The Design and Evolution of the MPI-2 C++ Interface

Jeff Squyres[*], Bill Saphir[†], Andrew Lumsdaine[‡]

Abstract

The original specification for the Message Passing Interface (MPI) included language bindings for C and Fortran 77. C++ programs that used MPI were thus required to use the C bindings. With MPI-2, a C++ interface for all of MPI is specified. In this paper, we describe the design of the C++ interface for MPI and provide some of the history and motivations behind the design decisions.

1 Introduction

The Message Passing Interface (MPI) is a specification for a library of routines that provide an infrastructure for parallel message passing applications. MPI provides routines for point-to-point communication and collective operations, as well as support for the development of safe libraries and miscellaneous related functionality. The MPI standard defines C and Fortran (77) bindings for all MPI functions.

The MPI Forum reconvened in 1995 to consider additions to the MPI standard, known as MPI-2. MPI-2 has now been finalized [1]. Included in MPI-2 are support for one-sided communication, cooperative I/O, dynamic process management, and many small clarifications and extensions. MPI-2 also defines a C++ interface for all MPI-1 and MPI-2 functions.

The development of a C++ interface followed a winding path as the Forum considered many different styles of interfaces. What eventually emerged is closely related to the C interface, but has a number of important features that make it more appealing to C++ programmers and exploit MPI features in new and unanticipated ways.

In this paper we describe the design of the MPI-2 C++ inteface, explaining what choices were made and why. We assume that the reader is familiar with MPI in general and the MPI C bindings in particular.

2 The Major Issues

2.1 Big or Small?

A number of proposals for the MPI C++ bindings were introduced during the course of the MPI-2 Forum. The original (preliminary) proposal was modeled closely after the MPI++ [2] class library. The initial proposal introduced a major question to the Forum: Should the bindings be a full-blown class library or should they be something closer to the C interface? Both options were explored, with proposals for each being made over a period of time. After the Forum had a chance to study and evaluate the class library proposal, it was felt that the role of the C++ bindings was to facilitate the development of class libraries, not to actually be a class library. The proposed class library went on to become Object-Oriented MPI [3].

After the class library approach was discarded, the pendulum swung the other way and a proposal for very low-level bindings was made. These proposed bindings were very close to the C bindings, but provided

[*]Laboratory for Scientific Computing, University of Notre Dame, Notre Dame, IN 46556; squyres@cse.nd.edu

[†]Computing Sciences, Lawrence Berkeley Laboratory; WCSaphir@lbl.gov

[‡]Laboratory for Scientific Computing, University of Notre Dame, Notre Dame, IN 46556; lums@lsc.nd.edu

a few C++ features such as `const` and reference semantics. However, the Forum felt that these bindings were too low-level and did not do enough to enable class library design.

Thus, the final, and accepted, proposal for MPI C++ bindings found the middle ground between big and small (or, perhaps like MPI itself, it was *both* big and small). The bindings contain a limited number of class library like features, but the bindings still remain limited enough not to constrain class libraries built using them.

2.2 Object-Based Design

The design of MPI itself is object-based. MPI defines a number of objects — Communicators, Groups, Requests, etc. These objects are referred to in C by handles and in Fortran by integers. It was an obvious choice for the C++ bindings (once there was a decision to go with the "small" interface) to turn the handles into regular C++ objects. Most MPI functions became methods associated with these objects. In most cases, which object to associate with a given function was "obvious" to the Forum, though the rationale is ultimately more intuitive than rigorous. Some of these obvious choices were `comm.send()` `request.wait()` and `comm.dup()`. There are good arguments for preferring `comm.send()` to `datatype.send()`, for instance, but this was not an issue for the Forum, and we do not discuss them here.

Which object to associate with each method turned out not to be so straightforward, however. We defer a discussion of these methods until the next sections.

2.3 Constructors and Destructors

MPI-1 has routines that clearly create objects (e.g., `MPI_COMM_DUP`) and routines that free them (e.g., `MPI_COMM_FREE`). It seems at first natural in C++ to turn these and related functions into constructors and destructors. The main problem with this is that creating and freeing a communicator are collective operations. Thus a declaration

```
MPI::Comm a(MPI_COMM_WORLD)
```

intended to implicitly `dup()` `MPI_COMM_WORLD` would be a "collective declaration." Worse, the return from the routine where this variable was declared would be a collective operation, when the object was explicitly freed.

MPI-2 therefore chooses a path that may be unfamiliar to C++ programmers: the application is responsible for explicitly creating and freeing objects. Memory management is not done automatically by constructors and descructors.

As a consolation prize, MPI-2 specifies constructors that initialize objects to NULL objects, and destructors that do free the "top-level" C++ object, but not the underlying object to which it refers.

2.4 Exceptions

The C bindings for almost all MPI functions (except `MPI_Wtime()` and `MPI_Wtick()`) return an error code. In principle, an application can check this error code and take some action if there is an error. In practice, this error code is never used, for several reasons. The first is that by default, errors cause an MPI program to abort. This is often the desired behavior. The second is that even if MPI is configured to return errors to the application, the MPI standard says that the state of a program is undefined after an MPI error. About the only thing an application can (semi)reliably do is print and error message and abort. Finally, it is simply tedious to check the return value from every call, and can be quite hard to figure out what to do with that error (e.g., pass it up the calling stack or take action directly). C++ exception handling provides a mechanism

to handle the last of these, in case the first two are ever solved by a robust MPI implementation. C++ applications are given the option of setting the default error handler to `MPI_ERRORS_THROW_EXCEPTIONS`, in which case MPI functions throw a C++ exception when there is an error. Along with this, C++ methods do not return error codes as function values. The only way to deal with errors in C++ programs is to abort (the default behavior) or use the exception mechanism.

2.5 Return values

In C and in Fortran, values are returned through the argument list. For instance,

```
MPI_Comm_dup(oldcomm, &newcomm)
```

returns a new communicator `newcomm`. Part of the reason for this is that the return value of the function is reserved for the error code. Since C++ MPI methods do not return error codes, the function return value is freed up to hold more than an error code.

In many MPI functions there is a designated "out" quantity that makes sense as a return value in the C++ case. In other functions, the out quantity may not be readily returned (e.g., when it is an array) or there may be no out quantity at all. In these cases, the corresponding C++ bindings return `void`.

2.6 Naming

MPI-1 did not use consistent naming rules. Often, names are of the form `MPI_Object_function` as in `MPI_COMM_SPLIT`, `MPI_INTERCOMM_MERGE`, and sometimes they are not, e.g., `MPI_TYPE_CONTIGUOUS`. Sometimes the verbs are consistent, e.g., `MPI_COMM_FREE`, `MPI_COMM_FREE` and sometimes they are not, e.g., `MPI_ERRHANDLER_SET` and `MPI_ATTR_PUT`.

While it was recognized that there was inconsistency, the Forum felt that it would be disruptive to change the familiar names. In cases where new functions replace deprecated old functions of the same functionality, the "correct" name was used, (e.g. `MPI_TYPE_CREATE_STRUCT`) but this was as far as name-changing went.

For the C++ bindings, the Forum decided to use rigorously consistent names. The names are thus slightly different from the C names in several cases. This was done for three reasons:

1. The bindings are new. There is no existing code that needs to be changed or book that must to be rewritten.

2. It was felt that using the inconsistent names was more of a problem in a C++ context where where the structure highlights discordant design, e.g., `status.get_count()`, `status.get_elements()` but `comm.size()` and `comm.rank()`.

3. The C++ names are necessarily different from the C names already, e.g., `comm.split()` instead of `comm.MPI_Comm_split()`.

One relatively new feature of ANSI C++ is the `namespace` construct which allows programs to provide explicit scoping of names. The MPI C++ bindings make use of this feature by including all named quantities within the scope of a `namespace MPI`.

2.7 References, Pointers and `MPI_STATUS_IGNORE`

The MPI C++ bindings use reference semantics when possible. Thus the binding for `MPI_COMM_SEND` is

```
void Comm::Send(..., const Datatype& datatype, ...)
```

The only pointer arguments are `char*` arguments for strings (because of convention) and `void*` arguments for choice buffer arguments.

This introduces a problem with the the new MPI-2 option to ignore a returned `status` by specifying `MPI_STATUS_IGNORE`. In C, this constant is associated with a `MPI_Status*` argument and has the value `NULL`. In C++, the corresponding status argument is passed by reference, so it's not possible to pass a NULL pointer. The C++ bindings take another route, which is to have two bindings for every function with an OUT `status` argument in the language-neutral specification. One binding has a reference to a `status` argument and the other has no `status` argument. `MPI_STATUS_IGNORE` is not defined in C++.

3 Design details

An abbreviated definition of the `MPI` namespace and its member classes is as follows:

```
namespace MPI {
  class Comm {...};
  class Intra_comm : public Comm  {...};
  class Graph_comm : public Intra {...};
  class Cart_comm  : public Intra {...};
  class Inter_comm : public Comm  {...};
  class Datatype {...};
  class Errhandler {...};
  class Group {...};
  class Op {...};
  class Request {...};
  class Prequest : public Request {...};
  class Grequest : public Request {...};
  class Status {...};
};
```

4 Conclusions

Our full paper will include further discussions of the design details of the C++ interface for MPI as well as extended examples.

References

[1] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing*, volume 1 of *Lecture Notes in Computer Science*, pages 128–135. Springer Verlag, 1996.

[2] Anthony Skjellum, Ziyang Lu, Purushotham V. Bangalore, and Nathan E. Doss. Explicit parallel programming in C++ based on the message-passing interface (MPI). In Gregory V. Wilson, editor, *Parallel Programming Using C++*. MIT Press, 1996. Also available as MSSU-EIRS-ERC-95-7.

[3] Jeffrey M. Squyres, Brian C. McCandless, and Andrew Lumsdaine. Object oriented MPI reference. Computer Science and Engineering Technical Report 96-10, University of Notre Dame, 1996.