h5perf, a parallel file system benchmarking tool

The HDF Group
October, 2008

Introduction

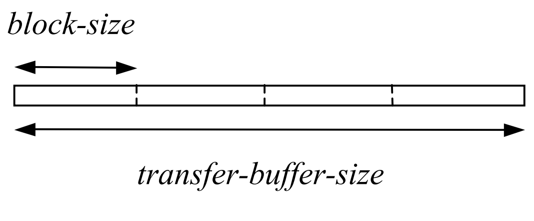
In order to measure the performance of a file system, the HDF5 library includes `h5perf` –a parallel file system benchmark tool. The original version of the tool performed testing using only one-dimensional datasets. `h5perf` capabilities were extended to handle two-dimensional datasets in order to demonstrate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

This document describes the data organization, access patterns and the impact in performance defined by the different options of the tool.

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*. 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 the following:



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

```
mpirun –np 3 h5perf –B 2 –e 8 –p 3 –P 3 –x 4 –X 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 the following scheme, the numbers show the byte locations corresponding to each process; the colors distinguish the regions that are accessed during each transfer request

|0000|**0000**|1111|**1111**|2222|**2222**|

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

When interleaved pattern is selected, 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 previous command can be modified to use interleaved access pattern as follows

```
mpirun –np 3 h5perf –B 2 –e 4 –p 3 –P 3 –x 4 –X 4 -I
```

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

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 in latency costs many times.

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 requires by the dataset and blocks are powers of two, 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 contiguous pattern is selected, the transfer buffer is a rectangular array of size *block-size* × *transfer-buffer-size*. On the other hand, when interleaved pattern is used, the transfer buffer becomes an array of size *transfer-buffer-size* × *block-size*. Figure 1 shows the case where the transfer buffer contains four blocks, i.e. *transfer-buffer-size* = 4 * *block-size*.

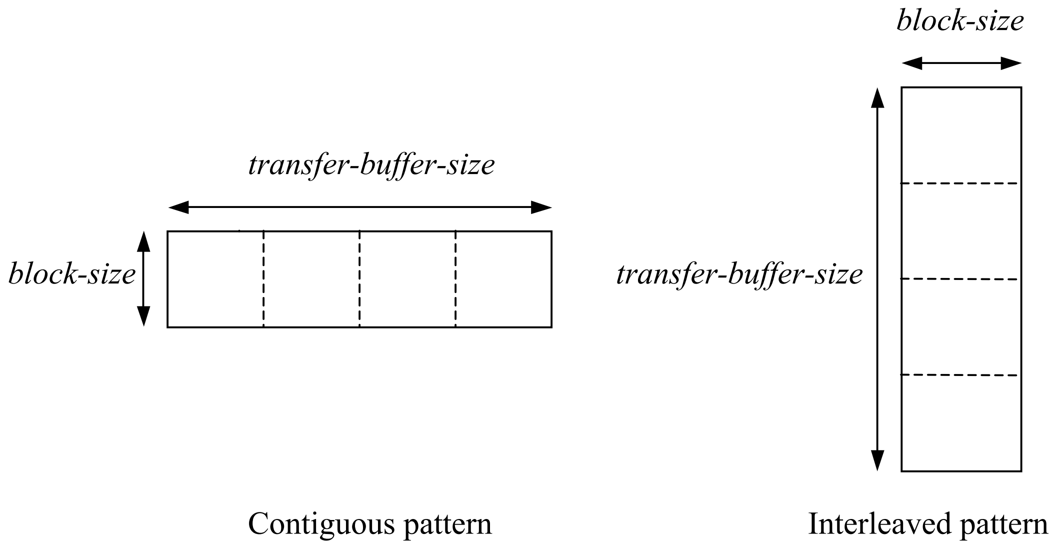


Figure 1 Transfer buffer configurations for 2D geometry

Contiguous pattern divides the first dimension (columns) 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 the following command

```
mpirun –np 3 h5perf –B 2 –e 4 –p 3 –P 3 –x 12 –X 12 -g
```

defines a dataset of 12×12 bytes, blocks of 2×2 bytes, and a transfer buffer of 2×12 bytes with the logical organization shown in Figure 2.

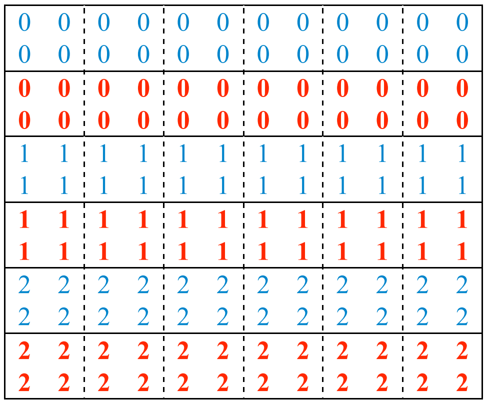


Figure 2 Logical data organization for contiguous access pattern

Because the number of columns of the transfer buffer is the same as that of the dataset, each processor can transfer the entire buffer using a single I/O operation. The following scheme shows the data storage in the file and the view of each process,

```
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|
```

Interleaved pattern uses the vertical implementation of the transfer buffer shown in Figure 1, and directs different processes to access dataset regions that are next to each other logically. 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. As before, the previous command can be modified to use interleaved access,

```
mpirun –np 3 h5perf –B 2 –e 4 –p 3 –P 3 –x 12 –X 12 -g -I
```

which defines a dataset of 12×12 bytes, blocks of 2×2 bytes, and a transfer buffer of 12×2 bytes with the logical organization shown in Figure 3.

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 3 Logical element organization for interleaved access pattern

The row-major storage implementation of the file causes each process to perform 12 small I/O operations every time they transfer the memory buffer into the file. The following scheme details the file access for the first two logical rows of the dataset,

```
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    …     |
```

The interleaved pattern causes different processes to perform many small I/O operations in overlapped locations. However, a fact unique to 2D geometry is the presence of gaps in the storage regions corresponding to the same interleaved request as described by the colors. As a consequence, interleaved pattern generally exhibits low performance due to the impact of latency costs.

MPI communication modes

The MPI standard provides two modes of communication: independent and collective. In independent mode, each process operates independently from the rest of processes. The resulting I/O performance will depend mostly on the logical access pattern, i.e. a contiguous pattern will yield higher performance than a non-contiguous interleaved pattern. As an illustration, we have the interleaved example in 1D geometry

|001122001122**001122001122**|

Independent mode would cause each process to perform two small I/O operations every time the memory buffer is transferred. On the other hand, collective mode 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 transfer of the buffer is requested. Therefore, only two I/O operations are needed to complete the access of the dataset.

Interleaved patterns are more likely to benefit from collective communications. In cases, where collective access does not provide an improvement in performance, MPI will switch to independent mode in order to eliminate unnecessary overhead.

Storage layout

`h5perf` provides two types of storage layouts: contiguous 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, writing a block of 256×256 bytes in contiguous storage would require 256 non-contiguous write operations of 256 bytes each. In chunked layout, this would require only one write operation of 64K bytes because each block is stored in a separate chunk.

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 was used for the chunking emulation for the tests with POSIX and MPI-IO APIs