

Dynamically-Sized Messages in MPI-3

Douglas Gregor, Torsten Hoefler, and Andrew Lumsdaine
{dgregor,htor,lums}@osl.iu.edu

January 11, 2008

1 Introduction

MPI provides support for sending messages of any size and any data type that can be described as a primitive type or a datatype derived from primitive types. For types that cannot be transmitted in this way, such as linked lists or other objects that must be serialized, MPI provides a mechanism to “pack” (serialize) data into a buffer that can be transmitted via MPI, then “unpacked” (de-serialized) at the receiver’s end. Bindings for many object-oriented and generic languages have used these mechanisms to serialize objects for transmission via MPI, offering improved support for MPI in high-level languages, including C++ [3, 6], Java [1], Python [7], and C# [2].

However, while MPI provides good support for serialization and sending serialized data, it does not provide adequate support for receiving serialized data. The problem is that, in general, the receiver cannot know the length of the serialized data before it posts the receive. Obvious approaches to solving this problem—e.g., probing for the message before receiving it—work well within a single-thread environment but fail within a multi-threaded environment; see Section 2 for more information.

We propose to extend MPI to support receiving messages of arbitrary length. In this scheme, the MPI implementation will allocate a buffer of the appropriate size and return that buffer to the user containing the received data. With this extension, it becomes significantly easier for users and library developers to transmit data of arbitrary size in a thread-safe manner. Additionally, since the thread-safe workarounds for this problem (described in section 2) involve the transmission of multiple messages (one with a synchronous send), these extensions may improve application performance in languages that require object serialization, such as C#, Java, and Python.

2 Workarounds: Receiving Data of Unknown Length

The first, obvious approach to receiving data of unknown length involves the use of `MPI_Probe`. For example:

```
MPI_Status status;  
MPI_Probe(source, tag, comm, &status);
```

```

int count;
MPI_Get_count(&status, MPI_BYTE, &count);
char* buffer = (char*)malloc(count);
MPI_Recv(buffer, count, MPI_BYTE, status.MPI_SOURCE, status.MPI_TAG, comm, &status);

```

In a multi-threaded MPI program, it is possible that two threads could attempt to execute this code concurrently. In this case, both threads could easily probe the same message, then post receives for that message. Only one of the receives will succeed in matching that message; the other receive will block until another message with the same source and tag is received. However, it is unlikely that this next message will have precisely the same length, causing an MPI failure. The matching problem could be avoided with several locks that essentially make the probe and the receive look like an atomic operation. However, this requires the locking of all MPI calls (similar to the `MPI_THREAD_FUNNELED` model) and incurs significant overheads similar to the overheads mentioned in [4].

Another popular approach to receiving data of unknown length is to split the message into two separate messages: a fixed-length message, containing message size information, and a variable-length message, containing the message data itself. Unlike the probe-based approach above, this approach changes the sender protocol. The receiver, in this code, is something like:

```

MPI_Status status;
int count;
MPI_Recv(&count, 1, MPI_INT, source, tag, comm, &status);
char* buffer = (char*)malloc(count);
MPI_Recv(buffer, count, MPI_BYTE, status.MPI_SOURCE, status.MPI_TAG, comm, &status);

```

Here, we assume that the variable-length message always follows the fixed-length message using the same tag. Unfortunately, this approach introduces race conditions in several multi-threaded scenarios. For example, since two threads could be receiving concurrently, the first `MPI_Recv` of one thread could end up receiving the variable-length message intended for another thread, since the messages using the same tag. Even if the variable-length messages used a different tag from the tag used for the fixed-length message (say, $tag + 100$), the race condition remains: two threads could receive fixed-length messages with the same tag, but end up incorrectly matching the variable-length messages (since they use the same tag). Moreover, the variable-length messages could still be matched by a receive using `MPI_ANY_TAG`, and—if it is a library implementing this protocol for sending serialized data—the tag value $tag + 100$ might conflict with the user’s tags.

There are two parts to solving the problems with selecting tags for the variable-length message. First, to avoid collisions with user tags, and to keep receives using `MPI_ANY_TAG` from ever matching the variable-length messages, a library can must create a second communicator (which we call the “shadow” communicator) for each existing communicator. The shadow communicator is dedicated to the transmission of the variable-length messages. Second, to eliminate the race condition associate with two threads receiving the wrong variable-length messages, a library can allocate a “unique” tag for each variable-length message, and send that tag as part of the fixed-length header message. In this case, the sender and receiver are as follows:

Sender	Receiver
<pre> struct header { int tag; int count; }; char* buffer; struct header hdr; hdr.tag = allocate_unique_tag(); serialize_object(obj, &buffer, &hdr.count); MPI_Send(&hdr, 1, header_type, tag, dest, comm); MPI_Ssend(buffer, hdr.count, MPI_BYTE, hdr.tag, dest, shadow_comm); free_tag(hdr.tag); </pre>	<pre> struct header { int tag; int count; }; MPI_Status status; struct header hdr; MPI_Recv(&hdr, 1, header_type, source, tag, comm, &status); char* buffer = (char*)malloc(hdr.count); MPI_Recv(buffer, hdr.count, MPI_BYTE, status.MPI_SOURCE, hdr.tag, shadow_comm, MPI_STATUS_IGNORE); // de-serialize from buffer </pre>

Much of this implementation is straight-forward. The receiver’s side merely receives the fixed-size header, and uses the information within that header to receive the variable-length message. The sender uses a `allocate_unique_tag` function that provides unique tag values for the variable-length messages; this function must be re-entrant. Additionally, since there is a finite number of tags for the variable-length message (which could be as few as 32k tags), the sender must re-use those “unique” tag values. To facilitate reuse, the `free_tag` function returns that tag to the pool of unique tags. However, to do so safely requires that the variable-length message be sent with a synchronous send¹, so that we can ensure that the receiver has already matched the message. Using a normal-mode send here could re-introduce a race condition at the receiver’s end, where two variable-length messages come in with the same “unique” tag.

The challenges we are describing are not unique; nor are our solutions. In fact, every MPI implementation must solve exactly the same issues to provide MPI semantics on top of network hardware, and typically do so with better (and more complicated) solutions that involve multiple protocols, such as using shared received queues for smaller messages and rendezvous protocols for larger messages. Thus, while this problem has been entirely solved in the MPI implementation, applications and libraries built on top of MPI cannot take advantage of it because the MPI interface does not expose this functionality. Instead, users must re-invent these solutions for each application or library, studiously avoiding the various pitfalls we have described here. In the end, their solution is likely to incur greater overhead than any solution that comes from the MPI implementation, because (for example) the actual message passing will include the overhead of two protocols: the one employed by the user (e.g., one normal-mode send and one synchronous-mode send) and the one employed by the MPI implementation to transmit these messages. Thus, addressing the problem of receiving

¹With a sufficient number of tags, one could cycle through the tags, wrapping around to zero when the maximum tag value is reached. This would eliminate the need for the synchronous send, but it could fail if a large number of messages are sent this way.

data of unknown length will simplify the implementation of MPI libraries and applications and is likely to improve their performance as well.

3 Proposed Extensions

In this section, we describe our proposed extensions to receive messages of arbitrary length. For brevity and readability, we have adopted an informal example-driven style not suitable for the MPI standard document itself. As the details of this proposal settle, we will provide precise wording for the MPI document.

3.1 Receiving Messages of Arbitrary Length

We propose to add new variants of the MPI receive functions that allow one to receive an arbitrary amount of data.

```
int MPI_Recv(MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
             int temporary, void *bufptr, MPI_Status *status)
```

IN datatype datatype of each receive buffer element (handle)

IN source rank of source (integer)

IN tag message tag (integer)

IN comm communicator (handle)

IN temporary **true** if the user will return the associated buffer “quickly”, **false** otherwise (see Section 3.3)

bufptr address of a pointer, which will receive the address of the receive buffer (choice)

OUT status status object (Status)

The MPI implementation will update the pointer whose address is passed as **bufptr** to point to a buffer that contains the received data (which may be of any size). The user can then query the **MPI_Status** object to determine how much data was actually received.

Example: the following example illustrates how one could use **MPI_Recv** to receive a string of unknown length.

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        char* mystring = "Hello, World!";
        MPI_Send(mystring, strlen(mystring)+1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        char* mystring;
        MPI_Status status;
        MPI_Recv(MPI_CHAR, 0, 0, MPI_COMM_WORLD, true, &mystring, &status);
        printf("Received string \"%s\" from rank 0", mystring);
        MPI_Free_buf(MPI_COMM_WORLD, mystring);
    }
}
```

```

    }
    MPI_Finalize();
}

```

On the receiver’s end, we pass the address of the pointer `mystring` as the `bufptr` argument of `MPI_Recv`. The MPI implementation receives the message into a newly-allocated buffer, then puts a pointer to that buffer into `mystring`. The program prints the string and then frees the receive buffer with a new function, `MPI_Free_buf`.

Note regarding the Fortran bindings: The implementation of our proposal can not be supported by the Fortran 77 bindings because Fortran 77 does not support the notion of pointers. The implementation of Fortran 90 bindings is problematic because Fortran 90 pointers have a specific data-layout attached, which leads to excessive interface definitions (for every datatype, cf. [8]).

Rationale: The `bufptr` argument is, semantically, a pointer to a pointer. However, using `void**` as the type of `bufptr` would force users to add additional type-cases, e.g., from `char**` to `void**` in the example above.

```

int MPI_Free_buf(MPI_Comm comm, void* buffer);

```

`MPI_Free_buf` frees a buffer allocated by a receive with one of the arbitrary-length receive operations. If the user has attached an allocator to the communicator (see Section 3.2), the corresponding deallocator will be invoked to free the buffer.

```

int MPI_Irecv(MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
             int temporary, void *bufptr, MPI_Request *request)

```

IN datatype datatype of each receive buffer element (handle)

IN source rank of source (integer)

IN tag message tag (integer)

IN comm communicator (handle)

IN temporary true if the user will return the associated buffer “quickly”, **false** otherwise (see Section 3.3)

bufptr address of a pointer, which will receive the address of the receive buffer (choice)

OUT request communication request (handle)

Initiates a non-blocking receive of arbitrary length.

```

int MPI_Recv_init(MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
                 int temporary, void *bufptr, MPI_Request *request)

```

IN datatype datatype of each receive buffer element (handle)

IN source rank of source (integer)

IN tag message tag (integer)

IN comm communicator (handle)

IN temporary true if the user will return the associated buffer “quickly”, **false** otherwise (see Section 3.3)

bufptr address of a pointer, which will receive the address of the receive buffer (choice)

OUT request communication request (handle)

Creates a persistent communication request for an arbitrary-length receive operation.

3.2 User-controlled Buffer Allocation

When receiving with messages of arbitrary length, the MPI implementation will allocate memory for the receive buffers as necessary. However, this behavior takes away the user's ability to control memory allocation, which is particularly important when MPI is being used from languages that maintain a separate garbage-collected heap (e.g., C#, Java). We propose a function `MPI_Attach_allocator` that allows one to attach a user-defined "allocator". The allocator is attached to a specific communicator, and will be invoked to provide MPI with a buffer used by arbitrary-length receives. Users can employ this function to introduce allocators that use special heaps for buffers.

An allocator function will have the following type:

```
typedef int MPI_User_allocator(MPI_Comm comm, int count, MPI_Datatype type,  
                               void* extra_state, void** bufferptr);
```

The `extra_state` argument is supplied by the user in `MPI_Attach_allocator`. The `ptr` argument is a pointer to the buffer pointer. Using this interface, one could implement an `MPI_User_allocator` in C# that not only allocates memory from the garbage-collected heap, but does so in a way that creates arrays of the appropriate type.

```
unsafe int ArrayAllocator(MPI_Comm comm, int count, MPI_Datatype type,  
                        IntPtr extra_state, out IntPtr bufferptr)  
{  
    Array array = null;  
    if (type == MPI_BYTE) array = new byte[count];  
    else if (type == MPI_INT) array = new int[count];  
    else if (type == MPI_FLOAT) array = new float[count];  
    else if (type == MPI_DOUBLE) array = new double[count];  
    else if ...  
  
    SaveArrayHandle(array, GCHandle.Alloc(array, GCHandleType.Pinned));  
    bufferptr = Marshal.UnsafeAddrOfPinnedArrayElement(array, 0);  
    return MPI_SUCCESS;  
}
```

This allocator would, for example, allow one to receive an array of integers whose length is unknown. The array itself would be allocated from the garbage-collected heap with the appropriate datatype, into which the MPI implementation would receive the message. Without a properly-typed allocation from the garbage-collected heap, a C# program (or, more likely, library) would need to de-serialize or copy the data from the memory that MPI allocates into garbage-collected memory, adding unnecessary overhead.

```
typedef int MPI_User_deallocator(MPI_Comm comm, void* extra_state, void* buffer);
```

```
int MPI_Attach_allocator(MPI_Comm comm, MPI_User_allocator* allocator,  
                        MPI_User_deallocator* deallocator, void* extra_state);
```

```
int MPI_Detach_allocator(MPI_Comm comm);
```

The `MPI_Attach_allocator` function attaches a user-defined allocator (such as the `ArrayAllocator` above) to a particular communicator. This allocator will be used to allocate buffers returned via arbitrary-length receives on that communicator. If no allocator is attached to a communicator, it is unspecified how the MPI implementation will allocate memory for these buffers.

An allocator attached with `MPI_Attach_allocator` can later be detached from that communicator with `MPI_Detach_allocator`. `MPI_Detach_allocator` can only be called when there are no pending arbitrary-length receives on that communicator and there are no allocated buffers for that communicator that have not been returned via `MPI_Free_buf`.

Rationale: allocators are associated with specific communicators to meet the requirements of applications using MPI from multiple languages. Components written in different languages will likely use different communicators, so each can attach an appropriate allocator (e.g., C# components might use an allocator that employs the garbage-collected heap, while C components would use the default allocator) to those communicators. The `extra_state` argument allows one to associate additional state with each allocation, for example to provide a specific memory pool to be used with the allocator.

3.3 Temporary Buffers and the Zero-Copy Optimization

There are two major use cases for receiving data of arbitrary size. The first involves receiving arrays of data whose size cannot be known beforehand, e.g., because a process has produced a data-dependent amount of data that needs to be received by another process. In this case, the receiver will presumably store the array for a long time before deallocating it with `MPI_Free_buf`. The second case involves serialized representations of objects. Here, the sender serializes an object into a series of bytes and sends those bytes over the wire. The receiver receives those bytes (it does not know how many bytes will be transferred) and de-serializes the data into a new object. In this case, the allocated buffer will be read once by the serialization routine and deallocated immediately.

The `temporary` argument to the arbitrary-length receive operations states whether the buffer will only be used temporarily (e.g., for de-serialization) or not. When true, it indicates to the MPI implementation that the user will return the allocated buffer “quickly” and will not modify the contents of the buffer. Implementations can provide better optimizations for this case. For example, the implementation can return a pointer to memory that has already received the message data via DMA or a pointer inside of a MPI implementation-managed ring buffer, rather than copying the data into a user buffer, eliminating the copy overhead and the overhead of memory allocation. If the message is being transferred via shared memory, implementation can map the sender’s pointer into shared memory and return the corresponding pointer to the receiver.

Since the arbitrary-length receive operations return pointers on the receiver side, and (barring user-defined allocators) those pointers are under the control of the MPI implementation, these extensions permit zero-copy point-to-point messaging. The `temporary` parameter helps users provide additional information about the usage of the buffer, so that MPI implementations can use more focused optimizations.

A Alternative interface

As an alternative to the addition of new receive functions `MPI_Recv`, `MPI_Irecv`, and `MPI_Recv_init`, we could add a new named constant, `MPI_ANY_SIZE`, that can be used as the `count` argument to the various point-to-point receive operations in MPI. When `count` is `MPI_ANY_SIZE`, the `buf` pointer argument, which refers to the receive buffer, takes on a different meaning: instead of pointing to the memory that will store the received data, `buf` will point to a single pointer. The MPI implementation will update that pointer to point to a buffer that contains the received data (which may be of any size). The user can then query the `MPI_Status` object to determine how much data was actually received (as we did with the “v” variants of the MPI receives).

The following example illustrates how one could use `MPI_ANY_SIZE` to receive a string of unknown length.

```
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        char* mystring = "Hello, World!";
        MPI_Send(mystring, strlen(mystring)+1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        char* mystring;
        MPI_Status status;
        MPI_Recv(&mystring, MPI_ANY_SIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
        printf("Received string \"%s\" from rank 0", mystring);
        MPI_Free_buf(MPI_COMM_WORLD, mystring);
    }
    MPI_Finalize();
}
```

Only the receiver portion of this program is interesting: instead of passing a receive buffer to `MPI_Recv`, we pass the address of the pointer `mystring` and use `MPI_ANY_SIZE` for the count. The MPI implementation receives the message into a newly-allocated buffer, then puts a pointer to that buffer into `mystring`. The program prints the string and then frees the receive buffer with a new function, `MPI_Free_buf`.

B MPI_ANY_SIZE in Collective Operations

While this proposal focuses on point-to-point operations with `MPI_ANY_SIZE`, the same principles and motivations apply to collective operations. All of the collective operations require *a priori* knowledge of the number of data elements that will participate in the collective operation, but this information is not always available, particularly in cases where the data consists of serialized objects. For example, one might want to broadcast the serialized representation of an object via `MPI_Bcast`, but the length of the serialized representation cannot be known in advance.

At this time, we do not propose support for `MPI_ANY_SIZE` in collective operations. Instead, we will outline some of the approaches we have considered for collectives on arbitrarily-sized data and the challenges we foresee with these approaches. We group the communicators into several categories, depending on how well the collective itself and its MPI interface can be adapted to work with arbitrarily-sized data.

Algorithmically, the broadcast, scatter, gather, all-to-all, and all-gather collectives work well with `MPI_ANY_SIZE`. In each of these cases, the collective using `MPI_ANY_SIZE` is semantically equivalent to calls to two existing collectives: the first collective will distribute size information, after which buffers can be allocated, and the second collective (usually one of the “v” variants such as `MPI_Scatterv`) will transfer the actual data. For example, one can broadcast a string of arbitrary length with the following code:

```
char* mystring;
int len;
if (rank == root) {
    /* read mystring from somewhere */
    len = strlen(mystring);
}
MPI_Bcast(&len, 1, MPI_INT, root, MPI_COMM_WORLD);

if (rank != root) mystring = (char *)malloc(len+1);
MPI_Bcast(mystring, len+1, MPI_CHAR, root, MPI_COMM_WORLD);
```

With `MPI_ANY_SIZE`, the root process will provide the size and the data, while the non-root processes will receive the string into the pointer `mystring`:

```
char* mystring;
if (rank == root) {
    /* read mystring from somewhere */
    MPI_Bcast(mystring, strlen(mystring)+1, MPI_CHAR, root, MPI_COMM_WORLD);
} else {
    MPI_Bcast(&mystring, MPI_ANY_SIZE, MPI_CHAR, root, MPI_COMM_WORLD);
}
```

Unfortunately, this use of `MPI_ANY_SIZE` has a problem in the general case: for non-root processes, there is no way to determine how many bytes were received by the broadcast operation. With point-to-point `MPI_ANY_SIZE` receives, one can query the resulting `MPI_Status` object, but collectives have no status, and some collectives (such as a scatter) will need to return multiple count values.² We have considered several solutions to the problem:

- *Non-blocking collectives*: With non-blocking collectives [5], completion of a collective operation produces an `MPI_Status` object that will contain information about the number of elements transferred. Thus, the non-blocking collective interface provides a suitable interface for `MPI_ANY_SIZE` with some collectives, including broadcast and scatter. However, this approach does not solve the problem for collectives that need to return multiple sizes. For example, a gather operation (where the root will use

²In these cases, `MPI_ANY_SIZE` might be expressible with the “v” variants of the algorithm, turning the received counts and displacements into read/write arrays.

MPI_ANY_SIZE) would need to provide the user with the `count` argument provided by each of the non-root processes. The `MPI_Status` object itself isn't useful for this retrieving these counts, because it provides only a single count through `MPI_Get_count`.

- *New collectives*: We could introduce addition collective operations with interfaces tailored for `MPI_ANY_SIZE`. Although this is the most direct route, it requires doubling the number of collectives in MPI, drastically expanding the interface.

The reduction operations, including reduce, scan, exclusive scan, and all-reduce, will require a very different interface to deal with `MPI_ANY_SIZE`. In fact, reductions on arbitrary-sized data often have a different character from reductions that work on data of fixed size. MPI's reductions are vector reductions, meant to be applied pairwise to the data elements in two arrays or equal length. However, reductions with arbitrarily-sized data might be operating on serialized objects or with arrays of differing length that are being combined in ways that can't be expressed as an elementwise reduction. One prototypical example is string concatenation: each process provides part of a string, and all of these parts will be concatenated together to produce one, long string. Other examples include computing longest common prefix of strings or computing the union or intersection of sparse sets with a reduction operation.

The biggest problem with using reduction operations for non-vector operations like string concatenation is that the facilities for providing user-defined operations are not well-suited to this operation. The current user-defined reduction operations must match the following signature:

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

However, this signature is not sufficient to express string concatenation. First, it only provides a single length argument, which is assumed to be the length of both input vectors. With string concatenation, the incoming string length may be different, requiring two incoming length arguments. Second, the result of the reduction must be written back into `inoutvec`. However, this array is not likely to be large enough to accommodate the complete result of the string concatenation, causing us to fail in any non-trivial string concatenation operation. Therefore, reduction of this kind will require a completely different signature for user-defined operations, which permits the input arrays to have different lengths and dynamic memory allocation of the output array. It is likely that reductions will either be unsupported (requiring the user to re-implement them) or have completely different interfaces from the existing reduction operations.

Another example would be parallel compression where the compression rate is not known in advance. All ranks start with the compression operation that produces a unpredictable amount of compressed data. This data is to be gathered to a single node for further processing or storing. An implementation with the current standard would be (assuming the compressed format supports heterogeneity):

```
int main(int argc, char* argv[]) {  
    MPI_Init(&argc, &argv);  
    int rank, p;  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_rank(MPI_COMM_WORLD, &p);
```

```

void *data, *rdata; int count, *counts, *displs;
get_data(&data);
compress(data, &count);
if(!rank) counts = malloc(p*sizeof(int));
MPI_Gather(count, 1, MPI_INT, counts, 1, MPI_INT, 0, MPI_COMM_WORLD)

if(!rank) {
    displs = (int*)malloc(p*sizeof(int));
    int allcount = 0;
    for(int i=0; i<p; i++) {
        if(!i) displs[i] = 0;
        else displs[i] = displs[i-1]+count[i-1];
        allcount += count[i];
    }
    rdata = malloc(allcount*sizeof(char));
}
MPI_Gatherv(data, count, MPI_BYTE, rdata, counts, displs, MPI_BYTE, 0,
            MPI_COMM_WORLD);
MPI_Finalize();
}

```

With `MPI_ANY_SIZE`, the code would be simplified to:

```

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int rank, p;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_rank(MPI_COMM_WORLD, &p);

    void* data; int count, *counts, *displs;
    get_data(&data);
    compress(data, &count);
    MPI_Gatherv(data, count, MPI_BYTE, &rdata, MPI_ANY_SIZE, 0, MPI_BYTE, 0,
                MPI_COMM_WORLD);
    MPI_Finalize();
}

```

Similar schemes are used commonly in scientific programming. Our proposal simplifies programming considerably and allows for special optimizations of the operation. However, it is unclear how the interface could be simplified to be similar to the current MPI standard so that no new function is necessary. We leave this definition open for discussion.

References

- [1] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 66–71, New York, NY, USA, 1999. ACM Press.

- [2] Douglas Gregor and Andrew Lumsdaine. Design and implementation of a high-performance MPI for C# and the common language infrastructure. In *Proceedings ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008. To appear.
- [3] Douglas Gregor and Matthias Troyer. Boost.MPI. <http://www.generic-programming.org/~dgregor/boost.mpi/doc/>, November 2006.
- [4] William D. Gropp and Rajeev Thakur. Issues in developing a thread-safe mpi implementation. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringer, and Jack Dongarra, editors, *PVM/MPI*, volume 4192 of *Lecture Notes in Computer Science*, pages 12–21. Springer, 2006.
- [5] Torsten Hoefer and Andrew Lumsdaine. Non-blocking collective operations for MPI-2. http://www.unixer.de/sec/standard_nbcoll.pdf, November 2007.
- [6] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. Modernizing the C++ interface to mpi. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, LNCS, pages 266–274, Bonn, Germany, September 2006. Springer.
- [7] Patrick Miller and Martin Casado. MPI Python. <http://sourceforge.net/projects/-pympi/>.
- [8] Craig E. Rasmussen and Jeffrey M. Squyres. A case for new MPI Fortran bindings. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.