI_Decode_contiguous_x(MPI_Datatype intype,
    MPI_Count * count, MPI_Datatype * basetype);
```

Repeated calls to `MPI_Type_get_envelope` and `MPI_Type_get_contents` are required to determine the original large-count associated with this type (and only this type).

81 lines of tedious code for the trivial case of a contiguous large-count type!

Vector-argument collectives

- Scatterv, Gatherv, Allgatherv, Alltoallv take a vector of counts and a *single datatype*.
- Vector of (count,type) turns into vector of (1,large_count_type), which means a *vector of types*.
- Only Alltoallw supports a vector of types…

```
int MPI_Gatherv(const void * sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void * recvbuf, const int recvcounts[],
                const MPI_Aint rdispls[], MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

Vector-argument implementation

1. Implementation in terms of two-sided is easy if we do a naïve implementation (not bad), but requires Comm_dup for THREAD_MULTIPLE at any target (who could post wildcard Recv).
2. Can implement in terms of one-sided in the same manner if we create and destroy a window on-the-fly.
3. Try to use a more general collective: Alltoallw.

BigMPI aspires to implement all three strategies, but 2 is not finished and 1 lacks MULTIPLE support.

Alltoallw to the rescue???

- Scatterv/Gatherv -> Alltoallw is gross.
- …*to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes*. [MPI-3 page 173]
- We have a large-count problem with the displacements even if none of the original counts overflows (just their sum has to)!

```
int MPI_Alltoallw(const void* sbuf, const int scounts[],
    const int sdispls[], const MPI_Datatype stypes[],
    void* rbuf, const int rcounts[],
    const int rdispls[], const MPI_Datatype rtypes[],
    MPI_Comm comm)
```

Neighborhood_alltoallw to the rescue!

- New in MPI-3; requires topological communicator.
- Displacement problem solved by MPI_Aint.
- Large-count Scatterv requires:
  - A distributed-graph communicator unique for each (root,comm).
  - Vector(s) of all ones (the counts).
  - Vector(s) of displacements, which are all zero in some cases.
  - Vector(s) of large-count types.

```
int MPI_Neighborhood_alltoallw(const void* sbuf, const int scounts[],
    const MPI_Aint sdispls[], const MPI_Datatype stypes[],
    void* rbuf, const int rcounts[],
    const MPI_Aint rdispls[], const MPI_Datatype rtypes[],
    MPI_Comm comm)
```

Associated utility functions

```
int BigMPI_Create_graph_comm(MPI_Comm mycomm,
                             int root, MPI_Comm * dist_graph_comm);

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[]);
```

Both of these functions are O(nproc) but negligible compared to Alltoallw.

The nonblocking problem again

- Point-to-point approach entails a vector of requests…
- MPI-3 lacks nonblocking RMA epochs, but window create/free cannot be nonblocking. Nonblocking RMA epochs proposed in SC14 paper…
- Nowhere to deallocate temporary vectors in the case of nonblocking alltoallw.
- Even if deallocation done during generalized request, cannot free comm there. Speculatively creating graph comms for each user comm is evil.

Addressing implementation issues

- Prior to 3.1, MPICH used C int internally in dataloop code and ROMIO. Rob and Clang did the heavy lifting required to eliminate all places where truncation/overflow could occur (at least according to test suite…).
- Linux, BSD and Darwin do one of two evil things with ssize_t write(**int** fd, **const void** *buf, size_t count); that will affect both I/O and sockets code. MPI implementations must chunk calls to this operation in spite of its apparent large-count safety.

MPI-4: Suggestion #1

Reconcile reductions and accumulate by generalizing reductions to include the features of accumulate.

This is a glaring asymmetry in the MPI standard that was noticed independent of large-count support.

1. G. Bosilca, "Extend predefined MPI Op's to user defined datatypes composed of a single, predefined type," 2008. https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/34
2. D. Goodell and J. Dinan, "MPI Accumulate-style behavior for predefined reduction operations," 2012. https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/338

MPI-4: Suggestion #2

Enhance user-defined reductions to support MPI_IN_PLACE as well as pipelining. Maybe…

There needs to be a new proposal here, because the existing ticket isn't likely to move forward.

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

MPI-4: Suggestion #3

Add large-count datatype support explicitly.

Only the contiguous case has been proposed, but it leads to an obvious consistency if not done for all cases.

MPI_Type_get_envelope_x and MPI_Type_get_contents_x are required as well as all of the large-count combiners.

1. J. Hammond, "Add MPI_Type_contiguous_x," 2014. https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/423

MPI-4: Suggestion #4

Support large-count vector collectives explicitly.

Fixes the glaring problem with unsafe displacements where the sum but not the individual counts exceed INT_MAX.

Makes the standard longer but the implementations are straightforward.

1. J. Hammond, “Large-count v-collectives,” 2014. https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/430

MPI-4: Suggestion #5

Improve the generalized request progress model.

This would solve many of the issues with nonblocking operations in BigMPI.

There are a myriad of other good uses of MPICH-style generalized requests…

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

Related Work

Model	Count	Displacement	Implementations
OpenSHMEM	size_t	ptrdiff_t	unknown; conduit-dependent
GASNet	size_t	size_t	unknown; conduit-dependent
GA/ARMCI	int	int	N/A

Good system software appears to have transitioned from int to size_t, but the MPI Forum stubbornly insists upon backwards ABI-compatibility.

https://github.com/jeffhammond/BigMPI

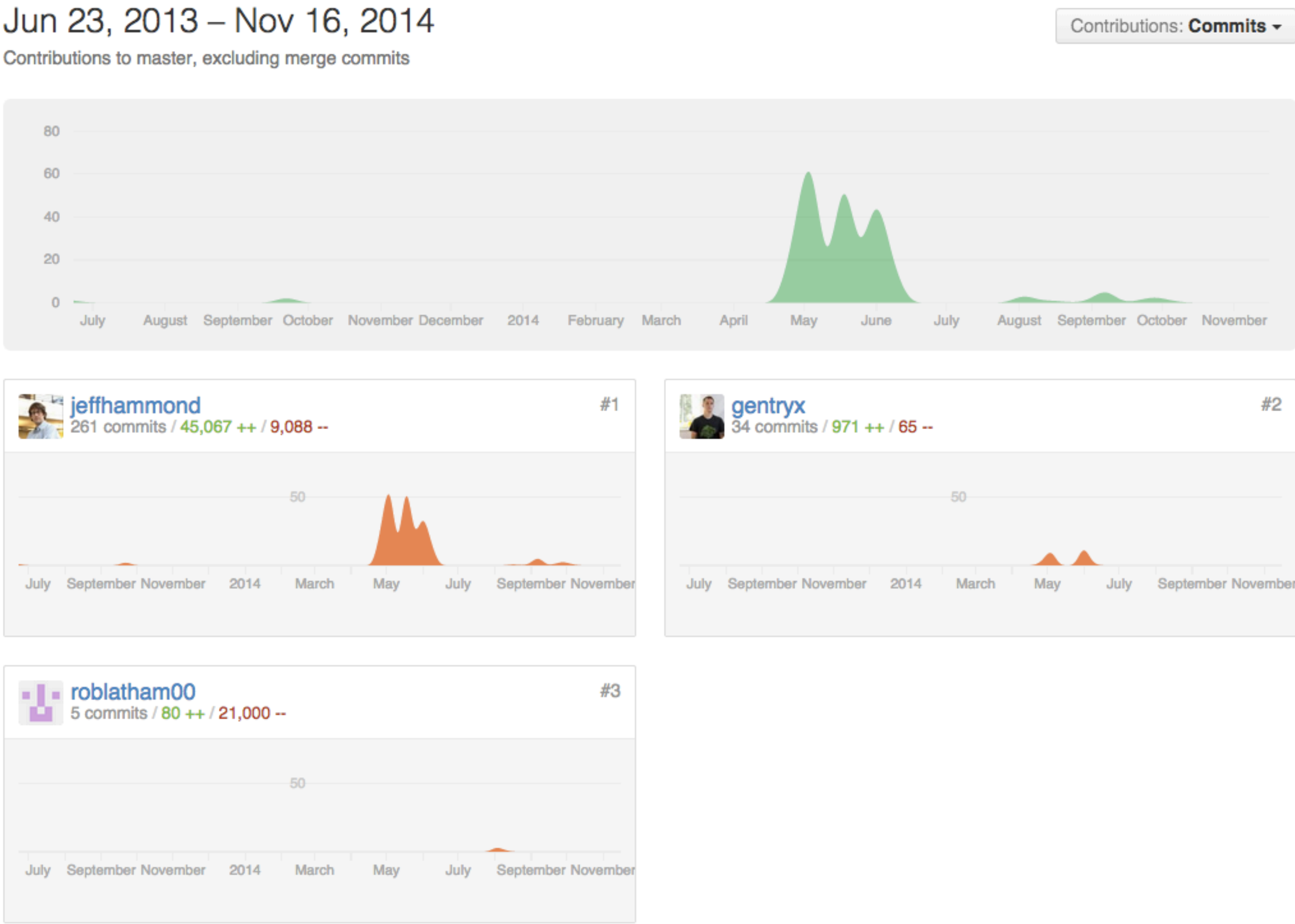
MIT License

This is research software, not an Intel product.

intel®

Application motivation

- **Why *shouldn't* users be able to straightforwardly do MPI stuff with any buffer they can allocate?**
- HDF [http://blogs.cisco.com/performance/new-things-in-mpi-3-mpi_count/]
- Quantum chemists in Europe asked me for help with this, which is what inspired the idea of BigMPI; some of their issues are Fortran-related…
- Andreas – a bona fide application developer – blog-shamed me into making BigMPI happen.
- I/O aggregation (e.g. ALCF GLEAN project) can easily end up with large-count subcomm Gatherv as memory per node increases.



Intel Labs – Systems and Software Research