Towards Java-based HPC using the MVAPICH2 Library: Early Experiences

Kinan Al-Attar*, Aamir Shafi*, Hari Subramoni† and Dhabaleswar K. Panda†
Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio
*Email: {alattar.2, shafi.16}@osu.edu
†Email: {subramoni, panda}@cse.ohio-state.edu

2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-9747-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPSW55747.2022.00091

***Abstract*—There has been sporadic interest in using Java for High Performance Computing (HPC) in the past. These earlier efforts have resulted in several Java Message Passing Interface (MPI) [1] libraries including mpiJava [2], FastMPJ [3], MPJ Express [4], and Java Open MPI [5]. In this paper, we present our efforts in designing and implementing Java bindings for the MVAPICH2 [6] library. The MVAPICH2 Java bindings (MVAPICH2-J) follow the same API as the Java Open MPI library. MVAPICH2-J also provides support for communicating *direct* New I/O (NIO) `ByteBuffer`s and Java arrays. Direct `ByteBuffer`s reside outside JVM heaps and are not subject to the garbage collection. The library implements and utilizes a buffering layer to explicitly manage memory to avoid creating buffers every time a Java array message is communicated. In order to evaluate the performance of MVAPICH2-J and other Java MPI libraries, we also designed and implemented OMB-J that is a Java extension to the popular OSU Micro-Benchmarks suite (OMB) [7]. OMB-J currently supports a range of benchmarks for evaluating point-to-point and collective communication primitives. We also added support for communicating direct `ByteBuffer`s and Java arrays. Our evaluations reveal that at the OMB-J level, `ByteBuffer`s are superior in performance due to the elimination of extra copying between the Java and the Java Native Interface (JNI) layer. MVAPICH2-J achieves similar performance to Java Open MPI for `ByteBuffer`s in point-to-point communication primitives that is evaluated using latency and bandwidth benchmarks. For Java arrays, there is a slight overhead for MVAPICH2-J due to the use of the buffering layer. For the collective communication benchmarks, we observe good performance for MVAPICH2-J. Where, MVAPICH2-J fairs better than Java Open MPI with `ByteBuffer`s by a factor of 6.2 and 2.76 for broadcast and allreduce, respectively, on average for all messages sizes. And, using Java arrays, $2.2\times$ and $1.62\times$ on average for broadcast and allreduce, respectively. The collective communication performance is dictated by the performance of the respective native MPI libraries.**

***Index Terms*—Java, MPI, MVAPICH2, OMB, HPC**

I. Introduction and Motivation

The Message Passing Interface (MPI) standard [1] continues to dominate the landscape of High Performance Computing (HPC) applications as the community is edging closer towards exascale computing systems. MPI currently provides support for C and Fortran programming languages. However, there is also interest in using MPI compliant libraries from higher-level programming languages like Java and Python. Java has been powering most of the Big Data computing stacks including Apache Hadoop and Apache Spark. Similarly, Python is at fore-front of the recent AI uptake and is powering many popular Deep Learning frameworks including PyTorch and TensorFlow.

Historically, there has been interest in using MPI to scale parallel and distributed Java applications on HPC systems. There are good reasons for popularity of Java, which include portability, widespread adoption in the software industry and Big Data community, and advanced features like garbage collection. The interest in Java led to the creation of several Java MPI libraries including Open MPI Java bindings [5] (called Open MPI-J hereafter), mpiJava [2], MPJ Express [4], and FastMPJ [3]. Some of the libraries—like MPJ Express and FastMPJ—provided support for the MPI standard in pure Java while providing communication devices for high-speed networks like InfiniBand and others using Java Native Interface (JNI). Note that JNI allows Java programs to invoke functions and methods written in other languages including C. This approach, of implementing the MPI standard, is tedious and requires substantial development effort. On the other hand, the approach pioneered by mpiJava and adopted by Open MPI-J, is to keep the Java layer as minimal as possible and use JNI to invoke MPI methods implemented by "native" production-quality MPI libraries. This approach allows easier development and maintenance as well as high-performance for Java MPI libraries. Currently Open MPI-J and FastMPJ are the two well-maintained Java MPI libraries in the community. The open-source version of FastMPJ only supports pure Java communication devices and hence not used in the comparative evaluation in this paper.

MVAPICH2 [6] is a production quality MPI library with support for high-speed RDMA networks like InfiniBand. This paper is an effort to produce initial prototype Java bindings for the MVAPICH2 library. Currently the Java bindings in MVAPICH2 are provided by a limited sub-set of the MPI standard including i) blocking/non-blocking point-to-point functions, ii) blocking collective functions, and iii) blocking vectored collective functions. In addition, some supporting communicator and group management functions are also implemented.

In the past, the Java MPI libraries have implemented a variety of APIs for application developers. These include the mpiJava 1.2 API, the MPJ API, and the Open MPI Java bindings API. MPJ Express and mpiJava libraries implement the mpiJava 1.2 API. FastMPJ supports both the mpiJava 1.2 and the MPJ API. The Java Grande Forum——formed in late

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00091

90s——came up with an API called mpiJava 1.2. The MPJ API followed that and is a minor upgrade to the mpiJava 1.2 API. The MPJ API is more inline with Java coding conventions. However, Open MPI-J adopted a custom API that is an extension of the MPJ API. The most important updates here include supporting communication to/from Java New I/O (NIO) `ByteBuffers` in addition to Java arrays. Also, the communication primitives in the Open MPI Java bindings do not provide support for communicating a sub-set of the `ByteBuffer` or array argument. This was possible in the mpiJava 1.2 and MPJ API through an offset argument to communication primitives. For the Java MVAPICH2 bindings, we have adopted the Open MPI Java API in order to facilitate end users.

The MVAPICH2 Java bindings are also equipped with a number of test-cases adopted from the MPJ Express library. In addition, we have also produced a Java version of the OSU Micro-Benchmark (OMB) suite [7]—we will refer to this as OMB-J. These are popular MPI benchmarks to evaluate performance of communication libraries using a variety of point-to-point and collective benchmarks. OMB-J currently has support for point-to-point, blocking collectives, and vectored blocking collective operations. OMB-J supports both the `ByteBuffer` and arrays API.

One of the main challenges in implementing an efficient Java MPI library is to minimize the overhead incurred by copying data from Java to C. This copy is essential because all high-speed networks, like InfiniBand, provide communication libraries in the C language. This copy is also needed for designs when the Java MPI library interfaces with the native MPI library to communicate data. The mpiJava 1.2 and MPJ APIs provided support for communicating data to/from Java arrays of basic datatypes as well as arrays of Java objects. The Java Native Interface (JNI) allows invoking C functions from Java and provides two main ways of copying Java arrays of basic datatypes: 1) Use JNI functions `Get<Type>ArrayElements` and `Release<Type>ArrayElements`—¡`<Type>` refers to all basic datatypes in Java—to retrieve corresponding pointers to Java arrays, and 2) Use JNI utility functions `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical` to obtain pointers to Java arrays in the C code. On modern Java Virtual Machines (JVMs) including OpenJDK and Oracle JDK that do not support "pinning", the first approach incurs a copy from Java to C. The second method—of using the `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical` pair of functions—does not incur data copying overhead. However, this method is not recommended because the JVM halts garbage collection between calls to `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical` functions. This can possibly have a detrimental performance on the application.

The Open MPI Java bindings provided an alternate approach to avoid data copying in Java MPI libraries by modifying the user API. The Java NIO package introduced a new data-structure called `ByteBuffers` in Java. There are two types of `ByteBuffers`: 1) Direct and 2) Indirect or heap. Direct `ByteBuffers`, while costly to create and destroy, do not reside in the Java heap and hence are not subject to garbage collection. Because of this, direct `ByteBuffers`—when passed to JNI methods—do not incur copy and C methods are simply passed pointers to the original allocated memory. These are ideal for applications like Java MPI libraries that need to invoke JNI methods to call native communication or MPI libraries. On the other hand, indirect or heap `ByteBuffers` are allocated on the JVM heap like normal Java objects and hence are subject to garbage collection. As a consequence, when indirect `ByteBuffers` are passed to JNI methods, modern JVMs make a copy of these to avoid stale or invalid pointers. Older APIs for Java MPI libraries—including the mpiJava 1.2 and MPJ API—only supported communicating data to/from arrays of basic Java datatypes and objects. However, the Java Open MPI library updated the API to be able to communicate data to/from direct `ByteBuffers`. This mandates modifications and updates to parallel Java HPC applications.

The MVAPICH2-J library also supports communicating data to/from direct `ByteBuffers`. In order to support communicating of arrays of Java datatypes and arrays, we utilize an internal buffering layer inspired by MPJ Express [4]. When communicating arrays of Java basic datatypes, it is possible to acquire pointer in the native code using `Get<Type>ArrayElements` or `GetPrimitiveArrayCritical`. Our buffering instead maintains a pool of direct `ByteBuffers`. The sender process copies data onto a direct `ByteBuffer` and a pointer to this buffer is retrieved in the JNI method call and used for communication along with the native MVAPICH2 library.

We evaluate and present the performance of point-to-point and collective communication primitives for the MVAPICH2-J library along with Java Open MPI bindings using OMB-J. Our evaluation reveals that using `ByteBuffers` provide better performance compared to Java arrays at the OMB-J level. The point-to-point performance of MVAPICH2, as depicted by latency and bandwidth benchmarks, is comparable to Java Open MPI for the `ByteBuffer` API. There is a slight overhead in the performance while communicating Java arrays using MVAPICH2-J due to the internal buffering layer. This layer is needed to support communicating derived datatypes and Java arrays with non-blocking point-to-point functions. Open MPI-J does not support communicating Java arrays with non-blocking point-to-point functions. As a consequence, it was not possible to calculate bandwidth numbers for Java arrays in Open MPI-J. For the collective communication benchmarks, we present evaluation for broadcast and allreduce primitives. MVAPICH2-J outperforms Open MPI-J for the `ByteBuffer` API by a factor of 6.2 and 2.76 for broadcast and allreduce, respectively, on average for all messages sizes. For Java arrays, we observe 2.2× and 1.62× better performance than Open MPI-J—on average for all message sizes—for broadcast and

allreduce, respectively. The performance advantage in collective benchmarks is mainly due to performance differences in native MPI libraries.

While our OMB-J evaluation concludes that the `ByteBuffer` API performs better than Java arrays. However, OMB-J only measures the communication performance and ignores the cost of copying user data onto `ByteBuffers` compared to Java arrays. To tackle this, we performed an experiment—detailed in Section VI-F—where we not only measure the communication time but also validate the contents of messages. This means that `ByteBuffers` and Java arrays are populated at the sender end and validated at the receiver end. We found that Java arrays perform better than direct `ByteBuffers` in this case. The reason is that it is faster to read/write data from Java arrays compared to `ByteBuffers`.

A. Contributions

This paper makes the following contributions:

1) Design and implementation of MVAPICH2-J, which is a Java binding for the MVAPICH2 library with the design goal to keep the Java layer as minimal as possible
2) MVAPICH2-J provides support for communicating user data to/from Java arrays and direct `ByteBuffers`. Direct `ByteBuffers` provide an option to acquire pointers to their storage in the JNI code making it possible to avoid data copying overhead incurred by Java arrays.
3) In order to evaluate performance of MVAPICH2-J and other Java MPI libraries, we architect and implement a Java version of the popular OSU Micro-Benchmark (OMB) suite named OMB-J. OMB-J currently supports point-to-point primitives (latency, bandwidth, and bi-bandwidth) and vectored and blocking collective communication primitives (latency).
4) The paper conducts extensive benchmark level experiments of MVAPICH2-J against Java Open MPI using OMB-J. These evaluations are done on the TACC's Frontera system. This includes latency and bandwidth comparisons for point-to-point communication primitives and latency comparisons for collectives (broadcast and allreduce). We also quantified the overhead of the Java layer for the buffering layer in our evaluation.
5) The paper also reveals that while using `ByteBuffers` led to better performance at the OMB-J level, this benefit might not translate to application-level benefits. This is due to slower read/write access to `ByteBuffers` compared to Java arrays. This contribution is detailed in Section VI-F.

Rest of the paper is structured as follows. The background of our work is given in Section II. Section III details the design and implementation of our approach. The experimental results are presented in Section VI. The related work is discussed in Section VII, followed by a conclusion in Section VIII.

II. Background

A. Java for HPC

After the emergence of the Java programming language in the late 1990s, there was an interest in using the Java programming language for HPC applications. This led to the emergence of many MPI libraries including mpiJava [2], MPJ Express [4], and FastMPJ [3]. While the adoption of Java has been relatively low for high-performance numerical codes, it has become a widely used language for Big Data computing and analytics. Some of the widely used Big Data frameworks like Apache Spark [8] and Apache Hadoop [9] are written in Java. There are also deep learning efforts written in Java such as DeepLearning4j (DL4J) [10], a suite for running Deep Learning training and inference workloads.

B. `ByteBuffers` and Java Arrays

The Java NIO package introduced the concept of non-blocking I/O to the language. In order to support networking and storage I/O efficiently, the package also introduced new user defined datatypes called `ByteBuffers` in Java. These buffers provide a variety of `put()` and `get()` methods to copy data from Java arrays of all basic datatypes. There are type-specific buffers that include `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, and `ShortBuffer`. The most relevant buffers for our work are `ByteBuffers`. The reason is because there are two types of `ByteBuffers`: 1) direct and 2) non-direct or heap—as shown in Figure 1. Direct `ByteBuffers` are created using the static `allocateDirect()` method. These are costly to create, however, these are not subject to garbage collection as they reside outside the Java heap. These buffers are attractive for Java MPI libraries because it is possible to acquire a pointer to their memory using `GetDirectBufferAddress()` in the JNI C code. On the contrary, non-direct or heap `ByteBuffers` are allocated using the static `allocate()` method. These are normal Java objects that reside in the JVM heap and hence are subject to garbage collection. Similarly, Java arrays are also regular objects that reside in the JVM heap.

C. APIs for Java MPI Libraries

This sub-section provides a review of APIs for Java MPI libraries. Historically, the Java Grande Forum provided a platform for the Java HPC community to produce a widely accepted API. This forum proposed the mpiJava 1.2 and the MPJ API. mpiJava, MPJ Express, and FastMPJ were three Java MPI libraries that adopted these two APIs. A Java version of the popular NAS Parallel Benchmark [11]—named NPB-MPJ [12]—also uses mpiJava 1.2 and MPJ APIs. The MPJ API is a modest upgrade to the mpiJava 1.2 API mainly motivated by adopting Java naming conventions for functions.

However, more recently, the Open MPI library introduced the support for Java bindings. Instead of going with existing APIs, the Java Open MPI library adopted a new API. Main reason was that the Java Open MPI library was MPI 3.0 standard compliant, whereas, the older Java MPI APIs, like

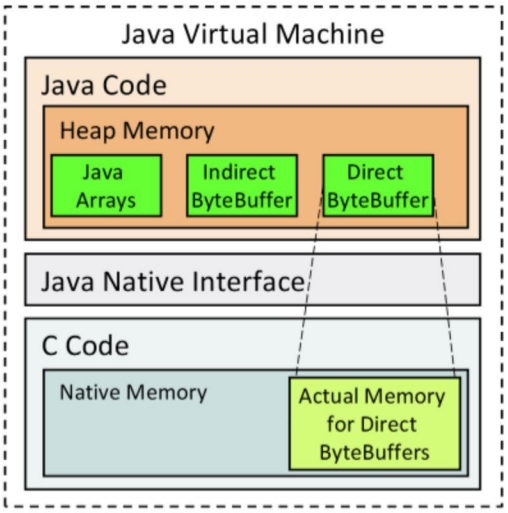

Figure 1. The Layout of Direct/Non-direct `ByteBuffers` and Java Arrays in the JVM.

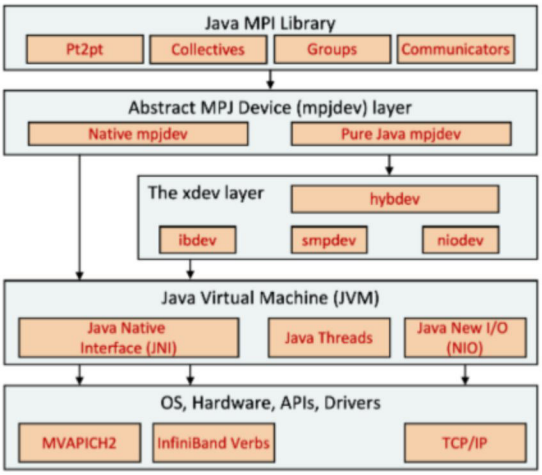

Figure 2. Layered Architecture of the Java Bindings for the MVAPICH2 Library)

mpiJava 1.2 and MPJ, were only defined until MPI 1.2 standard. On top of that, the Java Open MPI API introduced two major changes. The first change is that it supported communication to/from direct `ByteBuffers` on top of Java arrays. Secondly, the new API removed an `offset` argument to MPI communication primitives. When used with point-to-point communication methods, the `offset` field allowed communicating data from a sub-set of a Java array. This change mandates modifying Java HPC applications. Also, the Java Open MPI API does not allow using Java arrays with non-blocking communication primitives.

III. The Proposed Design

The design of the MVAPICH2-J library is inspired by the MPJ Express library as shown in Figure 2. The design depicts two communication device layers: the `mpjdev` and the `xdev` layers. In the context of MVAPICH2-J, only the `mpjdev` layer is relevant. The `mpjdev` is used to implement wrapper methods to native MPI libraries using JNI. The design philosophy is to keep the Java layer "as minimal as possible" for several reasons. This will help in easier development and maintenance of the Java MPI library. This is because the native implementation of MPI functionality can be re-used at the Java layer instead of re-implementing these in Java. Also, this will help in achieving optimal communication performance. Lastly, this will provide flexibility when porting newer systems and communication interconnects.

IV. Implementation of the MVAPICH2-J

This section presents implementation details of the MVAPICH2-J library. We begin this section with a discussion on the buffering layer, which is extensively used in the communication of Java arrays. Implementation details for Java arrays follow. Later, we present discussion on supporting communication to/from direct `ByteBuffers` in the MVAPICH2-J library. Towards the end of this section, we present implementation details of the collective communication routines.

A. The Buffering Layer

MVAPICH2-J utilizes a buffering layer that is inspired by the MPJ Express library [13]. The primary motivation of this layer is to utilize direct `ByteBuffers` to assist with communication of Java arrays. Direct `ByteBuffers` are attractive because their pointers can be retrieved in the native JNI functions. Also, this implies copying data to/from Java arrays onto `ByteBuffers`. However, this extra copy is not avoidable when communicating Java arrays through the JVM documentation recommended way of using `Get<Type>ArrayElements` and `Release<Type>ArrayElements` functions. The proposed buffering layer avoids the overhead of creating a `ByteBuffer` everytime a message comprising of Java arrays is communicated.

The buffering layer dynamically maintains a pool of direct `ByteBuffers`—backend storage—that can be used to support communication of Java arrays. It is possible to use other backend storage including indirect `ByteBuffers` or native memory created by C programs using `malloc()` or `calloc()` system calls. A buffer provided to upper layers of the software is an mpjbuf buffer that internally uses a `ByteBuffer` for storing user data.

Higher layers of the software, especially point-to-point communication primitives for Java arrays, use the buffering layer through an interface presented in Listing 1. These methods are encapsulated in the `mpjbuf.Buffer` class. The most important methods used for communicating Java arrays by point-to-point communication primitives are `write()` and `read()` methods. These methods allow copying data from Java arrays—of all basic datatypes—onto the mpjbuf buffer. Note that mpjbuf buffer utilizes direct `ByteBuffers` as backend storage mediums in our implementation. An mpjbuf buffer can possibly have multiple sections, each containing data from multiple Java arrays of the same different type. This is supported by functions like `putSectionHeader()` and `getSectionHeader()`. It is also possible to con-

figure the section size and encoding through functions like `setSectionSize()` and `setEncoding()`.

```
package mpjbuf ;

public class Buffer {
   ..

   // Write and read Methods
   public void write(type [] source,
                     int srcOff,
                     int numEls)
   public void read(type [] dest,
                    int dstOff,
                    int numEls)

   // Set and get section Headers
   public void putSectionHeader(Type type)
   public Type getSectionHeader()

   // Set and get section size
   public int setSectionSize()
   public int getSectionSize()

   // Set and get encoding
   public void setEncoding(ByteOrder encoding)
   public ByteOrder getEncoding()

   // Utility methods
   public void commit()
   public void clear()
   public void free()

   ..
}
```

Listing 1. The Functionality provided by the Buffering Layer [13]

B. Point-to-point Communication for Java Arrays

One approach for implementing communication to/from Java arrays is to utilize the JNI methods like `Get<Type>ArrayElements` and `Release<Type>ArrayElements` where ¡`<Type>` refers to all basic datatypes in Java. These methods allow the native C JNI function to retrieve a "copy" of the original array in the native code. It is possible to avoid this copy in JVMs that support memory pinning. However most current JVMs do not support memory pinning and hence incur a copy. The JVM performs a copy because the Java buffer, specified by the user, is subject to garbage collection. When this happens, the address of the buffer inside the JVM heap is likely to change and hence any pointer passed to C code for Java buffer becomes invalid. To cater this, JVM simply copies the Java buffer for the invoked C code to use—this typically happens when the `Get<Type>ArrayElements` function is called. The copy back, from C to Java is done when the `Release<Type>ArrayElements` function is called. The overhead of this copy from Java-to-C and C-to-Java is not present when the JVM supports memory "pinning". The address of the Java buffer does not change in JVM heap in this case. There are also a few drawbacks of this approach. The overhead of copying incurred by `Get<Type>ArrayElements` and `Release<Type>ArrayElements` functions for a subset of a Java array is the same as the full array. This was possible in the older mpiJava 1.2 and MPJ APIs where the communication primitives had an `offset` argument that could be used to specify the starting index of the array from where to send/receive data. However, the `offset` argument has been removed from the Java Open MPI bindings. The buffering layer—discussed in Section IV-A—allows us to avoid this since we copy only the subset of the data in the `ByteBuffer`. But since MVAPICH2-J follows the Java Open MPI bindings currently, it is not possible to demonstrate the effectiveness in sample benchmarks. However, this can be a useful feature for Java HPC applications if the `offset` argument is re-introduced in the API in future. Additionally, the buffering layer is useful for communicating derived datatypes since it is possible to copy scattered elements in the array onto consecutive location in the `ByteBuffer`.

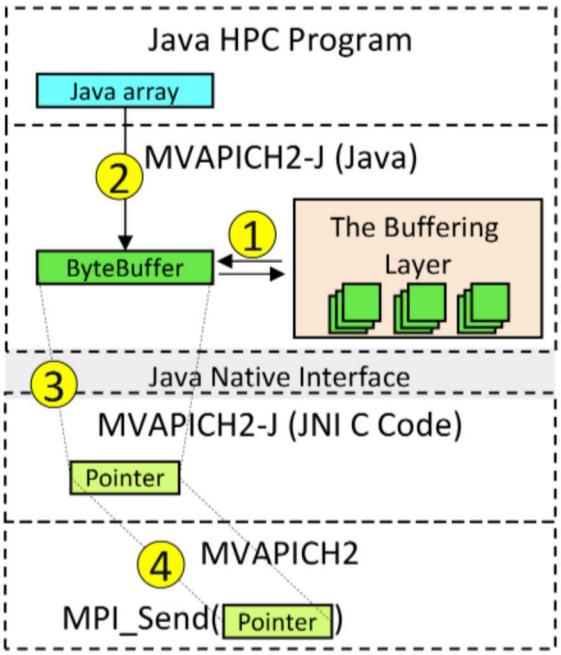

Figure 3. Communicating Java Arrays in the MVAPICH2-J Library. This is a four step process: 1) the Java side of MVAPICH2-J library gets a `ByteBuffer` from the buffering layer, 2) the user specified data is copied from Java arrays to `ByteBuffers`, 3) the C JNI function acquires a pointer to the `ByteBuffer`, and 4) this acquired pointer is used to communicate data using native C MPI library's communication primitives.)

Figure 3 depicts the overall process of communicating Java arrays in the MVAPICH2-J library. While communicating Java arrays, the user Java HPC program allocates this array and uses it for computational purposes. Note that the current discussion is in the context of basic datatype arrays in Java. As mentioned earlier, the MVAPICH2-J library uses a buffering layer to support communicating Java arrays. The first step—indicated in Figure 3 in this process—is to acquire a `ByteBuffer`) from the buffering layer. The second step is to copy contents of Java arrays on this `ByteBuffer`. In the third step of Figure 3, the `ByteBuffer` is passed to the native JNI C code that can access this buffer using a pointer without the need to incur another copy. This pointer to the `ByteBuffer`

is finally used in the fourth step to communicate data using native MPI library communication primitives.

C. Point-to-point Communication for Direct ByteBuffers

The Java Open MPI library has introduced the support for communicating data to/from direct ByteBuffers. This functionality was not part of the mpiJava 1.2 and MPJ APIs. To recap, direct ByteBuffers allocate memory outside the JVM heap and hence have a constant memory address throughout their lifecycle unlike other Java objects. This movement of Java objects in the JVM heap is due to garbage collection. Hence, it is attractive to provide support for these kinds of buffering in our Java MPI library.

Figure 4 depicts the overall process of communicating data directly from ByteBuffers in the MVAPICH2-J library. While communicating ByteBuffers, the user Java HPC program allocates this buffer and uses it either for computational purposes directly or copies data from Java arrays. The buffering layer in the MVAPICH2-J library is not used while communicating user allocated ByteBuffers. The first step—indicated in Figure 4 in this process—is to pass a reference to this buffer to the Java code in the MVAPICH2-J library. The second step is to pass this reference to the JNI C code. Finally in the third step, the JNI code uses the pointer to the ByteBuffer—allocated by the user directly—to invoke MPI communication primitives.

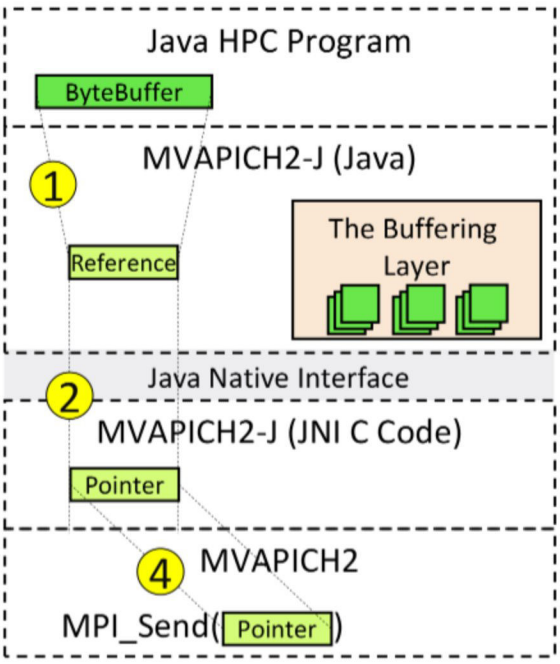

Figure 4. Communicating ByteBuffers in the MVAPICH2-J Library. This is a three step process: 1) the user passes the reference to the ByteBuffer to the MVAPICH2-J library, 2) the Java layer of the MVAPICH2-J library invokes JNI C methods with a reference to the ByteBuffer, and 3) the JNI C code uses a pointer to the buffer for communicating data using MPI primitives.

D. Collective Communication

MVAPICH2-J currently provides support for collective primitives—including vector variants. It is possible to communicate both Java arrays and direct ByteBuffers with these operations. Like the point-to-point primitives, the buffering layer is used for Java arrays. Again, the idea is to keep the Java layer as minimal as possible and utilize all optimizations and advanced collective algorithms available in the native MVAPICH2 library.

V. OSU Micro-Benchmark (OMB) for Java Bindings

The OSU Micro-Benchmark (OMB) [7] suite is a benchmarking tool—written in C—that is popularly used for evaluating MPI communication libraries on HPC systems. The suite supports an extensive range of benchmarks, ranging from blocking/non-blocking point-to-point & collectives operations to one-sided operations. It also provides a set of flags for running custom tests, varying the message sizes, among other things.

Each benchmark in OMB is designed to evaluate MPI operations, mimicking real applications while accurately reporting performance. Performance is reported under two metrics, latency (in microseconds) and bandwidth (in MBps).

OMB-J is designed with the aim of providing a benchmarking suite for Java MPI libraries that support NIO ByteBuffers along with Java arrays for communication. The suite currently has support for blocking/non-blocking point-to-point, blocking collective and vectored blocking collective operations. It also provides the ability to run custom tests as OMB does.

A. Example: OMB-J Latency Benchmark

Algorithm 1 shows a simple example of OMB-J's latency benchmark using NIO direct ByteBuffers. For Java arrays, the algorithm remains unchanged, where the only difference is the sender and receiver buffers are Java arrays. The latency benchmark reports the average latency by measuring the time a sender receives a response in a ping-pong fashion.

Algorithm 1: Latency Benchmark Example

```
1  latency = 0.0;
2  MPI.COMM_WORLD.Init(...);
3  sendBuffer = ByteBuffer.allocateDirect(maxMsgSize);
4  recvBuffer = ByteBuffer.allocateDirect(maxMsgSize);
5  for size in maxMsgSize do
6      for i:1 ... benchIters do
7          if myrank == 0 then
8              initTime = System.nanoTime();
9              MPI.COMM_WORLD.send(sendBuffer, size ...);
10             MPI.COMM_WORLD.recv(recvBuffer, size ...);
11             latency = (System.nanoTime() -
                 initTime)/(2.0*benchIters*1000);
12         else
13             MPI.COMM_WORLD.recv(recvBuffer, size ...);
14             MPI.COMM_WORLD.send(recvBuffer, size ...);
15         end
16     end
17     reportLatencyForMsgSize(latency);
18     MPI.COMM_WORLD.barrier();
19 end
```

VI. Performance Evaluation using OMB-J

This section presents performance evaluation of point-to-point and collective communication primitives for the MVAPICH2-J library along with Java Open MPI bindings using OMB-J. For point-to-point, we present the evaluation results for latency and bandwidth benchmarks. As for collectives, we present the evaluation results for allreduce and broadcast (bcast).

A. Experimental Setup

TACC Frontera: Located at the Texas Advanced Computing Center (TACC), Frontera is a large HPC system that hosts 8,368 Cascade Lake (CLX) compute nodes. Each compute node has two sockets, each carrying 28 cores (56 cores total) at 2.70GHz frequency. Each node has a total of 192GB of RAM.

The following library versions were used for the experiments: MVAPICH2-X v2.3.6 [6] and Open MPI v4.1.2 [5] + UCX 1.13.0 [14].

B. Point-to-point Communication Evaluation: Intra-node

Here we showcase the point-to-point intra-node evaluation for the two Java MPI communication libraries with `ByteBuffers` and Java arrays. The latency was measured with OMB-J's version of the `osu_latency` benchmark which reports the average latency in microseconds. The bandwidth was measured with OMB-J's `osu_bw` benchmark which reports the bandwidth in MBps.

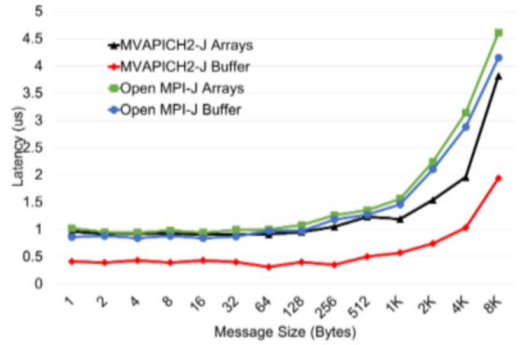

Figure 5. Intra-node latency numbers for small message sizes.

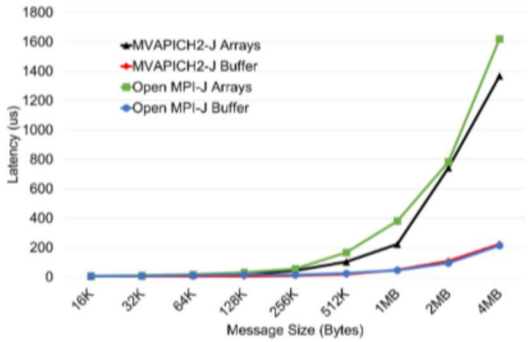

Figure 6. Intra-node latency numbers for large message sizes.

Starting with small messages (Figure 5), we can see that "MVAPICH2-J buffer" is outperforming "Open MPI-J buffer" by a factor of 2.46 on average.

At the large message end (Figure 6), "MVAPICH2-J buffer" is performing similarly to "Open MPI-J buffer". For "MVAPICH2-J arrays", there is some overhead in the performance as a result of the buffering layer.

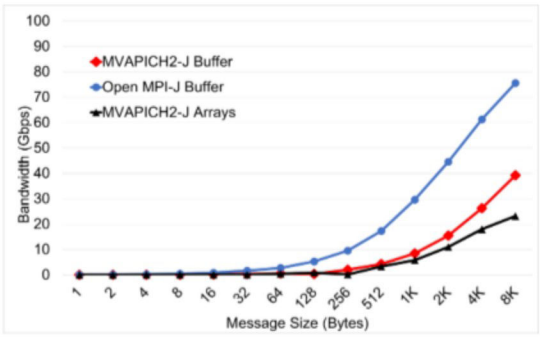

Figure 7. Intra-node bandwidth numbers for small message sizes. Bandwidth numbers for Open MPI-J arrays are not included because the library does not support Java arrays with non-blocking point-to-point operations.

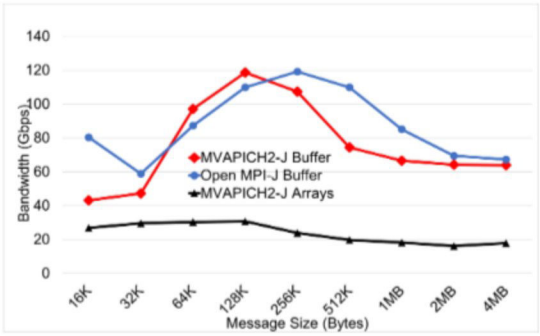

Figure 8. Intra-node bandwidth numbers for large message sizes. Since Open MPI-J does not support Java arrays for non-blocking point-to-point operations, no bandwidth numbers were collected there.

Figures 7 and 8 show the bandwidth numbers for small and large message sizes, respectively. Those figures don't have numbers for "Open MPI-J arrays" due to Open MPI-J's library not supporting communication of Java arrays using non-blocking point-to-point operations. In Figure 8, we can see that "MVAPICH2-J buffer" is picking up performance-wise with "Open MPI-J buffer".

C. Point-to-point Communication Evaluation: Inter-node

In this subsection, we showcase the inter-node point-to-point evaluation for the two Java MPI communication libraries with `ByteBuffers` and Java arrays. The latency and bandwidth were measured using the same benchmarks as discussed in the previous subsection VI-B.

From the latency numbers for small message sizes (Figure 9), we can see that "MVAPICH2-J buffer" performs comparably to "Open MPI-J buffer".

For large messages (Figure 10), "MVAPICH2-J buffer" is also performing about the same as "Open MPI-J buffer". For

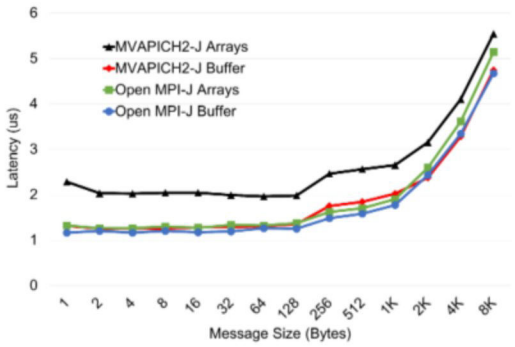

Figure 9. Inter-node latency numbers for small message sizes.

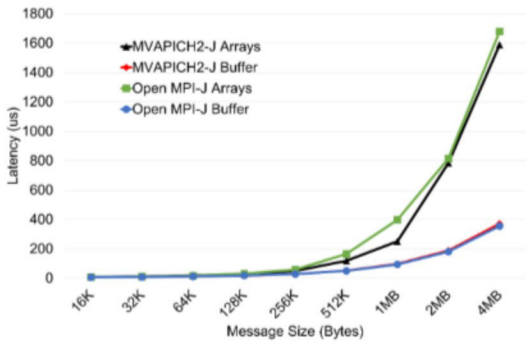

Figure 10. Inter-node latency numbers for large message sizes.

"MVAPICH2-J arrays", we see that it picks up in performance compared with "Open MPI-J arrays".

Figure 11 shows the latency overheads between the native libraries and their respective Java MPI libraries using direct `ByteBuffers`. The overheads are in the ballpark of 1 microsecond, with MVAPICH2-J having a smaller latency overhead compared to Open MPI-J.

Figures 12 and 13 are bandwidth numbers for inter-node experiments. Here, again, we see that the figures do not have "Open MPI-J arrays" numbers due to the same reasons mentioned earlier in VI-B.

For larger messages (Figure 13), we see that "MVAPICH2-J buffer" catches up in performance but still is slightly lagging behind "Open MPI-J buffer".

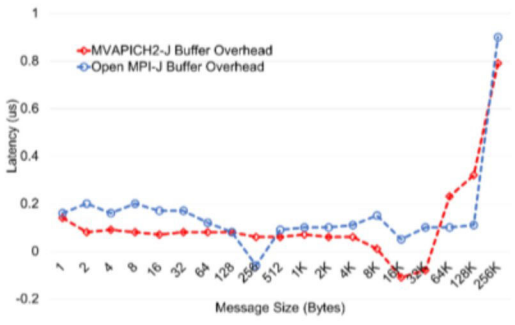

Figure 11. Inter-node latency overhead between native and MVAPICH2-J library for direct `ByteBuffers`.

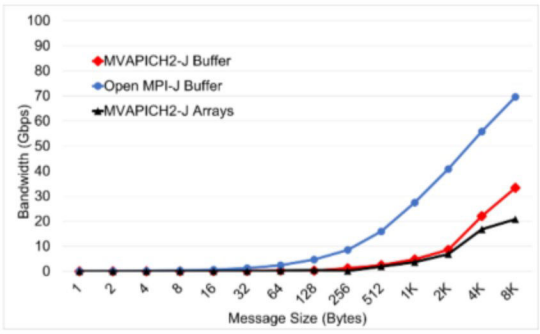

Figure 12. Inter-node bandwidth numbers for small message sizes.

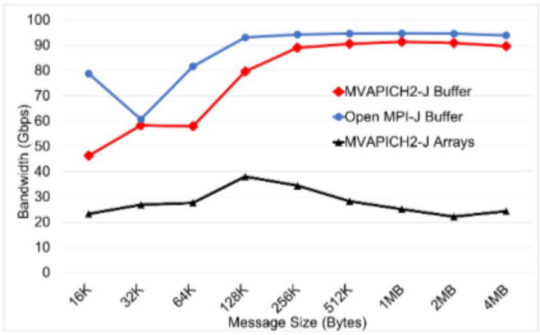

Figure 13. Inter-node bandwidth numbers for large message sizes.

D. Collective Communication Evaluation: Broadcast

We present the latency numbers for the broadcast collective operation with the two Java MPI communication libraries using `ByteBuffers` and Java arrays. The latency was measured using the Java version of the OMB `osu_bcast` benchmark which reports the average latency across all processes in microseconds. The benchmark uses `MPI_Reduce` as part of the latency calculation. The experiments were conducted on 4 nodes with 64 processes in total—16 processes for each node.

Figures 14 and 15 both present the latency numbers for the broadcast collective benchmark. For all message sizes, "MVAPICH2-J buffer" is outperforming "Open MPI-J buffer" by a factor of 6.2 on average, and "MVAPICH2-J arrays" is also outperforming "Open MPI-J arrays" by a factor of 2.2 on average. These performance benefits are due in large to the performance differences of the native libraries in place.

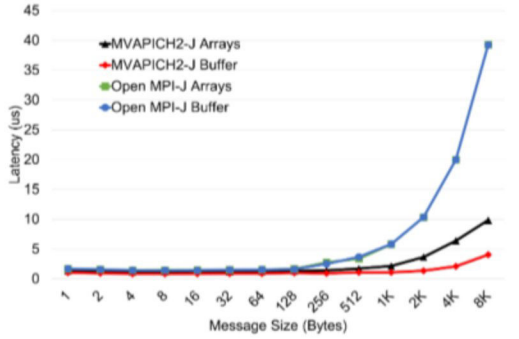

Figure 14. Broadcast latency numbers for small message sizes.

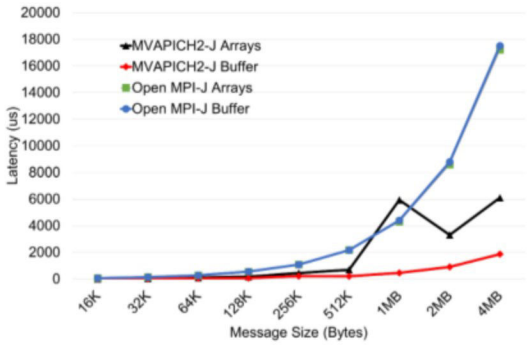

Figure 15. Broadcast latency numbers for large message sizes.

E. Collective Communication Evaluation: Allreduce

For this subsection, the latency numbers for allreduce are presented. The numbers were collected using the Java version of the OMB benchmark `osu_allreduce`. The benchmark reports the average latency in microseconds across all processes. We also ran the benchmark on 4 nodes with 64 processes in total—16 processes each.

For Figures 16 and 17, "MVAPICH2-J buffer" is performing better than "Open MPI-J buffer", where over all message sizes, "MVAPICH2-J buffer" is fairing well by a factor of 2.76 on average compared to "Open MPI-J buffer", and "MVAPICH2-J arrays" is outperforming "Open MPI-J arrays" by a factor of 1.62 on average. Again, these performance advantages are largely due to the underlying native MPI libraries and their performances.

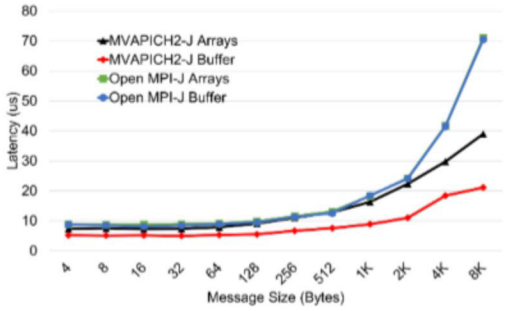

Figure 16. Allreduce latency numbers for small message sizes.

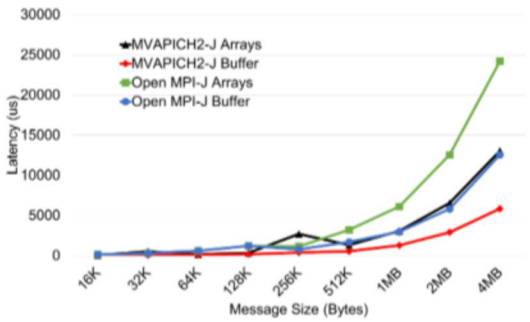

Figure 17. Allreduce latency numbers for large message sizes.

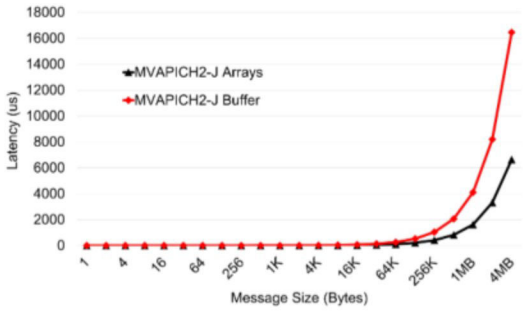

Figure 18. Inter-node point-to-point latency numbers with data validation comparing the `ByteBuffer` API against Java arrays for MVAPICH2-J.

F. Evaluation of the `ByteBuffer` API

Figure 18 presents the latency numbers for the MVAPICH2-J library—for Java arrays versus `ByteBuffers`—measured using the OMB `osu_latency` benchmark. Here, we plan to evaluate the effectiveness of using `ByteBuffers` instead of Java arrays. In order to emulate the application-level behavior, we have enabled data validation for the OMB `osu_latency` benchmark. This means that Java arrays and `ByteBuffers` are populated, at sender, and later the data is validated at the receiver process. As the results depict in Figure 18, the latency for populating, communicating, and verifying `ByteBuffers` is outperformed by Java arrays after the message size of 256 bytes. For the 4MB message size, Java arrays outperform `ByteBuffers` by 3×.

G. Discussion

The performance evaluation reveals that, overall, for collectives, allreduce and bcast, MVAPICH2-J is performing better than Open MPI-J for both `ByteBuffers` and Java arrays. This is largely due to the performance differences in the native MPI libraries. We saw for point-to-point, latency and bandwidth, the performance of MVAPICH2-J with `ByteBuffers` is similar to that of Open MPI-J. There are overheads that exist when using Java arrays and that is in part due to the buffering layer. The buffering layer is needed by our library to support communicating derived datatypes and Java arrays using non-blocking point-to-point communication primitives. Java Open MPI does not support non-blocking point-to-point communication with Java arrays. At the OMB-J level, `ByteBuffers` perform better than Java arrays. This is not the case at the application level. A `ByteBuffer` is basically an array that is wrapped with a higher-level interface. Naturally, one would assume that this extra layer of abstraction wouldn't incur any performance penalties. However, it appears that's not the case as seen in Figure 18. As a result of the extra abstraction layer, reads and writes for `ByteBuffers` are slower. In Figure 11, we can see that Java based libraries under-perform when compared to native libraries. This is because Java is simply running more errands (i.e. GC) during the runtime of the application as opposed to C.

VII. Related Work

Historically, there have been three approaches taken for writing Java MPI libraries. The first approach was to rely on Java alone in order to provide a fully portable solution. The second approach was to rely on native MPI libraries for communication using JNI. The first approach was followed by some MPI libraries. It proved out not to be practical almost all of the MPI features need to be re-implemented in this approach. The second approach was adopted by libraries like mpiJava [2] and Java Open MPI [5]. The third approach is a hybrid one where some MPI libraries have one or more device layers that allow providing pure Java as well as JNI-based communication devices—this is the approach taken by MPJ Express [4] and FastMPJ [3]. mpiJava and MPJ Express follow the mpiJava 1.2 API. FastMPJ has support for both mpiJava 1.2 and MPJ API. The Java Open MPI library follows its custom API that we attempt to follow in MVAPICH2-J. The design goal of MVAPICH2-J is to keep the Java layer as lean as possible and exploit the features and optimizations provided by the native MVAPICH2-J library.

VIII. Conclusion

This paper presented our initial experiences of designing and implementing Java bindings—named MVAPICH2-J—for the production-quality MVAPICH2 library. MVAPICH2-J follows the Java Open MPI library API that supports communicating data to/from Java arrays and `ByteBuffers`. The main idea for MVAPICH2-J is to utilize JNI and keep the Java layer as minimal as possible. We also implemented and utilized an internal buffering layer to facilitate communication of Java arrays. In order to evaluate performance of Java MPI libraries, we also designed and implemented OMB-J, which is a Java version of the popular OSU Micro-benchmarks (OMB) suite. OMB-J currently provides a variety of point-to-point and collective benchmarks with support for both `ByteBuffers` and Java arrays. Our evaluation reveals that using `ByteBuffers` provides better performance compared to Java arrays at the OMB-J level. The point-to-point performance of MVAPICH2-J is comparable to Java Open MPI for the `ByteBuffer` API. There is slight overhead in the performance while communicating Java arrays using MVAPICH2-J due to the internal buffering layer that is needed to support communicating derived datatypes and Java arrays with non-blocking point-to-point functions. For the collective benchmarks, MVAPICH2-J outperforms OpenMPI-J for the `ByteBuffer` API by a factor of 6.2 and 2.76 for broadcast and allreduce, respectively, on average for all messages sizes. For Java arrays, we observe $2.2\times$ and $1.62\times$ better performance than Open MPI-J—on average for all message sizes—for broadcast and allreduce, respectively. These gains are mainly due to better performance of MVAPICH2 for collective communication routines. We also showed—somewhat surprisingly—that Java arrays perform better than direct `ByteBuffers` when the time to populate and validate data was included for the point-to-point latency benchmark. The reason is that it is faster to read/write data from Java arrays compared to `ByteBuffers`. We plan to release MVAPICH2-J along with OMB-J in the near future. We also plan to work with other stakeholders in the Java HPC community to come up with an agreed Java MPI API.

IX. Acknowledgement

This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002.

References

[1] The MPI Forum, "The Message Passing Interface (MPI) 4.0 Standard," urlhttps://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2021, Accessed: March 19, 2022.

[2] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, "MPJ: MPI-like message passing for Java," *Concurrency: Practice and Experience*, vol. 12, no. 11, pp. 1019–1038, 2000. [Online]. Available: {https://onlinelibrary.wiley.com/doi/abs/10.1002/1096-9128\%28200009\%2912\%3A11\%3C1019\%3A\%3AAID-CPE518\%3E3.0.CO\%3B2-G}

[3] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo, "Low-latency Java communication devices on RDMA-enabled networks," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4852–4879, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3473

[4] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core HPC systems using Java," *Journal of Parallel and Distributed Computing*, vol. 69, no. 6, pp. 532–545, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731509000252

[5] O. Vega-Gisbert, J. E. Roman, and J. M. Squyres, "Design and implementation of Java bindings in Open MPI," *Parallel Computing*, vol. 59, pp. 1–20, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819116300758

[6] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The MVAPICH project: Transforming research into high-performance MPI library for HPC community," *Journal of Computational Science*, vol. 52, p. 101208, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750320305093

[7] The MVAPICH2 Development Team, "Ohio Micro-Benchmarks (OMB)," urlhttps://mvapich.cse.ohio-state.edu/benchmarks/, 2021, Accessed: March 19, 2022.

[8] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, oct 2016. [Online]. Available: https://doi.org/10.1145/2934664

[9] Apache Software Foundation, "Apache Hadoop," url-https://hadoop.apache.org, 2022, Accessed: March 19, 2022.

[10] Skymind Global Limited, "DeepLearning4j (DL4J)," url-https://deeplearning4j.konduit.ai/, 2022, Accessed: March 19, 2022.

[11] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, p. 63–73, sep 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306

[12] D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo, "Npb-mpj: Nas parallel benchmarks implementation for message-passing in java," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 181–190.

[13] M. Baker, B. Carpenter, and A. Shafi, "A Buffering Layer To Support Derived Types and Proprietary Networks for Java HPC," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, pp. 348–358, 2007. [Online]. Available: https://www.scpe.org/index.php/scpe/article/view/430/93

[14] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "Ucx: An open source framework for hpc network apis and beyond," pp. 40–43, Aug 2015.