Introduction to Scientific I/O

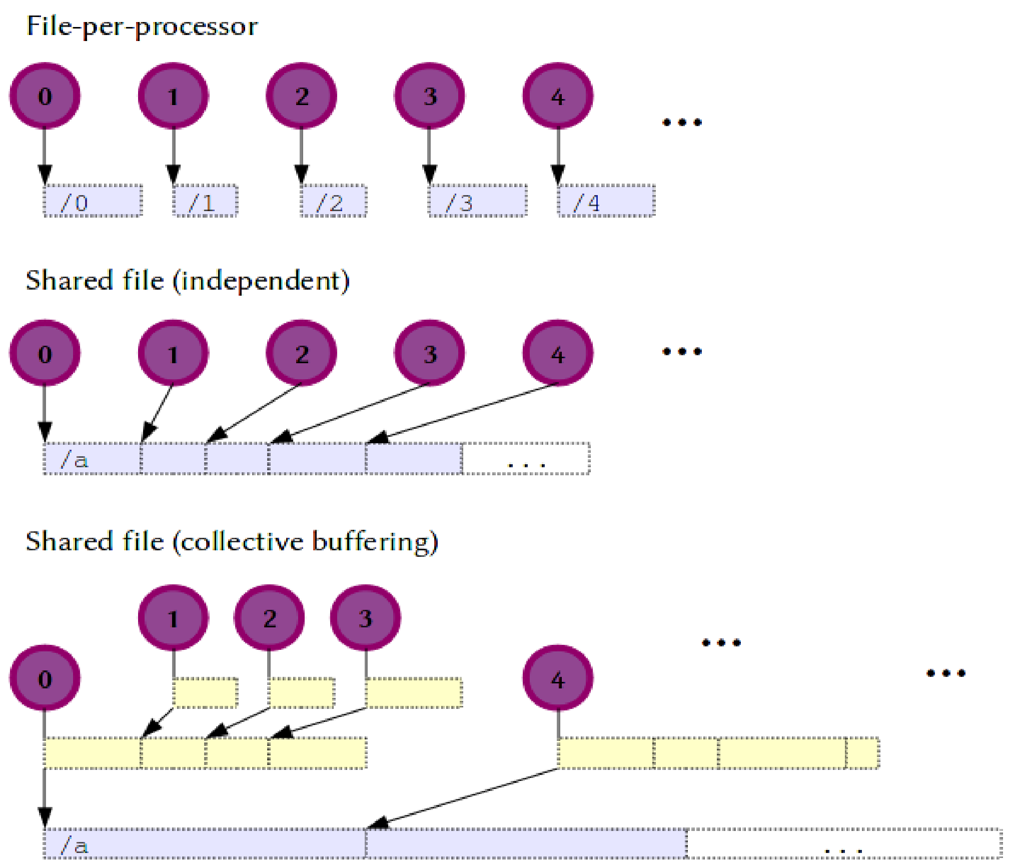
I/O is commonly used by scientific applications to achieve goals like:

- storing numerical output from simulations for later analysis;
- implementing 'out-of-core' techniques for algorithms that process more data than can fit in system memory and must page data in from disk;
- and checkpointing to files that save the state of an application in case of system failure.

In most cases, scientific applications write large amounts of data in a structured or sequential 'append-only' way that does not overwrite previously written data or require random seeks throughout the file.

Most HPC systems are equipped with a parallel file system such as Lustre or GPFS that abstracts away spinning disks, RAID arrays, and I/O subservers to present the user with a simplified view of a single address space for reading and writing to files. Three common methods for an application to interact with the parallel file system are shown in Figure 1.1.

Figure 1.1. Three common methods for writing in parallel from many nodes.



File-per-processor

File-per-processor access is perhaps the simplest to implement since each processor maintains its own filehandle to a unique file and can write independently without any coordination with other processors. Parallel file systems often perform well with this type of access up to several thousand files, but synchronizing metadata for a large collection of files introduces a bottleneck. One way to mitigate this is to use a 'square-root' file layout, for example by dividing 10,000 files into 100 subdirectories of 100 files.

Even so, large collections of files can be unwieldy to manage. Even simple utilitles like ls can break with thousands of files. This may not be a concern in the case of checkpointing, where the files are typically thrown away except in the infrequent case of a system failure. Still, another disadvantage of file-per-processor access is that later restarts will require the same number and layout of processors, because these are implicit in the structure of the file hierarchy.

Shared-file (independent)

Shared file access allows many processors to share a common filehandle but write independently to exlusive regions of the file. This coordination can take place at the parallel file system level. However, there can be significant overhead for write patterns where the regions in the file may be contested by two or more processors. In these cases, the file system uses a 'lock manager' to serialize access to the contested region and guarantee file consistency. Especially at high concurrency, such lock mechanisms can degrade performance by orders of magnitude. Even in ideal cases where the file system is guaranteed that processors are writing to exclusive regions, shared file performance can be lower compared to file-per-processor.

The advantage of shared file access lies in data management and portability, especially when a higher-level I/O format such as HDF5 or netCDF is used to encapsulate the data in the file. These libraries provide database-like functionality, cross-platform compatibility, and other features that are helpful or even essential when storing or archiving simulation output.

Shared-file with collective buffering

Collective buffering is a technique used to improve the performance of shared-file access by offloading some of the coordination work from the file system to the application. A subset of the processors is chosen to be the 'aggregators' who collect the data from all other processors and pack it into contiguous buffers in memory that are then written to the file system. Reducing the number of processors that interact with the I/O subservers is often beneficial, because it reduces contention.

Originally, collective buffering was developed to reduce the number of small, noncontiguous writes, but another benefit that is important for file systems such as Lustre is that the buffer size can be set to a multiple of the ideal transfer size preferred by the file system. This is explained in the next section on Lustre striping.

The Lustre File System

Lustre is a scalable, POSIX-compliant parallel file system designed for large, distributed-memory systems, such as Hopper and Edison at NERSC. It uses a server-client model with separate servers for file metadata and file content, as illustrated schematically in Figure 2.1.

For example, on Hopper, the /scratch and /scratch2 file systems each have a single metadata server (which can be a bottleneck when working with thousands of files) and 156 'Object Storage Targets' that store the contents of files.

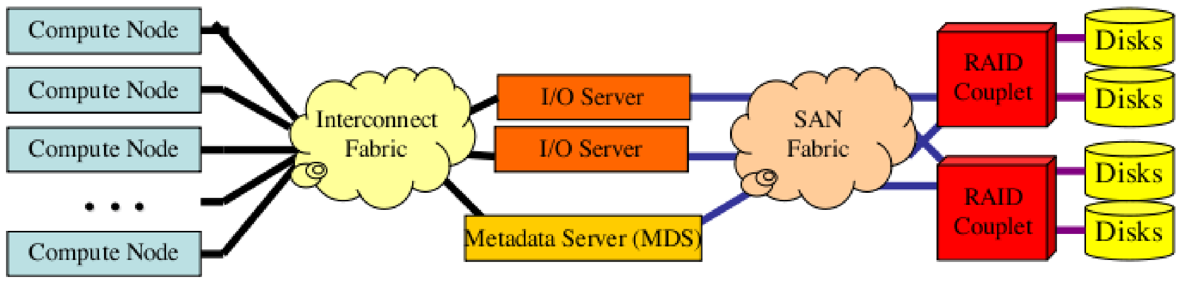


Figure 2.1. The general architecture of cluster file systems such as Lustre that have separate object and metadata stores. In Lustre, the I/O servers are called Object Storage Targets (OSTs).

Although Lustre is designed to correctly handle any POSIX-compliant I/O pattern, in practice it performs much better when the I/O accesses are aligned to Lustre's fundamental unit of storage, which is called a stripe and has a default size (on NERSC systems) of 1MB.

Striping is a method of dividing up a shared file across many OSTs, as shown in Figure 2.2. Each stripe is stored on a different OST, and the assignment of stripes to OSTs is round-robin.

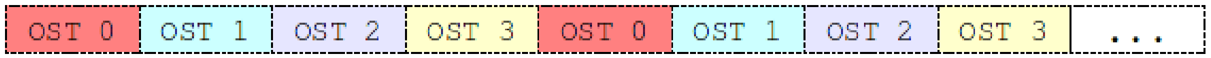


Figure 2.2. A shared file striped across four OSTs.

The reason for striping a file is to increase the available bandwidth. Writing to several OSTs in parallel aggregates the bandwidth of each of the individual OSTs. This is the exact same principal behind the use of striping in a RAID array of disks. In fact, the disk arrays backing a Lustre file system are also RAID striped, so you can think of Lustre striping as a second layer of striping that allows you to access every single physical disk in the file system if you want the maximum available bandwidth (i.e. by striping over all available OSTs).

Lustre allows you to modify three striping parameters for a shared file:

- the stripe count controls how many OSTs the file is striped over (for example, the stripe count is 4 for the file shown in Figure 2.2);
- the stripe size controls the number of bytes in each stripe; and
- the start index chooses where in the list of OSTs to start the round-robin assignment (the default value -1 allows the system to choose the offset in order to load balance the file system).

The default parameters on Hopper are [count=2, size=1MB, index=-1], but these can be changed and viewed on a per-file or per-directory basis using the commands:

```
lfs setstripe [file,dir] -c [count] -s [size] -i [index]
```

```
lfs getstripe [file,dir]
```

A file automatically inherits the striping parameters of the directory it is created in, so changing the parameters of a directory is a convenient way to set the parameters for a collection of files you are about to create. For instance, if your application creates output files in a subdirectory called output/, you can set the stiping parameters on that directory once before your application runs, and all of your output files will inherit those parameters.

NERSC has also created the following shortcut commands for applying roughly optimal striping parameters for a range of common file sizes and access patterns:

```
stripe_fpp [file,dir]
```

```
stripe_small [file,dir]
```

```
stripe_medium [file,dir]
```

```
stripe_large [file,dir]
```

For more details on these striping shortcuts, see our page on tuning Lustre I/O performance.

Unfortunately, striping parameters must be set before a file is written to (either by touching a file and setting the parameters or by setting the parameters on the directory). To change the parameters after a file has been written to, the file must be copied over to a new file with the desired parameters, for instance with these commands:

```
lfs setstripe newfile [new parameters]
cp oldfile newfile
```

The HDF5 Library

The Hierarchical Data Format v5 (HDF5) library is a portable I/O library used for storing scientific data in a database-like organization. HDF5's 'object database' data model enables users to focus on high-level concepts of relationships between data objects rather than descending into the details of the specific layout of every byte in the data file. Additional information can be found in the HDF5 Tutorial from the HDF Group.

An HDF5 file has a root group / under which you can add groups, datasets of various shapes, single-value attributes, and links among groups and datasets. The HDF5 library provides a 'logical view' of the file's contents as a graph, as in Figure 3.1. The flexibility to create arbitrary graphs of objects allows HDF5 to express a wide variety of data models. The library transparently handles how these logical objects of the file map to the actual bytes in the file. In many ways, HDF5 provides an abstracted filesystem-within-a-file that is portable to any system with the HDF5 library, regardless of the underlying storage type, filesystem, or conventions about binary data (i.e. 'endianess').

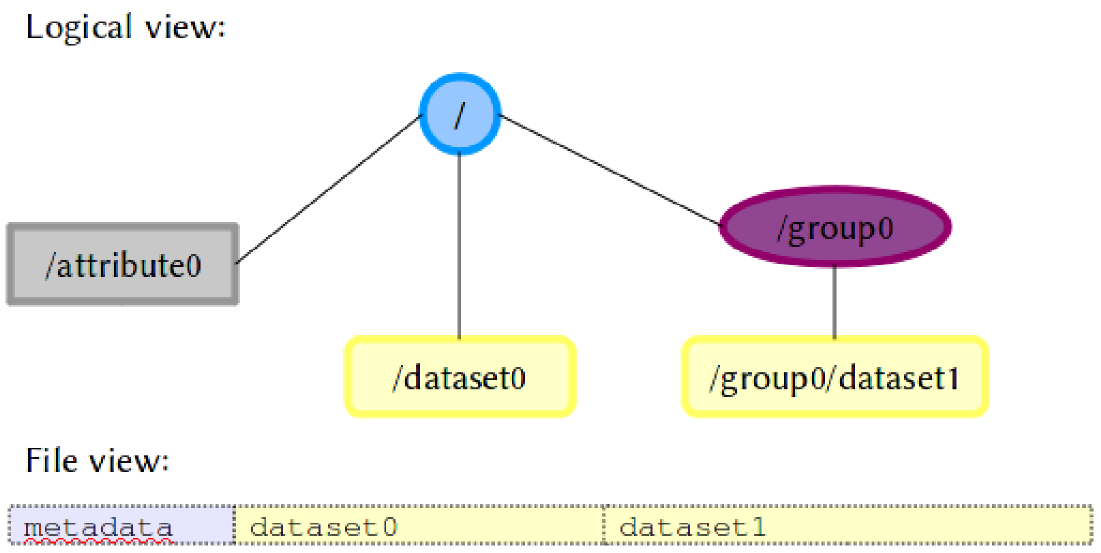


Figure 3.1. An example HDF5 file showing an attribute, a group, and two datasets. Datasets are stored as flattened arrays within the file, and attribute and group information is part of the HDF5 metadata (not to be confused with the filesystem's metadata for the file).

The example file shown in Figure 3.1 could be created with the following C code:

```
#include

/* create the file */
file_id = H5Fcreate("myfile.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

/* create attribute0 */
space_id = H5Screate(H5S_SCALAR);
attr_id = H5Acreate(file_id, "attribute0", H5T_NATIVE_INT32, space_id, H5P_DEFAULT);
H5Awrite(attr_id, H5T_NATIVE_INT32, 42);
H5Aclose(attr_id);
H5Sclose(space_id);

/* create dataset0 */
space_id = H5Screate_simple(rank, dims, maxdims);
dset_id = H5Dcreate(file_id, "dataset0", H5T_NATIVE_FLOAT, space_id, H5P_DEFAULT, H5P_DEFAULT,
H5P_DEFAULT);
H5Dwrite(dset_id, H5T_NATIVE_FLOAT, H5S_ALL, H5S_ALL, H5P_DEFAULT, somedata0);
H5Dclose(dset_id);
H5Sclose(space_id);

/* create group0 */
group_id = H5Gcreate(file_id, "/group0", H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
/* and dataset1 */
space_id = H5Screate_simple(rank, dims, maxdims);
dset_id = H5Dcreate(group_id, "dataset1", H5T_NATIVE_FLOAT, space_id, H5P_DEFAULT, H5P_DEFAULT,
dset_id = H5Dcreate(group_id, "dataset1", H5T_NATIVE_FLOAT, space_id, H5P_DEFAULT, H5P_DEFAULT, H5P_D
H5Dwrite(dset_id, H5T_NATIVE_FLOAT, H5S_ALL, H5S_ALL, H5P_DEFAULT, somedata1);
H5Dclose(dset_id);
H5Sclose(space_id);
H5Gclose(group_id);

/* finished! */
```

Parallel HDF5

The HDF5 library can be compiled to provide parallel support using the MPI library. The HDF Group maintains a special tutorial on parallel topics.

An HDF5 file can be opened in parallel from an MPI application by specifying a parallel 'file driver' with an MPI communicator and info structure. This information is communicated to HDF5 through a 'property list,' a special HDF5 structure that is used to modify the default behavior of the library. In the following code, a file access property list is created and set to use the MPI-IO file driver:

```
/* create the file in parallel */
fapl_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(fapl_id, mpi_comm, mpi_info);
file_id = H5Fcreate("myparfile.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);
```

The MPI-IO file driver defaults to independent mode, where each processor can access the file independently and any conflicting accesses are handled by the underlying parallel file system. Another option for independent access is the MPI-POSIX file driver, which bypasses the MPI-IO layer and uses direct POSIX (e.g. fwrite) calls that are coordinated internally by HDF5. In some scenarios, this lighter-weight MPI-POSIX driver exhibits better performance, especially on systems with a poorly implemented MPI-IO library. To set the MPI-POSIX file driver, replace the H5Pset_fapl_mpio call above with:

```
H5Pset_fapl_mpiposix(fapl_id, mpi_comm, use_gpfs_hints);
```

If the parameter use_gpfs_hints is set to 1, additional optimizations for the GPFS filesystem are invoked. The /project and global scratch filesystems on Hopper and Edison use GPFS, but currently the HDF5 optimizations for GPFS are not functional. There is an on-going project to resolve this.

Alternatively, the MPI-IO file driver can be set to collective mode to enable collective buffering. This is not specified during file creation, but rather during each transfer (read or write) to a datset by passing a dataset transfer property list to the read or write call. The following code shows how to write collectively to 'dataset0' in our previous example file:

```
dxpl_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(dxpl_id, H5FD_MPIO_COLLECTIVE);

/* describe a 1D array of elements on this processor */
memspace = H5Screate_simple(1, count, NULL);

/* map this processor's elements into the shared file */
filespace = H5Screate_simple(1, mpi_size*count, NULL);
offset = mpi_rank * count;
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, &offset, NULL, &count, NULL);

H5Dwrite(dset_id, H5T_NATIVE_FLOAT, memspace, filespace, dxpl_id, somedata0);
```

When accessing a dataset in parallel, two dataspaces are neccessary: one to specify the shape of the data in each processor's memory, and another to specify the layout of the data in the file. In the above code, each processor has the same number count of elements in a 1D array, and they write into a 1D array in the file according to their processor rank. The layout is specified with a 'hyperslab' in HDF5, and although this regular 1D array example is perhaps the simplest possible example, more complicated layouts are described in the next section.

Scientific I/O in HDF5

We will now show how HDF5 can be used to store the following three types of datasets commonly found in scientific computing:

- rectilinear 3D grids with balanced, cubic partitioning;

- rectilinear 3D grids with unbalanced, rectangular partitioning; and
- contiguous but size-varying arrays, such as those found in adapative mesh refinement.

Balanced 3D Grids

Assume we have a 3D grid of known dimension grid_size[3] and a partitioning partition[3] of the grid into equally sized blocks that are distributed across MPI tasks. We can write a task's block into an HDF5 file using the following code:

```
/* map the 1D MPI rank to a block in the 3D grid */
grid_index[0] = mpi_rank % partition[0];
grid_index[1] = (mpi_rank / partition[0]) % partition[1];
grid_index[2] = mpi_rank / (partition[0] * partition[1]);
/* calculate the size of the block (same for all tasks) */
block_size[0] = grid_size[0] / partition[0];
block_size[1] = grid_size[1] / partition[1];
block_size[2] = grid_size[2] / partition[2];
/* the file will hold the entire grid */
filespace = H5Screate_simple(3, grid_size, NULL);
/* while the hyperslab specifies only the block */
offset[0] = grid_index[0] * block_size[0];
offset[1] = grid_index[1] * block_size[1];
offset[2] = grid_index[2] * block_size[2];
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL, block_size, NULL);
```

Unbalanced 3D Grids

Now suppose that not all tasks share the same block size. If we are only given the dimension of the block on each task as block_size[3], we will need to gather the block sizes from all tasks to be able to be able to calculate the offsets. The offsets for a given block are simply the sum of block sizes that precede the block in each dimension. These can be calculated with the following code:

```
MPI_Allgather(block_size, 3, MPI_INT, all_block_sizes, 3, MPI_INT, MPI_COMM_WORLD);
for (i=0; i<grid_indx[0]; i++) {
    offset[0] += all_block_sizes[3*get_rank_of_block(i, grid_index[1], grid_index[2]) + 0];
}
for (i=0; i<grid_indx[1]; i++) {
    offset[1] += all_block_sizes[3*get_rank_of_block(grid_index[0], i, grid_index[2]) + 1];
}
for (i=0; i<grid_indx[2]; i++) {
    offset[2] += all_block_sizes[3*get_rank_of_block(grid_index[0], grid_index[1], i) + 2];
}
```

Note that we can calculate the MPI rank of a given grid index using the formula:

```
get_rank_of_block(i,j,k) = i + j * partition[0] + k * partition[0] * partition[1]
```

1D AMR Arrays

Storing a flattened 1D AMR array is similar to the unbalanced 3D grid case. If each MPI task has nelems in its local array, then MPI_Scan can be used to find the offset into the array on disk as follows:

```
memspace = H5Screate_simple(1, &nelems, NULL);
MPI_Allreduce(&nelems, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
filespace = H5Screate_simple(1, &sum, NULL);
MPI_Scan(&nelems, &offset, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, &offset, NULL, &nelems, NULL);
```

Optimizations for HDF5 on Lustre

The Lustre file system performs best when writes are aligned to stripe boundaries. This minimizes the overhead of the distributed lock manager which has to serialize access when a stripe is accessed by more than one client. Two methods to achieve stripe alignment on writes are described below.

Chunking and Alignment for Balanced 3D Grids

When every MPI task has the same amount of data to write into a 3D grid, a data layout techique called chunking can be enabled in HDF5. This layout maps the 3D block on each task to its own contiguous 1D section of the file on disk. Figure 6.1 shows an example in 2D where 4 MPI tasks each write their own 4x4 block contiguously into a file, and an index in the HDF5 metadata keeps track of where each block resides in the file. A second HDF5 tuning parameter called the alignment can be set to pad out the size of each chunk to a multiple of the stripe size. Although padding wastes space in the file on disk, the gain in write bandwidth often outweighs this, especially on HPC systems like Hopper and Edison with large scratch file systems. A more problematic restriction of the chunked layout, however, is that future parallel accesses to the file (for instance with an analysis or visualization tool) are optimized for reads that are multiples of the chunk size.

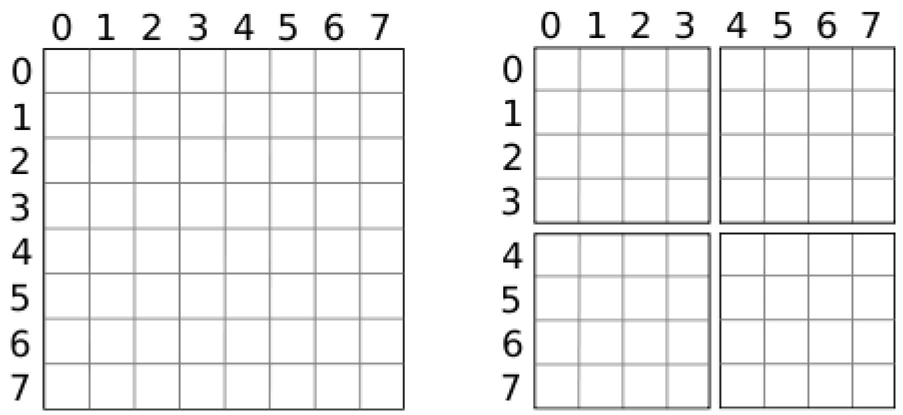


Figure 6.1. Four tasks write to a contiguous 2D grid on the left versus a chunked grid on the right.

In our 3D code example, chunking and alignment can be enabled using the following:

```
fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_alignment(fapl, 0, stripe_size);
file = H5Fcreate("myparfile.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl);
```

```
dcpl = H5Pcreate(H5P_DATASET_CREATE);
H5Pset_chunk(dcpl, 3, chunk_dims);
H5Dcreate(file, "mydataset", type, filespace, H5P_DEFAULT, dcpl, H5P_DEFAULT);
```

HDF5 uses a B-tree to index the location of chunks inside the file. The default size for this B-tree occupies only a few kilobytes in the HDF5 metadata entry. When this space is exceeded, additional B-trees are created, leading to many small metadata writes when thousands of chunks are written to a file. The small metadata writes, even when they are padded and aligned using the alignment parameter still adversely affect write performance. HDF5 provides the following mechanism for increasing the default size of the B-tree so that it is roughly the same size as a stripe:

```
btree_ik = (stripe_size - 4096) / 96;
fcpl = H5Pcreate(H5P_FILE_CREATE);
H5Pset_istore_k(fcpl, btree_ik);
file = H5Fcreate("myparfile.h5", H5F_ACC_TRUNC, fcpl, fapl);
```

HDF5 also maintains a metadata cache for open files, and if enough metadata fills the cache, it can cause an eviction that interrupts other file accesses. Usually, it is better to disabled these evictions

Metadata in HDF5 files is cached by the HDF5 library to improve access times for frequently accessed items. When operating in a sequential application, individual metadata items are flushed to the file (if dirty) and evicted from the metadata cache. However, when operating in a parallel application, these operations are deferred and batched together into eviction epochs, to reduce communication and synchronization overhead. At the end of an eviction epoch (measured by the amount of dirty metadata produced), the processes in the application are synchronized and the oldest dirty metadata items are flushed to the file.

To reduce the frequency of performing small I/O operations, it is possible to put the eviction of items from the HDF5 library's metadata cache entirely under the application's control with the following:

```
mdc_config.version = H5AC__CURR_CACHE_CONFIG_VERSION;
H5Pget_mdc_config(file, &mdc_config)
mdc_config.evictions_enabled = FALSE;
mdc_config.incr_mode = H5C_incr__off;
mdc_config.decr_mode = H5C_decr__off;
H5Pset_mdc_config(file, &mdc_config);
```

This sequence of calls disables evictions from the metadata cache, unless H5Fflush is called or the file is closed. Suspending automatic eviction of cached metadata items also prevents frequently dirtied items from being written to the file repeatedly. Suspending metadata evictions may not be appropriate for all applications however, because if the application crashes before the cached metadata is written to the file, the HDF5 file will be unusable.

Collective Buffering

In the case where each task writes different amounts of data, chunking cannot be used. Instead, a different optimization called Collective buffering (or two-phase IO can be used if collective mode has been enabled, as described in Section 4. In fact, collective buffering can also be used in the balanced 3D grid case discussed above instead of chunking, and the optimization in HDF5 for disabling cache evictions also applies to collective buffering.

Collective buffering works by breaking file accesses down into to stages. For a collective read, the first stage uses a subset of MPI tasks (called aggregators) to communicate with the IO servers (OSTs in Lustre) and read a large chunk of data into a temporary buffer. In the second stage, the aggregators ship the data from the buffer to its destination among the remaining MPI tasks using point-to-point MPI calls. A collective write does the reverse, aggregating the data through MPI into buffers on the aggregator nodes, then writing from the aggregator nodes to the IO servers. The advantage of collective buffering is that fewer nodes are communicating with the IO servers, which reduces contention. In fact, Lustre prefers a one-to-one mapping of aggregator nodes to OSTs.

Since the release of mpt/3.3, Cray has included a Lustre-aware implementation of the MPI-IO collective buffering algorithm. This implementation is able to buffer data on the aggregator nodes into stripe-sized chunks so that all read and writes to the Lustre filesystem are automatically stripe aligned without requiring any padding or manual alignment from the developer. Because of the way Lustre is designed, alignment is a key factor in achieving optimal performance.

Several environment variables can be used to control the behavior of collective buffering on Hopper and Edison. The MPIIO_MPICH_HINTS variable specifies hints to the MPI-IO library that can, for instance, override the built-in heuristic and force collective buffering on (in csh):

```
% setenv MPIIO_MPICH_HINTS "*:romio_cb_write=enable:romio_ds_write=disable"
```

Placing this command in your batch file before calling aprun will cause your program to use these hints. The * indicates that the hint applies to any file opened by MPI-IO, while romio_cb_write controls collective buffering for writes and romio_ds_write controls data sieving for writes, an older collective mode optimization that is no longer used and can interfere with collective buffering. The options for these hints are enabled, disabled, or automatic (the default value, which uses the built-in heuristic).

It is also possible to control the number of aggregator nodes using the cb_nodes hint, although the MPI-IO library will automatically set this to the stripe count of your file.

When set to 1, the MPICH_MPIIO_HINTS_DISPLAY variable causes your program to dump a summary of the current MPI-IO hints to stderr each time a file is opened. This is useful for debugging and as a sanity check againt spelling errors in your hints.

The MPICH_MPIIO_XSTATS variable enables profiling of the collective buffering algorithm, including binning of read/write calls and timings for the two phases. Setting it to 1 provides summary data, while setting it to 2 or 3 provides more detail.

More detail on MPICH runtime environment variables, including a full list and description of MPI-IO hints, is available from the intro_mpi man page on Hopper and Edison. Please also see our documentation on tuning Lustre I/O on both machines.