MPI for Java

Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum

(Draft Section for the Java Grande Report)

1 Introduction and background

A basic prerequisite for parallel programming 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 about as long as parallel computing has been fashionable, and neither of them has been popular in that field. Both communication models are optimized for client-server programming, whereas the parallel computing world is mainly concerned with "symmetric" communication, occurring in groups of interacting peers.

This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI), established a few years ago [5]. 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 [6], allows for dynamic process creation and access to memory in remote processes.

The MPI standard document thus far has provided language-independent specification as well as language-specific (C and Fortran) bindings [5]. While the MPI-2 release of the standard added a C++ binding [6], 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. Therefore, a standard specification is urgently needed to enable the development of portable Java Grande applications using MPI.

2 Current status

There are several known efforts towards the design of early MPI interfaces for Java with three fully functional but different Java-MPI implementations – mpiJava, JavaMPI, and MPIJ. The design 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 is implemented in pure Java within the DOGMA system for Java-based parallel programming. MPI Software Technology, Inc. have also announced that there is a commercial effort under way to develop a message-passing framework and parallel support environment for Java called JMPI [3]. Some of these "proof-of-concept" implementations have been available for more than a year with successful ports on clusters of workstations running Solaris or Windows NT, as well as IBM SP2, SGI Origin-2000, Fujitsu AP3000, and Hitachi SR2201 parallel platforms.

2.1 The mpiJava wrapper

The mpiJava API is modelled as closely as practical on the C++ binding defined in the MPI 2.0 standard but currently supports only the MPI 1.1 subset [1]. A number of changes to argument lists are forced by the restriction that arguments cannot be passed by reference in Java. In general outputs of mpiJava methods come through the result value of the function. In many cases MPI functions return more than one value. 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 return an object handle which is null if the operation fails. Occasionally an extra field is added to an existing MPI class to hold extra results – for example the Status class has an extra field, index, initialized by functions like Waitany. Rarely none of these methods work and we resort to defining auxilliary classes to hold multiple results from a particular function. In another change to C++, we often omit array size arguments, because they can be picked up within the wrapper by reading the length member of the array argument.

As a result of these changes mpiJava argument lists are often more concise than the corresponding C or C++ argument lists.

Normally in mpiJava, MPI destructors are called by the Java finalize method for the class. This is invoked automatically by the Java garbage collector. For most classes, therefore, no binding of the MPI_class_FREE function appears in the Java API. Exceptions are Comm and Request, which do have explicit Free members. In those cases the MPI operation could have observable side-effects (beyond simply freeing resources), so their execution is left under direct control of the programmer.

2.2 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) [4] 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 Pure Java implementation of MPI

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

MPIJ communication uses native marshaling of primitive Java types. This technique allows MPIJ to achieve communication speeds comparable to, and frequently exceeding that, of native MPI implementations. (Java communication speed would be greatly increased if native marshaling were a core Java function.)

Current MPIJ work involves porting native marshalling to platforms other than Win32, investigation of standard libraries (e.g. BLAS) for improved performance, and porting to an improved version of DOGMA.

3 Proposed joint efforts

3.1 Java-MPI – API Specification

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 Java-MPI API specification follows this model, lifting the structure of its class hierarchy directly from the C++ binding. The major classes of Java-MPI are illustrated in Figure 1.

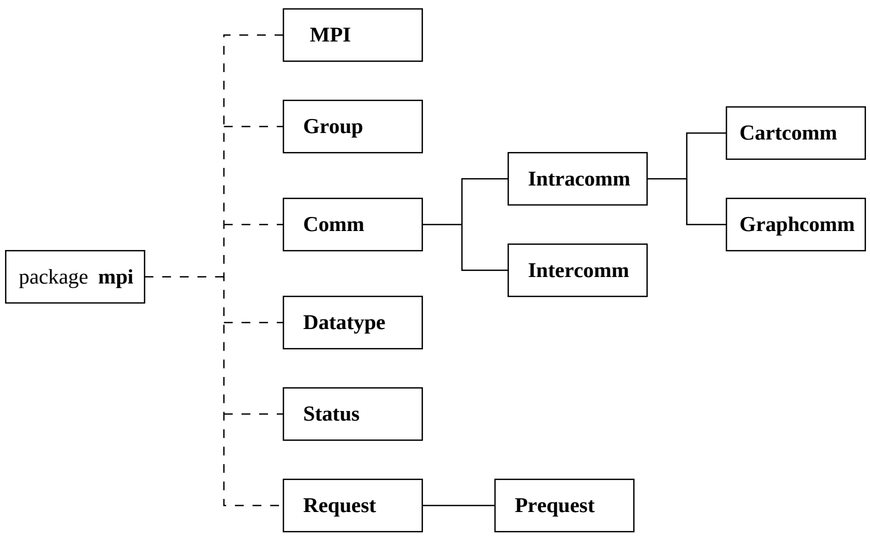

Figure 1: Principal classes of Java-MPI

The class `MPI` only has static members. It acts as a module containing global services, such as initialization of MPI, 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 Java-MPI 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. Another class that is important for the Java-MPI specification is the `Datatype` class. This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions.

The complete Draft Java-MPI API Specification is available as a separate document [2].

3.2 High-Level Message Passing for Java

[Awaiting input from Tony Skjellum]

3.3 Further Actions

The ACG currently consideres to release a Request for Proposals for the Java-MPI efforts to cover:

- Standard Java-MPI API Specification
- Java-MPI wrapper publicly available on the Web
- Intelligent generator of wrappers to legacy MPI libraries
- Pure Java MPI implementation
- Test suite
- Java-MPI Benchmarks

References

[1] B. Carpenter, G. Fox, G. Zhang, and X. Li. A draft Java binding for MPI. http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html, November, 1997.

[2] B. Carpenter, V. Getov, G. Judd. Java-MPI – API Specification (Draft Position Document). Java Grande TR-03-98, November 1998.

[3] G. Crawford III, Y. Dandass, and A. Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users. http://www.mpi-softtech.com/publications/JMPI_121797.html, MSTI, December 1997.

[4] S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In: M. Bubak, J. Dongarra, J. Waśniewski (Eds.), "Recent Advances in PVM and MPI", *Lecture Notes in Computer Science*, **1332**, 135–142, Springer Verlag, 1997.

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

[6] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, July 1997.