JMPI: Implementing the Message Passing Standard in Java

Steven Morin, Israel Koren, C. Mani Krishna
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
E-Mail: {srmorin,koren,krishna}@ecs.umass.edu

Abstract

The Message Passing Interface (MPI) standard provides a uniform Application Programmers Interface (API) that abstracts the underlying hardware from the parallel applications. Recent research efforts have extended the MPI standard to Java either through wrapper implementations or as subsets of larger parallel infrastructures. In this paper, we describe JMPI, a reference implementation of MPI developed at the Architecture and Real-Time Laboratory at the University of Massachusetts Amherst. In this implementation, we explore using Java's Remote Method Invocation (RMI), Object Serialization and Introspection technologies for message passing. We discuss the architecture of our implementation, adherence to emerging standards for MPI bindings for Java, our performance results, and future research direction.

Keywords: *Message Passing Interface, Java, Remote Method Invocation, Object Serialization*

1. Introduction

Java Message Passing Interface (JMPI) [9] is an experimental implementation of the Message Passing Interface (MPI) standard for distributed memory multiprocessing developed at the Architecture and Real-Time Laboratory at the University of Massachusetts. The purpose of JMPI is to build an implementation of the MPI standard [7] completely written in Java, utilizing its native language capabilities, such as Remote Method Invocation (RMI), Object Serialization, and Introspection. JMPI exploits Java's portability, providing an MPI implementation on any platform where the Java Virtual Machine (JVM) is available. We measure the results of our implementation in terms of portability and performance. Furthermore, we have integrated the KaRMI [10] toolkit into JMPI and provide here a performance comparison between Sun's RMI and KaRMI. Finally, we draw conclusions from our results and present directions for further research.

2. Previous Research

Previous research efforts to port MPI to Java have focused on two types of implementations: wrapper based or as subsets of larger parallel and distributed infrastructures. A wrapper based implementation consists of three components: a wrapper method for each Java MPI binding, the Java Native Interface [5], and an existing implementation of MPI. Each wrapper method is composed of code written in Java, and corresponding code written in C. The Java Native Interface provides a conduit between Java and C. The wrapper code in C makes a call to the appropriate MPI binding in an existing MPI implementation, such as MPICH or LAM. Two wrapper implementations have been developed: JavaMPI from the University of Westminster [8] and mpiJava from the Northeast Parallel Architectures Centre at Syracuse University [1].

The Distributed Object Group Metacomputing Architecture (DOGMA) from Brigham Young University [6] provides a framework for the development and execution of parallel applications on clusters of workstations and supercomputers. The DOGMA runtime environment, supports dynamic reconfiguration and remote management, decentralized dynamic class loading, fault detection and isolation, and web browser and screen-saver node participation. The Application Programmers Interface (API) is the topmost layer of the DOGMA environment. Two API specifications are provided: Distributed Object Groups (DOG) and Message Passing Interface. DOG allows applications to work with large groups of objects. The DOG API allows applications to perform asynchronous method invocations on large groups of objects, to dynamically partition and reassemble data, and to define group members for peer communication. The MPI API is provided as a subset to the Distributed Object Group interface. The MPI bindings

are based on the C++ bindings as specified in the MPI-1.1 specification as much as possible.

3. Java Message Passing Interface (JMPI)

JMPI is written entirely in Java and runs on any host that the Java Virtual Machine is supported on. In contrast to wrapper- based implementations, JMPI eliminates all portability problems associated with the Java Native Interface and interfacing with an existing port of MPI. Our implementation has been tested on the Solaris, Linux and Windows platforms.

JMPI implements Message Passing with Java's Remote Method Invocation (RMI) and Object Serialization [4]. RMI is a native construct in Java that allows a method running on the local host to invoke a method on a remote host. One benefit of using RMI is that a call to a remote method has the same semantics as a call to a local method. As a result, objects can be passed as parameters to and returned as results of remote method invocations. Object Serialization is the process of encoding a Java Object into a stream of bytes for transmission across the network. On the remote host, the object is reconstructed by deserializing the object from the byte array stream. If the object contains references to other objects, the complete graph is serialized.

JMPI has three distinct layers: the Message Passing Interface API, the Communications Layer, and the Java Virtual Machine. The Message Passing Interface API, the core set of MPI functions that MPI applications are written to, is based on the proposed set of Java bindings by the Northeast Parallel Architecture Centre (NPAC) at Syracuse University. The Communications Layer contains a core communication primitives used to implement the Message Passing Interface API. Finally, the Java Virtual Machine (JVM) compiles and executes Java Bytecode.

3.1. MPI Bindings for Java

The MPI Forum has not officially reconvened since the MPI-2 standard was released in the summer of 1997[7]. As a result, no official bindings exist for Java. Although no official MPI bindings exist for Java, we choose to follow the bindings proposed by the Northeast Parallel Architectures Centre[1, 2, 3]. First of all, the proposed JAVA bindings are derived from and closely follow the official C++ bindings supported by the MPI Forum. Secondly, we felt it was important to maintain compatibility between our implementation of MPI and other emerging implementations based on these proposed bindings. By following the MPI binding proposal, no source code modification or recompilation is required to switch between mpiJava and JMPI.

The Java Virtual Machine does not support direct access to main memory, nor does it support the concept of a global linear address space[4]. Consequently, JMPI provides supports for multi-dimensional arrays as if the arrays were laid out sequentially in memory as in C. Support for multi-dimensional arrays of any dimension is accomplished through the use of Java's Introspection[4]. Introspection allows the Java application to dynamically determine the type of any Object. When a message buffer is passed, JMPI uses Introspection to determine the number of dimensions and the number of elements in each dimension of the array. After the number of elements in each dimension is calculated, the reference to the array containing the first element in the message buffer as specified in the offset parameter is determined. The algorithm sequentially steps through each reference in the multi-dimensional array until the number of elements as specified in the MPI application is exhausted or the last element of the array is reached.

Error handling in the Java Bindings differs significantly from the C and Fortran bindings. For example, most MPI functions in the C API return a single integer value to indicate the success or failure of the call[7]. In contrast, Java throws exceptions when an error occurs[4]. For example, when a program tries to index an array outside of its declared bounds, the java.lang.ArrayIndexOutOfBoundsException exception is thrown. The program can choose to catch the exception or propagate the exception to the JVM.

3.2. Message Passing in JMPI

The Communications Layer within JMPI has three primary responsibilities: virtual machine initialization, routing messages between processes, and providing a core set of communications primitives to the Message Passing Interface API. The Communications Layer is multi-threaded and runs in a separate thread from the MPI process. This allows for the implementation of non-blocking and synchronous communication primitives in the MPI API. Messages are transmitted directly to their destination via Remote Method Invocation.

The Java registry, a key component of Remote Method Invocation, serves as an object manager and naming service for the Communications Layer. During virtual machine initialization, each process instantiates a registry on the local machine, and registers an instance of the Communications Layer with it. Each registered instance of the Communication Layer is addressable via a Uniform Resource Locator (URL) in the form:

rmi://hostname:*portno*/Comm*x*

where *portno* represents the port number that the registry accepts connections on, and *x* represents the MPI rank of the local process. The last step in virtual machine initialization is to perform a barrier with all other processes. The purpose of this barrier is two-fold: ensure that all processes have initialized properly, and to distribute a table with the addresses of all Communication Layers in the virtual machine.

Messages are passed between source and destination as a parameter to a remote method invocation invoked by the source process. The Message is a serializable object that consists of fields to represent the source rank, the destination rank, and message tag. In addition, the Message incorporates a Datatype object to indicate the type of data being sent in the message, as well as the actual data. The source method inserts the message object via remote method invocation into the destination processes message queue, and then notifies all processes blocked on the destination that a new message has arrived. For synchronous mode sends, where completion of the call indicates the receiver has progressed to the matching receive, the source blocks in the remote method invocation until the matching receive has started. Java's wait and notify methods are used to implement synchronous mode sends. For all other communication modes, the remote method returns after the message has been inserted into the local message queue.

The destination process receives messages by invoking a local method in the Communications Layer. This method returns the first message received, or the message that matches the ranks or tags specified. The message queue is implemented with a Java Vector Class for two reasons: synchronized access allows more than one thread to insert or remove messages at a time, and Vectors perform significantly faster than a hand-coded linked list implementation. In addition, Vectors also support out-of-order removal of messages, which is required to implement several MPI receive calls.

The Communications Layer provides communication primitives from which all other MPI bindings are implemented. The blocking point-to-point primitives have been discussed in the previous paragraphs. The non-blocking versions of the point-to-point primitives are implemented by creating a separate thread of execution, which simply calls the respective blocking version of the communication primitive. Collective communication primitives, such as broadcast, are built using point-to-point primitives. A separate message queue is used to buffer incoming collective operations to satisfy the MPI specification. Two probe functions allow the message queue to be searched for matching messages without retrieving them. Finally, a barrier primitive allows all processes within a communicator to synchronize their execution.

3.3. KaRMI

KaRMI is a drop-in replacement for Sun's Object Serialization and Remote Method Invocation components. KaRMI improves the performance of both Object Serialization and RMI by optimizing the code for use with clusters. For Object Serialization, KaRMI assumes that all nodes in the cluster have access to the same Bytecode needed to reconstruct the object from a persistent store. Furthermore, KaRMI replaces RMI's internal buffering routines with ones optimized to read the number of bytes needed to reconstruct objects with one read call. For Remote Method Invocation, KaRMI eliminates unneccessary method calls in a Remote Method Invocation. KaRMI also optimizes the use of data structures; hash tables are replaced with arrays or removed. KaRMI limits the use of native calls to device drivers. However, KaRMI adds complexity to the configuration setup before one can launch an MPI application. We consider to integrate KaRMI directly into JMPI and remove the need for the additional configuration files[10].

4. Validation

During the spring of 1993, IBM released a suite of C applications that test the functionality of an MPI implementation. The test suite is organized into six categories: point-to-point primitives, collective primitives, context, environment and topology. NPAC has ported the IBM test suite to Java and distributes it with mpiJava.

The NPAC test suite was used during the development of JMPI to validate correctness of the implementation as defined in the proposed specification. Each validation test was run against three test environments: homogenous environment on a single-processor machine, a homogenous environment on a dual-processor machine, and a heterogeneous environment of two machines connected via Fast Ethernet. The single processor workstation is a Pentium III running RedHat 7.2 and Sun's JDK version 1.2.2_008. The dual-processor is equipped with two Pentium-III processors and runs Intel Solaris 8 and Sun's JDK, version 1.2.2_005. The core of the network consists of an eight-port NetGear fast Ethernet switch. Out of the possible ninety-nine tests, seventy-eight tests passed and twenty-one tests validated functionality not yet implemented.

5. Performance

One of the most important characteristics of a parallel programming environment is performance. Two applications are used to compare performance between JMPI and mpiJava: PingPong and Mandelbrot. PingPong, distributed with NPAC's mpiJava, measures the round-trip latency of a message transmitted between two MPI processes. Mandelbrot computes a Mandelbrot fractal by distributing work to slave processes. For each application, we measure the performance of mpiJava with the MPICH backend, JMPI with Sun's RMI, and JMPI with KaRMI[10]. The performance of these three environments are measured by running all processes on a single processor Linux server, and by distributing all processes among the single processor Linux server, and the dual-processor Solaris server. Although we ran performance tests for the dual-processor Solaris server, we omit these test results due to MPICH's shared memory transport on this platform.

A comparision of performance between JMPI with mpiJava shows mixed results: applications that use small message sizes clearly perform better with mpiJava than in JMPI. This is a direct result of the additional overhead caused by Object Serialization and the added latency of a Remote Method Invocation. This overhead becomes less significant as the message size increases. For all message sizes, JMPI/KaRMI performs significantly better than JMPI/RMI. However, there were instances of some performance tests terminating abnormally in mpiJava with "mismatched error messages" from the MPICH backend. While the performance of JMPI lags behind mpiJava, applications are stabler in JMPI.

5.1. PingPong Performance

PingPong measures the round-trip delay of a message between two MPI processes and calculates the throughput available. The message consists of a one-dimensional array of bytes where the number of elements in the array varies as a function of 2x, where x varies from 0 to 20. All elements in the array are initialized to a value as defined by the equation

$$A[i] = (byte)'0'; //(double)1./(i+1); \quad (1)$$

before the message is sent. Each message is sent a total of 64 times. The rank 0 process records the round-trip time of each message and produces an average round-trip time and average throughput, measured in Mb/sec, for each array.

Figure 1 shows the performance of mpiJava, JMPI/RMI, and JMPI/KaRMI in the homogenous single processor environment with the PingPong application. For small message sizes, both JMPI/RMI and JMPI/KaRMI fare significantly worse than mpiJava. For large message sizes, the performance of mpiJava drops as a result of a change in protocol in the underlying MPICH implementation. The MPICH implementation on Linux uses sockets for message transport between processes in a single processor system. For small message sizes, the eager protocol assumes the remote process has sufficient memory to buffer messages. For large message sizes, MPICH transfers small sections of the message and waits for an acknowledgement. This results in a bottleneck for message sizes greater than 131,072 elements.

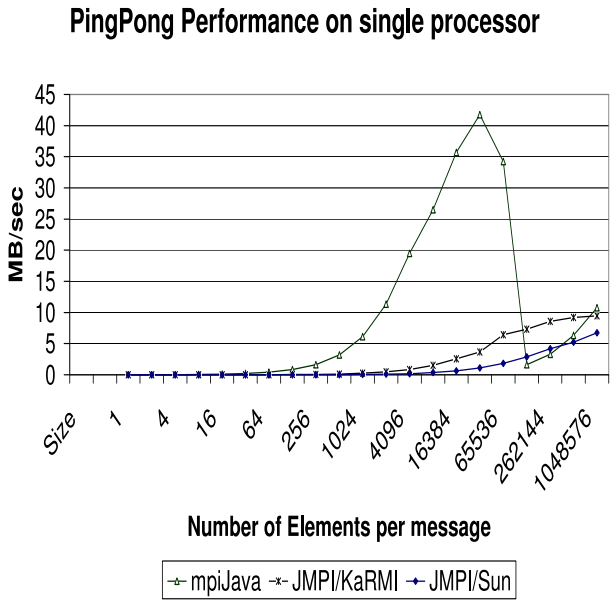

Figure 1. PingPong Single Processor results.

Figure 2 shows the performance of the PingPong application over Fast Ethernet. The poor performance for small message sizes is attributed to the overhead of Object Serialization and Remote Method Invocation. Object Serialization flattens objects into byte arrays for transmission across the network. For Sun's RMI implementation, the process of flattening increases message size by a fixed constant of 325 bytes. For small message sizes, object serialization adds a significant amount of overhead. To measure the latency added by Sun's Remote Method Invocation, we coded a simple remote method that performs no computation other than returning control of execution back to the caller. This method was invoked for 1,000 iterations and the average added latency of a remote method over an Ethernet network was 0.94 ms.

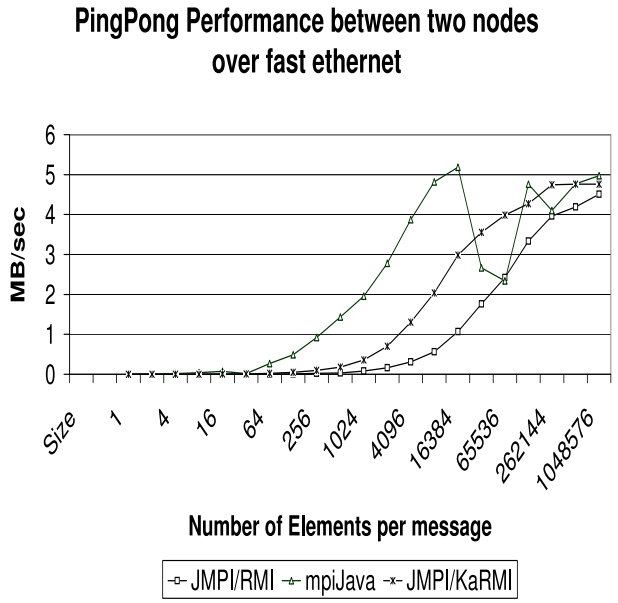

Figure 2. PingPong over Fast Ethernet results.

5.2. Mandelbrot Performance

A Mandelbrot generator is used to compare the performance of JMPI versus mpiJava. The Mandelbrot Curve algorithm was coded using the master/slave paradigm. The master process is responsible for assigning work to slave processes and assembling the results. The master process iteratively assigns each slave a 400 square pixel area of a 512 x 512 grid to each process. The slave calculates the Mandelbrot value for each point over this region, iterating up to 1,000 times for each pixel. After the iteration completes, the slave sends a response to the master containing the coordinates of the work area and the fractal values for each point. The master process copies these values into their respective positions in the master pixel array. Processing completes when the Mandelbrot function is computed for the entire 512 x 512 grid. Pixels are colored according to their Mandelbrot value.

The performance of JMPI and mpiJava is determined by measuring the amount of time required to compute the Mandelbrot values for the 512 x 512 pixel grid. For each test case, the elapsed time from three runs is averaged to produce a final result. The performance of JMPI/KaRMI is compared against mpiJava in two test environments: homogenous single processor environment and heterogeneous network environment. The number of slave processes is varied from one to five. Shorter elapsed times indicate better performance.

Figure 3 shows the amount of time elapsed calculating the Mandelbrot fractal in a single processor homogenous environment. The degradation in performance of JMPI/KaRMI as compared to mpiJava is attributed to the small message sized used to transport work assignments and results between the master and slave processes. Furthermore, each additional slave in JMPI/KaRMI adds three to four seconds of additional time needed to compute the fractal, while each additional slave in mpiJava slightly decreases the amount of time required.

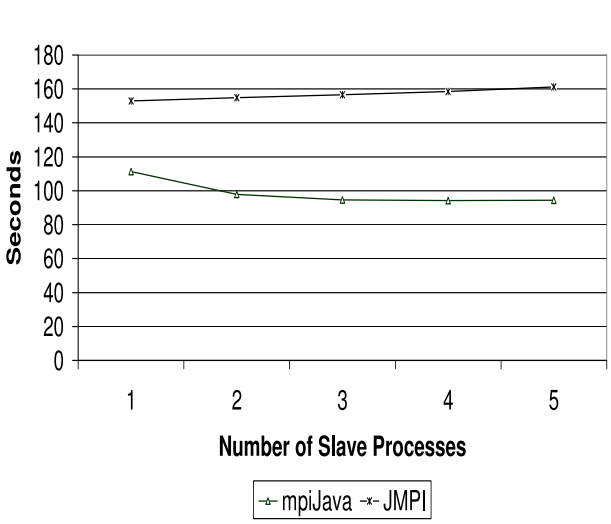

Figure 3. Homogenous Single Processor performance.

Figure 4 shows the amount of time elapsed computing the Mandelbrot fractal with processes distributed over a fast Ethernet network. Although the performance of JMPI/KaRMI is slightly faster than mpiJava with one slave process, the Mandelbrot generator with two or more slave processes completes the fractal in about half the time. Again, the overhead of Object Serialization and Remote Method Invocation affect small message sizes adversely. Although one would expect performance to decrease as new slave processes are added, the master process quickly becomes a bottleneck. Specifically, this bottleneck occurs as a result of copying the Mandelbrot results that are returned from the slave into the main pixel array. The linear slowdown in performance of JMPI is attributed to the overhead of Remote Method Invocation, Object Serialization and the overhead due to small message sizes.

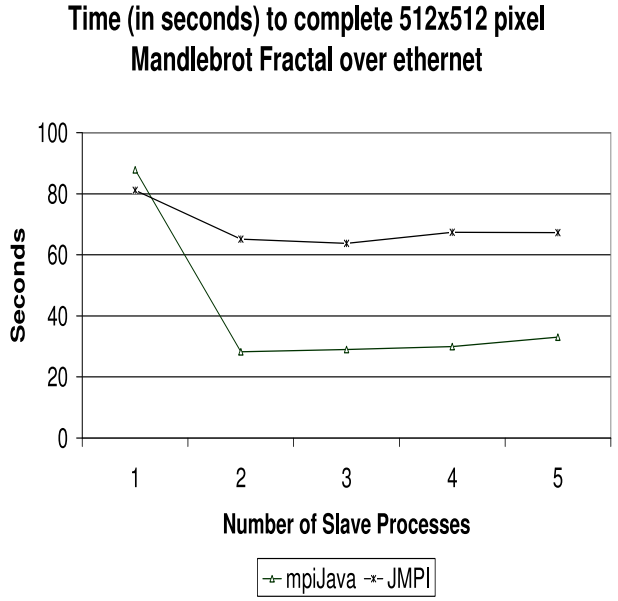

Figure 4. Fast Ethernet performance.

JMPI performance, either with Sun's RMI or KaRMI, suffers as a result of using Remote Method Invocation as the primary message transport between processes. In the multiprocessor environment, RMI is no match for the shared memory transport of MPICH. While RMI eases the implementation of MPI primitives, such as process synchronization, the performance is too slow. However, the JMPI environment proved to be more stable and reliable than mpiJava.

6. Future Research Goals

JMPI is a good starting point for a reference implementation of the Message Passing Interface for Java. We've demonstrated that our implementation is portable to any platform where Java is available, and specifically tested it on Solaris, Linux, and Windows. However, the issue of performance needs to be addressed. Remote Method Invocation (RMI) adds a significant amount of overhead to each message passed. Specifically, RMI added a constant 325 bytes to the size of each message transferred that adds an average of 0.94 ms to message latency.

One future research goal is to reduce the reliance of Remote Method Invocation for interprocess communication, and replace it with Berkeley Sockets. Although RMI would still play a role in virtual machine initialization and process synchronization, message passing between processes would be over Berkeley Sockets. In the present implementation, one Java Virtual Machine is instantiated for each process in the MPI application. For an MPI application with multiple processes on a single machine, each additional JVM represents a significant amount of overhead. A future research goal is to implement multiple processes of an MPI application on the same machine as separate threads in one JVM. This would reduce the amount of overhead caused by additional JVMs, and allow these processes to communicate with one another through shared memory. Finally, better reuse of threads and buffers would minimize the amount of time used to create these objects.

7. Acknowledgment

This work was partially supported by an AASERT grant number DAAH04-96-1-0206 from DARPA.

References

[1] Carpentar, B., Fox, G., Ko, S. H., and Lim, S. "mpiJava 1.2 API Specification," November, 1999.

[2] Carpentar, B., Fox, G., Ko, S. H., Li, X., and Zhang, G. "A Draft Binding for MPI," September, 1998.

[3] Carpentar, B., Getov, V., Judd, G., Skjellum, A., and Fox, G. "MPI for Java: Position Document and Draft API Specification," *Java Grande Forum*, 1998.

[4] Flanagan, D. Java in a Nutshell, second edition. Cambridge: O'Reilly & Associates, 1998.

[5] Gordon, R. Essential JNI Java Native Interface. Upper Saddle River, NJ: Prentice Hall PTR, 1998.

[6] Judd, G., Clement, M. and Snell, Q. "DOGMA: distributed object group metacomputing architecture," *Concurrency: Pract. Exp.*, **10**(11-13),(1998)

[7] Message Passing Interface Forum. Extensions to the Message Passing Interface. August, 1997.

[8] Mintchev, S. "Writing Programs in JavaMPI," London, England: University of Westminster, October, 1997.

[9] Morin, S. "JMPI: Implementing the Message Passing Standard Interface in Java," Master's thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, September, 2000.

[10] Nester, C., Philippsen, M., Haumacher, B. "A More Efficient RMI for Java," *JavaGrande99*, pgs. 152-159.