Cluster Comput (2014) 17:475–486
DOI 10.1007/s10586-013-0265-8

neCODEC: nearline data compression for scientific applications

Yuan Tian · Cong Xu · Weikuan Yu · Jeffrey S. Vetter · Scott Klasky · Honggao Liu · Saad Biaz

Received: 13 April 2012 / Accepted: 4 April 2013 / Published online: 24 April 2013

Abstract Advances on multicore technologies lead to processors with tens and soon hundreds of cores in a single socket, resulting in an ever growing gap between computing power and available memory and I/O bandwidths for data handling. It would be beneficial if some of the computing power can be transformed into gains of I/O efficiency, thereby reducing this speed disparity between computing and I/O. In this paper, we design and implement a NEarline data COmpression and DECompression (neCODEC) scheme for data-intensive parallel applications. Several salient techniques are introduced in neCODEC, including asynchronous compression threads, elastic file representation, distributed metadata handling, and balanced subfile distribution. Our performance evaluation indicates that neCODEC can improve the performance of a variety of data-intensive microbenchmarks and scientific applications. Particularly, neCODEC is capable of increasing the effective bandwidth of S3D, a combustion simulation code, by more than 5 times.

Keywords MPI-IO · Lustre · Data compression

Y. Tian · C. Xu · W. Yu (✉) · S. Biaz
Auburn University, Auburn University, AL 36849, USA
e-mail: wkyu@auburn.edu

Y. Tian
e-mail: tianyua@auburn.edu

C. Xu
e-mail: congxu@auburn.edu

S. Biaz
e-mail: biazsaa@auburn.edu

J.S. Vetter · S. Klasky
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

J.S. Vetter
e-mail: vetter@ornl.gov

S. Klasky
e-mail: klasky@ornl.gov

H. Liu
High Performance Computing, Louisiana State University, Baton Rouge, LA 70808, USA
e-mail: honggao@cct.lsu.edu

1 Introduction

Multicore processors have greatly expanded the aggregated computing power on large-scale systems. Scientific applications on these systems often have a great deal of redundancy in their data sets. For example, the checkpoint data from different time steps may contain substantial similarity. In our experience, we have observed that datasets from a real S3D simulation have compression ratios ranging from 1.5 to 876 via the gzip utility. Data with such internal redundancy is being transmitted as applications go through many phases of data collection, transport, analysis and visualization. A good portion of network I/O bandwidth is squandered during different phases of application execution, while many of CPU cores spend their time waiting on the data movement to be completed.

As a part of the Message Passing Interface (MPI) [8], MPI-IO [25, 27] provides a unified and portable I/O interface for scientific applications. Together with scientific data representation models such as HDF5 [28] and NetCDF-4 [1, 9, 13], MPI-IO forms the main backbone of the parallel I/O software stack on large-scale systems. It offers a crucial avenue for access to the underlying storage. Numerous techniques have been investigated and implemented to improve the scalability of MPI-IO data operations, such as extended two-phase IO [24], data sieving [26], and data shipping [19].

With the growing gap between the speed of multicore processors and that of I/O devices, it would be desirable if this disparity can be mitigated by leveraging the computing power to compress and consolidate data, thus saving the network bandwidth and relieving the data pressure on I/O devices. Along this line of considerations, scientists often undertake cumbersome post-processing steps to reduce the size of their simulation data. The back-end archival and storage system for a supercomputer can also be used to identify and purge redundant data through special hardware and software offerings. Both these approaches are typically performed in an offline manner, i.e., outside the normal execution of scientific applications. They can neither benefit the I/O efficiency at runtime, nor reduce the consumption of network bandwidth for applications. Other parallel I/O techniques, such as HDF5 [28] and NetCDF-4 [1, 9], support data compression. But the compression is carried out in an isolated manner by individual processes. Worse yet, data compression (while computationally intensive) is performed inline with respect to the productive computation, degrading the overall performance of applications. Thus it is critical for the I/O software stack to have a solution that can take advantage of the computing power from multicore and many-core processors, achieve asynchronous data compression, and reduce the consumption of network bandwidth when storing data to back-end storage devices.

In this paper, we introduce a NEarline data COmpression and DECompression (neCODEC) scheme for scientific applications. NeCODEC is designed to enable data compression at the MPI-IO level. It aims to provide a portable utility for many applications. Several techniques are introduced in neCODEC, including a nearline thread, elastic file representation, distributed metadata handling, and balanced subfile distribution. First, neCODEC is designed with a service thread for nearline data compression. It avoids burdening the main thread of applications with the onus of computationally intensive compression tasks. Secondly, to organize compressed data for efficient storage and easy retrieval, neCODEC is designed with an elastic file representation. A neCODEC file is composed of an elastic number of data files (a.k.a. subfiles), and a metafile that stores index records to data blocks in the subfiles. Thirdly, to provide scalable management of metadata records, distributed metadata handling is provided in neCODEC. Lastly, neCODEC supports balanced subfile distribution on parallel file systems such as Lustre [6]. With a balanced distribution of subfiles, neCODEC ensures that bandwidth from all storage devices can be effectively leveraged.

By realizing these techniques, we have developed a prototype of neCODEC. We evaluate its performance on the QueenBee cluster from Louisiana Optical Network Initiative. Our experiments demonstrate that neCODEC can improve the performance of data-intensive microbenchmarks and scientific applications. Particularly, it is capable of increasing the effective bandwidth of S3D, a combustion simulation code, by more than 5 times.

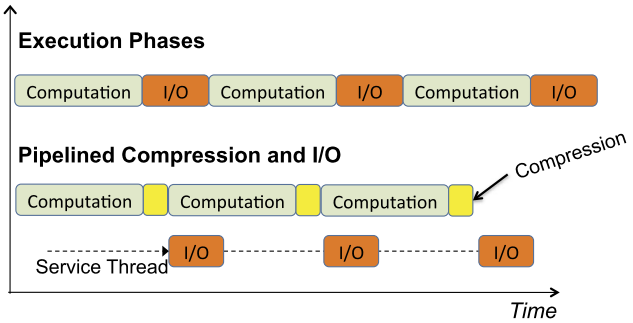


Fig. 1 Example execution with three time steps

The rest of the paper is organized as follows. In Sect. 2, we introduce the motivation of this paper and give an overview of related work. Sections 3 and 4 describe the design and implementation of neCODEC, respectively. We then provide our experimental results in Sect. 5. Finally, we conclude the paper in Sect. 6.

2 Motivation and related work

2.1 Execution phases of scientific applications

The execution of a scientific application typically consists of two main activity phases: computation and I/O. In the computation phase, applications perform numerical calculation from mathematical models that are construed from problems in various scientific disciplines. During or after the computation, data will be written out for checkpointing/restart or post-processing purposes. One pair of execution of computation and I/O phases is normally referred as one *time step*. An application normally includes one or multiple time steps. The execution of computation and I/O within a traditional simulation runtime is normally sequential. Each parallel process performs computation followed by I/O. After that, it moves on to the next time step. The upper half of Fig. 1 shows an example of the sequential execution of three time steps for one process.

With the increasing gap between computing power and storage speed, I/O bottleneck has become a critical issue for further speeding up scientific applications. One of the research efforts in solving the I/O issues focuses on utilizing the asynchronous I/O technology. A successful example of such endeavor is staging technology [3, 34], where output data are transferred into a *staging area* consisting of multiple designated compute nodes. Once data are moved into such area, computing processes resume the computation tasks while the I/O operations can be performed in the background simultaneously. By pipelining the computation with

I/O, a reduced turnaround time for applications is achieved. However, not only staging requires extra compute nodes for data storage, it also requires multiple data copies and movement over the network. The network contention could negatively impact performance on large-scale systems, particular when systems are busy. At high level, our work exploits the staging technology by creating an *in node* staging area. Such design removes the network communication and requires no extra memory copy, meanwhile enables a pipelined computation and I/O workflow. Note that for the system that has many cores per node may not be the ideal candidate for such design, as the workload becomes too heavy for one dedicated service thread. For this kind of platforms, dedicating a *staging area* becomes more reasonable than *in node staging*. However, comparing two different staging strategies is not the focus of this work. We aim to present a strategy that can be beneficial for scientific simulations running on many current large-scale systems.

Another line of thought is to reduce the I/O cost by reducing the amount of output data. Compression is a common technique for data reduction. It fits well in the consideration that to take advantage the CPU cycles which otherwise will be wasted waiting on the I/O operations to complete. The lower half of Fig. 1 shows an example of integration of I/O pipeline and compression technologies for three time steps. On top of I/O pipelining, I/O cost can be further reduced after compression, leading to further reduced simulation turnaround time. As we can see, a compression algorithm becomes essential in such design. The overall performance could be degraded by a compression algorithm with low compression ratio (calculated as the size of original data divided by the size of compressed data) because it not only introduces overhead to computation, it also does not save I/O cost. However, an efficient compression algorithm with high compression ratio is able to decrease the I/O overhead for both writing and reading. However, investigating high-performance compression algorithm for scientific data is not the focus of this work. Our goal is to demonstrate the feasibility of integrating I/O pipeline and compression technology, and introduce an efficient design that enables them for scientific applications. In addition, we integrate the compression algorithm as a pluggable module, allowing further investigation on other compression algorithms.

2.2 Related work

Many studies have been performed to improve I/O through data compression. Park et al. [18] developed a CZIP compression scheme in a file system. It is based on the Content-based naming (CBN) technique to eliminate redundant chunks. Vilayannur et al. [29] employed the content-addressable concept into the design and implementation of a file system, CAPFS (Content Addressable Parallel File System). However, potential gains of data compression can be greater at the MPI-IO level for scientific applications because it reduces the requirements on both network bandwidth and storage capacity. Our work represents an attempt in this direction.

Other efforts have been undertaken to improve the performance of parallel I/O. MPI-IO/GPFS [19] and MPI-IO/BlueGene [22] have introduced MPI-IO optimizations that are designed to take advantage of the specific features of General Parallel File System (GPFS) and BlueGene [4]. Tatebe et al. [23] have exploited the concepts of local file view to maximize the use of local I/O bandwidth in the design of a distributed file system for the Grid environment. Klasky et al. [10] and Ma et al. [17] have investigated the I/O performance benefits of multithreading. But none of them have taken into account transforming redundant computing power into gains of I/O efficiency. Abbasi et al. [3] and Zheng et al. [34] have shown the work of expediting scientific applications by I/O staging. Our work distributes the staging to each compute node to speed up I/O without impacting the computation.

There are a number of recent attempts to optimize parallel I/O on large-scale systems. Yu et al. [31, 33] have carried out a series of optimization studies on the Cray XT platforms. Yu et al. [32] also show the benefit of hierarchical file striping in allowing data access to multiple subfiles instead of a single shared file. A similar approach has been explored by Liao et al. [14] in the Parallel-NetCDF project. This paper builds on top of the earlier work of hierarchical striping to provide elastic and evenly distributed subfiles. It organizes individual data files (subfiles) in a hierarchical and elastic manner so that compressed data chunks can be striped to all storage devices, without causing the overhead of wide striping.

A recent effort has attempted to improve parallel I/O through an ADaptable IO System (ADIOS) [15, 16]. ADIOS has successfully showcased that data buffering and asynchronous I/O could bring significant benefits to scientific applications. Our neCODEC effort is orthogonal and complementary to ADIOS. It enables data compression and caching with a separate thread in a nearline manner without burdening the main thread for scientific computation.

3 Nearline data compression

To reduce data redundancy in scientific applications, we propose a new framework neCODEC for nearline data compression and decompression. We first describe software components of neCODEC and then discuss the design of neCODEC in detail, focusing on elastic file representation, data segmentation and compression, balanced subfile distribution, and distributed metadata handling.

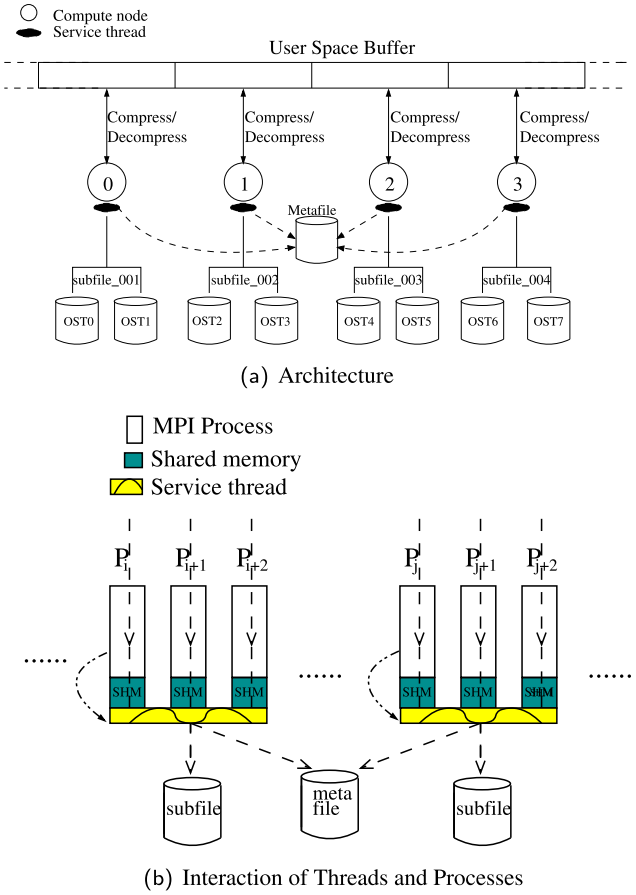


Fig. 2 Software components of neCODEC

3.1 Software components of neCODEC

Figure 2 shows the components of neCODEC. neCODEC is targeted for systems that consist of a set of compute nodes, all connected through an interconnection network to a set of storage nodes. neCODEC requires no software modification at the storage side. For data originally to be stored as a file, neCODEC will create a metafile and several data files (subfiles). neCODEC creates a dedicated service thread per node to support nearline (asynchronous) data compression. The service thread is spawned by the first process on a node. Main efforts in neCODEC will be focused on offloading all I/O related data processing from MPI processes to this service thread in an efficient manner. As shown in Fig. 2(a), application data are first divided into chunks among MPI processes. Then all MPI processes pass their chunks to the service thread on the same node for compression. Figure 2(b) shows the service threads on two compute nodes and their interaction with MPI processes through shared memory. In a round-robin manner, the service thread checks the ready flag of each process deployed in shared memory. Once one data segment in some MPI process is ready, the service thread detects that and begins compression according to the segment information provided by the MPI process.

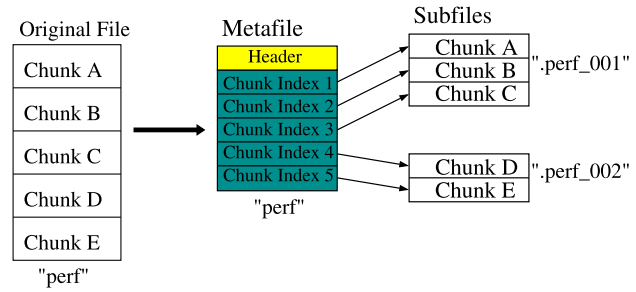


Fig. 3 The format for a neCODEC File

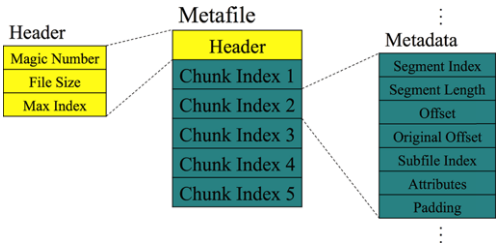


Fig. 4 The format for a metafile

3.2 Elastic file representation

File representation of neCODEC is a conceptually simple design. It divides application data into small chunks, and then compresses them using a configurable algorithm, currently zlib. Alternative algorithms such as bzip2 or other lossless compression algorithms for scientific data can be plugged into neCODEC.

Figure 3 shows the overall file format of one metafile and two subfiles. The metafile carries the original file name which is “perf”. The subfile name is a combination of the original file name and a subfile index. It is formatted as: .OriginalName_xxx. “xxx” stands for the subfile index. Thus two subfiles are named .perf_001 and .perf_002, respectively. Subfiles are dynamically created based on the growth of the original file. For each chunk, a record is created to store its attributes such as subfile location, size, and offsets. This record is then inserted into the metafile. The compressed data chunk is stored into a subfile.

As shown in Fig. 4, a neCODEC metafile consists of two types of records. It starts with a fixed-size header record, followed by an array of metadata records, one per data chunk. We describe their formats in detail below.

3.2.1 Format of header

A metafile consists many fixed-size records. The first record is called a metafile header. It is padded to the same size as the

other regular metadata records. The header enables quick access to general information about the file. Currently, it contains three fields:

- **Magic Number**—A predefined number used to identify the neCODEC file format.
- **File Size**—The size of the original file.
- **Max Index**—The biggest index number of all metadata records.

3.2.2 *Format of metadata records*

Metadata records contain many attributes of compressed data chunks in separated subfiles. The length of each record is 64 bytes. The fields include:

- **Segment Index**—The global index of the chunk in the original file. It is calculated by the chunk offset in the original file (*Original Offset*) divided by the chunk size.
- **Segment Length**—The actual length of the chunk after compression.
- **Offset**—The beginning offset of a compressed chunk inside its subfile.
- **Original Offset**—The beginning offset of a chunk in the original file (without compression).
- **Subfile Index**—The index of the subfile that contains the compressed chunk.
- **Miscellaneous Attributes**—A one-byte field that stores two attributes. The first bit is a flag that is set when the data chunk is compressed. The remaining 7 bits are used to store the version number. These attributes are described further in Sect. 3.3.
- **Padding**—This is currently unused. It may be used in future to store a hashed digest for the data chunk.

3.3 Data segmentation and compression

Figure 5(a) shows data segmentation and compression in neCODEC. Application data are divided into fixed-size data chunks. The actual chunk size is adjustable during run-time (by using MPI hints). Each data chunk is compressed before it is stored into a subfile. A metadata record is generated for each compressed data chunk, and then stored into the metafile. When the data buffer cannot be divided into chunks evenly, the flanking regions will be written into the subfile directly without compression. To read the data back, a process first retrieves the metadata record. Then it locates the corresponding subfile, offset, and length for the data chunk. Data are then read from the subfile, decompressed, and returned to the application.

Uncompressed partial chunks caused by unaligned flanking regions require special handling. Particularly, when the remaining data arrives, the partial chunk needs to be combined with new data to form a new chunk. We adopt a virtual

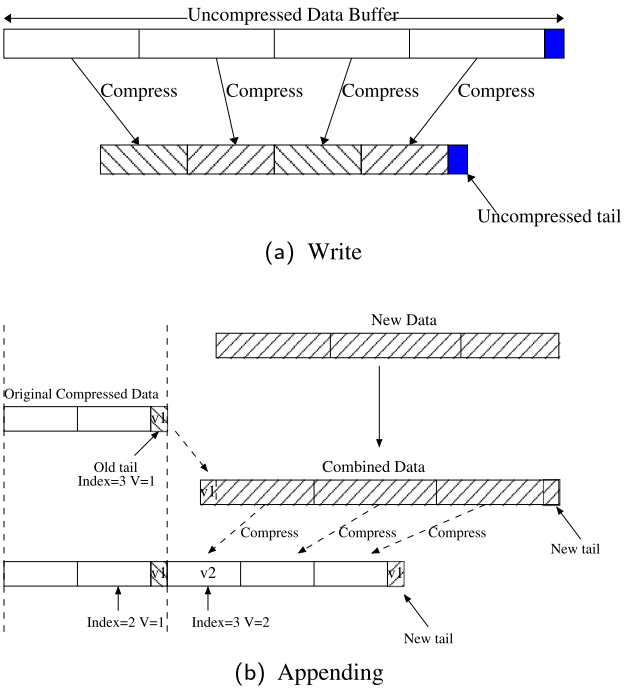


Fig. 5 Data compression during write

file appending approach as shown in Fig. 5(b). Every chunk in the file is tagged with a version number and a flag that indicates whether it is compressed. When a new chunk is formed, its metadata record is updated accordingly. To distinguish two versions of a data chunk with the same offset, a version number is stored inside the metadata record. This allows us to purge the stale chunks at a later point of time. A similar approach is adopted when a file segment is to be overwritten. An updated chunk will be marked with an increasing version number. When the file size grows to a certain level, a chunk purging operation can be performed to remove old chunks. These old chunks are found by traversing metadata records.

3.3.1 *Balanced subfile distribution*

Subfiles are dynamically created based on the growth of data. Any subfile can be selected to host incoming data chunks, thereby enabling elastic file growth. In addition, on the Lustre file system, a balanced distribution is enabled for subfiles to leverage aggregated bandwidth from all storage devices. The striping pattern of a subfile consists of three parameters: stripe size, stripe width, and stripe index. Based on our earlier study [32], we organize subfiles in a hierarchical manner so that all subfiles are evenly distributed and striped to all storage devices, without causing the overhead of wide striping. Figure 2 gives an example of the distribution of subfiles. All subfiles have a stripe width 2, with the first subfile located at the first storage device. Stripe indices of other subfiles are set accordingly.

3.4 Distributed metadata handling

In neCODEC, every node creates one service thread which writes metadata records to a shared metafile. When there are a large number of processes performing I/O, it is critical to enable an efficient approach for metadata access. To meet this requirement, we develop a distributed metadata handling scheme. The entire metadata for a file are fully distributed to all service threads. All service threads allocate a block of memory as the metadata buffer for storing transient metadata records. When the buffer usage reaches a threshold percentage, e.g., 80 %, the service thread will flush all existing metadata records into the on-disk metafile. In the meantime, new records can still be inserted into the other 20 % of the buffer. In this scheme, all metadata are stored as large and sequential blocks to the disk, allowing efficient disk access. For a read request, a service thread will first search for metadata records from its metadata buffer. If it is not found, the service thread will retrieve a number of contiguous metadata records, along with the required ones. Such design allows easy integration with data staging or prefetching techniques. With reasonable locality, this scheme combines multiple metadata records into one and avoids the cost of frequent metadata access. Note that, in this exploratory research, we focus on studying the benefits of compression and strictly divide the content of a file among MPI processes. Each service thread is managing a distinct set of metadata records. The consistency issue among metadata buffers from different threads is then avoided through this rigid distribution. In the future, we plan to investigate the feasibility of dynamic partitioning methods.

4 Implementation

We have implemented a prototype of neCODEC by modifying the MPI-IO implementation of MPICH2. In this prototype, several other implementation details are worth mentioning here. First, the majority of the code is introduced as an extension to the ADIO implementation of Lustre file system. Second, the service thread is created by the first MPI process, and it is not created until the first call to open (create) a file. This thread stays alert for upcoming I/O requests from any process on the same node. It will not exit until the MPI_Finalize() is called. Third, data cached in the service thread will be flushed either when the corresponding file is closed by the application, or when an explicit file synchronization request is made.

Because disk access is expensive, we support data caching in neCODEC to minimize the frequency of disk access. Two shared memory buffers are allocated for each process. One is a metadata buffer (currently 2MB), whose usage is described in Sect. 3.4. Another is the cache for compressed data chunks. Similar to the metadata cache, the data chunks will be flushed to the disk file when they have taken up more than 50 % of the total data cache.

5 Performance evaluation

Our experiments were conducted on the QueenBee cluster from Louisiana Optical Network Initiative (LONI). QueenBee consists of 668 nodes. Each node is equipped with dual-socket quad-core Intel Xeon 64-bit 2.33 GHz processors, 512 KB cache, 8 GB physical memory (1 GB per core). All nodes run Red Hat Enterprise Linux 4 operating system. The storage system is a 58 TB Lustre file system with 16 Lustre OSTs. In this paper, we focus on the performance of write operations, while read performance is also evaluated to examine the performance impact of metadata handling.

5.1 Independent I/O

For independent I/O performance measurements, we use *perf.c* from the ROMIO [20] distribution. This program performs concurrent writes to a shared file. Each process writes a contiguous 24 MB data at disjoint offsets based on its rank in the program. We fill in the test data with random alpha-numerical characters. Our measurement indicates that such data has a compression ratio of 2.3. Compression ratio is defined as the ratio between the size of the original data and the size after compression (via zlib). This generates a maximum of 3 GB data size with 128 processes. The average time taken for each iteration is recorded. Because neCODEC uses a dedicated asynchronous service thread to perform I/O, data chunks are buffered in the service thread when MPI_File_Write() returns. The service thread later flushes data chunks when the buffered data reaches a threshold or when the file is closed. So we calculate the write time as the sum of MPI_File_Write() time and the actual I/O time inside the service thread. Because neCODEC uses a dedicated I/O thread, which may impact performance when MPI process layout varies, we examine the write performance with different process arrangement.

Each node on QueenBee is equipped with 8 compute cores. When a program is launched to use all 8 cores, the service thread of neCODEC will compete with the regular MPI processes for compute cores. To examine the effect of using one service thread, neCODEC is tested with different numbers (7 and 8) of cores (processes) per node. The "original" case for ROMIO is always run with 8 processes per node as theoretically its performance is not influenced by the MPI process arrangement.

Figure 6 shows the performance comparison between ROMIO and neCODEC. Lustre enables client data cache in the kernel. To mitigate caching effects, we calculate the I/O time of ROMIO by forcing a file sync operation after the

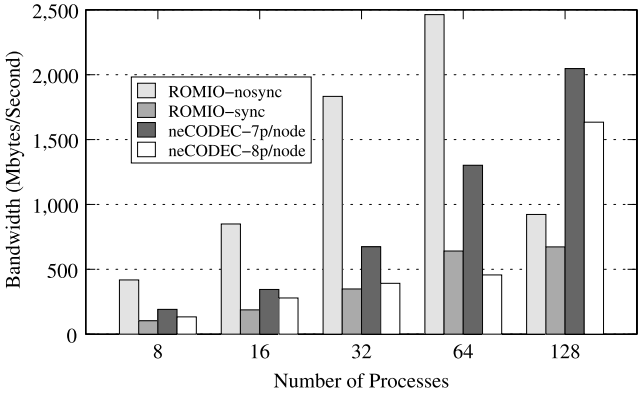


Fig. 6 Independent I/O performance

write operation. For complete comparison, we also include the performance of independent I/O operations without file sync. As shown in Fig. 6, ROMIO outperforms neCODEC in most cases with caching effects. However, the scalability issue is also observed. Hierarchical striping helps neCODEC achieves good scalability and eventually surpasses the original ROMIO (no sync) on 128 processes. When file sync operations are used to reduce caching effects, neCODEC outperforms the original ROMIO in all cases, with either 7 or 8 processes per node. When neCODEC is run with 8-process per node, the performance is constrained by the competition for CPU between service thread and processes. With 7-process per node, neCODEC benefits greatly from the dedicated core for compression and delivers significantly higher bandwidth. Compared to ROMIO (with sync), neCODEC achieves a bandwidth improvement up to 3 times with 128 processes.

Taken together, the performance evaluation of independent I/O tests, we conclude that neCODEC is able to improve I/O bandwidth for highly compressible data by leveraging a portion of the CPU power. A dedicated compute core will help neCODEC significantly. So one compute core is reserved on each compute node in the following experiments unless otherwise indicated.

5.2 Performance of MPI-Tile-IO

MPI-Tile-IO [21] tests the performance of tiled access to a two-dimensional dense matrix, simulating the type of workload that exists in some visualization applications and numerical applications. In our experiments, each process renders a 1×1 array of displays, each with 2048×1536 pixels. The size of each element is 8 bytes, leading to a file size of $24 * N$ MB, where N is the number of processes.

Figure 7(a) shows the performance improvement of neCODEC compared to ROMIO. We observe that neCODEC outperforms the original ROMIO in all cases. A maximum of 3 times improvement is achieved with 25 processes, while 1.6 times bandwidth improvement is

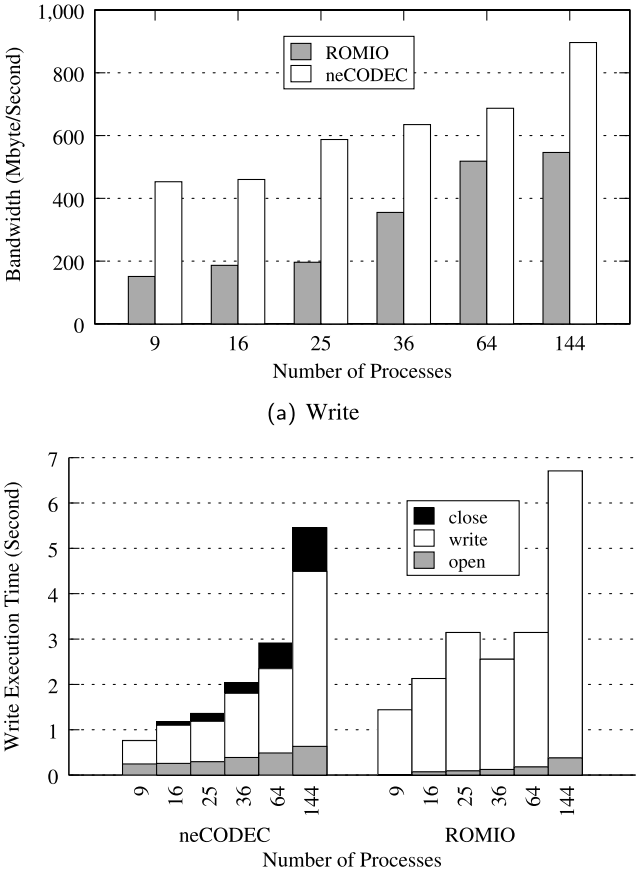


Fig. 7 MPI-Tile-IO write performance

achieved with 144 processes. Moreover, neCODEC presents a good overall scalability.

We further examine the time breakdown of MPI-Tile-IO. In the case of neCODEC, the file *open* time includes the time to spawn the service thread; the *write* time is a combination of MPI_File_Write() and the write() system call operation inside the service thread; and the *close* time includes the time to flush cached data and metadata records to the disk. Figure 7(b) shows the comparison of timing breakdown between ROMIO and neCODEC. Spawning service threads does not significantly increase the file open time. However, when a file is closed, much time is spent by the main thread to synchronize with the service thread, which has to flush cached data and metadata into storage. However, compressed data leads to much reduced time in writing the file. The savings from the file write time actually outweighs the overheads from file open and close calls. Therefore, this results in better I/O performance for neCODEC compared to ROMIO.

Figure 8 shows the read performance of MPI-Tile-IO when the data are initially in the disk. In this case, each process has to retrieve the metadata before reading data from disk. We observe that, due to the overhead of meta-

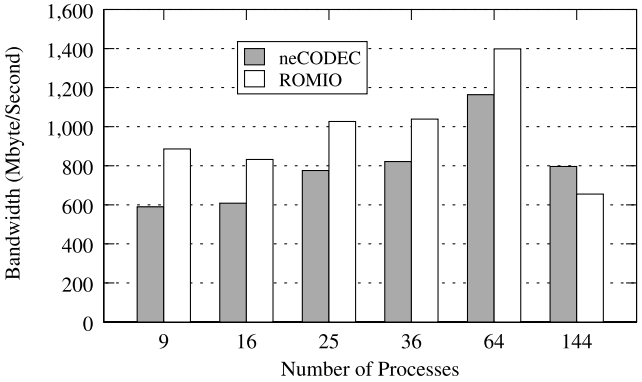


Fig. 8 MPI-Tile-IO read performance

data retrieving, neCODEC performs worse than the original ROMIO except when there are 144 processes. neCODEC is able to support good read performance at scale. It exhibits good performance benefits, and presents a viable I/O technique for data-intensive applications. Other techniques such as data staging may be further exploited to avoid the metadata overhead as part of future research.

5.3 Performance of NAS BT-IO

NAS BT-IO [30] is developed at NASA Ames Research Center based on NAS BT (Block-Tridiagonal) parallel benchmark. The entire dataset undergoes diagonal multi-partitioning and is distributed among a square number of MPI processes. Its data structures are represented as structured MPI datatypes and written to a file periodically, typically every 5 timesteps. There are several different BT-IO implementations, which vary on how its file I/O is carried out among all the processes. In our experiments, we use an implementation that performs I/O using MPI-IO collective I/O routines, so called *full mode* BT-IO. In this mode, BT-IO performs 40 iterations of collective writes, followed by a verification phase which performs 40 iterations of collective reads. We run BT-IO with 9 to 144 processes and two different classes. BT-IO Class A generates 400 MB and Class B 1697.93 MB.

Figure 9(a) shows the write performance of BT-IO. NeCODEC is still able to improve the effective I/O bandwidth for BT-IO even when the data are not highly compressible. Bandwidth improvements of 37 % and 18 % are observed for Class A and Class B, respectively. The peak bandwidth of Class A is achieved at 16 processes. This is because the Class A data size is relatively small. More writers lead to finer grained data elements for each process. Hence many smaller I/O requests are generated to Lustre, degrading the I/O performance. A similar trend is observed for the performance of Class B. For both classes, neCODEC exhibits better performance compared to the original ROMIO.

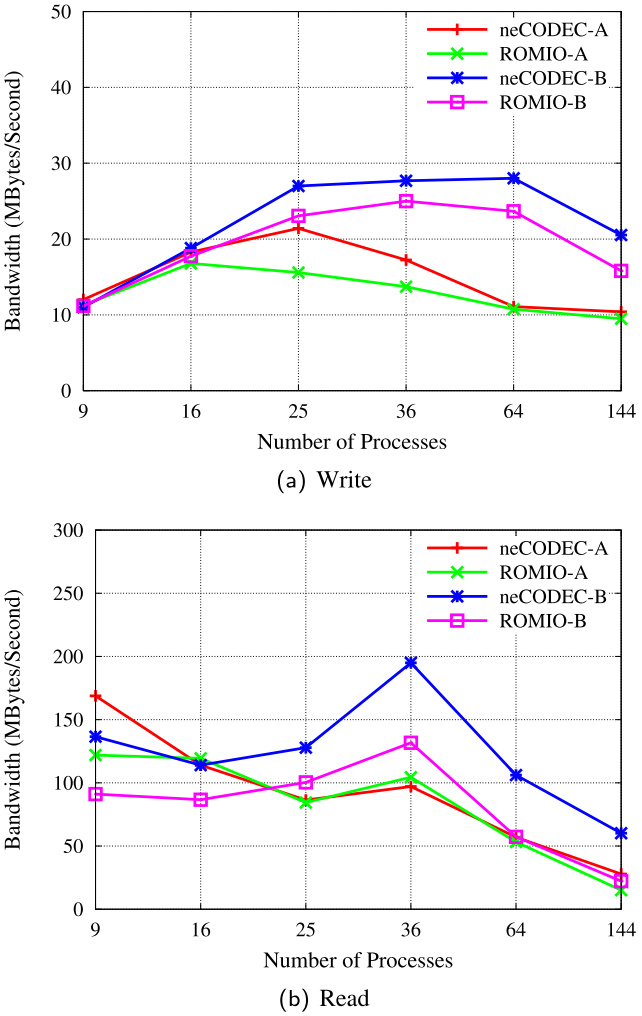


Fig. 9 Write/read performance of BT-IO

The read performance of BT-IO is shown in Fig. 9(b). As BT-IO performs read immediately after write, neCODEC is able to retrieve its metadata from cache. Accordingly we observe better performance for neCODEC. A maximum of 49 % improvement is achieved for Class B. For Class A, the performance gain is smaller.

As neCODEC introduces computation overhead for each process to compress data, it is also important to evaluate the overall performance impact to application. For this purpose, we measure the total execution time of BT-IO, which includes both computation and I/O. The results for Class A and Class B are shown in Fig. 10. As can be seen, neCODEC demonstrates shorter execution time comparing to the original ROMIO, particularly at larger process counts. The performance benefit is less significant for Class A due to smaller workload. Such result validates the rational of our design. Overall, neCODEC is able to achieve a 35 % improvement comparing to the original ROMIO.

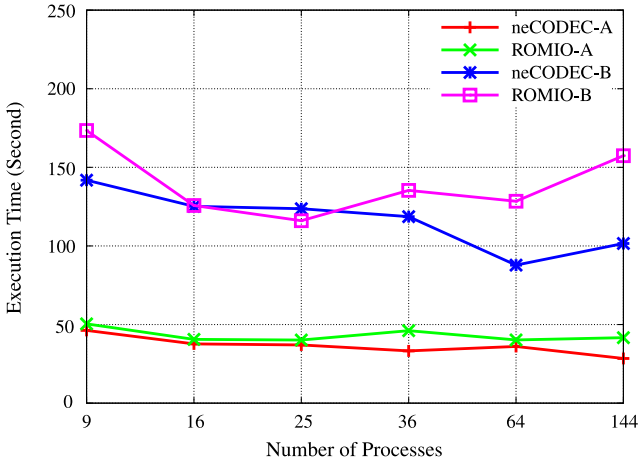


Fig. 10 Total execution time of BT-IO

5.4 S3D Combustion simulation application

S3D [5] is a combustion simulation application using direct numerical simulation (DNS) solver developed at Sandia National Laboratories. It solves fully compressible Navier-Stokes, total energy, species and mass continuity coupled with detailed chemistry. This code traditionally runs on a large number of processors, on many of the largest supercomputers in the Department of Energy and at the National Science Foundation centers. One of the more taxing parts of the simulation is spent in the I/O, and our group has performed lots of research to expedite this process [3, 15, 16, 34].

Typically the data that is produced in the simulation includes three-dimensional Cartesian mesh points of four global arrays, (mass, velocity, pressure, and temperature), along with many chemical species. We learn from our interaction with S3D scientists that datasets are usually written as either small (20^3) or medium (50^3), or large (100^3) data points per MPI process. In our performance evaluation we use the medium data size, which produces about 15.3 MB of data per process per checkpoint. Therefore, with 125 processes in our experiments, the total amount of data is 1.9 GB per output. We measure the time for ten checkpoints, and show the results in Fig. 11. In order to make the experiment comparable to "real" S3D data, we validate our runs with the early time steps of the S3D simulation. The datasets are confirmed to be highly compressible.

We observe significant performance improvements with neCODEC compared to the original ROMIO in all cases. With 64 processes, neCODEC achieves about 3.8 GB/s bandwidth, which is more than 4.4 times improvement of the original ROMIO. The bandwidth drops at 125 processes. We suspect that it can be partly attributed to the increasing number of metadata records and the metadata management overhead. Thus we further evaluate the performance of S3D with different chunk sizes.

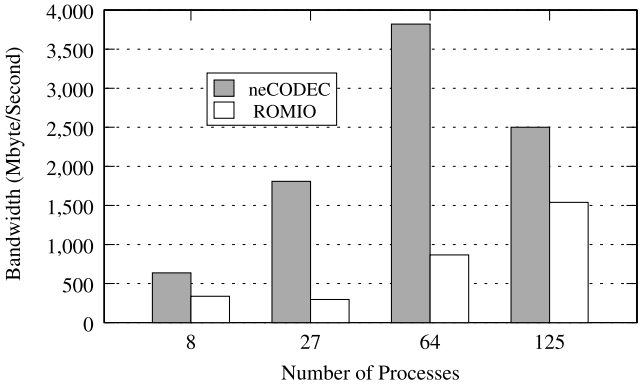


Fig. 11 S3D performance

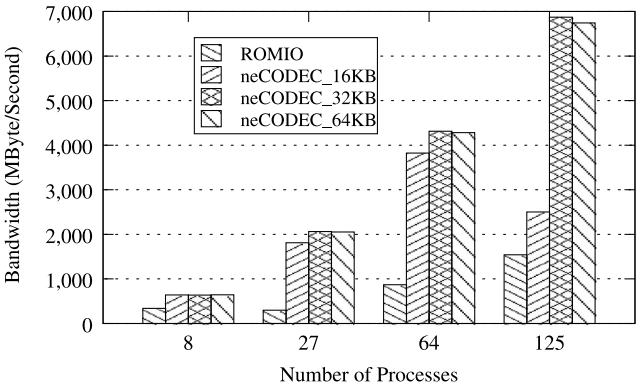


Fig. 12 The performance of S3D with varying chunk sizes

5.4.1 Tuning neCODEC chunk size for S3D

To further examine the impact of data management overhead, we measure the performance of S3D with different chunk sizes. Figure 12 shows the performance of ROMIO and neCODEC with different chunk sizes. As shown in the figure, the chunk size of 32 KB leads to the best bandwidth, an improvement of 5 times compared to ROMIO on 64 processes, and 4.5 times on 125 processes. Compared to the chunk size of 16 KB, 32 KB leads to better compression ratio, and less number of data chunks, therefore more efficient metadata handling. The 64 KB chunk size performs slightly lower than 32 KB. This is because that an even larger chunk size at 64 KB slows down compression, therefore degrading the overall performance.

6 Conclusions

In this paper, we have explored the feasibility to design an I/O compression framework to leverage a portion of the computing power to compress and consolidate scientific datasets, and mitigate the speed disparity between multi-core processors and I/O devices. To this aim, we have designed NEarline data COmpression and DECompression

(neCODEC) to deliver efficient I/O for data-intensive scientific applications. A prototype implementation of neCODEC has been accomplished. Inside neCODEC, a number of salient techniques are implemented accordingly, including a nearline service thread, elastic file representation, balanced subfile distribution, and distributed metadata handling. The performance of neCODEC is evaluated using a set of data-intensive microbenchmarks and scientific applications. Our experimental results demonstrate that neCODEC can achieve an overall performance improvement for scientific datasets with different compositions and varying compression ratios. For a combustion simulation application, S3D, we show that neCODEC increases its effective bandwidth by more than 5 times.

In the future, we intend to study the scalability of neCODEC, particularly how it can be integrated with ADIOS [15, 16] to exploit the data redundancy of scientific applications on petascale systems hosted at Leadership Computing Facilities. Such integration also provides us an opportunity to evaluate neCODEC with other high performance lossless or lossy compression algorithms, such as ISABELA [7, 11, 12]. In addition, we plan to investigate how to optimize the read performance of neCODEC so that it can benefit applications with predominantly read access. Furthermore, we plan to research the applicability of neCODEC to other large-scale computing environments such as BlueGene systems with other parallel file systems such as PVFS [2] and GPFS [22].

Acknowledgements This work is funded in part by National Science Foundation awards CNS-0917137 and CNS-1059376. This research is sponsored in part by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. This research is conducted with high performance computational resources provided by the Louisiana Optical Network Initiative (http://www.loni.org). We are very grateful for the technical support from the LONI team.

References

1. NetCDF-4. http://www.unidata.ucar.edu/software/netcdf
2. The parallel virtual file system, version 2. http://www.pvfs.org/pvfs2
3. Abbasi, H., Eisenhauer, G., Wolf, M., Schwan, K.: Datastager: scalable data staging services for petascale applications. In: HPDC '09, New York, NY, USA (2009)
4. Adiga, N., Almasi, G., Almasi, G., et al.: An overview of the BlueGene/l supercomputer. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (Supercomputing '02), Los Alamitos, CA, USA, pp. 1–22 (2002)
5. Chen, J.H., et al.: Terascale direct numerical simulations of turbulent combustion using S3D. Comput Sci. Discov. **2**(1), 015001 (2009). http://stacks.iop.org/1749-4699/2/015001
6. Cluster File System, Inc.: Lustre: a scalable, high performance file system. http://www.lustre.org/docs.html
7. Gong, Z., Lakshminarasimhan, S., Jenkins, J., Kolla, H., Ethier, S., Chen, J., Ross, R., Klasky, S., Samatova, N.: Multi-level layout optimization for efficient spatio-temporal queries on Isabela-compressed data. In: 2012 IEEE 26th International, Parallel and Distributed Processing Symposium (IPDPS), pp. 873–884. IEEE Press, New York (2012)
8. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput. **22**(6), 789–828 (1996)
9. Jenter, H.L., Signell, R.P.: NetCDF: a public-domain-software solution to data-access problems for numerical modelers (1992)
10. Klasky, S., Ethier, S., Lin, Z., Martins, K., McCune, D., Samtaney, R.: Grid -based parallel data streaming implemented for the gyrokinetic toroidal code. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC'03), p. 24, Washington, DC, USA, (2003). http://portal.acm.org/citation.cfm?id=1048935.1050175
11. Lakshminarasimhan, S., Shah, N., Ethier, S., Klasky, S., Latham, R., Ross, R., Samatova, N.: Compressing the incompressible with Isabela: in-situ reduction of spatio-temporal data. In: Euro-Par 2011 Parallel Processing, pp. 366–379 (2011)
12. Lakshminarasimhan, S., Shah, N., Ethier, S., Ku, S., Chang, C., Klasky, S., Latham, R., Ross, R., Samatova, N.: Isabela for effective in situ compression of scientific data. Concurr. Comput. **25**, 524–540 (2013)
13. Li, J., Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R.: Parallel netCDF: a high performance scientific I/O interface. In: Proceedings of the Supercomputing '03 (2003)
14. Liao, W.k., Choudhary, A.: Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08), Piscataway, NJ, USA, pp. 1–12 (2008)
15. Lofstead, J., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible I/O and integration for scientific codes through the adaptable I/O system (adios). In: 6th International Workshop on Challenges of Large Applications in Distributed Environments, Boston, MA (2008)
16. Lofstead, J., Zheng, F., Klasky, S., Schwan, K.: Adaptable, metadata rich IO methods for portable high performance IO. In: Parallel and Distributed Processing International Symposium, pp. 1–10 (2009)
17. Ma, X., Winslett, M., Lee, J., Yu, S.: Improving MPI–IO output performance with active buffering plus threads. In: Proceedings of International Parallel and Distributed Processing Symposium, p. 10 (2003). doi:10.1109/IPDPS.2003.1213165
18. Park, K., Ihm, S., Bowman, M., Pai, V.S.: Supporting practical content-addressable caching with czip compression. In: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07), Berkeley, CA, USA, pp. 1–14 (2007)
19. Prost, J.P., Treumann, R., Hedges, R., Jia, B., Koniges, A.: MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In: Proceedings of Supercomputing'01 (2001)
20. Thakur, R., Ross, R., Latham, R., Lusk, R., Gropp, B.: Romio: a high-performance, portable MPI-IO implementation (2012). http://www.mcs.anl.gov/research/projects/romio/
21. Ross, R.: Parallel I/O benchmarking consortium. http://www-unix.mcs.anl.gov/rross/pio-benchmark/html/
22. Schmuck, F., Haskin, R.: GPFS: a shared-disk file system for large computing clusters. In: FAST'02, pp. 231–244. USENIX, Berkeley (2002)
23. Tatebe, O., Morita, Y., Matsuoka, S., Soda, N., Sekiguchi, S.: Grid datafarm architecture for petascale data intensive computing. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02), Washington, DC, USA, p. 102 (2002)
24. Thakur, R., Choudhary, A.: An extended two-phase method for accessing sections of out-of-core arrays. Sci. Program. **5**(4), 301–317 (1996)

25. Thakur, R., Gropp, W., Lusk, E.: An abstract-device interface for implementing portable paralle-I/O interfaces. In: Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96) (1996). http://www.mcs.anl.gov/home/thakur/adio.ps
26. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation, pp. 182–189 (1999)
27. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI–IO portably and with high performance. In: Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems, pp. 23–32. ACM Press, New York (1999)
28. The National Center for SuperComputing. HDF5 home page. http://hdf.ncsa.uiuc.com/HPD5/
29. Vilayannur, M., Nath, P., Sivasubramaniam, A.: Providing tunable consistency for a parallel file store. In: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST'05), Berkeley, CA, USA, pp. 2 (2005)
30. Wong, P., Van der Wijngaart, R.F.: NAS parallel benchmarks I/O, version 2.4. Tech. rep. NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division
31. Yu, W., Vetter, J.: ParColl: partitioned collective I/O on the cray XT. In: International Conference on Parallel Processing (ICPP'08), Portland, OR (2008)
32. Yu, W., Vetter, J., Canon, R., Jiang, S.: Exploiting lustre file joining for effective collective I/O. In: 7th Int'l Conference on Cluster Computing and Grid (CCGrid'07), Rio de Janeiro, Brazil (2007)
33. Yu, W., Vetter, J., Oral, H.: Performance characterization and optimization of parallel I/O on the cray XT. In: 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08), Miami, FL (2008)
34. Zheng, F., et al.: Predata—preparatory data analytics on peta-scale machines. In: IPDPS, Atlanta, GA (2010)

Yuan Tian Yuan Tian is a Postdoctoral Research with Joint Institute of Computational Science of University of Tennessee. Yuan Tian holds a Ph.D. in Computer Science from Auburn University. She earned her master's degree in Computer Science in 2009 from Auburn University and her bachelor's degree from Chengdu University of Technology, Chengdu, China in 2002. She also worked as software engineer in both China and Japan. Her current research focus is data management and workflow systems for large-scale scientific applications.

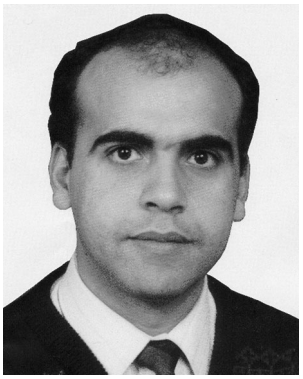
Cong Xu is a Ph.D. student of Parallel Architecture and System Laboratory (PASL) in the Department of Computer Science at Auburn University. Xu earned his master's degree in Computer Science from Auburn University in 2012. His research interests include High Performance Computing, Parallel Processing, and Data Analytics.

Weikuan Yu is currently an Associate Professor in the Department of Computer Science and Software Engineering at Auburn University. Prior to joining Auburn, he served as a Research Scientist for two and a half years at Oak Ridge National Laboratory (ORNL) until January 2009. Yu is also a Joint Faculty at ORNL. He earned his PhD in Computer Science from the Ohio State University in 2006. Yu also holds a master's degree in Developmental Biology from the Ohio State University and a Bachelor degree in Genetics from Wuhan University, China. At Auburn University, Yu leads the Parallel Architecture and System Laboratory (PASL) for research and development on big data analytics, parallel and distributing computing, storage and file systems, as well as interdisciplinary topics on computational biology. Yu is a member of AAAS, ACM, and IEEE.

Jeffrey S. Vetter Jeffrey S. Vetter is a computer scientist in the Computer Science and Mathematics Division (CSM) of Oak Ridge National Laboratory (ORNL), where he leads the Future Technologies Group and directs the Experimental Computing Laboratory. Dr. Vetter is also a Joint Professor in the College of Computing at the Georgia Institute of Technology, where he earlier earned his PhD. He joined ORNL in 2003, after four years at Lawrence Livermore National Laboratory. Vetter's interests span several areas of highend computing—encompassing architectures, system software, and tools for performance and correctness analysis of applications.

Scott Klasky Scott A. Klasky is the group leader for Scientific Data in the Computer Science and Mathematics Research Division at the Oak Ridge National Laboratory. He holds a Ph.D. in Physics from the University of Texas at Austin (1994), and has previously worked at the University of Texas at Austin, Syracuse University, and the Princeton Plasma Physics Laboratory. Dr. Klasky is a world expert in scientific computing and scientific data management, co-authoring over 150 papers. He is also the leader of the ADIOS project, http://www.olcf.ornl.gov/center-projects/adios/.

Honggao Liu Honggao Liu is the Deputy Director of Center for Computation and Technology (CCT) at Louisiana State University (LSU). He was the Director of High Performance Computing (HPC) at LSU and Louisiana Optical Network Initiative (LONI) from 2008–2011. Liu was the Principal Investigator on the LSU/LONI's NSF funded TeraGrid effort. Liu has overseen all HPC activities and led the HPC development efforts at LSU and LONI, and has been instrumental in establishing HPC at LSU as a nationally recognized facility. Liu conducted research on multiphase reactive polymer flow in porous media and reservoir simulations during 1997–2002. He received his Ph.D. in Chemical Engineering from LSU in 2002 and holds a B.S. in Chemical Engineering from Xi'an Jiaotong University and two M.S. degrees in Chemical Engineering from Tianjin University and from LSU.



Saad Biaz SaadBiaz received a Ph.D. in Computer Science in 1999 from Texas A&M University and a Ph.D. in Electrical Engineering in 1989 from the University Henri Poincaré in Nancy (France). He is presently an Associate Professor of Computer Science and Software Engineering at Auburn University. He has held faculty positions at the EcoleSupérieure de Technologie de Fès and Al Akhawayn University in Ifrane (Morocco). His current research is in the areas of distributed systems, wireless networking, mobile computing, and particularly on autonomous flight of Unmanned Aircraft Systems. His research is funded by the U.S. National Science Foundation. SaadBiaz is a recipient in 1995 of the Excellence Fulbright Scholarship. Saad has served on the committees of several conferences and as editor for several journals. For more information, please visit http://www.eng.auburn.edu/users/sbiaz.