OAK RIDGE
NATIONAL LABORATORY
MANAGED BY UT-BATTELLE
FOR THE DEPARTMENT OF ENERGY

ORNL/SPR-2017/496

OpenSHMEM Implementation of IOR Benchmark

October 2016

Prepared by
Eduardo D'Azevedo, Sarah Powers, Neena Imam

ORNL/SPR-2017/496

Computational Research & Development Programs, Computer Science and Mathematics Division

OpenSHMEM Implementation of IOR Benchmark

Eduardo D'Azevedo, Sarah Powers, Neena Imam

Prepared by
OAK RIDGE NATIONAL LABORATORY
P.O. Box 2008
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

Abstract

The benchmarking effort within the Computational Research & Development Programs at the Oak Ridge National Laboratory (ORNL) seeks to design and enable High Performance Computing (HPC) benchmarks and test suites. The work described in this report is a part of the effort focusing on the comparison and analysis of OpenSHMEM implementations using the Interleave Or Random (IOR) software for benchmarking parallel file system using POSIX, MPIIO, or HDF5 interfaces. We describe the effort to emulate the MPIIO parallel collective capabilities in the IOR benchmark using OpenSHMEM communication. One development effort was in emulating the MPI derived datatype used in the read/write operations and in setting the file view. Another effort was in implementing an internal cache in OpenSHMEM distributed shared memory to facilitate global collective I/O operations. Experiments comparing collective I/O in MPIIO implementations with the OpenSHMEM implementations were performed on the SGI Turing Cluster and the Cray XK7 Titan supercomputer at the Oak Ridge Leadership Computing Facility (OLCF). The preliminary results suggest that on the Cray XK7 Titan, the MPIIO implementations obtained higher write performance and the OpenSHMEM version obtained slightly higher read performance. On the SGI Turing Cluster, the MPIIO implementations obtained slightly higher performance over the OpenSHMEM implementations on large files.

INTRODUCTION

This OpenSHMEM implementation of the Interleave Or Random (IOR) benchmark modifies version 2.10.3 that is available from `https://github.com/LLNL/ior`. The user guide of the IOR benchmark is also available at the code repository.

The IOR software can be used for benchmarking the performance of parallel I/O file systems using various interfaces and access patterns. IOR uses MPI for processor synchronization and can be configured to use POSIX, MPIIO, or HDF5 interfaces [4, 5].

The MPIIO interface supports parallel collective I/O where all processors cooperate to perform concurrent I/O into a global shared file. Each processor may use `MPI_File_set_view()` to specify a processor-centric view of data in the file using the MPI data type.

In this work, the source code for MPIIO was modified in a straight-forward manner to use OpenSHMEM communication primitives. One development effort was in emulating the MPI derived datatype used in the read/write operations and in setting the file view. Another effort was in implementing an internal cache in OpenSHMEM distributed shared memory to facilitate global collective I/O operations.

Section 2 contains a short background on MPI data types and MPIIO. Section 3 describes the implementation details in emulating the MPIIO using OpenSHMEM. The options used in experiments with IOR are described in Section 4. Section 5 is a summary of results on the Cray XK7 Titan supercomputer in the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory. Section 6 summarizes the results on the Durmstrang SGI Turing cluster and finally, the summary analysis is in Section 7.

BACKGROUND ON MPI DATA TYPE AND MPIIO

This section begins with a brief review of MPI derived datatypes and the role of MPI derived datatype in MPIIO. Further details can be found in [1, 2]. MPI derived datatype is a way to describe the layout of data in memory. This can be used to send and receive non-contiguous data (such as a sub-block of a matrix) or message with different datatypes (such as part of a C structure that contains integers and floating point numbers) without first packing or unpacking into a buffer. A high quality implementation of MPI may avoid extra data movement in packing/unpacking the individual items but directly transfer the necessary data in-place. The derived type can be composed out of pre-defined basic types (such as `MPI_INTEGER` or `MPI_LONG_LONG`) using composition operations such as `MPI_Type_contiguous()`, `MPI_Type_vector()` or `MPI_Type_create_subarray()`. There can be further nesting of derived types such as say subarray of vector of structures. The `MPI_Type_contiguous()` and `MPI_create_subarray()` type operations are used in the MPIIO option in IOR Benchmark. Figure 1 shows the code fragment for using `MPI_Type_create_subarray()` to describe a sub-matrix out of a larger matrix of integers (see Figure 2). The routine is sufficiently general to handle an n-dimensional array. The variables `array_of_sizes`, `array_of_subsizes`, and `array_of_starts` contain the global size of the matrix, local sizes of the sub-matrix, and starting offsets in the larger matrix. The assignment `order = MPI_ORDER_C` describes the data is laid out using C ordering where the last index varies fastest. Note that `MPI_ORDER_FORTRAN` assumes the first index varies fastest.

When a file is opened using `MPI_File_Open()`, the default file view is to access every byte in the file. This view can be changed by providing MPI derived data types for the basic elemental "etype" and "filetype" to `MPI_File_set_view()` to access non-contiguous data. The processor can then access *only* the data elements exposed in the "filetype" and transfer data *only* in units of "etype". Figure 3 shows how

```
{
int ndims = 2;
const int array_of_sizes[2] = {108, 108};
const int array_of_subsizes[2] = {100,100};
const int array_of_starts[2] = {4,4};
int order = MPI_ORDER_C;
MPI_Datatype  oldtype = MPI_INT;
MPI_Datatype  newtype;
int istatus = MPI_SUCCESS;

  istatus = MPI_Type_create_subarray(
              ndims,
              array_of_sizes,
              array_of_subsizes,
              array_of_starts,
              order,
              oldtype,
              &newtype );
  istatus  = MPI_Type_commit( &newtype );
}
```

Fig. 1. Code to use `MPI_Type_create_subarray`.

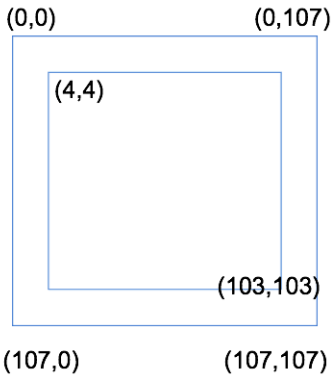

Fig. 2. Simple illustration of a sub-matrix out of a larger matrix.

different views can be imposed on the same file to access only a subset of the data.

MPIIO implements collective I/O where a group of processors collectively and in cooperation access the same shared file to perform I/O in large contiguous requests and then use the communication network to rearrange the data. Figure 4 shows that processor P0 and P1 each have a different file view. Here P0 uses one file view to access the “red" data items, and P1 uses a different file view to access the “yellow" data items. From the perspective of processor P0, P0 may issue a read operation for “contiguous" sequence of data, however, at a lower level, each data item may require performing a seek operation to position the internal file pointer and then require performing a small I/O operation to access the data item. If each processor performs independent I/O operations, then this may require many small I/O operations and may lead to poor I/O performance. A more efficient alternative is to perform a collective operation where each processor performs I/O operations in large contiguous blocks. Then P0 may be reading data that is required by P1 (and vice versa). The processors can use the communication network to rearrange the file data to assign the data to the correct processor. This collective way of performing I/O may lead to significant improvement in I/O performance [6]. Note that extra data in “holes” (the blank slots in Figure 4) may still be accessed to preserve the original data in the file. This may lead to extra overhead in transferring more data than strictly required. Collective I/O with and without explicit file view is available in the MPIIO option of IOR Benchmark.

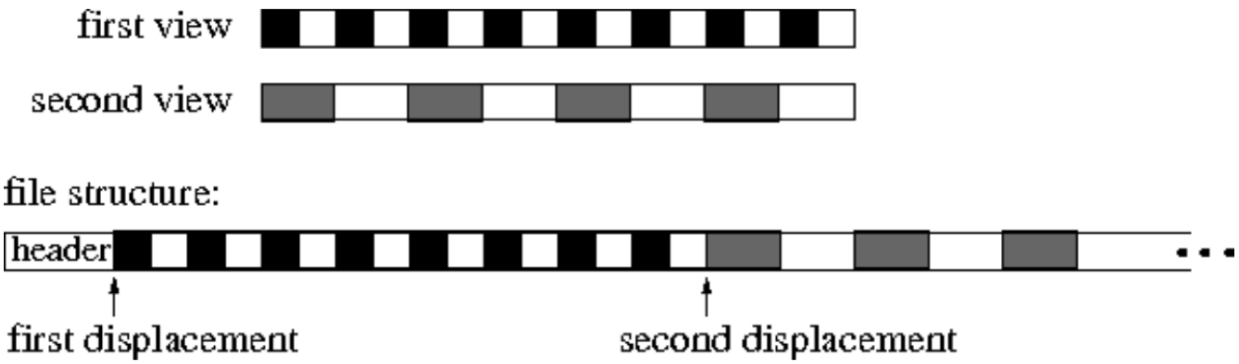

Fig. 3. Simple illustration of using MPI derived type to impose different file views on the same file.

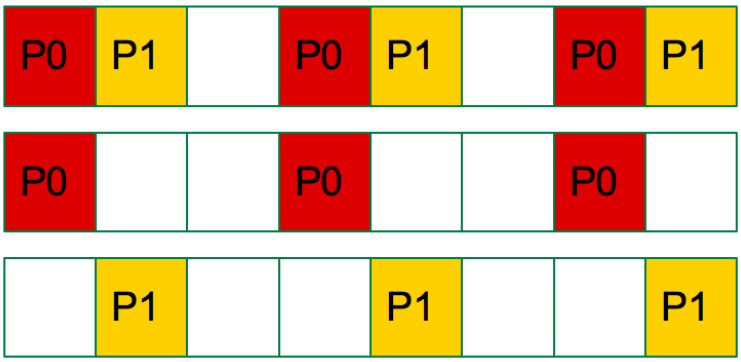

Fig. 4. Simple illustration of collective I/O on two processors.

IMPLEMENTATION DETAILS

The OpenSHMEM version of IOR emulates the collective I/O capabilities in MPIIO in performing concurrent I/O operations to a common shared file. In this implementation, each processor performs I/O operations in large contiguous blocks to read data from the shared file into an internal cache in distributed shared memory, then each processor uses `shmem_getmem()` or `shmem_putmem()` to access the required data. The modified data is then written out to the shared file in large contiguous blocks. In this implementation, the distributed memory cache is in “static" storage memory and its size is determined at compile time. Note that there can be contiguous access in memory to (and from) non-contiguous accesses in the shared file.

The OpenSHMEM implementation emulates MPI derived datatype using routines with similar interfaces such as `SHMEM_Type_contiguous()`, `SHMEM_Type_vector()`, `SHMEM_Type_commit()` and `SHMEM_Type_create_subarray()`. To enable a straight-forward conversion of the MPIIO version of IOR, the OpenSHMEM implementation further emulates MPIIO with a similar function interface for other parts of the MPI library including `MPI_Info` by `SHMEM_Info`, `MPI_Status` by `SHMEM_Status`. The MPI Communicators are also emulated using `SHMEM_COMM_WORLD` for `MPI_COMM_WORLD`, `SHMEM_COMM_SELF` for `MPI_COMM_SELF`. Since the team extensions available in Cray SHMEM are not widely available, the MPI sub-groups are emulated using the description of `(pe_start,logpe_stride,pe_size)` that is used in OpenSHMEM global reduction operations. Other MPIIO functions such as `MPI_File_read_at_all()` and `MPI_File_write_at_all()` are emulated in `SHMEM_File_read_at_all()` and `SHMEM_File_write_at_all()` with a similar interface.

Internally, this implementation stores the byte sizes and byte offsets of each derived type in a list. During a collective operation such as `SHMEM_read_at_all()` with a non-trivial file view, the processors determine the total global size and extent of the collective I/O operation using global reduction operations. The processors then read large contiguous data blocks from the shared file into the disk cache in distributed shared memory. If the distributed disk cache is smaller than the global extent of I/O request, the global collective operation will be decomposed into multiple smaller steps where each step transfers data that will fit entirely in the disk cache. Each processor will traverse its encoding of the derived type in the `SHMEM_File_read_at_all()` request and in the file view to perform `shmem_getmem()` in a read operation (or `shmem_putmem()` in a write operation). The contiguous data blocks will be written from the cache back into the shared file in a `SHMEM_write_at_all()` operation. The implementation also performs atomic operations to correctly update the maximum file size.

Since IOR was originally designed to use MPI for distributed communication, converting all communication to OpenSHMEM would be a significant undertaking. In this implementation, only the collective I/O capability of MPIIO is emulated using OpenSHMEM. Thus interoperability of OpenSHMEM and MPI is required. While this issue of interoperability with MPI is not explicitly addressed in the OpenSHMEM standard, several commonly available OpenSHMEM implementations (such as SGI MPT, Cray SHMEM, OpenMPI) do support interoperability or co-existence with MPI. However, minor tailoring may be needed for different OpenSHMEM implementations. For example one implementation of MPI may require calling `shmem_init()` and `shmem_finalize()` whereas calling `shmem_init()` or `shmem_finalize()` may cause a failure in another implementation of MPI. It might be helpful to software developers if this issue of interoperability with MPI can be addressed in the standard, or informally among developers of OpenSHMEM.

EXPERIMENTS WITH IOR

The experiments with IOR are focused on comparing the collective I/O capabilities (with and without file view) to a shared file in MPIIO and emulation of such capabilities using OpenSHMEM. IOR is highly configurable with many different options and our experiments used only a subset of the available options. The following options "-c -w -r -b 1m -t 1m" are used in the experiments to enable collective I/O ("-c") in write and read operations ("-w -r"). The transfers are performed in 1 MByte requests ("-b 1m -t 1m"). The IOR "-s" stripe count is used to adjust the file size to be twice the total available memory of the compute nodes. Generating a large file that is twice the total available memory of the compute nodes is one way to defeat aggressive caching of file data into memory by the operating system or Lustre I/O system. If the shared file is not sufficiently large, it may reside entirely in disk buffers and cached in memory and thus the IOR benchmark cannot truly measure the parallel disk I/O performance. The IOR options "-Q 16 -Z -X 16" are used to randomize the read access to avoid having the same processor re-read the same disk data that it produced. This is another way to defeat aggressive disk caching. The "-a MPIIO -c " options are used to enable collective I/O in MPIIO and similarly "-a SHMEMIO -c " options are used for collective I/O in OpenSHMEM implementation. The "-V " option is used to enable the `MPI_File_set_view` and similar `SHMEM_File_set_view` capabilities.

CRAY XK7 TITAN

The Cray XK7 Titan machine in the Oak Ridge Leadership Computing Facility (OLCF) at ORNL consists of 18,688 compute nodes. Each compute node has 32 GBytes of memory, one 16-core AMD Opteron 6200 Interlagos processor and a NVidia Kepler Graphics Processing Unit (GPU) with 6 GBytes of device memory (see Figure 5). Each Interlagos processor has eight 256-bit floating point compute units shared by 16 integer cores. Two compute nodes are connected to a Cray Gemini network device (NIC) that has over 160 GBytes/sec of routing capacity (see Figure 6). The global network is arranged as a three-dimensional (3D) torus. The Random Ring benchmark in the HPC Challenge Benchmark Suite [3] achieves transfer rates of about 0.055 GBytes/sec per rank and the STREAMS benchmark for testing memory subsystem achieves about 72 GBytes/sec. Figure 5 shows the compute architecture of the Cray XK7 and the configuration of the Gemini network is depicted in Figure 6.

For this IOR benchmark, only the CPU cores were used and the GPUs were untouched. The batch policy on Titan cannot guarantee allocation of contiguous nodes and this can lead to some variations in the communication performance. For example, two MPI tasks may be adjacent nodes on the 3D network in one batch run, but may require many hops across the network in another batch submission.

The native Cray SHMEM implementation (module cray-shmem version 7.2.5) was used to build the benchmark.

The Lustre parallel file system on Titan is a system-wide shared resource so I/O performance can be affected by other concurrently running applications. Conversely, running the IOR benchmark may also adversely affect the performance of other concurrently running applications. Since each compute node has 32 GBytes of main memory and 16 cores (2 GBytes per core), we design our experiment to write out 4 GBytes per core. Moreover, we increase the stripe count for large files, using 8 stripes for 1024 Gbytes, 16 stripes for 2048 GBytes, 32 stripes for 4096 GBytes. For example, running 512 tasks on 32 nodes (16 tasks per node) would require a file that is 32 * (2 * 32 GBytes) = 2048 GBytes.

Tables 1, 2, and 3 compare performance of MPIIO (with and without file view) with SHMEMIO. The results suggest the MPIIO implementation have a higher write performance over SHMEMIO but

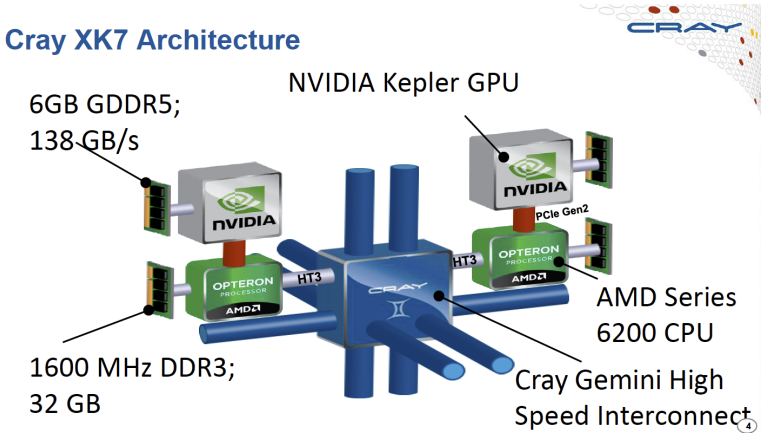

Fig. 5. Compute node architecture of Cray XK7. Picture provided by Cray and available at `https://www.olcf.ornl.gov/training-event/titan-workshop/`.

256 task, 1024 GiB (8)	MPI no file view	SHMEM no file view	MPI file view	SHMEM file view
Max write	1,777 MiB/s	1,655 MiB/s	1,807 MiB/s	1,681 MiB/s
Max read	878 MiB/s	1,330 MiB/s	872 MiB/s	1,342 MiB/s

Table 1. IOR performance on 256 tasks on 16 nodes on 1024 GBytes file and stripe count 8.

SHMEMIO may have a slightly higher read performance.

512 task, 2048 GiB (16)	MPI no file view	SHMEM no file view	MPI file view	SHMEM file view
Max write	2,947 MiB/s	2,538 MiB/s	2,972 MiB/s	2,523 MiB/s
Max read	1,369 MiB/s	2,260 MiB/s	1,321 MiB/s	2,473 MiB/s

Table 2. IOR performance on 512 tasks on 32 nodes on 2048 GBytes file and stripe count 16.

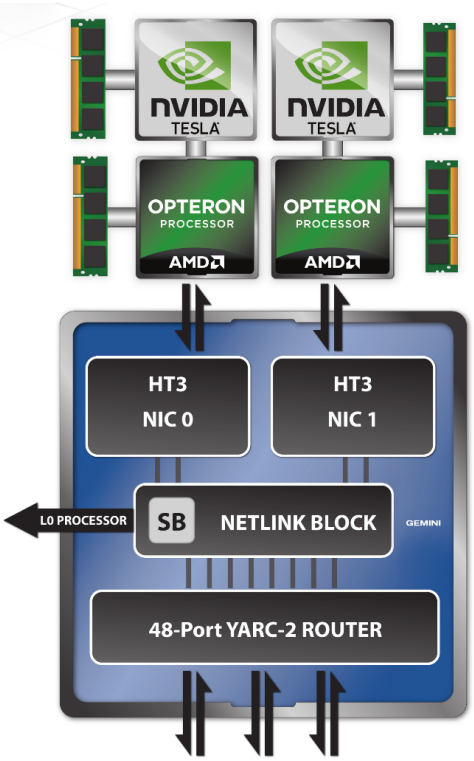

Fig. 6. Each Cray XK7 compute node is interfaced to the Gemini interconnect through HyperTransport 3.0 technology. Picture provided by Cray and available at `http://www.cray.com/Products/Computing/XK7.aspx`.

1024 task, 4096 GiB (32)	MPI no file view	SHMEM no file view	MPI file view	SHMEM file view
Max write	4,593 MiB/s	3,688 MiB/s	4,492 MiB/s	3,450 MiB/s
Max read	2,404 MiB/s	3,989 MiB/s	2,008 MiB/s	3,698 MiB/s

Table 3. IOR performance on 1024 tasks on 64 nodes on 4096 GBytes file and stripe count 32.

SGI TURING CLUSTER

The SGI Turing Cluster consists of 16 compute nodes, each node has two Intel Xeon E5-2660 processors, each Xeon has 10 cores running at 2.6 GHz (105 Watts) for a total of 20 physical cores (or 40 virtual cores with Intel Hyper-Threading enabled). Each node has eight 16 GBytes DDR4 memory cards for a total of 128 GBytes of memory. Each node also has a fast 1 TByte 10K revolutions per minute (RPM) SATA hard disk with 6 Gbits/sec peak transfer rate, one Intel Xeon Phi 7120P PCIE accelerator and is connected with a Mellanox ConnectX-4 VPI adapter card, EDR IB (100 Gbits/sec) and 100 Gbits/s ethernet, single-port QSFP, PCIe3.0 x16 network connector. The nodes are connected with a Mellanox InfiniBand Edge Switch with 36 QSFP ports with a non-blocking switching capacity of 7.2 Tbits/sec. The Turing Cluster has access to a Lustre file system (`/lustre/esscfs`) with 8 OST. For this benchmark only 8 lustre clients were used with only 1 MPI task per node and the directory was configured with stripe count of 8. The native SGI MPT implementation of SHMEM (module mpt version 2.13) was used to build the benchmark.

Table 4 shows the performance for a small file (4 GBytes per task) that can be easily cached in memory. The results show very high I/O performance. MPIIO has much higher write performance compared to SHMEMIO and slightly better read performance over the implementation using SHMEMIO as well. Table 5 shows the performance for a larger file that is twice the available memory (256 GBytes per task). The read and write performance using OpenSHMEM implementation is comparable or slightly slower compared to MPIIO.

4GB/task, total 32GB	MPI no file view	SHMEM no file view	MPI file view	SHMEM file view
Max write	1,605 MiB/s	695 MiB/s	3,578 MiB/s	974 MiB/s
Max read	4,191 MiB/s	3,031 MiB/s	4,056MiB/s	2,941 MiB/s

Table 4. IOR performance on 8 tasks on small 32GB file.

256GB/task, total 2048GB	MPI no file view	SHMEM no file view	MPI file view	SHMEM file view
Max write	3,601 MiB/s	3,714 MiB/s	3,735 MiB/s	3,655 MiB/s
Max read	5,286 MiB/s	4,867 MiB/s	5,266 MiB/s	4,888 MiB/s

Table 5. IOR performance on 8 tasks on 2048 GB file.

SUMMARY

A new interface using OpenSHMEM to emulate the parallel collective I/O in MPIIO has been developed as a new feature for IOR parallel I/O benchmark. Since the IOR benchmark uses the advanced capabilities of `MPI_File_set_view()` capability of MPIIO, and nested MPI derived datatypes constructed with `MPI_Type_create_subarray()` and `MPI_Type_contiguous()`, the OpenSHMEM implementation also emulates the capabilities and interfaces for MPI datatype in the implementation of `SHMEM_File_set_view()` with similar nested derived datatypes using `SHMEM_TYPE_create_subarray()` and `SHMEM_Type_contiguous()`. An internal cache in OpenSHMEM distributed global memory allows each process to perform I/O operations in large contiguous blocks and then redistributes the data using OpenSHMEM `shmem_putmem()` and `shmem_getmem()` operations.

The parallel collective IOR benchmark using MPIIO interface and OpenSHMEM interface were compared on the Cray XK7 Titan at OLCF and on the SGI Turing cluster. The results suggest the MPIIO obtains slightly higher write and read performance compared to the OpenSHMEM implementation.

ACKNOWLEDGMENT

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This work was supported by the United States Government and used resources of the Computational Research and Development Programs and the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory.

Notice: "This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (`http://energy.gov/downloads/doe-public-access-plan`).

References

[1] Gropp, W., Hoefler, T., Thakur, R., and Lusk, E. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, USA, 2014.

[2] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, third edition ed. The MIT Press, Cambridge, Massachusetts, USA, 2014.

[3] Luszczek, P., Dongarra, J. J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., Mccalpin, J., Bailey, D., and Takahashi, D. Introduction to the HPC challenge benchmark suite. Tech. rep., University of Tennessee, Knoxville, TN, 2005. Available at `http://icl.cs.utk.edu/hpcc/pubs/`.

[4] Shan, H., Antypas, K., and Shalf, J. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 42:1–42:12.

[5] Shan, H., and Shalf, J. Using IOR to analyze the I/O performance for HPC platforms, 06 2007. Permalink: `http://escholarship.org/uc/item/9111c60j`".

[6] Thakur, R., Gropp, W., and Lusk, E. Optimizing noncontiguous access in MPI-IO. *Parallel Computing 28*, 1 (2002), 83–105.