# IOR Documentation

*Release 0*

**IOR**

**Dec 24, 2020**

# Contents

Welcome to the IOR documentation.

**I**nterleaved **or** **R**andom is a parallel IO benchmark. IOR can be used for testing performance of parallel file systems using various interfaces and access patterns. IOR uses MPI for process synchronization. This documentation provides information for versions 3 and higher, for other versions check *Compatibility*

This documentation consists of tow parts.

The first part is a user documentation were you find instructions on compilation, a beginners tutorial (*First Steps with IOR*) as well as information about all available *Options*.

The second part is the developer documentation. It currently only consists of a auto generated Doxygen and some notes about the contiguous integration with travis. As there are quite some people how needs to modify or extend IOR to there needs it would be great to have documentation on what and how to alter IOR without breaking other stuff. Currently there is neither a documentation on the overall concept of the code nor on implementation details. If you are getting your hands dirty in code anyways or have deeper understanding of IOR, you are more then welcome to comment the code directly, which will result in better Doxygen output or add your insight to this sphinx documentation.

CHAPTER 1

# Introduction

Welcome to the IOR documentation.

**I**nterleaved **o**r **R**andom is a parallel IO benchmark. IOR can be used for testing performance of parallel file systems using various interfaces and access patterns. IOR uses MPI for process synchronization. This documentation provides information for versions 3 and higher, for other versions check *Compatibility*

This documentation consists of tow parts.

The first part is a user documentation were you find instructions on compilation, a beginners tutorial (*First Steps with IOR*) as well as information about all available *Options*.

The second part is the developer documentation. It currently only consists of a auto generated Doxygen and some notes about the contiguous integration with travis. As there are quite some people how needs to modify or extend IOR to there needs it would be great to have documentation on what and how to alter IOR without breaking other stuff. Currently there is neither a documentation on the overall concept of the code nor on implementation details. If you are getting your hands dirty in code anyways or have deeper understanding of IOR, you are more then welcome to comment the code directly, which will result in better Doxygen output or add your insight to this sphinx documentation.

# CHAPTER 2

# Install

## 2.1 Building

0. If “configure” is missing from the top level directory, you probably retrieved this code directly from the repository. Run “./bootstrap”.

   If your versions of the autotools are not new enough to run this script, download and official tarball in which the configure script is already provided.

1. Run “./configure”

   See “./configure –help” for configuration options.

2. Run “make”

3. Optionally, run “make install”. The installation prefix can be changed as an option to the “configure” script.

CHAPTER 3

# First Steps with IOR

This is a short tutorial for the basic usage of IOR and some tips on how to use IOR to handel caching effects as these are very likely to affect your measurements.

## 3.1 Running IOR

There are two ways of running IOR:

1) **Command line with arguments – executable followed by command line** options.

    **::** $ ./IOR -w -r -o filename

    This performs a write and a read to the file 'filename'.

2) **Command line with scripts – any arguments on the command line will** establish the default for the test run, but a script may be used in conjunction with this for varying specific tests during an execution of the code. Only arguments before the script will be used!

    **::** $ ./IOR -W -f script

    This defaults all tests in 'script' to use write data checking.

In this tutorial the first one is used as it is much easier to toy around with an get to know IOR. The second option thought is much more useful to safe benchmark setups to rerun later or to test many different cases.

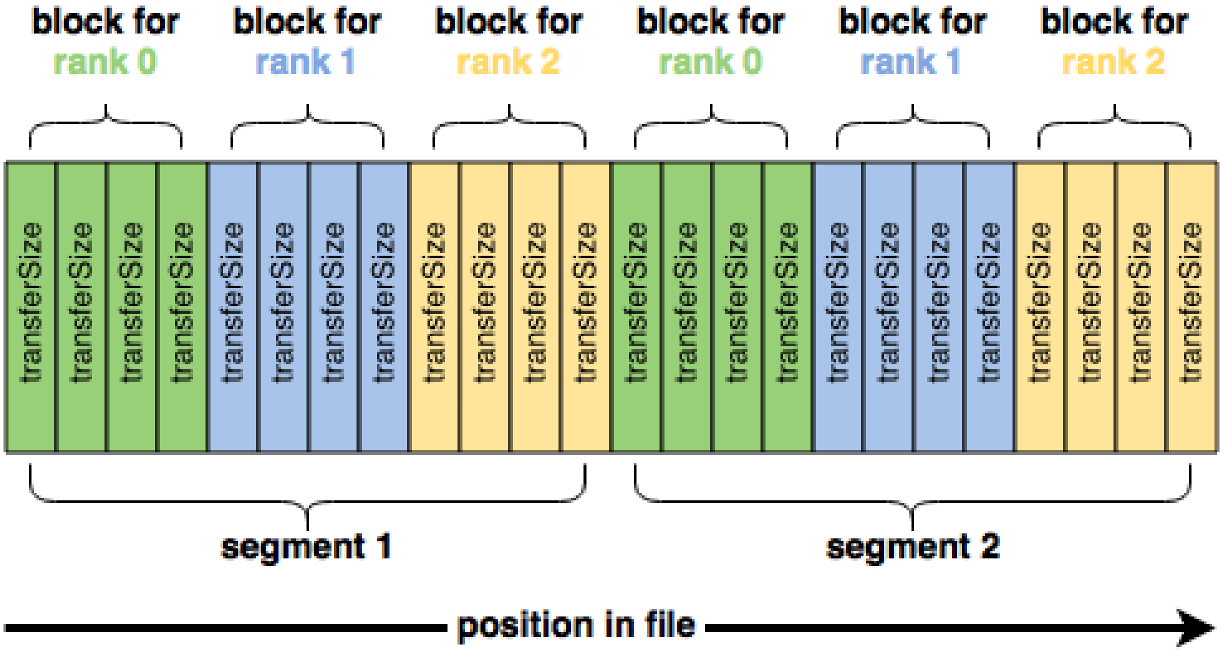
## 3.2 Getting Started with IOR

IOR writes data sequentially with the following parameters:

- blockSize (-b)
- transferSize (-t)
- segmentCount (-s)

- numTasks (-n)

which are best illustrated with a diagram:



These four parameters are all you need to get started with IOR. However, naively running IOR usually gives disappointing results. For example, if we run a four-node IOR test that writes a total of 16 GiB:

```
$ mpirun -n 64 ./ior -t 1m -b 16m -s 16
...
access bw(MiB/s) block(KiB) xfer(KiB) open(s)  wr/rd(s) close(s) total(s) iter
------ --------- ---------- --------- -------- -------- -------- -------- ----
write  427.36    16384      1024.00   0.107961 38.34    32.48    38.34    2
read   239.08    16384      1024.00   0.005789 68.53    65.53    68.53    2
remove -         -          -         -        -        -        0.534400 2
```

we can only get a couple hundred megabytes per second out of a Lustre file system that should be capable of a lot more.

Switching from writing to a single-shared file to one file per process using the -F (filePerProcess=1) option changes the performance dramatically:

```
$ mpirun -n 64 ./ior -t 1m -b 16m -s 16 -F
...
access bw(MiB/s) block(KiB) xfer(KiB) open(s)  wr/rd(s) close(s) total(s) iter
------ --------- ---------- --------- -------- -------- -------- -------- ----
write  33645     16384      1024.00   0.007693 0.486249 0.195494 0.486972 1
read   149473    16384      1024.00   0.004936 0.108627 0.016479 0.109612 1
remove -         -          -         -        -        -        6.08     1
```

This is in large part because letting each MPI process work on its own file cuts out any contention that would arise because of file locking.

However, the performance difference between our naive test and the file-per-process test is a bit extreme. In fact, the only way that 146 GB/sec read rate could be achievable on Lustre is if each of the four compute nodes had over 45 GB/sec of network bandwidth to Lustre–that is, a 400 Gbit link on every compute and storage node.

## 3.3 Effect of Page Cache on Benchmarking

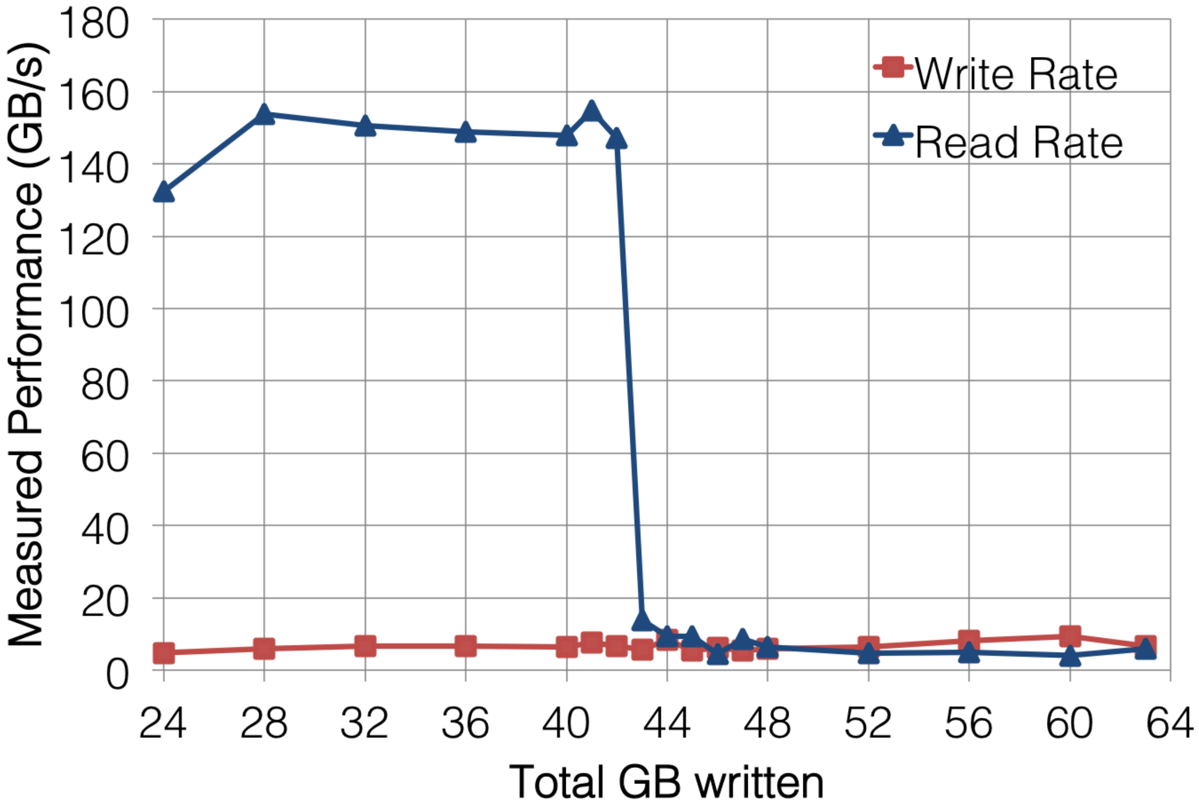
What's really happening is that the data being read by IOR isn't actually coming from Lustre; rather, files' contents are already cached, and IOR is able to read them directly out of each compute node's DRAM. The data wound up getting cached during the write phase of IOR as a result of Linux (and Lustre) using a write-back cache to buffer I/O, so that instead of IOR writing and reading data directly to Lustre, it's actually mostly talking to the memory on each compute node.

To be more specific, although each IOR process thinks it is writing to a file on Lustre and then reading back the contents of that file from Lustre, it is actually

1) writing data to a copy of the file that is cached in memory. If there is no copy of the file cached in memory before this write, the parts being modified are loaded into memory first.
2) those parts of the file in memory (called "pages") that are now different from what's on Lustre are marked as being "dirty"
3) the write() call completes and IOR continues on, even though the written data still hasn't been committed to Lustre
4) independent of IOR, the OS kernel continually scans the file cache for files who have been updated in memory but not on Lustre ("dirt pages"), and then commits the cached modifications to Lustre
5) dirty pages are declared non-dirty since they are now in sync with what's on disk, but they remain in memory

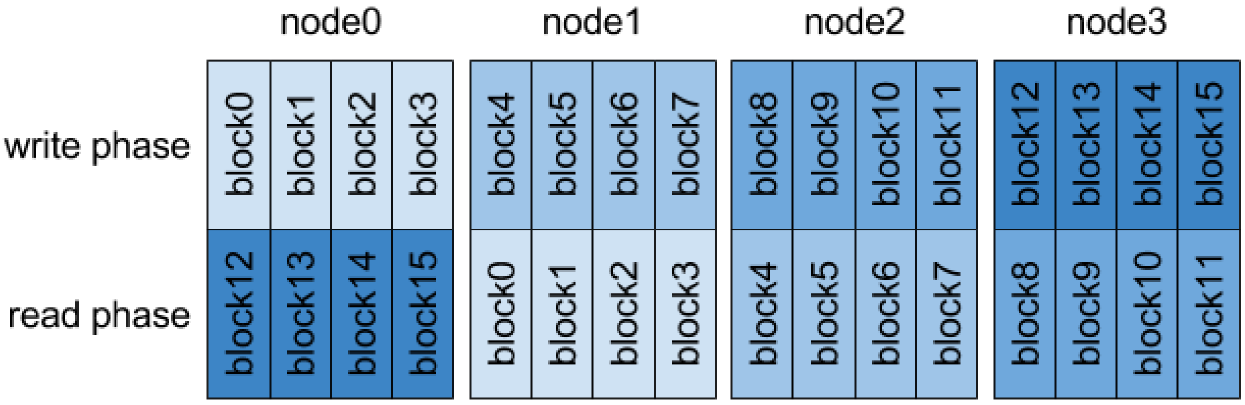
Then when the read phase of IOR follows the write phase, IOR is able to just retrieve the file's contents from memory instead of having to communicate with Lustre over the network.

There are a couple of ways to measure the read performance of the underlying Lustre file system. The most crude way is to simply write more data than will fit into the total page cache so that by the time the write phase has completed, the beginning of the file has already been evicted from cache. For example, increasing the number of segments (-s) to write more data reveals the point at which the nodes' page cache on my test system runs over very clearly:



However, this can make running IOR on systems with a lot of on-node memory take forever.

A better option would be to get the MPI processes on each node to only read data that they didn't write. For example, on a four-process-per-node test, shifting the mapping of MPI processes to blocks by four makes each node N read the data written by node N-1.



| | node0 | node1 | node2 | node3 |
|---|---|---|---|---|
| write phase | block0<br>block1<br>block2<br>block3 | block4<br>block5<br>block6<br>block7 | block8<br>block9<br>block10<br>block11 | block12<br>block13<br>block14<br>block15 |
| read phase | block12<br>block13<br>block14<br>block15 | block0<br>block1<br>block2<br>block3 | block4<br>block5<br>block6<br>block7 | block8<br>block9<br>block10<br>block11 |

Since page cache is not shared between compute nodes, shifting tasks this way ensures that each MPI process is reading data it did not write.

IOR provides the -C option (reorderTasks) to do this, and it forces each MPI process to read the data written by its neighboring node. Running IOR with this option gives much more credible read performance:

```
$ mpirun -n 64 ./ior -t 1m -b 16m -s 16 -F -C
...
```

(continues on next page)

(continued from previous page)

```
access bw(MiB/s) block(KiB) xfer(KiB) open(s)  wr/rd(s) close(s) total(s) iter
------ --------- ---------- --------- -------- -------- -------- -------- ----
write  41326     16384      1024.00   0.005756 0.395859 0.095360 0.396453 0
read   3310.00   16384      1024.00   0.011786 4.95     4.20     4.95     1
remove -         -          -         -        -        -        0.237291 1
```

But now it should seem obvious that the write performance is also ridiculously high. And again, this is due to the page cache, which signals to IOR that writes are complete when they have been committed to memory rather than the underlying Lustre file system.

To work around the effects of the page cache on write performance, we can issue an fsync() call immediately after all of the write()s return to force the dirty pages we just wrote to flush out to Lustre. Including the time it takes for fsync() to finish gives us a measure of how long it takes for our data to write to the page cache and for the page cache to write back to Lustre.

IOR provides another convenient option, -e (fsync), to do just this. And, once again, using this option changes our performance measurement quite a bit:

```
$ mpirun -n 64 ./ior -t 1m -b 16m -s 16 -F -C -e
...
access bw(MiB/s) block(KiB) xfer(KiB) open(s)  wr/rd(s) close(s) total(s) iter
------ --------- ---------- --------- -------- -------- -------- -------- ----
write  2937.89   16384      1024.00   0.011841 5.56     4.93     5.58     0
read   2712.55   16384      1024.00   0.005214 6.04     5.08     6.04     3
remove -         -          -         -        -        -        0.037706 0
```

and we finally have a believable bandwidth measurement for our file system.

Defeating Page Cache Since IOR is specifically designed to benchmark I/O, it provides these options that make it as easy as possible to ensure that you are actually measuring the performance of your file system and not your compute nodes' memory. That being said, the I/O patterns it generates are designed to demonstrate peak performance, not reflect what a real application might be trying to do, and as a result, there are plenty of cases where measuring I/O performance with IOR is not always the best choice. There are several ways in which we can get clever and defeat page cache in a more general sense to get meaningful performance numbers.

When measuring write performance, bypassing page cache is actually quite simple; opening a file with the O_DIRECT flag going directly to disk. In addition, the fsync() call can be inserted into applications, as is done with IOR's -e option.

Measuring read performance is a lot trickier. If you are fortunate enough to have root access on a test system, you can force the Linux kernel to empty out its page cache by doing

**::** # echo 1 > /proc/sys/vm/drop_caches

and in fact, this is often good practice before running any benchmark (e.g., Linpack) because it ensures that you aren't losing performance to the kernel trying to evict pages as your benchmark application starts allocating memory for its own use.

Unfortunately, many of us do not have root on our systems, so we have to get even more clever. As it turns out, there is a way to pass a hint to the kernel that a file is no longer needed in page cache:

```
#define _XOPEN_SOURCE 600
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char *argv[]) {
    int fd;
    fd = open(argv[1], O_RDONLY);
    fdatasync(fd);
    posix_fadvise(fd, 0,0,POSIX_FADV_DONTNEED);
```

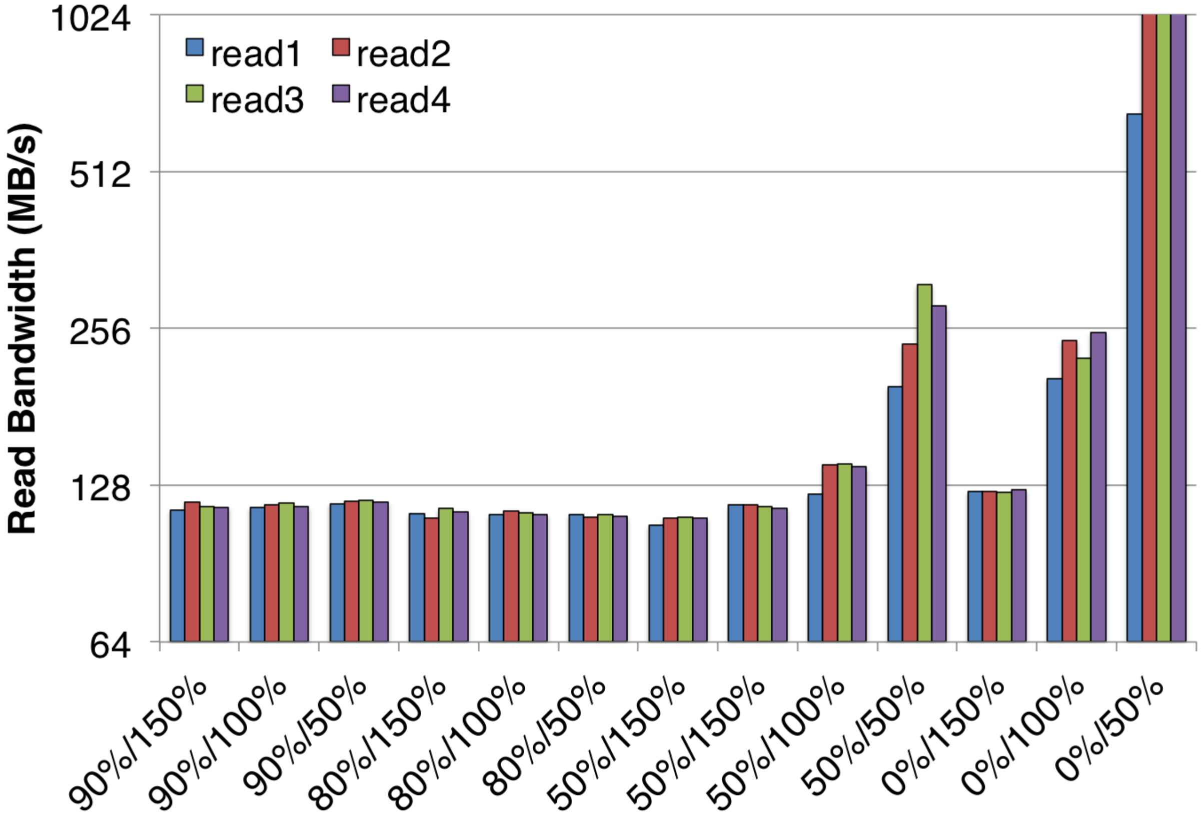
(continues on next page)

(continued from previous page)

```
    close(fd);
    return 0;
}
```

The effect of passing POSIX_FADV_DONTNEED using posix_fadvise() is usually that all pages belonging to that file are evicted from page cache in Linux. However, this is just a hint–not a guarantee–and the kernel evicts these pages asynchronously, so it may take a second or two for pages to actually leave page cache. Fortunately, Linux also provides a way to probe pages in a file to see if they are resident in memory.

Finally, it's often easiest to just limit the amount of memory available for page cache. Because application memory always takes precedence over cache memory, simply allocating most of the memory on a node will force most of the cached pages to be evicted. Newer versions of IOR provide the memoryPerNode option that do just that, and the effects are what one would expect:



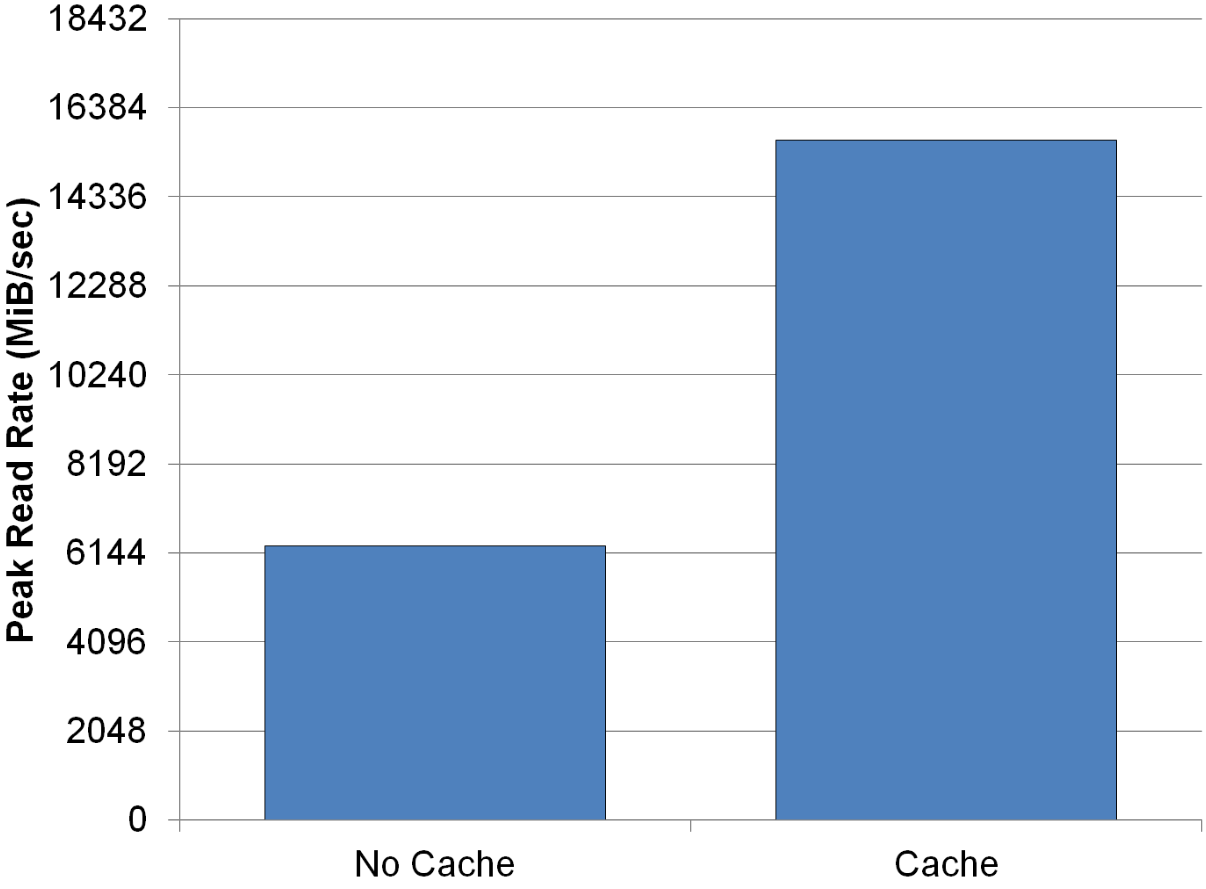
The above diagram shows the measured bandwidth from a single node with 128 GiB of total DRAM. The first percent on each x-label is the amount of this 128 GiB that was reserved by the benchmark as application memory, and the second percent is the total write volume. For example, the “50%/150%” data points correspond to 50% of the node memory (64 GiB) being allocated for the application, and a total of 192 GiB of data being read.

This benchmark was run on a single spinning disk which is not capable of more than 130 MB/sec, so the conditions that showed performance higher than this were benefiting from some pages being served from cache. And this makes perfect sense given that the anomalously high performance measurements were obtained when there was plenty of memory to cache relative to the amount of data being read.

## 3.4 Corollary

Measuring I/O performance is a bit trickier than CPU performance in large part due to the effects of page caching. That being said, page cache exists for a reason, and there are many cases where an application's I/O performance really is best represented by a benchmark that heavily utilizes cache.

For example, the BLAST bioinformatics application re-reads all of its input data twice; the first time initializes data structures, and the second time fills them up. Because the first read caches each page and allows the second read to come out of cache rather than the file system, running this I/O pattern with page cache disabled causes it to be about 2x slower:



Thus, letting the page cache do its thing is often the most realistic way to benchmark with realistic application I/O patterns. Once you know how page cache might be affecting your measurements, you stand a good chance of being able to reason about what the most meaningful performance metrics are.

CHAPTER 4

# Options

IOR provides many options, in fact there are now more than there are one letter flags in the alphabet. For this and to run IOR by a config script, there are some options which are only available via directives. When both script and command line options are in use, command line options set in front of -f are the defaults which may be overridden by the script. Directives can also be set from the command line via "-O" option. In combination with a script they behave like the normal command line options. But directives and normal parameters override each other, so the last one executed.

## 4.1 Command line options

**These options are to be used on the command line. E.g., 'IOR -a POSIX -b 4K'.**

| | |
|---|---|
| **-a S** | api – API for I/O [POSIX\|MPIIO\|HDF5\|HDFS\|S3\|S3_EMC\|NCMPI\|RADOS] |
| **-A N** | refNum – user reference number to include in long summary |
| **-b N** | blockSize – contiguous bytes to write per task (e.g.: 8, 4k, 2m, 1g) |
| **-B** | useO_DIRECT – uses O_DIRECT for POSIX, bypassing I/O buffers |
| **-c** | collective – collective I/O |
| **-C** | reorderTasksConstant – changes task ordering to n+1 ordering for readback |
| **-d N** | interTestDelay – delay between reps in seconds |
| **-D N** | deadlineForStonewalling – seconds before stopping write or read phase |
| **-e** | fsync – perform fsync upon POSIX write close |
| **-E** | useExistingTestFile – do not remove test file before write access |
| **-f S** | scriptFile – test script name |
| **-F** | filePerProc – file-per-process |
| **-g** | intraTestBarriers – use barriers between open, write/read, and close |

**-G N** setTimeStampSignature – set value for time stamp signature

**-h** showHelp – displays options and help

**-H** showHints – show hints

**-i N** repetitions – number of repetitions of test

**-I** individualDataSets – datasets not shared by all procs [not working]

**-j N** outlierThreshold – warn on outlier N seconds from mean

**-J N** setAlignment – HDF5 alignment in bytes (e.g.: 8, 4k, 2m, 1g)

**-k** keepFile – don't remove the test file(s) on program exit

**-K** keepFileWithError – keep error-filled file(s) after data-checking

**-l** data packet type– type of packet that will be created [offset|incompressible|timestamp|o|i|t]

**-m** multiFile – use number of reps (-i) for multiple file count

**-M N** memoryPerNode – hog memory on the node (e.g.: 2g, 75%)

**-n** noFill – no fill in HDF5 file creation

**-N N** numTasks – number of tasks that should participate in the test

**-o S** testFile – full name for test

**-O S** string of IOR directives (e.g. -O checkRead=1,lustreStripeCount=32)

**-p** preallocate – preallocate file size

**-P** useSharedFilePointer – use shared file pointer [not working]

**-q** quitOnError – during file error-checking, abort on error

**-Q N** taskPerNodeOffset for read tests use with -C & -Z options (-C constant N, -Z at least N) [!HDF5]

**-r** readFile – read existing file

**-R** checkRead – check read after read

**-s N** segmentCount – number of segments

**-S** useStridedDatatype – put strided access into datatype [not working]

**-t N** transferSize – size of transfer in bytes (e.g.: 8, 4k, 2m, 1g)

**-T N** maxTimeDuration – max time in minutes to run tests

**-u** uniqueDir – use unique directory name for each file-per-process

**-U S** hintsFileName – full name for hints file

**-v** verbose – output information (repeating flag increases level)

**-V** useFileView – use MPI_File_set_view

**-w** writeFile – write file

**-W** checkWrite – check read after write

**-x** singleXferAttempt – do not retry transfer if incomplete

**-X N** reorderTasksRandomSeed – random seed for -Z option

**-Y** fsyncPerWrite – perform fsync after each POSIX write

**-z** randomOffset – access is to random, not sequential, offsets within a file

**-Z** reorderTasksRandom – changes task ordering to random ordering for readback

**NOTES: * S is a string, N is an integer number.**

- For transfer and block sizes, the case-insensitive K, M, and G suffices are recognized. I.e., ‘4k’ or ‘4K’ is accepted as 4096.

## 4.2 Directive Options

For each of the general settings, note the default is shown in brackets. IMPORTANT NOTE: For all true/false options below [1]=true, [0]=false IMPORTANT NOTE: Contrary to appearance, the script options below are NOT case sensitive

### 4.2.1 GENERAL:

- **refNum - user supplied reference number, included in** long summary [0]
- **api - must be set to one of POSIX, MPIIO, HDF5, HDFS, S3,** S3_EMC, or NCMPI, depending on test [POSIX]
- **testFile - name of the output file [testFile]**

  **NOTE: with filePerProc set, the tasks can round** robin across multiple file names ‘-o S@S@S’
- hintsFileName - name of the hints file []
- repetitions - number of times to run each test [1]
- **multiFile - creates multiple files for single-shared-file or** file-per-process modes; i.e. each iteration creates a new file [0=FALSE]
- **reorderTasksConstant - reorders tasks by a constant node offset for writing/reading neighbor’s** data from different nodes [0=FALSE]
- **taskPerNodeOffset - for read tests. Use with -C & -Z options. [1]** With reorderTasks, constant N. With reordertasksrandom, >= N
- reorderTasksRandom - reorders tasks to random ordering for readback [0=FALSE]
- **reorderTasksRandomSeed - random seed for reordertasksrandom option. [0]** >0, same seed for all iterations. <0, different seed for each iteration
- **quitOnError - upon error encountered on checkWrite or checkRead,** display current error and then stop execution; if not set, count errors and continue [0=FALSE]
- **numTasks - number of tasks that should participate in the test** [0] NOTE: 0 denotes all tasks
- **interTestDelay - this is the time in seconds to delay before** beginning a write or read in a series of tests [0] NOTE: it does not delay before a check write or check read
- **outlierThreshold - gives warning if any task is more than this number** of seconds from the mean of all participating tasks. If so, the task is identified, its time (start, elapsed create, elapsed transfer, elapsed close, or end) is reported, as is the mean and standard deviation for all tasks. The default for this is 0, which turns it off. If set to a positive value, for example 3, any task not within 3 seconds of the mean displays its times. [0]
- intraTestBarriers - use barrier between open, write/read, and close [0=FALSE]

- **uniqueDir - create and use unique directory for each** file-per-process [0=FALSE]
- **writeFile - writes file(s), first deleting any existing file [1=TRUE]**

  **NOTE: the defaults for writeFile and readFile are** set such that if there is not at least one of the following -w, -r, -W, or -R, it is assumed that -w and -r are expected and are consequently used – this is only true with the command line, and may be overridden in a script

- **readFile - reads existing file(s) (from current or previous** run) [1=TRUE] NOTE: see writeFile notes
- **filePerProc - accesses a single file for each processor; default** is a single file accessed by all processors [0=FALSE]
- **checkWrite - read data back and check for errors against known** pattern; can be used independently of writeFile [0=FALSE] NOTES: - data checking is not timed and does not

  affect other performance timings

  - all errors tallied and returned as program exit code, unless quitOnError set
- **checkRead - reread data and check for errors between reads; can** be used independently of readFile [0=FALSE] NOTE: see checkWrite notes
- keepFile - stops removal of test file(s) on program exit [0=FALSE]
- **keepFileWithError - ensures that with any error found in data-checking,** the error-filled file(s) will not be deleted [0=FALSE]
- useExistingTestFile - do not remove test file before write access [0=FALSE]
- **segmentCount - number of segments in file [1]**

  **NOTES: - a segment is a contiguous chunk of data**

  accessed by multiple clients each writing/ reading their own contiguous data; comprised of blocks accessed by multiple clients

  - with HDF5 this repeats the pattern of an entire shared dataset
- **blockSize - size (in bytes) of a contiguous chunk of data** accessed by a single client; it is comprised of one or more transfers [1048576]
- **transferSize - size (in bytes) of a single data buffer to be** transferred in a single I/O call [262144]
- **verbose - output information [0]**

  **NOTE: this can be set to levels 0-5 on the command** line; repeating the -v flag will increase verbosity level

- **setTimeStampSignature - set value for time stamp signature [0]**

  **NOTE: used to rerun tests with the exact data** pattern by setting data signature to contain positive integer value as timestamp to be written in data file; if set to 0, is disabled

- showHelp - display options and help [0=FALSE]
- **storeFileOffset - use file offset as stored signature when writing** file [0=FALSE] NOTE: this will affect performance measurements
- **memoryPerNode - Allocate memory on each node to simulate real** application memory usage. Accepts a percentage of node memory (e.g. “50%”) on machines that support sysconf(_SC_PHYS_PAGES) or a size. Allocation will be split between tasks that share the node.
- **memoryPerTask - Allocate secified amount of memory per task to** simulate real application memory usage.
- **maxTimeDuration - max time in minutes to run tests [0]**

**NOTES: * setting this to zero (0) unsets this option**

- this option allows the current read/write to complete without interruption

- **deadlineForStonewalling - seconds before stopping write or read phase [0]**

  **NOTES: - used for measuring the amount of data moved**

  in a fixed time. After the barrier, each task starts its own timer, begins moving data, and the stops moving data at a pre- arranged time. Instead of measuring the amount of time to move a fixed amount of data, this option measures the amount of data moved in a fixed amount of time. The objective is to prevent tasks slow to complete from skewing the performance.

  - setting this to zero (0) unsets this option
  - this option is incompatible w/data checking

- **randomOffset - access is to random, not sequential, offsets within a file [0=FALSE]**

  **NOTES: - this option is currently incompatible with:** -checkRead -storeFileOffset -MPIIO collective or useFileView -HDF5 or NCMPI

- **summaryAlways - Always print the long summary for each test.** Useful for long runs that may be interrupted, preventing the final long summary for ALL tests to be printed.

### 4.2.2 POSIX-ONLY

- useO_DIRECT - use O_DIRECT for POSIX, bypassing I/O buffers [0]
- **singleXferAttempt - will not continue to retry transfer entire buffer** until it is transferred [0=FALSE] NOTE: when performing a write() or read() in POSIX,

  there is no guarantee that the entire requested size of the buffer will be transferred; this flag keeps the retrying a single transfer until it completes or returns an error

- fsyncPerWrite - perform fsync after each POSIX write [0=FALSE]
- fsync - perform fsync after POSIX write close [0=FALSE]

### 4.2.3 MPIIO-ONLY

- preallocate - preallocate the entire file before writing [0=FALSE]
- **useFileView - use an MPI datatype for setting the file view option** to use individual file pointer [0=FALSE] NOTE: default IOR uses explicit file pointers
- **useSharedFilePointer - use a shared file pointer [0=FALSE] (not working)** NOTE: default IOR uses explicit file pointers
- **useStridedDatatype - create a datatype (max=2GB) for strided access; akin** to MULTIBLOCK_REGION_SIZE [0] (not working)

### 4.2.4 HDF5-ONLY

- **individualDataSets - within a single file each task will access its own** dataset [0=FALSE] (not working) NOTE: default IOR creates a dataset the size of

  numTasks * blockSize to be accessed by all tasks

- noFill - no pre-filling of data in HDF5 file creation [0=FALSE]

- setAlignment - HDF5 alignment in bytes (e.g.: 8, 4k, 2m, 1g) [1]
- **collectiveMetadata - enable HDF5 collective metadata (available since** HDF5-1.10.0)

### 4.2.5 MPIIO-, HDF5-, AND NCMPI-ONLY

- collective - uses collective operations for access [0=FALSE]
- **showHints - show hint/value pairs attached to open file [0=FALSE]** NOTE: not available in NCMPI

### 4.2.6 LUSTRE-SPECIFIC

- lustreStripeCount - set the lustre stripe count for the test file(s) [0]
- lustreStripeSize - set the lustre stripe size for the test file(s) [0]
- lustreStartOST - set the starting OST for the test file(s) [-1]
- lustreIgnoreLocks - disable lustre range locking [0]

### 4.2.7 GPFS-SPECIFIC

- gpfsHintAccess - use gpfs_fcntl hints to pre-declare accesses
- **gpfsReleaseToken - immediately after opening or creating file, release** all locks. Might help mitigate lock-revocation traffic when many proceses write/read to same file.

## 4.3 Verbosity levels

The verbosity of output for IOR can be set with -v. Increasing the number of -v instances on a command line sets the verbosity higher.

Here is an overview of the information shown for different verbosity levels:

0) default; only bare essentials shown
1) max clock deviation, participating tasks, free space, access pattern, commence/verify access notification w/time
2) rank/hostname, machine name, timer used, individual repetition performance results, timestamp used for data signature
3) full test details, transfer block/offset compared, individual data checking errors, environment variables, task writing/reading file name, all test operation times
4) task id and offset for each transfer
5) each 8-byte data signature comparison (WARNING: more data to STDOUT than stored in file, use carefully)

## 4.4 Incompressible notes

Please note that incompressibility is a factor of how large a block compression algorithm uses. The incompressible buffer is filled only once before write times, so if the compression algorithm takes in blocks larger than the transfer size, there will be compression. Below are some baselines that I established for zip, gzip, and bzip.

1) zip: For zipped files, a transfer size of 1k is sufficient.

2) gzip: For gzipped files, a transfer size of 1k is sufficient.
3) bzip2: For bziped files a transfer size of 1k is insufficient (~50% compressed). To avoid compression a transfer size of greater than the bzip block size is required (default = 900KB). I suggest a transfer size of greather than 1MB to avoid bzip2 compression.

Be aware of the block size your compression algorithm will look at, and adjust the transfer size accordingly.

# CHAPTER 5

# Scripting

IOR can use a script with the command line. Any options on the command line set before the script will be considered the default settings for running the script. (I.e.,'$ ./IOR -W -f script' will have all tests in the script run with the -W option as default.) The script itself can override these settings and may be set to run run many different tests of IOR under a single execution. The command line is:

```
./IOR -f script
```

In IOR/scripts, there are scripts of test cases for simulating I/O behavior of various application codes. Details are included in each script as necessary.

**Syntax:**

- IOR START / IOR END: marks the beginning and end of the script
- RUN: Delimiter for next Test
- All previous set parameter stay set for the next test. They are not reset to the default! For default the musst be rest manually.
- White space is ignored in script, as are comments starting with '#'.
- Not all test parameters need be set.

An example of a script:

```
IOR START
  api=[POSIX|MPIIO|HDF5|HDFS|S3|S3_EMC|NCMPI|RADOS]
  testFile=testFile
  hintsFileName=hintsFile
  repetitions=8
  multiFile=0
  interTestDelay=5
  readFile=1
  writeFile=1
  filePerProc=0
  checkWrite=0
```

(continues on next page)

(continued from previous page)

```
  checkRead=0
  keepFile=1
  quitOnError=0
  segmentCount=1
  blockSize=32k
  outlierThreshold=0
  setAlignment=1
  transferSize=32
  singleXferAttempt=0
  individualDataSets=0
  verbose=0
  numTasks=32
  collective=1
  preallocate=0
  useFileView=0
  keepFileWithError=0
  setTimeStampSignature=0
  useSharedFilePointer=0
  useStridedDatatype=0
  uniqueDir=0
  fsync=0
  storeFileOffset=0
  maxTimeDuration=60
  deadlineForStonewalling=0
  useExistingTestFile=0
  useO_DIRECT=0
  showHints=0
  showHelp=0
RUN
  # additional tests are optional
  <snip>
RUN
  <snip>
RUN
IOR STOP
```

# CHAPTER 6

# Compatibility

IOR has a long history. Here are some hints about compatibility with older versions.

1) IOR version 1 (c. 1996-2002) and IOR version 2 (c. 2003-present) are incompatible. Input decks from one will not work on the other. As version 1 is not included in this release, this shouldn't be case for concern. All subsequent compatibility issues are for IOR version 2.

2) IOR versions prior to release 2.8 provided data size and rates in powers of two. E.g., 1 MB/sec referred to 1,048,576 bytes per second. With the IOR release 2.8 and later versions, MB is now defined as 1,000,000 bytes and MiB is 1,048,576 bytes.

3) In IOR versions 2.5.3 to 2.8.7, IOR could be run without any command line options. This assumed that if both write and read options (-w -r) were omitted, the run with them both set as default. Later, it became clear that in certain cases (data checking, e.g.) this caused difficulties. In IOR versions 2.8.8 and later, if not one of the -w -r -W or -R options is set, then -w and -r are set implicitly.

4) IOR version 3 (Jan 2012-present) has changed the output of IOR somewhat, and the "testNum" option was renamed "refNum".

# CHAPTER 7

## Frequently Asked Questions

HOW DO I PERFORM MULTIPLE DATA CHECKS ON AN EXISTING FILE?

Use this command line: IOR -k -E -W -i 5 -o file

-k keeps the file after the access rather than deleting it -E uses the existing file rather than truncating it first -W performs the writecheck -i number of iterations of checking -o filename

On versions of IOR prior to 2.8.8, you need the -r flag also, otherwise you'll first overwrite the existing file. (In earlier versions, omitting -w and -r implied using both. This semantic has been subsequently altered to be omitting -w, -r, -W, and -R implied using both -w and -r.)

If you're running new tests to create a file and want repeat data checking on this file multiple times, there is an undocumented option for this. It's -O multiReRead=1, and you'd need to have an IOR version compiled with the USE_UNDOC_OPT=1 (in iordef.h). The command line would look like this:

IOR -k -E -w -W -i 5 -o file -O multiReRead=1

For the first iteration, the file would be written (w/o data checking). Then for any additional iterations (four, in this example) the file would be reread for whatever data checking option is used.

HOW DOES IOR CALCULATE PERFORMANCE?

IOR performs get a time stamp START, then has all participating tasks open a shared or independent file, transfer data, close the file(s), and then get a STOP time. A stat() or MPI_File_get_size() is performed on the file(s) and compared against the aggregate amount of data transferred. If this value does not match, a warning is issued and the amount of data transferred as calculated from write(), e.g., return codes is used. The calculated bandwidth is the amount of data transferred divided by the elapsed STOP-minus-START time.

IOR also gets time stamps to report the open, transfer, and close times. Each of these times is based on the earliest start time for any task and the latest stop time for any task. Without using barriers between these operations (-g), the sum of the open, transfer, and close times may not equal the elapsed time from the first open to the last close.

HOW DO I ACCESS MULTIPLE FILE SYSTEMS IN IOR?

It is possible when using the filePerProc option to have tasks round-robin across multiple file names. Rather than use a single file name '-o file', additional names '-o file1@file2@file3' may be used. In this case, a file per process would have three different file names (which may be full path names) to access. The '@' delimiter is arbitrary, and may be set in the FILENAME_DELIMITER definition in iordef.h.

Note that this option of multiple filenames only works with the filePerProc -F option. This will not work for shared files.

HOW DO I BALANCE LOAD ACROSS MULTIPLE FILE SYSTEMS?

As for the balancing of files per file system where different file systems offer different performance, additional instances of the same destination path can generally achieve good balance.

For example, with FS1 getting 50% better performance than FS2, set the '-o' flag such that there are additional instances of the FS1 directory. In this case, '-o FS1/file@FS1/file@FS1/file@FS2/file@FS2/file' should adjust for the performance difference and balance accordingly.

HOW DO I USE STONEWALLING?

To use stonewalling (-D), it's generally best to separate write testing from read testing. Start with writing a file with '-D 0' (stonewalling disabled) to determine how long the file takes to be written. If it takes 10 seconds for the data transfer, run again with a shorter duration, '-D 7' e.g., to stop before the file would be completed without stonewalling. For reading, it's best to create a full file (not an incompletely written file from a stonewalling run) and then run with stonewalling set on this preexisting file. If a write and read test are performed in the same run with stonewalling, it's likely that the read will encounter an error upon hitting the EOF. Separating the runs can correct for this. E.g.,

IOR -w -k -o file -D 10 # write and keep file, stonewall after 10 seconds IOR -r -E -o file -D 7 # read existing file, stonewall after 7 seconds

Also, when running multiple iterations of a read-only stonewall test, it may be necessary to set the -D value high enough so that each iteration is not reading from cache. Otherwise, in some cases, the first iteration may show 100 MB/s, the next 200 MB/s, the third 300 MB/s. Each of these tests is actually reading the same amount from disk in the allotted time, but they are also reading the cached data from the previous test each time to get the increased performance. Setting -D high enough so that the cache is overfilled will prevent this.

HOW DO I BYPASS CACHING WHEN READING BACK A FILE I'VE JUST WRITTEN?

One issue with testing file systems is handling cached data. When a file is written, that data may be stored locally on the node writing the file. When the same node attempts to read the data back from the file system either for performance or data integrity checking, it may be reading from its own cache rather from the file system.

The reorderTasksConstant '-C' option attempts to address this by having a different node read back data than wrote it. For example, node N writes the data to file, node N+1 reads back the data for read performance, node N+2 reads back the data for write data checking, and node N+3 reads the data for read data checking, comparing this with the reread data from node N+4. The objective is to make sure on file access that the data is not being read from cached data.

> Node 0: writes data Node 1: reads data Node 2: reads written data for write checking Node 3: reads written data for read checking Node 4: reads written data for read checking, comparing with Node 3

The algorithm for skipping from N to N+1, e.g., expects consecutive task numbers on nodes (block assignment), not those assigned round robin (cyclic assignment). For example, a test running 6 tasks on 3 nodes would expect tasks 0,1 on node 0; tasks 2,3 on node 1; and tasks 4,5 on node 2. Were the assignment for tasks-to-node in round robin fashion, there would be tasks 0,3 on node 0; tasks 1,4 on node 1; and tasks 2,5 on node 2. In this case, there would be no expectation that a task would not be reading from data cached on a node.

HOW DO I USE HINTS?

It is possible to pass hints to the I/O library or file system layers following this form:

```
'setenv IOR_HINT__<layer>__<hint> <value>'
```

**For example::** 'setenv IOR_HINT__MPI__IBM_largeblock_io true' 'setenv IOR_HINT__GPFS__important_hint true'

**or, in a file in the form::** 'IOR_HINT__<layer>__<hint>=<value>'

**Note that hints to MPI from the HDF5 or NCMPI layers are of the form::** 'setenv IOR_HINT__MPI__<hint> <value>'

HOW DO I EXPLICITY SET THE FILE DATA SIGNATURE?

The data signature for a transfer contains the MPI task number, transfer- buffer offset, and also timestamp for the start of iteration. As IOR works with 8-byte long long ints, the even-numbered long longs written contain a 32-bit MPI task number and a 32-bit timestamp. The odd-numbered long longs contain a 64-bit transferbuffer offset (or file offset if the '-l' storeFileOffset option is used). To set the timestamp value, use '-G' or setTimeStampSignature.

HOW DO I EASILY CHECK OR CHANGE A BYTE IN AN OUTPUT DATA FILE?

There is a simple utility IOR/src/C/cbif/cbif.c that may be built. This is a stand-alone, serial application called cbif (Change Byte In File). The utility allows a file offset to be checked, returning the data at that location in IOR's data check format. It also allows a byte at that location to be changed.

HOW DO I CORRECT FOR CLOCK SKEW BETWEEN NODES IN A CLUSTER?

To correct for clock skew between nodes, IOR compares times between nodes, then broadcasts the root node's timestamp so all nodes can adjust by the difference. To see an egregious outlier, use the '-j' option. Be sure to set this value high enough to only show a node outside a certain time from the mean.

# CHAPTER 8

# Doxygen

Click here for doxygen.

This documentation utilities doxygen for parsing the c code. Therefore a doxygen instances is created in the background anyway. This might be helpful as doxygen produces nice call graphs.

CHAPTER 9

# Continues Integration

Continues Integration is used for basic sanity checking. Travis-CI provides free CI for open source github projects and is configured via a .travis.yml.

For now this is set up to compile IOR on a ubuntu 14.04 machine with gcc 4.8, openmpi and hdf5 for the backends. This is a pretty basic check and should be advance over time. Nevertheless this should detect major errors early as they are shown in pull requests.

CHAPTER 10

# Changes in IOR