Multi-threading HDF5: Paths Forward

*Quincey Koziol*

# Introduction

The HDF5 library is designed to be adaptable to the changing needs of its user community to stay relevant and useful to them. As we progress further into the multi-core era, developers using the HDF5 library are increasingly writing multi-threaded applications in order to fully utilize the underlying system hardware. These developers are looking for features in the HDF5 library that enable them to leverage the concurrency they are building into this new generation of multi-threaded applications.

Although the HDF5 library is currently threadsafe and able to be used in multi-threaded applications, the internal mechanism used for ensuring this threadsafety doesn’t allow multiple application threads to concurrently use the library. This document describes, at a high-level, the current mechanism for ensuring the HDF5 library’s threadsafety, and described two paths to improving on its limitations.

# Background: Current Implementation

The HDF5 library must use a mechanism to prevent multiple application threads from entering the library and modifying memory and file data structures simultaneously, in an uncontrolled manner. Uncontrolled simultaneous modification of data structures by multiple threads can result in corrupting those data structures and the files produced.

For example, if multiple application threads simultaneously call the HDF5 library and those threads causes the library to operate on an internal data structure, each thread could attempt to modify elements in the data structure. If two or more threads attempt to modify the same element in the structure, a race condition is created where one thread’s change to the element will overwrite another thread’s change, causing the data structure to become corrupted or changes to it to be lost.

In order to prevent this sort of corruption, and similar failures, the HDF5 library uses a single global semaphore to restrict multiple threads from entering the library simultaneously. As an application thread enters an HDF5 API routine, it attempts to acquire this semaphore. The first thread to acquire the semaphore will be allowed to enter the HDF5 library and complete its API call, blocking other threads from entering the library while doing so. When the thread holding the semaphore completes its API call and releases the semaphore, another thread will acquire the semaphore and proceed to enter the HDF5 library to perform its API call.

This single semaphore provides the threadsafety that the HDF5 library needs to avoid corrupting internal data structures, but doesn’t allow application threads to concurrently use the library. Solutions and alternatives to this single semaphore are described in the rest of this document.

# Path #1: Improving Concurrency

To improve the concurrency of application threads within the HDF5 library, the single global semaphore guarding the entire library would be replaced with many smaller, more tightly focused semaphores[[1]](#footnote-1) that guard individual data structures. When this is finished, semaphores will guard individual data structures within the HDF5 library, preventing their corruption, and the global semaphore can be removed. Then, each thread that enters the library will only wait when it needs to modify a data structure that is currently locked by another thread, greatly reducing the time that a thread waits for a resource to become available.[[2]](#footnote-2)

The process of replacing a global semaphore with individual semaphores on each data structure requires careful analysis of each data structure within the code to modify and how those data structures interact with each other, according to the flow of control within the code. This process requires significant effort by knowledgeable staff, and is proportional to the size of code to analyze and modify[[3]](#footnote-3).

Given that the HDF5 library is currently ~300,000 lines of C code, choosing this path to improvement will be a very significant undertaking, probably requiring 4-6 FTE years of work, plus ongoing effort to maintain the threadsafe nature of the data structures, as further modifications are made to the HDF5 library.

# Path #2: Reducing Latency

As an alternative to directly improving the concurrency of the HDF5 library, the length of time each thread waits to acquire the global semaphore could be reduced. Achieving this goal would allow the *appearance* of greater multi-threaded concurrency to the HDF5 library, without requiring the effort involved in replacing the global semaphore with individual semaphores on data structures.

In order to reduce the latency of operations in the HDF5 library, the library will need to be analyzed for performance bottlenecks, and those bottlenecks removed. Given our experience and current knowledge, we anticipate that those bottlenecks will fall into three categories:

* Time spent performing I/O
* Time spent performing “compute bound” operations that depend on the I/O operations, such as datatype conversions and applying filters to chunks (e.g. when compressing data)
* General overhead

Addressing each of those bottleneck categories will require a different technique.

Time spent performing I/O can be addressed with the use of asynchronous I/O operations to access data in the HDF5 file. Asynchronous I/O operations can be initiated within the library in response to an API call, and can complete in the background, after the API call has returned (and the global semaphore has been released).

Time spent performing compute bound operations within the HDF5 library, such as performing datatype conversions and compression, can be addressed by using multiple threads *within* the HDF5 library. Each compute-bound routine can be replaced with a new implementation that uses multiple threads to accelerate the operation[[4]](#footnote-4), reducing the amount of time that the calling thread holds the global semaphore.

Time spent in general overhead is the most difficult to address, as it will require optimizing the existing code in a traditional way, with algorithm or data structure improvements. However, reductions in the general overhead will reduce the time a thread holds the global semaphore, as well as improve the overall performance of the HDF5 library, for both single- and multi-threaded applications.

The effort required to address time spent performing I/O and time spent performing compute-bound operations is much smaller than the amount of time needed to remove the global semaphore in the library, probably on the order of 6-8 months of FTE effort for each aspect (i.e. perhaps 1.5 years of effort, total). An additional benefit of these improvements is that they are localized within the library and don’t require maintenance when future modifications are made to the HDF5 library, unless those modifications are related to I/O- or compute-bound aspects of the library. Optimizing general overhead is best done on an “as needed” basis, as it is essentially an unending task, as the library grows and is modified.

# Conclusions and Recommendation

Although the process of converting from a single global semaphore to individual semaphores (path #1) will give multi-threaded application a greater level of concurrency within the HDF5 library, the effort involved in performing this conversion, as well as the ongoing maintenance required to support it, argue against going down this path unless there is significant, and long-term, funding available to maintain the feature. Implementing support for asynchronous I/O and internally multi-threading compute-bound aspects of the library (path #2) will bring a large portion of the benefits of the semaphore conversion path, but will be much less expensive to implement and maintain.

Implementing a solution based on path #2 is the recommended path of action.[[5]](#footnote-5)

1. And/or other thread-safety structures, such as mutexs, locks, etc. [↑](#footnote-ref-1)
2. One possible (although incomplete and somewhat dated) description and implementation notes for this path is documented here: <http://www.hdfgroup.uiuc.edu/papers/features/mthdf/MTHDFpaper.htm> [↑](#footnote-ref-2)
3. This effort may actually be proportional to the square of the size of the code, due to the combinatoric interaction of internal data structures. [↑](#footnote-ref-3)
4. Either using OpenMP or by spawning new threads explicitly. [↑](#footnote-ref-4)
5. Also, the two paths are not mutually exclusive - they could both be implemented, in either order, and coexist in the long-term with enough funding. [↑](#footnote-ref-5)