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

Jeffrey M. Squyres¹, Bill Saphir², and Andrew Lumsdaine¹

¹ Department of Computer Science and Engineering, Notre Dame, IN 46656
² NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720

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) [2, 4, 6] 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, collective operations, as well as support for the development of safe libraries and other 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 [3]. MPI-2 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 ways.

In this paper we describe the design of the MPI-2 C++ interface, 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++ [5] 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 later became Object-Oriented MPI [7].

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 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, like MPI itself, it was both big and small). The bindings contain a number of class library-like features, but 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 by handles in C and by integers in Fortran. 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. These objects, however, retained the same handle-based semantics as their C counterparts. Namely, the C++ objects are user-level handles to underlying implementation-dependent 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. Examples of these obvious choices were Comm.Send() and Request.Wait(). 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.

Some functions were “obvious” candidates for a specific class even though they did not include a single IN or OUT argument of that type. For example, MPI::Datatype::Create_struct takes an IN array of MPI::Datatype. Such functions were still defined on that class, but were declared static since they have no corresponding this pointer.

2.3 Naming

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

Unlike MPI-1, MPI-2 uses a consistent naming scheme of the form MPI-Object_action_subset. For the C++ bindings, the Forum decided to use the consistent names. Although the MPI-2 C++ scheme for both MPI-1 and MPI-2 functions are slightly different from the MPI-1 C names in several cases, the