Pattern-Aware File Reorganization in MPI-IO

Jun He, Huaiming Song, Xian-He Sun, Yanlong Yin
Computer Science Department
Illinois Institute of Technology
Chicago, Illinois 60616
{*jhe24, huaiming.song, sun, yyin2*}*@iit.edu*

Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439
thakur@mcs.anl.gov

***Abstract*—Scientific computing is becoming more data-intensive; however I/O throughput is not growing at the same rate. MPI-IO and parallel file systems are expected to help bridge the gap by increasing data access parallelism. Compared to traditional I/O systems, some factors are more important in parallel I/O system in order to achieve better performance, such as the number of requests and contiguousness of accesses. The variation of these factors can lead to significant differences in performance. Programmers usually arrange data in a logical fashion for ease of programming and data manipulation; however, this may not be ideal for parallel I/O systems. Without taking into account the organization of file and behavior of the I/O system, the performance may be badly degraded. In this paper, a novel method of reorganizing files in I/O middleware level is proposed, which takes into account the access patterns. By placing data in a way favoring the parallel I/O system, gains of up to two orders of magnitudes in reading and up to one order of magnitude in writing were observed with spinning disks and solid-state disks.**

***Keywords*-MPI-IO; PVFS2; parallel I/O; file reorganization;**

I. Introduction

In the past decades, growth in computing capabilities has enabled scientists and engineers to process larger and more complex problems. At the same time, the scientific computing tends to be overwhelmed with data captured by instruments and generated by simulations. The increasing gap between computing power and I/O speed has become an imminent problem for the community. Numerous efforts have been made to bridge the gap. MPI-IO provides a uniform interface to developers for accessing data in a parallel environment without dealing with the underlying file systems. In addition, parallel file systems like PVFS, GPFS, Lustre, and PanFS have enabled parallel access to file systems, thereby increasing the throughput of file reads and writes.

The contiguousness of the I/O requests have an important impact on the performance of parallel file systems. Many HPC applications involve non-contiguous small requests [1] [2] [3]. Large numbers of these small requests leads to network congestion which then degrades the I/O performance significantly. Additionally, non-contiguous accesses lead to more disk seeks than contiguous ones and hurt performance of HDD-based parallel file systems, because the IOPS (I/O Operations Per Second) of HDDs can easily become the bottleneck of a file system. Moreover, these non-contiguous accesses break data locality, since two successive accesses may be very far from each other. In that case, many optimizations taking advantage of data locality may not take effect. In short, contiguous I/O accesses have much higher throughput in parallel file systems due to less overhead in the network, file servers and better data locality. Figure 1 shows a typical parallel file system.

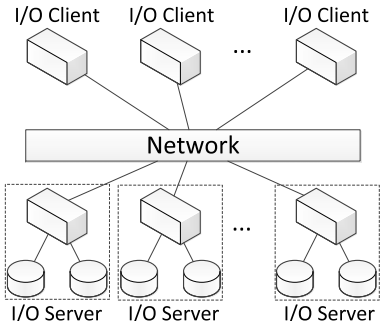

Figure 1. A typical parallel file system architecture. I/O clients send data requests over the network. Then the I/O servers access the local disks and respond to the requests. The data is striped over multiple servers to enhance the parallelism.

In this paper, we introduce a pattern-aware file reorganization approach, which significantly increases the contiguousness by remapping files in MPI-IO layer. The proposed approach makes a better integration of access patterns and underlying parallel file systems. It can be applied to applications in which noncontiguous access patterns are repeated. For example, the I/O access pattern of application start-up is usually fixed [4]. Checkpointing [1] also involves lots of these patterns. It can also be used at data analysis in which files with same format but different contents are read. Our approach works no matter whether the pattern is regular, i.e. n-d strided, or not. But it cannot improve performance of applications without any repeat patterns, since no applicable patterns can be found.

The main contributions of this paper are as follows.

1) We propose a novel method of bridging the mismatch between logical data and physical data for better I/O

performance.

2) We propose a novel I/O-signature-based remapping table, which has fewer entries than a traditional one-to-one remapping table and thus has smaller size and shorter lookup time.
3) Extensive experiments have been carried out to evaluate the potential of the proposed strategy. The experimental results show that our method improves the performance by up to two orders of magnitudes in reading and up to one order of magnitude in writing.

The rest of this paper is organized as follows. In Section II, the design of the pattern-aware system is introduced in detail. Section III evaluates the performance of data accesses after file reorganization. Section IV briefly reviews I/O optimizations related to MPI-IO, parallel file systems, and file organizations. Section V concludes the paper.

II. Pattern-Aware File Reorganization

A. System Overview

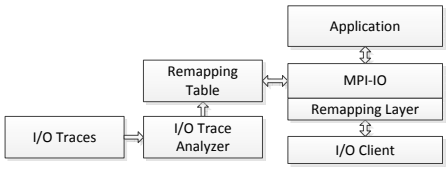

Figure 2. System overview

Figure 2 illustrates the architecture of the pattern-aware file reorganization system. Our system takes as input the I/O traces collected from the previous runs of the application. The analyzer then analyzes the I/O traces of the application to find any non-contiguous access patterns to which file reorganizing can be applied. After that, a remapping table is built based on the analysis results. It is used by MPI-IO for future file accesses, to translate old offsets to new offsets in the reorganized files.

B. Trace Collecting

File reorganizing is based on the information extracted from previous runs of the application. In order to predict the future access patterns and reorganize the file, it is required to collect the data access history of the application. For this purpose, we implemented an I/O tracing tool which wraps MPI-IO calls and records all the high-level information needed. After running the applications with the trace collecting enabled, we get process ID, MPI rank, file descriptor, type of operation (open, close, seek, read or write), offset, length, data type, time stamp, and file view information.

C. Pattern Classification

Access patterns can be classified in terms of five factors as shown in Figure 3 [5]. The most important factor is spatial pattern. In contiguous patterns, two successive accesses have no hole between them. If it is strided, disk head has to be moved over a hole to perform the next access.

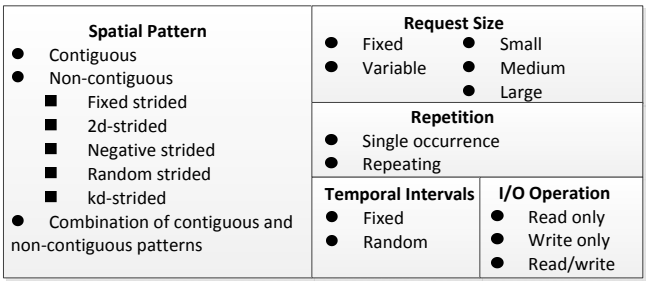

Figure 3. Classification of access patterns

Table I
Remapping table

OLD: File path, MPI_READ, offset0, 1, ([(hole size, 1), LEN, 1]), 4	**NEW**: Offset0'

D. I/O Trace Analyzer

The analyzer is used to identify I/O patterns as shown in Figure 3. It is an offline process that takes the I/O traces as input. First, the analyzer separates the trace entries by MPI rank. Then, it goes through the I/O traces chronologically and attempts to recognize the patterns described in Figure 3.

By using the five factors in Figure 3, we can describe an access pattern in a notation as follows, which is termed as I/O signature.

{*I/O operation, initial position, dimension, ([*{*offset pattern*}*,* {*request size pattern*}*,* {*pattern of number of repetitions*}*,* {*temporal pattern*}*], [...]), # of repetitions*}

For example, notation {MPI_READ, 4194304, 1, ([(2097152, 1), 1048576, 1]), 98} means an application reads data starting from offset 4194304 in a fixed-strided (1-d) manner. It reads 1048576 bytes of data for 98 times and there is a $2097152 \times 1 - 1048576$ byte hole between two successive reads.

E. Remapping Table

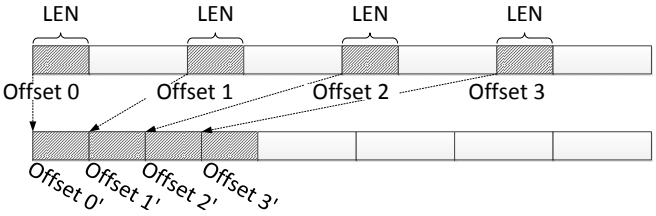

Figure 4. Remapping of 1-d strided accesses. After remapping, there is no hole between any two successive accesses.

If non-contiguous accesses are found, they are reorganized to be contiguous. For example, given a fixed strided pattern as shown in Figure 4, the data is reorganized by the remapping table as shown in Table 1. I/O signature notation is used in the remapping table instead of one-to-one offset remapping. The advantage of this approach is that it has only one entry per pattern. Comparatively in a one-to-one remapping, the number of entries is determined by the number of accesses. Our approach turns out to be very efficient and has negligible overhead.

In data processing applications, it is common that the data accessed by one process resides in multiple files. This kind of data is non-contiguous; however, by reorganizing, the data can be put together in the sequence of accesses. Thereby, requests can be merged and the spatial locality can be improved.

When one or more chunks of data is accessed by different patterns, they are assigned to one pattern that has accesses to most of the chunks. So we can get at least one contiguous access, while the other accesses are kept as contiguous as possible.

F. MPI-IO Remapping Layer

The application is executed with its remapping table loaded to memory. When the application issues an I/O request, the remapping layer captures it and finds the corresponding entry in the remapping table. Assuming that the application issues a read of m bytes data at offset f, we can use the formulas below to check whether this access falls in a 1-d strided with starting offset off, request size rsz, hole size hsz, and number of accesses of this pattern n.

$$(f - off)/(rsz + hsz) < n \quad (1)$$

$$(f - off)\%(rsz + hsz) = 0 \quad (2)$$

$$m = rsz \quad (3)$$

If a request satisfies (1), (2) and (3), this access is in the 1-d strided pattern. If we know, off, the offset of the first read of this sequence of 1-d strided accesses, then the corresponding new offset of f is $off + rsz * (f - off)/(rsz + hsz)$.

These formulas can be extended to handle other non-contiguous patterns. If not all the data of one access can be found in one entry of the remapping table, it can be split and fulfilled by different entries.

III. Evaluation

We conducted our experiments on a 64-node cluster. These nodes are Sun Fire X2200 servers with dual 2.3GHz Opteron quad-core processors, 8G memory, 250GB 7200RPM SATA hard drive and 100GB PCI-E OCZ Revodrive X2 SSD (read: up to 740 MB/s, write: up to 690 MB/s). The nodes are connected by both Ethernet and Infiniband.

Table II
1-D STRIDED REMAPPING TABLE PERFORMANCE (1,000,000 ACCESSES)

Table Type	Size (bytes)	Building Time (sec)	Time of 1,000,000 Lookups (sec)
1-to-1	64000000	0.780287	0.489902
I/O Signature	28	0.000000269	0.024771

The operating system is Ubuntu 9.04 (Linux kernel 2.6.28-11-server). We use MPICH2 1.3.1 and PVFS2 2.8.1 in our experiments. The stripe size of PVFS2 is 64KB.

A. Remapping

We compared our I/O-signature remapping table with the traditional one-to-one table and found that our remapping approach takes less time and much less memory space. The overhead introduced by our remapping table is negligible.

The experimental results are presented in Table II. The I/O signature table occupies much less space than the 1-to-1 design since it has only the pattern abstracts. In terms of lookup time, the 1-to-1 remapping table was implemented as hash table. The complexities of looking up an entry for both tables are close to $O(1)$. But the I/O signature table takes even less time.

B. Pattern Variations

We evaluate some variations of the access patterns to demonstrate how the variations may impact performance. Figure 5 and Figure 6 show the 1-d strided performance with variations of request sizes and start offsets, running on one I/O client and four I/O servers. In Figure 5, we can observe that for reading the performance goes down very slowly in most cases, and for writing the performance drops when holes appear and then stays stable. In Figure 6, we can observe that the variation of start offset does not degrade the performance.

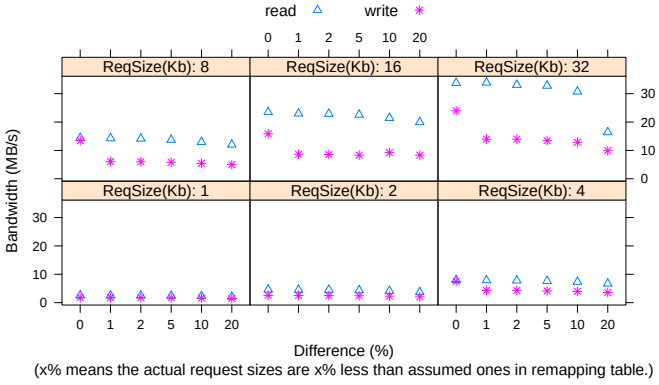

Figure 5. Impact of Request Size Variation

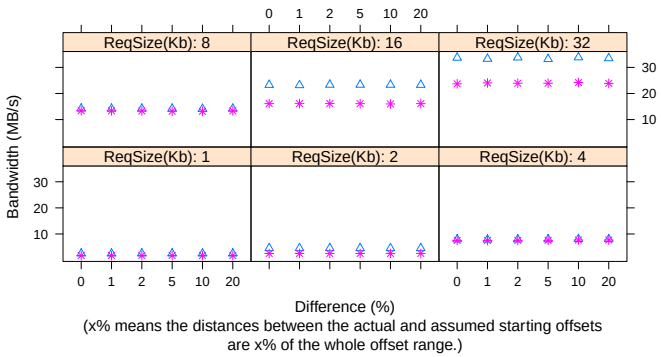

Figure 6. Impact of Start Offset Variation

C. IOR Performance

IOR [6] is a benchmark that can be configured to test various patterns including non-contiguous accesses. We ran IOR with non-contiguous patterns and then re-ran it with reorganization. There were 4 I/O clients and 4 I/O servers of PVFS2.

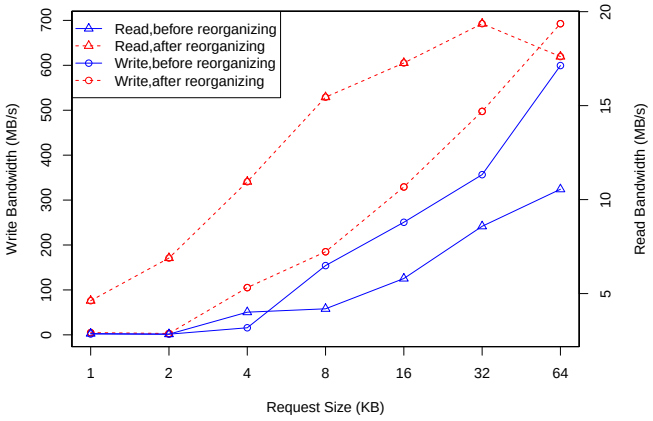

Figure 7. IOR read/write bandwidth of 64 processes, with HDD and Infiniband.

From Figure 7, great bandwidth improvement in reading and writing can be observed after reorganizing. The improvement is mainly resulted from that the disk seek times of contiguous accesses were shorter than those of non-contiguous accesses. In addition, after reorganizing, many other optimizations for contiguous accesses can take effect to further improve the performance.

D. MPI-TILE-IO Performance

Mpi-tile-io [7] is a widely used benchmark designed to test the performance of MPI-IO with non-contiguous accesses. It simulates the very common data accesses of matrix. In mpi-tile-io, the data set is divided into 2-D tiles by MPI_Type_create_subarray(). Each of the tiles is accessed by one process. In our experiment, the number of elements in a tile is fixed at 1024×1024. PVFS2 was also configured to have 4 I/O clients and 4 I/O servers.

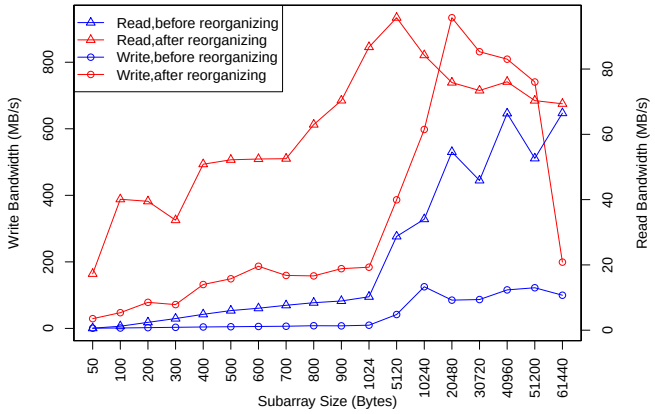

Figure 8. MPI-TILE-IO read/write bandwidth of 64 processes, with HDD and Infiniband.

From Figure 8, great improvement of reorganizing over original non-contiguous accesses can be observed, especially when the sizes of subarrays are small. The improvement is due to the number of requests being lower and sizes of requests are larger after reorganizing the data. Before reorganizing, small non-contiguous reads lead to longer network communication times and more I/O operations on the disks.

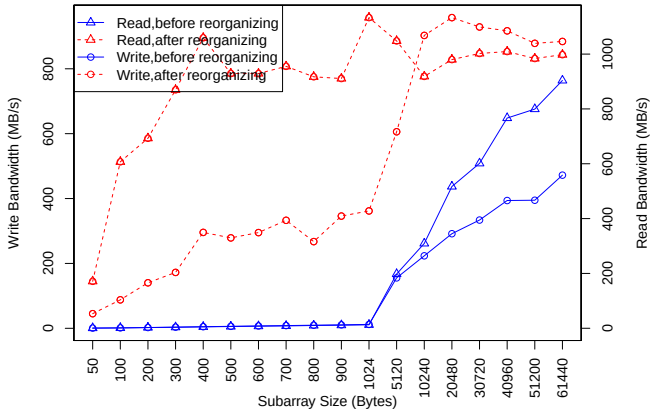

Figure 9. MPI-TILE-IO read/write bandwidth of 64 processes, with SSD and Infiniband.

From Figure 9, it can be observed that, before reorganizing, the performance is poor when the subarray size is small (i.e. request sizes are small and number of requests is large), although an SSD has much higher IOPS and bandwidth than a HDD. A typical commodity 7200 RPM SATA hard drive usually has around 90 IOPS, and 9 ms seek time, while the SSDs used in our experiments have 100,000 IOPS, 0.1 ms seek time and excellent random access performance. However, the performance of a parallel file system does not

only depend on disks, but also network. Before reorganizing, large amounts of requests were sent over the network, which significantly degraded the performance. After the file was reorganized, the parallel file system fully took advantage of the SSDs.

IV. Related Work

Due to the importance of the parallel file system in high performance computing, a variety of efforts have been made to improve their performance. Many techniques have been proposed to handle non-contiguous requests, such as collective I/O [3], data sieving [3], list I/O [8], and datatype I/O [9]. They are used to reduce number of request on network and/or on disks.

Zhang et al. have identified the disk thrashing problem in parallel file systems [10]. In [10], all file servers serve only one process at a time. It can solve the non-contiguous problem caused by different processes, but not the inner access behaviors of each process.

Some optimizations are based on moving data or caching data. Disk shuffling means moving data on disk at run-time. Regarding caching, most frequently accessed data is put together to a reserved part of disk [4], memory [11] or both [12]. Moreover, [13] proposed a new way of partitioning files on RAID. Each I/O process has its file partition. Data chucks frequently accessed by a process are put into the partition associated with that process.

Remapping is used to help processes access data after reorganizing the file in our approach. PLFS [14] also uses remapping to let processes actually access separate files as if all processes share the same file, in order to improve performance of checkpointing. Index files in PLFS are used to record lengths of writes, logical offsets and pointers to physical offsets, which is similar to the remapping table in our approach. But the index file in PLFS is actually an one-to-one remapping table, since it simply appends new record to its end [1][14]. Many file systems are also using similar table to keep track of free blocks on disk [15]. This method suffers from longer lookup time and larger table size as we showed in Section III-A. It is used since the systems are not aware of the patterns as our approach. In addition, PLFS introduces to underlying parallel file system big amount of files, which may lead to metadata problems when it is scaled up. Furthermore, it is designed for checkpointing, which makes it not flexible enough when it comes to other applications.

HDF5 [16] is a set of formats, libraries and tools, which is used to organize and access data. To accelerate accesses, data can be chunked. However, it requires users to manually configure it and it is only good for regular data. In addition, switching to HDF5 requires modifying codes.

Our work is different from all the other work above. Firstly, I/O traces are analyzed to discover access patterns. Then, the application-specific access patterns are exploited by reorganizing the data using remapping at a high level in MPI-IO. Therefore, all the layers under MPI-IO can benefit from the better contiguousness and improved data locality to allow better overall performance.

V. Conclusion

Due to the diverse access patterns of data-intensive applications, physical file organization in parallel file systems usually does not match the logical data accesses. Small and noncontiguous data accesses can lead to large overhead in network transmission and storage transactions. In this paper, a pattern-aware file reorganization approach is proposed to take advantage of application I/O characteristics by leveraging the file remapping layer in the existing MPI-IO library. The proposed approach maintains a data block remapping mechanism between user data accesses and file servers, to turn small and noncontiguous requests into large and contiguous ones. Experimental results demonstrate that this approach shows improvements of up to two orders of magnitudes in reading and up to one order of magnitude in writing. The proposed approach provides an effective integration of data access from I/O clients and data layout on the underlying parallel file systems, thereby being suitable for most data-intensive applications.

In the future, the file reorganization with more complicated access patterns will be explored. Additionally, this reorganization approach can be applied to workload balancing, which may offload hot data to free servers or to faster storage media such as SSDs according to access patterns.

Acknowledgment

The authors are thankful to Hui Jin and Spenser Gilliland of Illinois Institute of Technology, Ce Yu of Tianjin University, and Samuel Lang of Argonne National Laboratory for their constructive and thoughtful suggestions toward this study. This research was supported in part by National Science Foundation under NSF grant CCF-0621435, CCF-0937877 and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] PLFS website. [Online]. Available: http://institutes.lanl.gov/plfs/

[2] P. Crandall, R. Aydt, A. Chien, and D. Reed, "Input/output characteristics of scalable parallel applications," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*. ACM, 1995, p. 59.

[3] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, 1999, p. 182.

[4] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: block-reORGanization for self-optimizing storage systems," in *Proccedings of the 7th conference on File and storage technologies*. USENIX Association, 2009, pp. 183–196.

[5] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 44.

[6] IOR benchmark. [Online]. Available: http://sourceforge.net/projects/ior-sio/

[7] Mpi-tile-io benchmark. [Online]. Available: http://www.mcs.anl.gov/research/projects/pio-benchmark/

[8] A. Ching, A. Choudhary, K. Coloma, and W. Liao, "Noncontiguous I/O accesses through MPI-IO," in *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'2003)*. Citeseer, 2003.

[9] A. Ching, A. Choudhary, and W. Liao, "Efficient Structured Data Access in Parallel File Systems," in *Proceedings of the IEEE International Conference on Cluster Computing*. Citeseer, 2003.

[10] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: improving the performance of multi-node I/O systems via inter-Server coordination," in *Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2010, pp. 1–11.

[11] Linux. [Online]. Available: http://www.kernel.org/

[12] P. Gu, J. Wang, and R. Ross, "Bridging the gap between parallel file systems and local file systems: A case study with PVFS," in *37th International Conference on Parallel Processing*. IEEE, 2008, pp. 554–561.

[13] Y. Wang and D. Kaeli, "Profile-guided I/O partitioning," in *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 2003, pp. 252–260.

[14] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*. ACM, 2009, p. 21.

[15] A. S. Tanenbaum, *Operating Systems Design and Implementation, Third Edition*. Prentice Hall, 2006.

[16] HDF5. [Online]. Available: http://www.hdfgroup.org/HDF5/