MPJ: MPI-like Message Passing for Java

Bryan Carpenter[1], Vladimir Getov[2], Glenn Judd[3],
Anthony Skjellum[4] and Geoffrey Fox[1]

[1]NPAC, Syracuse University, Syracuse, USA
[2]School of Computer Science, University of Westminster, London, UK
[3]Computer Science Department, Brigham Young University, Provo, USA
[4]MPI Software Technology, Inc., Starkville, USA

Abstract

Recently, there has been a lot of interest in using Java for parallel programming. Efforts have been hindered by lack of standard Java parallel programming APIs. To alleviate this problem, various groups started projects to develop Java message passing systems modelled on the successful Message Passing Interface (MPI). Official MPI bindings are currently defined only for C, Fortran, and C++, so early MPI-like environments for Java have been divergent. This paper relates an effort undertaken by a working group of the Java Grande Forum, seeking a consensus on an MPI-like API, to enhance the viability of parallel programming using Java.

1 Introduction and Background

A likely prerequisite for parallel programming in a distributed environment is a good message passing API. Java comes with various ready-made packages for communication, notably an easy-to-use interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. Interesting as these interfaces are, it is questionable whether parallel programmers will find them especially convenient. Sockets and remote procedure calls have been around for approximately as long as parallel computing has been fashionable, and neither of them has been popular in that field. Both of these communication models are optimized for client-server programming, whereas the parallel computing world is mainly concerned with a more symmetric model, where communications occur in groups of interacting peers.

This peer-to-peer model of communication is captured in the successful Message Passing Interface (MPI) standard, established in 1994 [15]. MPI directly supports the Single Program Multiple Data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values. Reliable point-to-point communication is provided through a shared, group-wide communicator, instead of socket pairs. MPI allows numerous blocking, non-blocking, buffered or synchronous communication modes. It also provides a library of true collective operations (broadcast is the most trivial example). An extended standard, MPI-2 [16], allows for dynamic process creation and access to memory in remote processes.

The MPI standard documents provided a language-independent specification as well as language-specific (C and Fortran) bindings [15]. While the MPI-2 release of the standard added a C++ binding [16], no Java binding has been offered or is planned by the MPI Forum. With the evident success of Java as a programming language, and its inevitable use in connection with parallel as well as distributed computing, the absence of a well-designed language-specific binding for message-passing with Java will lead to divergent, non-portable practices. Indeed MPI-like binding for Java were developed independently by several teams. These will be briefly reviewed in the next section.

Over the last three years supporters of the *Java Grande Forum* [8] have been working actively to address some of the issues involved in using Java for technical computation. The goal of the forum has been to develop consensus and recommendations on possible enhancements to the Java language and associated Java standards, for large-scale ("Grande") applications. Through a series of ACM-supported workshops and conferences the forum has helped stimulate research on Java compilers and programming environments. The Message-Passing Working Group of the Java Grande Forum was formed just over a year ago as a response to the appearance of the various APIs for message-passing. An immediate goal was to discuss a common API for MPI-like Java libraries. An initial draft for a common API specification was distributed at Supercomputing '98 [5]. Since then the working group met in San Francisco and Syracuse, and a Birds of a Feather meeting was held at Supercomputing '99. Minutes of meetings are available at [9, 10]. To avoid confusion with standards published by the original MPI Forum the nascent API is called MPJ (Message Passing interface for Java).

2 Earlier Work

At the time the working group was created there were several known efforts towards the design of early MPI-like interfaces for Java with three fully functional but different implementations—mpiJava [3], JavaMPI [17], and MPIJ [12]. The implementation of mpiJava is based on the use of native methods to build a wrapper to existing MPI library (MPICH). A comparable approach has been followed in the development of JavaMPI, but the JavaMPI wrappers were automatically generated by a special-purpose code generator. A large subset of MPI-like functions called MPIJ is implemented in pure Java within the DOGMA system for Java-based parallel programming. MPI Software Technology, Inc. announced a commercial effort to develop a message-passing framework and parallel support environment for Java called JMPI [6]. Some of these "proof-of-concept" implementations have been available since 1997 with successful ports on clusters of workstations running Solaris, Windows NT, Irix, AIX, HP-UX, MacOS, and Linux, as well as the IBM SP2, SGI Origin-2000, Fujitsu AP3000, and Hitachi SR2201 parallel platforms.

2.1 The mpiJava wrapper

The mpiJava software [3] implements a Java binding for MPI proposed late in 1997. The API is modeled as closely as practical on the C++ binding defined in the MPI 2.0 standard, specifically supporting the MPI 1.1 subset of that standard. In some cases the extra runtime information available in Java objects allows argument lists to be simplified relative to the C++ binding. In other cases restrictions of Java, especially the fact that all arguments are passed by value in Java, forces some changes to argument lists. But in general mpiJava adheres closely to earlier standards.

The implementation of mpiJava is through JNI wrappers to native MPI software. Interfacing Java to MPI is not always trivial. We often see low-level conflicts between the Java runtime and the interrupt mechanisms used in MPI implementations. The situation is improving as JDK matures, and the mpiJava software now works reliably on top of Solaris MPI implementations and various shared memory platforms. A port to Windows NT (based on WMPI) is available, and other ports are in progress.

Other work in progess includes development of demonstrator applications, and Java-specific extensions such as support for direct communication of serializable objects.

2.2 JavaMPI—automatic generation of MPI wrappers

In principle, the binding of existing MPI library to Java using JNI amounts to either dynamically linking the library to the Java virtual machine, or linking the library to the object code produced by a stand-alone Java compiler. Complications stem from the fact that Java data formats are in general different from those of C. Java implementations will have to use JNI which allows C functions to access Java data and perform format conversion if necessary. Such an interface is fairly convenient for writing *new* C code to be called from Java, but is not adequate for linking *existing* native code.

Clearly an additional interface layer must be written in order to bind a legacy library to Java. A large library like MPI has over a hundred exported functions, therefore it is preferable to automate the creation of the additional interface layer. The *Java-to-C interface generator* (JCI) [7] takes as input a header file containing the C function prototypes of the native library. It outputs a number of files comprising the additional interface: a file of C stub-functions; files of Java class and native method declarations; shell scripts for doing the compilation and linking. The JCI tool generates a C stub-function and a Java native method declaration for each exported function of the MPI library. Every C stub-function takes arguments whose types correspond directly to those of the Java native method, and converts the arguments into the form expected by the C library function.

As the JavaMPI bindings have been generated automatically from the C prototypes of MPI functions, they are very close to the C binding. However, there is nothing to prevent from parting with the C–style binding and adopting a Java-style object–oriented approach by grouping MPI functions into a hierarchy of classes.

2.3 MPIJ—MPI-like implementation in pure Java

MPIJ is a completely Java-based implementation of MPI which runs as part of the Distributed Object Group Metacomputing Architecture (DOGMA) system. MPIJ implements a large subset of MPI-like functionality including all modes of point-to-point communication, intracommunicator operations, groups, and user-defined reduction operations. Notable capabilities that are not yet implemented include process topologies, intercommunicators, and user-defined datatypes but these are arguably needed for legacy code only.

MPIJ communication uses native marshaling of primitive Java types. On Win32 platforms this technique allows MPIJ to achieve communication speeds comparable to, and in some instances exceeding, native MPI implementations [13]. Our performance evaluation experiments show that Java communication speed would be greatly increased if native marshaling were a core Java function.

A key feature of a pure Java MPI-like implementation is the ability to function on applet-based nodes. In MPIJ, this provides a flexible method for creating clusters of workstations without the need to install any system or user software related to the message-passing environment on the participating nodes.

3 The MPJ API Specification

3.1 Rationale

The MPI standard is explicitly object-based. The C and Fortran bindings rely on "opaque objects" that can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. The C++ binding specified in the MPI-2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The draft MPJ API specification follows this model, lifting the structure of its class hierarchy directly from the C++ binding.

The initial specification builds directly on the MPI-1 infrastructure provided by the MPI Forum, together with language bindings motivated by the C++ bindings of MPI-2. The purpose of this phase of the effort is to provide an immediate, ad hoc standardization for common message passing programs in Java, as well as to provide a basis for conversion between C, C++, Fortran 77, and Java. Eventually, support for other parts of MPI-2 also

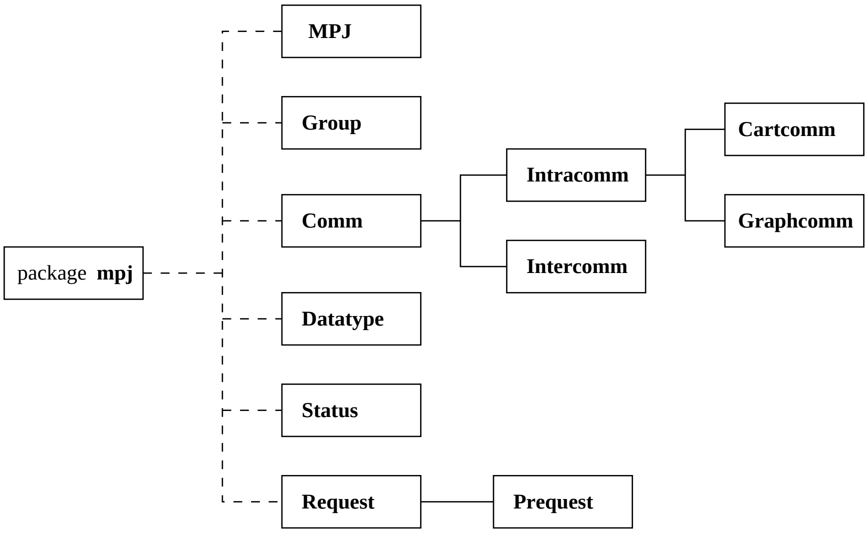

Figure 1: Principal classes of MPJ

belong here, particularly dynamic process management[1]. The position of the working group was that the initial MPI-centric API should subsequently be extended with more object-oriented, Java-centric features, although the exact requirements for this later phase have not yet been established.

The major classes of the MPJ specification are illustrated in Figure 1. The class `MPJ` only has static members. It acts as a module containing global services, such as initialization, and many global constants including the default communicator `COMM_WORLD`. The most important class in the package is the communicator class `Comm`. All communication functions in MPJ are members of `Comm` or its subclasses. As usual in MPI, a communicator stands for a "collective object" logically shared by a group of processors. The processes communicate, typically by addressing messages to their peers through the common communicator. A class that will be important in the following discussion is the `Datatype` class. This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions.

3.2 Example and data types

In general the point-to-point communication operations are realized as methods of the `Comm` class. The basic point-to-point communication operations are *send* and *receive*. Their use is illustrated in Figure 2. Consider, for example, the MPJ analogue of the operation `MPI_SEND`. The method prototype is:

```
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`	data type of each item in send buffer
`dest`	rank of destination
`tag`	message tag

The data part of the message consists of a sequence of `count` values, each of the type

[1] Given its spartan implementation in the non-Java space, we may not need the whole of MPI-2.

```
import mpj.* ;

class Hello {
  static public void main(String[] args) {
    MPJ.init(args) ;

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

    MPJ.finish();
  }
}
```

Figure 2: Example MPJ program

indicated by `datatype`. The actual argument associated with `buf` must be an array with elements of corresponding type. 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-compatible basic type, its value must be consistent with the element type of `buf`. Thus, the basic data type values included in the MPJ API specification are `MPJ.BYTE`, `MPJ.CHAR`, `MPJ.SHORT`, `MPJ.BOOLEAN`, `MPJ.INT`, `MPJ.LONG`, `MPJ.FLOAT`, `MPJ.DOUBLE`, and `MPJ.OBJECT`. If the `datatype` value is `MPJ.OBJECT` the objects in the buffer are transparently serialized and unserialized inside the communication operations.

The `datatype` argument is not redundant in the current specification of MPJ, because the proposal includes support for an analogue of MPI *derived types*. The derived types of MPJ are restricted to have a unique *base type*, one of the nine types enumerated above. If the `datatype` argument of a communication function represents an MPJ derived type, its base type must agree with the Java element type of the associated `buf` argument. Alternatively, if it was decided to remove derived types from MPJ, `datatype` arguments could be removed from many functions, and Java runtime inquiries could be used internally to extract the element type of the buffer[2].

3.3 MPJ as an MPI-like language binding

MPJ does not have the status of an official language binding for MPI. But, as a matter of interest, this section will compare some surface features of the Java API with standard MPI language bindings.

All MPJ classes belong to the package `mpj`. Conventions for capitalization, etc, in class and member names generally follow the recommendations of Sun's Java code conventions [19]. In general these conventions are consistent with the naming conventions of the MPI 2.0 C++ standard. Exceptions to this rule include the use of lower case for the first letters of method names, and avoidance of underscore in variable names.

[2]Or methods like `send` could be overloaded to accept buffers with elements of the nine basic types. The disadvantage of this approach is that it leads to a major proliferation in the number of methods.

With MPI opaque objects replaced by Java objects, MPI destructors can be absorbed into Java object destructors (`finalize` methods), called automatically by the Java garbage collector. MPJ adopts this strategy as the general rule. Explicit calls to destructor functions are typically omitted from the Java user code. An exception is made for the `Comm` classes. In MPI the destructor for a communicator is a collective operation, and the user must ensure that calls are made at consistent times on all processors involved. Automatic garbage collection would not guarantee this. Hence the MPJ `Comm` class has an explicit `free` method.

Some options allowed for derived data types in the C and Fortran bindings are absent from MPJ. In particular, the Java virtual machine does not support any concept of a global linear address space. Therefore, physical memory displacements between fields in objects are unavailable or ill-defined. This puts some limits on the possible uses of any analogues of the *MPI_TYPE_STRUCT* type constructor. In practice the MPJ `struct` data type constructor has been further restricted in a way that makes it impossible to send mixed basic data types in a single message. However, this should not be a serious problem, since the set of basic data types in MPJ is extended to include serializable Java objects.

Array size arguments are often omitted in MPJ, because they can be picked up within the function by reading the `length` member of the array argument. A crucial exception is for message buffers, where an explicit count is always given. Message buffers aside, typical array arguments to MPI functions (e.g., 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 an extra size argument to select a subset. (Moreover, if derived data types are being used, the required value of the `count` argument is always different to the buffer length.) C and Fortran both have ways of treating a section of an array, offset from the beginning of the array, as if it was an array in its own right. Java does not have any such mechanism. To provide the same flexibility in MPJ, an explicit integer `offset` parameter also accompanies any buffer argument. This defines the position in the Java array of the first element actually treated as part of the buffer.

The C and Fortran languages define a straightforward mapping (or "sequence association") between their multidimensional arrays and equivalent one-dimensional arrays. In MPI a multidimensional array passed as a message buffer argument is generally treated like a one-dimensional array with the same element type. Offsets in the buffer (such as offsets occuring in derived data types) behave like offsets in the effective one-dimensional array. 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 the MPJ interface message buffers are always treated as 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 and behaviour is significantly different.

Unlike the standard MPI interfaces, MPJ methods do not return explicit error codes. Instead, the Java exception mechanism is used to report errors.

3.4 Complete draft API

The appendix of this paper lists the public interfaces of all the classes. Of course this only defines syntax. A more complete description of the semantics of all methods is available in [5].

4 Open Issues

The API described in [5] is not assumed to be "final". It was originally presented as a starting point for discussion. In this section we will mention some areas we consider to be open to improvement.

4.1 Derived data types

It is unclear whether a Java interface should support MPI-like derived data types. A proposal for a Java-compatible subset of derived types is included in the draft specification document [5], but deleting it would simplify the API significantly. In particular `datatype` arguments for buffers could be dropped.

One factor in favor of including MPI-like derived data types in MPJ is the support for legacy MPI applications. The possible need to interact with native code that uses derived data types is probably best supported by including derived data types in the MPJ API specification.

It has been argued that the functionality of derived data types is already provided by Java objects, and supporting both only adds unneeded complexity. But in fact there are good reasons to retain some additional functionality of derived data types. Any scientific code, written in Java or otherwise, will benefit from the ability to efficiently and conveniently send *sections* (subsets) of program arrays. In MPI, this is one of the most useful roles of the so-called derived data types, and MPJ object data types do not address this requirement. The discussion of whether derived data types are to be supported in MPJ should therefore be closely linked with the discussion of how true "scientific" (multi-dimensional) arrays, allowing Fortran-90-like sectioning operations, should be handled.

4.2 Multidimensional arrays

Some specific support for communicating multidimensional arrays would be desirable. In the current proposal, sending a multidimensional array involves either sending one row at a time or using Java object serialization, both of which will introduce performance bottlenecks. For instance, our experience has shown that MPIJ sends a 200x200 array of doubles over Fast Ethernet much faster when multidimensional array support is included than when individual rows are sent. More detailed analysis of this problem is presented in [4, 13].

Trying to fix the problem for standard Java multidimensional arrays is probably the wrong approach. There is a deeper problem that the Java "array-of-arrays" model for multidimensional arrays is not especially well-suited for "scientific" computation. This issue is being actively addressed by other groups in the Java Grande Forum. In particular the work by IBM on the Array package [18], which has been adopted by the Java Grande Numerics working group, is very relevant. A more complete MPJ specification should probably include mechanisms for efficiently communicating standardized "scientific" arrays, and their sections.

In fact, if a standard like the Array package were adopted, and if it supported description of array sections (without copying elements), it is quite likely that the remaining arguments in favour of keeping an MPI-like derived data type mechanism would go away.

4.3 Overloaded communication operations

It has been suggested that many of the communication operations should be overloaded to provide simplified variants that omit arguments like `offset`, `count` (and possibly `datatype`). This suggestion is not included in the current proposal, but it could be added. The primary argument in favor is that it simplifies user code. For instance,

```
MPJ.COMM_WORLD.send(message, 0, message.length, MPJ.CHAR, 1, 99);
```

becomes

```
MPJ.COMM_WORLD.send(message, MPJ.CHAR, 1, 99);
```

The obvious counter-argument is that this very significantly increases the total number of methods in the API. A possible compromise is to provide overloaded versions only of specific common functions such as point-to-point communication functions (the argument against this, in turn, is that it looks inconsistent).

4.4 Other issues

The current draft MPJ specification supports all MPI-like error handling using the Java exception model. An alternative suggestion that has been put forward is that all MPJ exceptions be derived from two classes: `MPJException` and `MPJRuntimeException`. Subclasses of `MPJException` would represent errors that the user would be required to catch whereas subclasses of `MPJRuntimeException` would represent uncommon or unusual errors. It has also been suggested that certain MPJ exceptions could carry subexceptions when the cause of the error is another exception. Whether, or not, to utilize MPI-like user-defined and predefined error handlers is also an open question. In principle, these error handlers could still serve a purpose in addition to the exception mechanism mentioned above.

It has been suggested that the specification of user-defined operations could be simplified. In the current proposal, which is modelled after a procedural approach, a more complex or unique operation can be created in two phases. Initially users define functions and then create a new operation class (`Op`). This results in the creation of an extra class (`UserDefinedOperation`) which is not really necessary. An alternative approach would be to simply have users define subclasses of the class `Op` with a named method (for example, `call`). This design would also eliminate the overhead associated with method invocation.

A profiling interface for MPJ has not yet been defined. A possible general design approach is for profiling class and method names to exactly match those of the non-profiling classes and methods. Implementors would then place the compiled binary files in different locations. As Java linking is always dynamic, this would allow users to enable or disable profiling simply selecting the appropriate codebase (e.g. by changing the CLASSPATH environment variable).

5 Discussion and Conclusion

An initial goal of the Java Grande Message Passing working group was to promote a standardized MPI binding for Java. It became apparent that this road was likely to produce a collision of interest with the existing MPI community, and the name of the new API was changed to MPJ. MPJ was designated an "MPI-like" specification. The current specification is available in [5]. This specification is essentially complete and self-contained, but as discussed in section 4, it is not necessarily considered "final".

Because the proposed API was designed on object-oriented principles, most of the original MPI specification actually maps very naturally into Java. So long as one accepts the Java Grande premise that Java is an excellent basis for technical computing, an MPI-like approach to parallel computing seems very promising—more promising than some have assumed. But there remain non-obvious issues about supporting basic MPI functionality. Some of the more difficult ones boil down to the lack of a good model of scientific arrays in Java. This issue is somewhat outside the purview of this working group, but is being actively discussed by the Java Grande Numerics working group [11].

Reference implementations of the MPJ specification are currently (March, 2000) under development. An implementation based on JNI wrappers to native MPI will be created by adapting the mpiJava wrappers [3]. While this is a good approach in some situations, it has various disadvantages and conflicts with the ethos of Java, where pure-Java, write-once-run-anywhere software is the order of the day. A design for a *pure-Java* reference implementation of MPJ has also been outlined [2]. In this case, design goals were that the system should be as easy to install on distributed systems as we can reasonably make it, and that it be sufficiently robust to be useable in an Internet environment.

Back in 1994, MPI-1 was originally designed with relatively static platforms in mind. To better support computing in volatile Internet environments, modern message passing designs for Java will have to support (at least) features such as dynamic spawning of process groups and parallel client/server interfaces as introduced in the MPI-2 specification. In addition, a natural framework for dynamically discovering new compute resources and establishing connections between running programs already exists in Sun's Jini project [1], and one line of investigation is into MPJ implementations operating in the Jini framework.

Closely modelled as it is on the MPI standards, the existing MPJ specification should be regarded as a first phase in a broader program to define a more Java-centric high performance message-passing environment. In future a detachment from legacy implementations involving Java on top of native methods will be emphasized. We should consider the possibility of layering the messaging middleware over standard transports and other Java-compliant middleware (like CORBA). In a sense, the middleware developed at this level should offer a choice of emphasis between performance or generality, while always supporting portability. We note an opportunity to study and standardize aspects of real-time and fault-aware programs, drawing on the concepts learned in the *MPI/RT* activity [14]. For performance, we should seek to take advantage of what has been learned since MPI-1 and MPI-2 were finalized, or ignored in MPI standardization for various reasons—for instance drawing on the body of knowledge completed within the MPI/RT Forum. From here we may at least glean design hints concerning channel abstractions, and the more direct use of object-oriented design for message passing than was seen in MPI-1 or MPI-2. The value of this type of messaging middleware in the embedded and real-time Java application spaces should also be considered.

Of course, a primary goal in the above mentioned, both current and future work, should be the aim to offer MPI-like services to Java programs in an upward compatible fashion. The purposes are twofold: performance and portability.

References

[1] Ken Arnold, Bryan O'Sullivan, Robert Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison Wesley, 1999.

[2] Mark Baker and Bryan Carpenter. MPJ: A proposed Java message-passing API and environment for high performance computing. In *International Workshop on Java for Parallel and Distributed Computing*, Cancun, Mexico, May 2000. To be presented.

[3] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. In *First UK Workshop on Java for High Performance Network Computing*, September 1998. mpiJava Home Page: http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

[4] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[5] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPI for Java: Position Document and Draft Specification. Technical report, Java Grande Forum, November 1998. http://www.javagrande.org/reports.htm.

[6] George Crawford III, Yoginder Dandass, and Anthony Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users, December 1997. http://www.mpi-softtech.com/publications/JMPI_121797.html.

[7] Vladimir Getov, Paul Gray, Sava Mintchev, and Vaidy Sunderam. Multi-language programming environments for high performance java computing. *Scientific Programming*, 7(2):139–146, 1999.

[8] Java Grande Forum. http://www.javagrande.org.

[9] Java Grande Message Passing Working Group. Minutes of Jun 14, 1999 meeting in San Francisco. http://www.npac.syr.edu/projects/java-mpi/jul99/msg00000.html.

[10] Java Grande Message Passing Working Group. Minutes of Oct 1, 1999 meeting in Syracuse. http://www.npac.syr.edu/projects/java-mpi/oct99/msg00000.html.

[11] Java Grande Numerics Working Group. http://math.nist.gov/javanumerics/.

[12] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed Object Group Metacomputing Architecture. *Concurrency: Practice and Experience*, 10(11/13):977–983, 1998. MPIJ Home Page: http://ccc.cs.byu.edu/DOGMA/.

[13] Glenn Judd, Mark Clement, Quinn Snell, and Vladimir Getov. Design issues for efficient implementation of mpi in java. In *Proceedings of ACM 1999 Java Grande Conference*, pages 58–65. ACM Press, 1999.

[14] Arkady Kanevsky, Anthony Skjellum, and Anna Rounbehler. MPI/RT—an emerging standard for high-performance real-time systems. In *31st Hawaii International Conference on System Sciences*, volume III, January 1998. MPI/RT Home Page: http://www.mpirt.org.

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

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

[17] Sava Mintchev and Vladimir Getov. Towards portable message passing in Java: Binding MPI. In M. Bubak, J. Dongarra, and J. Waśniewski, editors, *Recent Advances in PVM and MPI*, volume 1332 of *Lecture Notes in Computer Science*, pages 135–142. Springer Verlag, 1997. JavaMPI Home Page: http://perun.hscs.wmin.ac.uk/JavaMPI/.

[18] Jose Moreira, Sam Midkiff, Manish Gupta, and Rick Lawrence. High performance computing with the array package for Java: A case study using data mining. In *Supercomputing 99*, November 1999.

[19] Sun Microsystems. Java code conventions. http://java.sun.com/docs/codeconv/.

A Public Interface of Classes in MPJ Draft Specification

A.1 MPJ

```
public class MPJ {
  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 bufferAttach(byte [] buffer) throws MPJException {...}

  public static byte [] bufferDetach() throws MPJException {...}


  // Environmental Management

  public static String [] init(String[] argv) throws MPJException {...}

  public static void finish() throws MPJException {...}

  public static String getProcessorName() throws MPJException {...}

  public static double wtime() throws MPJException {...}

  public static double wtick() throws MPJException {...}

  public static boolean initialized() throws MPJException {...}

  ...
}
```

A.2 Comm

```
public class Comm {

  // Communicator Management

  public int size() throws MPJException {...}

  public int rank() throws MPJException {...}

  public Group group() throws MPJException {...}  // (section "Group management" of spec)

  public static int compare(Comm comm1, Comm comm2) throws MPJException {...}

  public Object clone() {...}

  public void free() throws MPJException {...}


  // Inter-communication

  public boolean testInter() throws MPJException {...}

  public Intercomm createIntercomm(Comm localComm, int localLeader,
                                   int remoteLeader, int tag) throws MPJException {...}


  // Caching

  public Object attrGet(int keyval) throws MPJException {...}
```

```
// Blocking Send and Receive operations

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

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


// Communication Modes

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

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

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


// Nonblocking communication

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

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

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

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

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


// Probe and cancel

public Status iprobe(int source, int tag) throws MPJException {...}

public Status probe(int source, int tag) throws MPJException {...}


// Persistent communication requests

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

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

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

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

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


// 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 MPJException {...}

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

  // Pack and unpack

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

  byte[] pack(Object inbuf, int offset, int incount, Datatype datatype)
                      throws MPJException {...}

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

  public int packSize(int incount, Datatype datatype) throws MPJException {...}


  // Process Topologies

  int topoTest() throws MPJException {...}


  // Environmental Management

  public static void errorhandlerSet(Errhandler errhandler) throws MPJException {...}

  public static Errhandler errorhandlerGet() throws MPJException {...}

  void abort(int errorcode) throws MPJException {...}

  ...
}
```

A.3 Intracomm and Intercomm

```
public class Intracomm extends Comm {

  public Object clone() { ... }

  public Intracomm create(Group group) throws MPJException {...}

  public Intracomm split(int colour, int key) throws MPJException {...}

  // Collective communication

  public void barrier() throws MPJException {...}

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

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

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

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

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

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

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

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

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

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

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

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

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


  // Topology Constructors

  public Graphcomm createGraph(int [] index, int [] edges,
                               boolean reorder) throws MPJException {...}

  public Cartcomm createCart(int [] dims, boolean [] periods,
                             boolean reorder) throws MPJException {...}

  ...
}

public class Intercomm extends Comm {

  public Object clone() { ... }

  // Inter-communication

  public int remoteSize() throws MPJException {...}

  public Group remoteGroup() throws MPJException {...}

  public Intracomm merge(boolean high) throws MPJException {...}

  ...
}
```

A.4 Op

```
public class Op {
  Op(UserFunction function, boolean commute) throws MPJException {...}
```

```
  void finalize() {...}

  ...
}
```

A.5 Group

```
public class Group {

  // Group Management

  public int size() throws MPJException {...}

  public int rank() throws MPJException {...}

  public int [] translateRanks(Group group1, int [] ranks1) throws MPJException {...}

  public static int compare(Group group1, Group group2) throws MPJException {...}

  public static Group union(Group group1, Group group2) throws MPJException {...}

  public static Group intersection(Group group1, Group group2) throws MPJException {...}

  public static Group difference(Group group1, Group group2) throws MPJException {...}

  public Group incl(int [] ranks) throws MPJException {...}

  public Group excl(int [] ranks) throws MPJException {...}

  public Group rangeIncl(int [] [] ranges) throws MPJException {...}

  public Group rangeExcl(int [] [] ranges) throws MPJException {...}

  public void finalize() {...}

  ...
}
```

A.6 Status

```
public class Status {

  public int index ;


  // Blocking Send and Receive operations

  public int getCount(Datatype datatype) throws MPJException {...}

  public int getSource() throws MPJException {...}
  public int getTag() throws MPJException {...}

  // Nonblocking communication

  public int getIndex() throws MPJException {...}

  // Probe and Cancel

  public boolean testCancelled() throws MPJException {...}


  // Derived datatypes

  public int getElements(Datatype datatype) throws MPJException {...}

  ...
}
```

A.7 Request and Prequest

```
public class Request {

  // Nonblocking communication

  public Status wait() throws MPJException {...}

  public Status test() throws MPJException {...}

  public Request() throws MPJException {...}

  public void finalize() {...}

  public boolean isVoid() throws MPJException {...}

  public static Status waitAny(Request [] arrayOfRequests) throws MPJException {...}

  public static Status testAny(Request [] arrayOfRequests) throws MPJException {...}

  public static Status [] waitAll(Request [] arrayOfRequests) throws MPJException {...}

  public static Status [] testAll(Request [] arrayOfRequests) throws MPJException {...}

  public static Status [] waitSome(Request [] arrayOfRequests) throws MPJException {...}

  public static Status [] testSome(Request [] arrayOfRequests) throws MPJException {...}


  // Probe and cancel

  public void cancel() throws MPJException {...}

  ...
}

public class Prequest extends Request {

  // Persistent communication requests

  public void start() throws MPJException {...}

  public static void startAll(Request [] arrayOfRequests) throws MPJException {...}

  ...
}
```

A.8 Datatype

```
public class Datatype {

  // Derived datatypes

  public Datatype contiguous(int count) throws MPJException {...}

  public Datatype vector(int count, int blocklength, int stride) throws MPJException {...}

  public Datatype hvector(int count, int blocklength, int stride) throws MPJException {...}

  public Datatype indexed(int [] arrayOfBlocklengths,
                          int [] arrayOfDisplacements) throws MPJException {...}

  public Datatype hindexed(int [] arrayOfBlocklengths,
                           int [] arrayOfDisplacements) throws MPJException {...}

  public static Datatype struct(int [] arrayOfBlocklengths,
                                int [] arrayOfDisplacements,
                                Datatype [] arrayOfTypes) throws MPJException {...}
```

```
  public int extent() throws MPJException {...}

  public int size() throws MPJException {...}

  public int lb() throws MPJException {...}

  public int ub() throws MPJException {...}

  public void commit() throws MPJException {...}

  public void finalize() {...}

  ...
}
```

A.9 Classes for virtual topologies

```
public class Cartcomm extends Intracomm {

  public Object clone() { ... }

  // Topology Constructors

  static public dimsCreate(int nnodes, int [] dims) throws MPJException {...}

  public CartParms get() throws MPJException {...}

  public int rank(int [] coords) throws MPJException {...}

  public int [] coords(int rank) throws MPJException {...}

  public ShiftParms shift(int direction, int disp) throws MPJException {...}

  public Cartcomm sub(boolean [] remainDims) throws MPJException {...}

  public int map(int [] dims, boolean [] periods) throws MPJException {...}
}

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 MPJException {...}

  public int [] neighbours(int rank) throws MPJException {...}

  public int map(int [] index, int [] edges) throws MPJException {...}
}
```

```
public class GraphParms {

  // Return type for Graphcomm.get()

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