Multicore-enabling the MPJ Express Messaging Library *

Aamir Shafi Jawad Manzoor
Kamran Hameed
School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad, Pakistan
{aamir.shafi,jawad.manzoor,kamran.hameed}@seecs.edu.pk

Bryan Carpenter
School of Computing, University of Portsmouth, UK
bryan.carpenter@port.ac.uk

Mark Baker
School of Systems Engineering, University of Reading, UK
mark.baker@computer.org

Abstract

With the transition to multicore processors almost complete, the parallel processing community is seeking efficient ways to port legacy message passing applications on shared memory and multicore processors. MPJ Express is our reference implementation of Message Passing Interface (MPI)-like bindings for the Java language. Starting with the current release, the MPJ Express software can be configured in two modes: the multicore and the cluster mode. In the multicore mode, parallel Java applications execute on shared memory or multicore processors. In the cluster mode, Java applications parallelized using MPJ Express can be executed on distributed memory platforms like compute clusters and clouds. The multicore device has been implemented using Java threads in order to satisfy two main design goals of portability and performance. We also discuss the challenges of integrating the multicore device in the MPJ Express software. This turned out to be a challenging task because the parallel application executes in a single JVM in the multicore mode. On the contrary in the cluster mode, the parallel user application executes in multiple JVMs. Due to these inherent architectural differences between the two modes, the MPJ Express runtime is modified to ensure correct semantics of the parallel program. Towards the end, we compare performance of MPJ Express (multicore mode) with other C and Java message passing libraries—including mpiJava, MPJ/Ibis, MPICH2, MPJ Express (cluster mode)—on shared memory and multicore processors. We found out that MPJ Express performs signicantly better in the multicore mode than in the cluster mode. Not only this but the MPJ Express software also performs better in comparison to other Java messaging libraries including mpiJava and MPJ/Ibis when used in the multicore mode on shared memory or multicore processors. We also demonstrate effectiveness of the MPJ Express multicore device in Gadget-2, which is a massively parallel astrophysics N-body siimulation code.

* Portions of this work were previously reported in the Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '10), Atlanta, GA, April 19–23, 2010, pp. 1-7

PPPJ '10 September 15-17,2010, Vienna, Austria.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Algorithms, Design, Performance

Keywords Java Multicore Programming, Java MPI, MPI Java, MPJ Express, Java HPC

1. Introduction

The computer hardware and software industry has witnessed a significant change [17] with single power-hungry processing cores making way for multiple energy-efficient processing cores—better known as *multicore* processors. As a consequence of this sea change, the software programmers cannot rely on processor vendors to improve performance of their applications. Instead these programmers must explicitly utilize concurrency (or parallel computing) to exploit multiple cores on the multicore processors.

The emergence of multicore processors is also prevalent in modern compute clusters typically employed by the High Performance Computing (HPC) community. These modern clusters are built with compute nodes comprising of a combination of Symmetric Multi-Processors (SMP) and multicore processors. The TOP500 [18] community has even designated a new name—*constellations*—for this breed of clusters. Traditionally on clusters built with single processor compute nodes, Message Passing Interface (MPI) [12] compliant libraries are used for writing parallel applications. The MPI standard, by design, is targeted towards programming distributed memory clusters—the underlying model is Single Program Multiple Data (SPMD). MPI libraries and the associated runtime software must be adapted to execute legacy parallel applications efficiently on multicore processors.

The MPI standard provides bindings for traditional languages including C, C++, and Fortran. Some of the popular MPI libraries include Open MPI [8] and MPICH2 [10]. On the other hand, several scientists [4][7][9][13] have suggested that Java could make an excellent language for developing HPC software. Compared with traditional languages like C and Fortran, Java provides portability, higher-level programming concepts, improved compile time and runtime checking, and, as a result, faster problem detection and debugging. The JVM automatically manages the memory utilized by the program data structures. The built-in support for threads provides a way to insert parallelism in Java applications. The Java Development Kit (JDK) includes a large set of libraries that can be reused by developers for rapid application development. Another, interesting, argument in favour of Java is the large pool of developers—the main reason is that Java is taught as one of the major languages in many Universities around the globe.

Realizing these benefits of Java, we developed the MPJ Express [15] software that is an open source and free implementation of the mpiJava 1.2 API [6] standardized by the Java Grande Forum [11]. With the universal adoption of multicore processors in HPC hardware, it is important to multicore-enable the MPJ Express software. This essentially means that the software and its runtime must effectively parallelize and execute message passing user applications on multicore processors. By multicore-enabling the software, we intend to provide an efficient way to port legacy message passing applications on shared memory and multicore processors.

For this purpose, the current version of the MPJ Express software now provides two software configurations: the multicore and the cluster configuration. The multicore configuration is especially designed and implemented to allow application developers to efficiently execute legacy code—without any modification—on multicore and shared memory processors. On the other hand, the cluster configuration—the default configuration in earlier releases—is meant for parallel applications executing on clusters or network of computers. In this configuration there are two communication devices, Java New I/O (NIO) device (called *niodev*) and Myrinet eXpress (MX) device (called *mxdev*).

In this paper we describe our experiences of multicore-enabling the MPJ Express messaging library. For this purpose, we have developed the multicore device based on Java threads. This device is an implementation of the *xdev* API, which is a device layer in the MPJ Express software. The underlying idea is that instead of relying on network-based communication, the software must use shared memory communication on multicore and shared memory processors. There are several ways of implementing multicore device for MPJ Express. Let us consider a multicore processor as the target platform for executing message passing user application. The first option is to employ Java threads; each MPJ Express process is a thread running inside a single JVM on the multicore processor. Here the underlying device implements message passing using global or shared variables between threads. The other option is to start a single JVM (MPJ Express process) for each core and these processes communicate to each other via native shared memory APIs like `sysv` and `usysv`. Our design goals for developing multicore device are performance and portability. In our previous study [14] we evaluated these techniques and found out that employing Java threads is the most efficient and portable mechanism of implementing the shared memory communication and multicore device for MPJ Express.

We also discuss the challenges of integrating the multicore device in the MPJ Express software. This turned out to be a challenging task because the parallel application executes in a single JVM in the multicore mode. On the contrary in the cluster mode, the parallel user application executes in multiple JVMs. Due to these inherent architectural differences between the two modes, the MPJ Express runtime is modified to ensure correct semantics of the parallel program. In addition we evaluate the performance of our multicore device against the cluster configuration devices (niodev and mxdev) and other message passing libraries including MPJ/Ibis, mpiJava, and MPICH2. We found that the performance of MPJ Express (multicore configuration) is substantially better than MPJ Express (cluster configuration) on shared memory and multicore processors. Also MPJ Express library performs well in comparison to other MPI libraries. We also demonstrate performance benefits of MPJ Express (multicore mode) in a real-world cosmological simulation code Gadget-2.

To the best of our knowledge none of the existing pure Java messaging systems provide a portable and efficient method of porting legacy message passing parallel applications on multicore processors. In this context, the main contribution of this paper is the design, implementation, integration, and evaluation of the multicore device.

Rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents design of the MPJ Express software. We also present the API used to implement the multicore device. Section 4 presents implementation details and algorithms for the MPJ Express software. Section 5 discusses our methodology to integrate the multicore device into the MPJ Express software. We evaluate performance of the multicore device in the Section 6. We conclude and discuss future work in the Section 7.

2. Related Work

This section discusses MPI libraries that support programming multicore clusters by providing a shared memory communication device. The idea is that MPI processes running on multiple cores of a processor communicate to each other via shared memory.

In this context, popular native MPI libraries including Open MPI and MPICH2 provide high performance shared memory communication devices on various platforms. These devices can be used for communication between MPI processes executing on multiple cores of a processor. Open MPI is an open-source production quality implementation of both MPI-1 and MPI-2. It fully supports concurrent and multi-threaded applications. It includes the features of thread safety, concurrency and dynamic process spawning. MPICH2 is another high-performance and widely portable implementation of the MPI-2 standard from the Argonne National Laboratory. MPICH2 is designed for implementation of MPI on SMPs, clusters, and massively parallel processors. It provides various communication devices. One of these devices is the ch3 device which contains provides shared memory communication through ssm and shm channels and also through the nemesis communication subsystem. Again, these shared memory devices provide superior inter-node performance on target multicore processors.

The current generation of Java messaging systems—that include MPJ Express and MPJ/Ibis—lag behind in their support for programming multicore processors using shared memory communication devices. MPJ/Ibis [3] is an implementation of the MPJ API specification on top of Ibis. The communication devices used by MPJ/Ibis are not thread-safe. MPJ/Ibis has communication devices for java.io package, Java NIO package, and Myrinet. This software currently does not provide any shared memory communication device. mpiJava [1] is another Java messaging system which provides a fully functional and efficient object-oriented Java interface to MPI. A unique feature of this messaging library is that we can configure it to use any native C MPI library. In this way we can indirectly benefit from shared memory communication devices in these native libraries.

Earlier versions of the MPJ Express only provided two communication devices meant for Ethernet or Myrinet based interconnects. In this paper we introduce the multicore communication device—implemented using Java threads—that is recently added to the public MPJ Express release. With this shared memory or multicore device, legacy Java applications developed for distributed memory platforms like clusters/clouds can be executed on shared memory or multicore processor machines. MPJ Express is a thread-safe software, which means that it can be used in combination with JOMP threads to support nested parallelism [15], which exploits two levels of parallelism at the application level—threading and message passing. A drawback of this approach is that user applications must be modified to exploit multicore clusters.

To the best of our knowledge, this is the first attempt to efficiently port parallel message passing Java applications on multicore processors. Our contribution is the design, implementation, and evaluation of threads-based multicore communication device for MPJ Express.

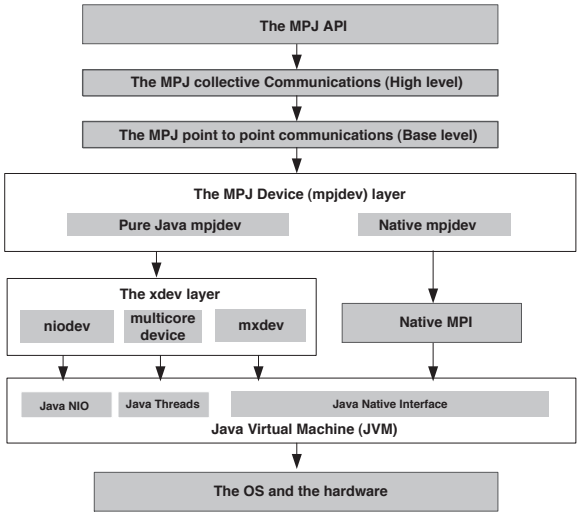

Figure 1. MPJ Express Architecture

3. MPJ Express Design

This section discusses the design of the MPJ Express software. We also discuss in detail a particular device API that is used to implement the multicore device for the software.

MPJ Express has a layered design that allows incremental development, and provides the capability to update and swap layers in or out as required. At runtime end users can opt to use high-performance proprietary network devices, or choose the pure Java devices that use sockets or Java threads for portability. Figure 1 illustrates the design and the different levels of the software: the MPJ API, high level, base level, *mpjdev*, and *xdev*. The top three layers are directly used by MPJ Express users to write their parallel Java applications. The base level contains point-to-point functionality provided by MPJ Express. Similarly high level and MPJ API refers to collective communications, derived datatypes, virtual topologies functionality.

The design shows that the software has two device layers: the mpjdev and the xdev layer. The first device layer, called mpjdev, has two implementations. The first implementation *Pure Java mpjdev* in turn uses the second device layer called xdev. The second implementation *Native MPI* uses JNI wrappers to a native MPI library. This device is currently not available in the public release of the software. The xdev device layer has three implementations: niodev, the multicore device, and mxdev. niodev and mxdev refer to communication devices built using Java New I/O (NIO) and Myrinet eXpress (MX) libraries respectively. These two libraries are used in the cluster mode of MPJ Express. The last implementation of xdev (the multicore device) is implemented using Java threads and is used in the multicore mode of the software. This paper discusses the implementation, performance evaluation, and integration of the multicore device of the MPJ Express software.

The motivation behind developing a multicore device for the MPJ Express software is to achieve efficient performance on multicore processors. Without such device MPJ Express processes running on various cores of a multicore machine communicate using niodev that is implemented using sockets. This mechanism involves ”loopback” from network card and it is inefficient for shared memory machines because it unnecessarily forces the messages to go through all the layers of the network protocol stack. This is depicted in Figure 2. Ideally in such a scenario MPJ Express processes should communicate via shared memory—this is shown in Figure 3.

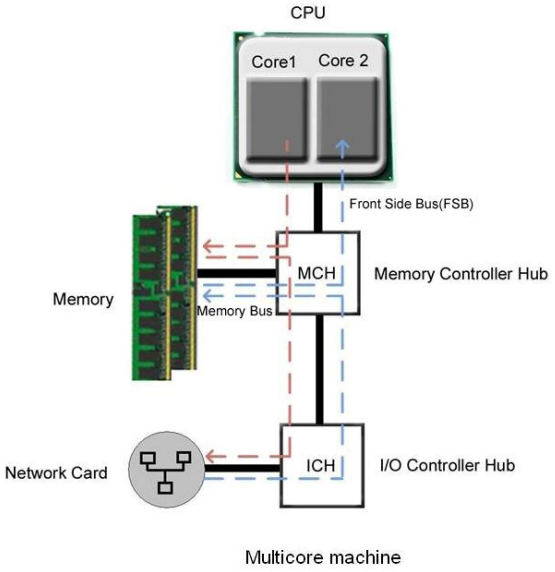

Figure 2. Communication using Sockets on a Shared Memory or Multicore Machine

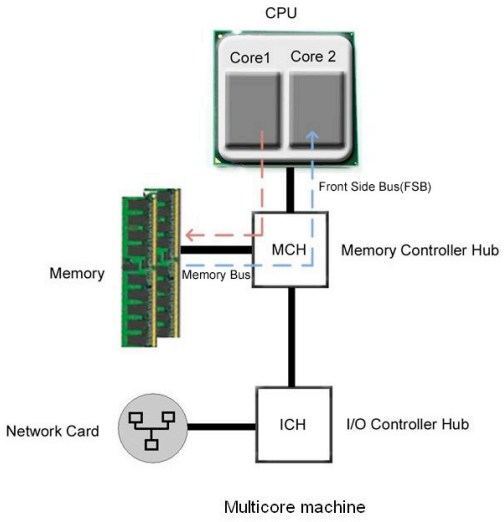

Figure 3. Communication using Shared Memory on a Shared Memory or Multicore Machine

```
public interface Device {

 /* Category 1: Initialize and shutdown the xdev device driver */
 public abstract ProcessID[] init(String[] args)
                                          throws XDevException;
 public abstract void finish() throws XDevException;

 /* Category 2: Get ProcessID for the current MPJ process */
 public abstract ProcessID id();

 /* Category 3: Blocking and non blocking send and receive methods */
 public abstract mpjdev.Request isend(mpjbuf.Buffer buf,
    ProcessID destID, int tag, int context) throws XDevException;
 public abstract void send(mpjbuf.Buffer buf,
    ProcessID destID, int tag, int context) throws XDevException;
 public abstract mpjdev.Request issend(mpjbuf.Buffer buf,
    ProcessID destID, int tag, int context) throws XDevException;
 public abstract void ssend(mpjbuf.Buffer buf,
    ProcessID destID, int tag, int context) throws XDevException;
 public abstract mpjdev.Status recv(mpjbuf.Buffer buf,
     ProcessID srcID, int tag, int context) throws XDevException;
 public abstract mpjdev.Request irecv(mpjbuf.Buffer buf,
     ProcessID srcID, int tag, int context, mpjdev.Status status)
                                          throws XDevException;

 /* Category 4: Utilities for checking incomimg messages*/
 public abstract mpjdev.Status probe(ProcessID srcID, int tag,
                              int context) throws XDevException;
 public abstract mpjdev.Status iprobe(ProcessID srcID, int tag,
                              int context) throws XDevException;
 public abstract mpjdev.Request peek() throws XDevException;

}
```

Figure 4. The xdev API

The multicore device, presented in this paper, is essentially an implementation of the xdev device layer. The purpose of the xdev API is to develop a thin communication device driver that can be easily adapted for new communication conduits.

Figure 4 presents the API for the xdev device layer. The methods provided by this device layer can be roughly divided into four categories:

1. Initialize and shutdown the xdev device driver
2. Get ProcessID for the current MPJ Express process
3. Blocking and non blocking send and receive methods
4. Utilities for checking incoming messages

4. Implementation of the Multicore Device

In this section, we discuss implementation details of the multicore device. The multicore device is built using Java threads. This device can also be used to program SMP machines where processors share the main memory. Our approach is inspired by the shared memory implementation of the Adlib communication library for HPJava [5]. Using this approach, each MPJ process is essentially represented by a Java thread and data is communicated using shared data structures. An obvious advantage of this approach—especially in the context of Java—is that an application does not compromise portability. Other shared memory devices rely on the JNI API and some underlying native implementation, which obviously varies for different OS platforms. Another advantage of this approach is better performance since we can avoid JNI and additional copying overheads.

The remainder of this section reviews various features of the multicore device in more detail.

4.1 Initialization and Finalization of the Multicore Device

Figure 5 shows the initialization routine for the native device. The MPJ Express runtime passes some meta-data like the total number of processors involved in computation. Also the device keeps track of registered threads—this is indicated by the `numRegisteredThreads` variable. Each time a thread calls the initialization procedure, `numRegisteredThreads` is incremented. When the value of this variable equals `nprocs`, then all threads are notified to continue execution. This implies that all threads have called the initialization routine and can begin their computational tasks.

4.2 Blocking and Non-blocking Communication

This subsection discusses implementation details of the standard non blocking send and receive methods.

Figure 6 and 7 shows implementation sketch of non-blocking send and receive methods. Our device extensively uses `sendQueue` and `recvQueue` for non-blocking communication functionality. We first focus on the functionality of non-blocking send method. Here a `sendRequest` is initialized, which stores the sending buffer reference, `destID`, `tag`, and context information. Later this `sendRequest` is used to find and later remove a `matchingRecvRequest` from `recvQueue`. A matching receive request object will only be found if the non-blocking receive method has already been called by the receiver. If it exists, then message is directly copied to buffer location specified by the receiver. Otherwise, the `sendRequest` is simply stored in the `sendQueue`—physical message transfer occurs when the non-blocking receive method is called.

Similarly Figure 7 shows the non-blocking receive functionality. Here a `recvRequest` is initialized and used to find a matching send request from the `sendQueue`. A match will only exist if the send method has already been called by the sender thread. If no match exists, then the `recvRequest` is added to the `recvQueue`.

The process of communication messages by writing to and reading from the shared queues is shown in the Figure 8. The function of each block is explained by the diagram next to it. The dotted lines show the movement of data in buffer or the whole Send Request/ Receive Request object.

4.3 Utilities for Checking Incoming Messages

The xdev API provides some utility methods that can be used for checking and probing incoming messages at the receiver process without actually receiving them. Examples of such methods include `iprobe()` and `probe()` methods. Psuedocode for the `iprobe()` is shown in the Figure 9. Another method—that falls in this category— provided by the xdev API is the `peek()` method, which is a blocking operation that returns the most recently completed `Request` object.

5. Integration with MPJ Express

This section presents a discussion of integrating the multicore device in the MPJ Express software. The MPJ Express runtime and some higher layers of the software were modified for this purpose and we outline some of the changes in this section.

5.1 Ensuring Correct Semantics of the Parallel Program by using Custom Class Loading

The task of integrating the multicore device into the MPJ Express software turned out to be a challenging one due to inherent architectural differences between the cluster and the multicore configuration mode. In the cluster mode the runtime starts new JVMs to represent individual MPI processes. But in the case of the multicore device, each MPI process is represented by a Java thread. This

```
1 public class SMPDeviceImpl {
2
3   ...
4   int numRegisteredThreads = 0;
5
6   ProcessID id = new ProcessID(
7                     UUID.randomUUID());
8   int size ;
9   Thread [] threads ;
10   HashMap ids ;
11   xdev.ProcessID id = null;
12   xdev.ProcessID[]  pids = null;
13
14   SMPDeviceImpl WORLD =
15                   new SMPDeviceImpl();
16   ...
17
18   ProcessID[] init(String file,
19                            int rank) {
20
21     Thread currentThread =
22           Thread.currentThread() ;
23     nprocs is the total number of procs
24
25     if (numRegisteredThreads == 0) {
26
27       WORLD.size = nprocs ;
28       WORLD.pids =
29         new ProcessID [WORLD.size];
30       WORLD.threads =
31            new Thread [WORLD.size];
32       WORLD.ids = new HashMap() ;
33
34       .. assign a context for the
35       xdev-level MPI communicator
36       representative ..
37
38     }
39
40     if(currentThread is not
41               already registered) {
42
43       WORLD.id =
44         new ProcessID(UUID.randomUUID()) ;
45       WORLD.pids[rank] = WORLD.id;
46       WORLD.threads [rank] = thread ;
47       WORLD.ids.put(thread, WORLD.id) ;
48
49       numRegisteredThreads++ ;
50
51       if(numRegisteredThreads
52                         == WORLD.size) {
53         initialized = true ;
54         notify all waiting threads
55       }
56       else {
57         currentThread waits
58       }
59
60     }
61
62     return WORLD.pids ;
63
64   }
65 }
```

Figure 5. Pseudocode for `init` method

```
1 ...
2 RecvQueue recvQueue = new RecvQueue() ;
3 SendQueue sendQueue = new SendQueue() ;
4
5 public Request isend(mpjbuf.Buffer buf,
6                       ProcessID destID,
7                       int tag, int context)
8                       throws XDevException {
9
10   initialize sendRequest
11
12   acquire class-level lock {
13
14     find and remove matchingRecvRequest
15                              from recvQueue
16
17     if(matchingRecvRequest is found) {
18       copy message from sender buffer
19                          to receiver buffer
20       set pending flag to false in
21        sendRequest and matchingRecvRequest
22
23       notify the receiver thread
24     }
25     else {
26       add sendRequest object to sendQueue
27     }
28   }
29
30   return sendRequest
31
32 }
33 ...
```

Figure 6. Pseudocode for `isend` method

```
1 ...
2 RecvQueue recvQueue = new RecvQueue() ;
3 SendQueue sendQueue = new SendQueue() ;
4
5 public Request irecv(mpjbuf.Buffer buf,
6                       ProcessID srcID,
7                       int tag, int context)
8                       throws XDevException {
9
10   initialize recvRequest ;
11
12   access class-level lock {
13
14     find and remove matchingSendRequest
15                             from sendQueue
16
17     if(matchingSendRequest is found) {
18       copy message from sender buffer
19                          to receiver buffer
20       set pending flag to false in
21       recvRequest and matchingSendRequest
22
23       notify the sender thread
24     }
25     else {
26       add recvRequest object to recvQueue
27     }
28   }
29
30   return recvRequest ;
31
32 }
33 ...
```

Figure 7. Pseudocode for `irecv` method

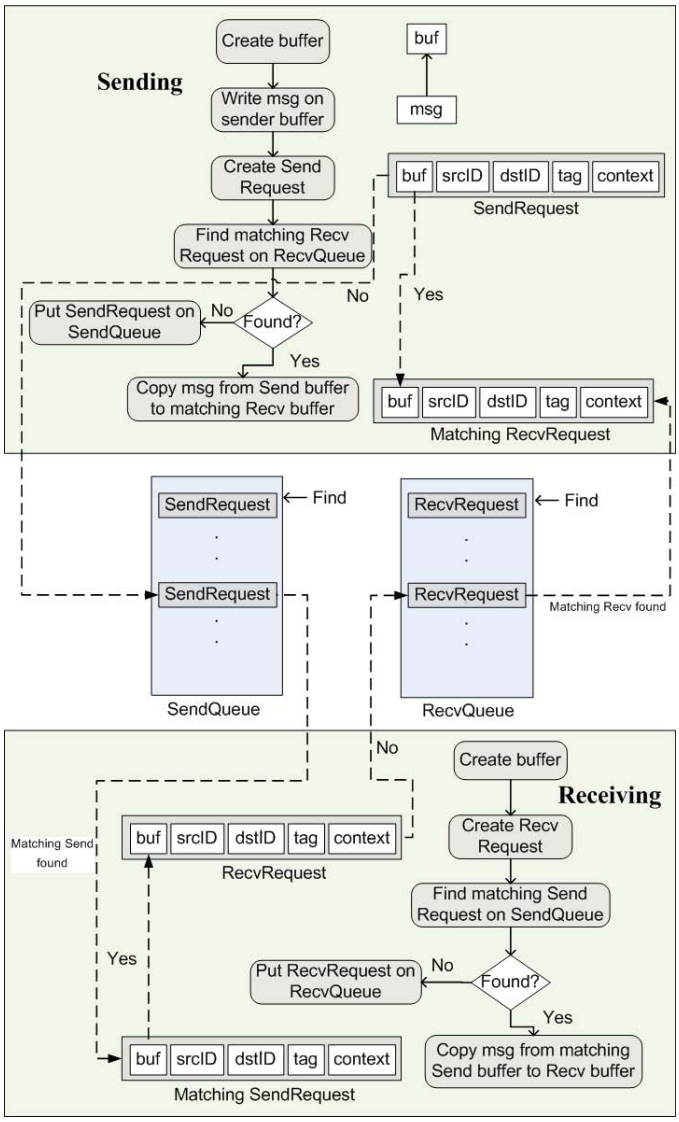

Figure 8. Communication between MPJ Express Sender and Receiver Threads

```
1  ...
2  RecvQueue recvQueue = new RecvQueue() ;
3  SendQueue sendQueue = new SendQueue() ;
4
5  public mpjdev.Status iprobe(ProcessID srcID, int tag,
6                              int context) throws XDevException {
7
8    mpjdev.Status status = null;
9    ProcessID myID = id();
10   SMPSendRequest request = sendQueue.check(context,
11                                       myID, srcID, tag);
12
13   if (request != null) {
14     status = new mpjdev.Status(request.srcID.uuid(),
15                 request.tag, -1, request.type, request.numEls);
16   }
17
18   return status;
19 }
20 ...
```

Figure 9. Pseudocode for `iprobe` method

```
1 import mpi.*;
2
3 public class HelloBug {
4
5   static int sharedVar = 0 ;
6
7   public static void main(String args[]) throws Exception {
8     MPI.Init(args) ;
9     int rank = MPI.COMM_WORLD.Rank() ;
10    sharedVar++ ;
11    System.out.println("Proc <"+rank+">: sharedVar = "+
12                                          "<"+sharedVar+">") ;
13    MPI.Finalize() ;
14  }
15 }
```

Figure 10. Source-code to Demonstrate Runtime Issues with the Multicore Device

essentially means that the MPJ Express runtime and some parts of the higher level middleware code were modified to take care of this major change.

A major hurdle is that in the multicore configuration, all MPJ Express processes share the same code and data variables. In the case of cluster configuration, since all processes execute in a seprate JVM, no code or variables are shared between processes. This kind of resource sharing is essential for writing the multicore device but it also creates problems at the top levels of the software. To explain this particular issue further, we take an example of a user parallel application that defines `static` variables in it. Figure 10 shows the source code that demonstrates this issue. The code shows a hello world MPJ Express. At line 5, a `static` variable named `sharedVar` is initialized to the value of 0. The code starts by executing the `main` method, which begins at line 7. The MPJ Express library is initialized at line 8 using the `MPI.Init()` method. Later each process obtains its own identity in the `MPI.COMM_WORLD` communicator and stores it in the `rank` variable. This is followed by incrementing the shared variable `sharedVar` on line 10. Later each process prints the value of its rank and the value of the `static` variable `sharedVar` and finalizes the MPJ Express library.

If we execute the code in the cluster configuration, we observe the following output:

```
Proc <0>: sharedVar = <1>
Proc <1>: sharedVar = <1>
```

This is the correct and desired output. Here the `HelloBug` class is executed by two MPJ processes in a SPMD fashion. Both of these processes execute in a separate JVM and thus do not share the static variable `sharedVar`—for this reason both processes increment the variable first and print 1. This execution model is also depicted in the Figure 11a.

On the other hand, when the code is executed in the multicore configuration, the following output is observed:

```
Proc <0>: sharedVar = <1>
Proc <1>: sharedVar = <2>
```

This output is incorrect primarily because it is inconsistent with the cluster configuration output. In this case, the `Hellobug` class is executed by two threads—representing MPI processes—inside a single JVM. Since they are in a single JVM, both of them share the static variable `sharedVar`. This situation is depicted in the Figure 11b.

The issue of application-level `static` variable only introduces a certain kind of inconsistent behaviour. There are several data variables in the MPJ Express source code that must not be shared between processes. On the other hand, the multicore device itself relies on shared data structures for its normal operation—send and receive queues are examples of these. In this case, we need to

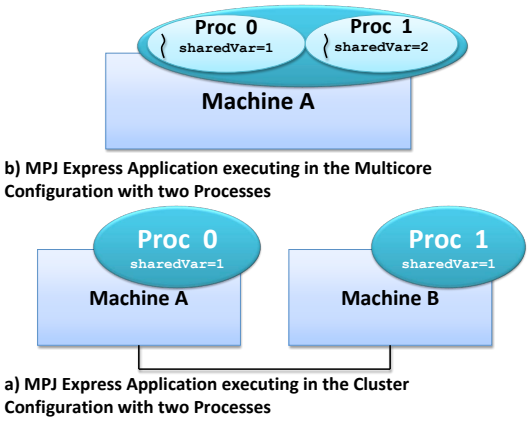

Figure 11. MPJ Express Application Running with two Processes in the Cluster and the Multicore Configuration

develop a new strategy where certain sections of the source code are shared and others are not.

We solved this issue by defining a custom class loader. But before diving into the details, we will briefly recap the basics of Java class loading.

The class loader is the means by which Java classes and resources are loaded into the JVM. In the Java language, class loaders have a hierarchical relationship and are organized in the form of a tree. Each class loader has a parent class loader except the bootstrap class loader. At the root of the tree lies the bootstrap class loader which loads core Java classes. It is followed by extension class loader which loads classes from the extension directories. Next in the hierarchy is the system class loader, which loads classes present on the `CLASSPATH` environment variable. At the leaves of the tree are the user-defined class loaders.

Now we shift our attention to the methodology used for fixing the issue of shared data variables. In the JVM, a class is typically identified by its full name and the class loader used to load it. Now imagine if we load two objects of the same class by two different user-defined class loaders, then these will be treated as two objects of distinct classes by the JVM. A side-effect of this is that any static variable in these objects will also not be shared.

There are some static variables in higher layers of MPJ Express that must not be shared by executing threads. On the other hand static variables and data structures in the lower layers must be shared because they are needed for shared memory communication implemented in the multicore device. So we devised a strategy of loading the classes in higher layers of MPJ Express through our user-defined class loader which we call as "thread-local class loader". This resulted in isolation of the static variables in these classes. On the other hand we loaded the classes in lower layers of MPJ Express through system class loader, which enabled us to use shared data structures for communication between threads. We achieved this by generating separate jars for the source classes of each layer of MPJ Express and dividing them into two groups.

1. *Group1*: this includes mpi, mpjdev and user-application classes.
2. *Group2*: this includes xdev, shmdev, mpjbuf and mpj-runtime classes.

The path of the jars of Group1 containing classes of user application and higher layers of MPJ Express is converted into a URL and passed to the constructor of thread-local class loader, which is a `URLClassLoader`. The thread-local class loader first delegates the class loading to its parent, which is the system class loader. Since these classes are not on the `CLASSPATH` so the parent fails to load the classes. Now the thread-local class loader searches the `URL`, opens the `JAR` files on the `URL` as needed and loads the classes. In this way the user application classes and the classes of higher layers of MPJ Express are loaded through thread-local class loader. On the other hand the jars of Group2 containing classes of lower layers of MPJ Express are placed on `CLASSPATH` so they are loaded by the system class loader.

5.2 Communication of Java Objects

We also encountered class loading issues while implementing communication of Java objects, which heavily relies on object serialization and de-serialization. Object serialization is the process of converting Java objects to a byte array and de-serialization is the process of converting a byte array to Java objects.

The MPJ Express software relies on the default serialization mechanism provided by the JDK for communication of objects. It makes extensive use of the buffering API to implement derived datatypes. The buffering API provides `write()` and `read()` methods. When the `Send()` method is called by the user, the message is first packed using the `write()` method onto a buffer that is used for communication by the underlying communication devices. Similarly, when receiving a message with the `Recv()` method, the `read()` method is called, which unpacks the message from `mpjbuf.Buffer` onto the user specified datatype. Sending and receiving objects using threads which have different class loader is not very simple and straightforward. In `write()` method the object is serialized by current class loader which is thread-local class loader but when we deserialize this object in `read()` method it is deserialized using the system class loader by default, using JDK standard `ObjectInputStream`. The JVM considers the deserialized object as different from the object that was sent despite the fact the both of these objects are created using the same class file. As a result `ClassNotFoundException` is thrown in `read()` method while deserializing.

To solve this issue we have created `CustomObjectInputStream` class which overrides the `resolveClass()` method of `ObjectInputStream` class. In this method we get the thread-local class loader of current thread which we initially set in `MultiThreadedStarter` class, using `getContextClassLoader()` method of `Thread` class. Then we load the class using thread-local class loader in `Class.forName()` method. The pseudocode for `CustomObjectInputStream` class is shown in the Figure 12.

6. Performance Evaluation

This section presents performance evaluation of the MPJ Express multicore configuration. We also compare the performance with other popular C and Java message passing systems.

For the purpose of evaluation, we use the following benchmarks or applications:

1. Ping Pong Benchmark for Basic Datatype
2. Ping Pong Benchmark for Indexed Datatype
3. Java Gadget-2 Application

For ping pong benchmarks used to evaluate performance of point-to-point communication, the test environment was a 32 processing core Linux cluster at NUST, Pakistan. The cluster consists of eight compute nodes. Each node contains a quad-core Intel Xeon processor. The nodes are connected via Myrinet and Gigabit Ethernet. The compute nodes run the SuSE Linux Enterprise Server (SLES) 10 Operating System and GNU C Compiler (GCC) version

```
1 public class CustomObjectInputStream extends
2                              ObjectInputStream {
3
4  public CustomObjectInputStream (InputStream in)
5    throws IOException {
6   super(in);
7  }
8
9  @Override
10 public Class resolveClass(ObjectStreamClass  desc)
11   throws IOException {
12
13  String name = desc.getName();
14  URLClassLoader ucl =
15      (URLClassLoader)Thread.currentThread()
16      .getContextClassLoader();
17
18  return Class.forName(name, false, u);
19 }
20 ...
21}
```

Figure 12. Psuedocode for the `CustomObjectInputStream` Class

4.1.0. Each compute node has 2 GBytes of main memory. We used MPICH2 version 1.2 as the C MPI library. For the parallel Java version, we used the latest development version of MPJ Express and MPJ/Ibis version 2.1 with the Sun Java Development Kit (JDK) 1.6 Update 12.

6.1 Ping Pong Benchmark for Basic Datatype

A ping pong benchmark is a reliable test for evaluating performance of message passing communication libraries. This particular test measures performance of blocking point-to-point communication methods, which are widely used in parallel applications. In this test, two parallel processes are started that repeatedly exchange messages of increasing size. For our experiments, we vary message size from 1 Byte to 8 Megabytes. At each particular data point, the experiment is repeated for 10,000 iterations and the average value is used for plotting results. Note that the two MPJ Express processes (or MPI processes for other libraries) are started on the same computational node of the cluster—we are interested in evaluating intra-node message passing communication.

The following message passing libraries are used for this particular test:

- MPICH2 using the nemesis subsystem (for shared memory communication),
- mpiJava using MPICH2,
- MPJ/Ibis,
- MPJ Express (cluster mode using niodev), and
- MPJ Express (multicore mode)

Figure 13 shows the transfer time comparison for message sizes from 1 Byte to 2 Kilobytes. Figure 14 shows throughput graph for message sizes from 4 Kilobytes to 8 Megabytes.

MPICH2 using nemesis shared memory communication achieves the lowest latency of 0.4 μs (for 1 Byte message). It is followed by mpiJava, which also has very low latency of 3 μs for 1 Byte message. The MPJ Express (multicore mode) also performs well outperforming MPJ Express (cluster mode) and MPJ/Ibis.

There are two curves representing MPJ Express in the multicore mode. The one labeled "MPJ Express (multicore mode)" is using MPJ Express buffering layer. The second labeled "MPJ Express (multicore mode without buffering)" is not using the buffering layer. The difference between the two curves depicts the packing and unpacking overhead incurred by the MPJ buffering layer.

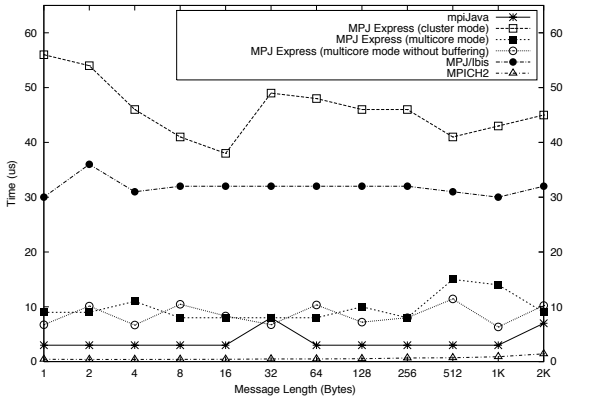

Figure 13. Transfer Time Comparison

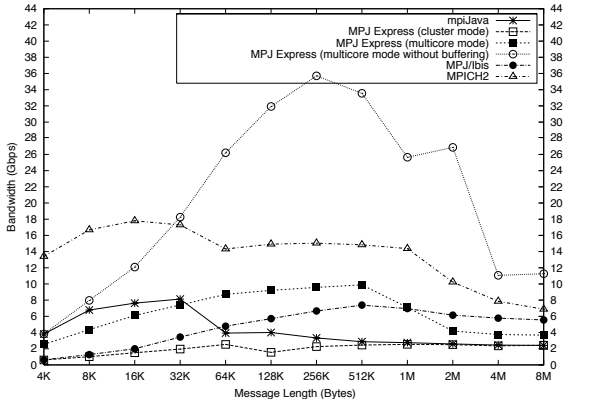

Figure 14. Throughput Comparison

The throughput graph shown in the Figure 14 shows that MPJ Express (multicore mode without buffering) performs the best. This is understandable because communication between threads is faster than all inter-process communication technologies. The maximum throughput achieved by MPJ Express (multicore mode without buffering) is 36.5 Gbps which is two times faster than MPICH2, four times faster than mpiJava and five times faster than MPJ/Ibis. MPJ Express (multicore mode)—that uses buffering layer—performs well until the 512K message size. The maximum throughput achieved in this configuration is 9.8 Gbps. But for messages larger than 512K the overhead of packing and unpacking becomes overwhelming and hence its performance drops abruptly. MPICH2 achieves a maximum throughput of 18.2 Gbps. The curve of mpiJava is very similar to MPICH2. It also performs well up to 32K message size and achieves throughput of 8.6 Gbps. After that the JNI data copying becomes a bottleneck and the throughput curve drops significantly. MPJ/Ibis is able to achieve the maximum throughput of 7.5 Gbps. In the end we have MPJ Express (cluster configuration) that achieves a maximum throughput of 2 Gbps.

The transfer time and throughput comparison results discussed in this subsection clearly show that MPJ Express (multicore mode) performs significantly better than MPJ Express (cluster mode). Also the buffering layer of MPJ Express remains a potential source of bottleneck in the cluster and the multicore configuration. This buffering layer is inevitable for implementing derived datatypes.

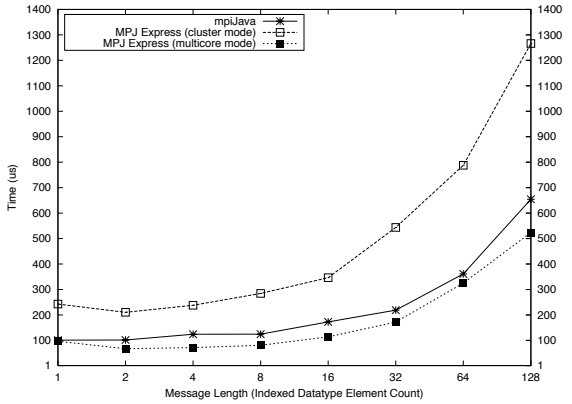

Figure 15. Transfer Time Comparison for Indexed Datatype

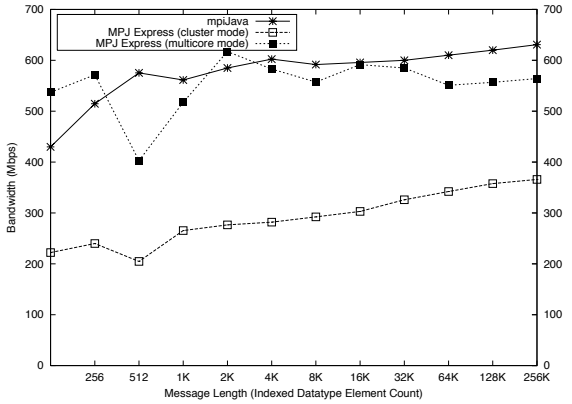

Figure 16. Throughput Comparison for Indexed Datatype

6.2 Indexed Datatype Benchmark

This subsection evaluates and compares the transfer time and throughput for communicating derived datatypes.

There are two schools of thoughts in the context of communicating non-contiguous data in Java messaging systems. The first is that Java objects could be used for this purpose. The second is that instead of Java objects, derived datatypes ought to be used. Although the MPJ Express software fully supports communication of Java objects, we firmly believe that JDK's default object serialization and de-serialization is slow and, for this reason, detrimental to the performance of a parallel application. On the other hand, MPJ/Ibis does not implement derived types—apart from contiguous—because developers of this system advocate [3] using Java objects instead.

In this subsection, our experiment uses the indexed datatype for performance evaluation. We will present results with MPJ Express and mpiJava because MPJ/Ibis does not implemented this particular datatype. The newly defined indexed datatype is built with `MPI.DOUBLE` datatype elements and initialized using block length array $[8, 7, 6, 5, 4, 3, 2, 1]$ and displacements array $[0, 1, 2, 3, 4, 5, 6, 7]$. This kind of data-structure might be used to communicate upper triangles during matrix operations.

Figure 15 and Figure 16 present the transfer time and throughput comparison of communicating our newly defined indexed datatype—the count is varied from 1 to 262144. These graphs show that MPJ Express (multicore mode) performs the best in terms of transfer time for small messages. mpiJava also performs reasonably well followed by MPJ Express (cluster mode). In terms of throughput achieved by relatively larger messages, mpiJava and MPJ Express (multicore mode) perform equally well. The overhead associated with MPJ Express (both multicore and cluster mode) is the additional packing and unpacking, which is inevitable for indexed datatype. But an additional performance penalty appears because our current implementation packs and unpacks individual element in all blocks in a separate method call. Since our storage medium is an instance of the `ByteBuffer` class, we can only use `putDouble(double element)` for this purpose. An alternative and efficient implementation would use bulk packing and unpacking method for all blocks—currently the Java NIO API does not support such an operation.

6.3 Java Gadget-2

Gadget-2 [16] is a free production code for cosmological N-body and hydrodynamic simulations. The code is written in the C language and parallelized using MPI. It simulates the evolution of very large, cosmological-scale systems under the influence of gravitational and hydrodynamic forces. The universe is modeled by a sufficiently large number of test particles, which may represent ordinary matter or dark matter. The main simulation loop increments time steps and drifts particles to the next time step. This involves calculating gravitational forces for each particle in the simulation and updating their accelerations. Its parallelization strategy is based on an irregular and dynamically adjusted domain decomposition, with copious communication between processors.

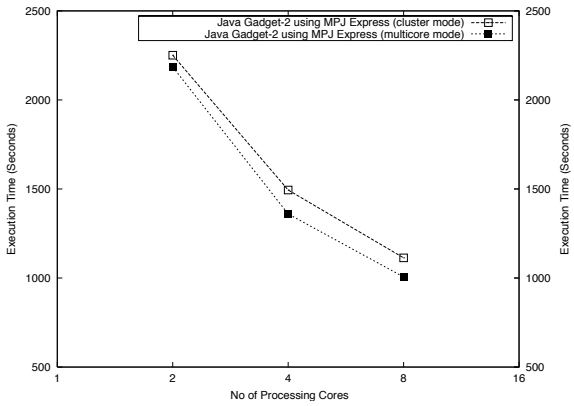

Figure 17. Execution Time for the Colliding Galaxies Simulation using Java Gadget-2 Application

In an earlier work [2], we developed a Java version of the Gadget-2 code. In this subsection, we evaluate and compare performance of this code for executing the colliding galaxies simulation using MPJ Express in the cluster and the multicore modes. Our goal is to quantify benefits of the MPJ Express multicore device using a real-world scientific application on a shared memory/multicore processor machine. The server used for this experiment contains two Intel Xeon E5404 quad-core processors operating at a clock speed of 2.0 GHz. Each CPU has two levels of cache: L1 and L2 having storage capacity of 64 Kilobyte (per core) and 6 Megabytes (per processor). We used the latest development version of the MPJ Express library.

Figure 17 shows the total execution time of the simulation on 2, 4, and 8 cores. As expected, the Java Gadget-2 code using MPJ Express (multicore mode) performs better. The performance gain appears modest because the communication cost is a small fraction of the overall execution time. Nonetheless, this shows that the multicore communication device helps MPJ Express achieve better performance in real-world applications.

7. Conclusions and Future Work

In this paper, we presented a new low-level multicore communication device to port existing legacy parallel Java applications on multicore and shared memory processors. The idea is that many scientific applications already exist that are parallelized using MPJ Express for distributed memory platforms like clusters. And in this work we provide an effective and efficient way of executing the same application—without any modification—on multicore and shared memory processors.

The multicore device for the MPJ Express software has been developed using Java threads. The device is recently integrated into the public release of the MPJ Express software after essential modifications to the runtime system to ensure correct semantics of the parallel program. The addition of the multicore device (and the multicore mode) to the MPJ Express software enables application developers to exploit the full potential of modern multicore machines.

Our performance evaluation reveals that MPJ Express (multicore mode) performs much better than MPJ Express (cluster mode) and MPJ/Ibis on multicore and shared memory processors machines. Especially throughput achieved by the multicore mode MPJ Express software is substantially higher than other competing libraries. The performance of the MPJ Express software still suffers from the overhead of the buffering layer, which is inevitable for implementing derived datatypes. We plan to explore in future if it is possible to avoid this layer for communication of basic datatypes. We also used MPJ Express (multicore mode) in a real-world simulation code called Gadget-2 on an eight core machine and compared its performance with MPJ Express (cluster mode). The multicore mode of the MPJ Express software performed better demonstrating effectiveness of our approach.

In the future, we plan to develop a hybrid MPJ Express device that uses multicore device for intra-node communication and message passing for inter-node communication—this kind of device will allow porting MPJ Express applications on clusters of multicore and shared memory processors. Currently the multicore mode can only be used on a single computer. We also plan to further improve performance of MPJ Express communication devices.

A free copy of the MPJ Express software can be obtained from `http://mpj-express.org`.

Acknowledgments

The authors would like to thank the British Council for a generous grant to support this work under the PMI2Connect programme.

References

[1] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. An Object-Oriented Java interface to MPI. In *Proceedings of the International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.

[2] M. Baker, B. Carpenter, and A. Shafi. MPJ Express Meets Gadget: Towards a Java Code for Cosmological Simulations. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 358–365, Bonn, Germany, September 2006. Springer.

[3] M. Bornemann, R. van Nieuwpoort, and T. Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 217–224. Springer, 2005.

[4] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, New York, NY, USA, 2001. ACM. ISBN 1-58113-359-6. doi: http://doi.acm.org/10.1145/376656.376823.

[5] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava: Data Parallel Extensions to Java. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

[6] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim. mpiJava 1.2: API Specification. Technical report, Northeast Parallel Architectures Center, Syracuse University, October 1999. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html.

[7] G. Fox. Editorial: Java for Computational Science and Engineering - Simulation and Modeling. *Concurrency: Practice and Experience*, 9 (6):413–414, June 1997.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. OpenMPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 97–104, Budapest, Hungary, September 2004. Springer.

[9] G.P.Nikishkov, Yu.G.Nikishkov, and V.V.Savchenko. Comparison of C and Java Performance in Finite Element Computations. *Computers and Structures*, 81(X):2401–2408, 2003.

[10] W. Gropp. MPICH2: A New Start for MPI Implementations. In D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Proceedings of the 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, page 7. Springer, October 2002. ISBN 3-540-44296-0.

[11] Java Grande. The Java Grande Forum Home Page. http://www.javagrande.org.

[12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.

[13] J. Moreira, S. Midkiff, and M. Gupta. From Flop to MegaFlops: Java for Technical Computing. In *Languages and Compilers for Parallel Computing*, volume 1656 of *Lecture Notes in Computer Science*. Springer, 1998.

[14] A. Shafi and J. Manzoor. Towards Efficient Shared Memory Communications in MPJ Express. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–7, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: http://dx.doi.org/10.1109/IPDPS.2009.5161083.

[15] A. Shafi, B. Carpenter, and M. Baker. Nested parallelism for multi-core HPC systems using Java. *J. Parallel Distrib. Comput.*, 69(6):532–545, 2009. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2009.02.006.

[16] V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105, 2005.

[17] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1095408.1095421.

[18] Top500. TOP500 Supercomputer Sites. http://www.top500.org.