mpiJava 1.2: API Specification

Bryan Carpenter, Geoffrey Fox,
Sung-Hoon Ko, Sang Lim

Northeast Parallel Architectures Centre,
Syracuse University,
111 College Place,
Syracuse, New York 13244-410
{dbc,gcf,shko,slim}npac.syr.edu

This document defines the API of *mpiJava*, a Java language binding for MPI 1.1. The document is not a standalone specification of the behaviour of MPI—it is meant to be read in conjunction with the MPI standard document [2]. Subsections are laid out in the same way as in the standard document, to allow cross-referencing. Where the mpiJava binding makes no significant change to a particular section of the standard document, we will just note here that there are no special issues for the Java binding. This does not mean that the corresponding section of the standard is irrelevant to the Java binding—it may mean it is 100% relevant! Where practical the API is modelled on the MPI C++ interface defined in the MPI standard version 2.0 [3].

Changes to the mpiJava 1.1 interface:

- The `MPI.OBJECT` basic type has been added.
- The interface for `MPI.Buffer_detach` has been corrected.
- The API of `User_function` has been changed.
- Attributes cached in communicators are now assumed to have integer values. `Attr_put` and `Attr_delete` have been removed.
- The interface to `Cartcomm.dimsCreate` has been corrected.
- `Errorhandler_set`, `Errorhandler_get` have changed from static members of `MPI` to instance methods on `Comm`.
- The method `Is_null` has been added to class `Comm`.
- The initialization method `MPI.Init` now returns the command line arguments set by MPI.
- MPI exception classes are now specified to be subclasses of `MPIException` rather than `IOException`. Methods are now declared to throw `MPIException` (see section 8).

The current API is viewed as an interim measure. Further significant changes are likely to result from the efforts of the Message-passing working group of the Java Grande Forum.

Contents

1 Introduction to MPI

Evidently, this document adds Java to the C and Fortran bindings defined in the MPI standard. Otherwise no special issues for the Java binding.

2 MPI Terms and Conventions

2.1 Document Notation

No special issues for Java binding.

2.2 Procedure Specification

In general we use *italicized* names to refer to entities in the MPI language independent procedure definitions, and `typewriter` font for concrete Java entities.

As a rule Java argument names are the same as the corresponding language independent names. In instance methods of `Comm`, `Status`, `Request`, `Datatype`, `Op` or `Group` (and subclasses), the class instance generally stands for the argument called *comm*, *status*, *request*, *datatype*, *op* or *group*, respectively in the language independent procedure definition.

2.3 Semantic terms

No special issues for Java binding.

2.4 Data types

Opaque objects are presented as Java objects. This introduces the option of simplifying the user's task in managing these objects. MPI destructors can be absorbed into Java object destructors, which are called automatically by the Java garbage collector. We adopt this strategy as the general rule. Explicit calls to MPI destructor functions are typically omitted from the Java user interface (they are absorbed into `finalize` methods). Exceptions are made for the `Comm` and `Request` classes. *MPI_COMM_FREE* is a collective operation, so the user must ensure that calls are made at consistent times by all processors involved—the call can't be left to the vagaries of the garbage collector. A similar case can be made for *MPI_REQUEST_FREE*.

2.5 Language Binding

Naming Conventions All MPI classes belong to the package `mpi`. Conventions for capitalization, etc, in class and member names generally follow the C++ MPI bindings.

Restrictions on *struct* derived type. Some options allowed for *derived data types* in the C and Fortran binding are deleted in mpiJava. The Java VM does not incorporate a concept of a global linear address space. Passing physical addresses to data type definitions is not allowed. The use of the *MPI_TYPE_STRUCT* datatype constructor is also restricted in a way that makes it impossible to send mixed *basic datatypes* in a single message. Since, however, the set of basic datatypes recognised by MPI is extended to include serializable Java *objects*, this should not be a serious restriction in practice.

Multidimensional arrays and offsets. The C and Fortran languages define a straightforward mapping (or "sequence association") between their multidimensional arrays and equivalent one-dimensional arrays. So in C or Fortran a multidimensional array passed as a message buffer argument is first interpreted as a one-dimensional array with the same element type as the original multidimensional array. Offsets in the buffer (such as offsets occuring in derived data types) are then interpretted in terms of the effective one-dimensional array (or—equivalent up to a constant factor—in terms of physical memory). In Java the relationship between multidimensional arrays and one dimensional arrays is different. An "n-dimensional array" is equivalent to a one-dimensional array of $(n-1)$-dimensional arrays. In mpiJava, message buffers are always one-dimensional arrays. The element type *may* be an object, which *may* have array type. Hence multidimensional arrays can appear as message buffers, but the interpretation is subtly different. In distinction to the C or Fortran case *offsets in multidimensional message buffers are always interpretted as offsets in the outermost one-dimensional array.*

Start of message buffer. C and Fortran both have devices for treating a section of an array, offset from the beginning of the array, as if it was an array in its own right. Java doesn't have any such mechanism. To provide the same flexibility, an `offset` parameter is associated with any buffer argument. This defines the position of the first actual buffer element in the Java array.

Error codes. Unlike the standard C and Fortran interfaces, the mpiJava interfaces to MPI calls do not return explicit error codes. The Java exception mechanism will be used to report errors.

> *Rationale.* The exception mechanism is very widely used by Java libraries. It is inconvenient to use up the single return value of a Java function with an error code. (Java doesn't allow function arguments to be passed by reference, so returning multiple values tends to be more clumsy than in other languages.) *(End of rationale.)*

Multiple return values. A few functions in the MPI interface return multiple values, even after the error code is eliminated. This is dealt with in mpiJava in various ways. Sometimes an MPI function initializes some elements in an array and also returns a count of the number of elements modified. In Java we typically return an array result, omitting the count. The count can be obtained subsequently from the `length` member of the array. Sometimes an MPI function initializes an object conditionally and returns a separate flag to say if the operation succeeded. In Java we typically return an object reference which is `null` if the operation fails. Occasionally extra internal state is added to an existing MPI class to hold extra results—for example the `Status` class has extra state initialized by functions like `Waitany` to hold the *index* value. Rarely none of these methods work and we resort to defining auxilliary classes to hold multiple results from a particular function.

Array count arguments. The mpiJava binding often omits array size arguments, because they can be picked up within the function by reading the `length` member of the array argument. A major exception is for message buffers, where an explicit count is always given.

> *Rationale.* In the mpiJava, message buffers have explicit `offset` and `count` arguments whereas other kinds of array argument typically do not. Message buffers aside, typical array arguments to MPI functions (eg, vectors of request structures) are small arrays. If subsections of these must be passed to an MPI function, the sections can be copied to smaller arrays at little cost. In contrast message buffers are typically large and copying them is expensive, so it is worthwhile to pass the extra arguments. Also, if derived data types are being used, the required value of the `count` argument is always different to the buffer length. *(End of rationale.)*

Concurrent access to arrays. In JNI-based wrapper implementations it may be necessary to impose some non-interference rules for concurrent read and write operations on arrays. When an array is passed to an MPI method such as a send or receive operation, the wrapper code will probably extract a pointer to the contents of the array using a JNI `Get...ArrayElements` routine. If the garbage collector *does not* support "pinning" (temporarily disabling run-time relocation of data for specific arrays—see [1] for more discussion), the pointer returned by this `Get` function may be to a temporary copy of the elements. The copy will be written back to the true Java array when a subsequent call to `Release...ArrayElements` is made. If two operations involving the same array are active concurrently, this copy-back may result in failure to save modifications made by one or more of the concurrent calls.

Such an implementation may have to enforce a safety rule such as: *when several MPI send or receive (etc) operations are active concurrently, if any one of those operations writes to a particular array, none of the other operations must read or write* any portion *of that array.*

If the garbage collector supports pinning, this problem does not arise.

2.6 Processes

No special issues for Java binding.

2.7 Error Handling

As explained in section 2.5, the Java methods do not return error codes. The Java exceptions thrown instead are defined in section 7.3.

3 Point-to-Point Communication

3.1 Introduction

In general the mpiJava binding of point-to-point communication operations realizes the MPI functions as methods of the `Comm` class. The basic point-to-point communication operations are *send* and *receive*. Their use is illustrated in the example below.

```
import mpi.* ;

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

    int myrank = MPI.COMM_WORLD.Rank() ;
    if(myrank == 0) {
      char [] message = "Hello, there".toCharArray() ;
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 99) ;
    }
    else {
      char [] message = new char [20] ;
      MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
      System.out.println("received:" + new String(message) + ":") ;
    }

    MPI.Finalize();
  }
}
```

3.2 Blocking Send and Receive operations

```
void Comm.Send(Object buf, int offset, int count,
               Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Blocking send operation. Java binding of the MPI operation *MPI_SEND*. The data part of the message consists of a sequence of `count` values, each of the type indicated by `datatype`. The actual argument associated with `buf` must be an array. The value `offset` is a subscript in this array, defining the position of the first item of the message.

The elements of `buf` may have primitive type or class type. If the elements are objects, they must be serializable objects. If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`: the basic MPI datatypes supported, and their correspondence to Java types, are as follows

MPI datatype	Java datatype
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.OBJECT	`Object`

If the `datatype` argument represents an MPI derived type, its *base type* must agree with the element type of `buf` (see section 3.12).

> *Rationale.* The `datatype` argument is not redundant in mpiJava, because we include support for MPI derived types. If it was decided to remove derived types from the API, `datatype` arguments could be removed from various functions, and Java runtime inquiries could be used internally to extract the element type of the buffer, or methods like `Send` could be overloaded to accept buffers with elements of the 9 basic types. *(End of rationale.)*

If a data type has `MPI.OBJECT` as its base type, the objects in the buffer will be transparently serialized and unserialized inside the communication operations.

```
Status Comm.Recv(Object buf, int offset, int count,
                 Datatype datatype, int source, int tag)
```

`buf`	receive buffer array
`offset`	initial offset in receive buffer
`count`	number of items in receive buffer
`datatype`	datatype of each item in receive buffer
`source`	rank of source
`tag`	message tag
returns:	status object

Blocking receive operation. Java binding of the MPI operation *MPI_RECV*. The actual argument associated with `buf` must be an array. The value `offset` is a subscript in this array, defining the position into which the first item of the incoming message will be copied.

The elements of `buf` may have primitive type or class type. If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`; if `datatype` represents an MPI derived type, its *base type* must agree with the element type of `buf` (see section 3.12).

The MPI constants *MPI_ANY_SOURCE* and *MPI_ANY_TAG* are available as `MPI.ANY_SOURCE` and `MPI.ANY_TAG`.

The source and tag of the received message are available in the publically accessible `source` and `tag` fields of the returned object. The following method can be used to further interrogate the return status of a receive operation.

```
int Status.Get_count(Datatype datatype)
```

`datatype`	datatype of each item in receive buffer
returns:	number of received entries

Java binding of the MPI operation *MPI_GET_COUNT*.

3.3 Data type matching and data conversion

The Java language definition places quite detailed constraints on the representation of its primitive types—for example it requires conformance with IEEE 754 for `float` and `double`. There may still be a requirement for representation conversion in heterogenous systems. For example, source and destination computers (or virtual machines) may have different endianess.

3.4 Communication Modes

```
void Comm.Bsend(Object buf, int offset, int count,
                Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Send in buffered mode. Java binding of the MPI operation *MPI_BSEND*. Further comments as for `send`.

```
void Comm.Ssend(Object buf, int offset, int count,
                Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Send in synchronous mode. Java binding of the MPI operation *MPI_SSEND*. Further comments as for `send`.

```
void Comm.Rsend(Object buf, int offset, int count,
                Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Send in ready mode. Java binding of the MPI operation *MPI_RSEND*. Further comments as for `send`.

3.5 Semantics of point-to-point communication

No special issues for Java binding.

3.6 Buffer allocation and usage

```
void MPI.Buffer_attach(byte [] buffer)
```

`buffer`	buffer array

Provides to MPI a buffer in user's memory to be used for buffering outgoing messages. Java binding of the MPI operation *MPI_BUFFER_ATTACH*.

```
byte [] MPI.Buffer_detach()
```

returns: buffer array

Detach the buffer currently associated with MPI and return it. Java binding of the MPI operation *MPI_BUFFER_DETACH*. If the currently associated buffer is system-defined, returns `null`.

The MPI constant *MPI_BSEND_OVERHEAD* is available as `MPI.BSEND_OVERHEAD`.

3.7 Nonblocking communication

Nonblocking communications use methods of the `Request` class to identify communication operations and match the operation that initiates the communication with the operation that terminates it.

```
Request Comm.Isend(Object buf, int offset, int count,
                   Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

returns: communication request

Start a standard mode, nonblocking send. Java binding of the MPI operation *MPI_ISEND*. Further comments as for `send`.

```
Request Comm.Ibsend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

returns: communication request

Start a buffered mode, nonblocking send. Java binding of the MPI operation *MPI_IBSEND*. Further comments as for `send`.

```
Request Comm.Issend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	communication request

Start a synchronous mode, nonblocking send. Java binding of the MPI operation *MPI_ISSEND*. Further comments as for `send`.

```
Request Comm.Irsend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	communication request

Start a ready mode, nonblocking send. Java binding of the MPI operation *MPI_IRSEND*. Further comments as for `send`.

```
Request Comm.Irecv(Object buf, int offset, int count,
                   Datatype datatype, int source, int tag)
```

`buf`	receive buffer array
`offset`	initial offset in receive buffer
`count`	number of items in receive buffer
`datatype`	datatype of each item in receive buffer
`source`	rank of source
`tag`	message tag
returns:	communication request

Start a nonblocking receive. Java binding of the MPI operation *MPI_IRECV*. Further comments as for `recv`.

The following functions are used to complete nonblocking communication operations (and also communications started using the persistent communication requests—subclass `Prequest`—introduced later). We use the following terminology. A request is "active" if it is associated with an ongoing communication. Otherwise it is inactive. An inactive instance of the base class `Request` is called a "void request". (Note, however, that an inactive instance of the `Prequest` subclass is not said to be "void", because it retains detailed information about a communication pattern even when no corresponding communication is ongoing.)

> *Rationale.* A "void request" corresponds to what is called a "null handle" in the C and Fortran MPI bindings. It seems impractical to have completion operations like `wait` set request object references to null references in the Java sense (because Java methods cannot directly modify references passed to them as arguments). To avoid a confusing semantic distinction between null MPI handles and null Java references we introduce the terminology of a "void request object". If an explicit reference to a void request is needed, one is available as `MPI.REQUEST_NULL`. The inquiry `Request.Is_null` can be used to determine if a particular request is void. *(End of rationale.)*

`Status Request.Wait()`

returns: status object

Blocks until the operation identified by the request is complete. Java binding of the MPI operation *MPI_WAIT*. After the call returns, the request object becomes inactive.

`Status Request.Test()`

returns: status object or null reference

Returns a status object if the operation identified by the request is complete, or a null reference otherwise. Java binding of the MPI operation *MPI_TEST*. After the call, if the operation is complete (ie, if the return value of `test` is non-null), the request object becomes an inactive request.

`boolean Request.Is_null()`

returns: true if the request object is void, false otherwise

Note that `Is_null` is always false on instances of the subclass `Prequest`.

```
void Request.Free()
```

Set the request object to be void. Java binding of the MPI operation `MPI_REQUEST_FREE`.

```
static Status Request.Waitany(Request [] array_of_requests)
```

`array_of_requests`	array of requests
returns:	status object

Blocks until one of the operations associated with the active requests in the array has completed. Java binding of the MPI operation *MPI_WAITANY*. The index in `array_of_requests` for the request that completed can be obtained from the status object from the publically accessible `Status.index` field. The corresponding element of `array_of_requests` becomes inactive.

The `array_of_requests` may contain inactive requests. If the list contains no active requests, the method immediately returns a status in which the `index` field is `MPI.UNDEFINED`.

```
static Status Request.Testany(Request [] array_of_requests)
```

`array_of_requests`	array of requests
returns:	status object or null reference

Tests for completion of either one or none of the operations associated with active requests. Java binding of the MPI operation *MPI_TESTANY*. If some request completed, the index in `array_of_requests` of that request can be obtained from the status object through the `Status.index` field. The corresponding element of `array_of_requests` becomes inactive. If no request completed, `testAny` returns a null reference.

The `array_of_requests` may contain inactive requests. If the list contains no active requests, the method immediately returns a status in which the `index` field is `MPI.UNDEFINED`.

```
static Status [] Request.Waitall(Request [] array_of_requests)
```

`array_of_requests`	array of requests
returns:	array of status objects

Blocks until all of the operations associated with the active requests in the array have completed. Java binding of the MPI operation *MPI_WAITALL*. The result array will be the same size as `array_of_requests`. On exit, requests become inactive. If the *input* value of `array_of_requests` contains any inactive requests, corresponding elements of the result array will contain null status references.

`static Status [] Request.Testall(Request [] array_of_requests)`

`array_of_requests`	array of requests
returns:	array of status objects, or a null reference

Tests for completion of *all* of the operations associated with active requests. Java binding of the MPI operation *MPI_TESTALL*. If all operations have completed, the exit values of the argument array and the result array are as for `Waitall`. If any operation has not completed, the result value is null and no element of the argument array is modified.

`static Status [] Request.Waitsome(Request [] array_of_requests)`

`array_of_requests`	array of requests
returns:	array of status objects

Blocks until at least one of the operations associated with the active requests in the array has completed. Java binding of the MPI operation *MPI_WAITSOME*. The size of the result array will be the number of operations that completed. The index in `array_of_requests` for each request that completed can be obtained from the `index` field of the returned status objects. The corresponding elements in `array_of_requests` become inactive.

If `array_of_requests` list contains no active requests, `testAll` immediately returns a null reference.

`static Status [] Request.Testsome(Request [] array_of_requests)`

`array_of_requests`	array of requests
returns:	array of status objects

Behaves like `waitSome`, except that it returns immediately. Java binding of the MPI operation *MPI_TESTSOME*. If no operation has completed, `Testsome` returns an array of length zero and elements of `array_of_requests` are unchanged. Otherwise, arguments and return value are as for `Waitsome`.

3.8 Probe and Cancel

`Status Comm.Iprobe(int source, int tag)`

`source`	source rank
`tag`	tag value
returns:	status object or null reference

Check if there is an incoming message matching the pattern specified. Java binding of the MPI operation *MPI_IPROBE*. If such a message is currently available, a status object similar to the return value of a matching `Recv` operation is returned. Otherwise a null reference is returned.

`Status Comm.Probe(int source, int tag)`

`source`	source rank
`tag`	tag value
returns:	status object or null reference

Wait until there is an incoming message matching the pattern specified. Java binding of the MPI operation *MPI_PROBE*. Returns a status object similar to the return value of a matching `Recv` operation.

`void Request.Cancel()`

Mark a pending nonblocking communication for cancellation. Java binding of the MPI operation *MPI_CANCEL*.

`boolean Status.Test_cancelled()`

returns:	true if the operation was succesfully cancelled, false otherwise

Test if communication was cancelled. Java binding of the MPI operation *MPI_TEST_CANCELLED*.

3.9 Persistent communication requests

```
Prequest Comm.Send_init(Object buf, int offset, int count,
                        Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a standard mode send. Java binding of the MPI operation *MPI_SEND_INIT*. Further comments as for `Send`.

```
Prequest Comm.Bsend_init(Object buf, int offset, int count,
                         Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a buffered mode send. Java binding of the MPI operation *MPI_BSEND_INIT*. Further comments as for `Send`.

```
Prequest Comm.Ssend_init(Object buf, int offset, int count,
                         Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a synchronous mode send. Java binding of the MPI operation *MPI_SSEND_INIT*. Further comments as for `Send`.

```
Prequest Comm.Rsend_init(Object buf, int offset, int count,
                         Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a ready mode send. Java binding of the MPI operation *MPI_RSEND_INIT*. Further comments as for `Send`.

```
Prequest Comm.Recv_init(Object buf, int offset, int count,
                        Datatype datatype, int source, int tag)
```

`buf`	receive buffer array
`offset`	initial offset in receive buffer
`count`	number of items in receive buffer
`datatype`	datatype of each item in receive buffer
`source`	rank of source
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a receive operation. Java binding of the MPI operation *MPI_RECV_INIT*. Further comments as for `Recv`.

```
void Prequest.Start()
```

Activate a persistent communication request. Java binding of the MPI operation *MPI_START*. The communication is completed by using the request in one of the operations `Request.Wait`, `Request.Test`, `Request.Waitany`, `Request.Testany`, `Request.Waitall`, `Request.Testall`, `Request.Waitsome`, or `Request.Testsome`. On successful completion the request becomes inactive again. It can be reactivated by a further call to `Start`.

```
static void Prequest.Startall(Prequest [] array_of_requests)
```

`array_of_requests`	array of persistent communication requests

Activate a list of communication requests. Java binding of the MPI operation *MPI_STARTALL*.

3.10 Send-receive

```
Status Comm.Sendrecv(Object sendbuf, int sendoffset,
                     int sendcount, Datatype sendtype,
                     int dest, int sendtag,
                     Object recvbuf, int recvoffset,
                     int recvcount, Datatype recvtype,
                     int source, int recvtag)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`dest`	rank of destination
`sendtag`	send tag
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of each item in receive buffer
`source`	rank of source
`recvtag`	receive tag
returns:	status object

Execute a blocking send and receive operation. Java binding of the MPI operation *MPI_SENDRECV*. Further comments as for `Send` and `Recv`.

```
Status Comm.Sendrecv_replace(Object buf, int offset,
                             int count, Datatype datatype,
                             int dest, int sendtag,
                             int source, int recvtag)
```

`buf`	buffer array
`offset`	initial offset in buffer
`count`	number of items to send
`datatype`	datatype of each item in buffer
`dest`	rank of destination
`sendtag`	send tag
`source`	rank of source
`recvtag`	receive tag
returns:	status object

Execute a blocking send and receive operation, receiving message into send buffer. Java binding of the MPI operation *MPI_SENDRECV_REPLACE*. Further comments as for `send` and `recv`.

3.11 Null processes

The constant *MPI_PROC_NULL* is available as `MPI.PROC_NULL`.

3.12 Derived datatypes

In C or Fortran bindings of MPI, derived datatypes have two roles. One is to allow messages to contain mixed types (for example they allow an integer count followed by a sequence of real numbers to be passed in a single message). The other is to allow noncontiguous data to be transmitted. In mpiJava the first role is abandoned. Any derived type can only include elements of a single basic type.

> *Rationale.* In the C binding of MPI, for example, the *MPI_TYPE_-STRUCT* constructor for derived types might be used to describe the physical layout of a *struct* containing mixed types. This will not work in Java, because Java does not expose the low-level layout of its objects. In C and Fortran another use of *MPI_TYPE_STRUCT* involves incorporating offsets computed as differences between absolute addresses, so that parts of a message can come from separately declared entities. It might be possible to contrive something analogous in a Java binding, somehow encoding object references instead of physical addresses. Such a contrivance is unlikely to be very natural—even in C and Fortran the mechanism is not particularly elegant. Meanwhile, the effect of either of these applications of *MPI_TYPE_STRUCT* can be achieved by using `MPI.OBJECT` as the buffer type, and relying on Java object serialization. *(End of rationale.)*

This leaves description of noncontiguous buffers as the essential role for derived data types in mpiJava.

Every derived data type constructable in mpiJava has a uniquely defined *base type*. This is one of the 9 basic types enumerated in section 3.2. Derived types inherit their base types from their precursors in a straightforward way.

In mpiJava a **general datatype** is an object that specifies two things

- A base type
- A sequence of integer displacements

In contrast to the C and Fortran bindings the displacements are in terms of subscripts in the buffer array argument, *not* byte displacements.

The base types for the predefined MPI datatypes are

MPI datatype	base type
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.OBJECT	`Object`
MPI.LB	$\perp$
MPI.UB	$\perp$
MPI.PACKED	`byte`

The symbol $\perp$ is a special undefined value. The displacement sequences for the predefined types (other than `MPI.LB`, `MPI.UB`) consist of a single zero.

If the displacement sequence of a datatype is

$$DispSeq = \{disp_0, \ldots, disp_{n-1}\}$$

we define

$$\begin{aligned} lb(DispSeq) &= \min_j disp_j, \\ ub(DispSeq) &= \max_j (disp_j + 1), \quad \text{and} \\ extent(DispSeq) &= ub(DispSeq) - lb(DispSeq) \end{aligned}$$

Rationale. This definition of the extent differs from the definition in the C or Fortran. It is in units of the buffer array index, *not* in units of bytes. *(End of rationale.)*

As discussed at the end of this section, these definitions have to be modified if the type construction involves `MPI.LB`, `MPI.UB`.

```
static Datatype Datatype.Contiguous(int count, Datatype oldtype)
```

`count`	replication count
`oldtype`	old datatype
returns:	new datatype

Construct new datatype representing replication of the old datatype into contiguous locations. Java binding of the MPI operation *MPI_TYPE_CONTIGUOUS*. The base type of the new datatype is the same as the base type of the old type. Assume the displacement sequence of the old type is

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent *ex*. Then the new datatype has a displacement sequence with `count` $\cdot$ `n` entries defined by:

$$\begin{aligned} \{ \quad & disp_0, \ldots, disp_{n-1}, \\ & disp_0 + ex, \ldots, disp_{n-1} + ex, \\ & \ldots, \\ & disp_0 + ex \cdot (\texttt{count} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{count} - 1) \quad \} \end{aligned}$$

```
static Datatype Datatype.Vector(int count,
                                int blocklength, int stride,
                                Datatype oldtype)
```

`count`	number of blocks
`blocklength`	number of elements in each block
`stride`	number of elements between start of each block
`oldtype`	old datatype
returns:	new datatype

Construct new datatype representing replication of the old datatype into locations that consist of equally spaced blocks. Java binding of the MPI operation *MPI_TYPE_VECTOR*. The base type of the new datatype is the same as the base type of the old type. Assume the displacement sequence of the old type is

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent *ex*. Let `bl` be `blocklength`. Then the new datatype has a displacement sequence with `count` $\cdot$ `bl` $\cdot$ `n` entries defined by:

$$\{ \quad disp_0, \ldots, disp_{n-1},$$

$$disp_0 + ex, \ldots, disp_{n-1} + ex,$$
$$\ldots,$$
$$disp_0 + ex \cdot (\texttt{bl} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{bl} - 1),$$

$$disp_0 + ex \cdot \texttt{stride}, \ldots, disp_{n-1} + ex \cdot \texttt{stride},$$
$$disp_0 + ex \cdot (\texttt{stride} + 1), \ldots, disp_{n-1} + ex \cdot (\texttt{stride} + 1),$$
$$\ldots,$$
$$disp_0 + ex \cdot (\texttt{stride} + \texttt{bl} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{stride} + \texttt{bl} - 1),$$

$$\ldots,$$

$$disp_0 + ex \cdot \texttt{stride} \cdot (\texttt{count} - 1), \ldots, disp_{n-1} + ex \cdot \texttt{stride} \cdot (\texttt{count} - 1),$$
$$disp_0 + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + 1), \ldots,$$
$$disp_{n-1} + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + 1),$$
$$\ldots,$$
$$disp_0 + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + \texttt{bl} - 1), \ldots,$$
$$disp_{n-1} + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + \texttt{bl} - 1) \quad \}$$

```
static Datatype Datatype.Hvector(int count,
                                 int blocklength, int stride,
                                 Datatype oldtype)
```

`count`	number of blocks
`blocklength`	number of elements in each block
`stride`	number of elements between start of each block
`oldtype`	old datatype
returns:	new datatype

Identical to `Vector` except that the stride is expressed directly in terms of the buffer index, rather than the units of the old type. Java binding of the MPI operation *MPI_TYPE_HVECTOR. Unlike other language bindings*, the value of `stride` is *not* measured in bytes. The displacement sequence of the new type is:

$$\{ \quad disp_0, \ldots, disp_{n-1},$$
$$disp_0 + ex, \ldots, disp_{n-1} + ex,$$
$$\ldots,$$
$$disp_0 + ex \cdot (\texttt{bl} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{bl} - 1),$$

$$disp_0 + ex \cdot \texttt{stride}, \ldots, disp_{n-1} + \texttt{stride},$$
$$disp_0 + \texttt{stride} + ex, \ldots, disp_{n-1} + \texttt{stride} + ex,$$
$$\ldots,$$
$$disp_0 + \texttt{stride} + ex \cdot (\texttt{bl} - 1), \ldots, disp_{n-1} + \texttt{stride} + ex \cdot (\texttt{bl} - 1),$$

$$\ldots,$$

$$disp_0 + \texttt{stride} \cdot (\texttt{count} - 1), \ldots, disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1),$$
$$disp_0 + \texttt{stride} \cdot (\texttt{count} - 1) + ex, \ldots,$$
$$disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1) + ex,$$
$$\ldots,$$
$$disp_0 + \texttt{stride} \cdot (\texttt{count} - 1) + ex \cdot (\texttt{bl} - 1), \ldots,$$
$$disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1) + ex \cdot (\texttt{bl} - 1) \quad \}$$

```
static Datatype Datatype.Indexed(int [] array_of_blocklengths,
                                 int [] array_of_displacements,
                                 Datatype oldtype)
```

`array_of_blocklengths`	number of elements per block
`array_of_displacements`	displacement of each block in units of old type
`oldtype`	old datatype
returns:	new datatype

Construct new datatype representing replication of the old type into a sequence of blocks where each block can contain a different number of copies and have a different displacement. Java binding of the MPI operation *MPI_TYPE_-INDEXED*. The number of blocks is taken to be size of the `arrayOfBlock-lengths` argument. The second argument, `array_of_displacements`, should be the same size. The base type of the new datatype is the same as the base type of the old type. Assume the displacement sequence of the old type is

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent ex. Let `B` be the `array_of_blocklengths` argument and `D` be the `array_of_displacements` argument. Then the new datatype has a displacement sequence with $n \cdot \sum_{i=0}^{count-1} \texttt{B}[i]$ entries:

$$\{ \quad disp_0 + \texttt{D}[0] \cdot ex, \ldots, disp_{n-1} + \texttt{D}[0] \cdot ex,$$
$$disp_0 + (\texttt{D}[0] + 1) \cdot ex, \ldots, disp_{n-1} + (\texttt{D}[0] + 1) \cdot ex,$$

$$\begin{aligned}
&\ldots, \\
&disp_0 + (\mathtt{D}[0] + \mathtt{B}[0] - 1) \cdot ex, \ldots, disp_{n-1} + (\mathtt{D}[0] + \mathtt{B}[0] - 1) \cdot ex, \\
&\ldots, \\
&disp_0 + \mathtt{D}[count - 1] \cdot ex, \ldots, disp_{n-1} + \mathtt{D}[count - 1] \cdot ex, \\
&disp_0 + (\mathtt{D}[count - 1] + 1) \cdot ex, \ldots, disp_{n-1} + (\mathtt{D}[count - 1] + 1) \cdot ex, \\
&\ldots, \\
&disp_0 + (\mathtt{D}[count - 1] + \mathtt{B}[count - 1] - 1) \cdot ex, \ldots, \\
&\qquad disp_{n-1} + (\mathtt{D}[count - 1] + \mathtt{B}[count - 1] - 1) \cdot ex \quad \}
\end{aligned}$$

Here, *count* is the number of blocks.

```
static Datatype Datatype.Hindexed(int [] array_of_blocklengths,
                                  int [] array_of_displacements,
                                  Datatype oldtype)
```

array_of_blocklengths	number of elements per block
array_of_displacements	displacement in buffer for each block
oldtype	old datatype
returns:	new datatype

Identical to **indexed** except that the displacements are expressed directly in terms of the buffer index, rather than the units of the old type. Java binding of the MPI operation *MPI_TYPE_HINDEXED. Unlike other language bindings*, the values in **array_of_displacements** are *not* measured in bytes. The displacement sequence of the new type is:

$$\begin{aligned}
\{ \quad &disp_0 + \mathtt{D}[0], \ldots, disp_{n-1} + \mathtt{D}[0], \\
&disp_0 + \mathtt{D}[0] + ex, \ldots, disp_{n-1} + \mathtt{D}[0] + ex, \\
&\ldots, \\
&disp_0 + \mathtt{D}[0] + (\mathtt{B}[0] - 1) \cdot ex, \ldots, disp_{n-1} + \mathtt{D}[0] + (\mathtt{B}[0] - 1) \cdot ex, \\
&\ldots, \\
&disp_0 + \mathtt{D}[count - 1], \ldots, disp_{n-1} + \mathtt{D}[count - 1], \\
&disp_0 + \mathtt{D}[count - 1] + ex, \ldots, disp_{n-1} + \mathtt{D}[count - 1] + ex, \\
&\ldots,
\end{aligned}$$

$$disp_0 + \texttt{D}[count-1] + (\texttt{B}[count-1]-1)\cdot ex, \ldots,$$
$$disp_{n-1} + \texttt{D}[count-1] + (\texttt{B}[count-1]-1)\cdot ex \quad \}$$

```
static Datatype Datatype.Struct(int [] array_of_blocklengths,
                                int [] array_of_displacements,
                                Datatype [] array_of_types)
```

`array_of_blocklengths`	number of elements per block
`array_of_displacements`	displacement in buffer for each block
`array_of_types`	type of elements in each block
returns:	new datatype

The most general type constructor. Java binding of the MPI operation *MPI_TYPE_STRUCT*. The number of blocks is taken to be size of the `array_of_blocklengths` argument. The second and third arguments, `array_of_displacements` and `array_of_types`, should be the same size. *Unlike other language bindings*, the values in `array_of_displacements` are *not* measured in bytes. All elements of `array_of_types` with definite base types *must have the* same *base type*: this will be the base type of new datatype. Let `T` be the `array_of_types` argument. Assume the displacement sequence of the old type $\texttt{T}[i]$ is

$$\{disp_0^i, \ldots, disp_{n_i-1}^i\}$$

with extent ex_i. Let `B` be the `array_of_blocklengths` argument and `D` be the `array_of_displacements` argument. Then the new datatype has a displacement sequence with $\sum_{i=0}^{c-1} \texttt{B}[i]\cdot n_i$ entries:

$$\begin{aligned}
\{\quad & disp_0^0 + \texttt{D}[0], \ldots, disp_{n_0-1}^0 + \texttt{D}[0], \\
& disp_0^0 + \texttt{D}[0] + ex_0, \ldots, disp_{n_0-1}^0 + \texttt{D}[0] + ex_0, \\
& \ldots, \\
& disp_0^0 + \texttt{D}[0] + (\texttt{B}[0]-1)\cdot ex_0, \ldots, disp_{n_0-1}^0 + \texttt{D}[0] + (\texttt{B}[0]-1)\cdot ex_0, \\
& \\
& \ldots, \\
& \\
& disp_0^{c-1} + \texttt{D}[c-1], \ldots, disp_{n_{c-1}-1}^{c-1} + \texttt{D}[c-1], \\
& disp_0^{c-1} + \texttt{D}[c-1] + ex_{c-1}, \ldots, disp_{n_{c-1}-1}^{c-1} + \texttt{D}[c-1] + ex_{c-1}, \\
& \ldots, \\
& disp_0^{c-1} + \texttt{D}[c-1] + (\texttt{B}[c-1]-1)\cdot ex_{c-1}, \ldots, \\
& \qquad disp_{n_{c-1}-1}^{c-1} + \texttt{D}[c-1] + (\texttt{B}[c-1]-1)\cdot ex_{c-1} \quad \}
\end{aligned}$$

Here, c is the number of blocks.

If any elements of `array_of_types` are `MPI.LB` or `MPI.UB`, the corresponding displacements are omitted in the displacement sequence. These displacements only affect the computation of `Datatype.Lb`, `Datatype.Ub` and `Datatype.Extent`, as explained below.

Revised definition of general datatype. In the presence of `MPI.LB`, `MPI.UB` component types, an mpiJava general datatype can be represented by four things:

- A base type
- A sequence, *DispSeq*, of proper displacements.
- A set, *LBDisps*, of pseudo-displacements for `MPI.LB` markers.
- A set, *UBDisps*, of pseudo-displacements for `MPI.UB` markers.

For basic datatypes other than `MPI.LB`, `MPI.UB` the displacements take the form

$$\begin{aligned} \mathit{DispSeq} &= \{0\} \\ \mathit{LBDisps} &= \emptyset \\ \mathit{UBDisps} &= \emptyset \end{aligned}$$

For `MPI.LB` they are

$$\begin{aligned} \mathit{DispSeq} &= \emptyset \\ \mathit{LBDisps} &= \{0\} \\ \mathit{UBDisps} &= \emptyset \end{aligned}$$

For `MPI.UB` they are

$$\begin{aligned} \mathit{DispSeq} &= \emptyset \\ \mathit{LBDisps} &= \emptyset \\ \mathit{UBDisps} &= \{0\} \end{aligned}$$

The two sets of pseudo-displacements are propagated to derived types by formulae identical to the ones given above for proper displacements. Below we will use the definition

$$\mathit{AllDisps} = \mathit{DispSeq} \cup \mathit{LBDisps} \cup \mathit{UBDisps}$$

`int Datatype.Extent()`

returns: datatype extent

Returns the extent of a datatype. Java binding of the MPI operation *MPI_TYPE_EXTENT*. Return value is equal to

$$\texttt{Ub()} - \texttt{Lb()}$$

`int Datatype.Lb()`

returns: displacement of lower bound from origin

Find the lower bound of a datatype. Java binding of the MPI operation *MPI_TYPE_LB*. If *LBDisps* is non-empty the return value of `Lb` is the least element of that set. Otherwise it is the least element of *AllDisps*[1].

`int Datatype.Ub()`

returns: displacement of upper bound from origin

Find the upper bound of a datatype. Java binding of the MPI operation *MPI_TYPE_UB*. If *UBDisps* is non-empty the return value of `Ub` is the greatest element of that set. Otherwise it is

$$\max_{disp \in AllDisps} disp + 1$$

`int Datatype.Size()`

returns: datatype size

Returns the total size of the type. Java binding of the MPI operation *MPI_TYPE_SIZE*. Size is defined as the total number of buffer elements incorporated by the data type, or equivalently as the length of the displacement sequence. *Unlike other language bindings*, the size is *not* measured in bytes.

`void Datatype.Commit()`

Commit a derived datatype. Java binding of the MPI operation *MPI_TYPE_COMMIT*.

`void Datatype.finalize()`

[1] If *AllDisps* is empty (which could happen for a derived datatype created with replication count of zero, for instance) the results of `Lb`, `Ub` and thus `Extent` are undefined.

Destructor. Java binding of the MPI operation *MPI_TYPE_FREE*.

```
int Status.Get_elements(Datatype datatype)
```

`datatype`	datatype used by receive operation
returns:	number of received basic elements

Retrieve number of basic elements from status. Java binding of the MPI operation *MPI_GET_ELEMENTS*.

3.13 Pack and unpack

```
int Comm.Pack(Object inbuf, int offset, int incount,
              Datatype datatype,
              byte [] outbuf, int position)
```

`inbuf`	input buffer array
`offset`	initial offset in input buffer
`incount`	number of items in input buffer
`datatype`	datatype of each item in input buffer
`outbuf`	output buffer
`position`	initial position in ouput buffer
returns:	final position in output buffer

Packs message in send buffer `inbuf` into space specified in `outbuf`. Java binding of the MPI operation *MPI_PACK*. The return value is the output value of `position`—the inital value incremented by the number of bytes written.

```
int Comm.Unpack(byte [] inbuf, int position,
                Object outbuf, int offset, int outcount,
                Datatype datatype)
```

`inbuf`	input buffer
`position`	initial position in input buffer
`outbuf`	output buffer array
`offset`	initial offset in output buffer
`outcount`	number of items in output buffer
`datatype`	datatype of each item in output buffer
returns:	final position in input buffer

Unpacks message in receive buffer `outbuf` into space specified in `inbuf`. Java binding of the MPI operation *MPI_UNPACK*. The return value is the output value of *position*—the inital value incremented by the number of bytes read.

```
int Comm.Pack_size(int incount, Datatype datatype)
```

`incount`	number of items in input buffer
`datatype`	datatype of each item in input buffer
returns:	upper bound on size of packed message

Returns an upper bound on the increment of `position` effected by `pack`. Java binding of the MPI operation *MPI_PACK_SIZE. It is an error to call this function if the base type of* `datatype` *is* `MPI.OBJECT`.

4 Collective Communication

4.1 Introduction and Overview

In general the mpiJava bindings of collective communication operations realize the MPI functions as members of the `IntraComm` class.

4.2 Communicator argument

No special issues for Java binding.

4.3 Barrier synchronization

```
void Intracomm.Barrier()
```

A call to `Barrier` blocks the caller until all processes in the group have called it. Java binding of the MPI operation *MPI_BARRIER*.

4.4 Broadcast

```
void Intracomm.Bcast(Object buffer, int offset, int count,
                     Datatype datatype, int root)
```

`buf`	buffer array
`offset`	initial offset in buffer
`count`	number of items in buffer
`datatype`	datatype of each item in buffer
`dest`	rank of broadcast root

Broadcast a message from the process with rank `root` to all processes of the group. Java binding of the MPI operation *MPI_BCAST*.

4.5 Gather

```
void Intracomm.Gather(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int recvcount, Datatype recvtype, int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of each item in receive buffer
`root`	rank of receiving process

Each process sends the contents of its send buffer to the root process. Java binding of the MPI operation *MPI_GATHER*.

```
void Intracomm.Gatherv(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       Object recvbuf, int recvoffset,
                       int [] recvcounts, int [] displs,
                       Datatype recvtype, int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	number of elements received from each process
`displs`	displacements at which to place incoming data
`recvtype`	datatype of each item in receive buffer
`root`	rank of receiving process

Extends functionality of `Gather` by allowing varying counts of data from each process. Java binding of the MPI operation *MPI_GATHERV*. The sizes of arrays `recvcounts` and `displs` should be the size of the group. Entry i of `displs` specifies the displacement relative to element `recvoffset` of `recvbuf` at which to place incoming data. Note that if `recvtype` is a derived data type, elements of `displs` are in units of the derived type extent, (unlike `recvoffset`, which is a direct index into the buffer array).

4.6 Scatter

```
void Intracomm.Scatter(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       Object recvbuf, int recvoffset,
```

```
                         int recvcount, Datatype recvtype,
                         int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items sent to each process
`sendtype`	datatype of send buffer items
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of receive buffer items
`root`	rank of sending process

Inverse of the operation `Gather`. Java binding of the MPI operation *MPI_SCATTER*.

```
void Intracomm.Scatterv(Object sendbuf, int sendoffset,
                        int [] sendcounts, int [] displs,
                        Datatype sendtype,
                        Object recvbuf, int recvoffset,
                        int recvcount, Datatype recvtype,
                        int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcounts`	number of items sent to each process
`displs`	displacements from which to take outgoing data
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of elements in receive buffer
`recvtype`	datatype of receive buffer items
`root`	rank of sending process

Inverse of the operation `Gatherv`. Java binding of the MPI operation *MPI_SCATTERV*.

4.7 Gather-to-all

```
void Intracomm.Allgather(Object sendbuf, int sendoffset,
                         int sendcount, Datatype sendtype,
                         Object recvbuf, int recvoffset,
                         int recvcount, Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items sent to each process
`sendtype`	datatype of send buffer items
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of receive buffer items

Similar to `Gather`, but all processes receive the result. Java binding of the MPI operation *MPI_ALLGATHER*.

```
void Intracomm.Allgatherv(Object sendbuf, int sendoffset,
                          int sendcount, Datatype sendtype,
                          Object recvbuf, int recvoffset,
                          int [] recvcounts, int [] displs,
                          Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	number of elements received from each process
`displs`	displacements at which to place incoming data
`recvtype`	datatype of each item in receive buffer

Similar to `Gatherv`, but all processes receive the result. Java binding of the MPI operation *MPI_GATHERV*.

4.8 All-to-All Scatter/Gather

```
void Intracomm.Alltoall(Object sendbuf, int sendoffset,
                        int sendcount, Datatype sendtype,
                        Object recvbuf, int recvoffset,
                        recvcount, Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items sent to each process
`sendtype`	datatype of send buffer items
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items received from any process
`recvtype`	datatype of receive buffer items

Extension of `Allgather` to the case where each process sends distinct data to each of the receivers. Java binding of the MPI operation *MPI_ALLTOALL*.

```
void Intracomm.Alltoallv(Object sendbuf, int sendoffset,
                         int [] sendcount, int [] sdispls,
                         Datatype sendtype,
                         Object recvbuf, int recvoffset,
                         int [] recvcount, int [] rdispls,
                         Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcounts`	number of items sent to each process
`sdispls`	displacements from which to take outgoing data
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	number of elements received from each process
`rdispls`	displacements at which to place incoming data
`recvtype`	datatype of each item in receive buffer

Adds flexibility to `Alltoall`: location of data for send is specified by `sdispls` and location to place data on receive side is specified by `rdispls`. Java binding of the MPI operation *MPI_ALLTOALLV*.

4.9 Global Reduction Operations

```
void Intracomm.Reduce(Object sendbuf, int sendoffset,
                      Object recvbuf, int recvoffset,
                      int count, Datatype datatype,
                      Op op, int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`count`	number of items in send buffer
`datatype`	data type of each item in send buffer
`op`	reduce operation
`dest`	rank of root process

Combine elements in input buffer of each process using the reduce operation, and return the combined value in the output buffer of the root process. Java binding of the MPI operation *MPI_REDUCE*.

The predefined operations are available in Java as `MPI.MAX`, `MPI.MIN`, `MPI.SUM`, `MPI.PROD`, `MPI.LAND`, `MPI.BAND`, `MPI.LOR`, `MPI.BOR`, `MPI.LXOR`, `MPI.BXOR`, `MPI.MINLOC` and `MPI.MAXLOC`.

The handling of *MINLOC* and *MAXLOC* is modelled on the Fortran binding. The extra predefined types `MPI.SHORT2`, `MPI.INT2`, `MPI.LONG2`, `MPI.FLOAT2`, `MPI.DOUBLE2` describe pairs of Java numeric primitive types.

`Op.Op(User_function function, boolean commute)`

`function`	user defined function
`commute`	`true` if commutative, `false` otherwise

Bind a user-defined global reduction operation to an `Op` object. Java binding of the MPI operation *MPI_OP_CREATE*. The abstract base class `User_function` is defined by

```
class User_function {
  public abstract void Call(Object invec, int inoffset,
                            Object inoutvec, int inoutoffset,
                            int count, Datatype datatype) ;
}
```

To define a new operation, the programmer should define a concrete subclass of `User_function`, implementing the `Call` method, then pass an object from this class to the `Op` constructor. The `User_function.Call` method plays exactly the same role as the `function` argument in the standard bindings of MPI. The actual arguments `invec` and `inoutvec` passed to `call` will be arrays containing `count` elements of the type specified in the `datatype` argument. Offsets in the arrays can be specified as for message buffers. The user-defined `Call` method should combine the arrays element by element, with results appearing in `inoutvec`.

`void Op.finalize()`

Destructor. Java binding of the MPI operation *MPI_OP_FREE*.

```
void Intracomm.Allreduce(Object sendbuf, int sendoffset,
                         Object recvbuf, int recvoffset,
                         int count, Datatype datatype,
                         Op op)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`count`	number of items in send buffer
`datatype`	data type of each item in send buffer
`op`	reduce operation

Same as `Reduce` except that the result appears in receive buffer of all processes in the group. Java binding of the MPI operation *MPI_ALLREDUCE*.

4.10 Reduce-Scatter

```
void Intracomm.Reduce_scatter(Object sendbuf, int sendoffset,
                              Object recvbuf, int recvoffset,
                              int [] recvcounts,
                              Datatype datatype, Op op)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	numbers of result elements distributed to each process
`datatype`	data type of each item in send buffer
`op`	reduce operation

Combine elements in input buffer of each process using the reduce operation, and scatter the combined values over the output buffers of the processes. Java binding of the MPI operation *MPI_REDUCE_SCATTER*.

4.11 Scan

```
void Intracomm.Scan(Object sendbuf, int sendoffset,
                    Object recvbuf, int recvoffset,
                    int count, Datatype datatype,
                    Op op)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`count`	number of items in input buffer
`datatype`	data type of each item in input buffer
`op`	reduce operation

Perform a prefix reduction on data distributed across the group. Java binding of the MPI operation *MPI_SCAN*.

4.12 Correctness

No special issues for Java binding.

5 Groups, Contexts and Communicators

5.1 Introduction

No special issues for Java binding.

5.2 Basic Concepts

The constant *MPI_GROUP_EMPTY* is available as `MPI.GROUP_EMPTY`. The constants *MPI_COMM_WORLD*, *MPI_COMM_SELF* are available as `MPI.COMM_WORLD`, `MPI.COMM_SELF`.

5.3 Group Management

```
int Group.Size()
```

returns: number of processors in the group

Size of group. Java binding of the MPI operation *MPI_GROUP_SIZE*.

```
int Group.Rank()
```

returns: rank of the calling process in the group

Rank of this process in group. Java binding of the MPI operation *MPI_GROUP_RANK*. Result value is `MPI.UNDEFINED` if this process is not a member of the group.

```
static int [] Group.Translate_ranks(Group group1, int [] ranks1,
                                    Group group2)
```

`group1`	first group
`ranks1`	array of valid ranks in `group1`
`group2`	second group

returns: array of corresponding ranks in `group2`

Translate ranks within first group to ranks within second group. Java binding of the MPI operation *MPI_GROUP_TRANSLATE_RANKS*. Result elements are `MPI.UNDEFINED` where no correspondence exists.

```
static int Group.Compare(Group group1, Group group2)
```

`group1`	first group
`group2`	second group

returns: result

Compare two groups. Java binding of the MPI operation *MPI_GROUP_COMPARE*. `MPI.IDENT` results if the group members and group order are exactly the same in both groups. `MPI.SIMILAR` results if the group members are the same but the order is different. `MPI.UNEQUAL` results otherwise.

`Group Comm.Group()`

returns: group corresponding to this communicator

Return group associated with a communicator. Java binding of the MPI operation *MPI_COMM_GROUP*.

`static Group Group.Union(Group group1, Group group2)`

`group1`	first group
`group2`	second group

returns: union group

Set union of two groups. Java binding of the MPI operation *MPI_GROUP_UNION*.

`static Group Group.Intersection(Group group1, Group group2)`

`group1`	first group
`group2`	second group

returns: intersection group

Set intersection of two groups. Java binding of the MPI operation *MPI_GROUP_INTERSECTION*.

`static Group Group.Difference(Group group1, Group group2)`

`group1`	first group
`group2`	second group

returns: difference group

Result contains all elements of the first group that are not in the second group. Java binding of the MPI operation *MPI_GROUP_DIFFERENCE*.

`Group Group.Incl(int [] ranks)`

ranks ranks from this group to appear in new group

returns: new group

Create a subset group including specified processes. Java binding of the MPI operation *MPI_GROUP_INCL*.

`Group Group.Excl(int [] ranks)`

ranks ranks from this group *not* to appear in new group

returns: new group

Create a subset group excluding specified processes. Java binding of the MPI operation *MPI_GROUP_EXCL*.

`Group Group.Range_incl(int [] [] ranges)`

ranges array of integer triplets

returns: new group

Create a subset group including processes specified by strided intervals of ranks. Java binding of the MPI operation *MPI_GROUP_RANGE_INCL*. The triplets are of the form (first rank, last rank, stride) indicating ranks in this group to be included in the new group. The size of the first dimension of `ranges` is the number of triplets. The size of the second dimension is 3.

`Group Group.Range_excl(int [] [] ranges)`

ranges array of integer triplets

returns: new group

Create a subset group excluding processes specified by strided intervals of ranks. Java binding of the MPI operation *MPI_GROUP_RANGE_EXCL*. Triplet array is defined as for `Range_incl`, the ranges indicating ranks in this group to be excluded from the new group.

`void Group.finalize()`

Destructor. Java binding of the MPI operation *MPI_GROUP_FREE*.

5.4 Communicator Management

`int Comm.Size()`

returns:	number of processors in the group of this communicator

Size of group of this communicator. Java binding of the MPI operation *MPI_COMM_SIZE*.

`int Comm.Rank()`

returns:	rank of the calling process in the group of this communicator

Rank of this process in group of this communicator. Java binding of the MPI operation *MPI_COMM_RANK*.

`static int Comm.Compare(Comm comm1, Comm comm2)`

`comm1`	first communicator
`comm2`	second communicator

returns:	result

Compare two communicators. Java binding of the MPI operation *MPI_COMM_COMPARE*. `MPI.IDENT` results if the `comm1` and `comm2` are references to the same object (ie, if `comm1 == comm2`). `MPI.CONGRUENT` results if the underlying groups are identical but the communicators differ by context. `MPI.SIMILAR` results if the underlying groups are similar but the communicators differ by context. `MPI.UNEQUAL` results otherwise.

`Object Comm.clone()`

returns:	copy of this communicator

Duplicate this communicator. Java binding of the MPI operation *MPI_COMM_DUP*. The new communicator is "congruent" to the old one, but has a different context.

Rationale. The decision to use the standard Java `clone` method means the static result type must be `Object`. The dynamic type will be that of the `Comm` subclass of the parent. MPI-defined and user-defined subclasses of `Comm` will generally override `clone` to ensure all relevant attributes are copied. *(End of rationale.)*

`Intracomm Intracomm.Create(Group group)`

`group`	group which is a subset of the group of this communicator
returns:	new communicator

Create a new communicator. Java binding of the MPI operation *MPI_COMM_-CREATE.*

`Intracomm Intracomm.Split(int color, int key)`

`color`	control of subset assignment
`key`	control of rank assignment
returns:	new communicator

Partition the group associated with this communicator and create a new communicator within each subgroup. Java binding of the MPI operation *MPI_COMM_-SPLIT.*

`void Comm.Free()`

Destroy this communicator. Java binding of the MPI operation *MPI_COMM_-FREE.*

Rationale. An explicitly called `Free` method is required rather than an implicitly called `finalize` method, because *MPI_COMM_FREE* is a collective operation. We cannot assume that the Java garbage collector will call a `finalize` method synchronously on all processors. *(End of rationale.)*

`boolean Comm.Is_null()`

returns:	true if the communicator object has been freed, false otherwise

Replaces comparision with `MPI_COMM_NULL`.

5.5 Motivating Examples

No special issues for Java binding.

5.6 Inter-Communication

```
boolean Comm.Test_inter()
```

returns: `true` if this is an inter-communicator, `false` otherwise

Test if this communicator is an inter-communicator. Java binding of the MPI operation *MPI_COMM_TEST_INTER*.

```
int Intercomm.Remote_size()
```

returns: number of process in remote group of this communicator

Size of remote group. Java binding of the MPI operation *MPI_COMM_REMOTE_SIZE*.

```
Group Intercomm.Remote_Group()
```

returns: remote group of this communicator

Return the remote group. Java binding of the MPI operation *MPI_COMM_REMOTE_GROUP*.

```
Intercomm Comm.Create_intercomm(Comm local_comm, int local_leader,
                                int remote_leader, int tag)
```

`local_comm`	local intra-communicator
`local_leader`	rank of local group leader in `localComm`
`remote_leader`	rank of remote group leader in this communictor
`tag`	"safe" tag

returns: new inter-communicator

Create an inter-communicator. Java binding of the MPI operation *MPI_INTERCOMM_CREATE*.

> *Rationale.* This operation is defined as a method on the "peer communicator", making it analogous to a `Send` or `Recv` communication with the remote group leader. *(End of rationale.)*

```
Intracomm Intercomm.Merge(boolean high)
```

`high` true if the local group has higher ranks in combined group

returns: new intra-communicator

Create an intra-communicator from the union of the two groups in the inter-communicator. Java binding of the MPI operation *MPI_INTERCOMM_MERGE*.

5.7 Caching

It is assumed that to achieve the effect of caching attributes in user-customized communicators programmers will create subclasses of the library-defined communicator classes with suitable additional fields. These fields may be copied or deleted by suitably overridden `clone` and `finalize` methods.

Hence the only "caching" operation surviving here is the binding of *MPI_ATTR_GET*, which is needed to access values of attributes predefined by the implementation. According the standard, the key values for such attributes include `MPI.TAG_UB`, `MPI.HOST`, `MPI.IO` and `MPI.WTIME_IS_GLOBAL`.

```
int Comm.Attr_get(int keyval)
```

`keyval` one of the key values predefined by the implementation

returns: attribute value

Retrieves attribute value by key. Java binding of the MPI operation *MPI_ATTR_GET*.

6 Process Topologies

6.1 Introduction

Communicators with Cartesian or graph topologies will be realized as instances of the subclasses `Cartcomm` or `Graphcomm`, respectively of `Intracomm`.

6.2 Virtual Topologies

No special issues for Java binding.

6.3 Embedding in MPI

No special issues for Java binding.

6.4 Overview of the Functions

No special issues for Java binding.

6.5 Topology Constructors

```
Cartcomm Intracomm.Create_cart(int [] dims, boolean [] periods,
                               boolean reorder)
```

`dims`	the number of processes in each dimension
`periods`	`true` if grid is periodic, `false` if not, in each dimension
`reorder`	`true` if ranking may be reordered, `false` if not
returns:	new Cartesian topology communicator

Create a Cartesian topology communicator whose group is a subset of the group of this communicator. Java binding of the MPI operation *MPI_CART_CREATE*. The number of dimensions of the Cartesian grid is taken to be the size of the `dims` argument. The array `periods` must be the same size.

```
static Cartcomm.Dims_create(int nnodes, int [] dims)
```

`nnodes`	number of nodes in a grid
`dims`	array specifying the number of nodes in each dimension

Select a balanced distribution of processes per coordinate direction. Java binding of the MPI operation *MPI_DIMS_CREATE*. Number of dimensions is the size of is `dims`. Note that `dims` is an *inout* parameter.

```
Graphcomm Intracomm.Create_graph(int [] index, int [] edges,
                                 boolean reorder)
```

`index`	node degrees
`edges`	graph edges
`reorder`	`true` if ranking may be reordered, `false` if not
returns:	new graph topology communicator

Create a graph topology communicator whose group is a subset of the group of this communicator. Java binding of the MPI operation *MPI_GRAPH_CREATE*. The number of nodes in the graph, *nnodes*, is taken to be size of the `index` argument. The size of array `edges` must be `index`$[nnodes - 1]$.

```
int Comm.Topo_test()
```

returns: topology type of communicator

Returns the type of topology associated with the communicator. Java binding of the MPI operation *MPI_TOPO_TEST*. The return value will be one of `MPI.GRAPH`, `MPI.CART` or `MPI.UNDEFINED`.

```
GraphParms Graphcomm.Get()
```

returns: object defining node degress and edges of graph

Returns graph topology information. Java binding of the MPI operations *MPI_GRAPHDIMS_GET* and *MPI_GRAPH_GET*. The class of the returned object is

```
public class GraphParms {
  public int [] index ;
  public int [] edges ;
}
```

The number of nodes and number of edges can be extracted from the sizes of the `index` and `edges` arrays.

```
CartParms Cartcomm.Get()
```

returns: object containing dimensions, periods and local coordinates

Returns Cartesian topology information. Java binding of the MPI operations *MPI_CARTDIM_GET* and *MPI_CART_GET*. The class of the returned object is

```
public class CartParms {
  public int [] dims ;
  public booleans [] periods ;
  public int [] coords ;
}
```

The number of dimensions can be obtained from the size of (eg) the `dims` array.

> *Rationale.* The inquiries *MPI_GRAPHDIMS_GET*, *MPI_GRAPH_GET*, *MPI_CARTDIM_GET*, and *MPI_CART_GET* are unusual in returning multiple independent values from single calls. This is a problem in Java. The Java binding could split these inquiries into several independent ones, but that would complicate JNI-based wrapper implementations. Hence we introduced the auxilliary classes `GraphParms` and `CartParms` to hold multiple results. *(End of rationale.)*

`int Cartcomm.Rank(int [] coords)`

`coords`	Cartesian coordinates of a process
returns:	rank of the specified process

Translate logical process coordinates to process rank. Java binding of the MPI operation *MPI_CART_RANK*.

`int [] Cartcomm.Coords(int rank)`

`coords`	rank of a process
returns:	Cartesian coordinates of the specified process

Translate process rank to logical process coordinates. Java binding of the MPI operation *MPI_CART_COORDS*.

`int [] Graphcomm.Neighbours(int rank)`

`coords`	rank of a process in the group of this communicator
returns:	array of ranks of neighbouring processes to one specified

Provides adjacency information for general graph topology. Java binding of the MPI operations *MPI_GRAPH_NEIGHBOURS_COUNT* and *MPI_GRAPH_NEIGHBOURS*. The number of neighbours can be extracted from the size of the result.

`ShiftParms Cartcomm.Shift(int direction, int disp)`

`direction`	coordinate dimension of shift
`disp`	displacement
returns:	object containing ranks of source and destination processes

Compute source and destination ranks for "shift" communication. Java binding of the MPI operation *MPI_CART_SHIFT*. The class of the returned object is

```
public class ShiftParms {
  public int rankSource ;
  public int rankDest ;
}
```

`Cartcomm Cartcomm.Sub(boolean [] remainDims)`

`remainDims`	by dimension, `true` if dimension is to be kept, `false` otherwise
returns:	communicator containing subgrid including this process

Partition Cartesian communicator into subgroups of lower dimension. Java binding of the MPI operation *MPI_CART_SUB*.

`int Cartcomm.Map(int [] dims, boolean [] periods)`

`dims`	the number of processes in each dimension
`periods`	`true` if grid is periodic, `false` if not, in each dimension
returns:	reordered rank of calling process

Compute an optimal placement. Java binding of the MPI operation *MPI_CART_MAP*. The number of dimensions is taken to be size of the `dims` argument.

`int Graphcomm.Map(int [] index, int [] edges)`

`index`	node degrees
`edges`	graph edges
returns:	reordered rank of calling process

Compute an optimal placement. Java binding of the MPI operation *MPI_GRAPH_MAP*. The number of nodes is taken to be size of the `index` argument.

7 MPI Environmental Management

7.1 Implementation information

The constants *MPI_TAG_UB*, *MPI_HOST* and *MPI_IO* are available as `MPI.-TAG_UB`, `MPI.HOST`, `MPI.IO`.

```
static String MPI.Get_processor_name()
```

returns: A unique specifier for the actual node.

Returns the name of the processor on which it is called. Java binding of the MPI operation *MPI_GET_PROCESSOR_NAME*.

7.2 Error handling

The constants *MPI_ERRORS_ARE_FATAL*, *MPI_ERRORS_RETURN* are available as `MPI.ERRORS_ARE_FATAL`, `MPI.ERRORS_RETURN`.

If the effective error handler is *MPI_ERRORS_RETURN*, the wrapper codes will throw appropriate Java exceptions (see section 7.3).

Currently mpiJava omits an interface for creating new MPI error handlers (the detailed interface of the handler function depends on unstandardized features of the MPI implementation).

```
static void Comm.Errorhandler_set(Errhandler errhandler)
```

`errhandler` new MPI error handler for communicator

Associates a new error handler with communicator at the calling process. Java binding of the MPI operation *MPI_ERRORHANDLER_SET*.

```
static Errhandler Comm.Errorhandler_get()
```

returns: MPI error handler currently associated with communicator

Returns the error handler currently associated with the communicator. Java binding of the MPI operation *MPI_ERRORHANDLER_GET*.

7.3 Error codes and classes

The `MPIException` subclasses

```
MPIErrBuffer
MPIErrCount
MPIErrType
MPIErrTag
MPIErrComm
MPIErrRank
MPIErrRequest
MPIErrRoot
MPIErrGroup
MPIErrOp
MPIErrTopology
MPIErrDims
MPIErrArg
MPIErrUnknown
MPIErrTruncate
MPIErrOther
MPIErrIntern
```

correspond to the standard MPI error classes. *[Not implemented in the current release.]*

7.4 Timers

`static double MPI.Wtime()`

returns: elapsed wallclock time in seconds since some time in the past

Returns wallclock time. Java binding of the MPI operation *MPI_WTIME*.

`static double MPI.Wtick()`

returns: resolution of `wtime` in seconds.

Returns resolution of timer. Java binding of the MPI operation *MPI_WTICK*.

7.5 Startup

`static String [] MPI.Init(String[] argv)`

`argv` arguments to `main` method.

returns: command line arguments returned by MPI.

Initialize MPI. Java binding of the MPI operation *MPI_INIT*.

`static void MPI.Finalize()`

Finalize MPI. Java binding of the MPI operation *MPI_FINALIZE*.

`static boolean MPI.Initialized()`

returns: `true` if `init` has been called, `false` otherwise.

Test if MPI has been initialized. Java binding of the MPI operation *MPI_INITIALIZED*.

`void Comm.Abort(int errorcode)`

`errorcode` error code for Unix or POSIX environments

Abort MPI. Java binding of the MPI operation *MPI_ABORT*.

8 Full public interface of classes

Section names appearing in comments refer to the preceding appendix. Specification of the methods immediately following those comments should be found in the referenced section.

8.1 MPI

```
public class MPI {
  public static Intracomm COMM_WORLD;

  public static Datatype BYTE, CHAR, SHORT, BOOLEAN, INT, LONG,
                         FLOAT, DOUBLE, OBJECT, PACKED, LB, UB ;

  public static int ANY_SOURCE, ANY_TAG ;

  public static int PROC_NULL ;

  public static int BSEND_OVERHEAD ;

  public static int UNDEFINED ;

  public static Op MAX, MIN, SUM, PROD, LAND, BAND,
                   LOR, BOR, LXOR, BXOR, MINLOC, MAXLOC ;

  public static Datatype SHORT2, INT2, LONG2, FLOAT2, DOUBLE2 ;

  public static Group GROUP_EMPTY ;

  public static Comm COMM_SELF ;

  public static int IDENT, CONGRUENT, SIMILAR, UNEQUAL ;

  public static int GRAPH, CART ;

  public static ErrHandler ERRORS_ARE_FATAL, ERRORS_RETURN ;

  public static int TAG_UB, HOST, IO ;


  // Buffer allocation and usage

  public static void Buffer_attach(byte [] buffer)
                                          throws MPIException {...}

  public static byte [] Buffer_detach() throws MPIException {...}


  // Environmental Management

  public static String [] Init(String[] argv) throws MPIException {...}

  public static void Finalize() throws MPIException {...}
```

```
  public static String Get_processor_name() throws MPIException {...}

  public static double Wtime() {...}

  public static double Wtick() {...}

  public static boolean Initialized() throws MPIException {...}

  ...
}

public class Errhandler {
  ...
}
```

8.2 Comm

```
public class Comm {

  // Communicator Management

  public int Size() throws MPIException {...}

  public int Rank() throws MPIException {...}

  public Group Group() throws MPIException {...}
                                      // (see ``Group management'')

  public static int Compare(Comm comm1, Comm comm2)
                                          throws MPIException {...}

  public Object clone() {...}

  public void Free() throws MPIException {...}

  public boolean Is_null() {...}


  // Inter-communication

  public boolean Test_inter() throws MPIException {...}

  public Intercomm Create_intercomm(Comm local_comm, int local_leader,
                                  int remote_leader, int tag)
                                           throws MPIException {...}

  // Caching

  public Object Attr_get(int keyval) throws MPIException {...}


  // Blocking Send and Receive operations

  public void Send(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag)
                                           throws MPIException {...}

  public Status Recv(Object buf, int offset, int count,
                    Datatype datatype, int source, int tag)
                                           throws MPIException {...}


  // Communication Modes
```

```
public void Bsend(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public void Ssend(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public void Rsend(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag)
                                          throws MPIException {...}


// Nonblocking communication

public Request Isend(Object buf, int offset, int count,
                     Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Request Ibsend(Object buf, int offset, int count,
                      Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Request Issend(Object buf, int offset, int count,
                      Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Request Irsend(Object buf, int offset, int count,
                      Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Request Irecv(Object buf, int offset, int count,
                     Datatype datatype, int source, int tag)
                                          throws MPIException {...}


// Probe and cancel

public Status Iprobe(int source, int tag) throws MPIException {...}

public Status Probe(int source, int tag) throws MPIException {...}


// Persistent communication requests

public Prequest Send_init(Object buf, int offset, int count,
```

```
                          Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Prequest Bsend_init(Object buf, int offset, int count,
                           Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Prequest Ssend_init(Object buf, int offset, int count,
                           Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Prequest Rsend_init(Object buf, int offset, int count,
                           Datatype datatype, int dest, int tag)
                                          throws MPIException {...}

public Prequest Recv_init(Object buf, int offset, int count,
                          Datatype datatype, int source, int tag)
                                          throws MPIException {...}


// Send-receive

public Status Sendrecv(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       int dest, int sendtag,
                       Object recvbuf, int recvoffset,
                       int recvcount, Datatype recvtype,
                       int source, int recvtag)
                                          throws MPIException {...}

public Status Sendrecv_replace(Object buf, int offset,
                               int count, Datatype datatype,
                               int dest, int sendtag,
                               int source, int recvtag)
                                          throws MPIException {...}

// Pack and unpack

public int Pack(Object inbuf, int offset, int incount,
                Datatype datatype,
                byte [] outbuf, int position)
                                          throws MPIException {...}

public int Unpack(byte [] inbuf, int position,
                  Object outbuf, int offset, int outcount,
                  Datatype datatype) throws MPIException {...}
```

```
  public int Pack_size(int incount, Datatype datatype)
                                          throws MPIException {...}


  // Process Topologies

  int Topo_test() throws MPIException {...}


  // Environmental Management

  public void Errorhandler_set(Errhandler errhandler)
                                          throws MPIException {...}

  public Errhandler Errorhandler_get() throws MPIException {...}

  void Abort(int errorcode) throws MPIException {...}

  ...
}
```

8.3 Intracomm and Intercomm

```
public class Intracomm extends Comm {

  public Object clone() { ... }

  public Intracomm Create(Group group) throws MPIException {...}

  public Intracomm Split(int colour, int key) throws MPIException {...}

  // Collective communication

  public void Barrier() throws MPIException {...}

  public void Bcast(Object buffer, int offset, int count,
                    Datatype datatype, int root)
                                        throws MPIException {...}

  public void Gather(Object sendbuf, int sendoffset,
                     int sendcount, Datatype sendtype,
                     Object recvbuf, int recvoffset,
                     int recvcount, Datatype recvtype, int root)
                                        throws MPIException {...}

  public void Gatherv(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int [] recvcount, int [] displs,
                      Datatype recvtype, int root)
                                        throws MPIException {...}

  public void Scatter(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int recvcount, Datatype recvtype, int root)
                                        throws MPIException {...}

  public void Scatterv(Object sendbuf, int sendoffset,
                       int [] sendcount, int [] displs,
                       Datatype sendtype,
                       Object recvbuf, int recvoffset,
                       int recvcount, Datatype recvtype, int root)
                                        throws MPIException {...}

  public void Allgather(Object sendbuf, int sendoffset,
                        int sendcount, Datatype sendtype,
                        Object recvbuf, int recvoffset,
                        int recvcount, Datatype recvtype)
```

```
                                    throws MPIException {...}

public void Allgatherv(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       Object recvbuf, int recvoffset,
                       int [] recvcounts, int [] displs,
                       Datatype recvtype) throws MPIException {...}

public void Alltoall(Object sendbuf, int sendoffset,
                     int sendcount, Datatype sendtype,
                     Object recvbuf, int recvoffset,
                     int recvcount, Datatype recvtype)
                                    throws MPIException {...}

public void Alltoallv(Object sendbuf, int sendoffset,
                      int [] sendcount, int [] sdispls,
                      Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int [] recvcount, int [] rdispls,
                      Datatype recvtype) throws MPIException {...}

public void Reduce(Object sendbuf, int sendoffset,
                   Object recvbuf, int recvoffset,
                   int count, Datatype datatype,
                   Op op, int root) throws MPIException {...}

public void Allreduce(Object sendbuf, int sendoffset,
                      Object recvbuf, int recvoffset,
                      int count, Datatype datatype,
                      Op op) throws MPIException {...}

public void Reduce_scatter(Object sendbuf, int sendoffset,
                           Object recvbuf, int recvoffset,
                           int [] recvcounts, Datatype datatype,
                           Op op) throws MPIException {...}

public void Scan(Object sendbuf, int sendoffset,
                 Object recvbuf, int recvoffset,
                 int count, Datatype datatype,
                 Op op) throws MPIException {...}


// Topology Constructors

public Graphcomm Create_graph(int [] index, int [] edges,
                              boolean reorder)
                                    throws MPIException {...}
```

```
  public Cartcomm Create_cart(int [] dims, boolean [] periods,
                              boolean reorder)
                                          throws MPIException {...}

  ...
}

public class Intercomm extends Comm {

  public Object clone() { ... }

  // Inter-communication

  public int Remote_size() throws MPIException {...}

  public Group Remote_group() throws MPIException {...}

  public Intracomm Merge(boolean high) throws MPIException {...}

  ...
}
```

8.4 Op

```
public class Op {
  Op(User_function function, boolean commute) throws MPIException {...}

  void finalize() throws MPIException {...}

  ...
}
```

8.5 Group

```
public class Group {

  // Group Management

  public int Size() throws MPIException {...}

  public int Rank() throws MPIException {...}

  public static int [] Translate_ranks(Group group1, int [] ranks1,
                                       Group group2)
                                            throws MPIException {...}

  public static int Compare(Group group1, Group group2)
                                            throws MPIException {...}

  public static Group Union(Group group1, Group group2)
                                            throws MPIException  {...}

  public static Group Intersection(Group group1, Group group2)
                                            throws MPIException  {...}

  public static Group Difference(Group group1, Group group2)
                                            throws MPIException  {...}

  public Group Incl(int [] ranks) throws MPIException {...}

  public Group Excl(int [] ranks) throws MPIException {...}

  public Group Range_incl(int [] [] ranges) throws MPIException {...}

  public Group Range_excl(int [] [] ranges) throws MPIException {...}

  public void finalize() throws MPIException {...}

  ...
}
```

8.6 Status

```
public class Status {

  public int source;
  public int tag;

  public int index ;

  // Blocking Send and Receive operations

  public int Get_count(Datatype datatype) throws MPIException {...}


  // Probe and Cancel

  public boolean Test_cancelled() throws MPIException {...}


  // Derived datatypes

  public int Get_elements(Datatype datatype) throws MPIException {...}

  ...
}
```

8.7 Request and Prequest

```
public class Request {

  // Nonblocking communication

  public Status Wait() throws MPIException {...}

  public Status Test() throws MPIException {...}

  public void Free() throws MPIException {...}

  public boolean Is_null() {...}

  public static Status Waitany(Request [] array_of_requests)
                                          throws MPIException {...}

  public static Status Testany(Request [] array_of_requests)
                                          throws MPIException {...}

  public static Status [] Waitall(Request [] array_of_requests)
                                          throws MPIException {...}

  public static Status [] Testall(Request [] array_of_requests)
                                          throws MPIException {...}

  public static Status [] Waitsome(Request [] array_of_requests)
                                          throws MPIException {...}

  public static Status [] Testsome(Request [] array_of_requests)
                                          throws MPIException {...}


  // Probe and cancel

  public void Cancel() throws MPIException {...}

  ...
}

public class Prequest extends Request {

  // Persistent communication requests

  public void Start() throws MPIException {...}

  public static void Startall(Prequest [] array_of_requests)
                                          throws MPIException {...}
```

```
    ...
}
```

8.8 Datatype

```
public class Datatype {

  // Derived datatypes

  public static Datatype Contiguous(int count, Datatype oldtype)
                                          throws MPIException {...}

  public static Datatype Vector(int count, int blocklength, int stride,
                                Datatype oldtype)
                                          throws MPIException {...}

  public static Datatype Hvector(int count, int blocklength, int stride,
                                 Datatype oldtype)
                                          throws MPIException {...}

  public static Datatype Indexed(int [] array_of_blocklengths,
                                 int [] array_of_displacements,
                                 Datatype oldtype)
                                          throws MPIException {...}

  public static Datatype Hindexed(int [] array_of_blocklengths,
                                  int [] array_of_displacements,
                                  Datatype oldtype)
                                          throws MPIException {...}

  public static Datatype Struct(int [] array_of_blocklengths,
                                int [] array_of_displacements,
                                Datatype [] array_of_types)
                                          throws MPIException {...}

  public int Extent() throws MPIException {...}

  public int Lb() throws MPIException {...}

  public int Ub() throws MPIException {...}

  public int Size() throws MPIException {...}

  public void Commit() throws MPIException {...}

  public void finalize() throws MPIException {...}

  ...
}
```

8.9 Classes for virtual topologies

```
public class Cartcomm extends Intracomm {

  public Object clone() { ... }

  // Topology Constructors

  static public Dims_create(int nnodes, int [] dims)
                                          throws MPIException {...}

  public CartParms Get() throws MPIException {...}

  public int Rank(int [] coords) throws MPIException {...}

  public int [] Coords(int rank) throws MPIException {...}

  public ShiftParms Shift(int direction, int disp) throws MPIException {...}

  public Cartcomm Sub(boolean [] remainDims) throws MPIException {...}

  public int Map(int [] dims, boolean [] periods) throws MPIException {...}
}

public class CartParms {

  // Return type for Cartcomm.get()

  public int [] dims ;
  public booleans [] periods ;
  public int [] coords ;
}

public class ShiftParms {

  // Return type for Cartcomm.shift()

  public int rankSource ;
  public int rankDest ;
}

public class Graphcomm extends Intracomm {

  public Object clone() { ... }

  // Topology Constructors

  public GraphParms Get() throws MPIException {...}
```

```
  public int [] Neighbours(int rank) throws MPIException {...}

  public int Map(int [] index, int [] edges) throws MPIException {...}
}

public class GraphParms {

  // Return type for Graphcomm.get()

  public int [] index ;
  public int [] edges ;
}
```

References

[1] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice Hall, 1998.

[2] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[3] Message Passing Interface Forum. MPI-2: Extension to the message passing interface. Technical report, University of Tennessee, July 1997. http://www.mpi-forum.org.