# File Composition Technique to Improve the Performance of Accessing a Number of Small Files

**Yoshiyuki Ohno**[1], **Atsushi Hori**[1], **and Yutaka Ishikawa**[1,2]
[1]RIKEN Advanced Institute for Computational Science, Kobe, Hyogo, Japan
[2]The University of Tokyo, Tokyo, Japan

**Abstract**— *One of the scalability issues in parallel applications, in which each process creates each file and writes data to the file, is the scalability of file management due to the increasing number of files. To mitigate this issue, a new file aggregation mechanism, called the file composition technique, is proposed. Unlike existing aggregation mechanisms, the file composition technique aggregates multiple files created by parallel processes into a single shared file without changing the code of file I/O operations. In contrast with the metadata operations in existing aggregation mechanisms, the metadata operations are distributed to each process in order to carry out scalability. The proposed file composition technique is evaluated using a climate simulation code, called SCALE. The result shows that the elapsed time of file output is approximately 30% faster than that of original POSIX I/O functions.*

**Keywords:** File I/O, Parallel file system, File aggregation

## 1. Introduction

I/O access patterns of parallel applications are broadly classified into three patterns: N-1, N-N, and N-M. The N-1 pattern means that all processes access some shared files, and the N-N pattern means that each process accesses some individual files. For example, in a climate simulation program, all processes save their local data to a single file or save each individual file. In the former case, the file I/O access pattern is called N-1 and is called N-N in the later case. Contrasting with applications employing the N-1 or N-N pattern in other parallel applications, such as data mining and processing huge sensor data, all processes following the N-M I/O pattern handle many files whose sizes are not uniform.

I/O access patterns of many parallel applications are categorized as the N-N pattern. For example, it was reported that, in ten projects using the Blue Gene/P at the Argonne National Laboratory [1], four, seven, and two projects employ the N-1, N-N, and N-M patterns, respectively. In the N-N pattern, as the number of processes increases, the number of files grows and the I/O performance becomes the bottleneck of scalability. For example, Figure 1 shows the execution time of parallel processes creating one individual file each. The execution time linearly increases according to the increasing number of processes. This is the result of using a cluster with the Lustre file system described in Section 4.

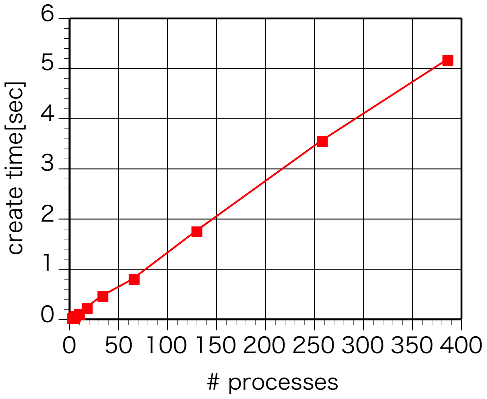


Figure 1: Time it takes parallel processes to create one file each

An approach to mitigating the above problem is introducing a file aggregation mechanism that gathers multiple data generated by an application and stores these data into one or a few files in order to reduce the number of files accessed by these processes. Several libraries carry out such a file aggregation mechanism, e.g., MPI-IO [2], PnetCDF [3], and SIONlib [4]. All of them assume that parallel applications are based on the single program multiple data ‘SPMD’ execution model, that is, all processes access files at the same time. Those libraries have two main issues, If an application is written using Posix file I/O APIs, it must be rewritten using their APIs. In the case that an application is not written in the SPMD manner, its modification cost is much higher. The other issue is the metadata management of libraries such as PnetCDF and SIONlib. Because the metadata of files is sequentially handled at the user level, metadata operations, such as creation and extension of files, limits scalability.

In this paper, a new file aggregation mechanism called the file composition technique is proposed. It makes application programmers select the I/O pattern such that each process may access multiple individual files. In the proposed technique, the middleware gathers files created in the application and stores them into a single shared file in a parallel file system. The Lustre file system [5] is currently utilized.

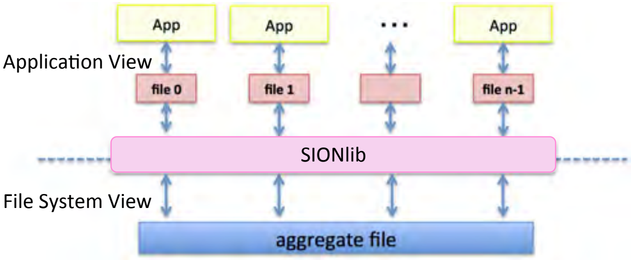


Figure 2: SIONlib

The remainder of the paper is structured as follows. In Section 2, existing file aggregation techniques are introduced. The concept of the file composition technique is proposed, followed by its design and implementation, in Section 3, and its basic performance is evaluated in Section 4. Section 5 presents a summary.

## 2. Related Work

Several techniques gather multiple data and store them into one file and that support mechanisms to access the gathered file by parallel processes.

### 2.1 SIONlib

Figure 2 shows the basic concept of SIONlib [4]. If the I/O pattern of a parallel application is such that each process creates multiple files, $N$, then the collective SIONlib function must be called $N$ times, and this yields $N$ aggregated files. As a result, the performance gain by the file aggregation can be degraded.

### 2.2 NetCDF and HDF5

NetCDF [6] and HDF5 [7] are the other I/O aggregation libraries providing self-describing data formats. NetCDF primarily supports a way to access files containing array-oriented data. HDF5 primarily supports a way to access hierarchical data. By using these libraries, the application programmer can describe and store various data into one file with the meta-information about the data and the data format. Parallel NetCDF [3] and Parallel HDF5 [7] are the parallel versions of NetCDF and HDF5, respectively. Both are extended by using MPI-IO and support storing data dispersed among parallel processes (Figure 3).

In both netCDF and HDF5, the data aggregation takes place so that the data structure is preserved. (P)netCDF and (parallel) HDF5 give users a good view of a complex data structure. However, when a user tries to change the file structure, then the aggregated file must be restructured. Thus, users sacrifice the flexibility of their data format.

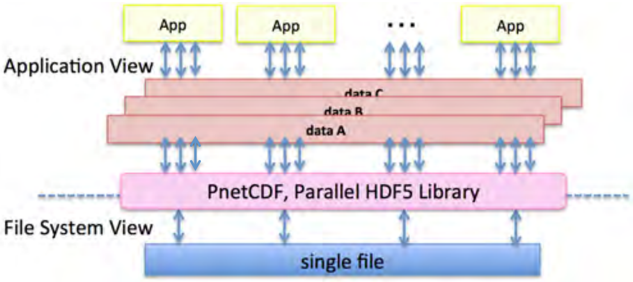


Figure 3: PnetCDF and Parallel HDF5

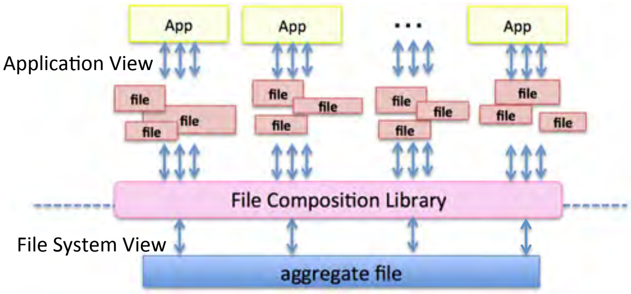


Figure 4: Concept of the file composition technique

## 3. File Composition Technique

In this section, we propose a new approach to store multiple data generated by a parallel application into a single shared file.

### 3.1 Proposal

We propose another approach to store multiple data generated by a parallel application into a single shared file by holding an application's I/O pattern. We call the new approach *file composition technique*. Figure 4 shows the basic concept of the file composition technique.

This file composition technique composes data, which looks like a file from the viewpoint of the application, into an aggregated larger file. Unlike SIONlib, (P)netCDF, and (parallel) HDF5, no restrictions are on how the "files" format the application view, and how they are organized and/or structured. Thus, the file composition technique can provide more flexibility than can the existing techniques.

In this paper, we call a file from the application view a "logical file," and a composed file "a physical file." In the application layer, application processes can access various data as if they are individual files. But in the file system layer, all logical files are stored on a single physical file. The file composition library aggregates I/O requests of application processes and translates them into I/O operations for a single shared physical file.

## 3.2 Design

In this subsection, a design of the implementation of the file composition library is described. We explain how to manage a logical file, how to map logical files to physical files, and how to access the physical file. Figure 5 shows a design overview of the file composition library.

Logical files are managed by our library at the computing node where each logical file has been created, not by a single server such as a metadata server. Even if parallel processes access each individual logical file at the same time, requests are not concentrated at a particular node. Thus, it is expected that the I/O performance does not decrease. Each library process treats logical files as separately divided into a metadata object and data objects. A metadata object holds the information about every logical file, filename, size, position in the physical file, and so on. A data object is an entity of each logical file.

The metadata object and the data objects are split into blocks of a constant size and are distributed into a logical file. By making the block size the same as the stripe size of the file system, each library process can access each individual stripe block without collisions with other processes.

The stripe size of a parallel file system is the maximum message size with which clients and the storage server can communicate. If each library accesses a physical file with the stripe size, then it is expected to have a higher I/O performance. So, each library process has a buffer for data objects with the length equal to the stripe size. If an application process writes many small logical files, then the buffer holds the data of these logical files and writes to a physical file when the buffer becomes full.

# 4. Evaluation

We implemented the file composition library and evaluated its performance on the Lustre file system. Figure 6 shows the evaluation environment where all of the following experiments took place. The cluster consists of 32 computing nodes and a parallel file system, Lustre. Each computing node has two Intel X5670 CPUs and 96 GiB memory and is connected with the other nodes and the file server nodes by Infiniband 4xQDR. Lustre consists of one Meta Data Server (MDS) and 12 Object Storage Servers (OSSs). Each OSS has 128 Object Storage Targets (OSTs) and the total capacity of the file system is 10 PB. Each OSS has 192 GiB memory. Two switches for the computing nodes and the file system servers are connected with four links. Throughout this evaluation section, the buffer size of the file composition technique is set to 16 MiB.

## 4.1 Micro Benchmark

We measured the basic performance of our file composition library and the performance of POSIX I/O for comparison. We show the performance of the create, write, and read operations. In each benchmark, the number of processes is increased from one process to 32 processes. Each process runs on one computing node.

On the Lustre file system, files are split into multiple object blocks and each object block is distributed to multiple servers. The size of each object block is called the *stripe size*, and the number of OSTs used to distribute the object blocks is called the *stripe count*. These parameters are factors of the I/O performance. In the case of POSIX I/O, the stripe count is set to 4 and 64, while in the case of using the file composition library, the stripe count is set to 64 and 160. The stripe size is set to 16 MiB constant in all cases.

### File Create

Figure 7 shows the time of file creation. Each process repeats the procedure of calling the create and then the close functions for a file 128 times, yielding 128 files. The creations of each file are synchronized and the time is measured from the time to the first create call until the termination of the slowest call of close on the 128th file.

In the case of 32 processes, POSIX with the 4 stripe count took 0.69 seconds and POSIX with the 64 stripe count took 37 seconds. The file composition with the 64 stripe count and with the 160 stripe count took 0.08 seconds.

In all cases, the elapsed time increases as the number of processes increases. But the increasing rate of the file composition case is smaller than that of POSIX. This phenomenon can be considered as that the I/O requests are concentrated on the Meta Data Server in the POSIX I/O. In contrast, the file composition technique can decrease the degree of concentration.

In the POSIX case, when the stripe count gets larger, the creation time gets slower. This is because each create operation involves the operation on the Object Storage Target. In the file composition case, the creation time is constant over the number of processes, independent from the stripe count.

### Write

Figure 8 shows the throughput of open-write-close. In this case, each process calls open, write, and close. Each process accesses 128 individual files. The open calls are synchronized and the time is measured from the time to the first create call until the termination of the slowest call of close on the 128th file.

The upper graph of Figure 8 shows the result of the case of setting the file size to 1 MiB, and the lower one shows the result of the case of setting the file size to 16 MiB.

In the case that the file size is set to 1 MiB, the file composition performance is almost twice better than that of POSIX. The assumed reason for this performance improvement is that the file creation time is reduced and that the

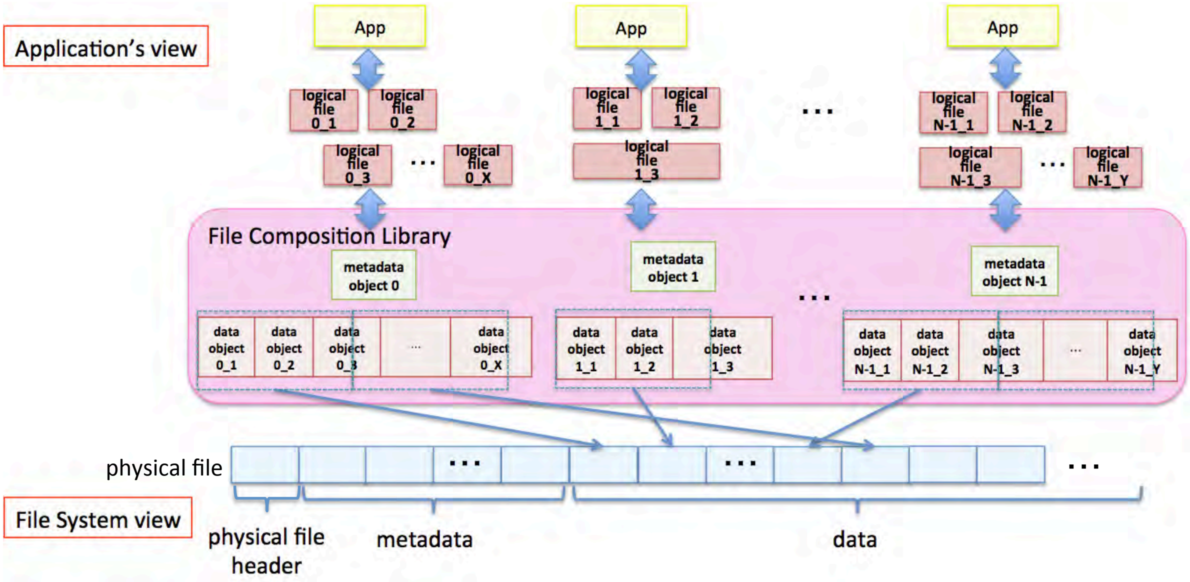


Figure 5: Design overview of the file composition library

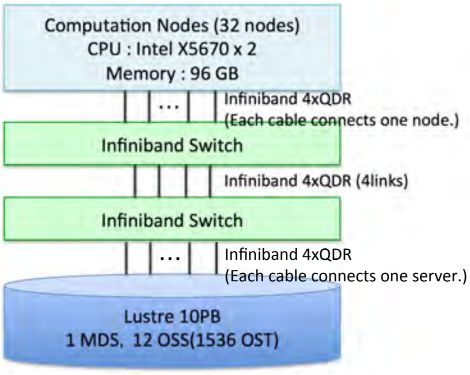


Figure 6: Evaluation environment

buffering in the composition technique decreases the number of write calls.

In the case that the file size is set to 16 MiB, the performance of the file composition is approximately twice faster than that of POSIX. This is because the file creation cost is reduced with the file composition technique. However, in the file composition case, when the stripe count is small and the number of processes is large, its performance gain is not high. This is because the number of Object Storage Targets that all processes can access is limited to the stripe counts with the file composition technique.

### Read

Figure 9 shows the throughput of open-read-close. Each process opens a file and reads data from the file and closes

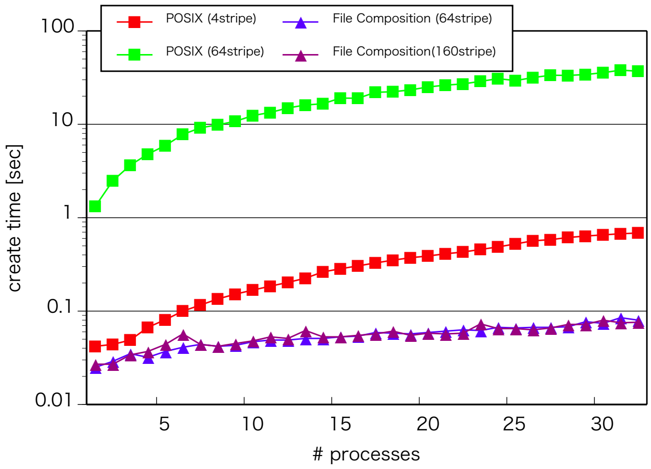
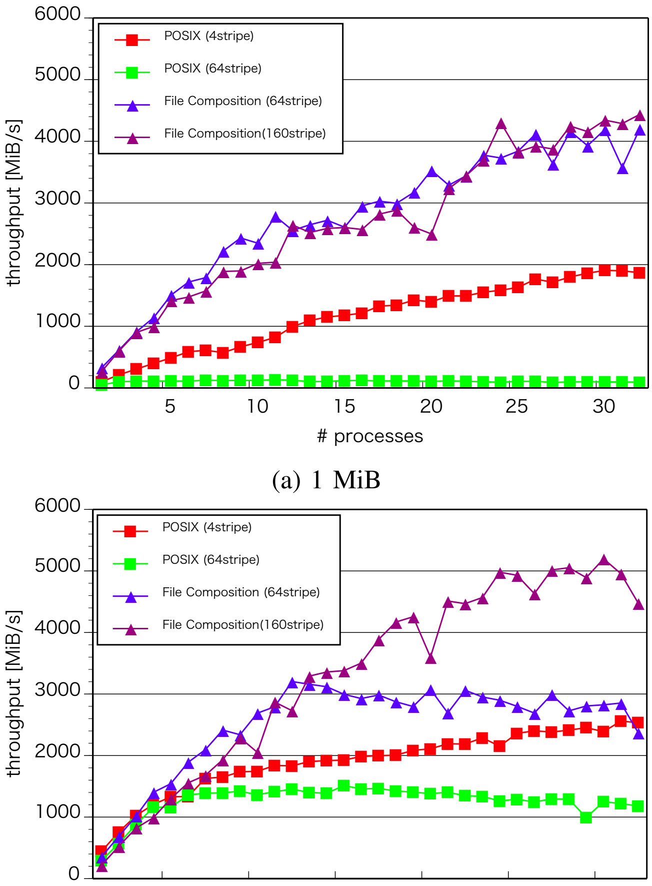


Figure 7: Time of create-close

the file. Each process accesses 128 individual files that were created in the previous write benchmark runs. To eliminate the effect of page cache in the Linux kernel, the cache of all computation nodes and all file system servers is cleared before running this benchmark.

The upper graph of Figure 9 shows the result of the case that each file size is set to 1 MiB, and the lower graph shows the result of the case that each file size is set to 16 MiB.

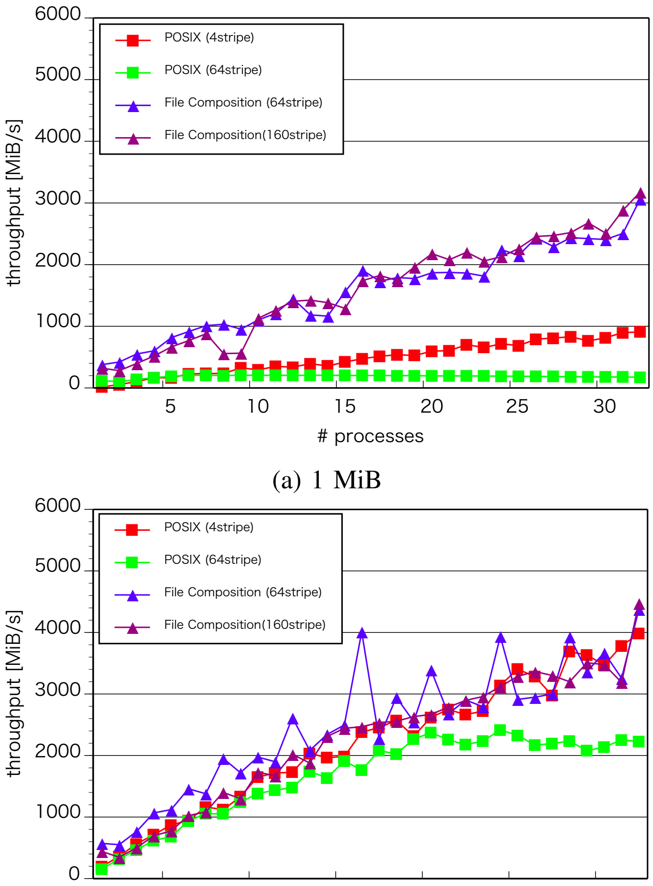
In the case that the file size is set to 1 MiB, the performance of the file composition is approximately three times faster than that of POSIX. In the case that the file size is set to 16 MiB, the performance of the file composition is



(a) 1 MiB

(b) 16 MiB

Figure 8: Throughput of open-write-close



(a) 1 MiB

(b) 16 MiB

Figure 9: Throughput of open-read-close

almost the same as that in the POSIX case. The assumed reason for this performance improvement in the case of 1 MiB is that the process can read the same stripe block as in the case of the file composition. In contrast, in the case that the file size is set to 16 MiB, which is the same size as the buffer of the file composition technique, the file composition technique has no buffering effect.

## 4.2 Evaluation with a real application

Finally, our file composition technique is applied to a real application that is a climate simulation program, called SCALE, being developed by RIKEN AICS Computational Climate Science Research Team.

When the program is running in parallel, each process periodically calls a file-output function that creates a new file and writes data into the file. When the file-output function is called, first, a new file is created and a file header is written. The file header has meta information about the data to be written in the file. Then, 16 arrays in the program are written and the file is closed.

We rewrote the file-output function by replacing the POSIX I/O functions with the function that our file composition library provides. The file composition library aggregates all files that the SCALE program creates and stores them into a single shared file throughout the whole program execution.

We executed the SCALE program in parallel and measured the elapsed time of the file-output function called by each application process. The program was executed with 32 processes and the size of one array was 10.1 MiB. The SCALE program was set up so that it repeated the file output 10 times. In the POSIX case, the stripe count was set to 0, because that was the fastest parameter variable setting. In the case of the file composition library, the stripe count was set to 160, which is the maximum value we can set.

Table 1 shows the result of the total elapsed time of the output function. The performance of the file composition library was approximately 30% faster than that of POSIX. The assumed reason for this performance improvement is

Table 1: Time of SCALE's file output.

| | the fastest process | the slowest process | average of all processes |
|---|---|---|---|
| POSIX | 12. 75 | 15.33 | 14.43 |
| File Composition | 8.79 | 12.17 | 10.37 |

[sec]

that the file composition technique could successfully reduce the number of accesses to a metadata server when each process creates files.

## 5. Summary

In this paper, we proposed the file composition technique that achieves good I/O scalability of parallel applications without changing the code using the POSIX file I/O interfaces. The library employing this technique aggregates I/O requests of parallel processes and translates them into I/O operations in a single shared file in a parallel file system, which is currently the Lustre file system. Two techniques are integrated to achieve good scalability of I/O operations. The metadata information is separately managed by each process to avoid the contention of metadata operations. The middleware of each process accesses its own stripe block exclusively in the parallel file system to avoid the contention of read/write operations.

The result of the basic performance showed that the proposed library is eight times faster than the regular POSIX I/O library in the case that 32 parallel processes create 128 files each. The results two times better throughput of parallel open-write-close operations and three times better throughput of parallel open-read-close operations. We also applied the file composition technique to the file-output function of a climate simulation program called SCALE. The elapsed time of the file-output function with the file composition library was approximately 30% faster than that of the POSIX I/O. These experiments demonstrated that I/O performance can be improved by using the file composition technique without changing the POSIX file I/O interfaces.

## Acknowledgment

This research work was partially supported by JST CREST. We use a cluster that the Information Technology Center at the University of Tokyo provides as a part of the support program for young or female researchers.

## References

[1] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, may 2011, pp. 1 –14.

[2] M. P. I. Forum. Mpi: A message-passing interface standard. [Online]. Available: http://www.mpi-forum.org/docs/docs.html

[3] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *Supercomputing, 2003 ACM/IEEE Conference*, nov. 2003, p. 39.

[4] W. Frings, F. Wolf, and V. Petkov, "Scalable massively parallel i/o to task-local files," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 17:1–17:11.

[5] O. Corporation. Lustre file system. [Online]. Available: http://wiki.lustre.org/index.php

[6] U. C. for Atmospheric Research. Netcdf (network common data form). [Online]. Available: http://www.unidata.ucar.edu/software/netcdf/

[7] T. H. Group. The hdf5 home page. [Online]. Available: http://www.hdfgroup.org/HDF5/