Towards Efficient Support for Parallel I/O in Java HPC

Ammar Ahmad Awan[1], Muhammad Sohaib Ayub[2], Aamir Shafi[2], Sungyoung Lee[1]

[1]Department of Computer Engineering, Kyung Hee University, South Korea
{ammar, sylee}@oslab.khu.ac.kr
[2]SEECS, National University of Sciences and Technology, Pakistan
{sohaib.ayub, aamir.shafi}@seecs.edu.pk

Abstract— **Modern HPC applications put forward significant I/O requirements. To deal with them, MPI provides the MPI-IO API for parallel file access. ROMIO library implements MPI-IO and provides efficient support for parallel I/O in C and Fortran based applications. On the other hand, Java based MPI-like libraries such as MPJ Express and F-MPJ have emerged but they lack parallel I/O support. Little research has been done to provide Java based ROMIO-like libraries due to the non-availability of MPI-IO-like API for the Java language. In this paper, we take the first step towards the development of parallel I/O API in Java by evaluating the newly introduced Java NIO API versus the legacy Java I/O API. We propose two simple approaches for performing parallel file I/O using NIO and evaluate them on two different computational platforms. The implementation of proposed approaches exploits the view buffers concept of NIO API to perform efficient array based file I/O operations from multiple processes. We report encouraging speedups and suggest that design of a parallel I/O API in Java should be based on the NIO API.**

Keywords--Performance Evaluation; Parallel I/O in Java; MPJ Express; MPI-I/O;

I. INTRODUCTION

Message Passing Interface (MPI) has become the de facto standard for writing parallel applications that run on distributed memory machines. MPI provides language bindings for C and Fortran, which continue to be the mainstream High Performance Computing (HPC) languages. Large-scale HPC applications put forward significant I/O requirements, which are classically considered a major bottleneck in the performance of parallel applications [19]. MPI deals with these requirements by providing the MPI-IO API [4]. ROMIO [5] library fully implements the MPI-IO API and provides efficient parallel I/O for C and Fortran based applications. Popular implementations of MPI including MPICH-2 [23] and Open MPI [24] contain the ROMIO library.

On the other hand, Java has been adopted as an alternative HPC language. Significant interest in developing messaging libraries for Java led to the formation of the Java Grande Forum [1], which developed the mpiJava 1.2 API specification [2] based on the MPI standard (version 1.2). This resulted in several implementations of the mpiJava API in the form of MPI-like Java libraries such as MPJ Express [3], MPJ/Ibis [22] and F-MPJ [20]. The main problem is that all of these Java libraries lack parallel I/O support. Little research has been done to develop ROMIO-like libraries for Java. The reason for unavailability of parallel I/O support in existing Java libraries is that mpiJava specifications do not contain an MPI-IO equivalent. We believe that developing such an API for Java is necessary to guide the development of Java based parallel I/O libraries. In this paper, we take the first step towards building a Java parallel I/O API by evaluating Java I/O and NIO APIs. We surveyed existing studies and found that very few researchers have focused on evaluating Java's I/O capabilities in this particular context.

The main contribution of this paper is that we propose parallel file I/O approaches based on view buffers concept of Java NIO API and evaluate them against existing approaches in Java I/O API across two different parallel computing platforms. To date, there has been relatively little research focused on evaluation of Java NIO in the context of parallel file I/O support for Java HPC libraries. Our performance evaluation suggests that the design of Java parallel I/O API should be based on Java NIO API as it natively provides the most efficient parallel file I/O methods.

The rest of the paper is organized as follows. In Section II, we discuss I/O requirements in HPC applications. In Section III, we provide a review of the Java NIO API. In Section IV, we describe the proposed approaches to parallel file I/O using Java NIO. Section V contains explanation of the computational platforms, benchmarking methodology and the analysis of results. Section VI contains the related work and Section VII concludes this paper.

II. I/O REQUIREMENTS IN HPC APPLICATIONS

Modern HPC applications contain large data structures distributed across multiple processes. These data structures are usually arrays of primitive datatypes that are read from and written to files. C based applications can use type casting to easily convert arrays or even portion of arrays of different datatypes to byte arrays. Multidimensional arrays in C can easily be treated as single dimensional arrays. C applications can improve performance by use native file system calls and hints to the file system e.g. O_DIRECT on Network File System (NFS) to turn off client-side data caching. On the other hand, Java lacks these facilities and its I/O interface is potentially much more restrictive than C. Java applications cannot directly pass hints to the file system like the earlier mentioned O_DIRECT. Moreover, Java I/O API doesn't support array based (bulk) read/write operations on files for primitive datatypes other than bytes. Real world applications however, do need to use integer, floating point or other primitive datatype arrays. Java NIO does support array based read/write operations but through an indirect view buffer approach. We will elaborate and discuss how the view buffers approach can be used to perform array (or bulk) I/O operations in Section IV. In this paper, we are mainly investigating the problem of array based read/write operations on a single file from multiple Java processes (and threads). This is an important research area in the context of developing a parallel I/O API for Java.

III. REVIEW OF JAVA NIO API

To fully understand the issues related to parallel file I/O in Java, it is necessary to briefly review the Java NIO API. The Java I/O API was primarily based on *streams* of data which operate on bytes whereas the Java NIO API is based on *channels* which operate on blocks of data. Review of Java I/O API can be studied from [9]. The following sub-sections discuss the class hierarchy shown in Figure 1.

A. Buffer class

The `java.nio.Buffer` class is the foundation of Java NIO API. A `Buffer` object acts as a container of data for a specific primitive data-type. It has three important parameters; `position`, `limit` and `capacity`. It is the base class for the primitive data-type buffers. Example of a primitive data-type buffer is `ByteBuffer` for the `byte` data-type.

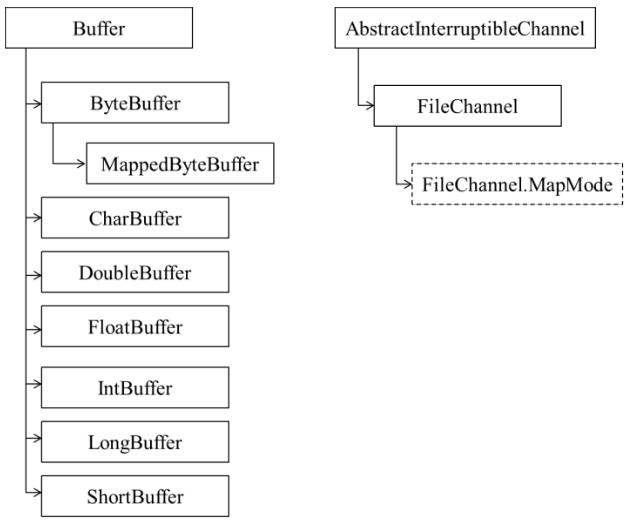


Figure 1. Hierarchy of the NIO classes relevant to our discussion

B. ByteBuffer class

`ByteBuffer` class extends the `Buffer` class. All `ByteBuffer` objects are based on a backing byte array. The method below is used for creating `ByteBuffer` objects.

```
public static ByteBuffer allocate(int capacity)
```

`ByteBuffer` object's position is controlled in byte offsets. In many practical applications we want to use primitive data-type offsets and Java supports this by providing buffers of primitive data-types. `IntBuffer` has been discussed as an example of primitive data-type buffers. All other types have similar properties so they are not discussed individually.

C. IntBuffer class

`IntBuffer` objects like `ByteBuffer` objects are backed by an integer array. This helps to change position of the `IntBuffer` using integer offsets. The method to create an IntBuffer is:

```
public static IntBuffer allocate(int capacity)
```

We can call `asIntBuffer()` method on a `ByteBuffer` object which is useful for parallel read/write operations as discussed further in Section IV.

D. FileChannel class

The `FileChannel` class resides in the `java.nio.channels` package. The `FileChannel` object can be created for a file by invoking `open()` methods provided by the class. `FileChannel` objects are thread safe and can be used by multiple concurrent threads. `FileChannel` can perform I/O operations based on a `ByteBuffer` object only. The signatures for performing read and write operations on a file are:

```
public int read(ByteBuffer destination)
public int write(ByteBuffer source)
```

The `position` of the `FileChannel` object is important for parallel reads and writes and can be set by calling `position(long newPosition)` method.

E. FileChannel.MapMode class

`FileChannel` is enclosing class of `FileChannel.MapMode`. The `FileChannel.MapMode` is type-safe enumeration for file-mapping modes. Java provides support for memory mapped I/O, which is a technique to map regions of file into the main memory. I/O operations performed on mapped regions of memory are faster than on physical disk. In Java, memory mapping is recommended for large file sizes only [15]. The mapped region of a file is reflected as an in-memory `MappedByteBuffer` object which can be acquired by calling the following method of the `FileChannel` class.

```
MappedByteBuffer map(FileChannel.MapMode mode,
                          long position,long size)
```

The changes done to mapped buffer object are reflected in the file so there is no need for calling the usual `write()` and `read()` methods of a `FileChannel` object.

IV. APPROACHES TO PARALLEL FILE I/O IN JAVA

We propose two new parallel file I/O approaches based on the Java NIO API. We will refer them as:

- Using FileChannel with View Buffer
- Using FileChannel in Mapped Mode

We observed that Java NIO API can facilitate bulk (array) I/O operations on a file while the legacy Java I/O API had no such support. Bulk I/O operations can be achieved by using methods defined by `ByteBuffer` class that create *views* of a given `ByteBuffer` object. These methods are of the form `as<<type>>Buffer()` where `<<type>>` includes all primitive data-types other than bytes. We are using `asIntBuffer()` for `int` data-type in our discussion but the same holds true for all other primitive data-types. A view buffer is simply another buffer whose content is backed by the byte buffer. We exploit this functionality, in our proposed approaches, to perform memory operations on the view buffer and use the backing `ByteBuffer` object for I/O operations on a file using the `FileChannel` object. In addition to the proposed approaches, we describe two other approaches for parallel file I/O (listed below) based on the legacy Java I/O API originally discussed in [9].

- Using RandomAccessFile
- Using BulkRandomAccessFile

The details and pseudo code of existing and proposed parallel file I/O approaches is discussed in the upcoming sub-sections.

A. *Using FileChannel with View Buffer*

In this approach, we are exploiting the view buffer facility provided by the `ByteBuffer` class. The pseudo code for this approach is written in Figure 2. Each process creates an integer array with length equal to `count`. Each process creates a `FileChannel` object and allocates a `buf` object of size `myInts*4` where `myInts` contains the count of integers for each process calculated as `count / numProcs`. Next, each process connects `buf` object to the `ibuf` object and then puts `myInts` integers from the `int_array` to the `ibuf` object. Each process then seeks to the correct file position, locks the processes' specific portion of the file, writes `buf` object to the file, releases the lock and then pushes the data and metadata changes to the file system. We have used the `FileLock` class to lock the process specific portion of the file before writing to it. `FileLock` class uses `fcntl()` locks on Unix file systems and is necessary for correct behavior in case of parallel writes on NFS. File locks are held on behalf of the entire Java virtual machine and are not suitable for controlling access to a file by multiple threads within the same virtual machine. We note that this approach avoids the overhead of explicit conversion from integers to bytes which can become a significant bottleneck for arrays of large sizes.

```
// each process executes the following code

int int_array[] = new int[count];
int numProcs = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();

FileChannel fc =
  FileChannel.open(Paths.get(fname,options);

ByteBuffer buf = ByteBuffer.allocate(myInts*4);

IntBuffer ibuf = buf.asIntBuffer();

long myPos = count/numProcs * rank;
long myInts = count/numProcs;

ibuf.put(int_array, myPos, myInts);

long fSize = myInts*4;
long fPos = myPos*4;

fc.position(fPos);

// lock the file for correct parallel writes

FileLock fl = fc.tryLock(fPos, fSize, false);

fc.write(buf);

fl.release();

fc.force(true);

```

Figure 2. Pseudo-code for Using FileChannel with View Buffer approach

B. *Using FileChannel in Mapped Mode*

Figure 3 contains the pseudo code for this approach. Each process creates the array and its own `FileChannel` object. Each process creates a `MappedByteBuffer` object based on specific region of the array. Next, each process connects its `ibuf` object to `mbbuf` object and puts `myInts` integers from `int_array` to the `ibuf` object. Unlike previous approach, `fc.write()` method is not explicitly called, as the changes to the `ibuf` object reflect to the `mbbuf` object and to the file as well. We note that we are not using explicit locking here because memory mapped portions are locked by the JVM itself. The standard JDK documentation suggests that locking and mapping should not be used simultaneously as it might fail on certain systems.

```
// each process executes the following code

int int_array[] = new int[count];
int numProcs = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();

FileChannel fc =
  FileChannel.open(Paths.get(fname,options);

long myPos = count/numProcs * rank;
long myInts = count/numProcs;

MappedByteBuffer mbbuf =
 fc.map(MapMode.READ_WRITE, myPos*4, myInts*4);

IntBuffer ibuf = mbbuf.asIntBuffer();

// lock the file for correct parallel writes

FileLock fl = fc.tryLock(fPos, fSize, false);

ibuf.put(int_array, myPos, myInts);

// We do not use fc.write() here as the changes
// done to buffer are reflected in file.

fl.release();

mbbuf.force();

```

Figure 3. Pesudo code for Using FileChannel in Mapped Mode approach

C. Using RandomAccessFile

The `RandomAccessFile` resides alone in the `java.io` hierarchy and duplicates the functionality of `InputStream` and `OutputStream` class hierarchy. `RandomAccessFile` class provides reading and writing of primitive data-types by implementing `DataInput` and `DataOutput` interfaces. The example of one such method for writing is `writeInt(int value)` and reading is `readInt()`. But there exist no methods in `java.io` hierarchy that directly read/write arrays of primitive data types. In addition, it provides a method for skipping (or seeking) bytes that is essential for parallel reads and writes. The method's signature is:

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

We originally planned to provide the pseudo-code and benchmark for this approach but our preliminary evaluation showed that this approach is extremely inefficient, so we omitted the benchmarks and results for this approach.

D. Using BulkRandomAccessFile

`BulkRandomAccessFile` is a custom built Java class which duplicates the functionality of `RandomAccessFile` class of Java. `BulkRandomAccessFile` is not part of the standard JDK but was presented in [25] and can easily be used in Java programs by including its package hierarchy. The authors presented the extensions for Java and Titanium language but we are discussing only Java version in this paper. The `BulkRandomAccessFile` class provides new methods for reading and writing arrays of primitive data-types. These methods claim to perform significantly better than the `RandomAccessFile` class methods for writing primitive data-types [9],[25]. These methods are overloaded for all primitive data-types but we are only using the `int` version in this paper. The signature for reading and writing integers is:

```
void readArray(int[] array, int offset, int count)
void writeArray(int[] array,int offset, int count)
```

The pseudo-code for this approach is shown in Figure 4. Each process creates an integer array with length equal to `count` and a `BulkRandomAccessFile` object. Next, each process seeks to the correct position, acquires the associated `FileChannel` object, locks the portion of file using this channel, writes `myInts` integers to the file using the bulk `write()` method and finally releases the lock. We have used `"rws"` as the `access` mode for opening the `BulkRandomAccessFile`, as its effect is equivalent to the `fc.force(true)` and `mbbuf.force()` method, i.e. to force both the data as well as metadata changes to the file system. It is interesting to note that legacy Java I/O API does not directly provide locking capabilities but it can be used by acquiring the associated `FileChannel` object and using locks on it.

```
// each process executes the following code

int int_array[] = new int[count];
int numProcs = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();

BulkRandomAccessFile braf =
      new BulkRandomAccessFile(filename,"rws");

long myPos = count/numProcs * rank;
long myInts = count/numProcs;

long fSize = myInts*4;
long fPos = myPos*4;

braf.seek(myPos);

FileChannel fc = braf.getChannel();

FileLock fl = fc.tryLock(fPos, fSize, false);

braf.write(int_array, myPos, myInts);

fl.release();

```

Figure 4. Pseudo code for Using BulkRandomAccessFile approach

V. Performance Evaluation

We wrote two different versions of our benchmarking code. The first version uses Java threads for evaluating performance on a shared memory machine, while the second uses MPJ Express processes for evaluating performance on a distributed memory machine.

A. Computational Platforms

The distributed memory machine we have used, is an HP ProLiant DL160SE G6 Server based Cluster comprising of 34 nodes. Each node has an Intel Xeon processor providing 8 physical cores per node and 24GB of memory per node. The cluster has a total memory of 816 GB and is based on RHEL 5.5 operating system. The head node of the cluster is directly connected to the SAN storage (22TB raw capacity) using Fibre Channel host bus adapter (HBA) and switch. All other nodes connect to the head node via the QDR Infiniband network with a theoretical peak bandwidth of 40Gbps. "Infiniband over IP" has been used for data sharing. The storage has been mounted using the popular NFSv3 but we plan to upgrade the cluster to use a modern parallel file system such as PVFS2 or Lustre. Each node has a Gigabit Ethernet card to connect to the Gig-E network as well but is not used for accessing the file system. Each node contains a physical disk drive, which uses the ext3 file system.

The shared memory machine is just one node of the distributed memory cluster, with a total of 8 physical cores (16 logical cores as each core has two threads).

B. Analysis of Results

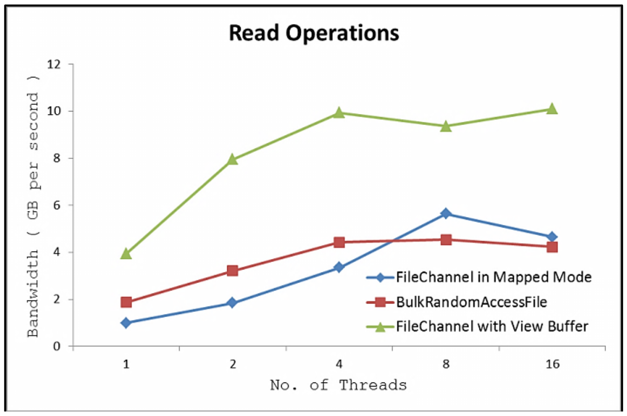
We have used three different configurations for our evaluation. All configurations use a shared array of 256 million integers accumulating to total data size of 1 GB. Each thread or process performs I/O operations on disjoint locations in the file. We execute these operations several times and use the average value of the time to generate performance graphs. The bandwidth shown in the graphs is the aggregate bandwidth calculated by diving data-size over time where time is the sum of time taken by each thread divided by number of threads. The consistency of file's data and metadata (file attributes) are important when multiple processes read and write simultaneously to a single file. In our experiments, we are executing the processes on a distributed memory machine, which connects to a Storage Area Network (SAN) server. This storage is mounted using the Sun NFS version 3 using the "-noac" option to turn off attribute caching. This poses a performance penalty but is necessary for consistency of metadata. To maintain data consistency when reading from and writing to a file by multiple processes, locking is necessary on NFS and `fcntl()` lock is the default mechanism to guarantee correct behavior. The alternate option to maintain data consistency is by disabling client-side data caching by using O_DIRECT flag when opening a file. This facility however is not available for Java applications so we are using the `FileLock` Java class which internally uses `fcntl()` locks on Unix file systems. We first lock the Process's specific portion of the file and write its portion of array to the file and then unlock that portion of the file. Each process does this to ensure correctness of write operation. The read operation follows a similar approach and is done after the write operation has fully completed. We are not considering the cases when processes read/write simultaneously to a file or read/write to overlapping regions of a file to avoid the problems of consistency and atomicity. These cases are beyond the scope of this paper. The three configurations mentioned previously are explained below along with a discussion related to respective graphs for each configuration.

Threads based I/O on local disk: This configuration uses Java threads. Each thread seeks to an appropriate distinct location in the file and writes its own chunk of shared integer array to the shared file. A similar approach is followed for reading the integers from the file to the array. The file resides on the local disk drive (ext3 file system) of the shared memory machine. Figure 5 shows the results for this configuration. The read operation sustained a maximum aggregate bandwidth of approximately 10 GB/sec. for file channel with view buffer approach. Exactly same trend and result was observed when the shared file was placed on NFS storage. Bulk random access file and file channel in mapped mode performed comparable for both the configurations, while file channel in mapped mode started to perform better when the file was placed on NFS storage. This approach achieved a maximum bandwidth of 6 GB/seconds. The write operation for shared file on disk could only achieve a maximum bandwidth of 94 MB/sec. for all the three approaches.

Threads based I/O on NFS storage: This configuration is similar to the previous one except that we don't use the local disk of the machine, instead we place the shared file on a network attached storage mounted using NFS. Figure 6 shows results for this configuration. We noticed that write operation for file channel in mapped mode performed inefficiently when file was moved to NFS storage. The reasons for this can be locking (mapping) mechanisms used by Java, for memory-mapped regions of a file on NFS, accessed by multiple threads. Overall bandwidth increased significantly for file channel with view buffer and bulk random access file approaches, both achieved a maximum write bandwidth of approx. 250 MB/sec. up from 94 MB/sec. The increase in speed is due to the fact that SAN storage attached to the machine has higher bandwidth than local disk.

MPJ processes based I/O on NFS storage: This configuration uses MPJ Express processes which are executed on a distributed memory cluster. These are remote processes and each process performs read/write operations on a shared file. The processes seek to the appropriate file location and perform read/write operations on it. The file resides on the NFS storage. Figure 7 shows the results for this configuration. We note that the graphs shown in Figure 5 and 6 are for threads where we don't lock the file before reading or writing whereas graphs in Figure 7 are for processes where we implement locking using the `FileLock` class. Hence, the graphs shall not be compared to each other as they are somewhat unrelated in terms of configuration. Write performance improved as we increased the number of processes from 2 to 8 which shows that speedups are possible with this approach. On the contrary, the performance decreased as we increased number of processes beyond 8 and the reason for this drop is the contention for resources and the under optimized cluster setup which makes available only a single path from the NFS server to the storage system. We plan to upgrade this configuration along with installation of the parallel file systems like PVFS2 and Lustre. Read performance had similar results except that maximum bandwidth achieved was approaching 500 MB per second for file channel in mapped mode with 4 processes. Overall read bandwidth was higher than write bandwidth but did not scale well with increasing number of processes.

The most stable performance across all configurations and tests was achieved by file channel with view buffers approach. Bulk random access file approach performed comparable. It is not available, however, as part of standard JDK and needs to be downloaded from a third party. File channel in mapped mode had mixed results and we plan to investigate this approach further in the extended version of this work.



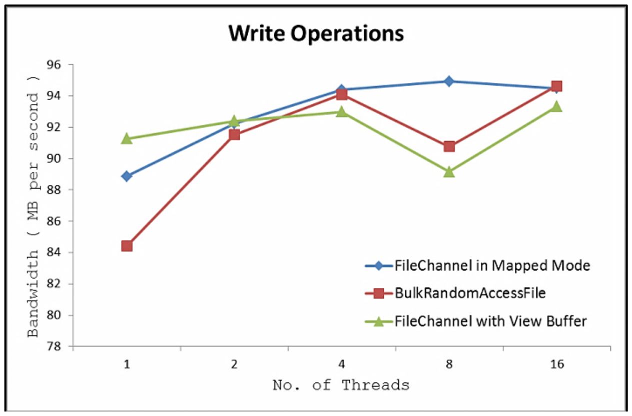
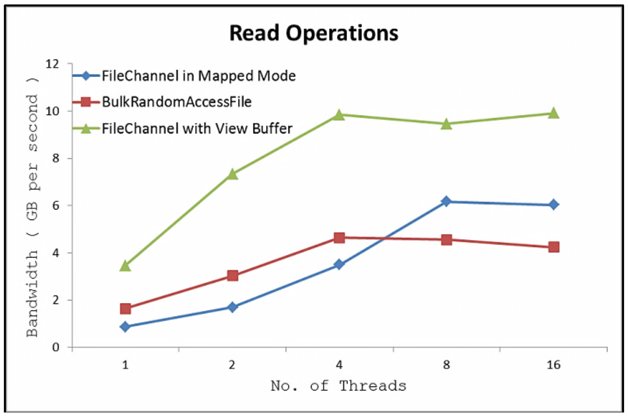


Figure 5. Performance of Tests using Java threads for parallel access to a shared file residing on local disk of the Shared Memory Machine



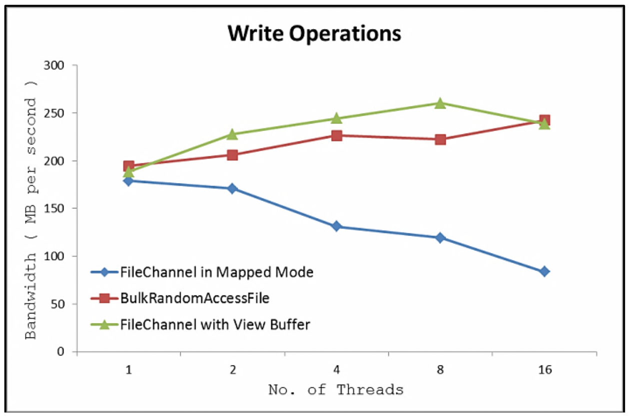
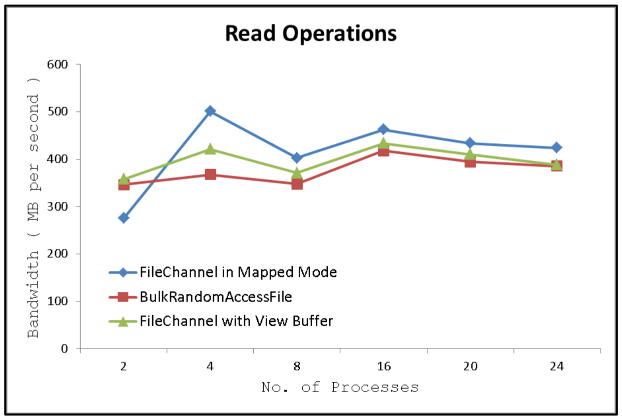


Figure 6. Performance of Tests using Java threads for parallel access to a shared file residing on NFS storage attached to the Shared Memory Machine



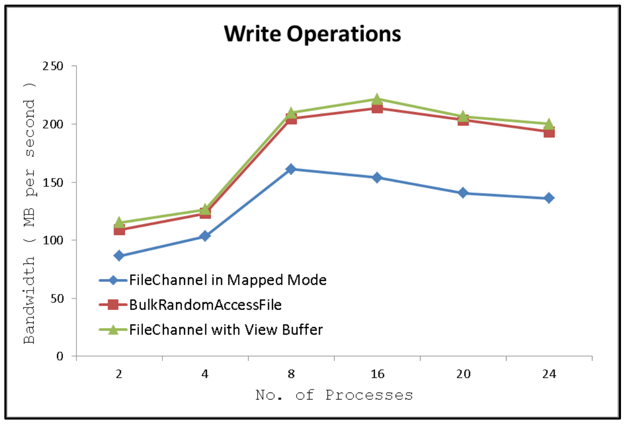


Figure 7. Performance of Tests using MPJ Express processes for parallel access to shared file residing on NFS storage of the Distributed Memory Machine

VI. Related Work

The most pertinent and related research was presented in [26] and [9] which evaluate the support for array based (bulk) I/O operations in Java. The work introduced bulk I/O operations for primitive datatypes other than bytes, which were not available in Java I/O API, in the form of JNI extensions originally presented in [25] and available from [16]. The C counterparts of this research include parallel I/O libraries like ROMIO [5], parallel HDF5 [6], Parallel I/O (PIO) [7], and parallel netCDF [8].

Table I provides a summary of the Java based parallel I/O libraries discussed in [10, 11, 12, 13, 14, 21 and 28]. The important features that are necessary for a high quality parallel I/O library include:

- Support for parallel I/O
- Support for inter-connects like Infiniband and Myrinet
- MPI-IO compliance
- Availability of the software and performance studies

TABLE I. EVALUATION OF JAVA PARALLEL I/O LIBRARIES

Java Parallel I/O Software	**Available for Download**	**Supports Parallel I/O**	**MPI-IO Compliant**	**Used in 3rd Party Software**	**Infiniband Support**	**Myrinet Support**	**Performance Studies Available**
JavaSeis	✓	✓	×	×	×	×	×
Java NetCDF	✓	×	-	✓	×	×	×
jExpand	×	✓	×	×	×	×	×
Parallel Java (PJ)	✓	✓	×	×	×	×	×
AgentTeamWork MPI-IO like Java Library	×	✓	×	×	×	×	×

VII. CONCLUSIONS AND FUTURE WORK

In this paper we addressed an important issue of parallel file I/O in Java. Java has two I/O APIs; a legacy Java I/O API which was previously benchmarked, and Java NIO API which has not been evaluated in the HPC context and is benchmarked in this paper. We observed that original Java I/O API provides poor file I/O performance, its extensions provide significant performance gains, and our proposed Java NIO approaches performed even better with increasing number of processes. In order to compete as a mainstream HPC language, Java based HPC libraries need to be equipped with efficient parallel I/O support. This can only be achieved if a standard MPI-IO like parallel file I/O API for Java be developed. Based on our performance evaluation, we can suggest that the design and implementation of a Java parallel I/O API shall be based on the Java NIO API as it natively provides the most efficient parallel file I/O methods. We plan to extend this work by evaluating Java NIO on popular parallel file systems like PVFS2 and Lustre, as well as developing and implementing an MPI-IO like Java API.

ACKNOWLEDGMENT

This work was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency)" (NIPA-2011- (C1090-1121-0003) and a grant from the NIPA in 2012. (Global IT Talents Program).

REFERENCES

[1] The Java Grande Forum, http://www.javagrande.org

[2] Carpenter, Bryan; Fox, Geoffrey; Ko, Sung-Hoon; and Lim, Sang, "mpiJava 1.2: API Specification" (1999). Northeast Parallel Architecture Center. Paper 66. http://surface.syr.edu/npac/66

[3] MPJ Express Project, http://www.mpj-express.org/

[4] William Gropp, Ewing Lusk, and Rajeev Thakur. Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.

[5] ROMIO: A High-Performance, Portable MPI-IO Implementation, http://www.mcs.anl.gov/research/projects/romio/

[6] Parallel HDF5 Project, http://www.hdfgroup.org/HDF5/PHDF5/

[7] Parallel I/O (PIO) Library, http://web.ncar.teragrid.org/~dennis/pio_doc/html/

[8] Parallel-NetCDF, http://trac.mcs.anl.gov/projects/parallel-netcdf

[9] Dan Bonachea, Phillip Dickens, and Rajeev Thakur, "High-performance file I/O in Java: Existing approaches and bulk I/O extensions," Concurrency and Computation: Practice and Experience, vol. 13, Aug. 2001, pp. 713–736.

[10] Omonbek Salaev, Parallel Datastore System for Parallel Java, A Capstone Project Final Report, January 2010, http://www.cs.rit.edu/~ark/students/obs8529/report.pdf

[11] Joshua Phillips, Munehiro Fukuda, Jumpei Miyauchi, "A Java Implementation of MPI-I/O-Oriented Random Access File Class in AgentTeamwork Grid Computing Middleware," Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim 07), Aug. 2007, pp.149-152.

[12] Jose M. Perez, L. M. Sanchez, Felix Garcia, Alejandro Calderon, Jesus Carreter, "High performance Java input/output for heterogeneous distributed computing," Proc. 10th IEEE Symp. on Computers and Communications (ISCC 2005), June 2005, pp. 969-974.

[13] The Java Net-CDF library, http://www.unidata.ucar.edu/software/netcdf-java

[14] JavaSeis Project, http://sourceforge.net/apps/mediawiki/javaseis/index.php?title=Main_Page

[15] Java™ Platform, Standard Edition 7 API Specification, http://docs.oracle.com/javase/7/docs/api/

[16] Bulk I/O Extensions to Java, http://www.eecs.berkeley.edu/~bonachea/java/index.html

[17] Titanium Project, http://titanium.cs.berkeley.edu/

[18] CPU-Z tool, http://www.cpuid.com/softwares/cpu-z.html

[19] Rajeev Thakur, Ewing Lusk, and William Gropp, "I/O in Parallel Applications: The Weakest Link," The Int'l Journal of High Performance Computing Applications, vol. 12(4), Winter 1998, pp. 389-395.

[20] Guillermo L. Taboada, Juan Touriño, Ramon Doallo, "F-MPJ: scalable Java message-passing communications on parallel systems," Journal of Supercomputing, vol. 60(1), 2012, pp. 117-140.

[21] Tsujita, Y, "MPI-I/O operations to a remote computer using Java," Proc. 11th International Conference on Parallel and Distributed Systems, July 2005. pp. 694-698, doi: 10.1109/ICPADS.2005.20

[22] Markus Bornemann, Rob V. van Nieuwpoort, and Thilo Kielmann, "MPJ/Ibis: a flexible and efficient message passing platform for Java," Proc. of 12th European PVM/MPI Users' Group Meeting, Sept. 2005, pp. 217-224.

[23] MPICH2 Project, http://www.mcs.anl.gov/research/projects/mpich2/

[24] Edgar Gabriel et al, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," Proc. Euro PVM/MPI 2004, Sept. 2004, pp. 97-104.

[25] Bonachea, Dan. "Bulk File I/O Extensions to Java," Proc. of the ACM Java Grande Conference, June 2000, pp. 16-25.

[26] Phillip Dickens and Rajeev Thakur, "An Evaluation of Java's I/O Capabilities for High-Performance Computing," Proc. of the ACM Java Grande Conference, June 2000, pp. 26-35.

[27] Alan Kaminsky, "Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java," Proc. 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), March 2007, pp. 1-8.

[28] Félix Garcia-Carballeira, Alejandro Calderon, Jesus Carretero, Javier Fernandez, and Jose M. Perez, "The Design of the Expand Parallel File System," International Journal of High Performance Computing Applications, vol. 17(1), Feb. 2003, pp. 21-37.