# h5perf User Guide

**1 Introduction**

In order to measure the performance of a file system, the HDF5 library includes h5perf – a parallel file system benchmark tool. The original version of the tool performed testing using only one-dimensional datasets, but its capabilities were extended to handle two-dimensional datasets in order to demonstrate in a more realistic way how different access patterns and storage layouts affect I/O performance. The selected strategy for the implementation of 2D geometry allows porting most of the assumptions and constraints of the original design.

**2 One-dimensional Testing**

The use of one-dimensional datasets and transfer buffers provides an easy way to test the parallel performance of HDF5. Fundamental findings like the performance impact of data contiguity can be quickly demonstrated.

The main configuration parameters for testing are *bytes-per-process* per dataset, *transfer-buffer-size*, *block-size*, and *num-processes*. Block size is a unit size in h5perf and does not refer to the blocks of the file system. Each dataset consists of a linear array of size *bytes-per-process* \* *num-processes*. A sample configuration in memory of a transfer buffer containing four blocks is shown in Figure 1.

*transfer-buffer-size*

*block-size*

**Figure 1 Transfer buffer configuration for 1D geometry**

h5perf provides two types of access patterns for testing: contiguous (default) and interleaved. The contiguous pattern divides the dataset into *num-processes* contiguous regions that are assigned to each process. For example, an execution of h5perf with the following parameters

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| *num-processes* | 3 |
| *bytes-per-process* | 8 |
| *block-size* | 2 |
| *transfer-buffer-size* | 4 |

defines a dataset of 8 \* 3 = 24 bytes, blocks of 2 bytes, and a transfer buffer of 4 bytes.

Because *bytes-per-process* is twice as large as *transfer-buffer-size*, every process must issue two transfer requests to complete access in its allocated region of the dataset. In Figure 2, the numbers show the byte locations corresponding to each process; the colors distinguish the regions that are accessed during each transfer request (blue for the first request and red for the second request).

|00|00|00|00|11|11|11|11|22|22|22|22|

**Figure 2 Data organization for 1D contiguous access pattern**

Note that each process can write the entire contents of its associated transfer buffer in a single I/O operation. This pattern yields good throughput performance because it minimizes high latency costs.

When interleaved pattern is selected (option -I), each process does not transfer the memory buffer at once. Instead, the constitutive blocks are written separately on interleaved storage locations, i.e. after a process writes a block, it skips *num-processes* block locations to write the next block, successively. The resulting interleaved pattern is shown in Figure 3.

|00|11|22|00|11|22|00|11|22|00|11|22|

**Figure 3 Data organization for 1D interleaved access pattern**

Now each process has to perform 4 / 2 = 2 lower level I/O operations in non-contiguous locations every time a transfer request is issued. Therefore, performance will decrease significantly because the numerous small write operations incur latency costs several times.

**3 Two-dimensional Testing**

One of the goals in extending the original h5perf design was to give the user flexibility in switching from 1D to 2D geometry using a single option (-g). This means, however, that the options defining the sizes of datasets, buffers, and blocks have to be interpreted in a different way.

For 2D testing, the dataset is a square array of size (*bytes-per-process \* num-processes*) × (*bytes-per-process \* num-processes*), and every block is a square array of size *block-size* × *block-size*. Because the total amount of bytes required by the dataset and blocks are squared quantities, it is important to use fairly small values for these parameters (up to the order of kilobytes) in order to avoid having extremely large datasets and blocks. As for the transfer buffer, its configuration is not unique; it depends on the selected access pattern.

When the contiguous pattern is selected (default), the transfer buffer is a rectangular array of size *block-size* × *transfer-buffer-size*. On the other hand, when the interleaved pattern is used (option –I), the transfer buffer becomes an array of size *transfer-buffer-size* × *block-size*. Figure 4 shows the case where the transfer buffer contains four blocks, i.e. *transfer-buffer-size =* 4 \* *block-size*.

*block-size*

*transfer-buffer-size*

*transfer-buffer-size*

*block-size*

Contiguous pattern

Interleaved pattern

**Figure 4 Transfer buffer configurations for 2D geometry**

The contiguous pattern divides the first dimension (columns) of the dataset evenly into *num-processes* sections. Given that the storage implementation of two-dimensional datasets is based on row-major ordering, such partition effectively divides the storage into *num-processes* contiguous regions. For example an execution of h5perf with the following parameters

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| *num-processes* | 3 |
| *bytes-per-process* | 4 |
| *block-size* | 2 |
| *transfer-buffer-size* | 12 |

defines a dataset of (4\*3)×(4\*3) bytes, blocks of 2×2 bytes, and a transfer buffer of 2×12 bytes with the logical organization shown in Figure 5.

Because the number of columns of the transfer buffer is the same as that of the dataset in this case, each processor can transfer its associated buffer entirely using a single I/O operation. Figure 6 shows the data storage in the file and the view of each process. As before, the data elements that are accessed during the first and second transfer requests are shown in blue and red, respectively.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

**Figure 5 Logical data organization for 2D contiguous access pattern**

P0’s view |0 0 … 0 0 0 … 0 |

P1’s view | 1 1 … 1 1 1 … 1 |

P2’s view | 2 2 … 2 2 2 … 2|

File |0 0 … 0 0 0 … 0 1 1 … 1 1 1 … 1 2 2 … 2 2 2 … 2|

**Figure 6 Physical data organization for 2D contiguous access pattern**

Because storage is implemented only in one dimension, the logical and physical data organization appear to be different when 2D geometry is enabled. Note that for 1D geometry, the logical and physical data organization are one and the same.

The interleaved pattern uses the vertical implementation of the transfer buffer shown in Figure 4, and directs different processes to access dataset regions that are next to each other logically in the horizontal direction. Every time a process transfers the memory buffer into the file, it skips *num-processes* locations horizontally to perform the next transfer, and so on. The application of the interleaved option defines a dataset of (4\*3)×(4\*3) bytes, blocks of 2×2 bytes, and a transfer buffer of 12×2 bytes with the logical organization shown in Figure 7.

The row-major storage implementation of the file causes each process to perform 12 small I/O operations every time it transfers the memory buffer into the file. Figure 8 details the file access for the first two logical rows of the dataset.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |
| 0 | 0 | 1 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 2 | 2 |

**Figure 7 Logical data organization for 2D interleaved access pattern**

P0’s view |0 0 0 0 0 0 0 0 … |

P1’s view | 1 1 1 1 1 1 1 1 ... |

P2’s view | 2 2 2 2 2 2 2 2 … |

File |0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2 … |

**Figure 8 Physical data organization for 2D interleaved access pattern**

The interleaved pattern causes different processes to perform many small I/O operations in overlapped locations. As a consequence, interleaved pattern generally exhibits low performance due to the impact of latency costs.

**4 MPI Communication Modes**

h5perf allows testing with the two MPI modes of communication: independent (default) and collective. In independent mode, each process operates independently from the rest of the processes. The resulting I/O performance will depend mostly on the access pattern, i.e. a contiguous pattern will yield higher performance than an interleaved pattern. As an illustration, we have the 1D interleaved access pattern,

|00|11|22|00|11|22|00|11|22|00|11|22|

Independent mode would cause each process to perform two small I/O operations every time the memory buffer is transferred to disk. On the other hand, collective mode (option -C) enables different processes to coordinate their transfer requests to improve I/O performance. A common strategy is to aggregate the many small requests of different processes that may be non-contiguous into fewer larger requests to minimize latency. The shown example would require a single large I/O operation every time a buffer transfer is requested. Therefore, only two collective I/O operations would be needed to complete the access of the dataset.

Access with interleaved patterns is more likely to benefit from collective communications. In cases where collective access does not provide an improvement in performance, many MPI implementations will switch to independent mode in order to eliminate unnecessary overhead.

**5 Storage Layout**

h5perf provides two types of storage layouts: contiguous (default) and chunked. Contiguous layout defines a single contiguous region of storage for the dataset. In contrast, chunked layout allocates several smaller regions of similar dimensions called chunks. Although HDF5 allows the user to define arbitrary dimensions for the chunks, h5perf uses the same dimensions of the blocks for the chunks, i.e. each block is stored in a separate chunk.

Since accessing data on a chunk requires a single I/O operation, the use of chunked layout can improve performance for certain patterns. For example, consider a 16×16 dataset with a transfer buffer and block of size 4×4. In contiguous storage, a transfer buffer request would require each process to perform 4 non-contiguous write operations of 4 bytes each as shown in Figure 9.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

P0’s view |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0… |

P1’s view | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1… |

File |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0…1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1… |

**Figure 9 Data organization for contiguous storage layout**

In a chunked layout (option -c), transferring a block would require only one write operation because each chunk is an unit of contiguous storage as shown in Figure 10. In this case, a single I/O operation of 16 bytes is performed instead of 4 I/O operations.

Chunk 0

Chunk 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Chunk 3

Chunk 2

P0’s view |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

Chunk 0 |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

P1’s view |1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |

Chunk 2 |1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |

**Figure 10 Data organization for chunked storage layout**

There are no restrictions on the location of each chunk in an HDF5 file. However, consistency issues associated to parallel access require the early allocation of chunks in the same order of the blocks in the logical organization. This ordering strategy is used for the chunking emulation for tests with POSIX and MPI-IO APIs.

**6 Programming Scheme**

h5perf allows the specification of a range of values for some of its execution parameters such as the number of processes, the transfer buffer size, number of test iterations, number of files, and number of datasets. The following pseudo-code illustrates how h5perf iterates over each of these parameters:

for each number of processes

for each transfer buffer size

for each test iteration

for each file

file open

for each dataset

access dataset

end for

file close

end for

end for

end for

end for

h5perf uses a unit stride for most of the iteration parameters with the exception of the number of processes and the transfer buffer size. For these parameters, the value during each iteration is either the double or half of the value at the previous iteration.

Although the selection of a proper value or range for a particular execution parameter depends on the benchmarking objectives, it is recommended to set the number of test iterations to at least 3. In this way, the attained peak performance can be considered to be most representative of the system, because poorer results may be caused by the presence of concurrent processes in the system.

**7 Tool Command Syntax**

The following is a description of the syntax and options used in the command line when executing h5perf. Some of the options specify a range of values for a particular parameter, e.g. number of processes. h5perf iterates over the range and summarizes the performance statistics in the output.

**Syntax:**

h5perf [-h | --help]

h5perf [*options*]

**Purpose:**

Tests Parallel HDF5 I/O performance.

**Description:**

h5perf is a tool for testing the I/O performance of the Parallel HDF5 Library. The tool can perform testing with one-dimensional and two-dimensional buffers and datasets. The following environment variables have the following effects on h5perf behavior:

|  |  |
| --- | --- |
| HDF5\_NOCLEANUP | If set, h5perf does not remove the test data files.  (*Default:* Data files are removed.) |
| HDF5\_MPI\_INFO | Must be set to a string containing a list of semi-colon separated key=value pairs for the MPI INFO object. |
| HDF5\_PARAPREFIX | Sets the prefix for parallel output data files. |

**Options and Parameters:**

|  |  |
| --- | --- |
| These terms are used as follows in this section: | |
| *file* | A filename |
| *size* | A size specifier, expressed as an integer greater than or equal to 0 (zero) followed by a size indicator:       K for kilobytes (1024 bytes)       M for megabytes (1048576 bytes)       G for gigabytes (1073741824 bytes)  Example: 37M specifies 37 megabytes or 38797312 bytes. |
| *N* | An integer greater than or equal to 0 (zero) |

|  |  |  |  |
| --- | --- | --- | --- |
| -h, --help | | | |
|  | Prints a usage message and exits. | | |
| -a *size*, --align=*size* | | | |
|  | Specifies the alignment size for objects in the HDF5 file.  (*Default:* 1)  File objects greater than or equal in size to the threshold size (option –T) will be aligned on a file address that is a multiple of the alignment size. | | |
| -A *api\_list*, --api=*api\_list* | | | |
|  | Specifies which APIs to test. *api\_list* is a comma-separated list with the following valid values:   |  |  | | --- | --- | | phdf5 | Parallel HDF5 | | mpiio | MPI-I/O | | posix | POSIX |   (*Default:* All APIs)  Example, --api=mpiio,phdf5 specifies that the MPI I/O and Parallel HDF5 APIs are to be tested, but not the POSIX API. | |
| -B *size*, --block-size=*size* | | | |
|  | Controls the block size within the transfer buffer.  (*Default:* Half the number of bytes per process per dataset)  Block size versus transfer buffer size:  The *transfer buffer size* is the size of a buffer in memory. The data in that buffer is broken into *block size* pieces and written to the file.  Transfer buffer size is discussed below with the -x (or --min-xfer-size) and -X (or --max-xfer-size) options.  The pattern in which the blocks are written to the file is described in the discussion of the -I (or --interleaved) option. | | |
| -c, --chunk | | | |
|  | Creates HDF5 datasets in chunked layout. If MPI-I/O or POSIX APIs are selected for testing, h5perf performs chunking emulation.  (*Default:* Off) | | |
| -C, --collective | | | |
|  | Use collective I/O for the MPI I/O and Parallel HDF5 APIs.  (*Default:* Off, i.e., independent I/O)  If this option is set and the MPI-I/O and PHDF5 APIs are in use, all the processes will coordinate their transfer requests to reduce the number of actual I/O operations. | | |
| -d *N*, --num-dsets=*N* | | | |
|  | Sets the number of datasets per file.  (*Default:* 1)  During file access, h5perf iterates over the number of datasets. | | |
| -D *debug\_flags*, --debug=*debug\_flags* | | | |
|  | Sets the debugging level. *Debug\_flags* is a comma-separated list of debugging flags with the following valid values:   |  |  | | --- | --- | | 1 | Minimal debugging | | 2 | Moderate debugging (“not quite everything”) | | 3 | Extensive debugging (“everything”) | | 4 | All possible debugging (“the kitchen sink”) | | r | Raw data I/O throughput information | | t | Times, in additions to throughputs | | v | Verify data correctness |   (*Default:* No debugging)  Example: --debug=2,r,v specifies to run a moderate level of debugging while collecting raw data I/O throughput information and verifying the correctness of the data.  Throughput values are computed by dividing the total amount of transferred data (excluding metadata) over the time spent by the slowest process. Several time counters are defined to measure the data transfer time and the total elapsed time; the latter includes the time spent during file open and close operations. A number of iterations can be specified with the option -i (or --num-iterations) to create the desired population of measurements from which maximum, minimum, and average values can be obtained. The timing scheme is the following  for each test iteration  initialize elapsed time counter  initialize data transfer time counter  for each file  start and accumulate elapsed time counter  file open  start and accumulate data transfer time counter  access entire file  stop data transfer time counter  file close  stop elapsed time counter  end for  save elapsed time counter  save data transfer time counter  end for   The reported write throughput is based on the accumulated data transfer time, while the write open-close throughput uses the accumulated elapsed time. |
| -e *size*, --num-bytes=*size* | | | |
|  | Specifies the number of bytes per process per dataset.  (*Default:* 256K for 1D, 8K for 2D)  Depending on the selected geometry, each test dataset can be a linear array of size *bytes-per-process \* num-processes* or a square array of size *(bytes-per-process \* num-processes)* × *(bytes-per-process \* num-processes)*. The number of processes is set by the -p (or --min-num-processes) and -P (or --max-num-processes) options. | | |
| -F *N*, --num-files=*N* | | | |
|  | Specifies the number of files.  (*Default:* 1) | | |
| -g, --geometry | | | |
|  | Selects 2D geometry for testing.  (*Default:* Off, i.e., 1D geometry) | | |
| -i *N*, --num-iterations=*N* | | | |
|  | Sets the number of test iterations to perform.  (*Default:* 1) | | |
| -I, --interleaved | | | |
|  | Sets interleaved block I/O.  (*Default:* Contiguous block I/O)  Interleaved and contiguous patterns in 1D geometry:  When a contiguous access pattern is chosen, the dataset is evenly divided into *num-processes* regions and each process writes data to its assigned region. When interleaved blocks are written to a dataset, space for the first block of the first process is allocated in the dataset, then space is allocated for the first block of the second process, etc., until space is allocated for the first block of each process, then space is allocated for the second block of the first process, the second block of the second process, etc.  For example, with a three process run, 512KB bytes-per-process, 256KB transfer buffer size, and 64KB block size, each process must issue two transfer requests to complete access to the dataset.  Contiguous blocks of the first transfer request are written as follows:  1111----2222----3333----   Interleaved blocks of the first transfer request are written as follows:   123123123123------------   The actual number of I/O operations involved in a transfer request depends on the access pattern and communication mode. When using independent I/O with an interleaved access pattern, each process performs four small non-contiguous I/O operations per transfer request. If collective I/O is turned on, the combined content of the buffers of the three processes will be written using one collective I/O operation per transfer request. | | |
| -m, --mpi-posix | | | |
|  | Sets use of MPI-posix driver for HDF5 I/O.  (*Default:* MPI-I/O driver) | | |
| -o *file*, --output=*file* | | | |
|  | Sets the output file name for writing data to *file*.  (*Default:* None) | | |
| -p *N*, --min-num-processes=*N* | | | |
|  | Sets the minimum number of processes to be used.  (*Default:* 1) | | |
| -P *N*, --max-num-processes=*N* | | | |
|  | Sets the maximum number of processes to be used.  (*Default:* All MPI\_COMM\_WORLD processes) | | |
| -T *size*, --threshold=*size* | | | |
|  | Sets the threshold size for aligning objects in the HDF5 file.  (*Default:* 1)  File objects greater than or equal in size to the threshold size will be aligned on a file address which is a multiple of the alignment size (option –a). | | |
| -w, --write-only | | | |
|  | Performs only write tests, not read tests.  (*Default:* Read and write tests) | | |
| -x *size*, --min-xfer-size=*size* | | | |
|  | Sets the minimum transfer buffer size.  (*Default:* Half the number of bytes per processor per dataset)  This option and the -X *size* option (or --max-xfer-size=*size*) control *transfer-buffer-size*, the size of the transfer buffer in memory. In 1D geometry, the transfer buffer is a linear array of size *transfer-buffer-size*. In 2D geometry, the transfer buffer is a rectangular array of size *block-size* × *transfer-buffer-size*, or *transfer-buffer-size* × *block-size* if the interleaved access pattern is selected. | | |
| -X *size*, --max-xfer-size=*size* | | | |
|  | Sets the maximum transfer buffer size.  (*Default:* The number of bytes per processor per dataset) | | |

**8 Examples**

The illustrated examples described in this guide can be executed using the following command lines.

Example of Figure 2

mpiexec –n 3 h5perf –B 2 –e 8 –p 3 –P 3 –x 4 –X 4

Example of Figure 3

mpiexec –n 3 h5perf –B 2 –e 4 –p 3 –P 3 –x 4 –X 4 -I

Example of Figure 5

mpiexec –n 3 h5perf –B 2 –e 4 –p 3 –P 3 –x 12 –X 12 -g

Example of Figure 7

mpiexec –n 3 h5perf –B 2 –e 4 –p 3 –P 3 –x 12 –X 12 -g -I

Example of Figure 9

mpiexec –n 2 h5perf –B 4 –e 4 –p 2 –P 2 –x 4 –X 4 –g

Example of Figure 10

mpiexec –n 2 h5perf –B 4 –e 4 –p 2 –P 2 –x 4 –X 4 –g –c

**9 Tool Output**

The performance statistics that h5perf displays by default are throughput figures (MB/s). These figures are computed by dividing the amount of data accessed over the benchmarking times. Consider the following sample command line:

mpiexec -n 3 h5perf -A phdf5 -B 2000 -e 8000 -p 3 -P 3 -x 4000 -X 4000 -i 3

The resulting output shown next lists the parameter values and the resulting throughput for write and read operations including and excluding the time of file open and close operations. As mentioned above, the peak performance is to be considered as the most representative since other results may be hindered by concurrent processes in the system. Additional information such as the data transfer times and used file names can be displayed by using the debugging option (option -D).

HDF5 Library: Version 1.8.7

rank 0: ==== Parameters ====

rank 0: IO API=phdf5

rank 0: Number of files=1

rank 0: Number of datasets=1

rank 0: Number of iterations=3

rank 0: Number of processes=3:3

rank 0: Number of bytes per process per dataset=8000

rank 0: Size of dataset(s)=24000:24000

rank 0: File size=24000:24000

rank 0: Transfer buffer size=4000:4000

rank 0: Block size=2000

rank 0: Block Pattern in Dataset=Contiguous

rank 0: I/O Method for MPI and HDF5=Independent

rank 0: Geometry=1D

rank 0: VFL used for HDF5 I/O=MPI-I/O driver

rank 0: Data storage method in HDF5=Contiguous

rank 0: Env HDF5\_PARAPREFIX=not set

rank 0: Dumping MPI Info Object(469762048) (up to 1024 bytes per item):

object is MPI\_INFO\_NULL

rank 0: ==== End of Parameters ====

Number of processors = 3

Transfer Buffer Size: 4000 bytes, File size: 0.02 MBs

# of files: 1, # of datasets: 1, dataset size: 0.02 MBs

IO API = PHDF5 (w/MPI-I/O driver)

Write (3 iteration(s)):

Maximum Throughput: 133.89 MB/s

Average Throughput: 83.97 MB/s

Minimum Throughput: 52.03 MB/s

Write Open-Close (3 iteration(s)):

Maximum Throughput: 28.39 MB/s

Average Throughput: 15.28 MB/s

Minimum Throughput: 8.57 MB/s

Read (3 iteration(s)):

Maximum Throughput: 313.73 MB/s

Average Throughput: 257.14 MB/s

Minimum Throughput: 211.92 MB/s

Read Open-Close (3 iteration(s)):

Maximum Throughput: 41.01 MB/s

Average Throughput: 36.29 MB/s

Minimum Throughput: 31.39 MB/s