# Using MPI with C# and the Common Language Infrastructure

Jeremiah Willcock jewillco@osl.iu.edu
Andrew Lumsdaine lums@osl.iu.edu
Open Systems Laboratory
Pervasive Technology Laboratories
Indiana University
Bloomington, IN 47405
Tel: (812) 855-3608 Fax: (812) 855-4829

Arch Robison arch.robison@intel.com
KAI Software Laboratory
Intel Corporation
1906 Fox Drive
Champaign, IL 61820
Tel: (217) 356-2288

## ABSTRACT

We describe two interfaces for using the Message Passing Interface (MPI) with the C# programming language and the Common Language Infrastructure (CLI). The first interface provides CLI bindings that closely match the original MPI library specification. The second library presents a fully object-oriented interface to MPI and exploits modern language features of C#. The interfaces described here use the P/Invoke feature of the CLI to dispatch to a native implementation of MPI (in our case, LAM/MPI). Performance results using the Shared Source CLI demonstrate there is only a small performance overhead incurred.

## 1. INTRODUCTION

Microsoft has recently announced .NET as a platform for supporting a wide variety of services and applications. Although at this early stage, it is not clear that .NET will be a suitable platform for high-performance computing, the certain ubiquity of .NET in the near future indicates that the technology should at least be investigated. Two important pieces of .NET have recently been standardized by ECMA: the C# programming language [3] and the Common Language Infrastructure (CLI) [4].

As a first step in our nascent HPC .NET initiative, we have designed and implemented interfaces to allow the Message Passing Interface (MPI) to operate with the C# language and the CLI. Two layers of interface are proposed: a low-level interface that is similar to the C++ MPI bindings, and a high-level interface that exploits modern programming language features of C#.

MPI is a standardized interface for message passing in parallel computers [9]. It provides several useful abstractions that simplify the use of distributed-memory parallel computers and clusters. These include *communicators*, which provide an abstraction for a set of processes (or *ranks*, in MPI terminology) as well as a separated communication domain (so that messages sent within one communicator cannot be received within another). There are also abstractions for various virtual network topologies, parallel input/output, and numerous other features used in distributed-memory computation.

The .NET initiative consists of two main parts: the Common Language Infrastructure and the C# programming language. The Common Language Infrastructure (CLI) provides a standardized, language-independent bytecode interpreter and a runtime library that is shared among all CLI-compliant languages. The C# programming language is a new object-oriented programming language created by Microsoft specifically for .NET. The C# language is similar to Java, but with new features such as delegates (similar to function objects) and properties (conceptual fields of an object with getter and setter methods). These features are implemented within the CLI, but the C# language provides more convenient interfaces to them.

## 2. C# FOR HIGH-PERFORMANCE COMPUTING

C# is an intriguing candidate for high performance computing, because it has the key attractions of Java, yet without throwing out some features dear to scientific programming. Like Java, C# is a strongly-typed bounds-checked object-oriented language with garbage collection. Thus it protects the programmer from notorious pointer and memory-management errors. C# goes beyond Java, however, by providing dense multidimensional arrays, operator overloading, an improved memory model for concurrency, and lightweight value types. The latter are essential for efficient representation of "small objects" such as complex numbers, coordinate vectors, rotation matrices, etc. Similar improvements for Java have been proposed by Java researchers [1, 13, 15, 20]. Standard C# has these now.

Furthermore, the CLI offers pragmatic, though oft forgotten, features for serious computing, high performance or not. The first is simple cross-language interoperability, which permits various parts of an application to be developed in languages appropriate for the parts. For instance, a user interface can be written in C# and connected to a numerical core written in Fortran or APL. Indeed, vendors have announced plans for CLI versions of both Fortran and APL. The "simple" is important here: the CLI standard includes a Common Language Specification, which specifies rules for types and interfaces to be shared across languages. Thus data structures can be passed between languages without obscure "handles", com-

This paper will appear as Indiana University Computer Science Department Technical Report 570.

plicated marshalling, or arcane name mangling.

The second critical feature is simple reuse of existing legacy data structures and routines. For legacy data structures, the key features are support for raw pointers ("unmanaged pointers" in CLI parlance) and optional control over structure layout. For legacy routines, the key features are the ability to make direct calls to such routines, and the ability to specify marshalling attributes for such calls.

Thus, C# and CLI have promise as a platform for practical object-oriented high performance computing, by allowing multilanguage development of new efficient type-safe garbage collected code with a simple bridge to legacy code. The chief drawback, currently shared by Java, is the lack of parametric polymorphism. Kennedy and Syme have recently proposed an approach for incorporating parametric polymorphism into the CLI [8].

# 3. INTERFACING .NET TO THE C MPI INTERFACE

The .NET programming environment and the C# language provide direct interfaces to existing libraries written in C and C++. These interfaces allow a native function to be imported as a method in a C# program. The .NET environment automatically generates code to convert parameters between .NET formats and the platform's native data representations. This part of .NET is named P/Invoke.

Creating an interface to a native function involves exporting the function from a shared library and writing a prototype for the function in C#. Assuming that the function is already exported from a shared library, each parameter must be examined to determine its .NET equivalent. Numbers map easily between languages, but pointer-based structures are more difficult to convert. One option, used in the interface to MPI, is to use pointer-sized integers (the `IntPtr` type). This requires the user to explicitly fix the object to be passed in one location and get its address. Another similar option is to use a pointer type in C#. However, this method requires users calling the native function to write "unsafe" C# code (explained later), and the user must still pin the pointer-based data structure during the native call. A last option is to use the `MarshalAs` attribute on the parameter, which passes the location of the object directly without any user intervention. This option is the simplest, but it does not guarantee that the object will not be moved after the call completes. It also requires the object to be copied to a separate buffer in some cases, which may lead to performance problems. Thus, the .NET interface to MPI requires explicit user pinning and unpinning of data structures. However, the high-level interface to MPI makes this process automatic.

As an example, a simple native function interface is given in Figure 1. This code provides a static method (named `MyClass.MyFunction`) that interfaces to a native function named `myfunc` (from the library `mylib.so`). The native function is required to accept one integer parameter and return an integer result, and the newly-created method will have the same interface.

However, most MPI functions are not as simple as the example given here. In particular, MPI functions use data buffers that are represented in C as values of type `void *`. There is a keyword `unsafe` in C# that allows raw pointers to be used directly within its scope, but a special compiler setting must be used to allow this. In addition, any user code using pointers must also be compiled in unsafe mode. However, there is a class in the standard library that

```
public class MyClass {
  [DllImport("mylib.so", EntryPoint="myfunc")]
  public static extern int MyFunction(int arg);
}
```

**Figure 1: Basic P/Invoke example**

```
// ... Declare and set value of some object
// obj (of any C# reference type)
GCHandle handle = GCHandle.Alloc(obj,
                    GCHandleType.Pinned);
IntPtr ptr = handle.AddrOfPinnedObject();
// Use value of ptr in native methods
handle.Free();
```

**Figure 2: GCHandle object pinning example**

allows access to a pointer to an object in safe mode. Objects in .NET can be moved arbitrarily by the garbage collector, and this must be prevented when they are in use by MPI functions. To solve these two problems, the `GCHandle` class from the Common Language Runtime is used. One use of this class is to keep an object's memory location fixed, as well as to obtain this location using only safe C# code. This method of obtaining object locations also allows the object to be fixed in a memory location (or "pinned") for any length of time, unlike the `fixed` keyword used to find object locations in unsafe code. An example use of the `GCHandle` class is shown in Figure 2.

In the .NET interface to MPI, the low-level bindings require the user to explicitly pin and unpin the data buffers for MPI to use, and thus all buffers are of C# type `IntPtr` (pointer-sized integer). The high-level bindings, on the other hand, automatically pin and unpin data buffers as needed. This is usually straightforward, but some request types (such as nonblocking and persistent requests) have more complicated functionality. The C# classes for MPI requests must store the data buffer in use, and ensure that it is unpinned when the request is completed.

Another, more MPI-specific, problem is that MPI implementations are only required to be source-compatible. That is, a program compiled with one MPI implementation is not likely to work when linked with a different implementation. In particular, MPI data types and constants are defined in C header files, and differ between implementations. Unlike C++, C# is not able to directly import these header files. There are several solutions used for these problems. MPI constants are represented as functions in a C library, each of which returns the value of a particular constant. These functions are called once each at program startup and their results are cached in C# variables. MPI opaque data types (which are required by the MPI standard to be either integers or pointers) are currently handled using a source file that is customized for each MPI implementation. These are then used as input to the code generator that creates the low-level bindings, and inserted at the top of each file of the high-level bindings (which are otherwise hand-written). The one transparent MPI data type (`MPI_Status`) is handled by creating functions in C to create and delete objects of this type, and to access this structure's members. A wrapper written in C# then provides an interface that looks more like a normal C# class.

Otherwise, interfacing the .NET environment to MPI is fairly

straightforward. Most functions take only relatively simple parameters (either integers, data buffer pointers, or MPI-defined data types). Thus, most of the native interfaces can be easily created by a code generator. This generator is described later in this paper.

# 4. LIBRARY DESIGN

The design of the C# bindings to MPI has two layers: a low-level interface based on the C++ bindings in MPI-2 [10] (which we call the CLI bindings), and a high-level interface based on the OOMPI library [17] (which we call MPI.NET). The high-level interface and low-level interface are both implemented using an internal set of bindings that are designed to exactly match the C MPI bindings. The low-level interface tries to follow the MPI C++ bindings, even when those bindings conflict with common C# conventions (such as in the names of constants). On the other hand, the high-level interface tries to follow C# naming conventions, even though it also attempts to allow an easy transition for programmers who already know the C or C++ MPI bindings or the OOMPI library.

## 4.1 CLI (low-level) bindings

The low-level C# bindings to MPI attempt to follow the existing C++ bindings given in the MPI-2 specification [10]. At the top level, there is one class, named `MPI`. This class is not instantiable, and all of its methods are marked as `static`. In the C++ MPI bindings, this grouping of classes, functions, and constants is implemented as a namespace, but C# namespaces can only contain classes and other namespaces. Thus, all of the MPI constants, functions, and classes are contained in a class instead. The low-level C# bindings try to follow MPI naming conventions whenever possible, even when they conflict with the preferred naming conventions for C#. This interface also requires explicit indications of the length and MPI data type of the data being sent or received, as the C++ bindings require. Buffers used for data in the low-level C# bindings need to be pinned explicitly by the user, and a pointer (represented as an integer) to the start of the data buffer is passed to the MPI bindings. Currently, there is only one top-level class (for the non-profiling interface), but a second top-level class (named `PMPI`) is planned to contain the profiling interface to MPI.

The implementation of this set of MPI bindings is fairly straightforward. Similar to the Notre Dame implementation of the C++ bindings [18], each object just contains the underlying C representation of the MPI object. For instance, the `MPI.Datatype` class contains an instance of the C `MPI_Datatype` type. There is one major difference, though, between the C++ bindings and the low-level C# bindings: exceptions in C++ are thrown by a custom error handler installed by the C++ bindings, but in C# they are thrown by an explicit check around each MPI function. This approach requires each error-handler object to record whether it is the special `ERRORS_THROW_EXCEPTIONS` handler, and for better performance each object that may have an error handler attached records whether its current error handler is required to throw an exception. Each MPI call is then wrapped in code that checks its return value and throws an exception if the appropriate flag is set and the MPI function was unsuccessful. The exception-throwing error handler, when installed, sets the underlying MPI implementation to return an error code on failure so that the exception mechanism will work correctly. This implementation may be slightly slower than throwing an exception through the underlying MPI implementation, but is simpler and not dependent on features of the underlying C compiler (allowing exceptions to be thrown through C programs).

```
class LowlevelRoundRobin {
  public static int Main(string[] args) {
    MPI.Init();
    MPI.Comm cw = MPI.COMM_WORLD;
    uint rank = cw.Get_rank();
    uint size = cw.Get_size();
    uint messageLength = 10;
    Console.WriteLine("Hello, I am {0} of {1}",
      rank, size);
    double[] data = new double[messageLength];
    GCHandle dataHandle = GCHandle.Alloc(data,
      GCHandleType.Pinned);
    IntPtr dataPtr =
      dataHandle.AddrOfPinnedObject();
    if (rank == 0)
      for (int i = 0; i < messageLength; ++i)
        data[i] = i;
    if (rank == 0)
      cw.Send(dataPtr, messageLength, MPI.
              DOUBLE, (rank+1) % size, 100);
    cw.Recv(dataPtr, messageLength, MPI.
            DOUBLE, (rank+size-1) % size,
            100, MPI.STATUS_IGNORE);
    if (rank != 0)
      cw.Send(dataPtr, messageLength, MPI.
              DOUBLE, (rank+1) % size, 100);
    dataHandle.Free();
    MPI.Finalize();
    return 0;
  }
}
```

**Figure 3: CLI bindings C# example program**

Another feature of MPI that causes implementation issues in an object-oriented binding is that some functions accept parameters that are arrays of MPI data types. In the object-oriented wrapper layer, these arrays are arrays of objects, which are different from the arrays of underlying data types that the MPI implementation requires. In these cases, the arrays are copied into buffers of the correct type (on input and/or output as necessary). This is the same technique that is used by the Notre Dame C++ bindings for MPI. Buffers of `MPI.Status` objects are more difficult to handle (since this type is a structure, not just an integer or pointer), but is handled similarly. A new array of C `MPI_Status` objects is created, and the information from the C# objects is copied in and out of the C array as needed. This copying of data arrays causes the correct behavior, but can lead to performance problems in some cases. This is one reason that MPI.NET is built atop a set of C-level bindings, rather than atop the lower-level C# object-oriented layer.

As an example of the use of the low-level C# bindings, a simple "Hello, world" and round-robin program is shown in Figure 3.

The CLI bindings were created using a code generator. This generator converted a list of MPI functions into both native interfaces for the C functions and into object-oriented wrappers that are based on the MPI C++ bindings. The list of functions contained the C binding information as well as extra information about the C++ class interface: the class (if any) of which each function is a member, whether each function is static or not, and whether any output parameters of each function become return values (rather than `out`

parameters). This is enough information to create most of the CLI bindings. However, some MPI classes are not just simple wrappers of C functions, and these are written by hand. The code generator made adding new MPI functions and fixing errors in the specifications very simple.

## 4.2 MPI.NET (high-level) bindings

The high-level C# bindings to MPI are based on the interface to the OOMPI C++ library [17]. The OOMPI library provides several useful abstractions for MPI programming, including ports and messages. A *port* abstracts a source or destination for a message. This class includes both the communicator to use for the send or receive operation and a particular rank within the communicator to interact with. A *message* (or *typed buffer* in MPI.NET) includes the starting address, length, and MPI data type of a data buffer to send or receive. The OOMPI library also provides a simplified way to create user data types that can be sent and received using MPI. The high-level C# bindings try to preserve these features of OOMPI, and also create an interface that better matches the standard naming conventions used in other C# libraries. Therefore, a constant such as `MPI_STATUS_IGNORE` becomes `MPICS.Status.Ignore` in the high-level C# bindings, and the class `MPI_Status` from the C++ bindings is named `MPICS.Status` (and so `Ignore` is a static property of this class). In addition, some methods in the C++ bindings (such as `MPI::Comm::Get_rank()`) become properties, which have the same user interface as member variables but are implemented as functions internally. The operator overloading used in OOMPI for message sending and receiving (using syntax such as `port << message`) will not be preserved in the C# interface, since C# does not use this syntax for I/O operations.

One particular feature of the MPI.NET system deserves attention: pinnable buffers, represented by the `IPinnable`, `IBuffer`, and `ITypedBuffer` interfaces. The MPI.NET interface, unlike the lower-level interfaces presented in this paper, provides automatic pinning and unpinning for user-defined buffers. To allow more flexibility for users, any buffer type implementing the proper interfaces can be used in MPI functions. The simplest interface is `IPinnable`. This interface requires two methods and one property: `Pin()` (to pin the data buffer), `Unpin()` (to unpin the data buffer), and the `Location` property (an `IntPtr` to the beginning of the buffer). Calls to the `Pin()` and `Unpin()` methods must also be nestable. This interface does not require the buffer's length or data type to be provided. The `IBuffer` interface extends `IPinnable` to add a property to obtain the length (in bytes) of the data buffer. The interface used by most MPI calls, however, is `ITypedBuffer` (equivalent to the `Message` class in OOMPI). This interface extends `IPinnable`, and also requires properties to access the MPI data type and count of the buffer. This information then is used to fill in several parameters in each MPI call using the buffer. This provides a simpler programming model for library users. These interfaces therefore allow the user to create custom data buffer types which will interoperate correctly with MPI.NET's automatic buffer pinning.

Other than buffer management, the high-level MPI bindings for .NET are fairly straightforward to implement. This set of bindings uses wrappers around each MPI call to throw exceptions on failure when a user selects this mode, just like the low-level bindings use. Similarly, the objects in the high-level bindings are usually just containers for the underlying MPI data types. However, this level has extra information in communicators (for knowing whether the current error handler should throw exceptions) and requests (to ensure that buffers are unpinned when the request is completed). This layer of bindings is almost all hand-written, but must of the code is repeated between various functions. Mostly, this layer of bindings provides a nicer user interface atop largely the same implementation strategy as the low-level bindings.

```
class HighlevelRoundRobin {
  public static void Main() {
    using (MPINET.Session session =
            new MPINET.Session()) {
      MPINET.Communicator cw =
        MPINET.Communicator.World;
      int rank = cw.Rank;
      int size = cw.Size;
      uint messageLength = 10;
      Console.WriteLine("Hello, I am {0}/{1}",
        rank, size);
      double[] data =
        new double[messageLength];
      MPINET.ArrayTypedBuffer dataBuffer =
        new MPINET.ArrayTypedBuffer(data);
      if (rank == 0)
        for (int i = 0; i<messageLength; ++i)
          data[i] = i;
      MPINET.Port dest = cw[(rank+1) % size];
      MPINET.Port src = cw[(rank+size-1)%size];
      if (rank == 0)
        dest.Send(dataBuffer);
      src.Recv(dataBuffer);
      if (rank != 0)
        dest.Send(dataBuffer);
    } // End of MPI session
  } // End of Main
} // End of HighlevelRoundRobin
```

**Figure 4: MPI.NET C# example program**

As an example of a simple program using the C# high-level bindings, Figure 4 is a "Hello, world" and round-robin program.

Proposed class hierarchies for both levels of bindings are given in Figures 5 and 6.

# 5. RELATED WORK

There are several interfaces to MPI from object-oriented languages in existence. The C++ MPI bindings in the MPI-2 standard [10] and the OOMPI library for object-oriented MPI communications [17] have already been mentioned as major influences on the C# bindings proposed by this paper. There are, however, at least two other bindings from C++ to MPI, as well as bindings from Java, Python, and Ruby. These interfaces provide different levels of abstraction, as well as use the varying features of their respective languages differently to achieve the interface to MPI.

## 5.1 C++ bindings

There are several interfaces from C++ to MPI. The MPI-2 standard contains a standardized set of bindings for all of the MPI functions [10]. These interfaces tend to follow the MPI interfaces closely, and do not try to provide a higher-level interface. However, the C++ bindings provide classes for communicators, datatypes, and other MPI objects. Some MPI functions then exist in C++ as

```
class MPI { // Low-level bindings
    class Comm {
        // Constructors
        // Point-to-point communications
        // Collective communications
    }

    class Datatype {
        // Datatype construction operations
    }

    class Status {
        // Constructors
        public uint Source {get;}
        public uint Tag {get;}
        public uint Error {get;}
        // Other methods
    }

    class Group { ... }
    class Intracomm: Comm { ... }
    class Cartcomm: Intracomm { ... }
    class Graphcomm: Intracomm { ... }
    class Intercomm: Comm { ... }

    class Errhandler { ... }
    class Op { ... }
}
```

**Figure 5: Class hierarchy for CLI bindings**

methods of these classes. The standard C++ bindings provide a fairly low-level interface to MPI, and the low-level bindings for C# proposed here are based on this set of C++ bindings.

A somewhat higher-level set of interfaces from C++ is provided by the Object-Oriented MPI (OOMPI) library [17]. This library provides classes similar to those in the standard C++ bindings, but also has simplified ways to create MPI data types and send point-to-point messages. The high-level C# bindings proposed by this paper are based on the interfaces in OOMPI, but with some name and structure changes to more closely match C# conventions.

Another interface from C++ to MPI is provided by MPI++ [16]. This interface is fairly low-level, and was one of the main influences for the MPI-2 C++ bindings [18]. It provides a relatively thin layer of abstraction on top of the C MPI bindings. It is very similar in structure to the C++ bindings.

Lastly, the mpi++ library provides another set of object-oriented wrappers around MPI functions [7]. This library is different from the others in that it uses C++ templates to encapsulate message information (such as the MPI data type in use, and whether the message is blocking or non-blocking) within the type of the message class, rather than in instance variables. This requires that these decisions be made statically rather than dynamically, but the MPI C bindings also require this (through the use of different functions for blocking vs. non-blocking communications). The advantage is that the same functions can be used, and the compiler can still know at compile-time which mode is in use.

```
namespace MPINET { // High-level bindings
    class Communicator {
        // Constructors
        public uint Rank {get;}
        public uint Size {get;}
        public static Intracommunicator World {get;}
        ...
        public Port this[uint index] {get;}
    }

    class Port {
        // Constructors
        // Point-to-point communications
    }

    class Datatype {
        // Datatype constructors
    }

    interface IPinnable { ... }
        // Methods to handle pinnable buffers
    interface IBuffer: IPinnable { ... }
        // Untyped, pinnable byte arrays
    interface ITypedBuffer: IPinnable { ... }
        // Buffer with defined MPI data type, count
    class ArrayTypedBuffer: ITypedBuffer { ... }
        // ITypedBuffer implementation for arrays
    // Other message types (single elements,
    // serialized data, ...)

    class Status {
        // Constructors
        public uint Source {get;} // Also Tag, Error
        public static Status Ignore {get;}
    }

    class Group { ... }
    class Intracommunicator: Communicator { ... }
    class CartesianCommunicator: Intracommunicator
        { ... }
    class GraphCommunicator: Intracommunicator
        { ... }
    class Intercommunicator: Communicator { ... }
    class Errhandler { ... }
    class ReductionOperator { ... }
}
```

**Figure 6: Class hierarchy for MPI.NET**

## 5.2 Java bindings

There are also several interfaces from the Java programming language to MPI. The mpiJava interface appears to be similar to the C++ bindings, but allowing automatic serialization of Java objects to be sent using MPI [2]. It also claims to be restricted to ensure that message passing is safe (does not allow buffer overflows, etc.). Another interface from Java to MPI, MPIJ, is more like the C++ bindings in layout [6]. This library was implemented completely in Java, as opposed to the other libraries that are interfaces to existing MPI implementations. It seems to require explicit indications of what MPI data type to use for messages, but it appears to be able to automatically determine the number of elements to send or receive. The JavaMPI bindings, on the other hand, are based on the MPI C bindings, which were automatically converted to Java [12]. Overall, there are several bindings to MPI in Java, mostly either based on the C or C++ standard bindings.

## 5.3 Bindings from other languages

There are also bindings to MPI from other object-oriented languages, such as Python and Ruby. There are several Python interfaces to MPI, including MPI Python [11], which appears to be based on the C++ bindings, but with automatic serialization of Python objects. The Scientific Python library also includes a similar interface between Python and MPI [5]. There is also a binding from MPI to the Ruby language. Named "MPI Ruby," this interface includes automatic serialization and unserialization of Ruby objects [14]. It otherwise appears to be like the C++ bindings in terms of class hierarchy, etc.

# 6. PERFORMANCE RESULTS

A preliminary version of the C# bindings was tested to determine the overhead of using C# rather than C for MPI message passing. The test program was a ping-pong test in which one rank sends a message to the other, and the other rank sends it back. The times given are for one round-trip using this procedure. Each process modifies the message slightly after it has been received, but this was not found to affect the performance significantly. The test was run between two 1.5 GHz Pentium 4 systems connected by full-duplex 100 Mbps Ethernet. Both systems were running FreeBSD 4.5. The C compiler used was GCC 2.95.3, and the Rotor environment was used for C# [19]. LAM/MPI 6.5.6 was used for communications between the computers, and the communications were done using client-to-client mode and with the homogeneous-network optimization (which removes the check to determine if byte-swapping is necessary for messages).

All versions of the ping-pong test were similar: a buffer of double-precision floating point numbers was created and initialized on rank 0. Then, rank 0 sent the buffer to rank 1, which modified one element of it and sent it back. Rank 0 then modified another element and passed it back to rank 1, and this process was repeated. One run consisted of 2000 ping-pongs using this method, of which only the last 1000 were timed in order to minimize the effect of startup and just-in-time compilation on the results. One set of tests used the `MPI_Send` and `MPI_Recv` functions for communications (blocking mode), and another set used the persistent communication functions defined by the MPI standard.

These simple tests were used because they isolate the features of the C# bindings that directly affect performance. Applications were not tested since the goal of this benchmark was not to test the C and C# languages, but to test the performance of the MPI bindings to each language. That is also why the tests were performed using only two systems. The goal of the benchmarks was not to test network latency or operating system overheads during MPI communications. Thus, the simple two-computer ping-pong test provides an adequate measure of the C# MPI binding overheads for the send and receive MPI functions used.

Several versions of the ping-pong test were run. The `C bindings` implementation was written in C using MPI directly. The `CLI bindings` version was written in C#, and used the `CLI bindings` interface to MPI functions. This version uses buffers that are pinned throughout the length of the ping-pong test. The `P/Invoke` version also uses pinned buffers, and uses a set of MPI bindings to .NET that are direct wrappers of the C MPI functions (using the P/Invoke feature of .NET). This set of bindings does not have classes wrapping the underlying MPI types, and is designed for the best performance possible. The `MPI.NET` version was written using a set of C# bindings that are wrappers around the interfaces used by the `P/Invoke` version. The `MPI.NET with explicit pinning` version uses the same set of bindings, but with explicit buffer pinning by the test program (similar to the `CLI bindings` version).

The performance results for each implementation are given in Figures 7 and 8. As can be seen from these graphs, C# adds a significant amount of overhead (around 50 microseconds minimum). These overheads grow and become more erratic as the message size increases, but they still grow slowly compared to the communication times. If explicit buffer pinning by the test program is used, adding extra layers of wrapper classes around the MPI library does not add a large amount of extra overhead. Each extra method call (for instance, for bookkeeping in to ensure that buffers are pinned during communications) adds some overhead to the communication operations. Repeatedly pinning and unpinning message buffers for each send or receive operation caused a much larger overhead. These overheads did not grow as much as the communication times as message sizes increased, though, and the overheads are relatively very small for large message sizes.

Using persistent communications with each of the three .NET interfaces improved performance somewhat. Persistent communications in MPI.NET automatically keep the buffer pinned during the lifetime of the persistent request, and so that does not cause any overhead in this set of tests. However, the C test program using persistent communications was slightly slower than the one using non-persistent communications. The overheads for persistent communications were below 100 microseconds for messages less than 1000 bytes. A graph of the overheads for persistent communications with each interface is given in Figure 9.

Overall, the .NET interfaces to MPI add some overhead to communications, but this is not a very large part of the total message transmission time for large messages. However, a process sending many small messages will probably be greatly affected by them, especially if the message buffers are pinned separately for each send and receive operation. The overheads may also cause greater problems on faster and lower-latency networks, since the language overheads are not likely to improve based on the network technology in use. In all, any barriers to writing high-performance MPI programs using C# are not likely to be due to message-passing overheads.

# 7. CONCLUSION

This paper has presented a design for both low-level and high-level bindings from the C# programming language to the MPI interface for distributed-memory parallel computing. The low-level bind-

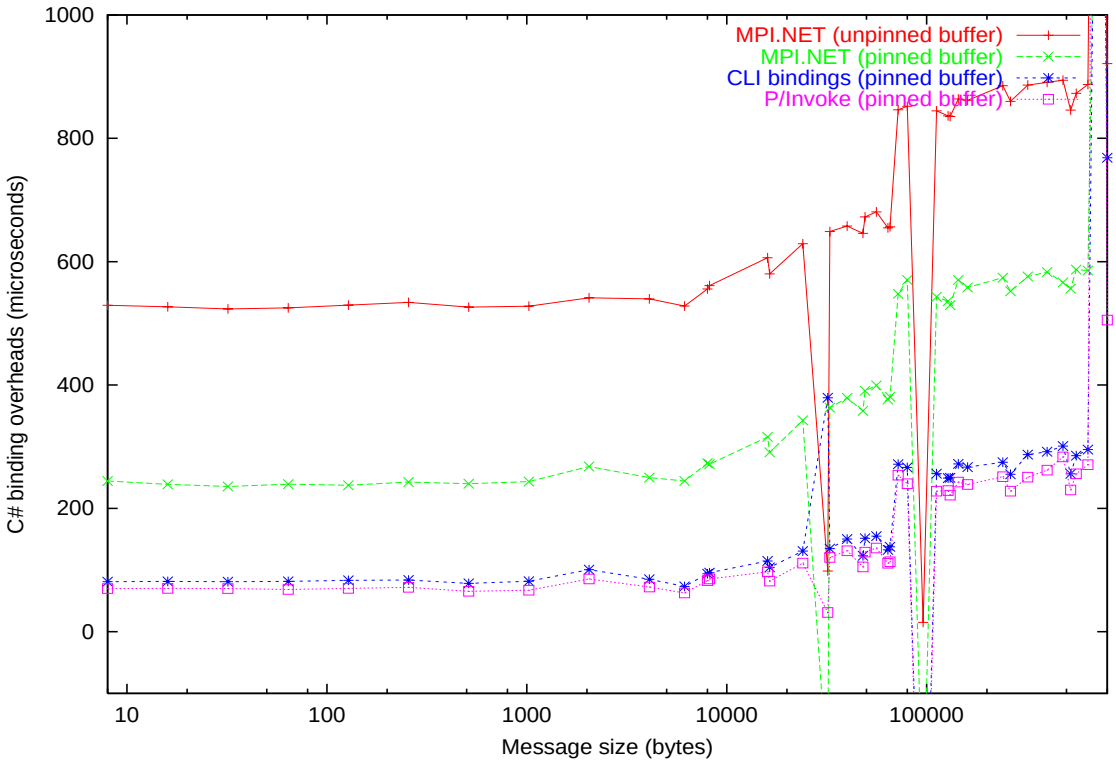


**Figure 7: C# binding overheads (relative to C bindings) by message size**

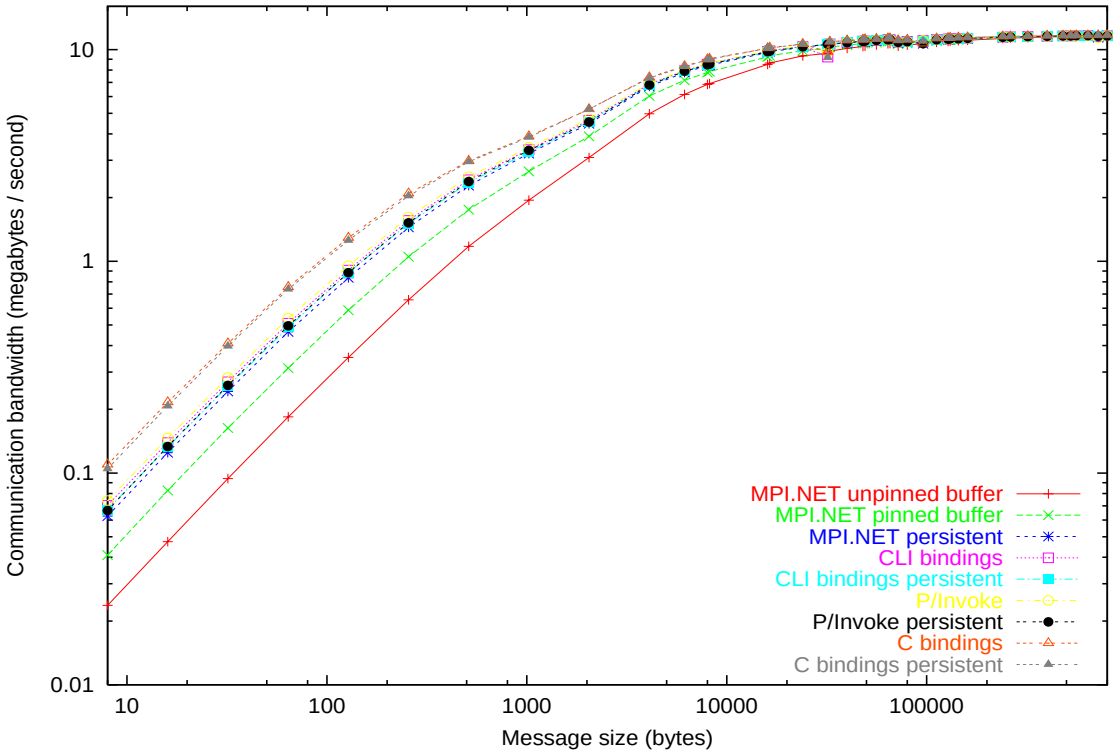


**Figure 8: C# binding bandwidths by message size**

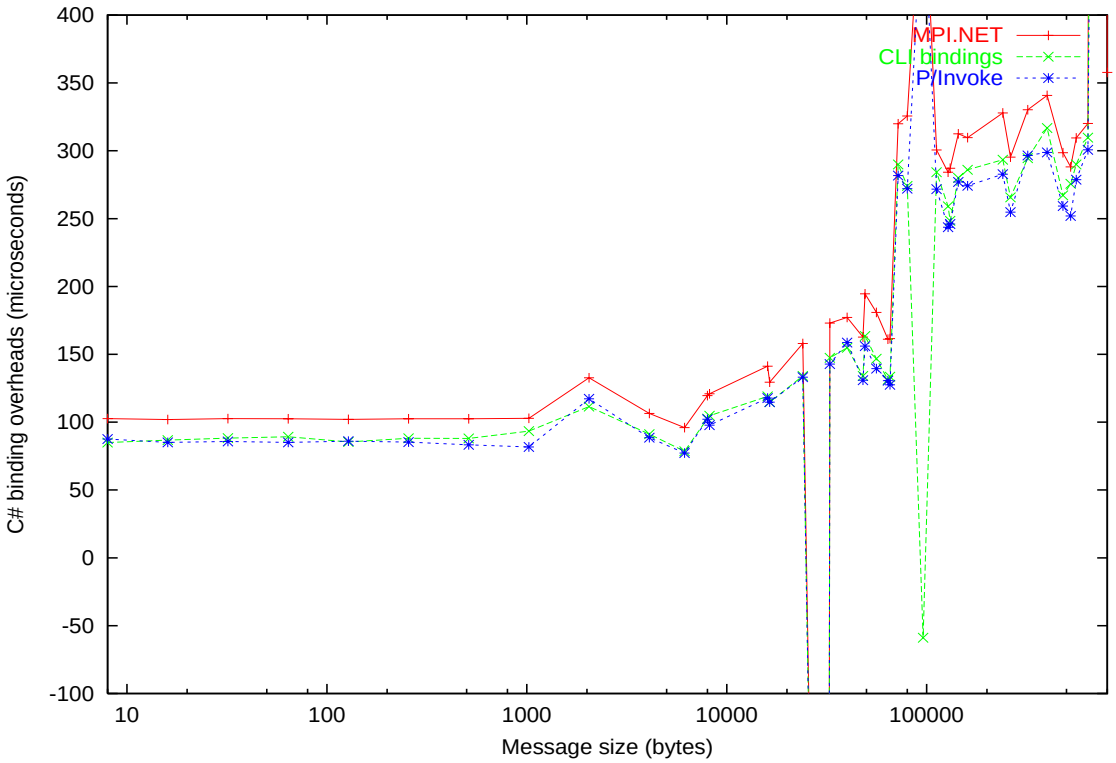


**Figure 9: C# binding overheads (relative to C bindings) by message size – persistent communications**

ings (CLI bindings) attempt to have a low overhead when compared to using MPI from C or C++, and have an interface very similar to the existing C++ bindings for MPI. The high-level bindings (MPI.NET), on the other hand, are based on the interface of the OOMPI library, which is a higher-level, simpler to use C++ object-oriented interface to MPI. The high-level bindings to C# try to follow the same conventions as the C# system libraries, and therefore be easier for C# programmers to learn. Both of these interfaces, however, try to follow MPI names for constants, data types, and functions as much as possible. This is intended to reduce the learning curve for existing MPI programmers to move to the C# bindings.

So far, most of the low-level C# bindings have been implemented. However, most of the implementation has not yet been tested. However, the subset shown in the example programs is known to work, and one LAM/MPI example program that uses a collective operation (`MPI_Reduce`) has been found to work. Many fewer of the high-level bindings have been implemented. Again, the example program given in this paper has been found to work correctly with the high-level bindings. The performance tests given in Section 6 were run using these implementations of the bindings.

The performance of the C# bindings has been fairly good when buffer pinning is not included in the C# overhead times. All buffers created by static allocation and the standard C and C++ memory allocation primitives are pinned, and C#'s overhead is low when this property is extended to its buffers. However, when the buffers used from C# are not already pinned, performance is significantly reduced. In all, the C# bindings' performance is not affected very much by adding more layers of abstraction. There is some abstraction penalty, though, and it appears that every extra method call used by the C# interface library adds some overhead.

The complete set of MPI bindings to .NET will be available at `http://www.lam-mpi.org/research/mpi-net/`.

## 8. ACKNOWLEDGMENTS

This work was supported by a grant from the Lilly Endowment and by NSF grant ACI-9982205.

## 9. REFERENCES

[1] D. Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. In *Proceedings of Java Grande/ISCOPE '01*, pages 68–77, Palo Alto, CA, June 2001.

[2] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim. mpiJava 1.2: API specification. Technical Report CRPC-TR99804, Rice University, Center for Research on Parallel Computation, September 1999.

[3] ECMA. *C# Language Specification*, December 2001. `http://www.ecma.ch/ecma1/STAND/ecma-334.htm`.

[4] ECMA. *Common Language Infrastructure (CLI)*, December 2001. `http://www.ecma.ch/ecma1/STAND/ecma-335.htm`.

[5] K. Hinsen. ScientificPython, January 2002. `http://-starship.python.net/crew/hinsen/scientific.html`.

[6] G. Judd, M. J. Clement, Q. Snell, and V. Getov. Design issues for efficient implementation of MPI in java. In *Java Grande*, pages 58–65, 1999.

[7] D. Kafura and L. Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI developers conference*, Notre Dame, IN, June 1995. http://www.cse.nd.edu/-mpidc95/proceedings/papers/html/huang/.

[8] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Snowbird, Utah, June 2001.

[9] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[10] Message Passing Interface Forum. MPI-2, July 1997. http://www.mpi-forum.org/.

[11] P. Miller and M. Casado. MPI Python. http://sourceforge.net/projects/pympi/.

[12] S. Mintchev. Writing programs in JavaMPI. Technical Report MAN-CSPE-02, School of Computer Science, University of Westminster, October 1997.

[13] J. Moreira, S. Midkiff, and M. Gupta. A comparison of three approaches to language, compiler, and library support for multidimensional array in Java. In *Proceedings of Java Grande/ISCOPE '01*, pages 166–125, Palo Alto, CA, June 2001.

[14] E. Ong. MPI Ruby – a Ruby binding of MPI. http://www-unix.mcs.anl.gov/mpi/mpi_ruby.

[15] W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, May 2000.

[16] A. Skjellum, Z. Lu, P. V. Bangalore, and N. E. Doss. Explicit parallel programming in C++ based on the message-passing interface (MPI). In G. V. Wilson, editor, *Parallel Programming Using C++*. MIT Press, 1996. Also available as MSSU-EIRS-ERC-95-7.

[17] J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object oriented MPI: A class library for the message passing interface. In *Parallel Object-Oriented Methods and Applications (POOMA '96)*, Santa Fe, 1996.

[18] J. M. Squyres, B. Saphir, and A. Lumsdaine. The design and evolution of the MPI-2 C++ interface. In *Proceedings, 1997 Internantional Conference on Scientific Computing in Object-Oriented Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[19] D. Stutz. The Microsoft shared source CLI implementation, March 2002. http://msdn.microsoft.com/library/-en-us/dndotnet/html/mssharsourcecli.asp.

[20] C. var Reeuwijk, F. Kuijlman, and H. J. Sips. Spar: a set of extensions to Java for scientific computation. In *Proceedings of Java Grande/ISCOPE '01*, pages 58–67, Palo Alto, CA, June 2001.

# APPENDIX
# A. CLI BINDINGS API LISTING

```
class MPI {
  delegate void ReductionOperation(IntPtr invec, IntPtr inoutvec, int len, Datatype datatype);
  delegate int AttributeCopyFunction(Comm oldcomm, int comm_keyval, object extra_state, IntPtr
attribute_val_in, IntPtr attribute_val_out, out bool flag);
  delegate int AttributeDeleteFunction(Comm comm, int comm_keyval, IntPtr attribute_val,
object extra_state);
  delegate void ErrorHandlerFunction(Comm comm, ref int errorcode);

  static void Abort([In] Comm comm, [In] int errorcode);
  static void Buffer_attach([In] byte[] buffer, [In] int Size);
  static IntPtr Buffer_detach([Out] out int Size);
  static void Compute_dims([In] int nnodes, [In] int ndims, [In][Out] int[] dims);
  static int Error_class([In] int errorcode);
  static void Finalize();
  static IntPtr Get_address([In] IntPtr location);
  static void Get_error_string([In] int errorcode, [Out] byte[] str, [Out] out int resultlen);
  static void Get_processor_name([In] byte[] name, [Out] out int resultlen);
  static void Init();
  static bool Initialized();
  static double Wtick();
  static double Wtime();

  class Status {
    uint Source {get;}
    uint Tag {get;}
    uint Error {get;}

    static Status CreateFromMPIStatus(IntPtr mpiStaticPtr);
    int Get_count([In] Datatype datatype);
    int Get_elements([In] Datatype datatype);
    bool Test_cancelled();
  }

  class StatusArray {
    readonly int Length;
    StatusArray(int len);
    ~StatusArray();
    static implicit operator IntPtr(StatusArray sa);
    Status this[int index] {get;}
  }

  class Comm: ICloneable {
    void Bsend([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int dest, [In]
int tag);
    Prequest Bsend_init([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    virtual object Clone();
    static int Compare([In] Comm comm1, [In] Comm comm2);
    static Errhandler Create_errhandler([In] ErrorHandlerFunction function);
    static int Create_keyval([In] AttributeCopyFunction copy_fn, [In] AttributeDeleteFunction
delete_fn, [In] object extra_state);
    void Delete_attr([In] int keyval);
    Comm Dup();
    void Free();
    static void Free_keyval([In][Out] ref int keyval);
    bool Get_attr([In] int keyval, [In][Out] ref object attribute_val);
    Errhandler Get_errhandler();
    Group Get_group();
    int Get_rank();
    int Get_size();
    int Get_topology();
```

```
    Request Ibsend([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    bool Iprobe([In] int source, [In] int tag);
    bool Iprobe([In] int source, [In] int tag, [Out] Status status);
    Request Irecv([Out] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    Request Irsend([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    bool Is_inter();
    Request Isend([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int destorsrc,
[In] int tag);
    Request Issend([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    void Probe([In] int source, [In] int tag);
    void Probe([In] int source, [In] int tag, [Out] Status status);
    void Recv([Out] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int source, [In]
int tag);
    void Recv([Out] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int source, [In]
int tag, [Out] Status status);
    Prequest Recv_init([Out] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    void Rsend([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int dest, [In]
int tag);
    Prequest Rsend_init([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    void Send([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int dest, [In] int
tag);
    Prequest Send_init([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
    void Sendrecv([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [In] int
dest, [In] int sendtag, [Out] IntPtr recvbuf, [In] int recvcount, [In] Datatype recvtype, [In]
int source, [In] int recvtag);
    void Sendrecv([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [In] int
dest, [In] int sendtag, [Out] IntPtr recvbuf, [In] int recvcount, [In] Datatype recvtype, [In]
int source, [In] int recvtag, [Out] Status status);
    void Sendrecv_replace([In][Out] IntPtr buf, [In] int count, [In] Datatype datatype, [In]
int dest, [In] int sendtag, [In] int source, [In] int recvtag);
    void Sendrecv_replace([In][Out] IntPtr buf, [In] int count, [In] Datatype datatype, [In]
int dest, [In] int sendtag, [In] int source, [In] int recvtag, [Out] Status status);
    void Set_attr([In] int keyval, [In] object attribute_val);
    void Set_errhandler([In] Errhandler errhandler);
    void Ssend([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int dest, [In]
int tag);
    Prequest Ssend_init([In] IntPtr buf, [In] int count, [In] Datatype datatype, [In] int
destorsrc, [In] int tag);
  }

  class Intracomm: Comm {
    void Allgather([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [Out]
IntPtr recvbuf, [In] int recvcount, [In] Datatype recvtype);
    void Allgatherv([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [Out]
IntPtr recvbuf, [In] int[] recvcounts, [In] int[] displs, [In] Datatype recvtype);
    void Allreduce([In] IntPtr sendbuf, [Out] IntPtr recvbuf, [In] int count, [In] Datatype
datatype, [In] Op op);
    void Alltoall([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [Out]
IntPtr recvbuf, [In] int recvcount, [In] Datatype recvtype);
    void Alltoallv([In] IntPtr sendbuf, [In] int[] sendcounts, [In] int[] sdispls, [In]
Datatype sendtype, [Out] IntPtr recvbuf, [In] int[] recvcounts, [In] int[] rdispls, [In]
Datatype recvtype);
    void Barrier();
    void Bcast([In][Out] IntPtr buffer, [In] int count, [In] Datatype datatype, [In] int root);
    int Cart_map([In] int ndims, [In] int[] dims, [In] bool[] periods);
    override object Clone();
```

```
    Intracomm Create([In] Group group);
    Cartcomm Create_cart([In] int ndims, [In] int[] dims, [In] bool[] periods, [In] bool
reorder);
    Graphcomm Create_graph([In] int nnodes, [In] int[] index, [In] int[] edges, [In] bool
reorder);
    Intercomm Create_intercomm([In] int local_leader, [In] Intracomm peer_comm, [In] int
remote_leader, [In] int tag);
    new Intracomm Dup();
    void Gather([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [Out] IntPtr
recvbuf, [In] int recvcount, [In] Datatype recvtype, [In] int root);
    void Gatherv([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [Out] IntPtr
recvbuf, [In] int[] recvcounts, [In] int[] displs, [In] Datatype recvtype, [In] int root);
    int Graph_map([In] int nnodes, [In] int[] index, [In] int[] edges);
    void Reduce([In] IntPtr sendbuf, [Out] IntPtr recvbuf, [In] int count, [In] Datatype
datatype, [In] Op op, [In] int root);
    void Reduce_scatter([In] IntPtr sendbuf, [Out] IntPtr recvbuf, [In] int[] recvcounts, [In]
Datatype datatype, [In] Op op);
    void Scan([In] IntPtr sendbuf, [Out] IntPtr recvbuf, [In] int count, [In] Datatype
datatype, [In] Op op);
    void Scatter([In] IntPtr sendbuf, [In] int sendcount, [In] Datatype sendtype, [Out] IntPtr
recvbuf, [In] int recvcount, [In] Datatype recvtype, [In] int root);
    void Scatterv([In] IntPtr sendbuf, [In] int[] sendcounts, [In] int[] displs, [In] Datatype
sendtype, [Out] IntPtr recvbuf, [In] int recvcount, [In] Datatype recvtype, [In] int root);
    Intracomm Split([In] int color, [In] int key);
  }

  class Graphcomm: Intracomm {
    override object Clone();
    new Graphcomm Dup();
    void Get_dims([Out] out int nnodes, [Out] out int nedges);
    void Get_neighbors([In] int rank, [In] int maxneighbors, [Out] int[] neighbors);
    int Get_neighbors_count([In] int rank);
    void Get_topo([In] int maxindex, [In] int maxedges, [Out] int[] index, [Out] int[] edges);
  }

  class Cartcomm: Intracomm {
    override object Clone();
    new Graphcomm Dup();
    int Get_cart_rank([In] int[] coords);
    void Get_coords([In] int rank, [In] int maxdims, [Out] int[] coords);
    int Get_dim();
    void Get_topo([In] int maxdims, [Out] int[] dims, [Out] bool[] periods, [Out] int[]
coords);
    void Shift([In] int direction, [In] int disp, [Out] out int rank_source, [Out] out int
rank_dest);
    Cartcomm Sub([In] bool[] remain_dims);
  }

  class Intercomm: Comm {
    override object Clone();
    new Graphcomm Dup();
    Group Get_remote_group();
    int Get_remote_size();
    Intracomm Merge([In] bool high);
  }

  class Datatype {
    Datatype Create_contiguous([In] int count);
    Datatype Create_hindexed([In] int count, [In] int[] blocklengths, [In] int[]
displacements);
    Datatype Create_hvector([In] int count, [In] int blocklength, [In] int stride);
    Datatype Create_indexed([In] int count, [In] int[] blocklengths, [In] int[] displacements);
    static Datatype Create_struct([In] int count, [In] int[] blocklengths, [In] int[]
```

```
displacements, [In] Datatype[] types);
    Datatype Create_vector([In] int count, [In] int blocklength, [In] int stride);
    void Pack([In] IntPtr inbuf, [In] int incount, [Out] IntPtr outbuf, [In] int outsize,
[In][Out] ref int position, [In] Comm comm);
    int Pack_size([In] int incount, [In] Comm comm);
    void Type_commit();
    void Type_free();
    int Type_size();
    void Unpack([In] IntPtr inbuf, [In] int insize, [In][Out] ref int position, [Out] IntPtr
outbuf, [In] int outcount, [In] Comm comm);
  }

  class Errhandler {
    void Free();
  }

  class Exception: System.Exception {
    readonly int ErrorCode;
    Exception(int ec);
    int Get_error_code();
    int Get_error_class();
    string Get_error_string();
  }

  class Group {
    static int Compare([In] Group group1, [In] Group group2);
    static Group Difference([In] Group group1, [In] Group group2);
    Group Excl([In] int n, [In] int[] ranks);
    void Free();
    Group Incl([In] int n, [In] int[] ranks);
    static Group Intersection([In] Group group1, [In] Group group2);
    Group Range_excl([In] int n, [In] int[] ranges);
    Group Range_incl([In] int n, [In] int[] ranges);
    int Rank();
    int Size();
    static void Translate_ranks([In] Group group1, [In] int n, [In] int[] ranks1, [In] Group
group2, [Out] int[] ranks2);
    static Group Union([In] Group group1, [In] Group group2);
  }

  class Op {
    void Free();
    Op Init([In] ReductionOperation function, [In] bool commute);
  }

  class Request {
    void Cancel();
    void Free();
    bool Test();
    bool Test([Out] Status status);
    static bool Testall([In] int count, [In][Out] Request[] requests);
    static bool Testall([In] int count, [In][Out] Request[] requests, [Out] Status[] statuses);
    static bool Testany([In] int count, [In][Out] Request[] requests, [Out] out int index);
    static bool Testany([In] int count, [In][Out] Request[] requests, [Out] out int index,
[Out] Status status);
    static int Testsome([In] int incount, [In][Out] Request[] requests, [Out] int[] indices);
    static int Testsome([In] int incount, [In][Out] Request[] requests, [Out] int[] indices,
[Out] Status[] statuses);
    void Wait();
    void Wait([Out] Status status);
    static void Waitall([In] int count, [In][Out] Request[] requests);
    static void Waitall([In] int count, [In][Out] Request[] requests, [Out] Status[] statuses);
    static int Waitany([In] int count, [In][Out] Request[] requests);
```

```
    static int Waitany([In] int count, [In][Out] Request[] requests, [Out] Status status);
    static int Waitsome([In] int incount, [In][Out] Request[] requests, [Out] int[] indices);
    static int Waitsome([In] int incount, [In][Out] Request[] requests, [Out] int[] indices,
[Out] Status[] statuses);
  }

  class Prequest {
    void Start();
    static void Startall([In] int count, [In][Out] Prequest[] requests);
  }
}
```

## B. MPI.NET API LISTING

```
using System;
using System.Runtime.InteropServices;
using System.Collections;

namespace IU_OSL {
  namespace MPICS {
    class ArrayTypedBuffer: ITypedBuffer {
      Datatype Type {get;}
      int Count {get;}
      IntPtr Location {get;}

      ArrayTypedBuffer(int[] buf);
      ArrayTypedBuffer(int[] buf, int len);
      ArrayTypedBuffer(double[] buf);
      ArrayTypedBuffer(double[] buf, int len);

      void Pin();

      void Unpin();
    }

    interface IPinnable {
      void Pin();    // Must be nestable
      void Unpin();
      IntPtr Location {get;}
    }

    interface IBuffer: IPinnable {
      int Length {get;}
    }

    interface ITypedBuffer: IPinnable {
      int Count {get;}
      Datatype Type {get;}
    }

    class Communicator : IDisposable, IObjectWithErrorHandler {
      readonly MPI_Comm MPICommunicator;

      static readonly Intracommunicator World;

      Communicator(MPI_Comm mpiComm);
      void Dispose();
      Port this[int r] {get;}
      Port NullProcess {get;}
      void Sendrecv(ITypedBuffer sendbuf, int destrank, int sendtag, ITypedBuffer recvbuf, int
srcrank, int recvtag, Status st);
      void Sendrecv(ITypedBuffer sendbuf, int destrank, int sendtag, ITypedBuffer recvbuf, int
srcrank, int recvtag);
      void Sendrecv_replace(ITypedBuffer buf, int destrank, int sendtag, int srcrank, int
```

```
recvtag, Status st);
      void Sendrecv_replace(ITypedBuffer buf, int destrank, int sendtag, int srcrank, int
recvtag);
      Group CommunicatorGroup {get;}
      int Rank {get;}
      int Size {get;}
      static int Compare(Communicator a, Communicator b);
      bool IsIntercommunicator {get;}
      int Topology {get;}
      Errhandler ErrorHandler {get; set;}
    }

    class Datatype : IDisposable {
      MPI_Datatype MPIDatatype {get;}

      int Extent {get;}
      int LowerBound {get;}
      int UpperBound {get;}
      int Size {get;}

      Datatype(MPI_Datatype datatype);
      void Dispose();
      ~Datatype();
      Datatype CreateContiguous(int count);
      Datatype CreateVector(int count, int blocklength, int stride);
      Datatype CreateHVector(int count, int blocklength, int stride);
      Datatype CreateIndexed(int[] blocklengths, int[] displacements);
      Datatype CreateHIndexed(int[] blocklengths, int[] displacements);
      static Datatype CreateStruct(int[] blocklengths, int[] displacements, Datatype[] types);
      void Commit();

      static readonly Datatype Char;
      static readonly Datatype Short;
      static readonly Datatype Int;
      static readonly Datatype Long;
      static readonly Datatype UnsignedChar;
      static readonly Datatype UnsignedInt;
      static readonly Datatype UnsignedLong;
      static readonly Datatype Float;
      static readonly Datatype Double;
      static readonly Datatype LongDouble;
      static readonly Datatype Byte;
      static readonly Datatype Packed;
      static readonly Datatype FloatInt;
      static readonly Datatype DoubleInt;
      static readonly Datatype LongInt;
      static readonly Datatype TwoInt;
      static readonly Datatype ShortInt;
      static readonly Datatype LongDoubleInt;
      static readonly Datatype LongLong;
      static readonly Datatype UB;
      static readonly Datatype LB;
      static readonly Datatype Null;
    }

    class Errhandler : IDisposable {
      readonly MPI_Errhandler MPIErrhandler;
      readonly bool IsErrorsThrowExceptions;

      Errhandler(MPI_Errhandler eh);

      void Dispose();
```

```
  delegate void CommunicatorErrorHandlerFunction(Communicator comm, ref int errorcode);

  static readonly Errhandler ErrorsThrowExceptions;
  static readonly Errhandler ErrorsReturn;
  static readonly Errhandler ErrorsAreFatal;
  static readonly Errhandler Null;
}

class Exception : System.Exception {
  readonly int ErrorCode;
  int ErrorClass {get;}
  string ErrorString {get;}
}

class Group : IDisposable {
  readonly MPI_Group MPIGroup;

  Group(MPI_Group g);
  void Dispose();
  int Size {get;}
  int Rank {get;}
  void TranslateRanks(Group g1, int[] ranks1, Group g2, int[] ranks2);
  int Compare(Group g1, Group g2);
  static Group operator&(Group g1, Group g2);
  static Group operator|(Group g1, Group g2);
  static Group operator-(Group g1, Group g2);
  Group Incl(int[] ranks);
  Group Excl(int[] ranks);
  Group RangeIncl(int[] ranges);
  Group RangeExcl(int[] ranges);
  static readonly Group Null;
  static readonly Group Empty;

class Intracommunicator: Communicator {
  // To be filled in
}

class Port : IDisposable {
  readonly Communicator Comm;
  readonly int Peer;

  Port(Communicator comm, int peer);
  void Dispose();
  void Send(ITypedBuffer buf, int tag);
  void Send(ITypedBuffer buf);
  void Recv(ITypedBuffer buf, int tag);
  void Recv(ITypedBuffer buf, int tag, Status st);
  void Recv(ITypedBuffer buf);
  void Recv(ITypedBuffer buf, Status st);
  void Bsend(ITypedBuffer buf, int tag);
  void Bsend(ITypedBuffer buf);
  void Ssend(ITypedBuffer buf, int tag);
  void Ssend(ITypedBuffer buf);
  void Rsend(ITypedBuffer buf, int tag);
  void Rsend(ITypedBuffer buf);
  Request Isend(ITypedBuffer buf, int tag);
  Request Isend(ITypedBuffer buf);
  PersistentRequest SendInit(ITypedBuffer buf, int tag);
  PersistentRequest SendInit(ITypedBuffer buf);
  PersistentRequest RecvInit(ITypedBuffer buf, int tag);
  PersistentRequest RecvInit(ITypedBuffer buf);
}
```

```
    class Request : IDisposable {
      MPI_Request MPIRequest {get;}
      virtual void Dispose();
      bool Completed {get;}
      void Wait();
      void Wait(Status st);
      bool Test();
      bool Test(Status st);
      void Cancel();
    }

    class PersistentRequest : Request {
      override void Dispose();
      void Start();
    }

    class Session : IDisposable {
      Session();
      void Dispose();
      bool IsInitialized {get;}
      void Abort(Communicator comm, int errorcode);
    }

    class Status : IDisposable {
      Status();
      void Dispose();
      ~Status();
      int Source {get;}
      int Tag {get;}
      int Error {get;}
      static readonly Status Ignore;
    }
  }
}
```