RFC: Breaking Free from the Collective Requirement for HDF5 Metadata Operations

Mohamad Chaarawi

Quincey Koziol

John Mainzer

This RFC proposes a new feature in HDF5 that breaks the collective requirement for operations that modify metadata and allows independent space allocation from the file’s “end of address” (EOA) location. The current implementation requires all operations that modify metadata or allocate space in an HDF5 file to be collective for synchronization purposes. This requirement is detrimental to the performance of some HPC applications and puts several limitations on application developers.

This work is still at the design phase, and this RFC details the several design options that have been considered. Several use cases will be outlined and used to draw out the pros and cons of the different design options.

# Introduction

Parallel HDF5 is layered on top of MPI to provide parallel I/O functionality. MPI defines a set of parallel I/O functions that can be used by processes to access data on disk independently or collectively. Independent I/O operations are similar to serial POSIX I/O, where each process individually accesses data on disk. On the other hand, collective I/O operations require that all processes participate in the I/O function call.

Some collective operations, such as file opens, require that all processes’ arguments to the function specified are the same. Other collective functions, such as writes, do not have that requirement, and the amount of data that each process accesses can be different. During a collective write call, processes that don’t have any data to write can specify a 0 byte selection; however they *do* have to call the function since it is collective.

When collective I/O is performed, the MPI library can apply different algorithms underneath the hood to optimize the I/O to disk. For example, when an MPI implementation detects that many processes are writing very small chunks of data to disk at once, it can have a small set of processes combine data from all the processes and write data to disk in larger blocks, improving performance on most parallel file systems.

Parallel HDF5 is positioned above MPI-I/O in the application’s software stack, and instead of treating the file as just a stream of bytes (as MPI-I/O does), it gives the application a structured view of the file. The user organizes his file using different HDF5 objects (groups, datasets, etc.) and dependencies (links) between those objects. The HDF5 library is responsible for maintaining information about the file structure in the form of metadata, which is also stored in the file.

Parallel HDF5 allows all processes that opened the file to access dataset elements independently or collectively. However due to synchronization issues, all operations that need to modify the structure of the file, or in other words the file’s metadata, are required to be collective. For example, to create a subgroup in the root group of the file, a process needs to tell the other processes that it is performing that operation. This is done to notify the other processes that this group now exists and that they should not attempt to re-create it or simultaneously modify the file’s metadata.

Another function that requires synchronization between the processes is space allocation within the file. Again, all processes must coordinate to avoid allocating overlapping space.

In this RFC, we propose several design options that will remove the collective requirement for space allocation and operations that require modifying the file metadata. Any solution proposed has to achieve the following:

1) Allow an independent (but globally visible), scalable "fetch-and-add" operation to the "end of allocated space" (EOA) value in the file (to allow new space in the file to be allocated to an individual process, without involving all the other processes).

2) Allow [the metadata for] an object to be locked, updated, unlocked, and made visible to other processes without requiring a collective operation (to allow changes to the structure of a file).

Before the design options are presented, the next section lists some basic use cases and some HPC application use cases that need to be considered when designing a solution.

# Use cases

This section presents several use cases for typical parallel applications using HDF5 and motivates the need for this RFC. At the same time, it outlines the challenges a solution must address. All of the use cases below assume that the MPI application is writing to a single, shared file.

Some of the basic use cases include:

U.1) A parallel application with *n* processes, where each process creates an object in the root group for its private use. This is currently done by having all processes call the object creation function *n* times, despite the fact that each process needs only 1 object. But since each operation is modifying the file’s metadata, *n* creation calls are required.

U.2) A process within a parallel application performs independent read/write operations to a dataset while another process comes in and modifies the metadata of that dataset, making the former process’s access possibly obsolete. Currently, this use case could not happen, because all processes are required to be present at the time of metadata modification. However, this use case becomes possible when the collective requirement to metadata modification is removed and should be taken into consideration by any design option. Modifications to metadata can include, for example, extending the size of a dataset.

U.3) Collective read/write to a dataset, where all processes that opened the file are required to call the function with or without data to read/write.

U.4) 2 or more processes creating an attribute on a dataset, where each of those datasets are only accessed by one process (so the dataset is “owned” by the process). In the current implementation, similar to use case U.1, the attribute creation function needs to be collective, even though they are being created on different datasets. The performance overhead is quite clear as we scale to a larger numbers of attributes and processes.

U.5) 2 or more processes creating an attribute on a shared dataset. A solution that makes this operation independent has to take into account the fact that there are other processes that can create an attribute (or any other object) at the same time and those objects might overlap in the file if they are not serialized.

U.6) Access to compressed datasets by processes (individually or collectively).

We analyze some different applications that are considered very common HPC applications that use parallel HDF5:

A.1) Flash I/O:

Flash I/O is a benchmark that mimics the I/O behavior of the FLASH application (<http://flash.uchicago.edu>). The applications I/O behavior can be described as follows:

* Nothing process specific: All objects are shared between all processes, and the number of HDF5 objects is the same with all process counts
* Very flat hierarchy, all datasets are under the root group
* Collective and Independent I/O can be used.
* Three files are created

A.2) *Others TBD*

# Space Allocation at EOA

In this section, we discuss methods of allowing independent space allocation between all processes.

Currently, space within the HDF5 file is allocated either from a free list (of previously used and released blocks within the file), or from the end of the file’s allocated space (EOA). Recycling space within the file is complex as well as unusual in parallel HDF5 applications, and therefore will be disabled, requiring all space allocation to be performed at the file’s EOA.

Allocating space at the file’s EOA is very simple in serial HDF5 applications: the EOA value begins at offset 0 in the file and when space is required, the EOA value is incremented by the size of the block requested. However, as mentioned earlier, space allocation using the EOA value in parallel HDF5 applications can result in a race condition if processes do not synchronize with each other, causing multiple processes to believe that they are the sole owner of a range of bytes within the HDF5 file.

The simplest solution to this synchronization issue is to use a globally visible atomic “fetch and add” (F&A) operation that guarantees each process can get the current EOA value and increment it by the size of the block that the process needs to allocate, all in an atomic operation. The MPI-2 standard does not define such an operation; however it does provide one-sided communication operations in the “Remote Memory Access” (RMA) section of the standard. Using MPI RMA routines, an F&A operation was implemented by Dr. William Gropp. The algorithm is detailed in his book, *Using MPI-2.*

To allocate space at the EOA using this operation, we do the following:

1. Create an MPI window on all processes with the root process exposing a memory buffer where the fetch and add will take place
2. All processes wanting to increment the EOA will need to lock the MPI window, increment its counter in the root’s memory, get the counter values for all other processes, and unlock the window.
3. Adding the counters for all processes to the requested space would yield the current EOA and requested space allocated.

The second solution is to set aside a process that will act as an EOA server. Using MPI point-to-point operations, the EOA server would keep listening to requests from all clients for space allocation. Once it receives a space allocation request, it updates the EOA and sends the updated the value to the client. The listening part can be implemented in two ways:

1. Using MPI\_Recv with MPI\_ANY\_SOURCE for the destination rank, where the server would listen for a space allocation request from clients, allocate space, and then send the allocated space information to the requesting client.
2. The server would post *n* MPI\_IRecv calls to keep listening to the clients. It then would check for requests that have arrived, update the current EOA value, and send the allocated space information to the requesting client.

We have measured the performance of all algorithms on Hopper[6], a Cray XE6 Linux cluster, using a micro benchmark, where each process requests space allocation 1000 times. The performance was measured when all processes are requesting space allocation at the same time (column 2 below), half of them at the same time (column 3 below), and one at a time (column 4 below). Performing this test showed the effect of different workloads when all the space allocation happens at one set-aside process. The results below show the performance in seconds for all 1000 operations. Column 1 indicates what algorithm is used (RMA = Gropp fetch & add, AS = MPI point-to-point with MPI\_ANY\_SOURCE, NR = MPI point-to-point using *n* IRecvs) and the number of processes the benchmark is executed with.

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm / num processes | All | Half | One |
| RMA 128 | 1.346992 | 0.613762 | 0.018283 |
| AS 128 | 0.223275 | 0.100825 | 0.005886 |
| NR 128 | 0.272892 | 0.231382 | 0.009668 |
| RMA 256 | 2.847602 | 1.4152 | 0.016407 |
| AS 256 | 0.528838 | 0.244721 | 0.005698 |
| NR 256 | 0.664842 | 0.648287 | 0.010718 |
| RMA 512 | 5.78936 | 2.84135 | 0.016669 |
| AS 512 | 1.291383 | 0.489296 | 0.006445 |
| NR 512 | 1.504239 | 1.906703 | 0.017408 |
| RMA 1024 | 16.555731 | 7.286146 | 0.023410 |
| AS 1024 | 2.78822 | 1.388011 | 0.009032 |
| NR 1024 | 3.854202 | 5.261063 | 0.039198 |
| RMA 5120 | 253.062082 | 60.183123 | 0.028895 |
| AS 5120 | 21.290575 | 9.169249 | 0.016118 |
| NR 5120 | 29.083971 | 35.961243 | 0.674165 |
| RMA 10240 | 1923.985398 | 208.543557 | 0.029808 |
| AS 10240 | 38.191935 | 19.414407 | 0.016055 |
| NR 10240 | 68.936267 | 151.279859 | 0.720665 |

As expected, when all processes are requesting space allocation at the same time, the performance does not scale well, because it’s still an all-to-1 communication. When the workload is lighter, the problem scales well, which is expected to be the case for HPC applications we are targeting.

The performance results also indicate that using point-to-point with MPI\_ANY\_SOURCE is the obvious choice that needs to be used between the three choices, however the reason that an RMA fetch and add was explored is that it takes away the explicit involvement of the server in space allocations, which could be better in cases when the server is not only responsible for handling space allocation, but also metadata synchronizations as will be explained in the next section. If a dedicated EOA server is available, then using point-to-point makes more sense as RMA implementations at the moment are very inefficient. In addition, using point-to-point allows exploration of different algorithms once we go beyond a single server, to handle scalability issues.

Some of the disadvantages of this approach are:

* Needs to set-aside a process (i.e. dedicate it to being available for MPI operations), or else that process will not be able to promptly respond to the space allocation requests.
* There are performance issues when a large number of processes try to request space allocation at the same time repeatedly, however we don’t see that as a common case.

# Independent Metadata Operations

This section presents the three classes of designs that have been studied and selected from a larger set of options as candidates for breaking the collective requirement for metadata operations.

## Distributed Lock Manager

A distributed lock manager (DLM) is a commonly used approach to handle access to shared resources. Lustre, for example, uses this approach to handle client requests for locking, disk allocation, storage and retrieval, all through its Object Storage Servers.

In our solution, we can use a DLM to serialize access to HDF5 objects. The DLM is a set-aside process, carved off from the application processes that opened the file and does not perform file I/O itself. All operations that read or modify metadata for an object need to acquire a lock on that object from the DLM.

The DLM can be implemented in several ways:

* MPI point-to-point communication & RMA:

Using the algorithm described in [3], processes acquire and release locks for a certain HDF5 object using a combination of RMA and point-to-point operations. A root process will hold, for each object to be locked, an n-bit value that other processes access using RMA operations to turn their corresponding bit on, asking for the lock, and check the other bits of other processes. If the lock is available, meaning all the bits of other processes are off, the process can assume it has been granted the lock; otherwise it posts an MPI\_Recv from the process that currently has the lock. When the process owning the lock is done with it, it turns off its bit at the root node and checks for the next process asking for the lock, then sends the lock to that process using MPI\_Send. For fairness issues, the lock is granted to the process with the next highest rank than the process currently holding the lock. This approach eliminates polling for the lock remotely and locally at memory regions, and just blocks on an MPI\_Recv.

This algorithm handles only exclusive locks, but it can be extended to handle read locks, using the same idea as an algorithm proposed in [7]. They do use compare & swap operations, which are not available in MPI-2, but I’m pretty sure we can come up with a similar algorithm to grant shared locks.

* Using non-blocking MPI point-to-point operations, where the DLM posts *n* MPI\_IRecv and calls MPI\_Testsome to check all the requests that have arrived. It then queues up those requests on their corresponding objects and process the queues, granting locks and posting another set of MPI\_IRecv to get the locks back.

In both approaches, at some point the single DLM server will become overwhelmed with requests from clients and there will be a need to use more than one process to act as a DLM and actually distribute the lock management of HDF5 objects. However, distributing the lock manager still does not solve the problem of how to handle “hot” objects that are being handled by only one of those servers. Another issue to note is that when a process gets a lock, it’s also getting a set of metadata entries that it can read/write, and sending the changes (in case of a write lock) back. So the interprocess communication is more complex than it appears.

The amount for messages sent and received with the DLM is fairly large. If we consider use case U.1 (or U.5, which is similar), each process must acquire a lock on the root group before being able to create its own dataset. This can get ugly once we have a very large number of processes. In use case U.4, this problem is better handled, because each process is getting a lock on its own dataset, so the attribute creations are done in parallel, but there are still a large amount of message traffic between the application processes and the DLM.

## Synchronization Step

In this approach, each process modifies its metadata cache independently and logs the changes locally. Whenever a collective dataset I/O operation, or an explicit flush/sync operation, is issued by the application, the processes enter a synchronization point, to exchange changes to the file and resolve any conflicts that appear in the logs. This approach is targeted at applications that guarantee that the HDF use of metadata is simple enough that the synchronization step does not introduce unresolvable conflicts. An example of conflicting operations would be something similar to use case U.2, where the process that is writing to a dataset might be writing at obsolete locations if there is no synchronization step between the time that this process issues the writes and another process modifies the metadata of the dataset. A drawback to this approach is that the metadata operations are not entirely independent, since a synchronization step is still (eventually) required. However, no set-aside processes are needed.

## Metadata Server

In the DLM approach, we considered using a lock manager to coordinate access to the metadata of an object. The lock manager would not open the file. Instead, the client will perform all the I/O to disk after coordinating with the lock manager. A different approach would be to have a metadata server (MDS) actually open the file and read/write metadata to the file.

The MDS would perform all metadata operations requested by a client and coordinate changes between clients’ requests internally. After making a request, a client would wait for the MDS to reply with the results from their request, instead of sending the metadata entries directly. For example, if a process needs to create a dataset, it will send the request with all the required parameters to the MDS, and it will get back a dataset identifier, along with associated metadata for that dataset to avoid unnecessary interprocess communication in the future. All synchronization issues are handled at the server. If we consider use case U.1, where n processes are creating *n* datasets in 1 group object, the MDS will receive *n* requests and reply back to each process with its own dataset. Using the DLM approach as we saw earlier, each process would need to get a lock on the group object, create the dataset, and release the lock. The interprocess communication in the DLM case will be much greater.

If we consider use case U.2, we can see that for the MDS approach to work we still need locking, because when a process is doing I/O to disk (raw data), this is not handled by the MDS. Thus, prior to executing a raw data read/write to an object a process must acquire a read lock from the MDS. The MDS is responsible for managing locks similarly to what the DLM does, but the number of locks and the amount of information communicated should be much lower. In use case U.6, where the dataset being accessed is compressed, a client might also need a write lock when it wants to access that dataset.

When a collective dataset access is performed (use case U.3), all processes need to inform the MDS that its participation is required. This is required because the MDS is part of the communicator that opened the file and has to participate in the operation with a 0 byte contribution. An alternative to this requirement is presented in section 4.3.2.

Having a single MDS is obviously not scalable. Having more than one MDS and spreading objects between them would alleviate the problem, but would not be a solution for “hot” objects. Some pieces of metadata, such as the root object, are intrinsically hot and little can be done within this design approach to eliminate that problem. To reduce the cost of access to hot objects, we can apply a distributed hash table to spread the objects among the MDSs with replication. We believe C-MPI[2] can be useful in this case, but it still would require a lot of work that may not be worth pursuing in the future if we consider moving towards the Data Staging approach described later, where the MDS design is still used, however only within the staging processes. Another alternative is multithreading within the MDS node. Given the current explosion of cores, allocating an entire node to act as an MDS is an option, where multiple threads handle requests from other processes. Again, this only alleviates the problem but does not eliminate it.

Clients accessing metadata for an object can still cache metadata and bypass the MDS in some cases when the file is opened read-only. The MDS could also maintain version information about the state of each object, where the client checks if the local version in its cache is the updated version or not, avoiding some communication overhead. Access to non-mutable objects would also bypass the MDS and would not require any locking.

Overall this approach looks the most attractive for the metadata problem, however given the architecture of the HDF5 library, the amount of work required to implement an MDS approach is fairly substantial.

### Enhanced MDS

Another approach to consider is using an enhanced version of the MDS. Objects in HDF5 will be marked as private or public. Public objects are those that are allowed to be accessed by all the processes that opened the file. The root group in HDF5, for example, should be a public object. A private object can be, for example, a dataset created as in use case U.1 for a process’ private use, where only this process will access that dataset. Access to metadata of public objects would go through the MDS, whereas access to metadata of private objects could be done locally. If a private object needs to be made public, the process that holds it would transfer the “ownership” to the MDS, including pushing all the metadata to the MDS or writing the metadata to disk and informing the MDS about it. A public object that needs to be made private by a process would require the process to send the request to the MDS asking to change the status of an object to private. When the MDS clears all locks on that object, the request to make the object private can be granted, and the MDS can release all ownership (metadata, cache items, lock queues, etc…) for that object.

### Two Files

The MDS, as proposed handles only metadata I/O. An alternative to having the MDS writing metadata and clients writing raw data to the same file is to have two separate files, one for metadata and one for raw data. The MDS does not open the raw data file, and hence collective I/O done by the clients (U.3) does not need the participation of the MDS in the collective function call. This would save a good amount of overhead from having all clients inform the MDS that it needs to participate in a collective I/O call. When multiple servers are needed, this overhead increases considerably as all servers need to be notified.

Another benefit from having two separate files is that the metadata file access could be handled differently than raw data file access, since the metadata file is considerably smaller than the raw data file. File systems usually offer different optimizations to access small vs. large files.

HDF5 has a split-file driver already implemented as a VFD layer, where metadata and raw data are split to separate files; however this VFD is implemented only for serial I/O. Modifying it to work with MPI-I/O should not be a hard task. When that is done, it would be straightforward to use that VFD with the original MDS design to achieve the two file approach for metadata and raw data.

The downside to this approach is that the metadata and raw data would be in two files that are inseparable which is extra book keeping for the user. Merging the two files together when the files are closed and splitting them apart when opened is one technique to remedy this issue, however it’s something for future consideration.

### Data Staging

The MDS approach delegates handling metadata operations to a metadata server. A wider extension to that approach would be Data Staging, which delegates all I/O operations to server processes. ADIOS [1], another high level I/O library, uses data staging for all I/O operations, where a set of processes is set aside to handle the I/O requests for application processes. While the purpose of this RFC is to break the collective requirement of metadata operations, it is worthwhile to explore a data staging option to achieve that, in addition to a much wider set of functionalities. Asynchronous independent metadata operations would be introduced, since the data staging nodes will handle all the consistency issues. Application processes would just post their I/O requests and return to computation immediately. Similar to other non-blocking operations in MPI, a mechanism to check on outstanding requests must be in place. The metadata problem still exists within the staging nodes that need to coordinate access to the metadata. Considering that the number of staging nodes is considerably smaller than the number of compute nodes, this problem becomes easier to handle using the MDS approach outlined earlier.

In order to handle the contention on network resources between the application communication and the background data transfer, a communication protocol would be used to block I/O data from being sent to staging nodes when application communication is occurring.

Some limitations to this approach include:

* set-aside process(es)
* scalability issues when not enough data staging processes are available
* the implementation is very complex in terms of API changes, functionality, etc…

# Performance

A simple prototype for the MDS and DLM is implemented to show the communication performance in comparison with the original approach where everything is done collectively. The current algorithm is prototyped by an MPI\_Barrier(), followed by 2 MPI\_Bcast() operations from process 0 to all the processes, then another MPI\_Barrier(). The broadcast operations represent the step when process 0 pushes all dirty metadata cache entries that need to be flushed to all processes so they can be marked as clean. Since all processes see the same stream of dirty metadata, synchronization is guaranteed by having only process 0 control the flush of metadata to disk. The MDS and DLM are prototyped to simulate as closely as possible the communication protocol that would be used. The performance tests are conducted on Hopper and the results shown are in seconds for a simulation doing 1, 2, and 100 operations. Note that for the original approach the number of operations would be multiplied by the number of processes to simulate a comparable behavior.

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator | 1 Op. | 2 Op. | 100 Op. |
| ORIG 128 | 0.002701 | 0.003401 | 0.035983 |
| MDS 128 | 0.003528 | 0.003715 | 0.029973 |
| DLM 128 | 0.03801 | 0.054422 | 2.441472 |
| ORIG 256 | 0.003089 | 0.004543 | 0.079608 |
| MDS 256 | 0.006303 | 0.00638 | 0.066449 |
| DLM 256 | 0.095199 | 0.151447 | 4.64511 |
| ORIG 512 | 0.003713 | 0.006698 | 0.175682 |
| MDS 512 | 0.01136 | 0.012178 | 0.147917 |
| DLM 512 | 0.246199 | 0.451026 | 9.407203 |
| ORIG 1024 | 0.004746 | 0.012626 | 0.458339 |
| MDS 1024 | 0.053389 | 0.058904 | 0.372549 |
| DLM 1024 | 0.694647 | 0.907409 | 20.436834 |
| ORIG 5120 | 0.014127 | 0.092728 | 5.431242 |
| MDS 5120 | 0.287151 | 0.314072 | 3.572677 |
| DLM 5120 | 4.484003 | 6.557662 | 145.138304 |
| ORIG 10240 | 0.110606 | 0.215091 | 8.376475 |
| MDS 10240 | 1.183902 | 1.206859 | 6.794207 |
| DLM 10240 | 14.448289 | 19.326462 | 368.379117 |

While we can see the performance overhead for communication, this doesn’t really qualify as good grounds for comparison, because we are not really measuring the I/O overhead from the operations themselves. In addition to that, the point behind the MDS and the DLM approaches is to provide a new programming model that is not currently possible. However, looking at just the communication overhead, the DLM seems to be bad compared to the others since we are assuming in the test case that all processes are acquiring exclusive locks on one object where the operation will be performed on. The MDS overhead seems to be comparable and sometimes better than the original, when the number of operations needed is large, which is likely the case in HPC applications.

A different benchmark that actually calls HDF5 operations that manipulate metadata has also been tested. This time we measure how the original approach compares against an MDS approach, where both need to create the same file structure specified by the use case. The MDS is simulated by having one process open the file serially and listen to requests from other processes. The other processes aren’t touching the file in any way, but just requesting the MDS to execute the operations they need. The MDS would then perform those operations serially to the file it opened.

Two test cases are implemented using the two approaches. The first one creates a group for each process in the file’s root group, and then creates two more subgroups under each group created. Finally under each subgroup, three datasets are created. The structure can be better understood looking at the following tree:



The second test case is a combination of U.1 and U.4, where each process wants to create a dataset and attach an attribute to that dataset.

The results are displayed in the following table and show the MDS performance benefits where there is no need to call an object creation function *n* times. Each process is requesting its objects independently from all other processes. The platform is Hopper and the results are in seconds.

|  |  |  |
| --- | --- | --- |
|  |  Test Case 1 | Test Case 2 |
| ORIG 128 | 0.19493 | 0.294393 |
| MDS 128 | 0.05169 | 0.126092 |
| ORIG 256 | 0.551161 | 5.948483 |
| MDS 256 | 0.213912 | 0.133184 |
| ORIG 512 | 0.805472 | 0.663113 |
| MDS 512 | 0.280865 | 0.153351 |
| ORIG 1024 | 2.780002 | 0.804936 |
| MDS 1024 | 0.680481 | 0.188209 |
| ORIG 5120 | 43.3329 | 6.306099 |
| MDS 5120 | 5.914479 | 0.755397 |
| ORIG 10240 | 118.918898 | 30.539587 |
| MDS 10240 | 17.398757 | 2.15023 |

# Progress Function

The MDS approach (and its variants) requires a set-aside process(es) to accept requests from clients to access metadata. A different approach to consider is layering, on top of HDF5, a library, where the user is responsible to call progress functions in his application to synchronize HDF5 metadata changes. This can be applied in two ways:

* The progress function mimics the collective behavior of how metadata operations are performed currently, where all processes will come together and coordinate access to metadata. This will provide the non-collective functionality for metadata operations, but suffers from the overhead that all processes have to join with the progress function, to be able to proceed.
* A process can be designated as an MDS to an object. When clients would like to access metadata to that object they send requests to its MDS (similarly to the MDS design). When a designated MDS process enters a progress function, it receives all the requests, processes them, and then goes back to the application. The enhanced MDS approach can be applied here too, where we only coordinate access to objects that are marked global.

# Recommendation

As mentioned earlier, this work is still at the design phase. While we have a tentative roadmap on how to process with this work, things might change depending on various reasons, such as new input from the HPC community, funding, etc… For now, we are looking at proceeding with the EOA server approach for the space allocation problem, and the MDS approach with two files for the metadata operations, while moving later towards Data Staging similar to ADIOS. We would eventually like to provide both options to the user in terms of either setting aside processes or calling progress functions in the application, but will begin with the set-aside process approach.

Revision History

|  |  |
| --- | --- |
| *September 2, 2011:* | Version 1 circulated for comment within The HDF Group.  |

 [References]

[1] ADIOS: <http://www.olcf.ornl.gov/center-projects/adios/>

[2] C-MPI: <http://www.mcs.anl.gov/publications/paper_detail.php?id=1229>

[3] RMA + point-to-point locking <http://www.mcs.anl.gov/~robl/papers/ross_atomic-mpiio.pdf>

[4] DSM 1 <http://nextmuse.cscs.ch/sites/default/files/Eurompi2010_paper_17.pdf>

[5] DSM 2 **Data Redistribution using One-sided Transfers to In-memory HDF5 Files**

**[6]** <http://www.nersc.gov/systems/hopper-cray-xe6/>

**[7]** http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.145.395&rep1&type=pdf