Design and implementation of Java bindings in Open MPI

Oscar Vega-Gisbert, Jose E. Roman*

D. Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera s/n, 46022 Valencia, Spain

Jeffrey M. Squyres

Cisco Systems, Inc., San Jose, CA 95134 USA

Abstract

This paper describes the Java MPI bindings that have been included in the Open MPI distribution. Open MPI is one of the most popular implementations of MPI, the Message-Passing Interface, which is the predominant programming paradigm for parallel applications on distributed memory computers. We have added Java support to Open MPI, exposing MPI functionality to Java programmers. Our approach is based on the Java Native Interface, and has similarities with previous efforts, as well as important differences. This paper serves as a reference for the application program interface, and in addition we provide details of the internal implementation to justify some of the design decisions. We also show some results to assess the performance of the bindings.

Keywords: Java, message-passing, Open MPI, Java Native Interface

1. Introduction

Since the introduction of the Java programming language, there has been a significant interest from part of the high-performance computing community in using it due to its many appealing features. Initially, this interest cooled down due to various problems, such as bad performance of the first versions of the Java Virtual Machine (JVM). Most of these issues have now been solved, and the last studies [1, 2] indicate that the overhead of Java is currently reasonable for computationally intensive applications. The performance can be improved even further by making use of numerical libraries in Java that in some cases employ efficient native computational kernels [3, 4, 5]. For instance, MTJ [6] can be configured to either use JLAPACK [7] or the native Fortran LAPACK.

The performance of modern JVM's is already very high, but still it is possible to tune some components of the Java runtime environment for best efficiency, such as the JIT compiler or the automatic Garbage Collector [8].

Several existing scientific computing codes written in Java illustrate the potential of this approach [9, 10, 11]. These kind of applications either incorporate some kind of parallelism, or point it as a subject for future development.

Roughly speaking, nowadays high-performance computing relies on three different parallel computing technologies and combinations thereof: message-passing for distributed-memory platforms (clusters), multi-threading mechanisms for shared-memory computers, and customized programming of accelerators (GPU's). Java provides efficient built-in support for threads, with cached thread pools that allows reusing inactive threads, and some numerical libraries are based on such mechanism [12]. GPU support in Java is more incipient but can be used to some extent [13].

Regarding message-passing parallelization, there are projects that try to define their own primitives in Java [14]. However, it seems clear that adopting the MPI standard also in Java is the preferred approach. We advocate for this in this paper, since we believe there are many benefits in doing this. On one hand, any programmer with experience in high-performance computing (HPC) that is going to start a new development in Java will likely want to have some

*Corresponding author

Email addresses: ovega@dsic.upv.es (Oscar Vega-Gisbert), jroman@dsic.upv.es (Jose E. Roman), jsquyres@cisco.com (Jeffrey M. Squyres)

kind of MPI interface. MPI provides a rich feature set such as collective communication that are missing in more general schemes such as sockets or RMI, both of which are more oriented to client-server applications. On the other hand, MPI can benefit from Java because its widespread use makes it likely to find new uses beyond traditional HPC applications.

In this work, we present Java MPI bindings implemented in the Open MPI [15] package. Integrating into a specific MPI implementation allows the Java bindings to be simpler by reaching directly back to the relevant MPI internals. The alternative is to only use public MPI API functionality, which can be problematic for some functionality (e.g., dealing with user-specified callback functions). Also, as will be discussed below, we follow a JNI-based approach in such a way that the Java bindings are as simple as possible, confining all the complexity within the well-established C implementation.

The rest of the paper is organized as follows. Section 2 provides an overview of MPI and Java, and surveys projects related to ours. Section 3 shows how to use the Java bindings in Open MPI, and includes several illustrative examples. Internal implementation details are deferred to section 4. Finally, in section 5 we show some results of the performance evaluation we have conducted. We wrap up with some concluding remarks.

2. Preliminaries and related work

This work aims at improving the availability of the Message Passing Interface (MPI) [16] in the Java language realm. MPI is a high-level library that is commonly used in HPC applications to abstract away many of the details of the underlying networking(s). MPI has multiple different major flavors of communication:

- Point-to-point: a pair of MPI process peers exchange messages.
- One-sided: messages are exchaged from one process to another, with one of them not explicitly participating in the operation apart from allowing the other peer to access its memory via the MPI window paradigm.
- Collective: a set of MPI processes synchronize and/or exchange messages.
- I/O: a set of MPI processes collectively access storage subsystems.

Each of these flavors, in both their blocking or non-blocking variants, are centered around the transfer of typed messages to and from application-specified buffers. Messages can be simple (e.g., containing only a single intrinsic type such as `int`, `float`, and `double`) or complex (e.g., containing a user-defined composite datatype comprised of multiple intrinsic datatypes).

The MPI application programming interface (API) utilizes many types of handles that refer to underlying implementation objects. For example, an MPI *communicator* is a set of MPI processes with a unique communication context. Its handle type in C is `MPI_Comm`. MPI handles can be any integer type; common implementation choices include `int` and pointers. As we will see, MPI handles are naturally represented as objects in Java.

There are several MPI implementations available. Open MPI is one of the most widely used, and it is developed as a coordinated international effort. The main goal of the project is to provide an open source MPI implementation that delivers high performance in a wide variety of platforms and environments. This paper describes how we have recently extended Open MPI so that it can be used from Java programs.

Java is an object-oriented programming language whose syntax is similar to C and C++. Its main distinguishable feature is that Java source files are compiled into an intermediate bytecode that is executed on a Java Virtual Machine (JVM). In this way, Java programs are portable to any system that has a JVM, without having to recompile. End users run Java applications via a Java Runtime Environment (JRE) installed on their computer, which contains the JVM. Software developers use the Java Development Kit (JDK), which contains various tools such as the compiler (`javac`), debugger (`jdb`), class packaging tool (`jar`) and documentation generator (`javadoc`), together with an extensive library of standard classes. These standardized classes provide a platform-independent way to access host-specific features such as threads, graphics, file management, and networking. Java was introduced in the mid 1990's and is now one of the most popular programming languages.

The use of a JVM to run the intermediate bytecode may be thought of as a serious performance penalty, compared to directly generating machine code. But JVM performance was greatly improved due to many optimizations, most

```
public class Win
{
    ...
    public Group getGroup() throws MPIException
    {
        MPI.check();
        return new Group(getGroup(handle));
    }

    private native long getGroup(long win) throws MPIException;
    ...
}
```

```
JNIEXPORT jlong JNICALL Java_mpi_Win_getGroup(JNIEnv *env, jobject jthis, jlong win)
{
    MPI_Group group;
    int rc = MPI_Win_get_group((MPI_Win)win, &group);
    ompi_java_exceptionCheck(env, rc);
    return (jlong)group;
}
```

Figure 1: Example to illustrate the invocation of a JNI method from Java (top) and the corresponding JNI implementation in C (bottom).

notably Just-In-Time (JIT) compilation systems. A JIT compiler translates bytecode segments to machine code during runtime for high-speed code execution. As a result, Java performance is often close to that of C/C++.

The Java runtime environment provides automatic memory management, so that the application developer does not have to worry about releasing allocated memory. The programmer is responsible for creating objects and the runtime is responsible for recovering memory once these objects are no longer in use. The Garbage Collector (GC) determines which objects are susceptible of being freed by taking into account the number of references to those objects. Hence the programmer must be careful not to keep references to unused objects. The use of GC reduces memory leaks and therefore improves the software quality. One important implication of the Java memory model is that explicit memory pointers (with pointer arithmetic) are not supported, in order to allow the GC to relocate memory to optimize allocated space.

In this work, we make extensive use of the Java Native Interface (JNI) [17], a mechanism that allows the application programmer to call *native* subroutines and libraries from Java and vice versa. Native code is typically written in C/C++ and runs directly on the operating system where the Java virtual machine is running. In the context of scientific computing, JNI is often used to invoke numerical kernels that are optimized for the particular machine characteristics (e.g., a tuned BLAS) and would be much slower if implemented in Java. The downside is that JNI breaks Java's spirit of automatic portability, since it requires recompiling the native code whenever we want to port the application to a new computer.

The example in Fig. 1 illustrates how to call a JNI method from Java, and also shows the corresponding JNI implementation in C. When the JVM invokes the native method, it passes a `JNIEnv` pointer, a `jobject` variable, and any arguments declared by the Java method. `JNIEnv` is a structure that contains the necessary information to interface to the JVM. There are a number of functions available to interact with the JVM and to work with Java objects, for instance to convert Java arrays to native arrays or to throw exceptions. The second argument of type `jobject` is a reference to the Java object inside which this native method has been declared. The types `jlong`, `jdouble`, `jchar`, etc. are defined in the C side in order to match Java basic datatypes. String and array arguments passed from Java have types `jstring` and `jintArray` (in the case of integers), respectively, and the native code must use specific functions to get access to the corresponding elements, typically with a pair of calls `GetXXXArrayYYY`/`ReleaseXXXArrayYYY`, where `XXX` must be replaced with the type of the array, and `YYY` indicates the access variant (we will discuss some of the variants in Section 4).

We conclude this section with a brief overview of related work. There are many projects attempting to provide Java bindings for MPI. A recent survey [2] analyzes 9 such projects. Almost all these projects follow the pure Java approach, that is, they provide an MPI implementation completely written in Java, without native code. The only exception is mpiJava [18], one of the first MPI Java bindings available, which relies on JNI to invoke MPI primitives

written in C. We took mpiJava as the starting point to develop the Open MPI bindings, although we have ended up with a total rewrite that contains almost nothing from the original code base.

Claiming that JNI has portability problems, most efforts after mpiJava preferred to develop the Java bindings on a pure Java basis. Examples are MPJava [19], MPJ Express [20], and F-MPJ [21], to name a few. These are based on implementing MPI primitives by means of Java functionality such as RMI or sockets. This implies a significant coding effort for re-implementing all of MPI, and also has potential problems related to poor performance, especially if the standard sockets library is used for communication (which does not exploit specific networking hardware). Recent research efforts focus on providing efficient Java communication middleware, particularly for low-latency networks such as InfiniBand or Myrinet [22], or for the case when communication takes place via shared memory [23].

As mentioned before, portability problems of JNI must be understood in the sense that native code must be recompiled in each different machine. Our approach is to provide the Java bindings within the Open MPI distribution, in such a way that both the native C library and the Java bindings are built simultaneously during installation. In our view, this satisfies portability needs in most practical situations. Furthermore, one of the main criticisms of the original mpiJava project was the failure to interoperate successfully with a broad variety of MPI implementations; by attaching to a particular MPI implementation, we also avoid this problem. Although our bindings are tied specifically to Open MPI, they can be easily adopted by other MPI implementations as discussed in Section 4.6.

Most of the MPI Java implementations try to adhere to a standardized API. Two main APIs have been proposed. One of them is the mpiJava 1.2 API [24], which mimics the MPI C++ interface defined in the MPI 2.0 specification. However, it is restricted to the MPI 1.1 subset, and, on the other hand, the C++ bindings have been removed in the MPI 3.0 specification. The other API was proposed by the Java Grande Forum[1] and is called MPJ (message-passing interface for Java) [25]. This API does not seem to have wide acceptance, since it has been adopted just by one project. The two APIs differ mainly with respect to naming conventions of variables and methods. When designing the Open MPI Java bindings, we felt that it was appropriate to divert from these two APIs and define a new one that can be more close to modern Java programming practices. Also, we cover the MPI 3.0 specification completely.

3. User interface

We propose Java MPI bindings consisting of a thin interface on top of the C native library, which is invoked via JNI. The Java-JNI layer just performs minimal bookkeeping required for language translation, such as wrapping MPI object handlers with Java objects. Hence, our Java bindings essentially are a 1-to-1 mapping to the MPI C bindings—they are not intended as a class library with higher-level semantics and functionality beyond the MPI specification.

It is desirable to avoid semantics that are not strictly necessary for the language and not part of the MPI specification. For example, MPI handles contained in Java objects are not freed when these objects are automatically garbage collected. Instead they must be explicitly freed by calling the corresponding MPI function because some of these methods are collective and must be called by the application, instead of when the GC wants (which may cause race conditions).

JNI communication usually takes place in the Java-C direction, that is, a Java method invokes an MPI primitive via JNI and returns the result back to Java. However, we also make provision for communication in the opposite direction, that is, when the MPI C library needs to invoke a method in the Java application code. As we will see, this is required for callback-based functionality such as performing a reduction with a user-defined operation.

Fig. 2 illustrates the Java bindings with a simple example to compute π by numerical integration of $\int_0^1 \frac{4}{1+x^2}$. Throughout this section we provide details of the class organization and how they are used in practice.

3.1. Integration into Open MPI

The Java bindings described in this paper are seamlessly integrated into Open MPI, starting from the v1.7 series. The source code can be found under a subdirectory `java` at the same level where the C, Fortran and C++ user interfaces are located.

In order to use Java bindings in Open MPI, they must be enabled during configuration. If the JDK can be found in a standard location, the simplest way to do this is:

[1] `http://www.javagrande.org`

```
public static void main(String args[]) throws MPIException
{
    MPI.Init(args);

    int rank = MPI.COMM_WORLD.getRank(),
        size = MPI.COMM_WORLD.getSize(),
        nint = 100; // Intervals.
    double h = 1.0 / (double)nint,
           sum = 0.0;
    for(int i = rank + 1; i <= nint; i += size) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }

    double sBuf[] = { h * sum },
           rBuf[] = new double[1];

    MPI.COMM_WORLD.reduce(sBuf, rBuf, 1, MPI.DOUBLE, MPI.SUM, 0);
    if(rank == 0) System.out.println("PI: "+ rBuf[0]);

    MPI.Finalize();
}
```

Figure 2: Example to illustrate the Java bindings.

```
$ ./configure --enable-mpi-java ...
```

Otherwise, it is necessary to indicate where the JDK binaries and headers are located:

```
$ ./configure --enable-mpi-java --with-jdk-bindir=<foo> --with-jdk-headers=<bar> ...
```

Java support requires that Open MPI be built at least with shared libraries (this is the default, so the `--enable-shared` option need not be provided).

After configuration and compilation, a class file `MPI.jar` will be created. It will be copied to the destination directory during `make install`. In addition to the compiled classes, the generated `javadoc` documentation will also be copied.

For convenience, an `mpijavac` wrapper compiler has been provided for compiling Java-based MPI applications. It ensures that all required MPI libraries and class paths are defined. Once the application has been compiled, the user can run it with the standard `mpirun` command line:

```
$ mpirun <options> java <java-options> <my-app>
```

For convenience, `mpirun` has been updated to detect the `java` command and ensure that the required MPI libraries and class paths are defined to support execution. Therefore, it is not necessary to specify the Java library path to the MPI installation, nor the MPI classpath. Any class path definitions required for the application should be specified either on the command line or via the `CLASSPATH` environmental variable.

3.2. Overall organization

We implement an `mpi` package that contains all classes of the MPI Java bindings: `Comm`, `Datatype`, `Request`, etc., see Fig. 3. Most of these classes have a direct correspondence with objects defined by the MPI standard. MPI primitives are just methods included in these classes. The convention used for naming Java methods and classes is the usual camel-case convention, e.g., the equivalent of `MPI_File_set_info(fh,info)` is `fh.setInfo(info)`, where `fh` is an object of the class `File`.

The main classes of the Java bindings (those with rounded corners in Fig. 3) follow a similar hierarchy as in the mpiJava and MPJ APIs (we have added the classes `Win` and `File`). These classes provide the core MPI functionality, such as message passing, communicator management, or input/output. Some additional classes defined in the MPI specification are covered as well, such as `Info`. Apart from these, we include a few auxiliary classes that aim at the following purposes:

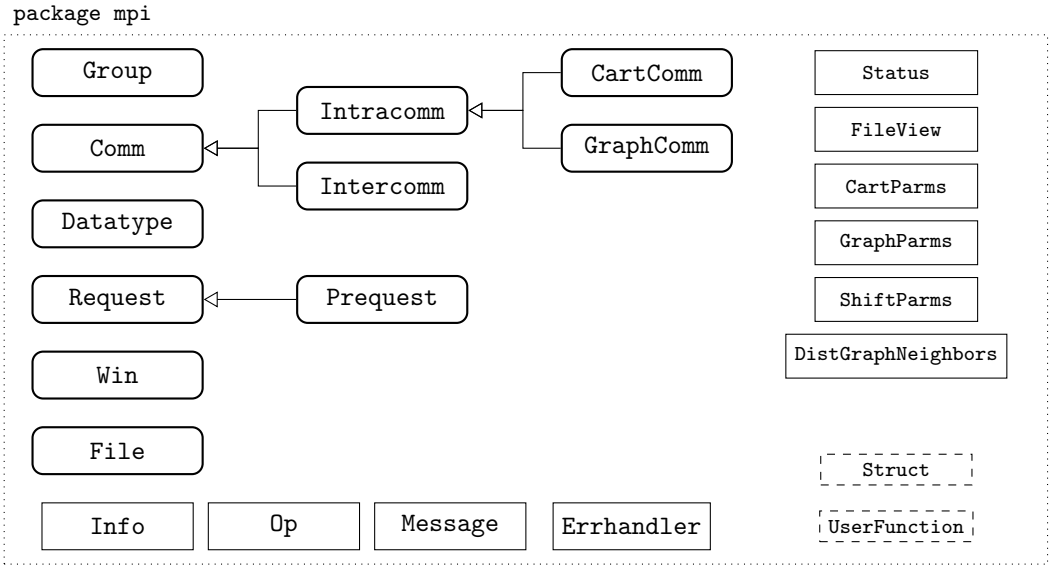

Figure 3: Class hierarchy of the Open MPI Java bindings. Main classes are depicted with rounded corners. Classes shown at the bottom are minor classes defined by the MPI specification. Classes on the upper-right corner are used when a method returns structured information. The two classes depicted with dashed lines are abstract classes intended to be extended by the user.

1. Classes intended to aggregate several elements, to be used when a method returns structured information. For instance, `Status` is defined as a C `struct` in the MPI specification, containing information about the received message such as the tag or the rank of the source process.
2. Abstract classes (shown as dashed boxes in Fig. 3) intended to be extended by the user.
3. Convenience classes (not shown in Fig. 3) to handle special datatypes such as `DoubleComplex`.

All of them will be described in detail in the rest of this section.

Apart from the classes, the `mpi` package contains predefined public attributes under a convenience class called `MPI` (not shown in Fig. 3). Examples are the predefined communicator `MPI.COMM_WORLD` or predefined datatypes such as `MPI.DOUBLE`. Also, MPI initialization and finalization are methods of the `MPI` class and must be invoked by all MPI Java applications, see Fig. 2. Any other MPI primitives not directly tied to a particular class, such as `MPI_Buffer_attach()`, also belong to the `MPI` class.

3.3. Exception handling

MPI routines return an error code. This is natural in C, but Java has exception handling as the standard method to treat errors. Hence, Java bindings in Open MPI support exception handling. All Java methods will throw an `MPIException` if an MPI primitive invoked in the JNI C side returns with an error code different from `MPI_SUCCESS`.

By default, errors are fatal, but this behavior can be changed by the application programmer. The Java API will throw exceptions if the `MPI.ERRORS_RETURN` error handler is set:

```
MPI.COMM_WORLD.setErrhandler(MPI.ERRORS_RETURN);
```

If this statement is added to the application code, it will show the line where it breaks, instead of just crashing in case of an error. Error-handling code can be separated from main application code by means of `try-catch` blocks, as shown in Fig. 4. The class `MPIException` extends the standard class `Exception` and hence allows for error handling and recovery in the usual Java way.

Exception handling has some advantages. It allows separating error-handling code from main application code. Errors can be grouped and differentiated in order to handle different kinds of errors separately.

```
try
{
    File file = new File(MPI.COMM_SELF, "filename", MPI.MODE_RDONLY);
}
catch(MPIException ex)
{
    System.err.println("Error Message: "+ ex.getMessage());
    System.err.println("  Error Class: "+ ex.getErrorClass());
    ex.printStackTrace();
    System.exit(-1);
}
```

Figure 4: Example to illustrate exception handling.

```
Comm comm = MPI.COMM_WORLD;
int me = comm.getRank();

if(me == 0) {
    comm.send(data, 5, MPI.DOUBLE, 1, 1);
} else if(me == 1)  {
    Status status = comm.recv(data, 5, MPI.DOUBLE, MPI.ANY_SOURCE, 1);
    int count = status.getCount(MPI.DOUBLE);
    int src = status.getSource();
    System.out.println("Received "+ count +" values from "+ src);
}
```

Figure 5: Example to illustrate point-to-point communication.

3.4. Point-to-point communication

Point-to-point communication is realized via methods of the `Comm` class. The standard `send()` and `recv()` operations take the usual arguments: the message, consisting of the buffer, number of elements and datatype, and the envelope, consisting of the process rank and the tag (the communicator is implicit by the object that effects the method call). Note that, as opposed to the mpiJava and MPJ APIs, we do not add an extra argument to specify an offset within the buffer; the rationale is to try to stay as close as possible to the standard MPI C specification (see Section 3.6 below for a discussion on how to treat the case where only part of the buffer array must be considered for the message).

The example in Fig. 5 shows a simple communication between two processes. It also exemplifies the use of the `Status` class returned by the `recv()` primitive. If `status` is not required, then it can just be omitted from the call, and hence there is no need for an equivalent to `MPI_STATUS_IGNORE`.

Apart from the standard communication primitives, we can also use the primitives for synchronous, buffered and ready modes, as well as the corresponding non-blocking variants. When invoking a non-blocking method, a `Request` object is returned, which can be subsequently used to manage the completion of the operation. This is illustrated in the example of Fig. 6 (in the context of collective communication), where the reader will notice that waiting for the finalization of a request is done via `waitFor()` due to a name clash with the standard Java method `wait()`. In addition to `waitFor()`, there is the `waitStatus()` version, that returns the `Status` of the completed communication, and similarly for the `test()` operation. The reason for having two methods, `waitFor()` and `waitStatus()`, instead of one overloaded method, is that the Java language does not allow overloading in the case of equal signature but different return type. The multiple completion operations, to test or wait for all, some or any of the pending requests (all of them also having a `xxxStatus()` version) must be invoked as follows

```
Request.waitAll(req);
```

where `req` is an array of `Request` objects.

The example of Fig. 6 also illustrates that it is necessary to call the `free()` method on objects whose class implements the `Freeable` interface (such as `Request`). Otherwise a memory leak is produced.

Persistent communication primitives such as `Comm.sendInit()` return a persistent request object `Prequest`, that can be used to start the actual communication at any later time with the method `Prequest.start()`.

Table 1: Predefined datatypes. The left panel shows the basic datatypes (discussed in Section 3.5) and their corresponding Java datatypes. The right panels lists the composite datatypes, described in Section 3.8.

BYTE	byte
CHAR	char
SHORT	short
BOOLEAN	boolean
INT	int
LONG	long
FLOAT	float
DOUBLE	double
PACKED	–

FLOAT_COMPLEX
DOUBLE_COMPLEX
INT2
SHORT_INT
LONG_INT
FLOAT_INT
DOUBLE_INT

The `probe()` and `iProbe()` methods allow checking for incoming messages without actually receiving them. They take as arguments the source and the tag, and return a `Status` objects. We have also implemented the MPI-3 *matching* variants `mProbe()` and `imProbe()`, together with the matching receive primitives; all of them are methods of the `Message` class.

The `Intracomm` and `Intercomm` classes extend the `Comm` class to represent intra- and inter-communicators, respectively. Methods that are defined only for intra-communicators, such as the scan-type collective communication primitives, are included in the former class, while the latter provides specific methods for inter-communicators, such as `getParent()` or `getRemoteSize()`. Methods that are defined for both intra- and inter-communicators, such as the point-to-point primitives, belong to the parent class, `Comm`. See Section 3.10 for additional details on inter-communicators.

3.5. Predefined datatypes

Table 1 lists the predefined datatypes available in the Open MPI Java bindings. We distinguish between basic and composite datatypes (the latter are discussed in Section 3.8). The length in bytes of the basic datatypes is established by the Java language specification. In particular, the `char` type occupies two bytes, since it represents a Unicode character.

In communication primitives, the datatype matching and conversion rules defined in the MPI specification apply, since message assembly and disassembly is carried out in the C side. For instance, it is possible to send 5 `MPI.DOUBLE` and receive 40 `MPI.BYTE`.

The special type `MPI.PACKED` is used for packed messages that have been assembled with `pack()` and unassembled with `unpack()`.

3.6. How to specify buffers

In general, in MPI primitives that require a buffer (either send or receive) the Java API admits a Java array. Since Java arrays can be relocated by the Java runtime environment, the MPI Java bindings need to make a copy of the contents of the array to a temporary buffer, then pass the pointer to this buffer to the underlying C implementation. From the practical point of view, this implies an overhead associated with all buffers that are represented by Java arrays. The overhead is small for small buffers but increases for large arrays.

An alternative is to use *direct buffers* provided by standard classes available in the Java SDK such as `ByteBuffer`. For convenience we provide a few static methods `new[Type]Buffer()` in the `MPI` class to create direct buffers for a number of basic datatypes. Elements of the direct buffer can be accessed with methods `put()` and `get()`, and the number of elements in the buffer can be obtained with the method `capacity()`. The example in Fig. 6 illustrates its use. When a direct buffer is created with one of the convenience methods `new[Type]Buffer()`, the native endianness is always established.

Direct buffers are available for: `BYTE`, `CHAR`, `SHORT`, `INT`, `LONG`, `FLOAT`, and `DOUBLE`. There is no direct buffer for booleans.

```
int myself = MPI.COMM_WORLD.getRank();
int tasks  = MPI.COMM_WORLD.getSize();

IntBuffer in  = MPI.newIntBuffer(MAXLEN * tasks),
          out = MPI.newIntBuffer(MAXLEN);

for(int i = 0; i < MAXLEN; i++)
    out.put(i, myself);      // fill the buffer with the rank

Request request = MPI.COMM_WORLD.iAllGather(out, MAXLEN, MPI.INT, in, MAXLEN, MPI.INT);
// do other work here
request.waitFor();
request.free();

for(int i = 0; i < tasks; i++) {
    for(int k = 0; k < MAXLEN; k++) {
        if(in.get(k + i * MAXLEN) != i)
            throw new AssertionError("Unexpected value");
    }
}
```

Figure 6: Example to illustrate the use of direct buffers.

```
import static mpi.MPI.slice;
...
int numbers[] = new int[SIZE];
...
MPI.COMM_WORLD.send(slice(numbers, offset), count, MPI.INT, 1, 0);
```

Figure 7: Example to illustrate slicing of buffers.

Direct buffers are not a replacement for arrays, because they have higher allocation and deallocation costs compared to arrays. In some cases arrays will be a better choice, typically with small buffers. One can easily convert a direct buffer into an array and vice versa. There is also the possibility of a non-direct `Buffer`, that employs an array internally. As a user guidance, we note the following:

- The only advantage of direct buffers is to avoid a copy when changing from Java to C inside Open MPI. This benefit will be noticed only for large buffers.
- Direct buffers should be allocated primarily for long-lived buffers. If the buffer is created and destroyed within a loop, then the overhead will be too high.
- The indexing notation of arrays is more natural for programmers than the explicit calls to `put()` and `get()`.

Even though in most cases it is possible to choose between arrays and direct buffers, there are some restrictions in certain operations. All non-blocking methods must use direct buffers and only blocking methods can choose between arrays and buffers (this is justified in Section 4.2). When a method in the API declares the buffer argument as an `Object`, then it can be either an array or a direct or non-direct buffer.

Buffer arguments must always be either arrays or buffers. If one wants to send or receive a simple variable such as an `int` it must be declared as an array: `int k[] = { value }`, as in the example of Fig. 2. In any communication operation, if the buffer argument is an array (or a non-direct buffer), the datatype specified in the call must match the type of the array elements, e.g., `MPI.INT` for `int[]`. If the call specifies a derived datatype (see discussion in Section 3.7), the basic datatype from which it was built must also match the array type.

In a C program, it is common to specify an offset in an array with `&array[i]` or `array+i`, for instance to send data starting from a given position in the array. The equivalent form in the Java bindings is to `slice()` the buffer to start at an offset, as shown in Fig. 7. Making a `slice()` on a buffer is only necessary when the offset is not zero. This slicing mechanism works for both arrays and direct buffers.

```
double A[] = new double[N*N];  // flattened 2D matrix
Datatype blk = Datatype.createVector(3, 3, N, MPI.DOUBLE);
blk.commit();
int rank = MPI.COMM_WORLD.getRank();
if(rank == 0) {
    MPI.COMM_WORLD.send(A, 1, blk, 1, 0);
} else if (rank==1) {
    MPI.COMM_WORLD.recv(A, 1, blk, 0, 0);
}
blk.free();
```

Figure 8: Example to illustrate the use of derived datatypes.

3.7. Derived datatypes

The Java bindings also allow the definition of derived datatypes, such as contiguous, vector or indexed. The example shown in Fig. 8 uses a derived datatype to send the 3×3 leading block of an $\mathtt{N} \times \mathtt{N}$ matrix that has been stored as a one-dimensional array.

When working with derived datatypes, it is often handy to query the size of the datatype (number of bytes of the data in a message that would be created with this datatype), with `getSize()`, and also about extent information, with `getExtent()` (and `getLb()` for the lower bound). The method `createResized()` can be used to create a datatype with a new lower bound and extent from an existing datatype. In the definition of vector and indexed types, the argument `stride` indicates the spacing between blocks as a multiple of the extent of the old datatype. We also provide the "H" variants, such as `createHVector()`, in which the stride is given in bytes, rather than in elements.

The `struct`-like derived datatypes require a special treatment. The definition of an MPI `struct` in C consists in two steps: first, creating the C `struct`, and second, getting the addresses of the `struct` attributes in order to call `MPI_Type_create_struct()`. A Java class with no methods is similar to a C `struct`, but in Java it is not possible to get the addresses of class members, so creating an MPI `struct` derived datatype must be done in a different way.

We propose the following mechanism. A user-defined `struct` must be a subclass of `Struct` and it must have three parts:

- A number of data fields defined using the `add[Type]()` methods, that return the corresponding offsets according to the size of the type. The offsets must be stored because they will be necessary to access data.
- A subclass of `Struct.Data` with `get`/`put` methods to access data. These methods will need the previously stored offsets.
- The implementation of the method `newData()` in order to tell the library how to create `Data` objects.

An example is shown in Fig. 9, where the `Struct` mechanism is used to represent a complex number. Once the `struct` has been defined, it can be used to create an object that corresponds to the type of the `struct`. Structs can define a singleton object as its type, but they could be dynamic depending on the constructor parameters.

The `Struct.Data` objects are references to a buffer region where the `struct` data is stored. They can be obtained by calling the method `Struct.getData()`, followed by the corresponding `get`/`put` methods. Fig. 10 is an example of how to access the elements of an array of complex numbers stored in a direct byte buffer. The MPI `struct` datatype is created automatically (it is committed in its first use), and it can be retrieved by calling the `getType()` method.

Structs need byte displacements, so they can only be used with byte buffers. If buffers are direct then the performance is reasonably good, otherwise reading and writing `struct` data will need serialization, with the corresponding performance degradation.

3.8. Composite datatypes

The composite datatypes shown on the right panel of Table 1, `INT2`, `SHORT_INT`, `LONG_INT`, `FLOAT_INT` and `DOUBLE_INT`, are implemented with `Struct` with the aim of being able to represent `SHORT`, `INT` and `LONG` in a way that are compatible with the native C MPI datatypes. In this way, it is possible to use the native implementation of the `MPI_MAXLOC` and `MPI_MINLOC` operations via JNI.

```
public class Complex extends Struct
{
    // This section defines the offsets of the fields.
    private final int real = addDouble(),
                      imag = addDouble();

    // This method tells the super class how to create a data object.
    @Override protected Data newData() { return new Data(); }

    public class Data extends Struct.Data
    {
        // These methods read from the buffer:
        public double getReal() { return getDouble(real); }
        public double getImag() { return getDouble(imag); }

        // These methods write to the buffer:
        public void putReal(double r) { putDouble(real, r); }
        public void putImag(double i) { putDouble(imag, i); }
    } // Data
} // Complex
```

Figure 9: Complex number class created as a subclass of `Struct`.

```
    Complex type = new Complex();
    ByteBuffer buffer = MPI.newByteBuffer(type.getExtent() * count);

    for(int i = 0; i < count; i++) {
        Complex.Data c = type.getData(buffer, i);
        c.putReal(0);
        c.putImag(i * 0.5);
    }

    MPI.COMM_WORLD.send(buffer, count, type.getType(), 1, 0);
```

Figure 10: Using complex numbers from the declaration of the `Complex` class in Fig. 9.

```
// Each process has an array of 30 double: ain
double[] ain  = new double[30];
...
double[] aout = new double[30];
int[]    ind  = new int[30];
int      rank = comm.getRank(),

ByteBuffer in  = MPI.newByteBuffer(30 * MPI.DOUBLE_INT.getExtent()),
           out = MPI.newByteBuffer(30 * MPI.DOUBLE_INT.getExtent());

for(int i = 0; i < 30; i++) {
    DoubleInt.Data d = MPI.doubleInt.getData(in, i);
    d.putValue(ain[i]);
    d.putIndex(rank);
}

comm.reduce(in, out, 30, MPI.DOUBLE_INT, MPI.MAXLOC, root);
// At this point, the answer resides on process root.

if(rank == root) {
    // Read ranks out
    for(int i = 0; i < 30; i++) {
        DoubleInt.Data d = MPI.doubleInt.getData(out, i);
        aout[i] = d.getValue();
        ind[i]  = d.getIndex();
    }
}
```

Figure 11: A reduction using `MPI.DOUBLE_INT` and `MPI.MAXLOC`.

The `Struct.Data` interfaces for these types use `get/putValue()` for the first value and `get/putIndex()` for the `INT`. All of these types have a corresponding object (subclass of `Struct`) in the `MPI` class: `int2`, `shortInt`, `longInt`, `floatInt` and `doubleInt`.

Fig. 11 is an example of the use of `MPI.DOUBLE_INT` and `MPI.MAXLOC`. It performs a reduction of a list of 30 value-index pairs per MPI process, keeping the largest values and the rank of the process owning them. It corresponds to the example 5.17 of [16] translated to Java.

In the original mpiJava implementation, `MPI.INT2` was the only composite datatype that could be used with `MPI.MAXLOC` and `MPI.MINLOC`. It was created with the Java binding of `MPI_Type_contiguous()`. So the old `MPI.INT2` was not really equal to the C `MPI_INT2` datatype, and hence, it was not possible to use the C implementation of `MPI_MAXLOC` and `MPI_MINLOC`. For this reason, these methods were implemented in Java.

Convenience classes. The Java language does not have a standard datatype for complex numbers, which are quite commonly needed in scientific computing applications. In our bindings, `FloatComplex` and `DoubleComplex` are convenience classes to access data of the MPI types `FLOAT_COMPLEX` and `DOUBLE_COMPLEX`. These classes provide methods to wrap buffers as well as `get/put` methods to access the real and imaginary parts. The interface is similar to `Struct` but the implementation is different in order to avoid data serialization in non-direct buffers.

3.9. Collective communication

Collective communication operations are very important in high-performance computing applications, since they allow for complex data exchange patterns in a simple and efficient way. Some of the already mentioned examples include collective primitives such as reduction (Fig. 2) and gather (Fig. 6). In Open MPI we interface to all the available collective communications, including the non-blocking variants introduced in MPI-3.

The Java collective methods have a similar signature as in the C API. In addition, for those primitives that take two buffers, we have added variants that only take one buffer in order to perform the corresponding operation in place. Further details about in-place collectives are discussed in Section 4.3.

In the case of reduction and prefix operations, the MPI primitives take an argument that represents the operation to be performed on the data. In Java, this argument belongs to the class `Op`, and there are several of them predefined in

```
UserFunction uf = new UserFunction()
{@Override public void call(Object in, Object inOut, int count, Datatype type)
{
    double[] vIn    = (double[])in,
             vInOut = (double[])inOut;

    for(int i = 0; i < count; i++)
        vInOut[i] += vIn[i];
}};

Op op = new Op(uf, true);
```

Figure 12: User-defined Op with arrays.

```
UserFunction uf = new UserFunction()
{@Override public void call(ByteBuffer in, ByteBuffer inOut, int count, Datatype type)
{
    DoubleBuffer bIn    = in.asDoubleBuffer(),
                 bInOut = inOut.asDoubleBuffer();

    for(int i = 0; i < count; i++)
        bInOut.put(i, bInOut.get(i) + bIn.get(i));
}};

Op op = new Op(uf, true);
```

Figure 13: User-defined Op with direct buffers.

class MPI: SUM, PROD, MAX, LAND, etc., as well as the ones discussed in the previous subsection (MAXLOC and MINLOC, see Fig. 11).

It is also possible to employ user-defined reduction operations, in the same way as predefined ones. For this, a new Op must be defined. The Op constructor requires a UserFunction as an input argument. As mentioned before, UserFunction is an abstract class so it is necessary to override the call() method when declaring an object of this class. Figures 12 and 13 are two analogue examples implementing a user-defined Op equivalent to the standard MPI.SUM operation for double arguments.

A UserFunction may be implemented with arrays or with direct buffers irrespective of whether the buffers are direct or arrays in the reduction call. This versatility is possible because user functions are called from the C side, where the buffers are in the C heap and so it is cheap to create direct buffers from them. Therefore, it is always recommended to use direct buffers to implement user functions.

3.10. Intra-communicators and Inter-communicators

There are two predefined intra-communicators in MPI: COMM_WORLD and COMM_SELF. New intra-communicators can be created from a Group or from another intra-communicator (e.g., with split()). In both cases, the operation may return a null communicator. This situation can be detected as in the following code excerpt:

```
newcomm = MPI.COMM_WORLD.split(color, key);
if (!newcomm.isNull()) {  // the local process belongs to newcomm
```

The concept of process topologies defined in the MPI specification is closely related to intra-communicators. In Java, they are implemented as subclasses derived from the Intracomm class. CartComm provides the functionality for the Cartesian grid topology, where each process belonging to the grid has unique coordinates (in one, two, three or more dimensions). The example in Fig. 14 illustrates its usage with a 2-D torus where each process sends a value to its left neighbor. The other subclass, GraphComm, implements the more general graph topology.

For the case of inter-communicators, there are several methods that are specific to them, such as getRemoteSize(), getRemoteGroup(), getParent(), and merge(). The latter creates an Intracomm from an Intercomm.

Inter-communicators are used when new MPI processes must be created dynamically at run time with spawn(), as in the following code fragment:

```
int size = MPI.COMM_WORLD.getSize() ;
int dims[] = { 0, 0 };
CartComm.createDims(size, dims);
boolean periods[] = { true, true };
CartComm comm = ((Intracomm)MPI.COMM_WORLD).createCart(dims, periods, false);
int me = comm.getRank();
int coords[] = comm.getCoords(me);
CartParms topoParams = comm.getTopo();
System.out.println("I have coordinates ("+ coords[0] +","+ coords[1]
     +") on a grid of "+ topoParams.getDim(0) +"x"+ topoParams.getDim(1));

ShiftParms shiftParams = comm.shift(0, -1);
int src = shiftParams.getRankSource();
int dst = shiftParams.getRankDest();
int sbuf[] = { me };
int rbuf[] = new int[1];
comm.sendRecv(sbuf, 1, MPI.INT, dst, 1, rbuf, 1, MPI.INT, src, 1);
System.out.println("I am process "+ me +", received "+ rbuf[0]);
```

Figure 14: Example illustrating the use of the Cartesian topology.

```
String spawn_argv[] = {
    Spawn.class.getName(), // Class to execute
    "-file",
    "results.dat"
};
int maxprocs = 6;
int errcode = new int[maxprocs];
Intercomm child = MPI.COMM_WORLD.spawn("java", spawn_argv, maxprocs, MPI.INFO_NULL, 0, errcode);
```

Note that the name of the executable must always be `java` (in Java MPI applications, not in the case of C/Fortran), with the first argument being the name of the class to run (`Spawn` in this example).

Also, inter-communicators may be necessary in the context of MPI client-server applications, where a group of processes connect another group via a port to establish communication. Fig. 15 shows a simple example that uses service name publication.

3.11. Parallel I/O

Since the 2.0 specification, MPI supports parallel I/O by means of a high-level interface providing collective operations to perform transfers of global data structures between process memories and files. In MPI, partitioning of file data among processes and disks is expressed using derived datatypes, and the concept of file views. With this, virtually any I/O pattern can be implemented, such as strided access, etc.

In brief, a file view is specified by a displacement relative to the beginning of the file, the elementary datatype (the unit of data access and positioning), and the filetype that defines a template for accessing the file. Fig. 16 shows how to use `setView()` in such a way that each process loads a fragment of a file. The last argument of `setView()` is a string denoting the data representation, and is intended for file interoperability.

There are many data access routines available, which can be classified by the scheme that is used for positioning: either with explicit offsets, with individual file pointers, or with a shared file pointer. For each of these categories, there are blocking and non-blocking primitives, as well as collective and non-collective. For instance, `iReadAt()` is the non-collective and non-blocking primitive for reading a file with explicit offset.

3.12. One-sided communication

In the Open MPI Java bindings we also include a `Win` class that provides the Remote Memory Access (RMA) functionality offered by the MPI-2 and MPI-3 specifications. One-sided communication primitives such as `put()`, `get()`, and `accumulate()` operate on an object that has been previously declared with the constructor `Win()`, which specifies a window of existing memory that is exposed to RMA accesses by the processes in a given communicator. Note that windows must always be created from direct buffers, while arrays are not supported in the context of one-sided communication.

```
if (rank == 0) {
  // Server code
  String portName = Intracomm.openPort();

  Intracomm.publishName(serviceName, portName);
  MPI.COMM_WORLD.barrier();

  Comm client = MPI.COMM_SELF.accept(portName, 0);
  client.recv(buf, 1, MPI.INT, 0, 1);
  client.disconnect();

  Intracomm.closePort(portName);
  Intracomm.unpublishName(serviceName, portName);
}
else if (rank == 1) {
  // Client code
  MPI.COMM_WORLD.barrier();
  String portName = Intracomm.lookupName(serviceName);

  // connect to the server
  Comm server = MPI.COMM_SELF.connect(portName, 0);
  server.send(buf, 1, MPI.INT, 0, 1);
  server.disconnect();
}
```

Figure 15: Example client-server code based on publishing a service name.

```
int myrank = MPI.COMM_WORLD.getRank();
int numprocs = MPI.COMM_WORLD.getSize();
File thefile = new File(MPI.COMM_SELF, filename, MPI.MODE_RDONLY);
long filesize = thefile.getSize() / 4;     // num of ints in file
int bufsize = (int)(filesize / numprocs);  // local num of ints to read
int buf = new int[bufsize];
thefile.setView(myrank * buf.length, MPI.INT, MPI.INT, "native");
thefile.read(buf, bufsize, MPI.INT);
thefile.close();
```

Figure 16: Example code that reads a binary file containing integers.

Fig. 17 shows a simple example using one-sided communication. The example uses `fence()` for synchronization. Other synchronization schemes based on `post()`/`waitFor()` and `lock()`/`unlock()` are also available.

3.13. Thread support

The Java bindings are thread safe, and adhere to the level of thread safety that the underlying Open MPI provides.

We have added the `MPI.InitThread()` initialization method, whose parameter `required` indicates the desired level of thread support, either `THREAD_SINGLE`, `THREAD_FUNNELED`, `THREAD_SERIALIZED`, or `THREAD_MULTIPLE`.

Threads in Java are created with standard mechanism, e.g., with method `start()` of a class that extends `Thread`. We have added `MPI.isThreadMain()` to distinguish among the main thread and the rest. There is also support for "M" variants of probe and receive primitives, as discussed already in Section 3.4.

4. Details of internal implementation

We now discuss some aspects of how the Java bindings are implemented under the hood. This may help to better understand why the user interface is the way it is, and what needs to be done to achieve a certain effect on the API. In some cases, we compare our approach with respect to the solution proposed in the original mpiJava implementation.

```
int rank = MPI.COMM_WORLD.getRank();
int size = MPI.COMM_WORLD.getSize();
IntBuffer winArea = MPI.newIntBuffer(size),
          putVals = MPI.newIntBuffer(size);

Win win = new Win(winArea, size, 1, MPI.INFO_NULL, MPI.COMM_WORLD);

// Set all the target areas to be -1
for (int i = 0; i < size; ++i) {
  winArea.put(i, -1);
  putVals.put(i, rank);
}
win.fence(0);

// Do a put to all other processes
for (int i = 0; i < size; ++i)
  win.put(slice(putVals, i), 1, MPI.INT, i, rank, 1, MPI.INT);
win.fence(0);

// Check to see that we got the right values
for (int i = 0; i < size; ++i)
  if (winArea.get(i) != i)
    System.out.println("Unexpected value!");

win.free();
```

Figure 17: Simple one-sided communication example.

4.1. Constant fields and other details of the MPI class

Constant fields such as `MPI.ANY_SOURCE` are initialized in two phases in order to make them effectively constant (`final` in Java). First the C native value (e.g., `MPI_ANY_SOURCE`) is copied via JNI in a member field of a private object of class `Constant`, and secondly this value is copied to the `final` data member. These two steps are carried out during the static initialization of class `MPI`. In the original mpiJava implementation, these fields were not `final`, which did not prevent their values being accidentally changed by the user. A similar process is done also for predefined objects such as datatypes (`MPI.DOUBLE`) and communicators (`MPI.COMM_SELF`).

All Java methods call `MPI.check()` to ensure that MPI has been initialized. If it was not, then an `MPIException` is thrown. Fig. 1 shows the internal implementation of a sample method of the Java bindings.

In the original mpiJava implementation there were explicit castings in C from `jint*` (Java integer pointer) to `int*` (C integer pointer). But the size of `jint` and `int` may be different in some systems. Our implementation fixes this problem, see Fig. 18.

4.2. Memory management

Since the concept of pointers is not available in Java, some tricks are required for memory management.

Some garbage collectors (GC) support *pinning*, an operation that prevents the GC from relocating an object until it is *unpinned*. This allows JNI code to obtain the real reference to a *pinned* object via the JNI functions `Get/Release[Type]ArrayElements()`, thus avoiding unnecessary copies. Unfortunately modern GCs do not support *pinning* because it hinders the GC and tends to cause memory fragmentation. Therefore, these functions are not a good solution to avoid data copies.

The functions `Get/ReleasePrimitiveArrayCritical()` also provide real pointer references to buffers. However, these methods have restrictions that *require* making copies (e.g., no JNI calls can be made between these methods). We use this mechanism only in the `pack()`/`unpack()` primitives.

As described in Section 3.6, we have opted for using direct buffers for the case that the number of data copies needs to be reduced, but still allow the user to employ regular arrays as buffers, since accessing direct buffers may be slower than accessing arrays. The original mpiJava package did not support direct buffers, because they were not available at the time it was written.

```
void ompi_java_getIntArray(JNIEnv *env, jintArray array, jint **jptr, int **cptr)
{
    jint *jInts = (*env)->GetIntArrayElements(env, array, NULL);
    *jptr = jInts;

    if(sizeof(int) == sizeof(jint))
    {
        *cptr = (int*)jInts;
    }
    else
    {
        int i, length = (*env)->GetArrayLength(env, array);
        int *cInts = calloc(length, sizeof(int));

        for(i = 0; i < length; i++)
            cInts[i] = jInts[i];

        *cptr = cInts;
    }
}
```

Figure 18: Internal conversion routine between Java and C integers.

The internal memory management depends on how the user has defined the buffers. If a buffer argument is an array, in some situations it will be possible to use the JNI functions `Get/ReleasePrimitiveArrayCritical()`, but in most cases it will be necessary to perform a costly copy of the contents of the array. This operation checks if the array datatype is noncontiguous in order to avoid copying the data gaps. When the buffer is direct the method `GetDirectBufferAddress()` provides the real pointer to the buffer.

The reason why all non-blocking methods in Open MPI require direct buffers is that in the case of Java arrays, internally the Java bindings copy the whole array, so different non-blocking operations acting on different offsets of the array would result in inconsistent results.

4.3. In-place communication primitives

`MPI_IN_PLACE` is implemented with overloaded methods instead of defining a constant for buffers, because in Java overloading methods is preferable to passing some parameters while ignoring other ones such as the count parameter.

For example, in the class `Comm`, the method

```
void gatherv(Object sendbuf, int sendcount, Datatype sendtype,
             Object recvbuf, int[] recvcount, int[] displs, Datatype recvtype, int root)
```

has the following in-place method for the root process:

```
void gatherv(Object recvbuf, int[] recvcount, int[] displs, Datatype recvtype, int root)
```

and the following in-place method for the non-root processes:

```
void gatherv(Object sendbuf, int sendcount, Datatype sendtype, int root)
```

Even when there is only one in-place version of a method, it must be called with certain parameters depending on whether the process is root or not. For example, the buffer parameter in the in-place version of `gather` is used by the root process to receive data, and by the non-root processes to send data.

```
if(myself == root)
    MPI.COMM_WORLD.gather(in, i, MPI.INT, root);
else
    MPI.COMM_WORLD.gather(out, i, MPI.INT, root);
```

4.4. Object disposal

In the original mpiJava implementation, MPI objects were stored in a list of garbage collected objects in order to be freed in a next call of an MPI method. This feature was removed because some methods to free MPI objects are collective and so race conditions may be provoked.

4.5. Returning structured data

In the C language, apart from the return value, a function can give back other computed values by means of reference arguments using pointers. In Java, the arguments of a method are always passed by value. It is only possible to alter a method parameter if it is a mutable object, i.e., its internal representation may be changed. However, it is not possible to return a new object using the same parameter, because the object reference is also passed by value.

Reference parameters could be simulated in Java using an object which contains the reference object inside. But this would be against the Java philosophy. In our implementation, when a method needs to return several values, these values are encapsulated in an auxiliary object:

- `GraphParms` encapsulates the topology information associated with a communicator, which can be obtained by the method `GraphComm.getDims()`.
- `CartParms` encapsulates the information related to the Cartesian topology, which is returned by the method `CartComm.getTopo()`.
- `ShiftParms` encapsulates the source and destination ranks for "shift" communication, which are computed by the method `CartComm.shift()`.

The Java binding of `MPI_Status` is the `Status` class. When an `MPI_Status` is created in C and returned to Java, the whole thing is copied in order to avoid having to release `Status` objects and calling JNI methods to get their values.

4.6. Specificities to Open MPI

As mentioned previously, our Java bindings benefit from accessing the internal Open MPI data structures, resulting in a cleaner integration. However, the most part of the bindings are independent of Open MPI and could thus be reused in an eventual migration to other popular MPI implementations, such as MPICH[2]. We next mention the few implementation details that are dependent on Open MPI internals.

In different situations, the Java bindings need to use temporary memory space, for example to set up buffers in the C side that hold elements of Java arrays passed in by the user. In order to avoid frequent calls to the system's memory allocation functions (`malloc`), we make use of a pooled memory management layer provided by Open MPI (`free_list`). Also, copies between the Java and the C side are optimized by making use of internal Open MPI functions that handle non-contiguous datatypes.

When employing user-defined operations for collective operations, as shown in Section 3.9, the Java bindings must be able to invoke the user callbacks. Open MPI maintains a list of user callback functions that is language-aware, so we just extended it to allow calling Java code by storing all the necessary information (a reference to the object, the JNI environment and the base type of the involved buffer).

Finally, as discussed in the previous subsection, `Status` objects contain a copy of all data stored in the opaque C struct `MPI_Status`, and hence this copy requires knowledge of the internal definition, which is not specified by the MPI standard.

5. Performance evaluation

The Open MPI Java bindings were first presented in the EuroMPI '13 conference [26], and were later improved and optimized to its present form included in Open MPI version 1.8.1. In this section, we evaluate this version to assess the performance overhead incurred by using the Java bindings rather than the underlying C library directly.

The analysis is based on basic bandwidth measurement in simple point-to-point communication, on one hand, and on well established benchmarks, on the other. Initially, we considered the Java Grande benchmarks [27], but then we decided to use the Java port of the NAS Parallel Benchmarks, NPB [28], since they represent better the kind of applications we are aiming at. Our analysis is similar to the one described in [29]. In our case, we compare the Java benchmarks against the original C/Fortran benchmarks, not against other implementations of the MPI Java bindings

[2] `http://www.mpich.org`

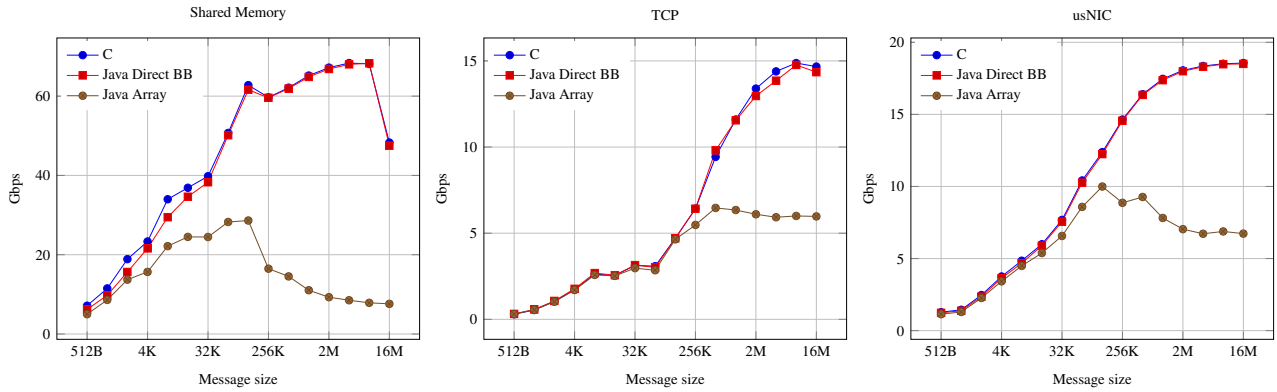

Figure 19: Bandwidth expressed in Gbps of point-to-point communication for varying message size in different transports: shared memory (left), TCP over Gigabit Ethernet (middle), and usNIC network interface (right). Plots compare native C Open MPI against the Java bindings using either the direct buffer or the array interface.

Table 2: Latency (in μsec.) of point-to-point communication in different transports.

	Shared Memory	TCP	usNIC
C	0.41	23.86	2.42
Java Direct BB	0.88	23.73	3.12
Java Array	1.01	23.86	3.20

(for this, see the study performed by other independent authors [30], that concludes recommending our Java bindings for their class of applications).

The evaluation has been carried out on a cluster consisting of 32 Sandy Bridge class servers, each of them with two 8-core processors (i.e., 16 cores per server), with 128 GB of RAM. They all have 3×10 GB Cisco usNIC interfaces, apart from a standard Gigabit Ethernet connection. The compilers used were GCC 4.8.2 and Java OpenJDK 1.6.

5.1. Bandwidth tests

In order to assess the potential performance loss of the Java bindings with respect to the native C implementation in the context of point-to-point communication, we measure the bandwidth using the OSU MPI micro-benchmarks[3] for various transports (shared memory, TCP and usNIC). We made two Java versions of the ping-pong benchmark, one using direct buffers and the other using plain Java arrays. Figure 19 shows the bandwidth in Gbps measured for messages up to 16 MB.

Bandwidth results indicate that the performance loss with respect to the C implementation is significant only if using the array interface in the Java bindings. As previously discussed, this approach requires extra copies of the data, which involves a big performance penalty especially for large messages. However, the direct buffer interface does not incur this overhead so performance is remarkably close to the C one. The small remaining overhead can be attributed to language change via JNI.

Table 2 shows latency, measured as the time to send a message of 1 byte. Results indicate that latency-bound applications will see a significant performance penalty compared to the C implementation, especially in the case of shared memory.

[3] `http://mvapich.cse.ohio-state.edu/benchmarks`

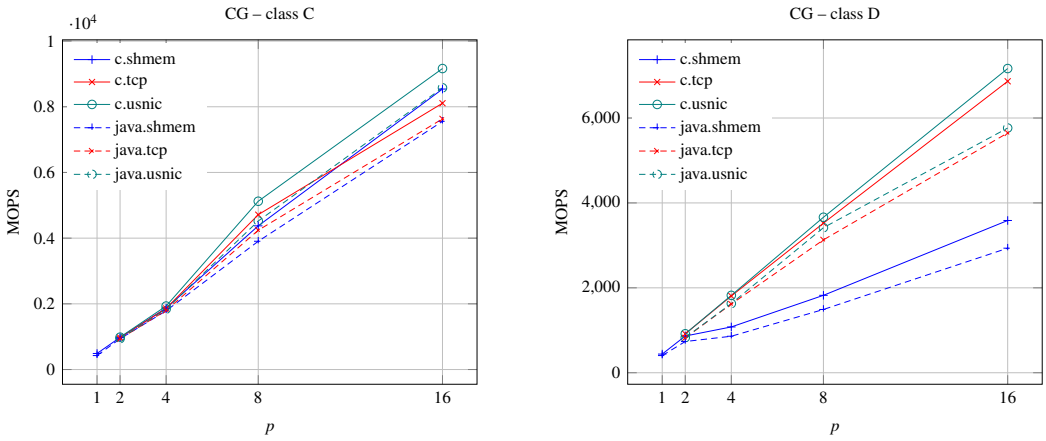

Figure 20: Results of NPB CG benchmark, for class C (left) and D (right).

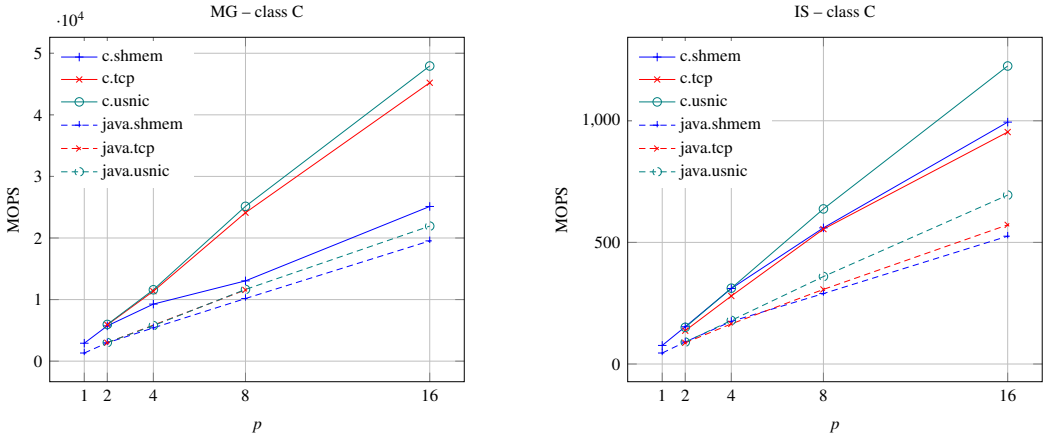

Figure 21: Results of NPB benchmarks: MG (left) and IS (right).

5.2. NAS Parallel Benchmarks

For a more complete evaluation of the Java bindings, including collective communication primitives, we run some of the NAS parallel benchmarks, in particular their Java ports described in [28]. We had to make minor changes in the source code in order to adapt them to our Java bindings, which differ slightly from the MPJ API. Also, following the bandwidth results discussed above, we adapted the benchmarks to use direct buffers in cases where message size is potentially large.

All benchmarks are run with 1 up to 16 MPI processes, which may be placed either in the same cluster node (shared memory) or one per node (TCP and usNIC). The benchmarks report whether the computation was successful, together with the value of MOPS (Millions of Operations Per Second), which will be the metric used for comparison. We use class C benchmarks (and also class D in some cases), which correspond to considerably large problem sizes.

Figure 20 shows the results for the CG benchmark, which runs the conjugate gradient iteration on a sparse linear system of equations of size 150,000 (class C) or 1.5 million (class D). In class D, the matrix has relatively more nonzero elements compared to class C, and this may explain why MOPS values are smaller in class D. Anyway, if we compare the performance of the Java version against the Fortran implementation, we see a small performance loss that does not change the trend when the number of processes is increased. Hence, in this case the Java benchmark is doing remarkably well, in terms of the performance of MPI operations as well as in what refers to efficiency of floating-point computation.

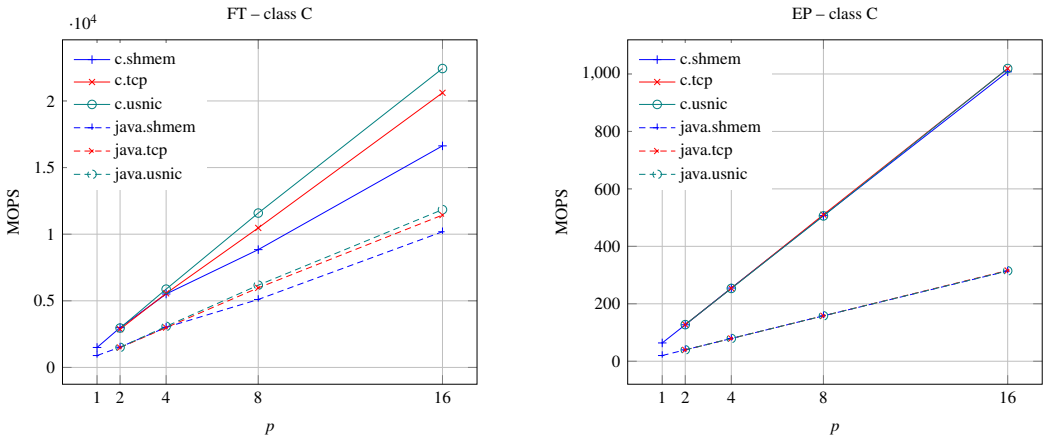

Figure 22: Results of NPB benchmarks: FT (left) and EP (right).

The situation with other benchmarks is quite different. Figures 21 and 22 show results for the benchmarks MG, IS, FT and EP. In all cases, it is evident that the Java versions of the benchmarks have lost the battle against their native C/Fortran counterparts. A closer look indicates that the reason behind this has nothing to do with the efficiency of the MPI bindings. For instance, EP is the embarrassingly parallel benchmark, that hardly uses MPI primitives for communication. As expected, the Fortran version scales linearly with the number of MPI processes, but we can see that the Java version also scales linearly, although with much smaller MOPS values. Hence, poor performance should be attributed to how the Java compiler and runtime are managing the local computation at each process.

6. Conclusion

We have presented details of the design and implementation of the Java bindings that we provide as part of the Open MPI distribution. The starting point for its development was the JNI-based mpiJava project, whose source code was totally reworked, optimized, customized for the Open MPI internals, and extended to latest MPI-3 features. The resulting interface is user-friendly from the point of view of a Java programmer. We provide two ways of managing buffers: trivially with one-dimensional Java arrays and by creating a direct buffer object. Using one or the other will be more appropriate depending on the context, specifically the length of the message to be sent.

The performance results for the Java benchmarks discussed in section 5 may seem disappointing. It is clear that in a one-to-one mapping between C/Fortran code and its Java equivalent, the former is likely to reach better result in terms of percentage of peak performance. This should not be understood as a reason for not considering Java as a good language for high-performance computing applications. In many cases, performance of Java will be close to that of C/Fortran, as illustrated in the CG benchmark. Furthermore, the Java MPI bindings can be very valuable for the parallelization of applications already written in Java, irrespective of their sequential performance.

It is not our aim to compete with pure-Java offerings for MPI bindings, but just to widen the availability of the powerful MPI functionality to Java programmers and applications. There are already users that are employing the Open MPI Java bindings for their applications [31], and we hope the number of users will increase as newer versions of Open MPI start to pervade many supercomputers across the world.

Acknowledgements. We are indebted to Siegmar Groß for his exhaustive testing of the Java bindings. We also thank Ralph Castain for helping in the integration of the Java bindings in the Open MPI infrastructure. The NPB-MPJ benchmarks used in section 5 were kindly provided by Guillermo López Taboada. The first two authors were supported by the Spanish Ministry of Economy and Competitiveness under project number TIN2013-41049-P.

References

[1] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbe, F. Huet, G. L. Taboada, Current state of Java for HPC, Tech. Rep. 0353, INRIA (2008).

[2] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, R. Doallo, Java in the high performance computing arena: Research, practice and experience, Science of Computer Programming 78 (5) (2013) 425–444.
[3] B. Blount, S. Chatterjee, An evaluation of Java for numerical computing, in: D. Caromel, R. R. Oldehoeft, M. Tholburn (Eds.), Computing in Object-Oriented Parallel Environments, Vol. 1505 of Lecture Notes in Computer Science, Springer, 1998, pp. 35–46.
[4] P. Knoll, S. Mirzaei, Scientific computing with Java, Computer Applications in Engineering Education 18 (3) (2010) 495–501.
[5] B. Oancea, I. G. Rosca, T. Andrei, A. I. Iacob, Evaluating Java performance for linear algebra numerical computations, Procedia Computer Science 3 (2011) 474–478.
[6] B.-O. Heimsund, Matrix toolkits for Java, https://github.com/fommil/matrix-toolkits-java, accessed December 2013.
[7] D. M. Doolin, J. Dongarra, K. Seymour, JLAPACK–compiling LAPACK Fortran to Java, Scientific Programming 7 (2) (1999) 111–138.
[8] A. Fries, The use of Java in large scientific applications in HPC environments, Ph.D. thesis, Universitat de Barcelona, http://hdl.handle.net/10803/98405, accessed December 2013 (2013).
[9] W. B. VanderHeyden, E. D. Dendy, N. T. Padial-Collins, Cartablanca—a pure-Java, component-based systems simulation tool for coupled nonlinear physics on unstructured grids—an update, Concurrency and Computation: Practice and Experience 15 (3-5) (2003) 431–458.
[10] A. Shafi, B. Carpenter, M. Baker, A. Hussain, A comparative study of Java and C performance in two large-scale parallel applications, Concurrency and Computation: Practice and Experience 21 (15) (2009) 1882–1906.
[11] W. F. Godoy, X. Liu, Parallel Jacobian-free Newton Krylov solution of the discrete ordinates method with flux limiters for 3D radiative transfer, Journal of Computational Physics 231 (11) (2012) 4257–4278.
[12] P. Wendykier, J. G. Nagy, Parallel Colt: A high-performance Java library for scientific computing and image processing, ACM Transactions on Mathematical Software 37 (3) (2010) 31:1–31:22.
[13] Y. Yan, M. Grossman, V. Sarkar, JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA, in: H. Sips, D. Epema, H.-X. Lin (Eds.), Euro-Par 2009 Parallel Processing, Vol. 5704 of Lecture Notes in Computer Science, Springer, 2009, pp. 887–899.
[14] M. Nowicki, P. Bala, Parallel computations in Java with PCJ library, in: International Conference on High Performance Computing and Simulation (HPCS), 2012, pp. 381–387.
[15] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.
[16] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, version 3.0 (2012).
[17] S. Liang, The Java Native Interface: Programmer's Guide and Specification, Addison-Wesley, 1999.
[18] M. Baker, B. Carpenter, G. Fox, S. Hoon Ko, S. Lim, mpiJava: An object-oriented Java interface to MPI, in: Parallel and Distributed Processing, Vol. 1586 of Lecture Notes in Computer Science, Springer, 1999, pp. 748–762.
[19] W. Pugh, J. Spacco, MPJava: High-performance message passing in Java using Java.nio, in: L. Rauchwerger (Ed.), Languages and Compilers for Parallel Computing, Vol. 2958 of Lecture Notes in Computer Science, Springer, 2004, pp. 323–339.
[20] A. Shafi, B. Carpenter, M. Baker, Nested parallelism for multi-core HPC systems using Java, Journal of Parallel and Distributed Computing 69 (6) (2009) 532–545.
[21] G. L. Taboada, J. Touriño, R. Doallo, F-MPJ: scalable Java message-passing communications on parallel systems, The Journal of Supercomputing 60 (1) (2012) 117–140.
[22] G. L. Taboada, J. Touriño, R. Doallo, Java Fast Sockets: Enabling high-speed Java communications on high performance clusters, Computer Communications 31 (17) (2008) 4049–4059.
[23] S. Ramos, G. L. Taboada, R. R. Expósito, J. Touriño, R. Doallo, Design of scalable Java communication middleware for multi-core systems, The Computer Journal 56 (2) (2013) 214–228.
[24] B. Carpenter, G. Fox, S.-H. Ko, S. Lim, mpiJava 1.2: API specification, http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html, accessed December 2013 (2002).
[25] B. Carpenter, V. Getov, G. Judd, A. Skjellum, G. Fox, MPJ: MPI-like message passing for Java, Concurrency: Practice and Experience 12 (11) (2000) 1019–1038.
[26] O. Vega-Gisbert, J. E. Roman, S. Groß, J. M. Squyres, Towards the availability of Java bindings in Open MPI, in: Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, ACM, 2013, pp. 141–142.
[27] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, R. A. Davey, A benchmark suite for high performance Java, Concurrency: Practice and Experience 12 (6) (2000) 375–388.
[28] D. A. Mallon, G. L. Taboada, J. Touriño, R. Doallo, NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java, in: Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09), Weimar, Germany, 2009, pp. 181–190.
[29] B. Qamar, A. Javed, M. Jameel, A. Shafi, B. Carpenter, Design and implementation of hybrid and native communication devices for Java HPC, Procedia Computer Science 29 (2014) 184–197.
[30] S. Ekanayake, G. Fox, Evaluation of Java message passing in high performance data analytics, http://grids.ucs.indiana.edu/ptliupages/publications/EvaluationOfJava_ieeeformat.pdf, accessed June 2015 (2014).
[31] Ö. Demirel, I. Smal, W. J. Niessen, E. Meijering, I. F. Sbalzarini, PPF - a parallel particle filtering library, arXiv preprint 1310.5045.