A Ubiquitous Message Passing Interface Implementation in Java: *jmpi*

Kivanc Dincer
Department of Computer Engineering
Baskent University
06530 Ankara TURKEY

Abstract

jmpi is a 100% Java-based implementation of the Message-Passing Interface (MPI-1) standard. jmpi comes with an efficient and effective MPI implementation in Java and supports a user-friendly Java Application Programming Interface (API) for MPI. We present the implementation details and give some early communication benchmark performance results on a cluster of SUN UltraSparc workstations.

Keywords: MPI, Java, JPVM.

1. Introduction

Message Passing Interface (MPI) [1] provides an infrastructure that enables users to build a high-performance distributed computing environment from networked computers with minimum effort and facilitates the use of this environment using a relatively simple high-level message-passing model. MPI was proposed as a standard message-passing interface by a committee of vendors, implementers, and users.

Since its introduction in 1994, Java has quickly become a popular parallel programming and system programming language in the high-performance parallel-computing arena. Its simple design and object-oriented features simplifies the development of parallel programs. On the other hand, its platform-independent execution model, uniform and portable interface to operating system services such as networking and multi-threading, and its object-oriented tendencies increases Java's use make it an attractive option for being used to build and program thread-based, object-oriented, networked computing environments

2. MPI Implementations

Current MPI implementations can be collected under three groups: implementations in traditional languages, Java wrapper implementations where legacy message-passing libraries are called through the native method interface, and pure Java implementations.

MPICH [2], LAM MPI [3], Unify [4] and the Chimp MPI are a few of the successful examples of portable MPI implementations in traditional languages. JavaMPI [5] and MPIJava [6] are representatives of the second group. *jmpi* [7] and the commercial JMPI project [8] underway at MPI Software Technology, Inc. target to build a pure Java version of MPI. JMPI project is currently on hold until an undetermined date. MPIJ [9] which can be placed in between the second or third group implementations runs as part of the DOGMA system and uses native marshaling of primitive data types. Although DOGMA system is publicly available, MPI codes are not.

3. *jmpi*: A Java-Based MPI System

We here present an object-oriented message-passing Java class library, named *jmpi* that supports almost all features of the MPI-1.1. *jmpi* combines the advantages of Java with the well established techniques and practices of message-passing parallel processing on the networked computing environments. Specification of a natural and effective Java API for MPI is important for providing an easy-to-use interface for experienced MPI programmers. We adopted the Java API proposed in [10] with minor changes in this second release of *jmpi*.

The 100% Java implementation helps the *jmpi* achieve cross-platform portability by benefiting from the standard Java execution environment. An application written in Java is compiled into an architecture-neutral bytecode format, which then executes on a Java Virtual Machine (JVM) whose purpose is to hide the characteristics of the underlying platform. The biggest problem with current Java implementations is their inferior performance as compared to native implementations. However, fast evolving hardware support such as Java chips and recent developments in just-in-time Java compilation techniques, is likely to bring a remedy to this situation in the near future.

```
public class Reduce_on_Group_Difference {
  public static void main(String[] args) {
    int me, size, count=16, root=0;
    int sndbuf[], rcvbuf[], ranks[] = new int[16];
   Group WorldGrp, NewGrp, DiscardGrp;
   Comm  NewComm;
    try {
      MPI.Init(args);
      WorldGrp= MPI.COMM_WORLD.Comm_group();
      ranks[0] = MPI.COMM_WORLD.Comm_size()-1;
      DiscardGrp = WorldGrp.Group_incl(1, ranks);
      NewGrp = WorldGrp.Group_difference(DiscardGrp);
      NewComm=MPI.COMM_WORLD.Comm_create(NewGrp);
      me = MPI.COMM_WORLD.Comm_rank();
      //... create and initialize rcvbuf & sndbuf
      if (me != 0)    // compute on slave
  NewComm.Reduce(sndbuf,rcvbuf,count,MPI.INT,MPI.SUM,root);
     else
  NewComm.Reduce(sndbuf,rcvbuf,count,MPI.INT,MPI.SUM,root);
      if (me == 0) ;//print contents of  rcvbuf
      MPI.Finalize();
   } catch (Exception e) {System.out.println(e);};
}
```

Figure 1. A new communicator is created based on a group created with a group difference operation and is used in a global sum operation.

jmpi is built upon the JPVM system [11], which is a Java-based implementation of PVM [12]. JPVM provides most of the functionality required to set up and communicate in a networked environment. JPVM offers some features not found in standard PVM such as thread safety, multiple communication end-points per task, and default-case direct message routing [13]. JPVM's implementation is more typical of MPI implementations, rather than the existing PVM architectures. For example, JPVM's message-passing implementation is based on communication over TCP sockets, whereas one would expect a UDP-based task-to-task communication to be supported in the PVM environment. Furthermore, instead of PVM's typical daemon-to-daemon routed communications, JPVM uses direct task-to-task message delivery.

4. Java Binding and Implementation

In this section, the way we implemented required functionality of MPI on top of the JPVM communication layer is presented. The following points should be noted in the MPI Java binding.

Java's exception mechanism is used to report errors that occur in MPI routines in contrast to returning integer status codes in MPI's other language bindings.

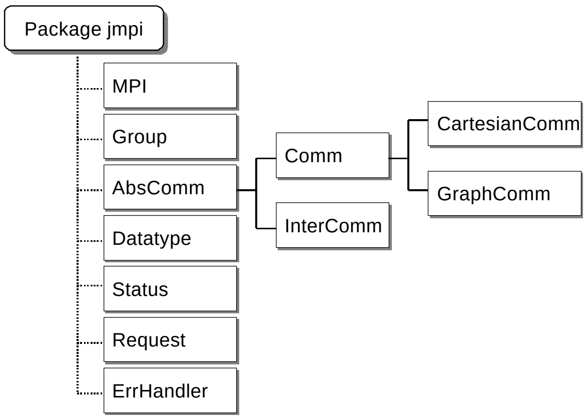

Figure 2. The *jmpi* package class hierarchy. (Adopted from [10])

Java automatic garbage collection facility removes the need for destructors. Basic MPI datatypes include byte, char, short, int, long, float, double and boolean primitive Java datatypes. Java eliminates the address concept and related issues found in other language APIs.

In order to be able to employ an argument as OUT or as INOUT in an MPI function, it must be either an object or an array. Arguments of primitive datatypes can not be used as reference parameters in Java. Furthermore, Java also requires those arguments to be initialized before being sent to the functions.

4.1. MPI Class Hierarchy

MPI specification describes various opaque objects that are accessed via handles. MPI procedures that operate on opaque objects are passed handle arguments to access these objects. This object-oriented methodology used in the development of the specification made it relatively easy to determine the major classes and their interactions with each other.

As shown in Figure 2, apart from the MPI and AbsComm classes, all other classes used in the *jmpi* implementation correspond one to one with MPI opaque objects. The static MPI class contains all MPI named constants, global variables and global static methods such as MPI_Init and MPI_Finalize. The AbsComm class and its derivatives form the basis of interprocess communication.

4.2. Point-to-Point Communication in MPI

A process involved in a communication operation is identified by a static process group and rank, or the order, within that group, whereas messages are considered to be labeled by communication context and message tag within that context (Figure 3.)

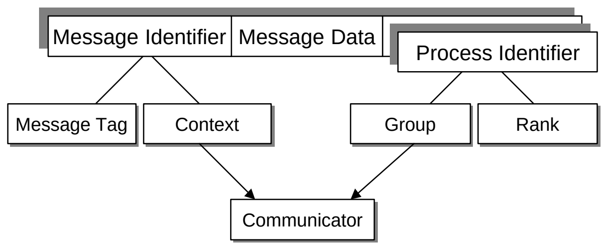

Figure 3. Parts of an MPI message.

The syntax of point-to-point primitives is the same for both inter- and intra-communication as follows:

```
public void Send(
  Object buf, int offset, int count,
  Datatype datatype, int dest, int tag);
public void Recv(
  Object buf, int offset, int count,
  Datatype datatype, int source,int tag);
```

The user-constructed process groups specify a subset of processes in the parallel machine that may participate in communication operations. *Contexts* are simply system-assigned (managed) tags that partition the communication space into non-interfering domains and therefore help to ensure that messages in one domain do not interfere with messages in another.

The context and group are specified by means of a *communicator* object in the argument list of the *jmpi* send and receive routines. *jmpi* upgraded the receipt selectivity of certain JPVM primitives and extended the JPVM's message envelope to include the sender's or receiver's rank, message tag, and a communicator.

The *jmpi* supports both blocking and non-blocking sends and receives. The messages to be sent and the messages that arrive from other processes are buffered in the JPVM layer. JPVM layer also allows sending or receiving multiple messages concurrently by using multiple threads. The threads' access to common communication buffers is restricted through the use of synchronous methods, and thread safety is ensured.

Message buffering decouples the send and receive operations. A blocking send operation is local and it can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, message buffering can be expensive, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering.

4.3. Communicator Object Classes

AbsComm class is an abstract class that contains point-to-point message-passing, attribute caching and other access methods' declarations that are common to both intra- and inter-communicators.

The specification states that the communicator will provide either intra- or inter-communication, but not both. Collective communication and application topology functions do not apply to inter-communicators. The Comm class contains collective communication routines and is further extended as GroupComm and CartesianComm classes that implement topology manipulation.

Comm_world is an instant of the Comm class. It is a global object defined in MPI class and is initialized in the MPI.Init() routine.

Communicators are represented in each process by a tuple consisting of group information, the rank of the process in this group, and context information. Intracommunicators have two different contexts, one for point-to-point communication, and the other for collective communication. In inter-communicators a send- and a receive-context is used instead.

An intra-communicator keeps track of the local group which the process that initiates the communication operation belongs to. Inter-communication is a point-to-point communication between processes in different groups therefore in addition to the local group information, InterComm class objects also contain information about the remote group containing the target process.

We implemented two types of communicator management methods: Local communicator accessor methods that return an integer result and communicator constructors that return new communicator objects. Constructors need to be invoked by all processes in the group associated with the communicator.

4.4. Assignment of Unique Contexts

A static method in the MPI class is responsible for assigning increasing contexts values to each process. Comm_world is assigned a zero context value upon creation. The communicators generated afterwards should share a unique context. This is ensured by broadcasting the context of group leader, the member with the minimum rank, to all other members of the same group. Construction of inter-communicators from two intra-communicators is a more elaborate operation that requires two such collective operations.

Communication contexts are carried as part of the 32-bit tag fields in the regular JPVM message envelope: lower half of the tag is used for the MPI context information, and the upper half carries the regular MPI tag. The leftmost bit of the upper half is reserved for wildcard send and receives but remaining 15 bits are sufficient to support the required MPI tag values up to 32767 specified as MPI_TAG_UB [14].

4.5. Process Groups: The Group Class

A *group* is represented in *jmpi* an ordered collection of JPVM process, or task, identifiers. Each process is identified in its group by its rank. Group operations are all local and all these operations were implemented as public methods of the Group class. These include:

1. Group accessor methods that return information about a given group's properties.
2. Group constructors that include various set operations to form new groups from existing ones.

4.6. Collective Communication

MPI defines an extensive set of collective communication operations, which require invocation of all processors within an intra-communicator for the collective operation to complete. These operations were implemented in layers as shown in Figure 4. For example, MPI Barrier function was implemented using a global all reduce function. Operations in layer 3 use binary-tree based fan-in and fan-out algorithms. This implementation is efficient and scales well when number of processors in the group is large. User-defined operations have not been implemented yet.

4.7. Pack/UnPack and Derived Datatypes

The use of basic MPI datatypes in communication routines involves only contiguous buffers containing a sequence of elements of the same type. If the user wants to send noncontiguous data or to pass messages that contain values with different data types, s/he has to use explicit pack and unpack routines. This could result in double buffering in our system. Derived datatypes, allow one, in most cases, to avoid this inefficiency. The user specifies the type and layout of the data to be sent or received, and the communication library functions access a noncontiguous buffer in place.

MPI provides functions for building contiguous, vector, indexed, and struct datatypes recursively. In the innermost level there is always a base type that corresponds to a Java primitive data type.

Completely ignoring data types as suggested in [10] may not be a good idea due to the reasons mentioned above. The lack of real address concept in Java makes the address and extent functions and lower-bound, upper-bound markers, absolute address specifiers obsolete. We omitted MPI Type_Struct since structures are not supported in Java and Type_Hindexed since displacements should be given in units of the base type as in Type_Indexed, not as bytes.

Derived datatype constructors are implemented as public members of the Datatype class and are called by sending a message to the base type. We restrict the use

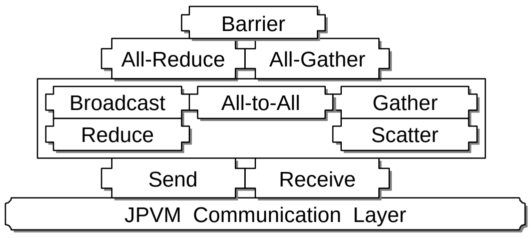

Figure 4. Layered implementation of collective communication primitives.

of derived datatypes only to when send/receive buffer element is a one-dimensional homogenous array of primitive datatypes. Future work will include exploiting the Reflection class and object serialization to be able to support arbitrary data buffers.

4.8. Attribute Caching

A key generator and a random-access object store implemented as an dynamically resizable array of objects helps to attach arbitrary pieces of information to communicators. The user has to use explicit cast/uncast operations to convert an attribute of basic type to the corresponding object type and vice versa.

4.9. Virtual Topologies

Cartesian and Graph topologies were implemented as subclasses of the Comm class, and topology information is kept as private data members of the classes. CartComm stores the number of dimensions, number of processes per coordinate direction, periodicity information and the processor's own position in grid. GraphComm keeps the number of nodes and edges, and two arrays describing node degrees, and graph edges, correspondingly.

Objects of type CartComm and GraphComm need to be created from regular intra-communicators using topology constructors, which are collective operations in that a new context needs to be determined. All other topology functions, but MPI_Cart_sub, are local functions that do not need any communication.

4.10. MPI Environment Management

MPI environmental inquiry functions access the set of constant attributes cached in the MPI Comm_world during the initialization. These functions, timers and startup functions were implemented as static methods of the MPI class.

The *jmpi* system is initially configured to have one active daemon on each host before starting any computations. The MPI_Init() primitive starts processes on each host an active daemon exist and initializes MPI Comm_world. This object represents all processes available at start-up time and each process constructs its own copy of the global system state, determines its group, and rank by accessing this object.

5. Performance Results

Our benchmark tests were performed on a cluster of five SUN UltraSPARC 1 workstations with 167 MHz processors running the Solaris 2.5.1 operating system connected with a 10 Mb/s Ethernet. The Java codes were compiled using SUN's 1.1.4 Java Development Kit and executed on SUN's Java Virtual Machine. We have used PVM version3.4.Beta4 and JPVM version 0.1 in performing the benchmark tests.

We performed a Ping-Pong test between a pair of connected machines over the local area network. Figures 5 and 6 illustrate the round-trip times and effective bandwidths obtained from our tests using PVM, JPVM, and *jmpi*, respectively. The large latencies associated with Java-based implementations significantly reduce the effective bandwidth for small messages as seen in Table 1.

We observe that an important portion of the available bandwidth is lost in both JPVM and *jmpi*. This is mainly due to the slow interpretation speed of Java programs, and partly the need for dynamic object creation costs. *jmpi* incurs a 10-15% more overhead as compared to JPVM due to additional wrapper layer that reformats function arguments for JPVM routines. The effect of this type of implementation details becomes less significant as the message size increases.

6. Acknowledgments

I would like to thank Professor Geoffrey C. Fox for his encouragement of this work and John Cornelius for proofreading this manuscript. Additional material on this work can be found at http://www.baskent.edu.tr/~kdincer/ or http://www.npac.syr.edu/users/kdincer/.

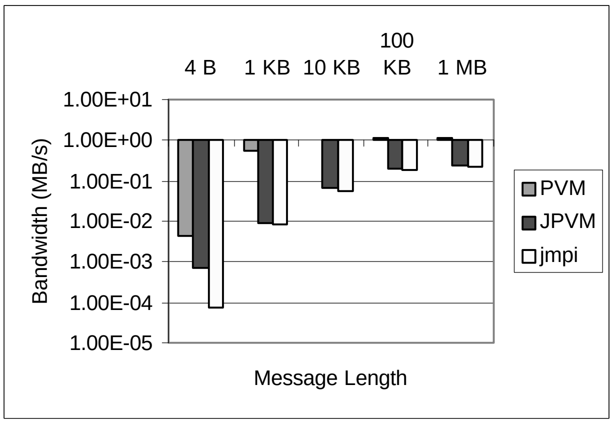

Figure 5. Message size vs. bandwidth.

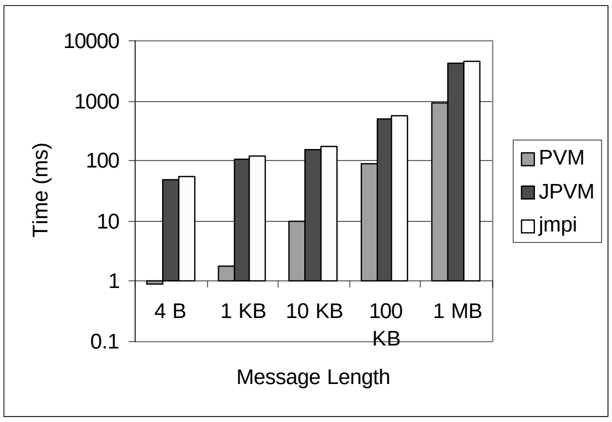

Figure 6. Message size vs. round-trip time.

	PVM	JPVM	*jmpi*
Start-up Latency (ms)	0.92	51.84	58.43
Asymp.Bandwidth(MB/s)	0.76	0.10	0.09
Time per Byte (μs)	0.435	1.98	2.176

Table 1. Latency, asymptotic bandwidth, and average send-time per byte.

7. References

[1] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, 1995.
[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard", TR, Argonne N. Lab, 1996.
[3] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI", TR,Ohio Supercomp.C, 1994.
[4] F-C Cheng, "Unifying the MPI and PVM 3 Systems," TR, Dept. of Comp.Sci., Mississippi State Univ., May 1994.
[5] S. Mintchev, "Writing Programs in JavaMPI", TR MAN-CSPE-02, Univ. of Westminster, London, UK, 1997.
[6] M. Baker, et al., "mpiJava: A Java interface to MPI," 1st UK Workshop on Java for HPCN, 1998.
[7] K.Dincer, "*jmpi* and a Performance Instrum. Analysis and Vis. Tool for *jmpi*," 1st UK Ws on Java for HPCN, 1998.
[8] G. Crawford III, Y. Dandass, and A. Skjellum, "The JMPI Commercial Message Passing Environment and Specification," TR, MPI Software Tech., Inc., 1998.
[9] "Distributed Object Group Metacomputing Architecture, System Overview, " at http://zodiac.cs.byu.edu/DOGMA/ .
[10] B.Carpenter, V.Getov, G. Judd, T. Skjellum, G. Fox, "MPI for Java,", TR JGF-TR-03, Java Grande Forum, 1998.
[11] D. Thurman, "JavaPVM," available from http://www.isye.gatech.edu/chmsr/JavaPVM/.
[12] A. Geist, et al., PVM: Parallel Virtual Machine, Cambridge, Mass., MIT Press. 1994.
[13] A.J.Ferrari, "JPVM: Network Parallel Computing in Java", in Proc. of ACM 1998 Ws. on Java for HPCN, 1998.
[14] Message Passing Interface Forum, "MPI 1.1," 1995.