Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure

Douglas Gregor and Andrew Lumsdaine
Indiana University
{dgregor, lums}@osl.iu.edu

Abstract

As high-performance computing enters the mainstream, parallel programming mechanisms (including the Message Passing Interface, or MPI) must be supported in new environments such as C# and the Common Language Infrastructure (CLI). Making effective use of MPI with the CLI requires an interface that reflects the high-level object-oriented nature of C# and that also supports its programming idioms. However, for performance reasons, this high-level functionality must ultimately be mapped to low-level native MPI libraries. In addition to abstraction penalty concerns, avoiding unwanted overhead in this mapping process is significantly complicated by the safety and portability features of the CLI virtual machine, such as garbage collection and just-in-time compilation. In this paper, we describe our approach to using features of C# and the CLI—such as reflection, unsafe code regions, and run-time code generation—to realize an elegant, yet highly efficient, C# interface to MPI. Experimental results demonstrate that there is no appreciable overhead introduced by our approach when compared to the native MS-MPI library.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages, Performance

Keywords Generic programming, message passing interface, C#

1. Introduction

The Message Passing Interface (MPI) is the *de facto* standard for writing high-performance message-passing applications. To support the principal languages used in high-performance computing at the time of its design, MPI specifies language bindings for C, Fortran, and C++. However, significant changes have occurred in the computing landscape in the years since MPI was originally developed. On the one hand, hardware technologies that were once in the specialized domain of high-performance computing are now entering the mainstream. On the other hand, software development in the mainstream is dominated by high-level, high-productivity languages and programming paradigms such as object-oriented and generic programming.

An important example of a modern software development and execution environment is C# and the Common Language Infrastructure (as implemented in Microsoft's .NET). C# is a high-level, object-oriented, garbage-collected language that executes within the CLI virtual machine. This environment is radically different from the C and Fortran environment for which MPI was designed, so simply using existing MPI language bindings represents a cognitive (as well as technological) mismatch. To be effective, an MPI library for C# must provide an interface that supports not only the C# language but also its foundational programming paradigms. For example, the interface must naturally support the transmission of objects as well as primitive types (MPI only directly supports the latter), and should avoid pointers, which are present throughout the low-level MPI interface but are generally eschewed in C# programs. Moreover, such a library must be highly efficient—ideally, as efficient as a native MPI library. Achieving the combined goals of elegance and efficiency would be complicated in any case but is further exacerbated by features of the CLI virtual machine such as garbage collection and just-in-time compilation.

We have recently developed MPI.NET [6], a C# library that provides a simple, intuitive, and complete MPI for C# and other languages that execute under the CLI. MPI.NET's interfaces typically require far fewer parameters than the associated C interface, eliminating significant redundancy and the potential for a number of common user errors. Even so, these interfaces are more flexible than the corresponding C interface, providing complete support for user- and library-defined types. For example, the datatype and array-length arguments are removed from the C# equivalent of MPI_Send, because they can be inferred from the other parameters. Moreover the same interface applies to all datatypes, from primitive types like integers to arbitrary user-defined classes such as employee records. Just like the C MPI interface provides a suitable level of abstraction that is intuitive for C programmers, MPI.NET provide a high-level interface intuitive for C# programmers. This approach stands in contrast to the standardized MPI C++ bindings [13] and to the Java bindings [15], which provide access to MPI from higher-level languages but do not support the common abstractions of those languages (e.g., classes, generics).

The second, and equally important, goal of this research is to ensure that, despite high-level and intuitive interfaces, MPI.NET still provides performance on par with existing, lower-level interfaces to MPI. Like many efforts to build MPI interfaces from a higher-level language, we build MPI.NET on top of the existing C MPI interface. Unlike most of these implementations, we provide a high-level interface whose implementation optimizes the use of the underlying MPI library based on knowledge of the specific data types and operations supplied by the user. For example, the Send routine applied to an object (say, a string) will automatically serialize the data for transmission and deserialize it on receipt, while the same Send routine applied to a primitive integer will use MPI's MPI_INT datatype without producing any additional copies of the data. Using several key features of C#, including value types, generics, P/Invoke, unsafe code regions, reflection, and run-time code generation, MPI.NET aggressively optimizes the use of the underlying MPI library to provide native-language performance from a virtual machine.

PPOPP '08, February 20–23, 2008, Salt Lake City, Utah, USA.

In this paper, we describe the design and implementation of MPI.NET, covering the most important aspects of MPI 1.1, including point-to-point operations (Section 3) and collective operations (Section 4). Within each section, we present the MPI interfaces that capture the essence of MPI for C# and discuss relevant implementation details, translating the high-level concepts of C# into efficient uses of the underlying MPI library. The techniques presented in this paper will also be useful in optimizing other high-level interfaces to low-level libraries, and (we hope) will lead to better acceptance of high-level languages in the high-performance computing community. In Section 5, we provide an evaluation of the *abstraction penalty* of MPI.NET, measuring the cost of our high-level abstractions against the performance that one could attain by using a lower-level (albeit less intuitive) interface.

2. C# for High-Performance Computing

C# is a high-level, object-oriented language that executes on the CLI virtual machine. Like Java, C# provides many features that make programming easier than in lower-level languages like C or Fortran, such as bounds checking and garbage collection (to eliminate many program bugs), along with cross-platform interoperability (to simplify software distribution). C# includes additional features useful for scientific computing, such as multi-dimensional arrays and operator overloading. Three specific features of C# and the CLI are particularly important for the development of a high-performance MPI library:

2.1 Value types.

C# provides support for lightweight *value types*, which are simple types that do not require heap allocation and are always copied by value. C#'s primitive types are value types, as are structures containing value types. For example, we could choose to express a point as either a struct or a class:

```
public struct Point {
  public float x;
  public float y;
}
```

```
[Serializable] public class Point {
  public float x;
  public float y;
}
```

The Point structure on the left is a value type. Thus, instances of the Point structure will be stored on the stack (for local variables and parameters) or inlined into object representations, rather than allocated as separate objects on the heap. In fact, the expression new Point(), despite the use of the keyword new, does not allocate memory on the heap—it merely creates a new Point value on the stack. The Point class on the right, on the other hand, is a class. Classes in C# are *reference types*, whose objects will always be allocated on the garbage-collected heap. Value types such as structures simplify compiler optimizations and reduce the number of objects that must be processed by the garbage collector, and therefore result in more efficient code. Since many common entities in scientific computing, such as complex numbers and rotation matrices, can be represented as value types, the C# compiler and runtime can provide additional optimizations not available for reference types.

2.2 Generics

Introduced in C# 2.0, generics allow the expression of type-parameterized classes and methods. One major feature of C# generics that makes a highly-efficient C# MPI possible is that type parameters can be value types (including primitive types, which are not permitted in Java generics). When a generic class or method is used with a reference type, C# treats all type arguments like the built-in object type. When a C# generic class or method is used with a value type, the CLI runtime synthesizes a new version of that class or method specialized to that particular value type [11]. The just-in-time compiler can allocate the appropriate stack frames and generate new structure definitions based on the size of the value type, then the optimizer can perform its usual set of optimizations for value types. No additional boxing (where a value is allocated on the heap) or unboxing (where a value is extracted from an object on the heap) is required when using generics with value types, so using generics with value types is as efficient as writing non-generic code that makes direct use of those value types.

```
public class Unsafe {
  [DllImport("msmpi.dll")]
  public static unsafe extern int
  MPI_Send(IntPtr buf, int count, MPI_Datatype datatype,
           int dest, int tag, MPI_Comm comm);
}
```

Figure 1. C# interface to the unmanaged (native) MPI_Send routine from Microsoft's MPI implementation.

MPI data type	CLI	C#
MPI_BYTE	Byte	byte
MPI_SHORT	Int16	short
MPI_INT	Int32	int
MPI_LONG_LONG_INT	Int64	long
MPI_FLOAT	Single	float
MPI_DOUBLE	Double	double

Table 1. Selected MPI predefined data types and their equivalents in CLI and C#. We have chosen to give the appropriate mappings for Microsoft's 32- and 64-bit platforms.

2.3 Interoperability

Both C# and Java provide virtual machines that support the interaction among multiple languages, including languages like C and Fortran that live *outside* the virtual machine. We use this interoperability to build MPI.NET on top of the existing, low-level MPI. C# provides a direct binding from the virtual machine (running "managed" code) to native ("unmanaged") code via its Platform Invoke (P/Invoke) interface. Figure 1 illustrates how one can express the interface to MPI_Send in C#. Here, extern indicates that the routine is external (i.e., in a different .NET assembly), while the DllImport attribute states that the actual code will reside in the shared library msmpi.dll, part of Microsoft's MPI. The unsafe specifier states that this routine can only be called from code explicitly marked "unsafe", and the IntPtr type used for the buffer argument stores a native-sized pointer that acts similar to the C void*.

3. Point-to-point Communication

MPI's point-to-point communications permit the transmission of data from one process to another. Each message is transmitted via a given *communicator*. Each communicator provides a separate communication space, so that different libraries within the same application can communicate over MPI without interfering with each other. In addition to the actual message contents, each message also contains an integer message tag, which allows users to more precisely specify matching of messages. The message data is typed using one of MPI's predefined data types (shown in Table 1) or an MPI *derived* datatype, which can represent user-defined structures.

The following example illustrates a typical use of the MPI send primitive in C:

```
int my_value = get_local_value();
MPI_Send(&my_value, 1, MPI_INT, dest, tag,
         MPI_COMM_WORLD);
```

```
public void Send<T>(T value, int dest, int tag) {
  using (MemoryStream stream = new MemoryStream()) {
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(stream, value);
    unsafe {
      fixed (byte* buffer = stream.GetBuffer()) {
        Unsafe.MPI_Send(new IntPtr(buffer),
          (int)stream.Length, Unsafe.MPI_BYTE,
          dest, tag, comm);
      }
    }
  }
}
```

Figure 2. Implementation of an MPI send in C#, using serialization and MPI's low-level operations. comm is a low-level MPI communicator.

Using the P/Invoke interface and unsafe code, we can directly invoke the MPI_Send routine shown in Figure 1 from C#, providing a C-like interface:

```
int myValue = getLocalValue();
unsafe {
  Unsafe.MPI_Send(new IntPtr(&myValue), 1,
                  Unsafe.MPI_INT, dest, tag,
                  Unsafe.MPI_COMM_WORLD);
}
```

While it is possible to use such an interface in C#, and it will certainly be necessary as an implementation mechanism, there are several aspects of this code that do not fit well with the C# development model. The explicit use of the & operator to take the address of a variable requires the use of an unsafe block in C# (which end users of MPI should not be exposed to), and, in most cases, also forces the user to explicitly "pin" the memory associated with the communication, so that the garbage collector will not relocate the data. The division of the message data into three arguments—a pointer, a count of the number of elements, and a data type indicator—violates the principle of data encapsulation and is inherently unsafe. Without encapsulation, users may inadvertently state an incorrect number of values (e.g., causing reads of uninitialized memory) or provide an improper data type (e.g., causing data type translation errors). The result returned from MPI_Send is an MPI error code that the user must manually check, whereas in C#, errors are typically reported via exceptions.

An ideal C# interface for point-to-point operations should be far simpler than the existing C or Fortran interface, eliminating the redundant information from the message data and permitting the transmission of any type of data. For example, one should be able to simply send the result of getLocalValue(), without specifying type information, and the same code should work regardless of the type returned by the function:

```
world.Send(getLocalValue(), dest, tag);
```

Throughout this section, we will explore the implementation and optimization of this simplified interface.

3.1 Serialization, Pinning, and User-defined Types

The generic implementation of the simplified Send operation requires that we treat all types, including primitive, library-defined, and user-defined types, equally from the user's perspective. The simplest implementation for all types requires serialization of the object. Figure 2 illustrates how to serialize any C# object and transmit the data via MPI. The most interesting C# feature in use is the fixed statement, which pins the memory associated with the transmission buffer within its scope, preventing the garbage collector from moving the buffer while the low-level MPI is accessing it.

```
public void Receive<T>(int source, int tag, out T value) {
  MPI_Status status;
  int length;
  unsafe {
    Unsafe.MPI_Probe(source, tag, comm, &status);
    Unsafe.MPI_Get_count(&status, Unsafe.MPI_BYTE,
                         &length);
  }
  using (MemoryStream stream = new MemoryStream()) {
    stream.setLength(length);
    BinaryFormatter formatter = new BinaryFormatter();
    unsafe {
      fixed (byte* buffer = stream.GetBuffer()) {
        Unsafe.MPI_Recv(new IntPtr(buffer), length,
          Unsafe.MPI_BYTE, status.MPI_SOURCE,
          status.MPI_TAG, comm,
          Unsafe.MPI_STATUS_IGNORE);
      }
    }
    value = (T)formatter.Deserialize(stream);
  }
}
```

Figure 3. Implementation of an MPI receive in C#, using serialization and MPI's low-level operations.

Figure 3 illustrates the receive operation that corresponds to the send in Figure 2. Here, we probe for the next message that matches the $(source, tag)$ criteria, allocate a buffer into which we receive the serialized data, and, finally, de-serialize and cast the result. We note that this particular implementation is not safe in a multi-threaded environment, but delay the discussion of this issue until Section 3.5.

3.2 Value types

The initial implementation of a high-level interface to MPI's send operation in Figure 2 has some interesting performance implications. On the positive side, for data types that require serialization (containers, strings, etc.), the data is serialized once into a buffer and is transmitted without requiring any explicit copying from the "managed", garbage-collected heap into an unsafe buffer, because a pointer into the garbage-collected memory is passed directly to the lower-level MPI. However, for primitive types this implementation results in unacceptable overhead. First of all, since the Serialize method used to serialize the data accepts a value of type object, the value passed by the user will be implicitly boxed. Then, through several method calls and additional object constructions, the primitive type will be serialized into a newly-allocated buffer. At the receiver's end, we undergo the same overhead for the reverse process, probing for and receiving a message, de-serializing into an object, and unboxing the result. This entire process is far less efficient than the direct, albeit unwieldy, interface to MPI via P/Invoke.

Ideally, Send would use the most efficient low-level MPI operations for the data type given. A string should be serialized, while an integer value or a floating-point value (or a fixed-size array of such values) should be sent with the appropriate MPI datatype. To implement this optimization, we employ run-time type checks to determine whether the type is a primitive type, then map primitive types to their associated MPI datatype. Figure 4 illustrates how this mapping might be used to improve the performance of Send.

3.2.1 Pinning Value Types

Unfortunately, the Send implementation in Figure 4 is not valid C# code. C# does not permit the creation of pointers to a type parameter such as T, because T might not be a value type. Unfortunately, this limitation persists even if T is restricted to a value type.

```
public void Send<T>(T value, int dest, int tag) {
  if (typeof(T).IsPrimitive) {
    MPI_Datatype datatype;
    if (typeof(T) == typeof(int))
      datatype = Unsafe.MPI_INT;
    else if (typeof(T) == typeof(uint))
      datatype = Unsafe.MPI_UNSIGNED;
    else ...
    unsafe {
      Unsafe.MPI_Send(new IntPtr(&value), 1, datatype,
                      dest, tag, comm);
    }
  } else {
    // Serialize and send
  }
}
```

Figure 4. Using run-time type information to efficiently transmit primitive types via MPI. Note that this code will not compile in this form due to C#'s limitations on generic types.

One immediate workaround is to use the GCHandle class from the System.Runtime.InteropServices namespace, which allows one to pin an object in memory and retrieve its address. For example:

```
GCHandle handle
  = GCHandle.Alloc(value, GCHandleType.Pinned);
unsafe {
  Unsafe.MPI_Send(handle.AddrOfPinnedObject(), 1,
                  datatype, dest, tag, comm);
}
handle.Free();
```

With this implementation, we have solved most of the performance problems with the serialization-only approach of Figure 2: primitive types are sent directly using the most appropriate MPI datatype. However, the GCHandle.Alloc method accepts an object as its first parameter, implicitly boxing the primitive type before pinning and transmitting it. We have introduced several new branches into the implementation of Send, but these branches rely on values known when the type parameter T is known. Thus, when the CLI runtime creates a specialized implementation of Send for a particular primitive type, the exact MPI datatype will be known immediately.

To remove the overhead due to boxing in GCHandle.Alloc, we must step outside the bounds of C# into a language with a more relaxed type system. The Common Intermediate Language (or CIL, formerly called MSIL) is the assembly language of the CLI, to which languages such as C# and Visual Basic are compiled. Code written in CIL can be linked with and used directly from C#. We use CIL to extract the address of an existing value whose type is a type parameter. Figure 5 contains the LoadAddress routine, that accepts a value of any type and returns its address as a system pointer. Within C#, the signature of this method is:

```
public static IntPtr LoadAddress<T>(ref T value);
```

Using this LoadAddress<T> routine, we can finally implement the low-level call to MPI_Send without any boxing for primitive types:

```
unsafe {
  Unsafe.MPI_Send(MPIUtils.LoadAddress(ref value), 1,
                  datatype, dest, tag, comm);
}
```

At this point, we have a Send routine that provides a unified interface for users that supports all types equally well. Arbitrary class types will be automatically serialized before transmission, while primitive types will be transmitted directly via MPI's low-level facilities, without any additional copies or memory allocation.

```
.method public hidebysig static native int
    LoadAddress<T>(!!T& 'value') cil managed {
  // Code size 14 (0xe)
  .maxstack 1
  .locals init ([0] !!T& pinned ptr)
  ldarg.0
  stloc.0
  ldloc.0
  conv.i
  newobj instance void [mscorlib]System.IntPtr::.ctor(void*)
  ret
}
```

Figure 5. Loading the address of an arbitrary value type in CIL.

```
int[] blocklens = new int[] { 1, 1 };
MPI_Aint[] indices = new MPI_Aint[] {
  new MPI_Aint(Marshal.OffsetOf(typeof(Point).GetField("x"))),
  new MPI_Aint(Marshal.OffsetOf(typeof(Point).GetField("y")))
};
MPI_Datatype[] datatypes = new MPI_Datatype[] {
  Unsafe.MPI_FLOAT, Unsafe.MPI_FLOAT
};
MPI_Datatype datatype;
unsafe {
  fixed (int* blocklensPtr = blocklens) {
    fixed (MPI_Aint* indicesPtr = indices) {
      fixed (MPI_Datatype* typesPtr = types) {
        Unsafe.MPI_Type_struct(2, blocklensPtr, indicesPtr,
                               typesPtr, &datatype);
      }
    }
  }
  Unsafe.MPI_Type_commit(&datatype);
}
```

Figure 6. Constructing an MPI datatype for the Point structure.

3.2.2 User-defined Value Types

Beyond the primitive types, value types in the CLI can be user-defined and library-defined structures, such as the Point structure in Section 2. These value types enjoy the same performance benefits of primitive types in the CLI, but there are no corresponding predefined MPI datatypes for them. Thus, the default handling of such datatypes would be to serialize them just as strings or containers are serialized.

MPI's support for *derived datatypes* presents another option for handling structure types like Point. Using the low-level MPI_Type_struct function, we can describe the fields of the Point type to MPI, which will create a new MPI_Datatype that we can use to directly send Point values, without serialization. Figure 6 summarizes the construction of an MPI datatype for Point, which requires three arrays: the first states how many elements are in each field (> 1 represents an array of elements of the same type), the second provides the offsets to each field in the structure, as reported by the C# Marshal class, and the final array states the MPI datatypes of each field. Passing fixed pointers to the elements in these arrays to MPI_Type_struct constructs the datatype, which is finished by MPI_Type_commit. Once these operations have completed, Points can be transmitted directly via MPI, with no need for serialization or extra memory copies.

Forcing users of MPI.NET to manually produce MPI datatype-generation code would severely hamper efforts to provide an easy-to-use MPI interface, and it is likely that few users would go through such an effort. Instead, MPI.NET automatically constructs MPI derived datatypes for each of the CLI value types it encounters,

```
public void Send<T>(T[] values, int dest, int tag) {
  if (typeof(T).IsValueType) {
    GCHandle handle
      = GCHandle.Alloc(values, GCHandleType.Pinned);
    unsafe {
      Unsafe.MPI_Send
       (handle.UnsafeAddrOfPinnedArrayElement(values,0),
        values.length, FastDatatypeCache<T>.datatype,
        dest, tag, comm);
    }
    handle.Free();
  } else {
    // Serialize array and send
  }
}
```

Figure 7. Transmission of arrays via MPI using run-time type information.

using the CLI's extensive reflection capabilities. MPI.NET walks the fields of the value type, building the three arrays needed by MPI_Type_struct. The resulting MPI derived datatypes are stored in a datatype cache, which manages the mappings from all value types to MPI datatypes. The datatype cache replaces the cascading if statements of the Send in Figure 4, providing seamless support for efficient transmission of CLI value types. Our final Send operation, therefore, only requires serialization for reference types, providing transparent optimization for all of the CLI value types, from primitive types to user-defined structures.

3.3 Arrays

The transmission of array data via MPI is similar to the transmission of single values. For arrays of reference types, we need only serialize the array at the sender's side and deserialize it on the receiver's side. For arrays of value types, we can compute the MPI datatype associated with that value type and send the array as a single block of data. Figure 7 illustrates the array-send operation, which takes care of pinning the memory associated with the array prior to transmitting the array of values via MPI. Note that, unlike the single-element case where the GCHandle caused implicit boxing, the pinning occurs on the actual array object (which is already a reference type). The handle is then used to extract the address of the first element in the now-pinned array, which is then passed to the lower-level MPI implementation.

3.4 Non-Blocking Communication

MPI's non-blocking communication allows users to initiate a communication operation (e.g., a send or receive) that can be completed in the background. Users can query whether a communication has completed or wait until one or more communication requests have completed. Non-blocking communication makes it possible to overlap communication with computation, potentially improving performance, and makes it possible to safely perform send-to-self operations.

Non-blocking communication presents some additional challenges in C#, which are best illustrated with the non-blocking counterpart to the Receive operation in Figure 3. The value parameter of Receive is specified as an out parameter. In C#, out parameters are, in effect, pointers to variables or fields, allowing one to replace a reference to an object with a reference to a different object. Thus, the Receive operation is able to directly receive into a variable supplied by the caller. Therefore, one might expect the non-blocking receive to provide a similar signature to the (blocking) Receive, with the addition of a Request object that provides the user with the ability to query the status of the communication:

```
public Request IReceive<T>(int source, int tag, out T value);
```

However, this approach is not workable in the C# language. At issue is the inability to store a pointer to a variable or field in C#. With the Receive operation in Figure 3, the out parameter value allowed MPI to receive directly into a variable or field designated by the caller of Receive, potentially replacing that value with an object of a different (dynamic) type. At an implementation level, an out parameter passes a pointer to its argument, allowing direct modification of the argument.

With non-blocking communication in C#, the memory associated with value will need to be pinned until the request itself is completed. However, C# does not provide a mechanism with which one can store an out parameter within the request object, because doing so could involve referencing a variable stored on the stack even after its associated stack frame has been destroyed. Thus, instead of attempting to receive into a user-provided variable, MPI.NET allocates storage as part of the request object on the heap, and receives into that storage. Once communication has completed, the user extracts the received value from the request object. The final signature of the IReceive operation in MPI.NET is:

```
public Request IReceive<T>(int source, int tag);
```

3.5 Transmitting Serialized Messages

As previously noted, the Receive operation shown in Figure 3 is not safe in multi-threaded programs. It is possible for two concurrent receive operations to probe the same message, but only one of those operations can successfully receive the message, leaving the other to fail.

In fact, any scheme that relies on an MPI probe operation to determine the length of a message before receiving it will fail in a multi-threaded context. Therefore, MPI.NET introduces a special high-level protocol that permits concurrent receive operations based on the existing low-level MPI interface. Each serialized send is split into two messages. First, a fixed-size header, which contains both the size of the "data" message and a unique tag value (assigned by MPI.NET), is sent with the tag supplied by the user. Then, a second data message containing the serialized representation is sent over a special "shadow" communicator with the unique tag stored in the fixed-size header. On the receiving end, the receiver will receive the fixed-size header first. From the information in the header, it can allocate a receive buffer and will receive the message directly from the shadow communicator.

The unique tag and "shadow" communicator are essential to restoring thread-safety when sending serialized objects. The unique tag ensures that no other message will have the same tag as this serialized data; since MPI.NET may need many such tags for transmitting objects (and cannot use any tag available to the user), MPI.NET maintains a separate "shadow" communicator that has the same processes and ranks as its primary communicator, but is reserved only for serialized data messages. Thus, these serialized messages are completely hidden from the user. However, the number of tags available for a given communicator is limited, which both limits the number of messages in flight at any particular time and, if tags are not reused, limits the total number of messages that can be sent over a communicator. Therefore, MPI.NET will re-use a unique tag for serialized data once the receiver of the message has matched the message and initiated the receive operation, using the native MPI's synchronous send operation [12, §3.4].

MPI.NET's protocol for transmitting messages of arbitrary length via the native MPI interface introduces significant overhead to the already-expensive point-to-point operations for serialized data, due to the higher message volume and the use of synchronous-mode communication. However, the native MPI's inability to receive messages of unknown size leaves few alternatives. We hope to address this shortcoming in a future revision of the MPI standard.

```
public delegate T ReductionOperation<T>(T x, T y);
public T Allreduce<T>(T value, ReductionOperation<T> op);
```

Figure 8. The ReductionOperation delegate and its use in the signature of the Allreduce operation.

4. Collective Communication

MPI collective communication operations are parallel algorithms that involve all processes in a communicator. Collectives range from simple operations such as a barrier to global reduction and all-to-all communications. Like MPI's point-to-point operations, collectives typically require the use of untyped memory (void pointers), explicit specification of datatypes, etc. For MPI.NET, we would instead like to provide a simplified interface that eliminates extraneous parameters while providing complete support for user-defined data types and C# programming idioms.

We focus on the reduction operations (Allreduce in particular), which present the most interesting challenges for interface design and implementation. Allreduce performs a reduction on the values supplied by each process using a specific reduction operation to combine values. MPI offers some predefined reduction operations, such as MPI_SUM and MPI_MIN, for the sum and minimum of the values, respectively; users are also free to supply their own reduction operations. With MPI.NET, we would like to subsume both of these approaches by allowing the user to provide any function to Allreduce:

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

The first call to Allreduce produces the sum of the integers provided by each process, while the second concatenates all of the strings provided by the processes. The actual reduction operation is provided via a *delegate*, which is the C# equivalent to function pointers. The MPI.NET ReductionOperation delegate encapsulates the notion of reduction operation, which is simply a function that accepts two values of type T and returns a new T, and can bind to any method that provides this signature. Figure 8 illustrates the ReductionOperation delegate and its use in Allreduce.

MPI.NET's collectives provide more functionality than the native MPI collectives, despite MPI.NET's simplified interface. For example, the native MPI_Allreduce cannot support concatenation of strings, because it supports neither serialization nor reduction with varying amounts of data provided by each process. Thus, each of the MPI collectives must be reimplemented within MPI.NET to support serialized data types. MPI.NET's collectives are generally implemented on top of MPI.NET's point-to-point operations, so that they provide support for both reference and value types.

4.1 Value Types

As with MPI.NET's point-to-point operations, we can provide a better mapping of collectives to the low-level MPI implementation if we examine the properties of types. For example, an ideal implementation would translate the first Allreduce call in Section 4 into a call to the lower-level MPI_Allreduce using the datatype MPI_INT and the reduction operation MPI_SUM. In this section, we detail the use of C# and the CLI to improve the mapping from the general Allreduce operation to the most specific use of MPI.

4.1.1 User-Defined Operations

The low-level MPI_Allreduce operation can be employed directly for the reduction of any value type (via its corresponding MPI datatype), so long as the reduction operation can be mapped to an MPI operation. Using the CLI's interface to native code, we can wrap a C# delegate to give it the interface expected by the low-level MPI. In particular, the type of the lower-level MPI's user-defined functions is a function pointer with the C signature:

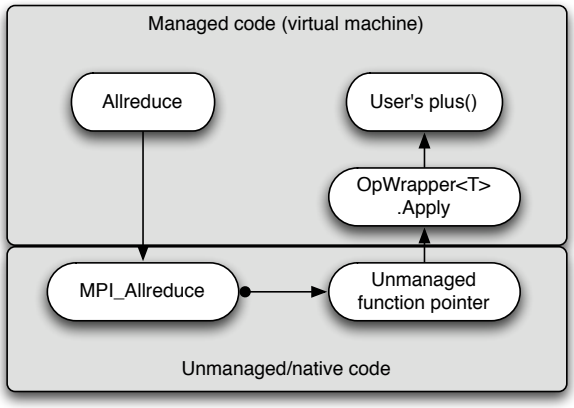

Figure 9. Invocation sequence used to bridge the managed code/unmanaged code barrier several times during an MPI collective operation.

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

Figure 9 shows the calling sequence for a typical Allreduce invocation on value types. The C# Allreduce method invokes the unmanaged MPI_Allreduce directly via the P/Invoke interface. Whenever MPI_Allreduce needs to call the reduction operation, it makes a call through an (unmanaged) function via a function pointer, which bridges to the managed code and the user's reduction operation.

Wrapping a reduction operation delegate into an MPI user-defined function therefore requires two steps. The first step involves wrapping the interface of MPI_User_function around the interface of ReductionOperation, which means applying the reduction operation to each of the pairs of elements in the two incoming arrays and writing the results to the second array. The second step involves taking that function—which itself executes within the virtual machine—and translating it into a function pointer that can be invoked from the unmanaged code of the low-level MPI library. Figure 10 illustrates how the first step can be achieved, by marshaling data from the raw memory provided by MPI into values of the appropriate type with the CLI Marshal class.

The Apply method in Figure 10 provides exactly the same signature as MPI_User_function, but it is still managed code written in C#. The CLI Marshal facility provides a mechanism for creating a function pointer that can be invoked from unmanaged code via its GetFunctionPointerForDelegate method. The result of this operation can be passed to MPI_Op_create, so that MPI can make calls back into the managed C# code.

While the OpWrapper class meets our original goal of providing a bridge between the low-level MPI_User_function and the high-level ReductionOperation, its reliance on the Marshal class's PtrToStructure and StructureToPtr methods quickly become a performance bottleneck. Semantically, these routines are just copying from unmanaged memory (provided by MPI) into managed objects. However, the result type of Marshal.PtrStructure is object, meaning that the result is boxed before being returned and then unboxed, causing unnecessary overhead. To reduce this overhead, we must eliminate the boxing and unboxing steps.

```
class OpWrapper<T> {
  public unsafe void
  Apply(void* invec, void* inoutvec, int* len,
        MPI_Datatype* datatype) {
    int size = Marshal.SizeOf(typeof(T));
    for (int i = 0; i < *len; ++i) {
      IntPtr inPtr = new IntPtr((byte*)invec + i * size);
      IntPtr inOutPtr = new IntPtr((byte*)inoutvec + i * size);
      T x = (T)Marshal.PtrToStructure(inPtr, typeof(T));
      T y = (T)Marshal.PtrToStructure(inOutPtr, typeof(T));
      T result = op(x, y);
      Marshal.StructureToPtr(result, inOutPtr, true);
    }
  }
  private ReductionOperation<T> op;
}
```

Figure 10. Wrapper around a user's ReductionOperation delegate that provides a method with the same interface as the low-level MPI_User_function.

```
public unsafe void
Apply(void* invec, void* inoutvec, int* len,
      MPI_Datatype* datatype) {
  int count = *len;
  for (int i = 0; i < count; ++i)
    *((T*)invec + i)
      = op(*((T*)invec + i), *((T*)inoutvec + i));
}
```

Figure 11. An ideal (but ill-formed) implementation of the Apply operation in Figure 10, which eliminates boxing and unboxing. C# rejects this method because one cannot create pointers to type parameters.

Figure 11 illustrates what would be a more efficient implementation of the Apply method, eliminating all boxing/unboxing and uses of the Marshal class. Unfortunately, the C# type system prevents any attempts to create a pointer to a type parameter, so this code is ill-formed. To effectively realize the implementation of Figure 11, we again step outside the bounds of C# to work directly with CIL assembler. In this case, we employ run-time code generation techniques using the facilities in the System.Reflection.Emit namespace to synthesize the analogue to Figure 11 at run time.

Figure 12 provides a sketch of the implementation of run-time code generation for code similar to Figure 11's Apply. The CLI DynamicMethod class allows one to generate a new method at run-time, by supplying the parameter and return types of the method. The DynamicMethod then provides an ILGenerator that can be used to add CIL instructions directly to the dynamic method body via the Emit method. We omit the details of looping over each of the elements in the two arrays, and focus on the invocation of the user's delegate, named op. We emit code to load the first argument (the delegate), then load the two values of type T from the current positions in invec and inoutvec, respectively. Finally, we call the delegate's Invoke method to execute the user-defined operation, and store the result back into the current location in inoutvec. With the proper looping code, the resulting method provides essentially the same virtual machine code as Figure 11 would (if the latter could be expressed in C#), but for the specific value type T.

The final operation in Figure 12 makes it possible to call our newly-generated method from a delegate whose signature matches the C MPI_User_function. The generated method has five parameters—a ReductionOperation<T>, two void pointers, an int*, and a MPI_Datatype*—while MPI_User_function

```
unsafe delegate void
  MPIUserFunction(void* invec, void* inoutvec, int* len,
                  MPI_Datatype* datatype);

public MPIUserFunction
BuildUserFunction(ReductionOperation<T> op) {
  DynamicMethod method
    = new DynamicMethod
        ("reduce:" + typeof(T).ToString(),
         typeof(void),
         new Type[] {
           /*op*/typeof(ReductionOperation<T>),
           /*invec*/typeof(void*),
           /*inoutvec*/typeof(void*),
           /*count*/typeof(int*),
           /*datatype*/typeof(MPI_Datatype*)
         },
         typeof(ReductionOperation<T>));
  ILGenerator generator = method.GetILGenerator();
  // Generate loop header...
  // Load the position where we will store the result
  generator.Emit(OpCodes.Ldarg_2); // inoutvec
  // Build argument list for the reduction
  generator.Emit(OpCodes.Ldarg_0); // delegate
  generator.Emit(OpCodes.Ldarg_1); // invec
  generator.Emit(OpCodes.Ldobj, typeof(T));
  generator.Emit(OpCodes.Ldarg_2); // inoutvec
  generator.Emit(OpCodes.Ldobj, typeof(T));
  // Call the delegate
  generator.EmitCall
   (OpCodes.Callvirt,
    typeof(ReductionOperation<T>).GetMethod("Invoke"),
    null);
  // Store the result
  generator.Emit(OpCodes.Stobj, typeof(T));
  // Generate loop increment...
  // Turn the method into a delegate and bind "op"
  return (MPIUserFunction)method
            .CreateDelegate(typeof(MPIUserFunction), op);
}
```

Figure 12. Run-time code generation to create an efficient bridge between the low-level C MPI_User_function and the high-level C# ReductionOperation.

(and its C# counterpart, MPIUserFunction) only have four. The CreateDelegate member of DynamicMethod binds the first argument to a specific object, returning a delegate that itself only accepts the latter four parameters, precisely matching the MPIUserFunction signature. This resulting delegate provides the same functionality as the Apply method in Figure 10, but eliminates all of the unnecessary boxing and unboxing.

4.1.2 Pre-defined Reduction Operations

The ability to use the low-level MPI_Allreduce for value types provides MPI.NET with the benefits of years of optimizations in the implementation of collectives within the low-level MPI library. However, the need to perform many calls from the unmanaged collective into the managed reduction operation (see Figure 9) can cause performance problems, particularly when the reduction operation itself is very simple, e.g., addition of two integers. MPI provides several predefined reduction operations for primitive types, such as MPI_SUM (addition), MPI_MIN (minimum), and MPI_BAND (bitwise and). Using these predefined reduction operations when possible eliminates the overhead of the managed-unmanaged round-trips for reduction operation applications.

MPI.NET provides a set of predefined reduction operations that can be applied to any type, built-in or user-defined. For example, one can sum all of the local values with Allreduce using MPI.NET's Add reduction operation:

```
int sum = world.Allreduce(myValue, Operation<int>.Add);
```

Add is a reduction operation that uses the + operator to add two values. With primitive types, this maps to the CIL add instruction. With user-defined class and structure types, Add will instead use an overloaded + operator. Thus, one can perform a generic "addition" of two values of type T via Operation<T>.Add. Having a common name for addition allows MPI.NET to detect reduction operations that map to MPI's predefined reduction operations. For example, Operation<int>.Add and Operation<float>.Add map to MPI_SUM, while Operation<Point>.Add would require the use of user-defined MPI operations described in the previous section. The mechanism for selecting the predefined MPI operation is quite simple: the type of the values passed to the reduction operation are classified based on MPI's type classifications [12, §4.9.2], and the reduction operation is then checked against MPI.NET's predefined reduction operations. If the reduction operations match, a predefined MPI operation such as MPI_SUM will be used; otherwise, a user-defined MPI operation will automatically be generated and used. Thus, MPI.NET seamlessly provides the most efficient use of the underlying MPI collectives possible, using known information about the data type and reduction operation.

Unfortunately, operations such as Operation<T>.Add cannot be implemented directly. The intuitive formulation in C# would be, simply:

```
public class Operation<T> {
  public T Add(T x, T y) { return x + y; }
}
```

However, the C# type checker rejects the Add method because the type T is not guaranteed to have a suitable + operator. While this problem is typically solved be introducing a constraint on T (e.g., that T must implement a particular interface), C# does not provide support for constraints on specific overloaded operators. Drawing from Farmer [5], we employ run-time code generation to build a ReductionOperation delegate for addition of any particular type T. MPI.NET provides a full set of these predefined operations, corresponding to MPI's predefined reduction operations, which both simplify the use of collectives in MPI.NET and provide greater optimization opportunities for the MPI.NET library.

5. Performance Evaluation

Since MPI.NET is layered on the MPI C bindings, its ultimate performance will be determined by the lower-level library. Accordingly, the essential performance metric for MPI.NET is *abstraction penalty*, i.e., the overhead incurred by a high-level interface when compared to an equivalent low-level interface. (And, indeed, the majority of the optimizations described in this paper were developed specifically to miminize abstraction penalty.)

We conducted several experiments to measure the abstraction penalty of MPI.NET. Our experiments were conducted on a 9-node cluster, where each node contains a dual-core 2.13GHz Intel Xeon 3050 processor and 2GB of RAM. Each node runs Microsoft Windows Compute Cluster Server 2003 [14], containing Microsoft MS-MPI. The nodes were connected by Gigabit Ethernet over a private network. All examples (whether C or C#) were compiled with Microsoft Visual Studio .NET 2005 with full optimization, and C# examples were executed with version 2.0.50727 of Microsoft's .NET framework.

Our first experiment involves NetPIPE [16], a micro-benchmark that measures the throughput of a network based on simple ping-pong tests with messages of various sizes. To determine the abstraction penalty of point-to-point messaging in MPI.NET, we ported NetPIPE (originally written in C) to C#, using MPI.NET as its communication layer.

Figure 13 illustrates the performance characteristics of NetPIPE. The most important comparison to make from this figure is between the C NetPIPE and its direct port to C# and MPI.NET, with both operating on primitive types (bytes). The difference between the C# and C variants is the abstraction penalty, and includes the costs associated with the .NET virtual machine, garbage collector, and interaction between managed and unmanaged code. Over TCP (center plots in Figure 13), any abstraction penalty due to C# or MPI.NET is lost in the noise. Shared memory (top plots in Figure 13), on the other hand, provides stable results that allow us to isolate the abstraction penalty. Figure 14 illustrates the efficiency of the C# implementation as a ratio of C# NetPIPE bandwidth to C NetPIPE bandwidth; less than one indicates that C# NetPIPE is slower. Here, we see that the abstraction penalty is generally very small (1-2%) for smaller messages, but results for larger messages are less obvious, with C# varying from 15% slower to 10% faster.

The bottom two plots of Figure 13 reflect the performance of the C# NetPIPE when it is forced to serialize the transmitted data, e.g., when it is provided with objects of class type rather than primitive values. This is the worst-case scenario for MPI.NET, and we see that performance suffers greatly when the optimizations described in Section 3 are not automatically performed by the library. We also note that the TCP and shared-memory performance for serialized types levels off when messages become larger than 100 bytes, indicating that NetPIPE is CPU-bound at this point.

We extended the NetPIPE program by replacing its ping-pong messaging with an all-reduction operation that computes the sum of double values. In this benchmark, we perform reductions that perform vector sums on arrays of double values using several different variants of data types and reduction operations. Figure 15 illustrates the performance of several of these variants when performing the reductions on all 18 cores of our cluster. As expected, the most efficient implementation is the most specialized, when MPI.NET translates the Allreduce call into a use of the lower-level MPI_Allreduce with the predefined reduction operation MPI_SUM. When MPI.NET's Allreduce is provided with a user-defined operation (e.g., the plus method from Section 4) or a user-defined value type, MPI.NET creates a user-defined MPI operation via MPI_Op_create. For reductions of larger arrays, the use of user-defined MPI operations is somewhat slower than predefined MPI operations like MPI_SUM due to cost of translating between managed and unmanaged code. As with the ping-pong NetPIPE, the use of serialization for class types drastically reduces the level of performance.

6. Related Work

Some of the earliest efforts to provide high-level language support for MPI were for C++, including Object-Oriented MPI (OOMPI) [17], MPI++ [1], mpi++ [9], and the C++ bindings themselves. These approaches varied in the abstraction level provided, e.g., with the C++ bindings providing almost a direct mapping of the C bindings while OOMPI [17] provides a higher-level interface that simplifies some construction of datatypes and the use of point-to-point operations. More recently, Boost.MPI [7, 10] has been developed, providing direct support for the generic programming paradigm. Boost.MPI supports automatic serialization of objects, and has served as the model for our work on MPI.NET. In particular, the high-level interfaces in MPI.NET and their resulting use of the low-level MPI library were ported directly from Boost.MPI [7] and its predecessor [10]. However, the generic programming techniques used in Boost.MPI could not be duplicated within C#, forcing MPI.NET

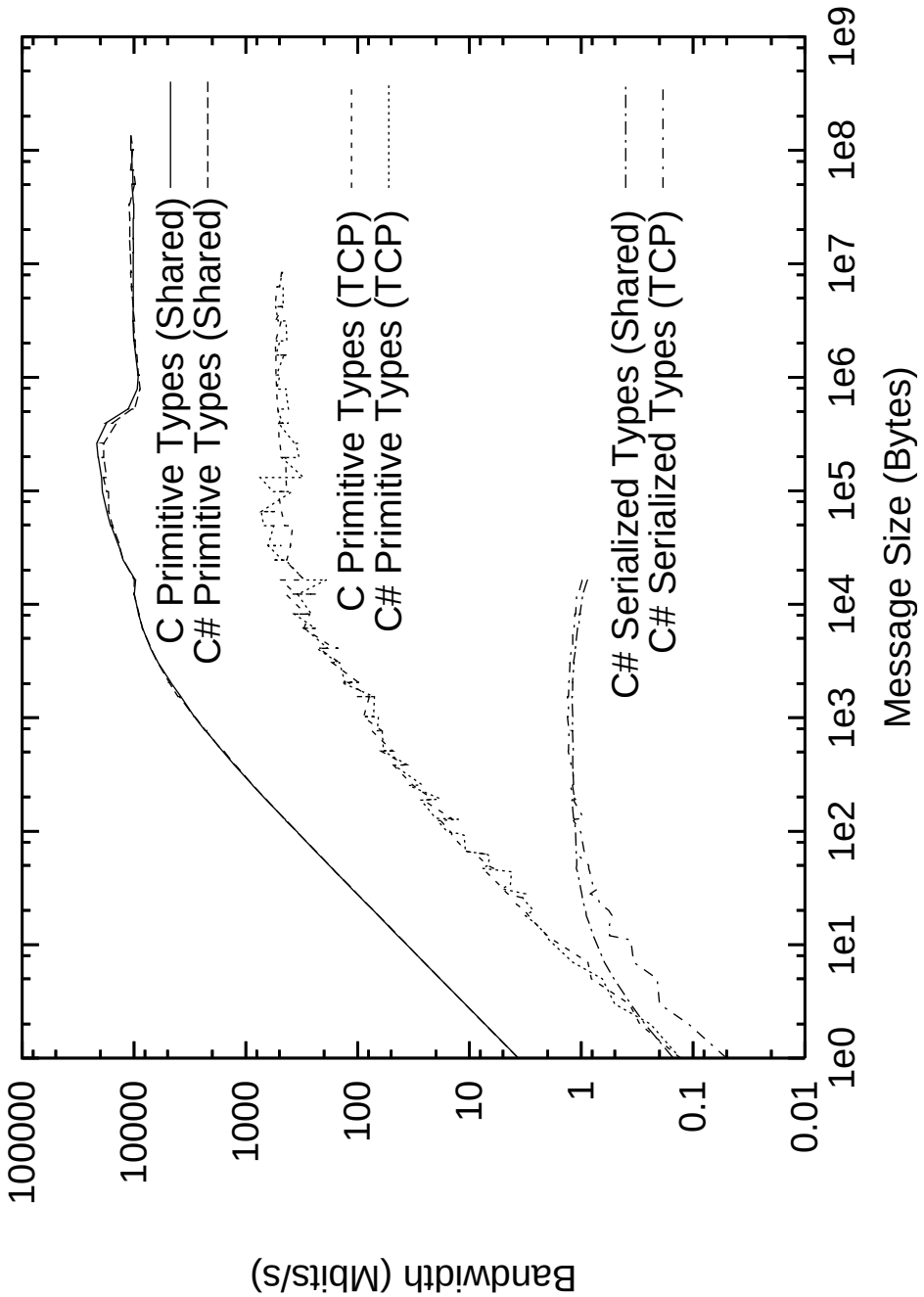

Figure 13. NetPIPE performance of MPI.NET (C#) and MPI (C) over Gigabit Ethernet (TCP) and shared memory (Shared) with various kinds of data types.

to use completely different implementation techniques to achieve suitable performance.

The Java language and its virtual machine present many of the same challenges as C# and the CLI for the implementation of MPI. There have been several versions of MPI in Java, including mpiJava [2] and JavaMPI [15] (both of which are layered on top of the native MPI library), the pure-Java MPIJ [8], and the MPI-inspired MPJ [4]. While some of the techniques from MPI.NET are shared with the Java MPI implementations (e.g., automatic serialization of Java objects [3]), MPI.NET provides a higher-level interface than existing Java bindings while still using the underlying MPI efficiently. For example, datatype arguments are not needed in MPI.NET, but its run-time optimization still transmits primitive types and other value types without serialization. MPI.NET also provides better support for user-defined data types, e.g., within collectives, and can be easily used from generic code.

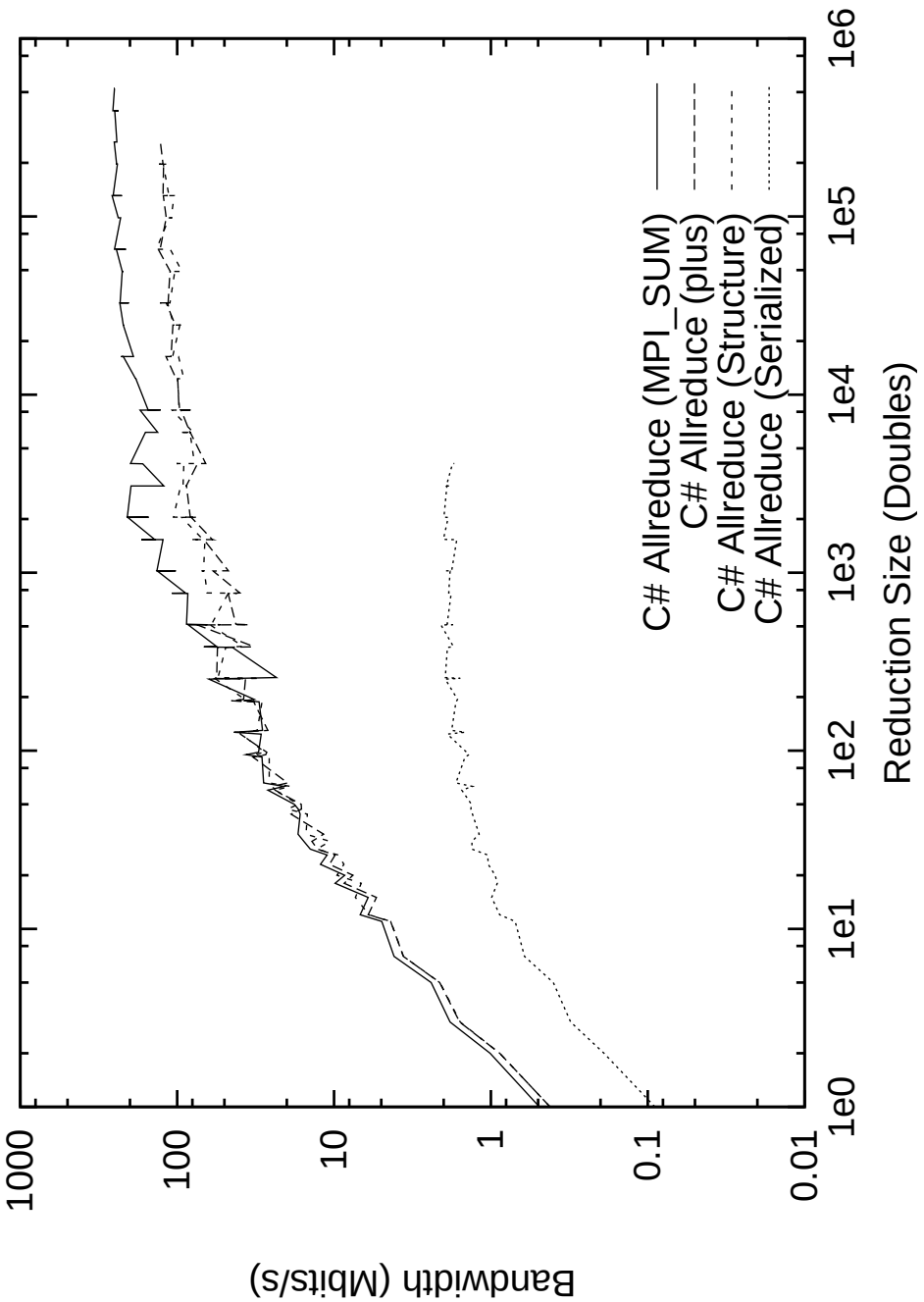

Figure 15. Allreduce performance of MPI.NET computing the sum of double values with MPI_SUM, double values with a user-defined plus method, a structure containing a double, and a class containing a double (which requires serialization).

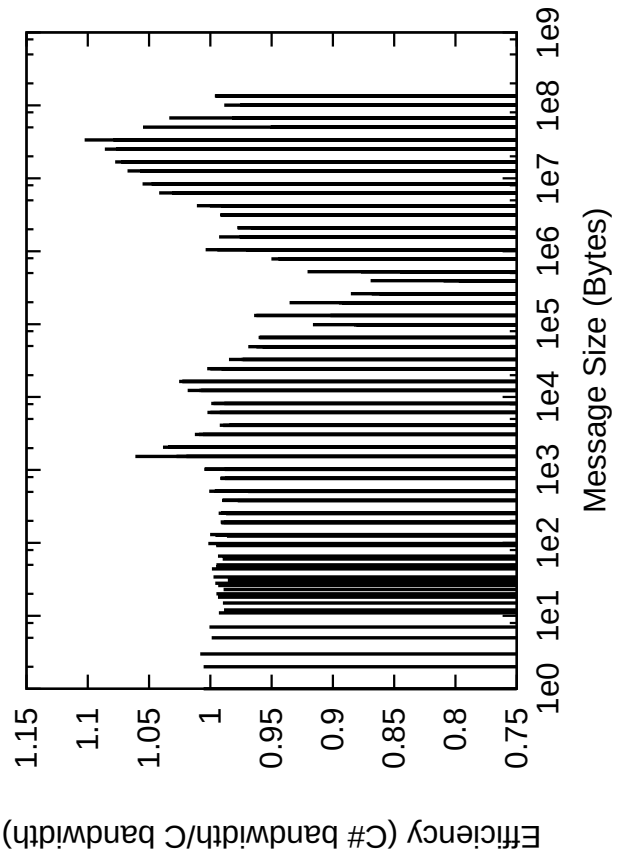

Figure 14. Efficiency of the C# NetPIPE relative to the native C NetPIPE using primitive types in shared memory.

Similar to the Java-based MPI implementations, an earlier C# MPI.NET [18] effort provided both a low-level interface to MPI (automatically generated from the MPI specification) and a OOMPI-like high-level interface to MPI [17]. MPI.NET utilizes many C# and CLI techniques not available when the earlier MPI.NET was developed, and therefore is able to provide a more uniform interface along with more aggressive optimizations.

7. Conclusion

C# and the Common Language Infrastructure provide an intriguing platform for high-performance and scientific computing. The combination of several C# and CLI features, including value types, generics, unsafe code regions, and run-time code generation, make it possible to provide high-level, high-performance bindings to existing libraries such as the Message Passing Interface. Our MPI.NET library exploits all of these features to provide a simple, intuitive interface to MPI that fits well with the C# programming model while mapping to efficient use of the underlying MPI implementation. Our performance results show that the ease-of-use and expanded functionality of MPI.NET is essentially free, making C# a suitable alternative to C, C++, and Fortran for high-performance MPI programming.

Acknowledgments

The authors thank Benjamin Martin for suggesting the use of run-time code generation to address some of the limitations of the C# language. This research was supported by the High-Performance Computing group at Microsoft and by a grant from the Lilly Endowment.

References

[1] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, pages 323–338, April 1994.

[2] Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko, and Sang Lim. mpiJava 1.2: API specification. Technical Report CRPC-TR99804, Rice University, Center for Research on Parallel Computation, September 1999.

[3] 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.

[4] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[5] Keith Farmer. Operator overloading with generics. `http://www.codeproject.com/csharp/genericoperators.asp`, June 2005.

[6] Douglas Gregor and Andrew Lumsdaine. MPI.NET: High-performance C# library for message passing. `http://www.osl.iu.edu/research/mpi.net/`, November 2007.

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

[8] Glenn Judd, Mark J. Clement, Quinn Snell, and Vladimir Getov. Design issues for efficient implementation of MPI in Java. In *Java Grande*, pages 58–65, 1999.

[9] Dennis Kafura and Liya 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/`.

[10] 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.

[11] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[12] Message Passing Interface Forum. MPI, June 1995. `http://www.mpi-forum.org/`.

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

[14] Microsoft. Windows Compute Cluster Server 2003. `http://www.microsoft.com/technet/ccs/overview.mspx`, August 2005.

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

[16] Q. Snell, A. Mikler, and J. Gustafson. NetPIPE: A network protocol independent performace evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[17] Jeffrey M. Squyres, Brian C. McCandless, and Andrew 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] Jeremiah Willcock, Andrew Lumsdaine, and Arch Robison. Using MPI with C# and the Common Language Infrastructure. *Concur-*

rency and Computation: Practice & Experience, 17(7-8):895–917, June/July 2005.