High-Performance File I/O in Java: Existing Approaches and Bulk I/O Extensions

Dan Bonachea* Phillip Dickens† Rajeev Thakur‡

Abstract

There is a growing interest in using Java as the language for developing high-performance computing applications. To be successful in the high-performance computing domain, however, Java must not only be able to provide high computational performance, but also high-performance I/O. In this paper, we first examine several approaches that attempt to provide high-performance I/O in Java—many of which are not obvious at first glance—and evaluate their performance on two parallel machines, the IBM SP and the SGI Origin2000. We then propose extensions to the Java I/O library that address the deficiencies in the Java I/O API and improve performance dramatically. The extensions add bulk (array) I/O operations to Java, thereby removing much of the overhead currently associated with array I/O in Java. We have implemented the extensions in two ways: in a standard JVM using the Java Native Interface (JNI) and in a high-performance parallel dialect of Java called Titanium. We describe the two implementations and present performance results that demonstrate the benefits of the proposed extensions.

1 Introduction

There is a growing interest in using Java for high-performance computing because of the many advantages that Java offers as a programming language. To be useful as a language for high-performance computing, however, Java must not only have good support for computation but must also be able to provide high-performance file I/O, as many scientific applications have significant I/O requirements [6, 22, 34]. In this paper, we investigate the I/O capabilities of Java for high-performance computing and provide suggestions for relatively simple changes to the Java I/O model that can improve performance significantly.

We first examine several approaches that attempt to provide high-performance I/O in Java—many of which are not obvious at first glance—and evaluate their performance. We perform experiments on two different parallel machines, a distributed-memory system (IBM SP) and a shared-memory system (SGI Origin2000), both of which employ modern parallel/high-performance file systems. We then propose extensions to the Java I/O library that address the deficiencies in the Java I/O API and improve performance dramatically. The extensions add bulk (array) I/O operations to Java, thereby removing much of the overhead currently associated with array I/O in Java. We have implemented the extensions in two ways: in a standard JVM using the Java Native Interface (JNI) [24] and in a high-performance parallel dialect of Java developed at U.C. Berkeley called Titanium [35, 38]. We describe the two implementations and present performance results that demonstrate the benefits of these extensions.

*EECS Department, University of California at Berkeley, Berkeley, CA 94720. bonachea@cs.berkeley.edu

†Dept. of Computer Science, Illinois Institute of Technology, 10 West 31st Street, Chicago, IL 60616. pmd@work.csam.iit.edu

‡Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. thakur@mcs.anl.gov

1.1 I/O in High-Performance Computing

Many computationally intensive scientific applications also need to access large amounts of data, and I/O is often the bottleneck in such applications [6, 22, 34]. A common I/O requirement is as follows. The application has some large data structures, say multidimensional arrays, distributed among processes in some fashion. The arrays must be read from or written to a file containing the global array. The program may begin by reading in an input array and may then write arrays to files several times during the course of the computation. The arrays in these applications are not just byte arrays, but rather consist of integers, or floating-point numbers, or some other data type. As we shall see in this paper, the fact that they are not just byte arrays is important in the context of using Java for I/O in such applications. In addition, the files are usually random-access files, and processes seek to different locations in the files to read/write data.

In this paper, we focus on the problem of concurrent reading or writing of data from multiple processes/threads to a common file in Java. We assume that a large one-dimensional array of integers is block-distributed among processes and must be read from or written to a common file containing the global array. While simple, this example is sufficient to demonstrate the strengths and weaknesses of the Java I/O model as applicable to the basic needs of high-performance computing applications. Our experiments assume (and employ) a random-access file that is striped across the disks of a parallel file system.

Much of the research related to parallel I/O has been performed in the context of C, and C provides excellent support for such operations. In particular, C allows the casting of an array of *any* type into an array of bytes, and multidimensional arrays can be treated as one-dimensional arrays of the same size. The Unix I/O functions simply take a pointer to a one-dimensional array, the number of bytes to be read or written, and the offset into the file, and they carry out the request as a single I/O operation. It is also quite simple to perform parallel reads and writes in C without the need for synchronization (on file systems that support such access). In particular, each process can seek to an independent (non-overlapping) region of a shared random-access file and then perform its reads or writes to disjoint regions of the file in parallel.

There are other advantages of C/Unix based I/O as well. One advantage is that local (nonportable) hooks to a parallel file system can provide excellent performance enhancements on some machines. For example, the `O_DIRECT` option available on the XFS file system on the SGI Origin2000 allows the application to bypass the system file cache and write directly to disk. On systems with high disk bandwidth, this option can improve performance significantly [12]. The disadvantage of this approach, of course, is that it is not portable. Another advantage of C-based I/O is that there are portable APIs, such as MPI-IO [17], that are implemented in an optimized fashion for different machines and file systems.

The situation in Java, however, is quite different. Achieving high-performance parallel file I/O in Java is currently a very difficult issue, primarily because of the constraints imposed by the interface design of the Java I/O library. However, the widespread standardization and platform independence of Java provide an ideal vehicle for deploying a high-performance I/O library interface whose implementation can be individually tuned to fully utilize the capabilities of each underlying architecture.

1.2 Contributions of this Paper

The contributions of this paper are mainly twofold. First, we provide a detailed discussion and performance analysis of several approaches to parallel file I/O available in Java and do so across two different parallel architectures and file systems. To date, there has been relatively little research focusing on the I/O capabilities of Java in general, and on its capabilities to perform parallel file I/O in particular. Second, we propose extensions to the Java I/O API that can improve performance significantly. These extensions allow users to perform bulk (array) I/O operations with a single method call. We have implemented these extensions and validated their performance benefits.

1.3 Related Work

Other than the large body of work related to parallel I/O [4, 8, 9, 13, 23, 27, 28, 32, 33], the work most closely related to ours is the Jaguar project [36, 37], which aims to improve Java I/O performance as one of its goals. Jaguar allows the Java runtime system to be extended with new primitive operations that enable efficient access to hardware resources. These primitives are specified as short machine code segments that are directly inlined into the Java bytecode as it is compiled. The Jaguar project is, in fact, complementary to the work discussed in this paper, the difference being the level at which performance improvement is targeted. This paper deals with the Java I/O facilities available to the user at the application level. The Jaguar project provides performance enhancements at a lower system level. Another interesting aspect of the Jaguar project is the idea of pre-serialized objects, where objects are stored in a pre-serialized format ready for communication or I/O. A similar idea could be applied to arrays of Java primitive data types, with the required encoding/decoding being performed by threads executing in the background while the main thread engages in other computation/communication.

A preliminary version of our work was presented in [2, 11].

1.4 Organization

The rest of this paper is organized as follows. In Section 2 we describe the basic I/O mechanisms defined in Java. In Section 3 we discuss several approaches for performing parallel file I/O in Java. We study the performance of these approaches in Section 4. Suggestions for improving the Java I/O model are presented in Section 5. The implementation of these extensions is discussed in Section 6. Performance results with the extensions are presented in Section 7. Conclusions and and ideas for future work are presented in Section 8.

2 I/O in Java

To understand the issues associated with performing parallel I/O in Java, it is necessary to briefly review the Java I/O model [18].

Generally, I/O in Java is divided into two parts: *byte-oriented* I/O, which includes bytes, integers, floats, doubles and so forth, and *text-oriented* I/O, which includes characters and text. In this paper, we are concerned only with byte-oriented (binary) file I/O. In Java, byte-oriented I/O is handled by input streams and output streams, where a stream is an ordered sequence of bytes of unknown length.

Java provides a rich set of classes and methods for operating on byte input and output streams. These classes are hierarchical, and at the base of this hierarchy are the abstract classes `InputStream` and `OutputStream`. It is useful to briefly discuss this class hierarchy in order to clarify the possible approaches to performing high-performance I/O in Java. To facilitate this discussion, Figure 1 provides a graphical representation of this I/O hierarchy. We note that we have not included every class that deals with byte-oriented I/O but have included only those classes that are pertinent to our discussion.

2.1 InputStream and OutputStream Classes

The abstract classes `InputStream` and `OutputStream` are the foundation for all input and output streams. They define methods for reading/writing raw byte input/output streams.

The `InputStream` class provides three methods for reading bytes from an input stream. One method reads a

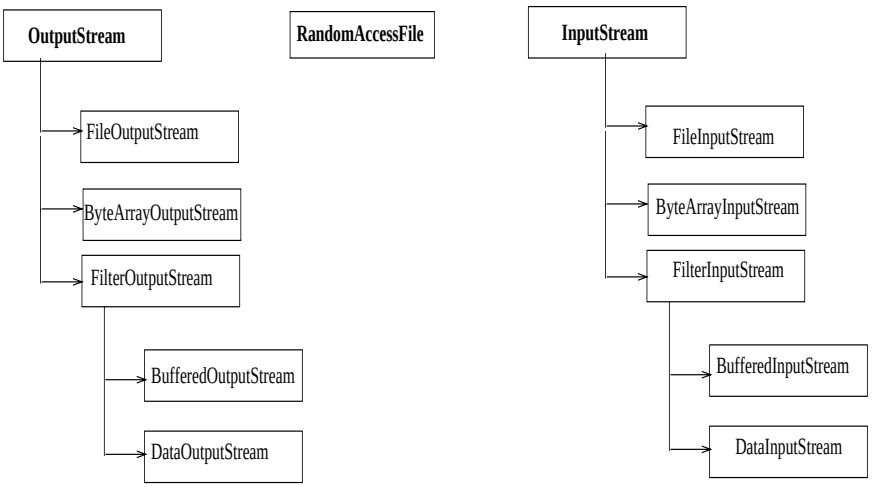

Figure 1: This figure shows the I/O class hierarchy pertinent to this investigation. Note that the `RandomAccessFile` class is completely outside of the `InputStream` and `OutputStream` hierarchy. As discussed in Section 2.5, however, a connection can be made between a `RandomAccessFile` and a `FileInputStream` or `FileOutputStream`.

single byte, another method reads available data into a byte array, and the third method reads the available data into a particular region of a byte array. We are interested in the third method since it allows distinct threads to read into distinct regions of the same byte array in parallel. The signature for this method is:

```
public int read(byte[] buf, int offset, int length) throws IOException
```

In addition to the three read methods, the `InputStream` class defines methods to skip over bytes in the input stream, to determine the number of bytes available in an input stream, and to close an input stream.

The `OutputStream` class provides methods for writing that are analogous to those of `InputStream`. In particular, it provides three write methods: one to write a single byte to an output stream, one to write an array of bytes to an output stream, and one to write a subarray of bytes to an output stream. We are interested primarily in the third method, which can be used as the basis for performing parallel writes (when used in the context of random-access files, as discussed below). The signature for this method is:

```
public void write(byte[] buf, int offset, int length) throws IOException
```

In addition to the three write methods, this class also supports methods to flush and close output streams. A very significant feature of the `OutputStream` class is that, unlike the `InputStream` class, it does *not* support skipping (or seeking) over bytes in the output stream. This precludes multiple threads from writing to distinct regions of the output stream, which basically precludes performing parallel writes. The solution to this problem is discussed in Section 3.

2.2 File Input and Output Streams

The `FileInputStream` and `FileOutputStream` classes are concrete subclasses of `InputStream` and `OutputStream`, respectively, and provide a mechanism to read from and write to files. `FileInputStream` provides all the methods of the `InputStream` class and defines only one new method, which can be used to obtain an opaque *file descriptor* object. The signature for this method is:

```
public final FileDescriptor getFD() throws IOException
```

Note that the ability to skip over bytes in a file input stream means that multiple threads can seek to disjoint regions in an input file. This feature, in addition to the fact that multiple threads can read into disjoint sections of a byte array in parallel, provides the basis for parallel reads into a common array.

There are three constructors for file input streams. One constructor takes as a parameter a string representing the file name. Another constructor takes as a parameter a `Java.io.File` object. The third constructor requires a `FileDescriptor` object. For reasons discussed below, the third constructor is most pertinent to this discussion and has the following signature:

```
public FileInputStream(FileDescriptor fd)
```

Similar to the `FileInputStream` class, the `FileOutputStream` class also provides the three write methods available in its superclass and defines only one new method for obtaining a `FileDescriptor` object. The constructor for this class most pertinent to our discussion takes as a parameter a `FileDescriptor` and has the following signature:

```
public FileOutputStream(FileDescriptor fd)
```

We note that it is not possible for multiple threads to seek to different locations in a file output stream since the class provides no method to do so.

2.3 Byte Array Streams

The `ByteArrayInputStream` class reads data from a byte array using the methods of the superclass. It provides two constructors: one that takes a byte array as its parameter (and uses this byte array as the input source), and one that takes a byte array plus an offset and a length, and uses this subarray as the input source. Otherwise, it defines no new methods.

The `ByteArrayOutputStream` class writes bytes into successive components of an internal byte array. The size of this internal byte array is determined by the class constructors. One constructor takes no arguments and employs a default buffer size of 32 bytes. The second constructor takes as an argument the initial size of the buffer. In either case, the size of the byte array grows to accommodate additional data. A copy of the internal byte array can be obtained through the `toByteArray` method. The signature for this method is:

```
public synchronized byte[] toByteArray()
```

2.4 Filter Streams

Filter streams provide methods to chain streams together to build composite streams. For example, a `BufferedOutputStream` can be chained to a `FileOutputStream` to reduce the number of calls to the file system.

The `FilterInputStream` and `FilterOutputStream` classes define a number of subclasses that manipulate the data of an underlying stream. The constructor for a `FilterInputStream` object takes as a parameter an `InputStream` object, and the constructor for a `FilterOutputStream` object takes as a parameter an

`OutputStream` object. Otherwise, these classes provide the same methods defined by the `InputStream` and `OutputStream` classes.

Two subclasses of filter streams are pertinent to this investigation. One subclass is `DataInputStream`, which allows raw byte input to be treated at the level of Java primitive types. The other subclass, `BufferedInputStream`, provides buffering for an underlying stream. Similar subclasses are defined by `FilterOutputStream`. It is worthwhile to briefly discuss these two subclasses.

2.4.1 Buffered Streams

The `BufferedInputStream` and `BufferedOutputStream` classes provide buffering for an underlying stream, where the stream to be buffered is passed as an argument to the constructor. The buffering is provided by an internal system buffer whose size can (optionally) be specified by the user.

2.4.2 Data Streams

All the classes discussed thus far manipulate raw byte data only. Applications, however, deal with higher-level data types, such as integers, floats, doubles, and so forth. Java defines two interfaces, `DataInput` and `DataOutput`, that define methods to treat raw byte streams as these higher-level Java data types. Together, these interfaces define methods for reading and writing all Java data types. The `DataInputStream` and `DataOutputStream` classes provide default implementations for these interfaces. For example, the two methods that read and write integers are the following:

```
public final int readInt() throws IOException
public final void writeInt(int i) throws IOException
```

It is important to note that these methods read or write a *single* integer at a time. No method exists in Java for reading or writing an array of integers (or an array of any data type other than bytes).

2.5 Random-Access Files

As mentioned above, it is not possible to seek to some location in the file when writing with the `FileOutputStream` class because, unlike `FileInputStream`, `FileOutputStream` provides no methods for seeking. To overcome this problem, we use the `RandomAccessFile` class that provides more sophisticated file I/O. In particular, it provides the seek method that we require.

```
public void seek(long position) throws IOException
```

It is interesting to note that the `RandomAccessFile` class sits alone in the I/O hierarchy and duplicates, rather than inherits, methods from the stream I/O hierarchy. In particular, `RandomAccessFile` duplicates the read and write methods defined by the `InputStream` and `OutputStream` classes and implements the `DataInput` and `DataOutput` interfaces that are implemented by the data stream classes. However, since `RandomAccessFile` is not in the stream hierarchy, it cannot be directly used where input or output streams are required.

There is, however, a (not entirely obvious) way to form a connection between the `RandomAccessFile` class and the rest of the stream hierarchy. This can be done by getting the file descriptor of a random-access file with `getFD()` and using the file descriptor as a parameter to the constructor for a `FileInputStream` or

`FileOutputStream` object. Once this connection is made, a random-access file can be chained to filter streams and byte-array streams.

3 Approaches to Parallel File I/O in Java

In this section we describe six different approaches for performing parallel file I/O in Java. Most of these approaches are different ways of working around the problem that Java does not directly support the reading or writing of arrays of any data type other than bytes.

3.1 Using Raw Byte Arrays

If the data to be read or written is already in the form of a byte array, it is trivial to read or write the data using the Java methods for reading/writing byte arrays. As noted above, however, byte is the only data type for which such array operations are defined.

Let us assume that multiple threads of a parallel program need to write different parts of a byte array to a common file. Assume further that the file system permits concurrent writes to disjoint locations in a file. We can perform the I/O as follows. Each thread in the parallel program creates a `RandomAccessFile` object, calculates its offset in the shared file, and seeks to that position. It then uses the `write` method defined by the `RandomAccessFile` to write its portion of the byte array in a single operation, as shown below.

```
// this is executed by the main thread

  byte buf[] = new byte[buf_size];

// this code is executed by all of the threads.
// First create a RandomAccessFile object, then
// calculate offset in file

  RandomAccessFile raf =  new RandomAccessFile (filename,access);
  raf.seek(position);

// calculate offset within byte array and number
// of bytes to write, then perform write

  raf.write(buf,my_start_buf,num_bytes);
```

It is important to note that this approach works correctly both when an existing file is overwritten and when a new file is created, because of the semantics of the `seek` method. In particular, a `seek` to a location past the end of the file, followed by a write, extends the length of the file [30].[1]

3.2 Converting to/from an Array of Bytes

As we shall see in Section 4, I/O involving byte arrays is simple and also performs well. The problem, however, is that real applications do not operate on arrays of bytes. Rather, they deal with arrays of other data types,

[1] These semantics were introduced in the Java 1.1 language specification.

such as integers, floats, and doubles. Java, unfortunately, provides no methods for performing I/O operations on such arrays. Furthermore, unlike C, Java does not allow users to simply cast an array of some other type into an array of bytes. Nonetheless, we can still use the byte-array methods by explicitly converting an array of some other data type into an array of bytes, and vice versa.

For example, we can write an array of integers by first right-shifting one byte at a time into a byte array and then writing the byte array. Similarly, we can read an array of integers by first reading into a byte array and then converting the bytes into integers. The only issue encountered in the conversion from bytes to integers stems from the fact that Java does not have unsigned data types. Thus, if the high bit of a given byte is set, it is interpreted as a negative number when converted to an integer. More precisely, the lower eight bits of the integer are copied from the eight bits of the byte, and the upper 24 bits are set to 1 (sign extension). We must, therefore, take care of the sign bit when converting bytes to integers. The conversion can be done as follows without explicitly checking the sign bit (that is, without a branch):

```
// Assume we are converting the byte array, buf, into integers in
// an integer array, int_array.

   for (int i=0; i < int_array.length; i++) {
      int_array[i] =     (((int)buf[4*i+3]) & 255)
                       | ( (((int)buf[4*i+2]) & 255) << 8 )
                       | ( (((int)buf[4*i+1]) & 255) << 16 )
                       | ( (((int)buf[4*i+0]) & 255) << 24 );
   }
```

3.3 Using Data Streams

It is possible to read/write a *single* integer at a time by using the methods defined in the `DataInput` and `DataOutput` interfaces. As noted above, the `RandomAccessFile` class implements these interfaces, making it relatively easy to perform parallel I/O operations using data streams. The pseudo-code for this approach is shown below. Note that the `writeInt` method is called several times in a loop, writing one integer at a time, which is very expensive. (It induces a method-call overhead linear in the number of primitive data values to be written.)

```
// main program

  int[] int_array = new int[num_ints];

// each thread calculates its position in the
// file and the array, and calculates the number of
// integers it needs to read or write.

  RandomAccessFile raf = new RandomAccessFile(filename,access);
  raf.seek(position);
  for (int i = start_buf; i < (start_buf+num_ints_to_write); i++)
      raf.writeInt(int_array[i]);
```

3.4 Using Buffered Data Streams

As we shall see in Section 4, using regular (unbuffered) data streams results in the poorest performance across all approaches studied, because a call to the I/O subsystem is made for *every* integer read or written. It is thus

desirable to seek approaches that internally buffer data before reading/writing. The problem, however, is that the `RandomAccessFile` class does not implement buffering, and the `FilterInput` and `FilterOutput` streams (of which buffered streams are a subclass) only work with objects of type `InputStream` and `OutputStream`.

There is a way to use system buffering for a `RandomAccessFile` object as follows. A `RandomAccessFile` can be chained to a `FileInputStream` or `FileOutputStream` object through its file descriptor. The `FileInputStream` or `FileOutputStream` object can be chained to a `BufferedInputStream` or `BufferedOutputStream` object, which can then be chained to a `DataInputStream` or `DataOutputStream` object.[2]

We note, however, that it is *not* safe to use buffered data streams for *writing* concurrently from multiple processes or threads to overlapping regions of a common random-access file. This is because each thread or process maintains its own local buffer, and the buffers of different processes may not be coherent. This problem does not exist in the case of concurrent *reads*, of course.

The pseudo code for using buffered streams is shown below, with the caveat that, depending on the implementation, there is potential for erroneous results. Specifically, it is probably a bad idea to call `seek()` on the `RandomAccessFile` object after any reads or writes take place on the associated stream objects, as this will most likely result in buffer-consistency errors, leading to data corruption. (One can avoid this particular difficulty by closing and reopening all the file objects when a seek is necessary.) It is also advisable to explicitly `flush()` the `DataOutputStream` object before closing any file objects to prevent the possibility of losing the final buffer of data written.

```
  RandomAccessFile raf    = new RandomAccessFile(filename,access);
  FileDescriptor fd       = raf.getFD();
  FileOutputStream fos    = new FileOutputStream(fd);
  BufferedOutputStream bos= new BufferedOutputStream(fos);
  DataOutputStream dos    = new DataOutputStream(bos);

// each thread calculates its offset within the array,
// its offset in the file, and the number of
// elements to write to disk.

  raf.seek(position);
  for (int i = start_buf; i < (start_buf + num_ints_to_write; i++)
      dos.writeInt(int_array[i]);
```

Although this approach has introduced buffering, the Java method-call overhead is still linear in the number of primitive data values being read or written, which we shall see is a performance problem.

3.5 Using Buffering with Byte Array Streams

Another approach to buffering a data input or output stream is to chain it to an underlying byte array stream. Then the read and write methods invoked on the data stream will be directed to the underlying byte array stream rather than directly to disk. This composite stream is defined as follows:

```
RandomAccessFile raf = new RandomAccessFile(filename,access);
ByteArrayOutputStream bos = new ByteArrayOutputStream(size);
DataOutputStream dos = new DataOutputStream(bos);
```

[2] This trick relies on some under-specified aspects of the Java I/O subsystem. Specifically, it assumes an implementation where the seek pointer state is associated with the opaque FileDescriptor object and not the enclosing RandomAccessFile object.

Note that it is advantageous to specify the correct buffer size to the `ByteArrayOutputStream` constructor, instead of just using the default buffer size of 32 bytes, in order to avoid the cost of having the implementation grow (reallocate) the buffer as needed.

As in the previous cases, the individual threads seek to their correct position in the integer array and the shared file. In the case of a write, the thread simply writes all its data to the output data stream, which in turn writes it to the underlying byte array stream. Once the write is complete, the thread uses the `toByteArray` method to write the data from the byte array to the shared file. This is shown below.

```
for(int i = start_buf; i < (start_buf + num_ints_to_write; i++)
    dos.writeInt(int_array[i]);
raf.seek(position);
raf.write(bos.toByteArray());
```

Note that the `toByteArray()` method returns a newly allocated copy of the internal byte array maintained by the `ByteArrayOutputStream`; therefore, this approach imposes the CPU and memory-footprint overheads of an additional data copy.

It is slightly more complicated to use byte array streams for read operations. First, each thread declares its own byte array, creates the `ByteArrayInputStream` and `DataInputStream` objects, and seeks to the appropriate location in the file. Next, each thread reads from the file into its byte array using the low-level read method. Finally, the data is transfered from the byte array into the integer array using the read method of the data input stream class. The pseudo-code for this operation is given below.

```
// each thread allocates its own buffer
  byte[] buf = new byte[num_bytes_to_read];
  ByteArrayInputStream bis = new ByteArrayInputStream(buf);
  DataInputStream dis = new DataInputStream(bis);
  raf.seek(position);
  raf.readFully(buf, 0, num_bytes_to_read);
  for (int i = start_buf; i < (start_buf + num_ints_to_read); i++)
      int_array[i] = dis.readInt();
```

Note that, in both cases, the method-call overhead is still linear in the number of primitive data values being read or written.

3.6 Other Approaches

There are at least two other ways of performing I/O in Java. One way is to use object serialization [18]. We explored this approach initially, but found that Java adds some additional bytes to the file in order to store object-related information. This makes it difficult to perform parallel reads or writes because the threads would not know where to seek in the file. Object serialization in Java is also known to be very slow [5].

Another way is to not use the I/O methods defined in Java, but rather to use the Java Native Interface (JNI) [24] to extend the existing libraries with new methods specialized for handling array-based I/O. We used this method to implement the bulk I/O extensions proposed in this paper. See Section 6 for details.

4 Performance Results for Existing Java I/O Methods

In this section we present the results of our experiments with the various approaches described above. We first describe the two machines used for our experiments.

4.1 Computational Platforms and Experimental Setup

We conducted experiments on two parallel machines located at Argonne National Laboratory, an IBM SP and an SGI Origin2000. At the time we performed our experiments, the SP was configured with 80 compute nodes and 4 I/O processors. Each I/O processor controlled four SSA disks, each of 9 Gbyte capacity. The Origin was configured with 128 compute processors and ten Fibre Channel controllers connected to a total of 110 disks of 9 Gbyte capacity each. On both machines, we used the native parallel/high-performance file systems, namely, PIOFS on the SP and XFS on the Origin2000.

The programs we ran on both machines were parallel multiprocess Java programs. Each process ran on a different Java Virtual Machine. We could have simply spawned Java processes, but our parallel program also needed some additional information that MPI [16] typically provides, such as the total number of processes in the computation and the rank of a process in the process group (in order to determine its position in the shared file). One way to get around this problem is to use one of the several research projects in this area, such as JavaNOW [31] or an MPI wrapper for Java [25]. We used a simpler approach, however, in which we invoked the Java program from within a simple MPI program written in C. The MPI program used MPI functions to determine the rank of the process and the number of processes, and then invoked the Java program using the `system()` call in C, passing the rank and number of processes as command-line arguments. The timings were measured across the I/O calls in the Java program. Each Java process had its own private array, but all processes shared the global file. On the SP, we used a 4 Mbyte array per process, whereas on the Origin we used a 32 Mbyte array per process. These sizes were chosen based on some experiments to determine the right size for good I/O performance on these machines. Each process read (or wrote) multiple times; the total file size read (or written) was 1 Gbyte. On the SP, we used IBM's Java software, which was conformant with the behavior of Sun's JDK 1.1.2. On the Origin, we used version 3.1.1 of SGI's Java software, which was conformant with the behavior of Sun's JDK 1.1.6.

4.2 Results

The results of our experiments are shown in Figure 2. We note that our intention was not to compare performance between the two machines since they have very different I/O configurations. Rather, we wanted to compare the performance of the various approaches on a particular machine, for two different machines.

The experiments can basically be divided into two categories. The first category, which includes the first two approaches discussed in Section 3, uses the Java I/O methods for reading/writing arrays of bytes. In the first case of this category, we assume the data is already in byte form; in the second case (called encode/decode in Figure 2), we explicitly perform the conversion from integer arrays to byte arrays and vice versa. The second category, which includes all the other experiments, uses the data stream classes either alone or chained to some underlying stream that provides buffering.

The I/O performance is quite poor when using the data stream classes and methods, even when buffered. The poor performance of the data stream classes stems from three factors. First, when used without buffering, this approach requires a call to the I/O subsystem for *every* element of the array. This may be acceptable when I/O requirements are small, but is certainly not acceptable for large scientific applications. Secondly, even when buffering is provided by an underlying stream, this approach still requires invoking a method for

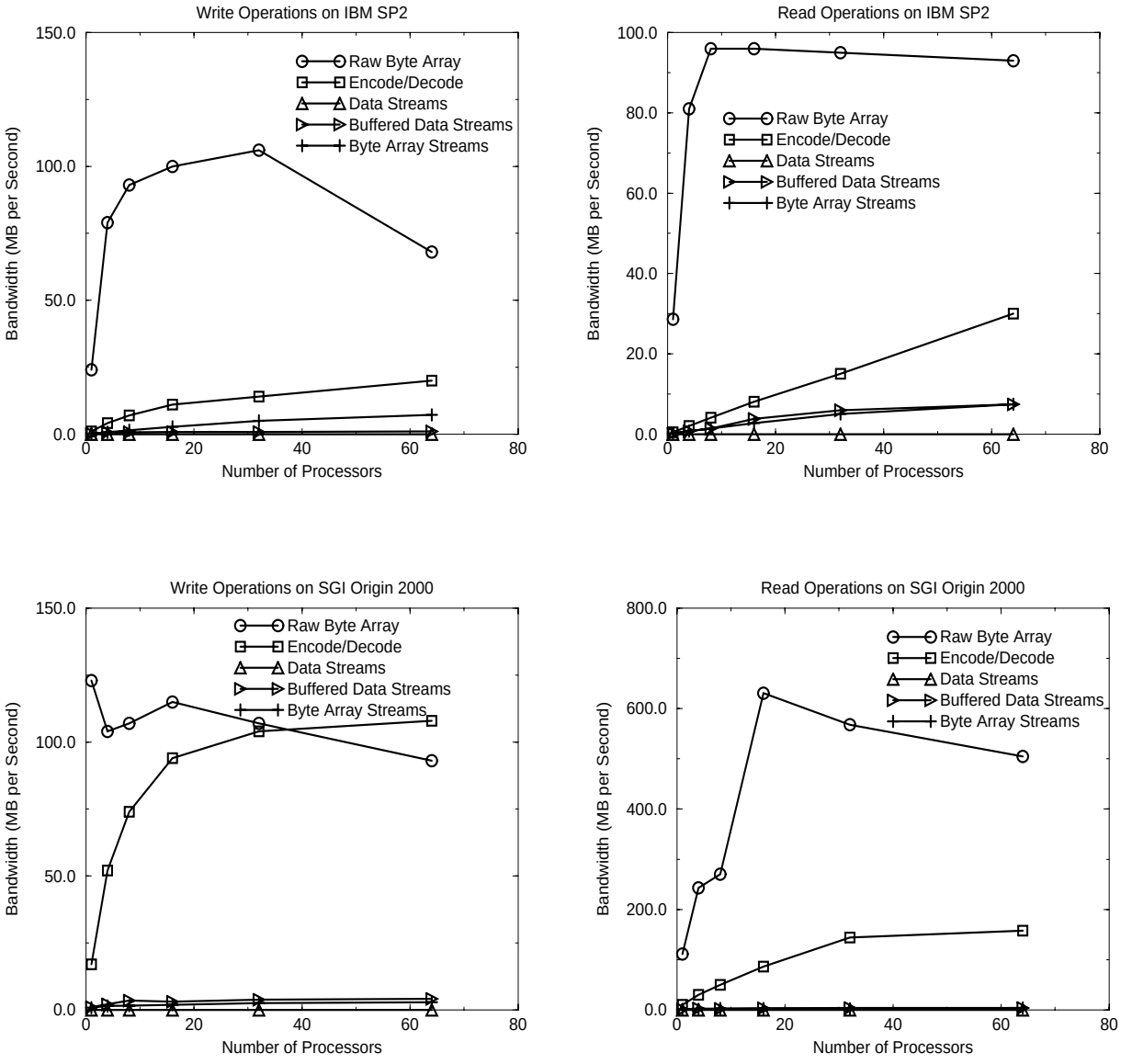

Figure 2: The performance of various approaches to high-performance file I/O in Java

every element of the array. With 64 processes and a 1 Gbyte array, each process must make over four million calls to the `readInt` or `writeInt` methods. With a single process, this number increases to over 268,000,000. Clearly this is a significant obstacle to achieving high-performance file I/O. The third problem is that many of the methods of the `DataOutputStream` class write to the underlying stream one byte at a time, and each such write requires a lock acquisition [19].

Although buffering improved the performance of data streams by orders of magnitude (for example, from 0.00074 Mbytes/sec to 0.19 Mbytes/sec), it could not match the performance of writing byte arrays directly, which was more than 100 Mbytes/sec. We also observed that the size of the buffer was quite important when using the buffered data streams. In particular, choosing the correct buffer size more than tripled the throughput. (We should also note that a nontrivial amount of experimentation was required to find the best buffer size.) Again, the difference in performance, however, was only in the range of 1 Mbyte/sec to 3 Mbytes/sec, for example.

As expected, the best performance was obtained when using the Java I/O facilities for directly reading and writing arrays of bytes. In fact, the first approach, which simply assumed the data was already in byte form, provided performance essentially identical to that obtained when using C. However, there was a significant drop in performance (for all but one experiment) when the application itself had to convert data from an array of integers to an array of bytes or vice versa (encode/decode). With this method, a more realistic scientific application that actually performs non-trivial computations may see an even larger performance degradation—in our test program, the CPU could be devoted more or less entirely to performing the encode/decode transformations with no degradation to the overall running time.

4.3 Results on the IBM SP

One striking result on the SP is the rather significant drop in performance observed when moving from 32 to 64 processors using raw byte arrays. The reason for this drop is the contention caused by the underconfigured I/O subsystem with only four I/O processors. This trend was not observed for any other approach because they were not operating at a bandwidth approaching the hardware limit. The best write performance was obtained using raw byte arrays with 32 processors (resulting in a bandwidth of 106 Mbytes/sec). The best result with encode/decode was 20 Mbytes/sec with 64 processors. The maximum throughput observed across all the other approaches was 7.5 Mbytes/sec, obtained with 64 processors and using byte array streams for buffering.

The best performance obtained for the read operations was 96 Mbytes/sec when using raw byte arrays with 16 processors. There was a small decrease in performance when the number of processors was increased to 32 and 64, this again due to the underconfigured I/O subsystem. The best performance obtained using encode/decode was 30 Mbytes/sec with 64 processors. The best performance for all the data stream methods was 7.5 Mbytes/sec, again obtained with 64 processors and using byte array streams for buffering.

4.4 Results on the SGI Origin2000

For writing on the Origin2000, encode/decode performed quite close to raw byte arrays. We believe this is because the bottleneck in the case of writing is the serialization that the XFS file system imposes on concurrent writes, rather than the extra computation and memory copy that encode/decode entails. With 64 processors, raw byte arrays achieved a throughput of 97 Mbytes/sec, while encode/decode resulted in a throughput of 89 Mbytes/sec. The best performance observed using data streams was 4.1 Mbytes/sec, obtained using buffered output streams with a 0.5 Mbyte buffer.

Raw byte arrays achieved excellent performance for reading. For example, a throughput of 631 Mbytes/sec

was observed when using 16 processors. We see a decrease in performance when the number of processors was increased to 64 because of increased contention for I/O resources. Encode/decode resulted in a maximum throughput of 158 Mbytes/sec with 64 processors. The maximum throughput obtained using the data stream methods was 4 Mbytes/sec, when either byte arrays or buffered streams were used to buffer the data streams.

5 Improving Java I/O Performance

The above results demonstrate that the I/O methods that directly read/write arrays of bytes are the only existing methods in Java that provide reasonable I/O performance. Real applications, however, do not operate on byte arrays; they need the ability to read or write arrays of other data types, such as integers and floats. The data stream methods that operate on such data types do not allow users to read or write *arrays* of data types. One can read or write only a *single* data item at a time, resulting in poor I/O performance.

We propose a straightforward extension to the Java I/O libraries that alleviates this problem. The extension adds bulk (array) I/O operations to the existing libraries, thereby removing most of the method-call overhead currently associated with array I/O.

5.1 Bulk I/O Extensions

We propose adding three new subclasses (`BulkDataInputStream`, `BulkDataOutputStream`, and `BulkRandomAccessFile`) to the `java.io` hierarchy as pictured in Figure 3. These new classes implement the methods from two new interfaces, `BulkDataInput` and `BulkDataOutput`, which are subinterfaces of the `DataInput` and `DataOutput` interfaces that currently provide single-primitive I/O. `BulkDataInput` and `BulkDataOutput` are both very simple. Each class adds two new methods for performing array-based I/O (with overloads to handle the different data types): one method for performing I/O on an entire array and a second for performing I/O on a contiguous subset of the elements in an array. The interfaces are shown below with the methods for `int[]` and `float[]` (the overloaded methods for `boolean[]`, `char[]`, `byte[]`, `short[]`, `long[]`, and `double[]` have been omitted for brevity).

```
public interface BulkDataInput extends DataInput {
  public void readArray(int[] array) throws IOException;
  public void readArray(int[] array, int arrayoffset, int count) throws IOException;

  public void readArray(float[] array) throws IOException;
  public void readArray(float[] array, int arrayoffset, int count) throws IOException;
  }

public interface BulkDataOutput extends DataOutput {
  public void writeArray(int[] array) throws IOException;
  public void writeArray(int[] array, int arrayoffset, int count) throws IOException;

  public void writeArray(float[] array) throws IOException;
  public void writeArray(float[] array, int arrayoffset, int count) throws IOException;
  }
```

The `BulkDataInputStream` class implements the methods from `BulkDataInput`; `BulkDataOutputStream` implements the methods from `BulkDataOutput`; and `BulkRandomAccessFile` implements both interfaces.

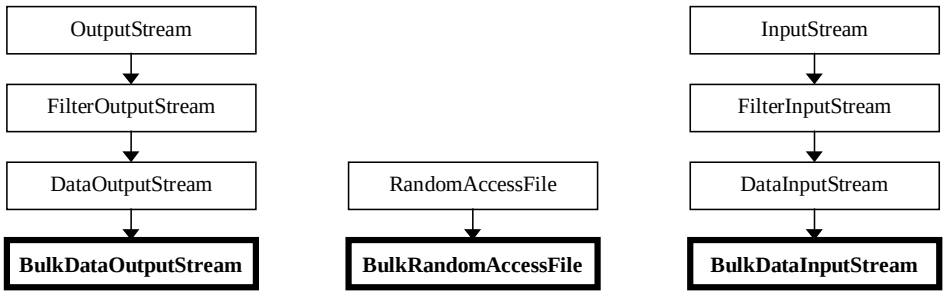

Figure 3: I/O class hierarchy with the bulk I/O extensions

5.2 Design Motivation

The extensions use subclassing to add support for bulk I/O to the `java.io.*` classes that implement the `DataInput/DataOutput` abstract interfaces. The new bulk classes add support for performing bulk I/O on a single-dimensional array of any primitive type, while inheriting all the traditional single-element I/O methods from their respective superclasses. To use the extensions, programmers merely change the declaration of their top-level I/O object to the new bulk equivalent and add calls to the `readArray` and `writeArray` methods, where appropriate, to perform bulk I/O on arrays of arbitrary length with a constant method-call overhead.

Below is a simple example using the `BulkRandomAccessFile` object that reads some header information (using the inherited single-value method `readInt()`) and then calls a bulk read into a array of doubles:

```
BulkRandomAccessFile braf = new BulkRandomAccessFile("myfile","r");
int numEntries = braf.readInt();
double[] myArray = new double[numEntries];
braf.readArray(myArray);
```

Note that the bulk extensions do not directly support arrays of multiple dimensions or whose elements are of reference type. However, multidimensional arrays can be accessed by calling the methods for one-dimensional arrays several times.

Finally, we note that the two new stream-based classes (`BulkDataInputStream` and `BulkDataOutputStream`) are not only useful for file I/O, but could also be used with network I/O streams; therefore, these extensions could also benefit high-bandwidth networking applications.

6 Implementation of the Extensions

While it is certainly possible to naïvely implement these methods entirely at the application level, it is best to implement them with a small amount of help from native code to achieve the desired performance improvement. As demonstrated in the previous sections, Java already provides relatively high-performance routines for I/O operations on byte arrays, so all we really need is a way to efficiently convert an array of regular primitive types to or from an array of bytes. Once this is accomplished, the converted array of bytes can be passed to the appropriate byte array I/O method of the superclass to execute the operation.

Leveraging the existing functionality of the parent classes in this way makes the implementation relatively simple and portable. Moreover, this implementation strategy is essential in the case of `BulkDataInputStream` and `BulkDataOutputStream` where the programmer is free to construct the object by composing it with any arbitrary object implementing the stream interface. In the case of `BulkRandomAccessFile` (which is not composable as a stream), we have the option of directly making calls to the underlying file system, but this approach requires intimate knowledge of the native code that implements the `RandomAccessFile` methods and is therefore inherently JVM-specific and nonportable. For this reason, we did not explore that option. Nonetheless, it is an optimization that should probably be considered when implementing the extensions for a particular JVM. In general, encapsulating array I/O within specialized bulk methods as we have done provides the Java library implementation the opportunity to optimize such methods for a particular JVM, architecture, and file system.

The only general implementation complexity that arises is maintaining the platform-independent on-disk representation required by the Java standard. Specifically, implementations of the `writeArray()` methods on a little-endian architecture (such as Intel x86) must perform a byte-swapping pass on the array data to ensure that data is written out in big-endian order as required by the Java standard [15]; an analogous transformation must take place during input using `readArray()` on little-endian machines. We implemented the bulk I/O extensions in two environments to evaluate their effectiveness: in a standard JVM using the Java Native Interface (JNI) and in a high-performance parallel dialect of Java called Titanium. We discuss each implementation below.

6.1 Implementation Using JNI

The JNI specification [24] describes an interface to native code libraries that is provided by all fully compliant JVM implementations. The JNI routines used to access arrays provide the JVM a great deal of flexibility to avoid constraining the implementation. For example, when native code requests a pointer to the elements of an array, the JVM may freely choose to return a direct pointer to the elements or return a pointer to a copy of the elements (although it must report which option it chose).[3]

Implementing the extensions using JNI was relatively straightforward; the only challenge was in reducing the number of data copies to the absolute minimum to reduce CPU and memory overheads. It turns out that, at the very least, one data copy is required to convert an array of primitive type (such as `int[]`) into a byte array. This is due to the fact that JNI abstracts away the internal in-memory representation of arrays, which prevents an in-place, zero-copy type cast. If the JVM insists on performing copies rather than providing native code with direct pointers to array elements, then the number of copies may be increased to at most three copies. However, in all the JVMs we have tested thus far, our extensions operate in single-copy mode.

An underlying assumption in the JNI implementation is that the single required copy can be performed faster than the encode/decode approach presented in Section 3. The implementation uses the `memcpy()` routine provided in the standard C library, which presumably operates close to the full memory bandwidth of the underlying architecture and, in general, should be faster than a lengthy computational loop in Java. Performance results for the JNI implementation of the extensions are presented in Section 7.

Our JNI implementation of the bulk extensions should work without modification on any standard JVM and is available for public download from [3].

[3] The JNI 1.1 specification (which corresponds to Java 1.2) adds a new `GetPrimitiveArrayCritical()` function that increases the probability of obtaining a direct pointer to an array's elements in a JVM that employs a copying garbage collector. Our implementation uses this function when it is available.

6.2 Implementation in Titanium

Titanium is a high-performance, explicitly parallel, SPMD dialect of Java developed at U.C. Berkeley for programming shared-memory and distributed-memory parallel systems. Titanium incorporates the power of Split-C [29], a low-level SPMD language, into a high-level object-oriented programming language that frees the programmer from much of the tedium associated with writing and debugging parallel programs. Titanium is almost a superset of Java 1.0 [15], including all the expressiveness and safety features of that language, with a wealth of new features that support high-performance SPMD programming, such as user-defined immutable classes, zone-based memory management, local and global references, flexible and efficient multi-dimensional arrays, unordered loop iteration, and a library of useful parallel primitives including barrier, broadcast, exchange, and various reductions [1, 20, 38]. The compiler performs extensive static analysis (with some assistance from programmer-inserted type qualifiers) to statically guarantee freedom from deadlock on barrier synchronization [14]. The primary goals of the language, in order of importance, are performance, safety, and expressiveness. Titanium is especially well adapted for writing grid-based scientific parallel applications, and several such major applications have been written and continue to be further developed [35].

The Titanium compiler performs various optimizations using knowledge of the parallel control flow and translates programs entirely to C, where they are compiled (and optimized further) by a C compiler and then linked to the proper Titanium runtime libraries (there is no JVM). The Titanium backend has been ported to several platforms, including SMPs running Solaris or POSIX threads, Solaris and Linux uniprocessors, Cray T3E, IBM SP2, IBM SPPower3, Tera MTA, SGI Origin2000, and the Berkeley NOW (a shared-nothing cluster of Ultra-SPARCs [7, 26]).

The bulk I/O extensions described in the previous section were integrated into the Titanium I/O libraries with a minimal amount of effort. The Titanium runtime system exposes the in-memory representation of arrays to native code, which allows the extensions to be implemented as a direct type cast with zero data copies in the common case (however, a single data copy is still required on little-endian platforms where byte-swapping is necessary). Note that this zero-copy implementation strategy is only valid because of the restriction to single-dimensional arrays of nonreference type; this restriction guarantees that the array element data all resides contiguously in memory and can be cast to a byte array with no data motion.[4]

Bonachea [2] investigates the performance of the Titanium implementation of the extensions on an Ultra-SPARC with a single-disk local filesystem. They report that the extensions provide a performance improvement exceeding 2x for sequential access and 40x for random access over the fastest configurations that the legacy I/O libraries have to offer. Furthermore, they show that the I/O performance of the bulk extensions is virtually identical to the I/O performance of C on that platform—this is not a terribly surprising result, because the Java code using the bulk I/O extensions is compiled by Titanium down to C code that looks very similar to the I/O code that a programmer would hand-write in C. Performance results of the Titanium implementation of the extensions on our two parallel architectures are shown in Section 7.

6.3 Safety Issues

The new bulk I/O extensions maintain the level of language safety present in the legacy Java I/O library. Safety is a very important feature of Java, and when evaluating a change or extension to the language, it is crucial to stop and consider whether the change compromises the existing safety of the language. We now sketch the reasoning why the bulk extensions don't affect the safety of Java.

Intuitively, the new bulk methods accomplish what can already be done given the existing Java I/O li-

[4] The single-dimensional restriction of the bulk I/O methods is not a serious limitation in Titanium because the language includes a more powerful structured-array abstraction called grids that provide better support for multidimensional calculations. Bonachea [2] reports the bulk I/O extension described in this paper has also been successfully adapted to work with grids and the I/O performance gains are comparable.

brary, albeit much faster. In Section 3 we demonstrated that one could use an appropriate composition of `DataOutputStream` and `ByteArrayOutputStream` objects to change an arbitrary list of Java primitive values into a single dimensional, untyped byte array using the `write*()` methods and a loop. Similarly, `DataInputStream` and `ByteArrayInputStream` allow one to extract an arbitrary list of Java primitive values from an untyped byte array. These untyped byte arrays can be used to perform bulk I/O using the existing methods in the `DataInput/DataOutput` interface (which as noted, currently only provide bulk I/O methods for byte arrays).

The bulk extensions accomplish exactly this behavior,[5] except they do it much faster by reducing the number of method calls necessary to a small constant, providing enormous speedups in practice.

7 Performance of the Extensions

We ran the same set of experiments used in Section 4 to test the performance of the bulk I/O extensions implemented with JNI and in Titanium. We compare the results for the bulk extensions to the performance of the encode/decode approach, which provided the best integer-array performance within the confines of the legacy Java I/O libraries (recall that this is the approach where the application itself performs the conversion between integers and bytes). We also measured the performance of an MPI-based, native C implementation of the same test program (with identical buffer sizes) that directly uses the `read()` and `write()` system calls to perform I/O. The results are shown in Figure 4.

For read operations on the SP, the performance of the bulk extensions in Titanium was almost identical to the native C performance. The performance using JNI was slightly degraded due to the extra copy that the JVM performs when using JNI. The performance of both implementations was vastly superior to that of encode/decode.

For writing on the SP, we observe similar results. When executing on 32 processors, Titanium achieved a thoughput of 117 MBytes/sec compared with 121 MBytes/sec for native C code. When executing on 64 processors, however, the relative performance of Titanium dropped rather significantly from 104 MBytes/sec to 75 MBytes/sec. (The reason for this drop in relative performance is unclear, and we are currently investigating it.) The JNI extensions also performed quite well, resulting in a throughput of 96 MBytes/sec with 32 processors and 87 MBytes/sec with 64 processors.

On the Origin2000, bulk I/O with Titanium again performed almost as well as native C, achieving a throughput of 536 Mbytes/sec with 16 processors and 511 Mbytes/sec with 64 processors. The JNI version performed worse than the native C and Titanium implementations, due to the extra data copy involved. However, as expected, the JNI implementation still outperformed the encode/decode approach by a significant margin. For writes on the Origin, both Titanium and JNI performed much better than encode/decode up to 16 processors, but for 32 and 64 processors, encode/decode performed slightly better. We are investigating this anomaly, but we believe we can tune the implementations of the bulk I/O extensions to achieve comparable performance.

7.1 Preliminary Results with JIT Optimizations

A fair question to ask is whether very clever compiler optimizations (for example, those provided by the best modern JIT technology) can improve the bulk I/O performance of the legacy Java I/O libraries. Preliminary tests on such systems give results very similar to those we've reported above, showing the legacy I/O libraries

[5] There is actually a very subtle difference that may arise depending on how the bulk extensions are implemented. If the "casting" operation is implemented as a literal type-cast (as in the Titanium implementation), then the byte array produced will be an alias of the typed array. Implementations in safety-critical dialects can allocate a temporary buffer and perform a single `memcpy()` operation to remedy this detail (the way arrays are handled by JNI requires this copy of any JNI-based implementation).

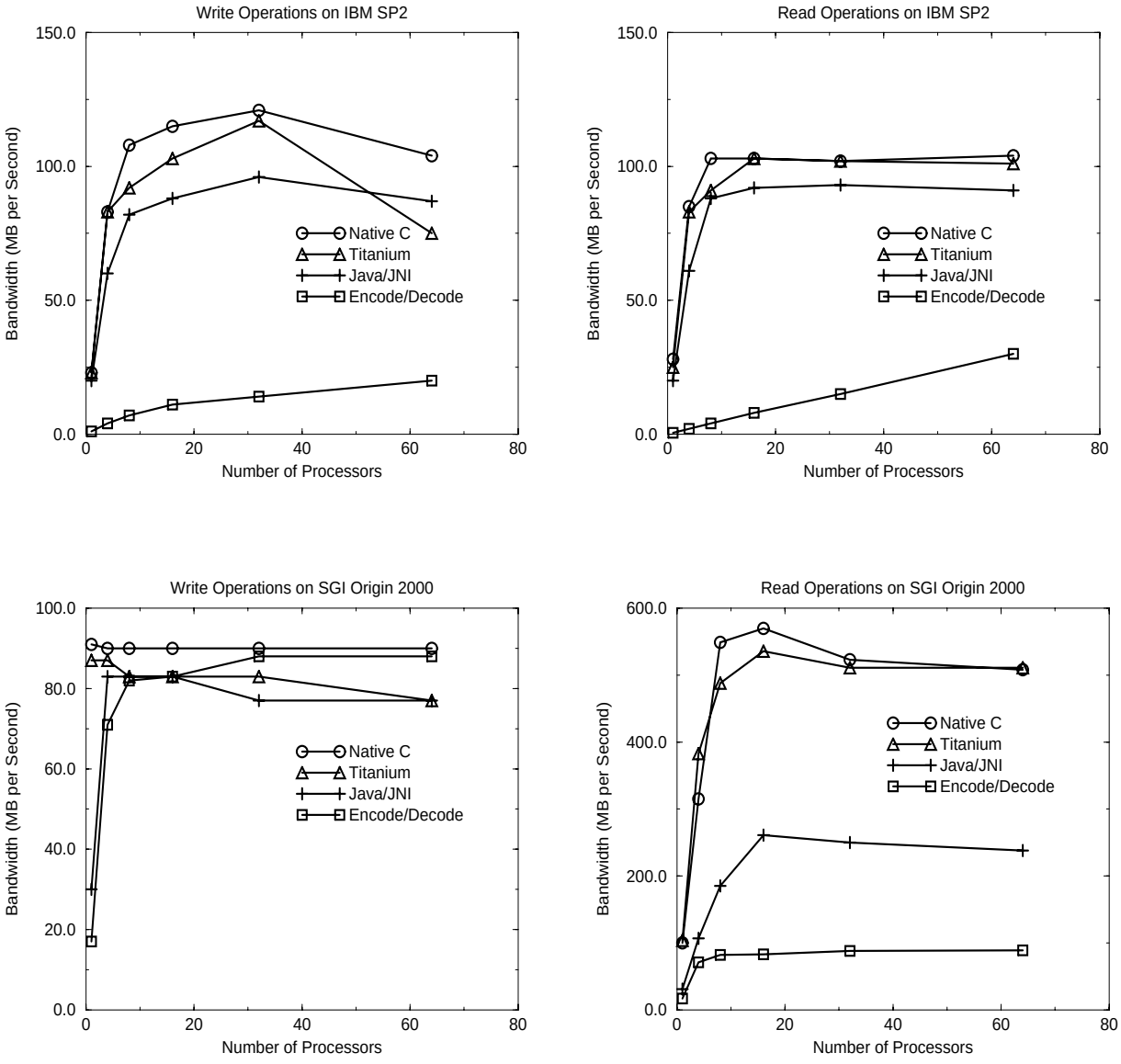

Figure 4: Performance of the bulk I/O extensions

still perform rather poorly for bulk I/O. This result is not really that surprising if one considers the nature of the code in question. The performance of the single-value-at-a-time methods on un-buffered DataInputStream/DataOutputStream and RandomAccessFile objects is fundamentally limited by the high number of system calls, and no amount of compiler optimization can remove this bottleneck. The outlook for optimizing calls to the single-value-at-a-time methods on buffered stream objects seems equally hopeless, because the flow of control is interprocedural and interclass, it performs frequent synchronization, and contains calls to native code that may throw exceptions. Even if the majority of the method calls can be removed, there is still the high overhead of synchronization, frequent buffer management operations, and numerous extra data copies. For these reasons, even the best JIT optimizers have little effect on the performance of bulk I/O using the standard legacy interfaces.

It seems the only serious candidate for achieving reasonable bulk I/O performance from the legacy Java I/O libraries through clever optimization is the encode/decode approach. It turns out this approach is significantly more amenable to traditional optimization techniques than the other approaches which use the legacy I/O libraries, although the resulting performance still falls slightly short of the bulk extensions. Recall this approach spends the majority of its computation time inside a loop similar to the one presented in Section 3.2, so let us take a moment and examine the optimization opportunities on this code. A good optimizer will perform common subexpression elimination and strength reduction on the array indexing operations, converting the four multiplication operations into a single addition operation. Similarly, a good optimizer would probably also unroll the loop several times to reduce loop overheads and allow better code scheduling. Assuming both arrays are local variables, a very clever optimizer may even be able to hoist all the array bounds checks out of the loop and into the loop preheader. However, it seems extremely unlikely that even the smartest optimizer would be capable of recognizing that this sequence of memory references and arithmetic computations is semantically equivalent to a simple bulk memory copy operation - this is a crucial transformation that is explicitly implemented in the bulk extensions and helps them to perform so much better than any other approach. Initial results reflect this intuition and show the bulk extensions still outperform the encode/decode approach by a noticeable margin, even in the presence of the best JIT optimizers.

8 Conclusions and Future Work

This work demonstrates that using the data stream methods in Java generally provides poor results, even with careful buffer size selection. Thus, to obtain reasonable performance, the application is forced to use the low-level I/O methods that read and write arrays of bytes. To use these methods, the application must itself convert the array of integers (for instance) to an array of bytes. A better solution is for Java to provide data stream methods that operate on arrays of integers and other primitive data types. This would significantly simplify the implementation of array I/O operations in Java, and would provide the Java implementation the opportunity to optimize such methods for each different platform. We have proposed extensions to the Java I/O API that support bulk (array) I/O. We have implemented these extensions using JNI and in Titanium, and our performance results indicate that they perform as well as native C code for reading/writing arrays.

A limitation of the proposed extensions is that they support file I/O on one-dimensional arrays of nonreference types only. The basic reason is that multidimensional arrays in Java are unstructured and their data elements are stored noncontiguously (multidimensional arrays are represented as a hierarchy of references to one-dimensional arrays which could possibly differ in size). In any case, a programmer could certainly perform I/O on the constituent one-dimensional fragments of a multidimensional Java array with the caveat that the application may have to store some additional application-dependent meta-information in order to recover the shape of a multidimensional array read in this fashion. It is not clear what it means to perform I/O on nonprimitive (that is, reference) types, although the object serialization approach pioneered in Java 1.1 is probably a good start.

Another limitation of the Java I/O API is that it does not support asynchronous (or nonblocking) I/O.

Asynchronous I/O can be useful for overlapping I/O with computation and communication in the program and is supported by other I/O APIs such as MPI-IO [17] and POSIX [21]. We are currently working on defining bulk asynchronous I/O extensions to Java and implementing them using JNI and in Titanium.

The bulk I/O extensions we have presented overcome the performance limitations of the lowest-level I/O methods in Java. For high-performance computing, application developers may also benefit from a higher-level parallel I/O library (such as MPI-IO [17]) for Java. Such libraries, if implemented in Java, would undoubtedly benefit from the proposed bulk extensions.

One optimization not yet explored in the JNI implementation of the bulk extensions is to recycle the internal temporary byte-array buffer in subsequent calls to the bulk I/O methods, thereby amortizing the allocation costs over many calls. We expect the results of this optimization to be somewhat application dependent; therefore, one possibility is to support it as an application-tunable option.

Acknowledgments

We would like to thank the entire Titanium team, especially Kathy Yelick and Ben Liblit, for their invaluable help. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. This material is based in part on work supported by DARPA contract No. F30602-95-C-0136, an Army Research Office grant no. DAAG55-98-1-0153, and a Sloan fellowship. The information presented here does not necessarily reflect the position or the policy of the Government or other supporting agencies, and no official endorsement should be inferred.

References

[1] Aiken, A. and D. Gay. Memory Management with Explicit Regions. In *Proceedings of Programming Language Design and Implementation Conference*, Montreal, June 1998.

[2] Bonachea, D. Bulk File I/O Extensions to Java. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 16–25, June 2000.

[3] Dan Bonachea's Home Page. `http://www.cs.berkeley.edu/~bonachea`.

[4] Bordawekar, R., Del Rosario, J., and Alok Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing '93*, pages 452-461, Portland, OR, 1993. IEEE Computer Society Press.

[5] Carpenter, B., Fox, G., Ko, S.H., and S. Lim Object Serialization for Marshalling Data in a Java Interface to MPI. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 66–71, June 1999.

[6] Crandall, P., Aydt, R., Chien, A., and D. Reed Input-Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, ACM press, December 1995.

[7] Culler, D. et al. Parallel Computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing*, 1997.

[8] Del Rosario, J., Bordawekar, R., and Alok Choudhary. Improved Parallel I/O via a Two-Phase Run-Time Access Strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems* pages 56-70, Newport Beach, CA, 1993.

[9] Del Rasario, J. and A. Choudhary. High Performance I/O for Parallel Computers: Problems and prospects. IEEE Computer, 27(3):59-68, March 1994.

[10] Dickens, P. and R. Thakur. A Performance Study of Two-Phase I/O. In *Proceedings of the 4th International Euro-Par Conference*. Lecture Notes in Computer Science 1470. Springer-Verlag, pages 959-965, September 1998.

[11] Dickens, P. and R. Thakur. An Evaluation of Java's I/O Capabilities for High-Performance Computing. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 26–35, June 2000.

[12] Dickens, P. and R. Thakur. On Implementing High-Performance Collective I/O. Submitted to The Journal of Parallel and Distributed Computing

[13] Feitelson, D., Corbett, P., Baylor, S., and Y. Hsu. Parallel I/O Subsystems in Massively Parallel Supercomputers. In *IEEE Parallel and Distributed Technology*, 3(3):33-47, Fall 1995.

[14] Gay, D. and A. Aiken. Barrier Inference. In *Proceedings of Principles of Programming Languages Conference*, San Diego, January 1998.

[15] Gosling, J. and G. Steele. The Java Language Specification, Addison-Wesley, June 1996.

[16] Gropp, W., Lusk, E., and A. Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. Second Edition. The MIT Press, Cambridge, Massachusetts, 1999.

[17] Gropp, W., Lusk, E., and R. Thakur. Using MPI-2: Advanced Features of the Message-Passing Interface. The MIT Press, Cambridge, Massachusetts, 1999.

[18] Harold, E.R. Java I/O. O'Reilly & Associates, March 1999.

[19] Heydon, A. and M. Najork. Performance Limitations of the Java Core Libraries. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 35–41, June 1999.

[20] Hilfinger, P. Titanium Language Working Sketch, rev 0.22, September 1999.

[21] IEEE/ANSI Std. 1003.1. Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language], 1996 edition.

[22] Kotz, D. and N. Nieuwejaar. Dynamic File-Access Characteristics of a Production Parallel Scientific Workload. In *Proceedings of Supercomputing '94*, pages 640-649, November 1994.

[23] Kotz, D. Disk-Directed I/O for MIMD Multiprocessors. it ACM Transactions on Computer Systems, 15(1):41-74, February 1997.

[24] Liang, S. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, 1999.

[25] MPI-Java Home Page. `http://www.npac.syr.edu/projects/pcrc/HPJava/mpijava.html`.

[26] NOW Project web page. `http://now.cs.berkeley.edu/`.

[27] Parallel I/O Archive. `http://www.cs.dartmouth.edu/pario`.

[28] Seamons, K., Chen, Y., Jones, P., Jozwiak, J., and M. Winslett. Server-Directed Collective I/O in Panda. In *In Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[29] Split-C Project web page. `http://www.cs.berkeley.edu/Research/Projects/parallel/castle/split-c/`.

[30] Sun Microsystems Java 1.1 Documentation. `http://java.sun.com/products/jdk/1.1/docs.html`

[31] Thiruvathukal, G., Dickens, P., and S. Bhatti. Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda, Actors, and the Message Passing Interface. Submitted to *Concurrency: Practice and Experience*.

[32] Thakur, R. and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming* 5(4):301-317, Winter 1996.

[33] Thakur, R., Choudhary, A., More, S, and S. Kuditipudi. Passion: Optimized I/O for Parallel Applications. *IEEE Computer*, 29(6):70-78, June 1996.

[34] Thakur, R., Lusk, E., and W. Gropp. I/O in Parallel Applications: The Weakest Link. *International Journal of High Performance Computing Applications*, 124:389–395, Winter 1998.

[35] Titanium Project web page. `http://www.cs.berkeley.edu/Research/Projects/titanium/`.

[36] Welsh, M. and D. Culler. Jaguar: Enabling Efficient Communication and I/O from Java. To appear in *Concurrency: Practice and Experience*, Special Issue on Java for High-Performance Applications.

[37] Welsh, M. Tigris: A Java-Based Cluster I/O System Technical report, June 1999.

[38] Yelick, K. et al. Titanium: A High-Performance Java Dialect. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, CA, February 1998.