University of Portsmouth
PORTSMOUTH
Hants
UNITED KINGDOM
PO1 2UP

This Article


Has been retrieved from the University of Portsmouth's Research Repository:

http://eprints.port.ac.uk

To contact the Research Repository Manager email:

ir@port.ac.uk
A comparative study of Java and C performance in two large-scale parallel applications

Aamir Shafi\textsuperscript{1,*}, Bryan Carpenter\textsuperscript{2}, Mark Baker\textsuperscript{3} and Aftab Hussain\textsuperscript{1,4}

\textsuperscript{1}National University of Sciences and Technology, SEECS Campus, Sector H-12, Islamabad, Pakistan
\textsuperscript{2}University of Southampton, Southampton, U.K.
\textsuperscript{3}University of Reading, Reading, U.K.
\textsuperscript{4}CERN, Switzerland

SUMMARY

In the 1990s the Message Passing Interface Forum defined MPI bindings for Fortran, C, and C++. With the success of MPI these relatively conservative languages have continued to dominate in the parallel computing community. There are compelling arguments in favour of more modern languages like Java. These include portability, better runtime error checking, modularity, and multi-threading. But these arguments have not converted many HPC programmers, perhaps due to the scarcity of full-scale scientific Java codes, and the lack of evidence for performance competitive with C or Fortran. This paper tries to redress this situation by porting two scientific applications to Java. Both of these applications are parallelized using our thread-safe Java messaging system—MPJ Express. The first application is the Gadget-2 code, which is a massively parallel structure formation code for cosmological simulations. The second application uses the finite-domain time-difference method for simulations in the area of computational electromagnetics. We evaluate and compare the performance of the Java and C versions of these two scientific applications, and demonstrate that the Java codes can achieve performance comparable with legacy applications written in conventional HPC languages. Copyright © 2009 John Wiley & Sons, Ltd.

Received 28 April 2008; Revised 5 January 2009; Accepted 6 January 2009

KEY WORDS: MPJ Express; Java for HPC; mpiJava; MPJ; Parallel Java

*Correspondence to: Aamir Shafi, National University of Sciences and Technology, SEECS Campus, Sector H-12, Islamabad, Pakistan.
†E-mail: aamir.shafi@niit.edu.pk

Copyright © 2009 John Wiley & Sons, Ltd.
A COMPARATIVE STUDY OF JAVA AND C PERFORMANCE

1. INTRODUCTION

Almost immediately following its release in 1996 Java became a ‘mainstream’ programming language of the software industry. Compared with C or Fortran, the advantages of the Java programming language include higher-level programming concepts, improved compile time and runtime checking, and, as a result, faster problem detection and debugging. In addition, Java’s automatic garbage collection, when exploited carefully, relieves the programmer of many of the pitfalls of lower-level languages. 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.

A highly attractive feature of applications written in Java is that they are portable to any hardware or operating system, provided that there is a Java Virtual Machine (JVM) for that system. The contribution of the JVM is significant, keeping in mind that it allows programmers to focus on issues related to their application and domain of interest, and not on system heterogeneity.

Various computer scientists have argued\[1–7\] that Java could make an excellent language for developing scientific codes. To date this argument has not convinced too many practising computational scientists. The scarcity of high-profile, number-crunching codes implemented in Java does not help the case.

In its very early days, Java was an interpreted language, and performance lagged orders of magnitude behind, say, C. This initial strategy was soon replaced in JVMs that incorporated Just In Time (JIT) compilers. In modern JVMs like the HotSpot machine from Sun Microsystems, byte code for Java classes may still be interpreted when it is initially loaded. But runtime analysis rapidly identifies code sections that are most critical; such sections are compiled to machine code and may be adaptively optimized. Potentially this strategy can produce excellent performance, by focussing compiler attention on the most frequently executed sections of code—something that can only be accurately determined at runtime.

The focus of this paper will be on parallel applications. Out of numerous proposals and standards for parallel programming, the Message Passing Interface (MPI)\[8\] has been the most successful. To enable High Performance Computing (HPC) using Java, we developed and released MPJ Express\[9\]. This is a thread-safe Java messaging system, closely modelled on the MPI standard. It can be used to write parallel high performance applications in Java.

The performance of a parallel application depends on the performance of serial code on each processor and the performance of communication library. In this context, the relative performance of a parallel application written in C using an MPI library and the same application written in Java using MPJ Express will depend on two main factors. The first is the relative performance of the serial C and Java code. The second major factor is the relative performance of messaging libraries, which directly influences the communication cost.

To assess concerns about performance of Java for HPC, we describe our experiences porting two scientific applications to Java, and parallelizing them using MPJ Express. The Java versions were developed as an experiment to help understand where Java stands in comparison with C—a well-established HPC language.
Our first experiment involved developing a Java version [10] of the massively parallel cosmological simulation code Gadget-2 [11]. Versions of the C Gadget-2 code have been used in the ‘Millennium Simulation’ [12], which was heralded as the largest ever simulation of the Universe. It evolved $10^{10}$ dark matter particles from the early Universe to the current day.

In our second experiment we developed a parallel version of the Finite-Difference Time-Domain (FDTD) method in Java [13]. Again we used MPJ Express to parallelize the Java FDTD code. The FDTD method is a widely used and increasingly popular way of studying electromagnetic wave propagation. It has been successfully applied to a broad range of applications, including antenna design, radar, and photonic crystals. In addition, there are emerging applications of the FDTD method in areas such as biophotonics and nanophotonics [14].

We evaluate and compare the performance of the Java and C versions of the Gadget-2 and the FDTD code. We find that on popular HPC platforms the Java versions often match the performance of C codes. This reinforces our belief that Java is a viable option for HPC.

The remainder of the paper is organized as follows: Section 2 introduces MPJ Express. Section 3 presents an overview of Gadget-2 and goes on to discuss our experiences in porting Gadget-2 to Java. Section 4 presents the FDTD algorithm and its parallel implementation. Sections 5 and 6 evaluate and compare the performance of Gadget-2 and FDTD applications, respectively. We discuss related work in Section 7 and conclude the paper in Section 8.

2. OVERVIEW OF MPJ EXPRESS

Early interest in using Java for HPC led to the formation of the Java Grande Forum [15]—a vehicle for the HPC community to voice their opinion about Java and how to improve it. The Java Grande Forum was a group of leading researchers from academia and industry that aimed to exploit Java for Grande, or large-scale, applications. The forum proposed various improvements to Java for numerical computing—some of these were introduced into the standard Java language. The Message Passing Working Group of the forum defined Java bindings for the MPI standard, including the MPJ API [16].

MPJ Express is our implementation of Java bindings for the MPI standard. The system provides thread-safe communication in a Java messaging system. It addresses potentially contradictory issues of performance and portability by supporting pluggable transport devices based either on Java NIO (pure Java) or high performance interconnects like Myrinet. Other implementations of MPI Java bindings include mpiJava [17,18] and MPJ/Ibis [19].

There is a growing community of users who are developing their parallel applications using MPJ Express. An example of a scientific library using MPJ Express is Cartablanca [20], which is an object-oriented physical system simulation package. The code uses Jacobian-free Newton-Krylov [21] methods to solve non-linear physics simulations on unstructured meshes. Researchers at the University of Leeds have used [22] MPJ Express in an Economic and Social Research Council funded project called Modelling and Simulation in e-Social Science.

This section starts with a discussion about MPJ Express design, including the communication functionality provided by the Java NIO and Myrinet devices. This is followed by a brief introduction to buffer management in MPJ Express—a feature that enables explicit memory management instead
of relying on Java’s garbage collector to achieve optimal performance. In addition, we discuss the portable runtime system that is used to bootstrap MPJ Express processes. For brevity, we do not discuss the implementation details of higher-level MPI features like collective communications, derived datatypes, and virtual topologies.

2.1. MPJ Express design

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 for portability. Figure 1 illustrates the MPJ Express design and the different levels of the software: the MPJ API, high level, base level, mpjdev, and xdev.

The topmost MPJ API layer in Figure 1 represents the system’s full API. The next two layers in MPJ Express design are high level and base level, representing the collective and point-to-point communications, respectively. These layers rely on the mpjdev and xdev levels for actual communications and interaction with the underlying networking hardware.

The point-to-point or base level is implemented on top of mpjdev—the MPJ Device layer [23]. We illustrate two implementations of the mpjdev level. The first implementation labelled the pure Java mpjdev uses the lower level device called xdev to provide access to Java sockets or specialized communication libraries.
The pure Java implementation of mpjdev relies upon implementations of the xdev device layer. Figure 1 shows this layer. The reason for introducing a second device layer in MPJ Express design is to reduce the development time and effort for writing new mpjdev communication drivers. Unlike the mpjdev layer, xdev only provides communication-related methods and is not aware of higher-level MPI abstractions like communicators.

2.2. Communication devices and functionality

The current release of MPJ Express provides two transport-level devices to implement the basic point-to-point messaging. There is a Java NIO-based device called niodev and Myrinet eXpress-(MX) based device called mxdev. We now briefly discuss these two communication devices. The user level functions of the MPJ Express software, like the point-to-point and collective communication layers, are implemented on top of these devices.

2.2.1. niodev: The Java NIO device driver

The implementation of the xdev device layer that provides communication through the Java NIO package is called niodev. Java NIO was introduced in Java 1.4. The older java.io package did not support non-blocking I/O. Implementing MPI-like non-blocking communication in terms of the old I/O package involved starting a new thread, which is a relatively expensive operation, especially if a separate ‘listener’ thread is needed for every peer process in a large cluster. The Java NIO package solves this problem by providing non-blocking communication through an abstraction called a socket channel. These channels register with a selector, which yields read, write, accept, and connect events. This concept is similar to the select() system call in UNIX, which helps provide scalable and efficient I/O. With this functionality we can implement MPI communication using a small constant number of Java threads in each node.

The NIO socket channels provide variants of read() and write() methods that send and receive messages to and from byte buffers. The byte buffer is another important new abstraction in NIO. A byte buffer can be either direct or indirect. A direct buffer is a chunk of memory that is allocated in the native operating system. Unlike conventional Java objects, the storage does not reside on the JVM heap and the JVM only maintains a reference to the native memory region. An indirect buffer is like any Java object that is created on the JVM heap. Our experiments [24] with direct and indirect byte buffers showed that the former provide faster I/O but at higher creation cost. For devices that use JNI, direct byte buffers are desirable because they make it possible to avoid some of the data copying overheads introduced by JNI. To implement various send modes defined by the MPI standard document, the niodev implements two communication protocols—eager send and rendezvous. A detailed discussion of these communication protocols can be found in [9].

2.2.2. mxdev: The Myrinet device driver

In mid 2005, Myricom—the manufacturer of Myrinet—introduced a new low-level communication software called MX. The MX library has various advantages over the historical GM library. One is that methods like non-blocking send and receive are thread-safe. Another is that the xdev API closely matches the MX API, resulting in a simple mapping of xdev to MX methods. In fact the
idea for one of the methods in xdev—peek()—was adopted from the MX library. Third, the MX library implements some higher-level functionality like buffered and synchronous modes. In addition, the send method automatically registers the memory region for Direct Memory Access (DMA) transfers unlike the GM library where the users had to explicitly register and de-register memory.

Our communication device called mxdev uses JNI to interface with the MX library. Unlike niodev, mxdev does not directly implement any communication protocols, because the MX library has internally implemented these protocols. Because our buffering API mpjbuf can use direct byte buffers, we have been able to avoid one of the main overheads of using JNI—copying data between the JVM and the OS.

2.3. Buffer management in MPJ Express

One challenge in implementing Java HPC messaging software is providing an efficient intermediate buffering layer. The low-level communication devices and higher levels of the messaging software use this buffering layer to write and read messages. The heterogeneity of the low-level communication devices poses further design challenges.

To appreciate this fully, imagine that the user of a messaging library sends 10 elements of an integer array. The C programming language can retrieve the memory address of this array and pass it to the underlying communication device. If the communication device is based on Transmission Control Protocol, it can then pass this address to the socket’s write() method. For proprietary networks, like Myrinet, this memory region can be registered for DMA transfers, or copied to a DMA capable part of memory and sent using low-level Myrinet communication methods.

On the other hand, if the user of a Java messaging system based on the NIO package sends an array of 10 integers, these must first be copied to an NIO byte buffer that is used as an argument to the socket channel write() method. It may in fact be possible to avoid any buffering (outside the OS itself) if the communication device uses the traditional java.io package, where the JVM may be able to write the user’s Java array directly to the output stream of the native socket. In simple cases of point-to-point communication involving simple Java arrays this occasionally leads to better performance than with NIO, because there is less copying. But java.io does not support the non-blocking I/O features provided by the Java NIO package, and trying to use the java.io API where it was advantageous would make implementation of the general features of MPI very much more complicated. Thus, in xdev we commit to an explicit buffering API—introducing buffer interfaces that may be implemented naturally in terms of NIO byte buffers, or by other means.

We have designed an extensible buffering layer for MPJ Express called mpjbuf. This buffering layer allows various implementations based on different storage medias like direct or indirect NIO byte buffers, byte arrays, or memory allocated in the native C code. The higher levels of MPJ Express use the mpjbuf buffering layer through an interface. This implies that functionality is not tightly coupled to the storage medium. The motivation behind developing different implementations of buffers is to achieve the best practical performance for lower-level communication devices.

The MPJ Express buffering layer implements its own application level memory management mechanism based on Knuth’s buddy algorithm [25]. The motivation is to avoid creating an instance of a buffer (mpjbuf.Buffer) for every communication operations like Send() orRecv().
which may dominate the total communication cost, especially for large messages. We can make efficient use of resources by pooling buffers for future reuse. We prefer such pooling instead of letting the garbage collector reclaim the buffers and create them all over again. We discovered by experiments, presented in [26], that such constant creation or destruction of buffers had detrimental effects on the performance of Java messaging applications.

2.4. The runtime system

An important component of an MPI-like messaging system is the mechanism used to bootstrap processes across various platforms. A challenge here is to make the mechanism cope with heterogeneous platforms. If the compute nodes are running a UNIX-based OS, it is usually possible to remotely execute commands using the ssh command. But if the compute nodes are running Microsoft Windows, these utilities are not universally available.

The MPJ Express runtime provides a unified way of starting Java processes on compute nodes irrespective of the operating system. The runtime system consists of two modules. The daemon module executes on compute nodes and listens for requests to start MPJ Express processes. The daemon is a Java application listening on an IP port, which starts a new JVM whenever there is a request to execute an MPJ Express process. The mpjrun module acts as a client to the daemon module. This module may be invoked on, for example, a cluster’s head-node. It will contact daemons, which will start MPJ Express processes in a new JVM. MPJ Express uses the Java Service Wrapper Project [27] software to install daemons as a native OS service. It is possible to run the daemons in a multi-user environment, though the users will have to agree to use distinct ports—port numbers can be configured at runtime. The distribution of MPJ Express provides scripts for Windows and Linux that can be used to start the daemon services on compute nodes.

The MPJ Express runtime allows running MPJ Express applications either remotely or locally. With the remote loader, it is possible to load all classes (application and MPJ Express code) from the user’s development node. This is useful in scenarios when there is no shared file system and the code is constantly being modified at the head-node. With the local loader, it is possible to load all classes (application and MPJ Express code) from the compute node. This might be useful if there is a shared file system. As all classes are loaded locally, this might provide better performance in comparison with the remote loader.

2.5. Collective communications

The MPI specification provides collective communication operations as a convenience to application developers. Using these methods can save significant development time. These operations include Broadcast(), Barrier(), Reduce(), Allreduce(), Gather(), Allgather(), Alltoall(), Scatter(), Scan(), and Allscatter(). In addition, there are versions of these methods like Allgatherv() that allow displacements between individual elements of communication data.

In MPJ Express, the collective methods are implemented on top of point-to-point communication. We will discuss here the implementation of two collective methods: Barrier() and Broadcast().

The `Barrier()` method synchronizes all the calling processes. There are three phases in barrier algorithms: initialization, check-in, and notification. Algorithms using shared memory can omit the notification phase.

The first implementation we consider is based on a tree algorithm. A message is propagated from the root of the tree to the leaves and back. We also implemented a pair-wise exchange with recursive doubling algorithm [28] to implement the `Barrier()` functionality. This algorithm avoids the worst-case scenario of the Butterfly algorithm [28].

The `Broadcast()` method is used to send data from a process to all the other processes. The current implementation uses an n-ary tree that is generated dynamically.

### 2.6. Performance evaluation

This subsection evaluates the messaging performance of MPJ Express and compares it with other C and Java messaging libraries. Here we only present the latency and throughput measured using a simple ping-pong test over Myrinet.

The test environment was a cluster run by the Distributed Systems Group at the University of Portsmouth, consisting of eight dual Intel Xeon 2.8 GHz nodes using the Intel E7501 chipset. The nodes were equipped with 2 Gbytes of ECC RAM with 533 MHz Front Side Bus, and connected via a 2 Gbit Myrinet network. The nodes were running the Debian GNU/Linux with the 2.4.32 Linux kernel. The JDK version used for mpiJava and MPJ Express is Sun JDK 1.5 (Update 6). The C compiler used was GNU GCC 3.3.5. The version of MX library was 1.1.0. In addition, we used MPICH-MX version 1.2.6.0.94, which runs on top of the MX library.

Figures 2 and 3 show the transfer time and throughput comparison. The latency of MPICH-MX is 4 μs. MPJ Express and mpiJava have latency of 23 and 12 μs, respectively. Throughput achieved by MPICH-MX is 1800 Mbps for 16 Mbytes messages. MPJ Express achieves 1097 Mbps for the same message size. The overhead is caused by packing and unpacking of the message data.
mpiJava achieves a maximum of 1347 Mbps for 64 kbyte messages. After this, there is a drop, bringing throughput down to 868 Mbps for 16 Mbyte messages. Note that mpjdev alone achieves 1826 Mbps using mxdev on Myrinet for 16 Mbyte messages.

The reason for higher latency of MPJ Express is a combination of the use of thread-safe algorithms and additional copying done on the user data. As described in Section 2.3, MPJ Express uses an intermediate buffering layer. This implies additional copying. Using Java over proprietary networks like Myrinet, this overhead seems hard to avoid without extending the MPJ Express API to support communication to and from byte buffers. Typical JVMs copy arrays when they are passed through the JNI interface. We use NIO direct byte buffers as the storage mechanism for buffering layer. It is possible to get pointers to these buffers from native C code. This helps avoid further copying of data between the JVM heap and JNI code, but it is still necessary to shift data between user arrays and these buffers. When using mpjdev directly, the data have already been copied onto a direct byte buffer, hence, the difference between the performances of MPJ Express and mpjdev shows the overhead of packing (at sender) and unpacking (at receiver). The throughput achieved by MPJ Express is limited by the additional intermediate packing and unpacking required at the sender and the receiver, respectively.

3. GADGET-2

Gadget-2 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 modelled 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.
A COMPARATIVE STUDY OF JAVA AND C PERFORMANCE

To give some feeling for the scale of interesting problems, consider the so-called ‘Millennium Simulation’ [12]. This simulation follows the evolution of $10^{10}$ dark matter particles from the early Universe to the current day. It was performed on 512 processors and used 1 Terabyte of distributed memory. The simulation used 350,000 CPU hours over 28 days of elapsed time. It used an adapted version of Gadget-2.

One of the main tasks of a structure formation code is to calculate gravitational forces exerted on a particle. As gravity is a long-range force, every particle in the system exerts gravitational force on every other particle. A naïve summation approach costs $O(N^2)$, which is not feasible for the scale of problems that Gadget-2 aims to solve. To deal with this, Gadget-2 can use either of two efficient algorithms, the first is the Barnes–Hut (BH) [29] oct-tree and the second is a hybrid of the BH tree and a Particle Mesh (PM) method called TreePM. We restrict our attention to the pure BH tree algorithm.

Being a massively parallel code, Gadget-2 needs to divide space or the particle set into domains, where each domain is assigned to a single processor. This is particularly challenging in Gadget-2 because it is not practical to divide space evenly. That would result in poor load balancing because some regions have more particles than the others. Conversely, it is also not possible to divide particles evenly in a fixed way because they move throughout space and it is desirable as well as practical to keep physically close particles on the same processor.

To solve this, Gadget-2 uses a space-filling Peano-Hilbert curve originally suggested by Warren and Salmon [30].

Rest of this section discusses our Java implementation of the Gadget-2 code. This is followed by presenting hand optimizations that were applied to the Java code. We also present a performance graph, which quantifies the performance improvement in the optimized code.

3.1. Porting Gadget-2 to Java

The original C version of Gadget-2 was manually translated to the Java language. The C Gadget-2 code has approximately 18 thousand lines of code. It took one developer approximately 4 months to port and optimize the Java version.

An important design principle was to use standard Java and not to propose or use language extensions. The main rationale behind this is that we would like to evaluate Java and comment on its feasibility for common HPC programmers and users.

We deliberately kept similar data structures in the translated version, so that we could cross-reference the original source code for debugging. While doing so we found that very efficient C data structures could easily, or naively, be translated to inefficient Java data structures. An example of this could be an array of structures in the C language. Typically, this would be translated into an array of objects in the Java language. This would result in inefficient cache utilization as Java objects in the array could be scattered anywhere in the main memory. We discuss tackling this particular issue in Section 3.2.2.

There are three dependencies for Gadget-2: the GNU Scientific Library (GSL), a parallel version of Fastest Fourier Transform in the West (FFTW), and of course an MPI library. It was straightforward to handle the last dependency—we of course used MPJ Express for point-to-point and collective communication in Gadget-2. But resolving dependencies on numerical libraries like the GSL and FFTW is not simple. The main reason is that there are no robust Java libraries or bindings for GSL and FFTW.
The Gadget-2 code uses a handful of GSL functions. These include performing numerical integration to calculate particles drift and generating random numbers. We used the `Math.random()` method for generating random numbers. Other required methods from GSL were translated to Java.

For the other dependency, it was not feasible to produce a Java version of FFTW. This results in one functional limitation of our Java code. The Java version only provides the option of using BH oct-tree for calculating gravitational forces. The TreePM algorithm, which is an alternative to BH oct-tree, is not implemented in the current Java version, because it depends on FFTW.

The biggest simulation that we carried out with the Java version contained 56 million particles on 16 nodes—each MPJ Express process contained roughly 3.5 million particles. The Millennium Simulation contained approximately 20 million particles on each node. The C version is still more memory efficient than the Java version—the latter takes three times more memory. We aim to explore this further in the future.

### 3.2. Hand optimizations in the Java Gadget-2 version

Evaluation of our initial Java translation revealed that the performance was approximately three times slower than the C version.

The principal optimization applied to improve the performance of the final version are discussed in this subsection.

#### 3.2.1. Custom serialization and deserialization

Our initial version of Java Gadget-2 communicated Java objects. This exploits the JDK default serialization and de-serialization mechanism made available by MPJ Express. Object serialization and de-serialization is the process of converting Java objects to a byte array and vice versa. It can have detrimental effects on the performance of a parallel application. Thus, we decided to replace Java object communication in Java Gadget-2 with primitive datatypes.

In the original C Gadget-2, initial conditions are read into an array of C structs called `ParticleData`. In the original Java version, this array is replaced by an object array called `ParticleData`. In the original C version, particles that need to be exported are copied to a contiguous memory region called `CommBuffer`. Initially we replaced this with a `CommBuffer` object, which contained object arrays. Before the communication operation, the data were copied from the `ParticleData` array onto a related object array in `CommBuffer` object.

In the optimized version of Java Gadget-2, the `CommBuffer` object is replaced by a contiguous memory—an instance of the `ByteBuffer` class. Before the actual communication, we copy primitive data from each element of `ParticleData` array to `CommBuffer`. Once all the data have been packed onto this `ByteBuffer`, it is communicated to the receiver process. The receiver process receives the data in `CommBuffer`, and unpacks it onto the `ParticleData` object array. This technique not only helped us to avoid the Java object serialization overhead, but also reduced the memory footprint of the JVM by 60%.

Currently the MPJ Express library does not support communication to or from `ByteBuffers` directly. For this reason, the data stored on a `ByteBuffer` have to be copied onto a byte array before specified as an argument to `Send()` or `Recv()` method. Naturally this implies additional
copying. In the Java Gadget-2 code, we have used indirect wrapped ByteBuffers to avoid this overhead. Such ByteBuffers are initialized using the ByteBuffer.wrap() that takes a byte array as an argument. Any changes in the ByteBuffer can be seen in the byte array and vice versa. We are considering changing the MPJ Express API to add support for communicating data to and from ByteBuffers.

3.2.2. Maintaining memory locality

Arguably it is harder to maintain memory locality in Java HPC applications than their C counterparts. The C structs and arrays are laid out at contiguous locations in the OS virtual memory. However, there is no guarantee of storage structure of Java objects, which are also subject to garbage collection at the Virtual Machine level. In particular, for arrays of objects, there is no reason to expect that objects that are consecutive in the array will be adjacent in physical memory.

In the Java version of Gadget-2, we made an effort to maintain memory locality by flattening sensitive data structures. Using this technique, we replaced Java object arrays with primitive datatype arrays. For example, BH tree-nodes are stored in an array of Java objects called Nodes_base. Each element of this array has members like an array of doubles called center and a double called len that represents the side length of a tree-node. In the Java version, these two members center and len are stored in a doubles array. This ensures that when a particular tree-node is accessed, all members of particular object element in Nodes_base array are in close vicinity in the memory.

We also flattened the ParticleData array, where each object has attributes like a three element array of pos and vel representing position and velocities in three dimensions. In addition, we also flattened the TopNodes array.

Although flattening objects array has helped to improve the performance, this technique has deteriorated the readability of the code. In the future we plan to use Perl templates to generate Java code to tackle this problem.

3.2.3. Avoiding expensive array operations

Arrays form the building blocks of many HPC applications. Our approach of flattening arrays in the Java version resulted in additional copying of array elements in some sections of the code. For example, during the tree walk, the C code copies an element by assigning a pointer to tree-node number no as follows:

nop = &Nodes_base[no];

This behaviour could be mimicked in the Java version if Nodes_base is an array of objects. But as explained in Section 3.2.1 we have replaced object array with primitive arrays containing their contents to make the best possible use of the OS virtual memory and processor cache. As a result, the only option to copy an element of Nodes_base is to store the individual components in temporary arrays. The Nodes_base represents a tree-node that can possibly have eight daughter nodes. References to these daughter nodes are stored in an integer array with eight elements. Each element of the daughter nodes array stores −1 if the tree-node is a leaf, or the index of the daughter otherwise. Our initial versions of Java Gadget-2 manually copied each element of the daughters array. Further investigations showed that the daughter nodes were not required throughout the tree
walk and therefore it was not required to copy the elements of the daughter array. This produced a speed-up of 15–20% in the tree walk.

The JDK includes a utility class called java.util.Arrays that provides useful methods like Arrays.fill(), Arrays.sort(), Arrays.binarySearch(). Another useful utility is the System.arraycopy() method, which provides a fast way to copy arrays. These methods should be used wherever possible to ensure optimal performance.

Java arrays are still somewhat different from C arrays. One of the costliest differences is that individual array elements are initialized when a Java array is created. It might also be useful for experimental purposes if the JVM supported turning off array bounds checking.

3.2.4. Quantify the performance gain of Java optimizations

The execution time comparison of the initial and final Java Gadget-2 code is shown in Figure 4. This graph quantifies the performance improvement of the optimized code. We analyse these results and discuss them in detail in Section 5.

4. JAVA FDTD METHOD FOR COMPUTATIONAL ELECTROMAGNETICS

This section discusses the Java implementation of the FDTD method for solving these equations, and the simulation that we carried out using this numerical technique. We start by discussing the basics of the FDTD method. We later introduce our own implementation technique and evaluate its performance.

This study of the FDTD method stems from a freely available simulation software package developed by researchers at the Massachusetts Institute of Technology (MIT) called the MIT
Electromagnetic Equation Propagation (MEEP) [31]. This software is based on the FDTD method to model and analyse electromagnetic systems such as Photonic Crystals through Time Domain electromagnetic simulation. MEEP uses a novel subpixel dielectric averaging technique for dealing with dielectric materials described in [31].

Different levels of subtle complexities and dependencies employed in MEEP lead us to develop our own parallel versions of the general FDTD algorithm for this paper. On other frontiers, we are still using MEEP to study wave propagation in photonic crystals.

4.1. The Finite-Domain Time-Difference method

Maxwell’s curl equations for the electromagnetic field in a sourceless and homogeneous medium are:

\[
\nabla \times \vec{E} = -\mu \frac{\partial \vec{H}}{\partial t} \tag{1}
\]

\[
\nabla \times \vec{H} = \varepsilon \frac{\partial \vec{E}}{\partial t} \tag{2}
\]

where \(\varepsilon\) is the electrical permittivity and \(\mu\) is the magnetic permeability. Equations (1)–(2) model the 3D wave propagation of electromagnetic waves. Electromagnetic waves have various propagation modes, one of which is the Transverse Magnetic (TM) mode. In this mode the magnetic field components are transverse or perpendicular to electric field components and to the direction of propagation of electromagnetic wave. In this paper we only consider modelling the TM mode propagation of a 2D electromagnetic wave.

The numerical foundation of the FDTD method is the discretization of Maxwell’s curl equations through the use of central difference approximations to both the space and time differentials. These discretizations result in coupled finite difference equations that govern the propagation of electric and magnetic fields on a discrete numerical grid. With appropriate boundary conditions these difference equations are then iterated in a time and space marching sequence to study the propagation of electromagnetic waves.

The discretized coupled finite difference equations evolve the electromagnetic wave in both the time and the space on a staggered FDTD grid. This evolution of the electromagnetic wave is an iterative process. In this scenario the coupled difference equations act as engines of evolution. The future values for any field component are calculated based on the past values of the other field components and the field component itself. For instance, for the calculation of electric field components we require past electric and magnetic field values. This process is repeated on the whole computational space and for a particular time period.

For simple and homogeneous media, the FDTD method models the propagation of electromagnetic energy on the discrete computational FDTD grid, such that in one simulation time step the electromagnetic energy moves one spatial step. But for complex media the modelling of correct physical phenomena in the medium dictates that electromagnetic energy moves according to the material properties of the medium. This can lead to finer spatial resolution of the FDTD grid, which results in increased simulation time. Similarly, if we study electromagnetic wave propagation in a complex geometrical structure, we must make the spatial grid finer to capture the fine detail.
of the structure. However, if we make the spatial steps smaller, there is a restriction called the Courant stability criterion [14], which dictates that one must set the time step small enough to obtain meaningful simulation results.

4.2. Implementation of the Java FDTD method

The FDTD method is amenable to parallelization using the message passing paradigm. The reason is that the update of the field values on the FDTD computational grid are dependent on the immediate neighbouring field values.

In the parallel version, the domain decomposition is achieved by dividing the computational grid equally among the processors—this allows each processor to concurrently execute its computational domain. For instance, imagine a computational grid of size\(_x\) and size\(_y\) grid points in the \(x\) and \(y\) directions, respectively. With \(P\) processes, the main computational grid can be equally divided into \((\text{size}_x \times \text{size}_y)/P\) sub-grids.

While updating the boundary electric and magnetic field data structures, each process requires field values from its neighbouring process. For this purpose, processes exchange field values at the boundaries. This exchange of values can be done by communicating the respective field values in a point by point manner. This kind of fine grain communication normally leads to an excessive communication cost. In order to avoid this, the entire column of ghost values—the required column on neighboring processor—is exchanged between the processes leading to less communication cost. In our implementation, the ghost values are communicated using non-blocking point-to-point MPI functionality.

Figure 5 shows the pseudocode for MPI version of the FDTD code.

The Java programming language supports multidimensional arrays by implementing these as ‘arrays of arrays’. This lack of direct support for multidimensional arrays results in significant performance penalty for scientific Java applications. Moreira et al. [32] extended a Java compiler to support C-like multidimensional arrays. In the parallel C FDTD code magnetic field components \(H_x\) and \(H_y\) and electric field component \(E_z\) are implemented using two-dimensional arrays. However in the Java code, we map these two-dimensional arrays to a single-dimensional array to better exploit processor cache by maintaining memory locality.

4.2.1. Cavity resonator simulation using the parallel FDTD method

Our parallel codes are based on the sequential FDTD code given in [33]. The simulation is of a Ricker Wavelet propagating in free space surrounded by Perfectly Electrically Conducting (PEC) walls, which reflect impinging electromagnetic waves.

The computational problem domain consists of an \(8192\times8192\) grid. The source is a Ricker Wavelet, which is a pulse-like waveform. It is equal to the second derivative of a Gaussian. Pulse-like sources are used in FDTD simulations when it is desired to excite a structure with a broad range of frequencies. The Ricker Wavelet emerges from the middle of the computational grid and propagates towards the walls of the cavity resonator. When it reaches the walls of the grid, it is reflected back because of the PEC boundary conditions. As the simulation proceeds, the Ricker Wavelet is confined to propagate within the resonator and we observe interference patterns generated by the confined waves.
Hx, the magnetic field component along x axis
Hy, the magnetic field component along y axis
Ez, the electric field component
rank, process rank in the world communicator
size, total number of processes

initialize field components

for initial time:maxtime {

    // Ez communication - part 1
    if(rank != 0) {
        Asynchronously send Ez edge column to process rank-1
    } else if(rank != size-1) {
        Asynchronously receive Ez edge column from process rank+1
    }

    // Hx computation
    for (i=1 to imax) {
        for (j=1 to jmax) {
            Hx(i,j) = Hx(i,j) - (Ez(i,j+1) - Ez(i,j))
        }
    }

    // Ez communication - part 2
    Wait for the Ez edge column communication to complete

    // Hy computation
    for (i=1 to imax) {
        for (j=1 to jmax) {
            Hy(i,j) = Hy(i,j) + (Ez(i+1,j) - Ez(i,j))
        }
    }

    // Hy communication
    if(rank != size-1) {
        Synchronously send Hy edge column to process rank+1
    } else if(rank !=0) {
        Synchronously receive Hy edge column from process rank-1
    }

    // Ez computation
    for (i=1 to imax) {
        for (j=1 to jmax) {
            Ez(i,j) = Ez(i,j) + (Hy(i,j) - Hy(i-1,j) - (Hx(i,j) - Hx(i,j-1))
        }
    }
}

Figure 5. Pseudocode for the parallel FDTD method.
5. PERFORMANCE EVALUATION OF GADGET-2

This section presents performance evaluations of the Java and C Gadget-2 versions of the code. We present results for various simulations including the Colliding Galaxies and the Cluster Formation simulation.

Our aim here is to confirm that Java with MPJ Express can achieve competitive performance in a real-world application.

5.1. Colliding galaxies simulation

This subsection evaluates the performance of the Java and C Gadget-2 code by using the initial conditions—consisting of 60,000 particles—of the Colliding Galaxies. These comparisons were performed on the Starbug cluster at the Distributed Systems Group in Portsmouth.

We used MPJ Express (version 0.23) with Sun JDK 1.5 (Update 6) to run the Java version of Gadget-2. The original C Gadget-2 code used MPICH (version 1.2.5.2) on Fast Ethernet and MPICH-MX (version 1.2.6.0.94) using Myrinet.

Figure 6 shows execution time of C and Java Gadget-2 on 1, 2, 4, and 8 dual CPU nodes using Ethernet. Note that we are running only one MPI or Java process on an SMP node. Here we observe that the Java code performs better than the C code. The results are somewhat surprising. Mainly the optimizations discussed in Section 3.2 helped achieve such good performance.

The results presented in [10] for the same simulation showed that the Java version was almost 30% slower than the C version. Since then we have fixed a bug in the Java code that meant that the Java code exchanged twice as much data per particle during domain decomposition and tree walk. In addition, we have improved the algorithms for Allgather and Allreduce in MPJ Express. One of the optimizations discussed under ‘Avoiding Expensive Array Operations’ in Section 3.2.3 was also applied since taking the results published in [10].

![Figure 6. Execution time comparison for the colliding galaxies simulation.](image-url)
We now turn our attention to profiling information shown in Table I. The Gadget-2 code gathers profiling information at each time step. This information includes the total execution time, domain decomposition time, tree walk time, tree construction time, communication time, and work-load imbalance. Other profiled sections include time for calculating particles drifts, time steps, and peano keys.

As Table I shows that the most compute intensive section of the code is the tree walk stage, which calculates gravitational forces by traversing the BH tree. As the number of MPI processes increases, the percentage of tree walk decreases. The communication cost increases from 0 to 16.3% (for C code) and 25.8% (for Java code) for the 8 processes case. The C MPI library helps achieve better performance in this profiled section. Here, better latency and bandwidth of the C MPI library is one reason but another, and more dominating factor, is the packing and unpacking carried out before and after the communication stage, respectively. This happens because the Java version uses a `ByteBuffer` object for communication of exported particles as discussed in Section 3.2.1. The current domain decomposition strategy results in work-load imbalance—it is 22.6% for C and 25.07% for Java with 8 processes.

### Table I. Comparative profiling information for the colliding galaxies simulation.

<table>
<thead>
<tr>
<th></th>
<th>1 MPI Proc</th>
<th>2 MPI Proc</th>
<th>4 MPI Proc</th>
<th>8 MPI Proc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>Java</td>
<td>C</td>
<td>Java</td>
</tr>
<tr>
<td>Total time (s)</td>
<td>5151.16</td>
<td>4701</td>
<td>3657.14</td>
<td>3217</td>
</tr>
<tr>
<td>Domain decomposition (%)</td>
<td>7.2</td>
<td>6.9</td>
<td>7.39</td>
<td>4.45</td>
</tr>
<tr>
<td>Particles and tree drifts (%)</td>
<td>0.72</td>
<td>0.85</td>
<td>0.69</td>
<td>1.4</td>
</tr>
<tr>
<td>Kicks and time stepping (%)</td>
<td>0.95</td>
<td>0.66</td>
<td>0.68</td>
<td>0.55</td>
</tr>
<tr>
<td>Miscellaneous (IO, etc.) (%)</td>
<td>0.83</td>
<td>0.88</td>
<td>2.47</td>
<td>2.1</td>
</tr>
<tr>
<td>Tree walk (%)</td>
<td>83</td>
<td>85</td>
<td>63</td>
<td>64.2</td>
</tr>
<tr>
<td>Tree construction (%)</td>
<td>2.8</td>
<td>3.4</td>
<td>2.6</td>
<td>2.9</td>
</tr>
<tr>
<td>Communication sum (%)</td>
<td>0</td>
<td>0</td>
<td>6.16</td>
<td>9.8</td>
</tr>
<tr>
<td>Work-load imbalance (%)</td>
<td>0</td>
<td>0</td>
<td>13.6</td>
<td>14.2</td>
</tr>
<tr>
<td>Peano keys and ordering (%)</td>
<td>4.5</td>
<td>2.31</td>
<td>3.5</td>
<td>0.40</td>
</tr>
</tbody>
</table>

5.2. The cluster formation simulation

This subsection presents performance results of a cluster formation simulation with 2 million particles in the system.

We carried out the tests on a larger cluster called NW-GRID located at the Daresbury Laboratory, U.K. The system consists of 96 nodes divided into 3 racks of 32 nodes. These nodes are connected by a Gigabit Ethernet network. Each node contains 2 dual core 2.4 GHz AMD Opteron 64-bit processors, has a minimum of 8 GBytes of memory, and 3 Gbit Ethernet cards. The nodes are running SuSE GNU/Linux with kernel 2.6.11.4–21.11-smp. The C library used is part of SCore 5.8.4 from Streamline Computing. The C compiler used is GNU C Compiler (GCC) 3.3.5 with support for 64-bit processor and POSIX threads enabled. JDK 1.5 update 7 was used to compile and run the Java code. The JDK used was specialized for AMD Opteron 64-bit processors and the...
Figure 7. Execution time comparison for the cluster formation simulation on NW-GRID cluster with PBCs turned on.

Figure 8. Tree walk time comparison for the custom system on the NW-GRID cluster with PBCs turned on.

virtual machine used in Java HotSpot 64-Bit Server VM. The evaluation that is presented is based on the first hundred time steps of the simulation, which is 10% of the total simulation.

Figure 7 shows the total execution time of the C and Java Gadget-2 code. The execution time of the Java version is 1.38, 1.34, 1.40, and 1.39 times the C version on 4, 8, 16, and 32 processors, respectively.

Figure 8 plots tree walk times. Tree walk times for the Java version are 1.26, 1.29, 1.35, 1.16 times the C version on 4, 8, 16, and 32 processors.
The main reason for the slower tree walk performance includes inherent security features like array bounds checking. Another overhead that occurs at the start of the execution is the time taken to convert bytecode to native machine code—also referred to as JIT compilation. In addition, the Java code cannot optimally exploit the processor cache due to the additional JVM layer and garbage collection cycles.

Analysis of our results suggests that the tree walk is the most compute intensive part of the code. Tree walk in the Java code is 84, 83.8, 78.7, and 58.2% of the total execution time on 4, 8, 16, and 32 processors, respectively. Similarly, the tree walk in the C code is 91.7, 86.9, 81.8, and 69.7% of the total execution time on 4, 8, 16, and 32 processors. The second most time-consuming section of the code is the work-load imbalance. Overall the tree walk time and work-load imbalance accounts for more than 95% of the total execution time on all processor counts.

6. PERFORMANCE EVALUATION OF THE JAVA FDTD CODE

We begin by describing our test environment, which consisted of a 32 processing-core Linux cluster at the National University of Sciences and Technology, Pakistan. The cluster consists of eight compute nodes. Each node contains a quad-core Intel Xeon processor. The nodes are connected via Myrinet and Fast Ethernet and we configured the Open MPI runtime to use Myrinet with MX driver for communication. The compute nodes run the SuSE Linux Enterprise Server 10 Operating System and GCC version 4.1.0. Each compute node has 2 GBytes of main memory. We used Open MPI version 1.2.4 as the C MPI library. For the parallel Java version, we used the latest development version of MPJ Express with the Sun JDK 1.6 Update 4.

Figure 9 presents the execution time graphs for the Java and C versions of the FDTD algorithm. Both versions scale in a similar manner although the Java code is marginally faster than the C code. The C code was executed with the GCC compiler version 4.1.0 and optimization switch -O3 was specified.

Figure 9. Java and C FDTD codes comparison.

Figures 10 and 11 plot the profiling information for Java and C versions of the FDTD code. This helps in understanding the comparative performance. The profiling information is gathered by calculating the time spent by each process in various stages, which include $H_x$ computation, $H_y$ computation, $E_z$ computation, $H_y$ communication, and $E_z$ communication. These stages are identified in Figure 5.

The profiling information shows that the Java code performs better in computation stages. Modern JVMs are equipped with JIT compilers, which first convert the Java bytecode to native machine code. In addition, the processor architecture is Intel Xeon and Java codes run efficiently on these architectures. Moreover, we observe that with 32 MPI processes, the communication cost for exchanging edge columns for $H_y$ and $E_z$ computation is higher for the Java code. This is understandable because of higher latency of the MPJ Express Myrinet device than its Open MPI counterpart. A more detailed performance analysis of MPJ Express is presented in Section 2.6.

Figure 10 plots the profiling information for the C FDTD code; here Figure 10(a)–(d) corresponds to execution results with 4, 8, 16, and 32 MPI processes. Similarly, Figure 11 plots profiling...
Figure 11. Profiling information for the Java FDTD code: (a) 4 Proc; (b) 8 Proc; (c) 16 Proc; and (d) 32 Proc.

information for the parallel Java FDTD code; here Figure 11(a)–(d) corresponds to execution results with 4, 8, 16, and 32 MPJ Express processes.

7. RELATED WORK

This section reviews the existing literature to evaluate Java’s potential for HPC. We cover efforts to improve serial performance of Java. In addition, we review existing messaging libraries that support execution of parallel Java applications on modern distributed memory clusters.

Fox et al. [3,4], in early works about Java for HPC, identified its potential for scientific computing. The NINJA project [34] did some pioneering work in improving the serial performance of the Java programming language. This project identified the following three characteristics as major hurdles in writing high performance scientific code: performance overhead of exception checks for NULL pointer and out-of-bounds array accesses, lack of true multidimensional arrays, and inefficient complex number arithmetic. Moreira et al. [32,35] address some of these shortcomings by
introducing Java packages, which specifically support regular-shaped arrays and efficient complex numbers arithmetic. In addition, certain compiler optimization techniques were implemented for the IBM High Performance Compiler to bring Java’s performance at par with Fortran. The project was very successful in achieving Fortran-like performance for the Java code.

Vivanco and Pizzi [7] conducted a case study, which compared the performance of Java and C versions of a functional magnetic resonance neuroimages application. The authors note that the C++ version outperformed the Java version. On the other hand, the Java programming language being easier to use leads to more robust code with shorter development times.

Bull et al. [2] produced C and Fortran versions of the Java Grande Benchmarks Suite and found that the performance gaps between these languages are decreasing, especially on Intel Pentium processors.

Nikishkov et al. [5] evaluated and compared Java and C versions of two solution algorithms for finite element equations. The authors demonstrate that the Java version can achieve up to 90% of the C version performance by simple optimization.

In a recent report by Amedro et al. [1], the authors perform an evaluation of Java against C and Fortran, using the Java Grande Benchmarks suite to find out the current state of HPC. The authors found out that similar computational performance was achieved although there were scalability issues for the Java codes on large clusters.

The literature survey shows that the performance of Java is catching up with C language on Intel processors especially.

In the context of Java messaging libraries there have been various efforts in the last decade to develop Java messaging systems. These typically follow one of three approaches: use a JNI interface to a native MPI implementation; implement Java messaging from scratch on top of Java RMI; or implement the system on top of lower-level Java sockets. We discuss each approach in turn, and give examples of corresponding messaging systems.

mpiJava [18] is a Java messaging system that uses JNI to interact with the underlying native MPI library. One of the unique features of the mpiJava library is that it is possible to run it on most high-speed interconnects by interfacing it with a native MPI library that supports the particular interconnect. For example, mpiJava has been used on top of MPICH-GM and MPICH-MX using the Myrinet hardware. mpiJava has been perhaps the most successful Java-based HPC messaging system, in terms of uptake by the community. It has been used as a teaching tool and for the development of performance measurement and analysis tool as well as for parallel simulations.

MPJ/Ibis [19] is an implementation of the MPJ API specification on top of Ibis [36]. Ibis claims to be a flexible and efficient Grid programming environment. The design of Ibis contains a messaging layer called Ibis Portability Layer (IPL). This layer is essentially a set of Java interfaces. The IPL provides communication functionality using abstractions like send and receive ports. IPL makes it possible to implement pure Java devices or use special HPC hardware such as Myrinet. In addition, MPJ/Ibis does not fully implement the higher-level features of MPI like derived datatypes (apart from contiguous datatype), virtual topologies, and inter-communicators. This library relies on scripts that use SSH for bootstrapping.

This paper does not claim to improve serial performance of the Java language. In addition, one of our design goals was to rely on standard Java compilers—the most popular is the Sun compiler. Our main aim is to evaluate the overall performance of Java and C versions of real-world parallel scientific applications.
8. CONCLUSIONS AND FUTURE WORK

This paper has discussed the suitability of the Java programming language for writing high-performance parallel scientific applications. Java is considered as a modern language with good support for advanced features like portability, modularity, garbage collection, and multi-threading. Multi-threading is especially important in the context of emerging ‘many-core’ processors. To enable Java for HPC, we have developed MPJ Express, which is an MPI-like library for writing high-performance parallel Java applications. MPJ Express currently follows the mpiJava 1.2 API and we plan to add support for the MPJ API in the near future. Its layered design makes it possible to quickly implement new communication devices for other protocols.

To help establish the practicality of real scientific computing, we ported two applications—Gadget-2 and FDTD method for computational electromagnetics—to Java using MPJ Express. The first application Gadget-2 is a massively parallel structure formation code. Versions of the original C code have been used in the so-called ‘Millennium Simulation’ that evolves ten billion dark matter particles from the origin of the universe to the current day. The second application is a parallel version of the FDTD algorithm for computational electrodynamics.

The performance evaluation of Java versions of two applications revealed that it could achieve comparable performance to the C code. In addition, the Java and C versions scale in a similar fashion. The comparative performance of these two applications reinforces our belief that Java is a viable option for HPC. With careful programming, it is possible to achieve performance in the same general ballpark as the C code.

In general, Java encourages better software engineering by being an object-oriented language that is more portable than its precursors. In addition, Java has many extra safety features including array bounds checking that could help identify potential bugs in the code. For example, we discovered a scenario in the original C Gadget-2 where the seventh element of a six elements array was accessed. The Java Gadget-2 helped identify this scenario by throwing an ArrayOutOfBound exception. We have informed the developer of C Gadget-2, who has fixed this problem in the distribution.

Currently, MPJ Express, our implementation of Java bindings, provides communication devices using Java NIO and Myrinet. We have plans to develop drivers for Infiniband, Quadrics, and shared memory platforms. In addition, keeping in view the popularity of the mpiJava library, we are currently implementing a communication device on this software. MPJ Express can be downloaded from http://mpj-express.org.

REFERENCES


17. The mpi4j2 project. Available at: http://www.java-mpi.org [29 January 2009].


33. Schneider webpage. Available at: http://www.eecs.wsu.edu/~schneidj/ee535 [29 January 2009].

