MPI for High-Level Languages

Douglas Gregor, Jeff Squyres, and Andrew Lumsdaine

October 10, 2008

1 Introduction

Much of MPI's success can be attributed to its ability to support programs written in the programming languages that have been the mainstay of high-performance computing for years—Fortran, C, and (to a lesser extent) C++. However, programmers are increasingly turning to higher-level languages like Python and Java for their ease-of-use, even for HPC applications. To support these applications, a number of different language bindings for MPI have been developed for these languages, including bindings for Java [1], Python [2, 7–9, 11], and C# [4], and Ruby [10], along with improved bindings for C++ [5, 6].

The availability of MPI bindings for a wide variety of languages is a great asset to the MPI community, and will help insure that MPI remains relevant for years to come. MPI should strive to provide better support for bindings in high-level languages, but it should not attempt to provide these bindings as part of the standard, for two reasons: first, the amount of work to standardize and maintain MPI bindings is enormous, and the MPI Forum itself will not always have the necessary expertise in these languages to maintain these bindings in the longer term; second, regardless of whatever languages the MPI Forum chooses to support, advocates of other high-level languages will still need to build their own bindings.

We propose to improve MPI's support for high-level language bindings with extensions that address the major issues encountered in the development of bindings for MPI in popular high-level languages. Specifically, we address the following issues encountered by many high-level MPI bindings:

Transmitting objects (§2.1) Most of the high-level languages in use today support object-oriented programming. Therefore, MPI bindings for these languages must cope with the transmission of objects, which typically requires serialization of the object representation into a variable-length, platform-neutral format. This proposal includes changes that better support such variable-length data in two major areas:

- Point-to-point communication (§2.1.1)
- Collective communication (§2.1.2)

User-defined operations (§2.2) High-level language bindings are often required to support user-defined operations (e.g., for MPI_Reduce) that are expressed in the high-level language and may operate on user-defined or serialized data types. This proposal includes changes that make it possible to build user-defined operations that interact better with high-level languages.

Garbage collection and memory management (§2.3) Many high-level languages include some form of garbage collection. MPI bindings for these languages must interact with the garbage collector, e.g., by pinning memory provided to MPI or copying data between memory managed by the garbage collector and memory outside of the garbage collector. This proposal includes changes that improve the interaction between MPI and garbage collectors.

Deprecate C++bindings If all of the above issues are addressed, the current C++ MPI bindings should be deprecated and (later) removed. The bindings themselves provide very little utility over the C bindings (and in fact are typically lightweight wrappers over the C bindings), and are being superceded by more advanced bindings (e.g., [5]) that address modern C++ development practices. As with other high-level languages, the C++ community is better prepared to build usable MPI bindings for C++ than the MPI Forum is, and the maintenance of the C++ bindings has already taken a considerable amount of effort in the MPI Forum.

Provide only C bindings With the exception of the MPI standard Fortran bindings, all MPI bindings to other languages are built on top of the C MPI bindings. For this reason, our proposed changes are intended to be available from within C, only, and will not affect the Fortran bindings.

2 Summary of High-Level Language Binding Issues

This section summarizes the changes required to provide support for high-level language bindings in MPI. We make relatively few specific proposals, opting instead to point to existing proposals where possible and outlining the requirements on a solution where no such proposal exists.

2.1 Transmitting Objects

Nearly every high-level language encourages programmers to create their own data types, and provides abstraction mechanisms so that those data types can be used in similar ways to primitive data types. Thus, MPI bindings for these high-level languages attempt to provide similar interfaces for both user-defined and primitive types, e.g., by allowing the user to transmit a (Python, Java, C#, Ruby) object via MPI's communication routines.

The transmission of objects typically requires the MPI binding library to serialize the object into a platform-neutral format, using either MPI's serialization routines (e.g., MPI_Pack, MPI_Unpack) or language-specific mechanisms (e.g., Python's pickle module, Java's serialization interface). The serialized representation is then transmitted as raw bytes (or MPI_PACKED) and de-serialized by the receiver. However, the serialized representation of an object tends to have a variable length, and MPI provides insufficient support for transmitting variable-length data in its point-to-point communication (see [3]), and no support for variable-length data either in its collective or one-sided communication. The inability to communicate serialized objects forces high-level MPI bindings to use complex and inefficient implementations, including re-implementing all of MPI's collectives and one-sided communication over point-to-point.

2.1.1 Point to Point

The transmission of serialized data over MPI's point-to-point is complicated by the need to support multi-threading in high-level languages. In particular, since one cannot know the length of serialized data to be received *a priori* when receiving an object, the implementation of the high-level MPI binding must either probe for the message (which does not work in a multi-threaded application [3]) or send a separate message containing the size of the serialized data (which requires the use of an additional communicator).

The proposal to fix probe for multi-threaded applications [3] addresses the problem of transmitting serialized object data in a thread-safe manner over point-to-point. For example, the following code illustrates how to use matched probe and matched receive to receive an arbitrary number of bytes to be deserialized.

```
MPI_Message msg;
MPI_Status status;
/* Match a message */
MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &msg, &status);

/* Allocate memory to receive the message */
int count;
MPI_get_count(&status, MPI_BYTE, &count);
char* buffer = malloc(count);

/* Receive this message. */
MPI_Mrecv(buffer, count, MPI_BYTE, &msg, MPI_STATUS_IGNORE);

/* De-serialized data from the buffer. */
```

2.1.2 Collectives

With collective communication, MPI provides no facilities for coping with serialized objects. To illustrate the issue, consider the following code, which uses MPI.NET's equivalent to MPI_Allreduce to concatenate strings [4]:

```
public static string concat(string x, string y) { return x + y; }
string longString = world.Allreduce(myString, concat);
```

In this example, the string class requires serialization whenever it is transmitted, and must be de-serialized to perform the actual user-defined reduction operation (concatenation). The same Allreduce that supports string concatenation also supports arbitrary, user-defined operations on serialized object types, e.g., computing the intersection/union of hash tables, finding common subsequences within larger sequences, or computing global metrics of complex data structures.

One particularly interesting property of user-defined reductions like string concatenation is that the reduction actually increases the amount of data transmitted at each step. The MPI_Allreduce interface cannot accommodate such a growth:

```
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

The problem with this interface is that the count argument is a fixed value known to each processor, but this information cannot be known *a priori* in the string concatenation example: each processor only knows the length of its own string (stored in sendbuf), but cannot know the length of the resulting string and therefore cannot allocate an appropriate recvbuf without an additional round of communication. In more complex cases (e.g., using Allreduce to perform a complex reduction on containers of objects), the size of the required recvbuf cannot be computed without actually performing the reduction itself.

The problem of needing to precompute the size of the receive buffer also extends into the interface for creating user-defined MPI operations. A user-defined MPI operation is a function that has the following signature:

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

A user-defined string-concatenation function cannot be properly implemented with this interface, because there is no way to reallocate inoutvec to refer to a buffer that is large enough to contain the contents of the strings in invec and inoutvec.

MPI's Reduce, Scan, Exscan, and Reduce_scatter collectives all suffer from the same limitations as Allreduce with respect to their use with serialized object types. Thus, it is relatively common for high-level language bindings to reimplement each of these collectives over point-to-point communication, a task that requires a huge amount of duplicated effort and is likely to result in poorer performance than if the MPI library directly supported such variable-length collectives.

For the remaining collectives, the implementation burden on developers of high-level language bindings isn't nearly as large. For example, a broadcast of serialized objects can be implemented with two broadcasts: a broadcast of the length of the serialized data, after which all of the receivers can allocate buffers, followed by a broadcast of the data itself. A similar pattern follows for the remaining collectives, where one can typically use the collective itself to determine the size of the serialized data, then use the "v" variant of the collective to communicate the actual serialized data.

Although there has been some discussion of more powerful variable-length collectives, we know of no active proposals that would address the issues described above. However, see Section 3 for one potential solution.

2.1.3 One-Sided

One-sided communication via MPI_Put, MPI_Get, and MPI_Accumulate provides no support for operations on serialized object types. The first major problem is that, unlike with point-to-point communication, there is no way to use the existing one-sided communication operations to support variable-length data. A second problem is that, even if one could transmit the serialized representations of objects, there is no way for a user program to react to the completion of a MPI_Put or MPI_Get to de-serialize the transmitted data into a proper object. Finally, MPI_Accumulate cannot be used with user-defined MPI operations, making it completely unusable with user-defined types; we defer this last problem to Section 2.2.2.

We know of no attempts to make one-sided communication possible for serialized object types or user-defined types. This is likely due to a combination of factors, including the less-than-widespread use of MPI-2 one-sided communication and the significant complexity required to build one-sided communication for serialized objects into high-level MPI bindings, which requires emulation of one-sided communication using point-to-point communication.

At this point, we do not believe that it makes sense to extend the existing MPI-2 one-sided routines to support serialized data types. However, as the MPI Forum considers revisions to MPI's one-sided communication model, we will reconsider the impact of those changes on serialized data types.

2.2 User-Defined Operations

MPI allows users to supply their own MPI operations via MPI_Op_create, which can then be used in the various reduction and parallel-prefix collectives provided by MPI. Through this mechanism, users can extend the existing reduction and parallel-prefix collectives to work with new operations and new, user-defined data types. The suggestions in this section aim to make user-defined MPI operations more widely applicable for use in high-level MPI bindings.

2.2.1 Associating Data with MPI Operations

User-defined MPI operations are creating with the function MPI_Op_create:

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op);
```

In this case, function is a standalone function that will be invoked whenever the resulting MPI operation op is needed, e.g., to perform a reduction. The function is only permitted to access global data and its arguments, because there is no way to associate extra data with the MPI operation that will be passed to the function. Thus, it is not possible to express the idea that an MPI operation is a function with additional state. This extra state is typically very algorithm-specific, e.g., placing bounds on the computation performed by the MPI operation. For high-level languages, the extra state is often used to store more information about the calling context, e.g., a reference to the Python object whose method is being called by the MPI operation.

At present, there is no workaround to permit the use of stateful user-defined MPI operations in multi-threaded MPI applications. Within a single-threaded MPI application, only a single user-defined MPI operation can be in use at any given time, so the extra state associated with that oepration can be written into a global variable. However, this workaround will fail in multi-threaded MPI applications (which can have several collectives executing concurrently) or with the introduction of non-blocking collectives.

To properly support user-defined operations for high-level languages, MPI will need a way to associate extra data with user-defined MPI_Ops. Section 4 proposes a simple solution to this problem.

2.2.2 User-Defined Accumulation

MPI_Accumulate bans the use of user-defined operations, which makes it unusable with any user-defined types. While it is infeasible to support user-defined operations in the current MPI-2 one-sided communication model, it is possible that a revamped model would make accumulation of user-defined operations possible.

2.3 Garbage Collection and Memory Management

2.3.1 Send Buffer Access

A garbage collector may scan the contents of a send buffer while the MPI implementation is processing an MPI send of that data, violating the current prohibition on reading the send buffer before the send has completed. The current proposal to remove the restriction on send buffer access for MPI 2.2 addresses the problem of supporting send operations where the buffers themselves live in garbage-collected memory.

2.3.2 Managing Multiple Heaps

In many garbage-collected languages, there are actually two separate heaps from which memory can be allocated: a garbage-collected heap that is used for allocating memory within the language itself, and a separate system heap used by other libraries (like MPI) that are written in lower-level languages that explicitly malloc and free memory. The two heaps are often handled in different ways.

The garbage-collected heap generally does not permit explicit free operations, waiting instead for the garbage collector itself to find unused objects and reclaim them as necessary. More advanced garbage collectors may even move objects within the heap to reduce fragmentation, meaning that the address of existing objects can change. In these cases, the high-level language bindings must be careful to "pin" any memory allocated on the garbage-collected heap if that memory might be passed down into an MPI function, e.g., as the location of a send or receive buffer.

The system heap is typically the same heap as would be used by a C or Fortran program, which requires explicit malloc and free operations. For the most part, users of a garbage-collected language can ignore the existence of the system heap, since any memory required from that heap will be allocated or deallocated by native libraries (like MPI) that don't interact directly with the garbage-collected language. However, there are certain cases where high-level MPI bindings might need to interact with the system heap:

- If the language provides no ability to "pin" memory in the garbage-collected heap, the serialized representations of objects may need to be copied between the two heaps.
- If the language cannot read memory from the system heap, e.g., to deserialize an object that was transmitted via MPI_BLOB (see Section 2.1.2), the memory may need to be copied to the garbage-collected heap and deallocated from the system heap.

- Memory allocation behavior on either the system or garbage-collected heaps can affect allocations on the other heap, leading to unpredictable performance issues.

While memory pinning is typically in the domain of high-level MPI bindings, some MPI extensions might be required to help eliminate the need for extraneous copies of serialized data between the system and garbage-collected heaps. At present, it is unclear what these extensions might need to do.

3 Transmitting Serialized Data with Type Blobs

MPI's communication routines are designed to cope primarily with data types of fixed length, e.g., primitive types and structures of primitive types. Serialized objects are, by nature, variable in length. However, we could invent a new datatype that uses a fixed-length structure to represent that variable-length data, e.g.,

```
struct MPI_Blob {
  MPI_Aint length;[1]
  MPI_Aint address;
};
```

Coupled with a special new MPI datatype, MPI_BLOB, MPI can support serialized object data. Whenever MPI needs to send or receive data of type MPI_BLOB, it will actually send the length bytes stored at the given address rather than sending the MPI_Blob structure. Thus, each serialized object will map to an instance of MPI_Blob, with the address pointing to the serialized representation. The user is responsible for serializing/de-serializing the objects where needed—when sending or receiving a buffer of MPI_Blob objects, or manipulating those objects within a user-defined MPI operation—because serialization is very language-specific.

In our string-concatenation example, we could use MPI_BLOB as follows:

```
char *myString = ...;
MPI_Blob myStringBlob;
MPI_Blob longStringBlob;

myStringBlob.length = strlen(mystring)+1;
myStringBlob.address = myString;
MPI_Allreduce(&myStringBlob, &longStringBlob, 1, MPI_BLOB, concatOp, MPI_COMM_WORLD);

char* longString = (char*)longStringBlob.address[2];
printf("Long string = %s\n", (char*)longStringBlob.address);
```

The memory associated with MPI_Blobs needs to be carefully managed. In some cases, such as myStringBlob in the string-concatenation example above, the user will both allocate

[1]The type of the length field might change, depending on the resolution to the discussion about supporting large message counts.

[2]This cast will not always be valid; most likely, MPI will need a function that maps from MPI_Aint values to void*. Alternatively, addresses in blobs can be expressed as void* values.

and deallocate the memory associated with the blob. For longStringBlob, however, the implementation will allocate the memory associated with the blob (since the user cannot do so in advance) while the user is responsible for its deallocation. The situation is further complicated by temporary buffers passed to user-defined MPI operations for intermediate reductions and by MPI_IN_PLACE.

One simple strategy is to require that all memory allocated for MPI_Blob objects be allocated with MPI_Alloc_mem and freed with MPI_Free_mem. Users would be responsible for deallocating any memory returned to them via a receive buffer, and any memory they have allocated to be passed to MPI in a send buffer. Note, however, that this simple strategy may cause problems for some garbage-collected languages; see Section 2.3.2.

Additionally, the MPI_BLOB should not be permitted in any derived datatypes, because it is only meant to be used directly as the datatype of a communication operation.

4 Data In User-Defined Callbacks

Most C libraries that provide callbacks of some sort also allow the user to provide an extra **void**∗ argument that will be passed on to the callback function whenever it is invoked. MPI could do the same, with an improved version of MPI_Op_create and MPI_User_function that allow users to provide a data pointer at the time that the MPI_Op is created, e.g.

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

int MPI_Op_create_data(MPI_User_function_data *function, int commute,
                       MPI_Op *op, void *data);
```

MPI_Op_create_data creates an MPI_Op that also includes a user pointer. The MPI_Op can be used anywhere that an MPI operation can be used, even though it calls a slightly different kind of user function. Therefore, there is no need to introduce additional variants of, e.g., Reduce or Scan, because the MPI implementation will handle the difference between MPI_Op values created with MPI_Op_create vs. MPI_Op_create_data internally.

It is very likely that high-level language bindings will create a new MPI_Op each time that a reduction operation is called, since the user data pointer will likely change each time. This usage is acceptable, since MPI_Op creation is typically very quick and requires no communication.

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] Lisandro Dalcin. MPI for Python. `http://mpi4py.scipy.org/`, 2007.

[3] Brian Barrett Douglas Gregor, Torsten Hoefler and Andrew Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical report, MPI Forum, 2008.

[4] 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.

[5] Douglas Gregor and Matthias Troyer. Boost.MPI. `http://www.generic-programming.org/~dgregor/boost.mpi/doc/`, November 2006.

[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] Wilfred Li. MYMPI. `http://peloton.sdsc.edu/~tkaiser/mympi/`, 2006.

[8] Patrick Miller and Martin Casado. MPI Python. `http://sourceforge.net/-projects/pympi/`.

[9] Ole Nielsen. Pypar. `http://sourceforge.net/projects/pypar`, 2007.

[10] Emil Ong and Rudi Cilibrasi. MPI Ruby. `http://mpiruby.sourceforge.net/`, July 2001.

[11] Mike Steder. Maroon MPI. `http://code.google.com/p/maroonmpi/`, 2006.