An Evaluation of Java's I/O Capabilities for High-Performance Computing

Phillip M. Dickens
Illinois Institute of Technology
10 West 31st Street
Chicago, Illinois 60616
pmd@work.csam.iit.edu

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

ABSTRACT

Java is quickly becoming the preferred language for writing distributed applications because of its inherent support for programming on distributed platforms. In particular, Java provides compile-time and run-time security, automatic garbage collection, inherent support for multithreading, support for persistent objects and object migration, and portability. Given these significant advantages of Java, there is a growing interest in using Java for high-performance computing applications. To be successful in the high-performance computing domain, however, Java must have the capability to efficiently handle the significant I/O requirements commonly found in high-performance computing applications.

While there has been significant research in high-performance I/O using languages such as C, C++, and Fortran, there has been relatively little research into the I/O capabilities of Java. In this paper, we evaluate the I/O capabilities of Java for high-performance computing. We examine several approaches that attempt to provide high-performance I/O—many of which are not obvious at first glance—and investigate their performance in both parallel and multithreaded environments. We also provide suggestions for expanding the I/O capabilities of Java to better support the needs of high-performance computing applications.

1. INTRODUCTION

There is a growing interest in the use of Java for high-performance computing, stemming from Java's significant support for programming on distributed platforms. This support includes compile-time and run-time security that can be used as the basis for writing secure applications. Java also provides inherent support for multithreading, allowing the overlapping of computation with communication or I/O. Java can save the state of an object and recreate that object on another machine, supporting both persistent objects and object migration. Java provides automatic garbage collection, alleviating the programmer from memory management. Perhaps the greatest benefit of Java is its portability: a Java application can be executed on any platform with Java support.

Despite the many advantages of Java-based computation, it is still unclear whether Java has the capability to support the significant I/O demands found in large scientific applications. In this paper, we investigate the I/O capabilities of Java for high-performance computing. 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 investigate I/O mechanisms defined in Java that can be used to take advantage of such parallel file systems and study the performance implications of each such approach. Finally, we provide suggestions for relatively simple changes to the Java I/O model that can significantly improve performance.

1.1 Contributions of this Paper

The key contribution of this paper is that it provides a detailed discussion and performance analysis of several approaches to parallel file I/O available in Java and does 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. To make our results as general as possible, we do not consider any approaches that cannot be performed by a user at the application level.

1.2 Organization

The rest of this paper is organized as follows. In Section 2 we discuss I/O in high-performance computing applications. In Section 3 we provide background information on the I/O mechanisms defined in Java. We describe several approaches for performing parallel file I/O in Java in Section 4. Experimental results are presented in Section 5. We offer simple suggestions for improving the Java I/O model in Section 6. Related work is discussed in Section 7, followed by conclusions in Section 8.

2. 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 [3, 13, 22]. 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 later on this paper, the fact that they are not just byte arrays is important in the context of using Java for I/O in these kinds of 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, the offset in the file, and 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 [7]. 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 [10], that are implemented in an optimized fashion for different machines and file systems.

The situation with Java, however, is quite different. Because of various Java language constraints, performing parallel file I/O in Java is a much more complex issue. This is the focus of the next section.

3. 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 [11].

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 `OuputStream`. 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.

3.1 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 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:

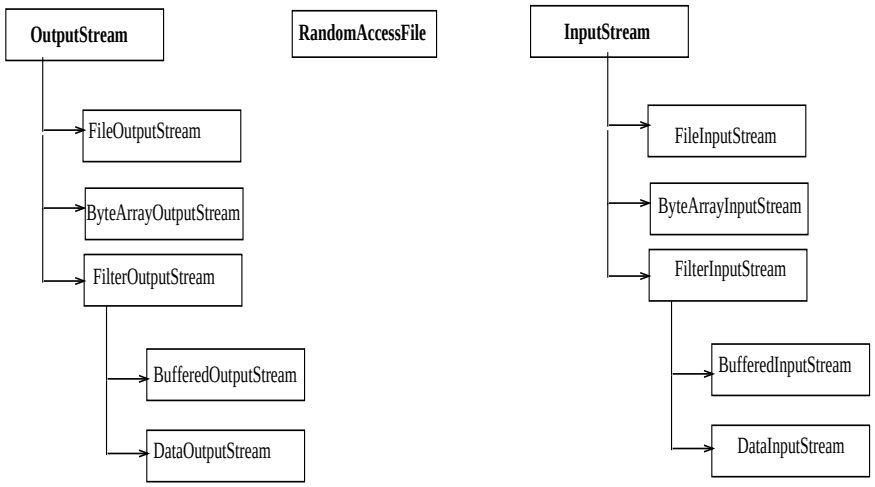

Figure 1: This figure shows the I/O class hierarchy pertinent to this investigation. Note that the `RandomAccessFile` class is completely outside of the `InputStream` and `OutputStream` hierarchy. As discussed in Section 3.5, however, a connection can be made between a `RandomAccessFile` and a `FileInputStream` or `FileOutputStream`.

```
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 4.

3.2 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 a *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 file descriptor 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 file descriptor object. The constructor for this class most pertinent to our discussion takes as a parameter a file descriptor 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.

3.3 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()
```

Note that the `toByteArray` method is synchronized: accesses to this method are serialized by the implementation.

3.4 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.

3.4.1 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.

3.4.2 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).

3.5 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.

4. 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.

4.1 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 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 [18].

4.2 Converting to/from an Array of Bytes

As we shall see in Section 5, 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, 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):[1]

```
// Assume we are converting bytes 0 to 3 of a byte
// array (buf) into element 0 of an integer array.

  int temp;
  for (int i=0; i<4; i++) {
      temp = (int) (buf[i]);
      temp = temp & 255;
      temp = temp << (i * 8);
      int_array[0] = int_array[0] | temp;
  }
```

4.3 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.

```
// 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]);
```

4.4 Using Buffered Data Streams

As we shall see in Section 5, 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.

We note, however, that it is *not* safe to use buffered data streams for *writing* concurrently from multiple processes or threads to a common random-access file.[2] 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.

```
  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++)
```

[1] We thank an anonymous referee for suggesting this solution.

[2] We again thank an anonymous referee for bringing this issue to our attention.

```
dos.writeInt(int_array[i]);
```

4.5 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);
```

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());
```

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.read(buf, 0, num_bytes_to_read);
  for (int i = start_buf;
         i < (start_buf + num_ints_to_read); i++)
      int_array[i] = dis.readInt();
```

4.6 Other Approaches

There are two other approaches that we did not investigate. One approach is to use the JNI interface and call Unix I/O functions in C. We did not use this approach for two reasons. First, it significantly reduces the portability of the code. Second, we were interested in evaluating the performance of the I/O methods defined in Java itself.

The other approach we do not report on is the use of object serialization to perform I/O. We did explore 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 [2].

5. PERFORMANCE RESULTS

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.

5.1 Computational Platforms

We conducted experiments on both a shared-memory and a distributed-memory parallel machine. The distributed-memory machine was an IBM SP located at Argonne National Laboratory. This machine has 80 compute nodes and 4 I/O processors. Each I/O processor controls four SSA disks, each of 9 Gbyte capacity. The shared-memory machine used in these experiments was an SGI Origin2000, also housed at Argonne National Laboratory. This machine is configured with 128 compute processors and ten Fibre Channel controllers connected to a total of 110 disks of 9 Gbyte capacity each. Both machines have parallel/high-performance file systems, namely, PIOFS on the SP and XFS on the Origin2000.

On the shared-memory Origin2000, we wrote a multithreaded Java program, each thread running on a separate processor. The threads all shared the array to be read or written, but each thread operated on a distinct subarray region. The shared array was divided equally among all the threads. Similarly, all threads accessed distinct portions of the file. Each thread wrote 32 Mbytes at a time several times, resulting in a total file size of 1 Gbyte. We used version 3.1.1 of SGI's Java software, which was conformant with the behavior of Sun's JDK 1.1.6.

On the IBM SP, which is a distributed-memory machine, we wrote a multiprocess parallel program. Each process ran on a different node of the SP (and a different Java Virtual Machine). We could have simply spawned a Java process on each node, but our parallel program also needed some additional information that MPI [9] 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 [19] or an MPI wrapper for Java [15]. 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. After the Java program completed, it returned to the MPI program, which then accumulated performance statistics. Each Java process had its own private array, but all processes shared the global file. We used a 4 Mbyte array per process, based on previous experiments that have shown this to be a good size for performing I/O on this SP. Each process wrote multiple times resulting in a total file size of 1 Gbyte, as on the Origin2000. We used IBM's Java software, which was conformant with the behavior of Sun's JDK 1.1.2.

5.2 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 4, 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 every element of the array. With 64 threads and a 1 Gbyte array, each thread must make over four million calls to the `readInt` or `writeInt` methods. With a single thread, 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 [12].

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.

5.2.1 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 the first approach (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, due to the fact that the extra computation resulted in the separation in time of some of the concurrent write requests. The best write performance was obtained using the first approach with 32 processors (resulting in a bandwidth of 106 Mbytes/sec). The best result obtained when using the second approach (conversion from integers to bytes) was 20 Mbytes/sec with 64 processors. The maximum throughput observed across all the other experiments 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 the first approach 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 the second approach 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.

5.2.2 Results on the SGI Origin2000

It is interesting to note that on the Origin2000, the second approach, where the application performed the conversion between integers and bytes itself, outperformed the first approach when writing with 64 processors. The reason for this disparity again has to do with contention. As noted above, the extra computation of the second approach has the effect of separating in time some of the concurrent write requests. This approach resulted in a throughput of 108 Mbytes/sec with 64 processors. The best performance observed using data streams was 4.1 Mbytes/sec, obtained using buffered output streams with a 0.5 Mbyte buffer.

The first approach resulted in excellent performance for the read operations. For example, a throughput of 631 Mbytes/sec was observed when using 16 processors. Again we see a decrease in performance when increasing the number of processors to 64 because of increased contention for I/O resources. The second approach 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.

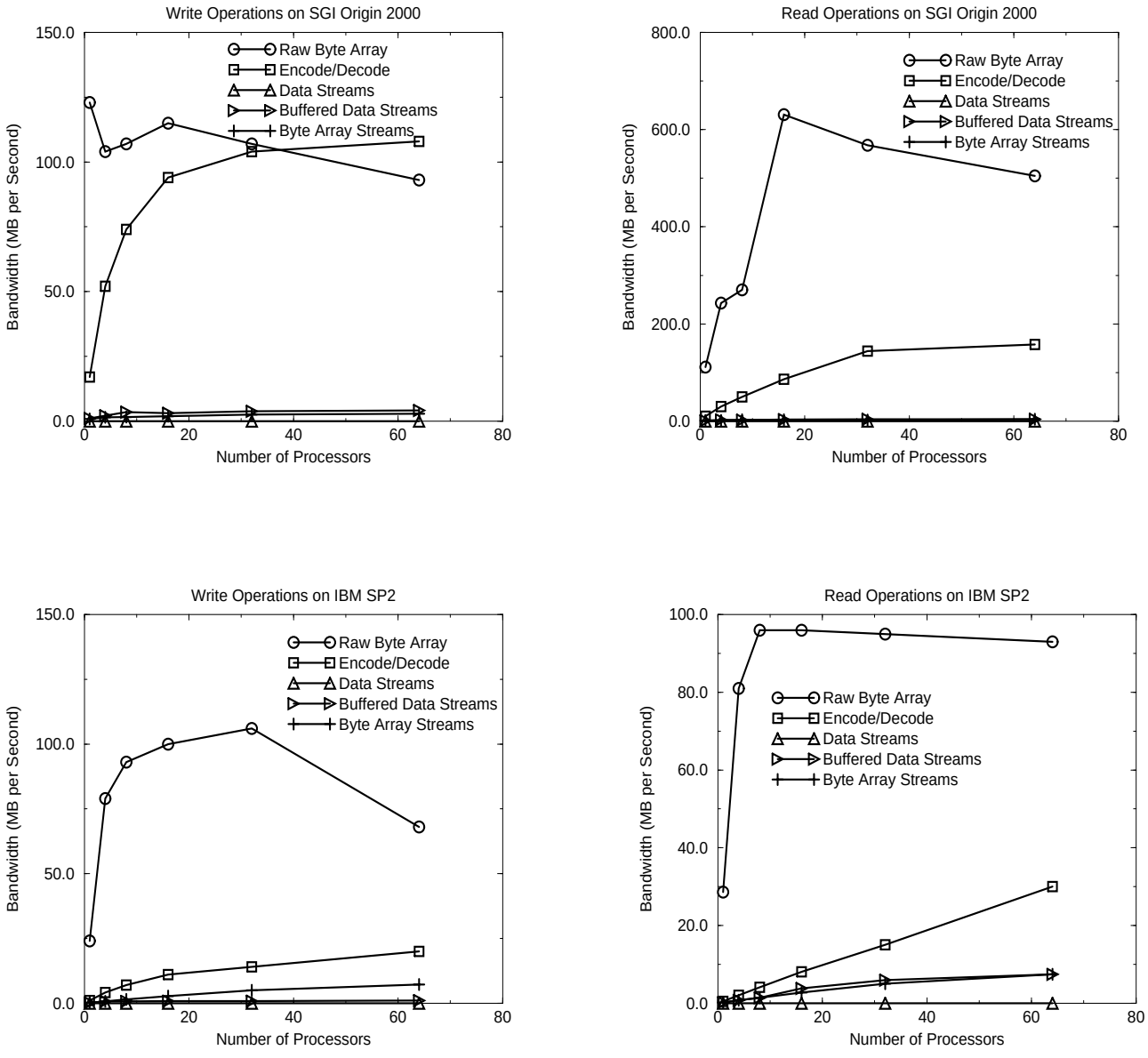

Figure 2: The performance of various approaches to high-performance file I/O in Java

6. SUGGESTIONS FOR IMPROVING JAVA I/O PERFORMANCE

The above results demonstrate that the I/O methods that directly read/write arrays of bytes are the only methods 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.

A relatively simple fix to these problems is for Java to provide data stream methods that read/write arrays of all the primitive data types. Since Java already knows how to write a single data type as a sequence of bytes and to read a single data type from a sequence of bytes, it can easily be extended to read or write an array of data types. This fix would not only eliminate the need for many expensive I/O or method calls, but it would also provide the Java implementation the opportunity to optimize such methods for a particular machine and file system.

This extension would require the introduction of six new methods for writing and another six for reading. The suggested write methods are shown below; the read methods are analogous and are not shown.

```
writeShortArray (short [] data)
writeCharArray  ( char [] data)
writeIntArray   (  int [] data)
writeLongArray  ( long [] data)
writeFloatArray (float [] data)
writeDoubleArray(double[] data)
```

While it is certainly possible to implement these methods at the application level (as done in this study), implementing them natively as part of the language should provide much better performance. These methods do not solve the problem for multidimensional arrays. However, multidimensional arrays can be accessed by calling the methods for one-dimensional arrays several times.

The proposed methods overcome the performance limitations of the lowest-level I/O methods in Java. For high-performance computing, application developers would also need a higher-level parallel I/O library (such as MPI-IO [10]) for Java. Such libraries, if implemented in Java, would undoubtedly benefit from the proposed methods.

Finally, we note that the proposed methods are not just useful for I/O, but also for interprocess communication, and would therefore benefit networking applications as well.

7. RELATED WORK

Other than the large body of work related to parallel I/O [1, 4, 5, 8, 14, 16, 17, 20, 21], the work most closely related to ours is the Jaguar project [23, 24], 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. It seems clear that modifications to Java at all levels will be necessary to provide truly high-performance file I/O.

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.

8. CONCLUSIONS

In this paper, we have investigated the capabilities of Java for high-performance file I/O. 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 data types. This would significantly simplify the implementation of parallel I/O operations in Java, and would provide the Java implementation the opportunity to optimize such methods for each different platform.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

9. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] DelRosario, 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.

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

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

[17] 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.

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

[19] 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.*

[20] 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.

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

[22] 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.

[23] 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.

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