A Low–Level Communication Library for Java HPC

Sang Boem Lim[1], Bryan Carpenter[2], Geoffrey Fox[3]
and Han-Ku Lee[4]⋆

[1] Korea Institute of Science and Technology Information (KISTI)
Daejeon, Korea
slim@kisti.re.kr
[2] OMII, University of Southampton
Southampton SO17 1BJ, UK
dbc@ecs.soton.ac.uk
[3] Pervasive Technology Labs at Indiana University
Bloomington, IN 47404-3730
gcf@indiana.edu
[4] School of Internet and Multimedia Engineering, Konkuk University
Seoul, Korea
hlee@konkuk.ac.kr

Abstract. Designing a simple but powerful low-level communication library for Java HPC environments is an important task. We introduce new low-level communication library for Java HPC, called *mpjdev*. The mpjdev API is designed with the goal that it can be implemented *portably* on network platforms and *efficiently* on parallel hardware. Unlike MPI which is intended for the application developer, mpjdev is meant for library developers. Application level communication may be implemented on top of mpjdev. The mpjdev API itself might be implemented on top of Java sockets in a portable network implementation, or-on HPC platforms-through a JNI (Java Native Interface) to a subset of MPI.

1 Introduction

HPJava [1] is an environment for scientific and parallel programming using Java. It is based on an extended version of the Java language. HPJava incorporates all of the Java language as a subset. This means any ordinary Java class can be invoked from an HPJava program without recompilation. Moreover, a translated and compiled HPJava program is a standard Java class file that can be executed by a distributed collection of Java Virtual Machines.

Locally held elements of *multiarrays* and *distributed arrays* can be accessed using some special syntax provided by HPJava. HPJava does not provide any special syntax for accessing non-local elements. Non-local elements can only be accessed by making explicit library calls. This policy in the HPJava language,

⋆ Correspondence to: Han-ku Lee, School of Internet and Multimedia Engineering, Konkuk University, Seoul, Korea

attempts to leverage successful library-based approaches to SPMD parallel computing. This idea is in very much in the spirit of MPI, with its explicit point-to-point and collective communications. HPJava raises the level of abstraction a notch, and adds excellent support for development of libraries that manipulate distributed arrays. But it still exposes a multi-threaded, non-shared-memory, execution model to programmer. Advantages of this approach include flexibility for the programmer, and ease of compilation, because the compiler does not have to analyze and optimize communication patterns.

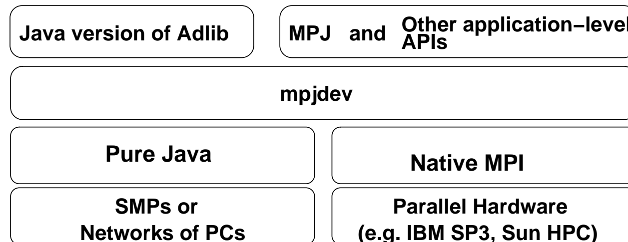

Fig. 1. An HPJava communication stack

The mpjdev [2] [3] API is designed with the goal that it can be implemented *portably* on network platforms and *efficiently* on parallel hardware. Unlike MPI which is intended for the application developer, mpjdev is meant for library developers. Application level communication libraries like the Java version of Adlib (or MPJ [1]) may be implemented on top of mpjdev. The mpjdev API itself might be implemented on top of Java sockets in a portable network implementation, or-on HPC platforms-through JNI (Java Native Interface) to a subset of MPI. The positioning of the mpjdev API is illustrated in Figure 1. Currently not all the communication stack in this figure is implemented. The Java version of Adlib, the pure Java implementation on SMPs, and native the MPI implementation are developed and included in the current HPJava or mpiJava releases. The rest of the stack may be filled in the future.

2 Communications API

In MPI there is a rich set of communication modes. Point-to-point communication and collective communication are two main communication modes of MPI. Point-to-point communication support blocking and non-blocking communication modes. Blocking communication mode includes one blocking mode receive, **MPI_RECV**, and four different send communication modes. Blocking send communication modes include standard mode, **MPI_SEND**, synchronous mode, **MPI_SSEND**, ready mode, **MPI_RSEND**, and buffered mode, **MPI_BSEND**. Non-blocking communication mode also uses one receives, **MPI_IRECV** and the same four modes as blocking send: standard, **MPI_ISEND**, synchronous,

```
public class Comm {

  public void size() { ... }
  public void id() { ... }
  public void dup() { ... }
  public void create(int [] ids) { ... }
  public void free() { ... }

  public void send(Buffer buf, int dest, int tag) { ... }
  public Status recv(Buffer buf, int src, int tag) { ... }
  public Request isend(Buffer buf, int dest, int tag) { ... }
  public Request irecv(Buffer buf, int dest, int tag) { ... }

  public static String [] init(String[] args) { ... }
  public static void finish() { ... }

  . . .
}
```

Fig. 2. The public interface of mpjdev `Comm` class.

MPI_ISSEND, ready, **MPI_IRSEND**, and buffered, **MPI_IBSEND**. Collective communication also includes various communication modes. It has characteristic collective modes like broadcast, **MPI_BCAST**, gather, **MPI_GATHER**, and scatter, **MPI_SCATER**. Global reduction operations are also included in collective communication.

The mpjdev API is much simpler. It only includes point-to-point communications. Currently the only messaging modes for mpjdev are standard blocking mode (like **MPI_SEND**, **MPI_RECV**) and standard non-blocking mode (like **MPI_ISEND**, **MPI_IRECV**), together with a couple of "wait" primitives.

The communicator class, **Comm**, is very similar to the one in MPI but it has a reduced number of functionalities. It has communication methods like **send()**, **recv()**, **isend()**, and **irecv()**, and defines constants **ANY_SOURCE**, and **ANY_TAG** as static variables. Figure 2 shows the public interface of **Comm** class.

We can get the number of processes that are spanned by this communicator by calling **size()** (similar to **MPI_COMM_SIZE**). Current id of process relative to this communicator is returned by **id()** (similar to **MPI_COMM_RANK**).

The two methods **send()** and **recv()** are blocking communication modes. These two methods block until the communication finishes. The method **send()** sends a message containing the contents of **buf** to the destination described by **dest** and message tag value **tag**.

The method **recv()** receives a message from matching source described by **src** with matching tag value **tag** and copies contents of message to the receive buffer, **buf**. The receiver may use wildcard value **ANY_SOURCE** for **src** and **ANY_TAG** for **tag** instead specifying **src** and **tag** values. These indicate that

```
public class Request {
  public Status iwait() { ... }

  public Status iwaitany(Request [] reqs) { ... }
  . . .
}
```

Fig. 3. The public interface of `Request` class.

a receiver accepts any source and/or tag of send. The **Comm** class also has the initial communicator, **WORLD**, like **MPI_COMM_WORLD** in MPI and other utility methods. The capacity of receive buffer must be large enough to accept these contents. It initializes the **source** and **tag** fields of the returned **Status** class which describes a completed communication.

The functionalities of **send()** and **recv()** methods are same as standard mode point–to–point communication of MPI (**MPI_SEND** and **MPI_RECV**). A **recv()** will be blocked until the send if posted. A **send()** will be blocked until the message have been safely stored away. Internal buffering is not guaranteed in **send()**, and the message may be copied directly into the matching receive buffer. If no **recv()** is posted, **send()** is allowed to block indefinitely, depending on the availability of internal buffering in the implementation. The programmer must allow for this–this is a low-level API for experts.

The other two communication methods **isend()** and **irecv()** are non-blocking versions of **send()** and **recv()**. These are equivalent to **MPI_ISEND** and **MPI_IRECV** in MPI. Unlike blocking send, a non-blocking send returns immediately after its call and does not wait for completion. To complete the communication a separate send complete call (like **iwait()** and **iwaitany()** methods in the **Request** class) is needed. A non-blocking receive also work similarly. The **wait()** operations block exactly as for the blocking versions of **send()** and **recv()** (e.g. the **wait()** operation for an **isend()** is allowed to block indefinitely if no matching receive is posted). The method **dup()** creates a new communicator the spanning the same set of processes, but with a distinct communication context. We can also create a new communicator spanning a selected set of processes selected using the **create()** method. The ids of array **ids** contain a list of ids relative to this communicator. Processes that are outside of the group will get a null result. The new communicator has a distinct communication context. By calling the **free()** method, we can destroy this communicator (like **MPI_COMM_FREE in MPI**). This method is called usually when this communicator is no longer in use. It frees any resources that used by this communicator.

We should call static **init()** method once before calling any other methods in communicator. This static method initializes mpjdev and makes it ready to use. The static method **finish()** (which is equivalent of **MPI_FINALIZE**) is the last method should be called in mpjdev.

The other important class is Request (Figure 3). This class is used for non-blocking communications to ensure completion of non-blocking send and receive. We wait for a single non-blocking communication to complete by calling **iwait()** method. This method returns when the operation identified by the current class is complete. The other method **iwaitany()** waits for one non-blocking communication from a set of requests **reqs** to complete. This method returns when one of the operations associated with the active requests in the array **reqs** has completed. After completion of **iwait()** or **iwaitany()** call, the source and tag fields of the returned status object are initialized. One more field, index, is initialized for **iwaitway()** method. This field indicates the index of the selected request in the **reqs** array.

3 Message Format

This section describes the message format used by mpjdev. The specification here doesn't define how a message vector which contained in the **Buffer** object is stored internally-for example it may be as a Java **byte** [] array or it may be as a C **char** [] array, accessed through native methods. But this section does define the organization of data in the buffer. It is the responsibility of the user to ensure that sufficient space is available in the buffer to hold the desired message. Trying to write too much data to a buffer causes an exception to be thrown. Likewise, trying to receive a message into a buffer that is too small will cause an exception to be thrown. These features are (arguably) in the spirit of MPI.

A message is divided into two main parts. The *primary payload* is used to store message elements of primitive type. *The secondary payload* is intended to hold the data from object elements in the message (although other uses for the secondary payload are conceivable). The size of the primary payload is limited by the fixed capacity of the buffer, as discussed above. The size of the secondary payload, if it is non-empty, is likely to be determined "dynamically"-for example as objects are written to the buffer.

The message starts with a short *primary header*, defining an *encoding scheme* used in headers and primary payload, and the total number of data bytes in the primary payload. Only one byte is allocated in the message to describe the encoding scheme: currently the only encoding schemes supported or envisaged are *big-endian* and *little-endian*. This is to allow for native implementations of the buffer operations, which (unlike standard Java read/write operations) may use either byte order. A message is divided into zero or more *sections*. Each section contains a fixed number of elements of homogeneous type. The elements in a section will all have identical *primitive* Java type, or they will all have **Object** type (in the latter case the exact classes of the objects need *not* be homogeneous within the section).

Each section has a short header in the primary payload, specifying the type of the elements, and the number of elements in the section. For sections with primitive type, the header is followed by the actual data. For sections with object type, the header is the only representation of the section appearing in the primary

payload–the actual data will go in the secondary payload. After the primary payload there is a *secondary header*. The secondary header defines the number of bytes of data in the secondary payload. The secondary header is followed in the logical message by the secondary payload. The mpjdev specification says nothing about the *layout* of the secondary payload. In practice this layout will be determined by the Java Object Serialization specification.

4 Discussion

We have explored enabling parallel, high-performance computation–in particular development of scientific software in the network-aware programming language, Java. Traditionally, this kind of computing was done in Fortran. Arguably, Fortran is becoming a marginalized language, with limited economic incentive for vendors to produce modern development environments, optimizing compilers for new hardware, or other kinds of associated software expected by today's programmers. Java looks like a promising alternative for the future.

Java introduces implementation issues for message-passing APIs that do not occur in conventional programming languages. One important issue is how to transfer data between the Java program and the network while reducing overheads of the Java Native Interface. As contribution toward new low-level APIs, we developed a low-level Java API for HPC message passing, called mpjdev. The mpjdev API is a device level communication library. This library is developed with HPJava in mind, but it is a standalone library and could be used by other systems. We discussed message buffer and communication APIs of mpjdev and also format of a message. To evaluate current communication libraries, we did various performance tests. We developed small kernel level applications and a full application for performance test. We got reasonable performance on simple applications without any serious optimization. We also evaluated a communication performance of the high- and low-level libraries for future optimization.

References

1. HPJava project home page. www.hpjava.org.
2. Sang Boem Lim. *Platforms for HPJava: Runtime Support for Scalable Programming in Java*. PhD thesis, Florida State University, June 2003.
3. Sang Boem Lim, Bryan Carpenter, Geoffrey Fox, and Han-Ku Lee. A device level communication library for the hpjava programming language. In *the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, November 2003.