An Efficient and Flexible Parallel I/O implementation for the CFD General Notation System

Kyle Horne, Nate Benson
Center for High Performance Computing, Utah State University
Thomas Hauser
Academic & Research Computing, Northwestern University
horne.kyle@gmail.com, nate.benson@usu.edu, t-hauser@northwestern.edu

Abstract—**One important, often overlooked, issue for large, three dimensional time-dependent computational fluid dynamics (CFD) simulations is the input and output performance of the CFD solver, especially for large time-dependent simulations. The development of the CFD General Notation System (CGNS) has brought a standardized and robust data format to the CFD community, enabling the exchange of information between the various stages of numerical simulations. Application of this standard data format to large parallel simulations is hindered by the reliance of most applications on the CGNS Mid-Level Library. This library has only supported serialized I/O. By moving to HDF5 as the recommended low level data storage format, the CGNS standards committee has created the opportunity to support parallel I/O. In our work, we present the parallel implementation of the CGNS Mid-Level Library and an I/O request-queuing approach to overcome some limitations of HDF5. We also present detailed benchmarks on a parallel file system for typical structured and unstructured CFD application I/O patterns.**

I. Introduction

Linux clusters can provide a viable and more cost-effective alternative to conventional supercomputers for the purposes of computational fluid dynamics (CFD). In some cases, the Linux supercluster is replacing the conventional supercomputer as a large-scale, shared-use machine. In other cases, smaller clusters are providing dedicated platforms for CFD computations. One important, often overlooked, issue for large, three-dimensional time-dependent simulations is the input and output performance of the CFD solver. The development of the CFD General Notation System (CGNS) (see [1], [2], [3]) has brought a standardized and robust data format to the CFD community, enabling the exchange of information between the various stages of numerical simulations. CGNS is used in many commercial and open source tools such as Fluent, Fieldview, Plot3D, OpenFOAM, Gmsh and VisIt. Unfortunately, the original design and implementation of the CGNS interface did not consider parallel applications and, therefore, lack parallel access mechanisms. By moving to HDF5 as the recommended low level data storage format, the CGNS standards committee has created the opportunity to support parallel I/O. In our work, we present the parallel implementation of the CGNS Mid-Level Library and detailed benchmarks on parallel file systems for typical structured and unstructured CFD applications.

II. I/O for Computational Fluid Dynamics Simulations

Current unsteady CFD simulations demand not only a great deal of processing power, but also large amounts of data storage. Even a relatively simple unsteady CFD simulation can produce terabytes of information at a tremendous rate, and all this data must be stored and archived for analysis and post processing. Most CFD programs have one of the following three approaches implemented:

1) One process is responsible for I/O. This means that all I/O activity is channeled through one process and the program cannot take advantage of a parallel file system since this process is inherently serial. One CFD code using this approach is OVERFLOW [4]developed by NASA.
2) A large number of parallel CFD codes are programmed to access their input and output data in a one-file-per-process model. This approach is simple but has two major disadvantages. First, in order to get access to the full solution for visualization or post-processing, one must reassemble all files into one solution file. This is an inherently serial process and can take a long time. Second, the simulation program may only be restarted with the same number of processes (see Fig. 1a) which may reduce the potential throughput for a user on a busy supercomputer. One CFD code which takes this approach is OpenFOAM[5].
3) The most transparent approach for a user is writing from all process in parallel into one shared file. The current MPI-2 standard [6]defines in chapter nine the MPI-I/O extension to allow parallel access from memory to files. Parallel I/O is more difficult to implement and may perform slower compared to approach 2, when the I/O requests are not coordinated properly. An overview of the shared file I/O model is provided in figure (1 b).

It is highly desirable to develop a set of parallel APIs for accessing CGNS files that employ appropriate parallel I/O techniques. Programming convenience and support for all types of CFD approaches, from block-structured to unstructured, is a very important design criteria, since scientific users may desire to spend minimal effort on dealing with I/O operations. This new parallel I/O API, together with a queuing-approach to overcome some limitations of HDF5 for a generally distributed multi-block grid, and the benchmarks on the parallel file system is the core of our storage challenge entry.

A. The CGNS system

The specific purpose of the CFD General Notation System (CGNS) project is to provide a standard for recording and recovering computer data associated with the numerical solution of the equations of fluid dynamics. The intent is to facilitate the exchange of Computational Fluid Dynamics (CFD) data between sites, between applications codes, and across computing platforms, and to stabilize the archiving of CFD data.

The CGNS system consists of three parts: (1) the Standard Interface Data Structures, SIDS, (2) the Mid-Level Library (MLL), and (3) the low-level storage system, currently using ADF and HDF5.

The "Standard Interface Data Structures" specification constitutes the essence of the CGNS system. While the other elements of the system deal with software implementation issues, the SIDS specification defines the substance of CGNS. It precisely defines the intellectual content of CFD-related data, including the organizational structure supporting such data and the conventions adopted to standardize the data exchange process. The SIDS are designed to support all types of information involved in CFD analysis. While the initial target was to establish a standard for 3D-structured multi-block compressible Navier-Stokes analysis, the SIDS extensible framework now includes unstructured analysis, configurations, hybrid topology and geometry-to-mesh association.

A CGNS file consists of nodes with names, associated meta data and data. Figure 2 shows the abstract layout of a CGNS file. The nodes of a CGNS tree may or may not be in the

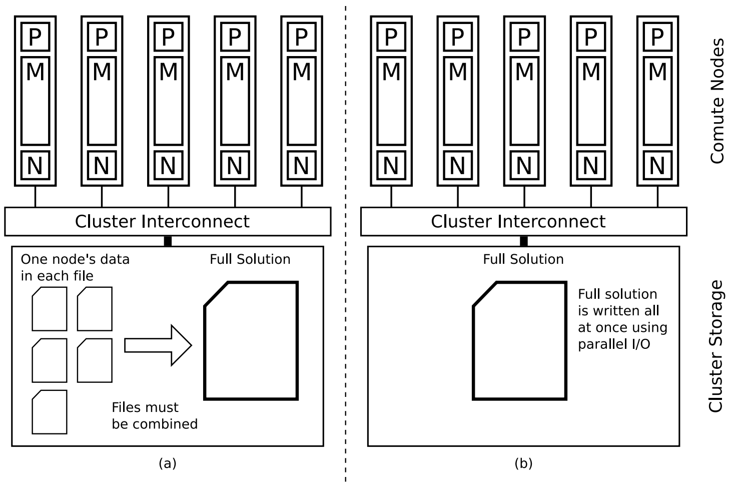


Figure 1: Storage topology of a modern computer cluster in two configurations. Configuration (a) shows the traditional CFD approach to I/O, where each node writes a separate file to disk. These files must be combined to obtain the full solution in a process which can be quite time consuming. Configuration (b) shows the object of this proposal, a CFD I/O method in which the partial solution from each node is written into the appropriate location within a single solution file for the whole problem.

same file. CGNS provides the concept of links (similar to soft links in a file system) between nodes. This enables users to write the grid coordinates in one file and then add links to the solution written in several files to the grid file. For a visualization system these linked files would look like one big file containing the grid and solution data.

Although the SIDS specification is independent of the physical file formats, its design was targeted towards implementation using the ADF Core library, but one is able to define a mapping to any other data storage format. Currently in CGNS 3.0, HDF5 is the recommended storage format, but access to the older ADF format is transparent to the users of the MLL.

1) ADF data format: The "Advanced Data Format" (ADF) is a concept defining how data is organized on the storage media. It is based on a single data structure called an ADF node, designed to store any type of data. Each ADF file is composed of at least one node called the "root". The ADF nodes follow a hierarchical arrangement from the root node down. This data format was the main format up to the 3.0 release.

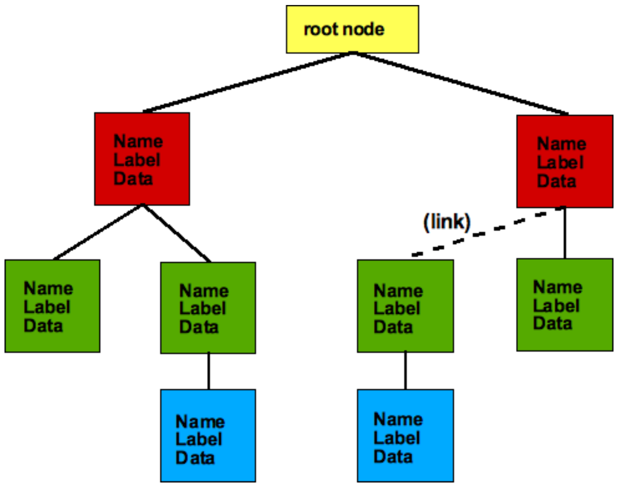


Figure 2: Typical layout of a CGNS file consisting of a tree structure with meta data and data stored under each tree node. The dashed link shows that CGNS also supports the concept of links to other nodes which reside in other files.

2) HDF5 data format: In the current beta version of CGNS 3.0, HDF5 is the default low-level storage format. The format of an HDF5 file on disk encompasses several key ideas of the HDF4 and AIO file formats as well as addresses some shortcomings therein.

The HDF5 library uses these low-level objects to represent the higher-level objects that are then presented to the user or to applications through APIs. For instance, a group is an object header that contains a message that points to a local heap and to a B-tree which points to symbol nodes. A data set is an object header that contains messages that describe data type, space, layout, filters, external files, fill value, etc. with the layout message pointing to either a raw data chunk or to a B-tree that points to raw data chunks.

III. Parallel I/O for the CGNS system

To facilitate convenient and high-performance parallel access to CGNS files, we have defined a new parallel interface and provide a prototype implementation. Since a large number of existing users are running their applications over CGNS, our parallel CGNS design retains the original MLL API and introduces extensions which are minimal changes from the original API. Currently our pCGNS library is intended as a companion library to the CGNS Mid-Level Library, since pCGNS only supports the writing of grids and data arrays. The much smaller meta data should still be written by a single process using the standard CGNS library. The parallel API is distinguished from the original serial API by prefixing the C function calls with "cgp_" instead of "cg_" as in the MLL API. Tabulated in the appendix are the functions defined in *pcgnslib.h* which constitute the pCGNS API as it currently exists. The functions chosen to be implemented in the initial version of the library were selected on the basis of two criteria. The first was the need for the function to provide the most basic functionality of the library. The second criteria was the possible benefit from parallelization of the function. The result of using these priorities is that the pCGNS library can write the basic components of a CGNS file needed for the standard CGNS tools to function, but still provides routines for parallel file access on the largest of the data sets that need to be written to the file.

The CGNS Mid-level Library supports writing files in HDF5 as of version 2.5, but the support is not native. The main library calls are written to the ADF interface and HDF5 support is achieve through an ADF to HDF5 compatibility layer. While this solution works well for serial operation, since ADF has no support for parallel I/O, there is no way to access HDF5's parallel abilities through this compatibility layer. Therefore the decision was made that the new parallel routines would be written directly to HDF5 with no connections to ADF.

A limited prototype implementation was done by Hauser and Pakalapati[7], [8]. In an effort to improve performance and better integrate the parallel extension in to the CGNS Mid-level Library, new routines are being written which access HDF5 directly and take full advantage of the collective I/O support in HDF5 1.8. Much of the work is to provide the basic structures in the file required for the standard CGNS library to be able to read files created by the new extension routines, but the most important parts are in writing large data sets describing CFD grids and solutions on those grids in parallel. This is done with functions which queue the I/O until a flush function is called, at which point the write commands are analyzed and executed with HDF5 calls. This is done to provide MPI-IO sufficient data to effectively recognize the collective and continuous nature of the data being written.

The parallel capabilities of HDF5 allow a single array to be accessed by multiple processes simultaneously. However, it is currently not possible to access multiple arrays from a single process simultaneously. This prevents the implementation of parallel I/O in pCGNS to allow for the most generic case of partition and zone allocation.

To resolve this deficiency, the pCGNS library implements an I/O queue. An application can queue I/O operations and then flush them later. This allows the flush routine to optimize the operations for maximum bandwidth. Currently, the flush routine simply executes the I/O operations in the order they were added to the queue, but based on the results of benchmarking the flush routine will be modified to limit the requests which cause multiple processes to access a single array, since that operation is slower than the others.

IV. Related Work

Considerable research has been done on data access for scientific applications. The work has focused on data I/O performance and data management convenience. Three projects, MPI-IO, HDF5 and parallel netCDF (PnetCDF) are closely related to this research.

MPI-IO is a parallel I/O interface specified in the MPI-2 standard. It is implemented and used on a wide range of platforms. The most popular implementation, ROMIO [9] is implemented portably on top of an abstract I/O device layer [10], [11] that enables portability to new underlying I/O systems. One of the most important features in ROMIO is collective I/O operations, which adopt a two-phase I/O strategy [12], [13], [14], [15] and improve the parallel I/O performance by significantly reducing the number of I/O requests that would otherwise result in many small, noncontiguous I/O requests. However, MPI-IO reads and writes data in a raw format without providing any functionality to effectively manage the associated meta data, nor does it guarantee data portability, thereby making it inconvenient for scientists to organize, transfer, and share their application data.

HDF is a file format and software, developed at NCSA, for storing, retrieving, analyzing, visualizing, and converting scientific data. The most popular versions of HDF are HDF4 [16] and HDF5 [17]. Both versions store multidimensional arrays together with ancillary data in portable, self-describing file formats. HDF4 was designed with serial data access in mind, whereas HDF5 is a major revision in which its API is completely redesigned and now includes parallel I/O access. The support for parallel data access in HDF5 is built on top of MPI-IO, which ensures its portability. This move undoubtedly inconvenienced users of HDF4, but it was a necessary step in providing parallel access semantics. HDF5 also adds several new features, such as a hierarchical file structure, that provide application programmers with a host of options for organizing how data is stored in HDF5 files. Unfortunately this high degree of flexibility can sometimes come at the cost of high performance, as seen in previous studies [18], [19].

Parallel-NetCDF [20] is a library that provides high-performance I/O while still maintaining file-format compatibility with Unidata's NetCDF [21]. In the parallel implementation the serial netCDF interface was extended to facilitate parallel access. By building on top of MPI-IO, a number of interface advantages and performance optimizations were obtained. Preliminary test results show that the somewhat simpler netCDF file format coupled with a parallel API combine to provide a very high-performance solution to the problem of portable, structured data storage.

V. Benchmarking Systems - Architectural Details

The HPC@USU infrastructure consists of several clusters connected through a ProCurve 5412zl data center switch to shared NFS and parallel storage. The entire configuration is shown in the appendix. The details of the compute and storage solutions are described in the following subsections.

Our Wasatch cluster consists of 64 compute nodes. Each compute node has two quad-core AMD Opteron 2376 running at 2.3 GHz and 16 GByte of main memory. The cluster has two interconnects: A DDR infiniband network connects all nodes and is the network for parallel simulations code, and a network consisting of two bonded GBit networks, which connects the cluster directly to the parallel storage system.

All clusters at Utah State University's Center for High Performance Computing have access to the same network-attached storage (NAS) so that researchers can freely utilize all of the available computing resources. To accomplish this, the large ProCurve data center switch is the core switch connecting all storage and administrative connections for the clusters. These connections are independent of the message-passing fabric of the clusters, which are unique to each machine. The nodes of Wasatch, the newest addition to our center, are all connected to the core switch directly. Thus they enjoy the highest speed access to the NAS.

The NAS system consists of four shelves of the Panasas ActiveScale Storage Cluster with 20 TB each for a total raw capacity of 80 TB and a usable capacity of about 60 TB. Panasas storage devices are network-attached Object-based Storage Devices (OSDs). Each OSD contains two Serial-ATA drives and sufficient intelligence to manage the data that is locally stored. Meta data is managed in a meta data server, a computing node separate from the OSDs but residing on the same physical network. Clients communicate directly with the OSD to access the stored data. Because the OSD manages all low-level layout issues, there is no need for a file server to intermediate every data transaction. To provide data redundancy and increase I/O throughput, the Panasas ActiveScale File System (PanFS) stripes the data of one file system across a number of objects (one per OSD). When PanFS stripes a file across 10 OSDs attached to a Gigabit Ethernet (GE) network, PanFS can deliver up to 400 MB/sec split among the clients accessing the file.

Processes from our compute nodes can invoke POSIX read/write function or call through MPI-I/O or parallel HDF5 built on top of MPI-I/O. For all of our benchmarks we are using OpenMPI 1.3.3 with parallel I/O for the PanFS enabled. The parallel extension to the CGNS library, called pCGNS, uses the HDF5 library for all file access. HDF5 in turn uses MPI-I/O for all parallel access to files. This abstraction of the file system into layers of libraries is crucial if a scientific code is to be run at many sites with different hardware and software configurations. In addition to these libraries, custom test programs are being developed to provide CFD-like I/O requests to test the performance of the pCGNS library.

VI. Parallel I/O Characterization

We ran two benchmarks to test the performance of our implementation on current parallel file systems: IOR [22] and a self developed parallel CGNS benchmark program which covers most cases of parallel I/O for a structured or unstructured CFD code. All benchmarks were run five times and the averages of these runs together with the standard deviation of the results are plotted in the figures in this section.

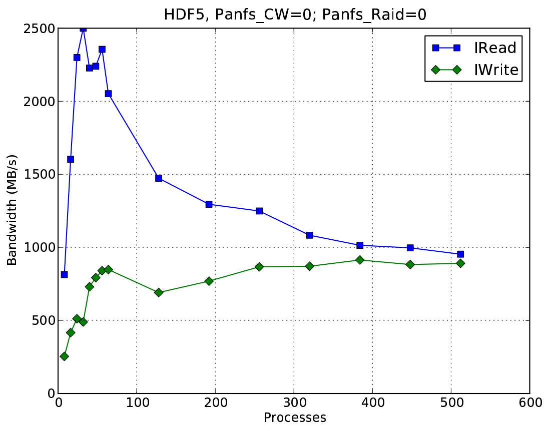
A. *IOR benchmark results*

The IOR benchmarks were run with parameters to test the performance achievable on our benchmarking system with HDF5. In Fig. 3 the results for read and write are shown. IOR was configured such that each process access 512 MB in a single file on disk using the HDF5 interface. These operations were done using independent I/O to match the settings used to benchmark pCGNS.

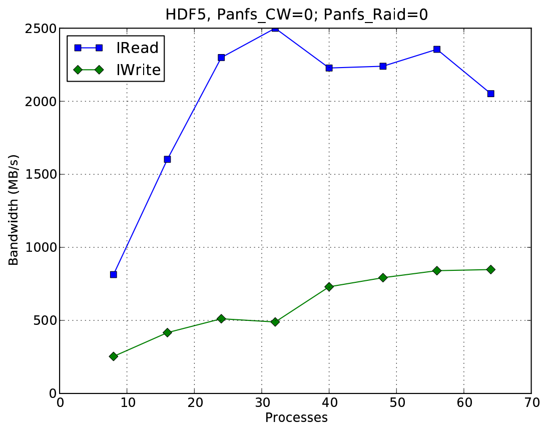
B. *Benchmarks of the parallel CGNS implementation*

To test the performance of the new library, benchmark test programs have been written which make CFD-like I/O requests to the library and measure how long each task takes to complete. Five categories of benchmark were selected to be run on the pCGNS library. The data distribution for each of the benchmark categories are shown in Figure (4).

Each scenario of data distribution models a general method of splitting up data among the processes of a parallel program. The scenarios will be hereafter referred to by the designation



(a) Data for the entire cluster, scaling up to 8 processes per node



(b) Data for the entire cluster, only one process per node

Figure 3: Results of the IOR benchmark on our benchmarking system.

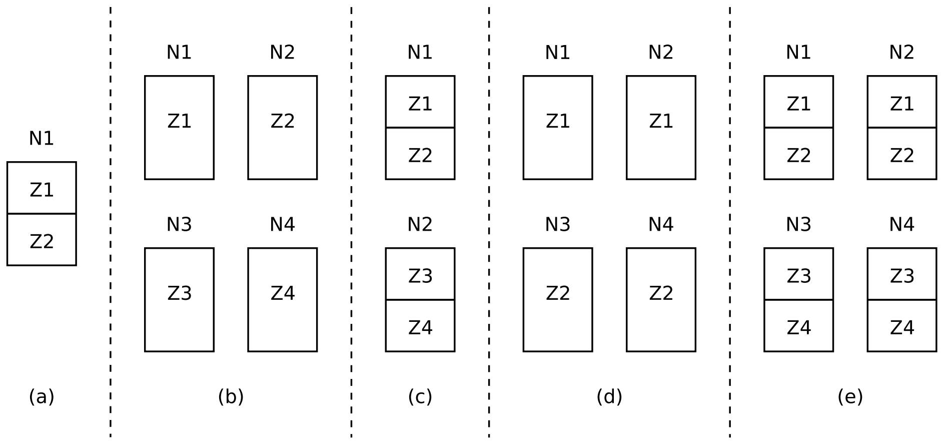


Figure 4: The five data distribution scenarios for benchmarking. Benchmark (a) has two separate zones on one node, (b) has one zone per node on four nodes, (c) shows four zones on two node, (d) shows two zones, each distribute between two nodes, on a total of four nodes, and finally benchmark (e) shows four zones on four nodes with zones spread across multiple nodes.

each scenario bears in the figure. In all of the benchmarks, the file size is kept constant, since during practical use of the library the file size will generally not be a function of the number of nodes used for a simulation, but rather a function of the simulation itself.

The data size in the benchmarks shown in the following sections is the size in MB of an individual array in the CGNS file. The CGNS file for each benchmarks contains seven three-dimensional double precision data arrays: the x, y, z coordinates of the grid, the three velocity components and a temperature field. In addition an integer field containing the cell connectivity is stored. A data size of 1024 means that seven 1024 MB arrays and one 512 MB array have been written to the file.

1) I/O performance for a single node with multiple zones: This *type-a* benchmark, explores the performance of the library when multiple zones are stored on a single node. This is the only benchmark in which multiple processes are run on a single node. Normally this is avoided since having multiple processes on a node forces those processes to share a single network connection, thus affecting the measurement of the parallel performance of the library. In the *type-a* benchmark, however, this limitation is desired since it allows the single process with multiple zones to have access to the same storage bandwidth as the aggregate bandwidth of multiple processes on the node. The results are shown in Fig. 5.

2) I/O performance for one not partitioned zone per process: The *type-b* benchmarks measure the performance of pCGNS in a configuration very close to the benchmarks run by IOR. In this scenario, each node runs a single process, and each process only writes a single zone. Because each zone has its own data array in the file, and each zone is written by a single process, each process writes to a single data array in the file. This benchmark resembles a structured or unstructured multi-block code where each process owns a single zone.

3) I/O performance for multiple not partitioned zones per process: The *type-c* benchmarks test the ability of the library to write multiple zones from each node in parallel. Because the file size is kept constant, more zones per node result in smaller data arrays written to the file in each I/O operation. Since this is expected to affect performance, multiple cases are run with different numbers of zones per node. This benchmark resembles a structured or unstructured multi-block code where each process owns several zones.

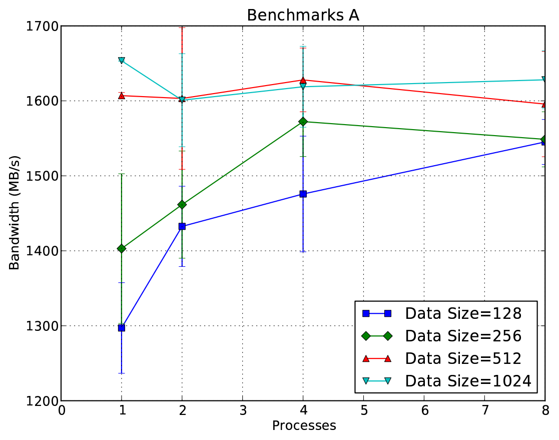
4) I/O performance for one partitioned zone per process with multiple zones: Type-d benchmarks test the ability of the library to write a single zone from multiple nodes. This causes multiple processes to access a single data array simultaneously, which is quite different from multiple processes accessing multiple data arrays. In this scenario, each node only writes to a single zone, but each zone is written to by multiple processes. This is analogous to a CFD code in which each processes must share a single zone with another process.

5) I/O performance for multiple partitioned zone per process with multiple zones: The last scenario, *type-e*, tests the most general access pattern, in which multiple processes access each zone and multiple zones are accessed by each process. This access pattern would result from the zone definitions and parallel partitioning processes being completely independent one from another. Thus, this is the most general of the benchmarks and resembles a CFD code in which each process must share several zones with other processes.

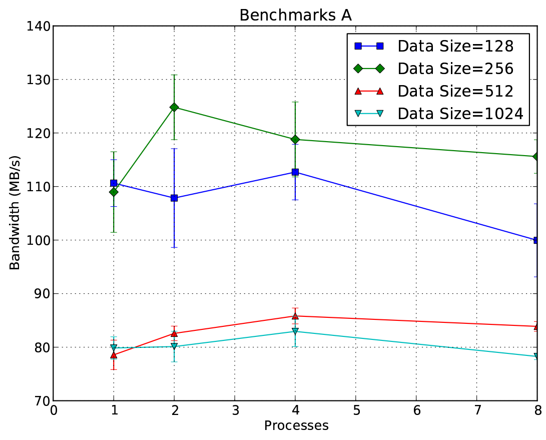
C. Discussion

It can be observed from the benchmarks of pCGNS that the write process is very dependent on the amount of data being transferred. Using larger data sets results in significantly improved performance when writing to the file, but does not affect the read performance. This phenomenon can be explained by the extra operations that must be executed in the case of writing to the file, such as disk allocation for the new data, etc. In the case of larger data sets, the overhead is much smaller compared to the time required to transfer the data itself, and results in better performance. Read operations have no need for disk allocation, and are not affected. This also explains the performance seen in the *type-c* benchmarks, since the addition of more zones for each node results in more disk allocations.

The optimal performance of the pCGNS library is seen in the *type-b* benchmark. When running on 24 nodes, the code writes the data at more than 450 MB/s, which compares very favorably with the 500 MB/s seen in the IOR benchmark for the same number of processes and nodes. The variance between pCGNS and IOR is not large, and is likely caused by better organization of the I/O and disk allocation in the IOR benchmark, since pCGNS does not know ahead of time what the I/O and data will look like. The lesser performance of pCGNS with larger numbers of nodes is likely due to the decreasing size of the data per process, whereas IOR keeps the data per process constant.

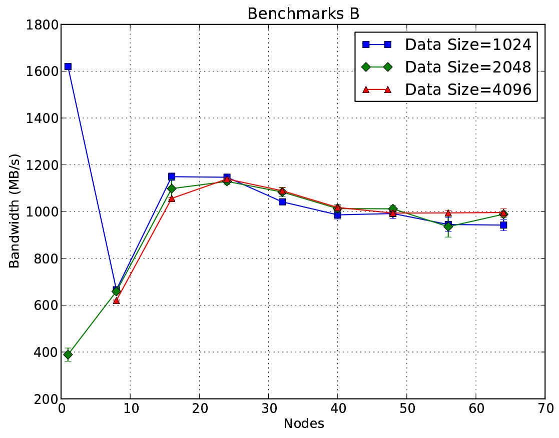


(a) Read bandwidth.

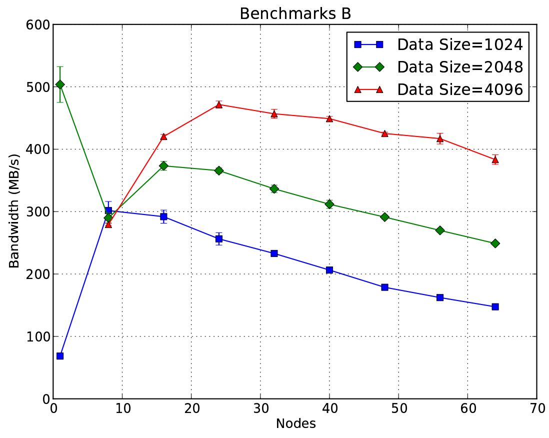


(b) Write bandwidth.

Figure 5: Multiple zones on one single node with up to eight cores.



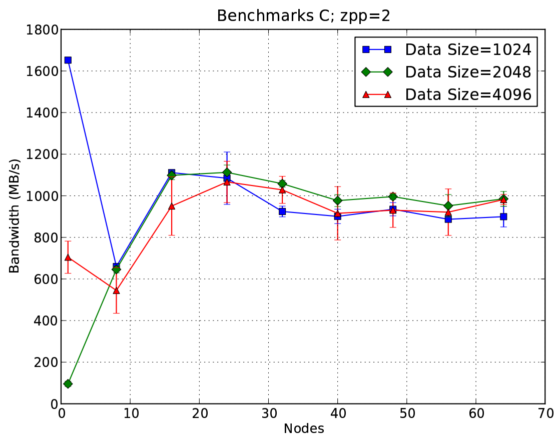
(a) Read bandwidth



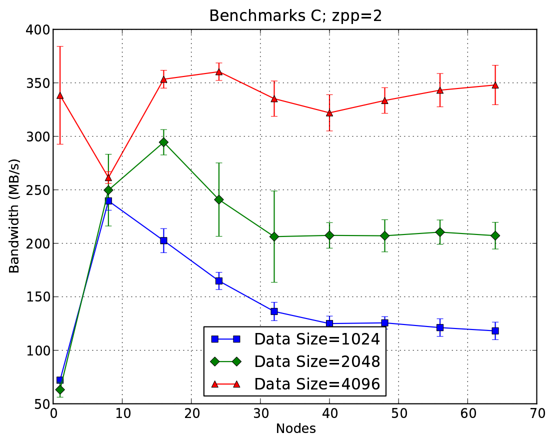
(b) Write bandwidth

Figure 6: I/O performance for one non-partitioned zone per process

The read performance of pCGNS does not compare as favorably with the IOR benchmark as the write performance. The peak read performance from pCGNS is seen with 16 nodes, where it achieves a read bandwidth of ~1100 MB/s, whereas IOR sees ~1600 MB/s. After this point, pCGNS's performance drops off while IOR's continues to improve up to a peak of 2500 MB/s. The post-peak performance is again due to the increasing size of data with IOR and constant data size with pCGNS. The discrepancy in the range of lower node counts is more indicative of the performance difference between IOR and pCGNS. This difference may be caused by the fact that IOR runs a single read request against the storage, whereas the pCGNS benchmark runs seven. With data rates so high compared to the size of the files used, the difference due to added requests by pCGNS may be the culprit responsible for the performance drop.

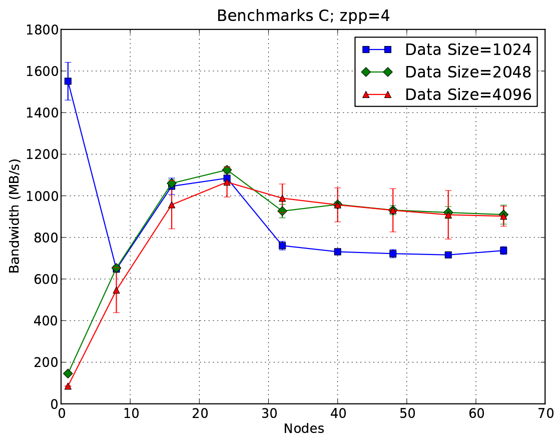


(a) Read bandwidth

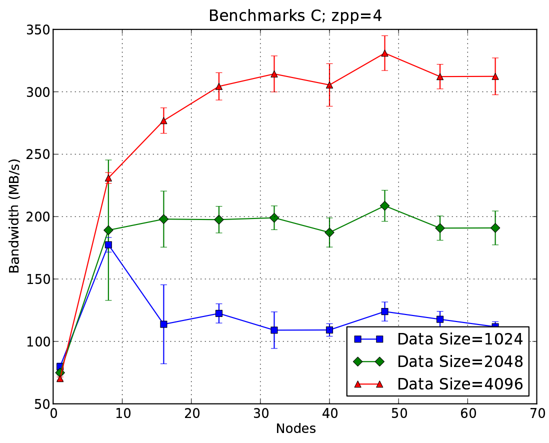


(b) Write bandwidth

Figure 7: I/O performance for two non-partitioned zones per process



(a) Read bandwidth



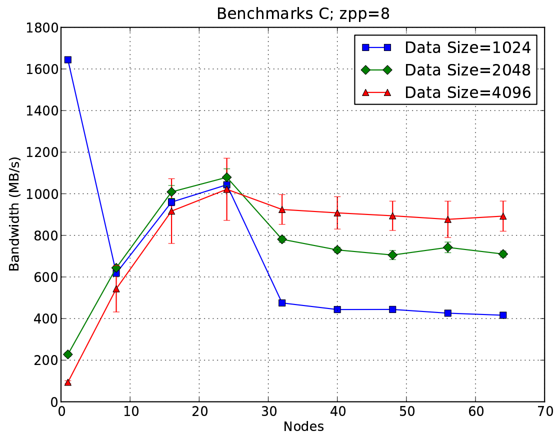
(b) Write bandwidth

Figure 8: I/O performance for four non-partitioned zones per process

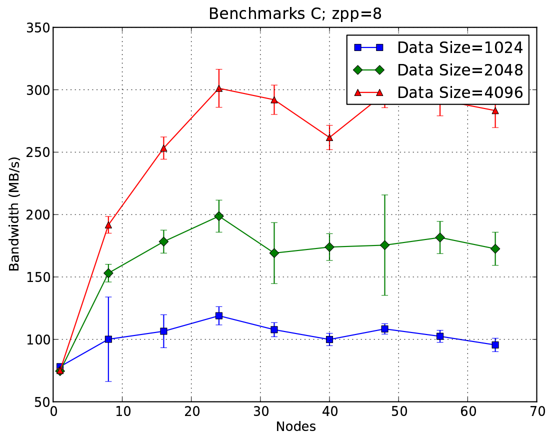
VII. Summary

The presented solution to the problem of high performance parallel CFD I/O provides an extension to the existing CGNS standard Mid-Level Library API. This API provides the CFD application developer with a general way to write mesh and solution data to the CGNS file in parallel for multi-block structured and unstructured grids. To get the best performance for CFD codes with multiple block distribute in a general way over an existing set of node, I/O requests are queued until a flush function is called, at which point the write commands are analyzed, reorganized for best performance, and executed with HDF5 calls. This is done to provide MPI-IO sufficient data to effectively recognize the collective and continuous nature of the data being written.

This challenge entry has provided benchmark data on a four shelf Panasas ActiveScale Storage Cluster and a 64 node compute cluster, demonstrating how much I/O performance for an unsteady structured and unstructured multi-

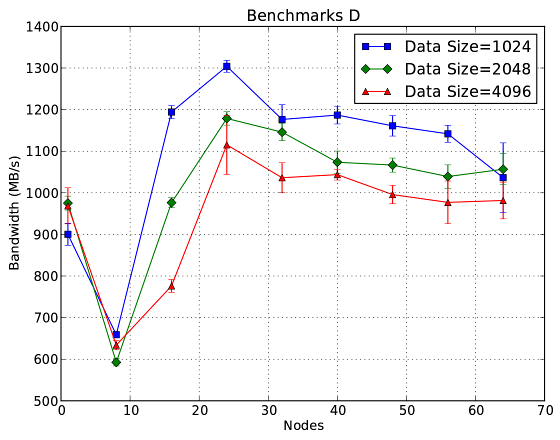


(a) Read bandwidth

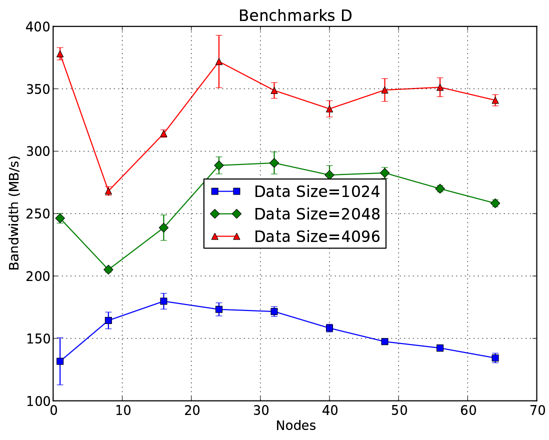


(b) Write bandwidth

Figure 9: I/O performance for eight non-partitioned zones per process



(a) Read bandwidth



(b) Write bandwidth

Figure 10: I/O performance for one partitioned zone per process

block CFD application can be expected.

VIII. Acknowledgements

The authors would like to thank Bruce Wedan for making the initial CGNS 3.0 implementation available, and his contribution to CGNS. Part of the work was support by the National Science Foundation under Grant CTS-0321170 and USDA-CREES under the project “High performance computing Utah”.

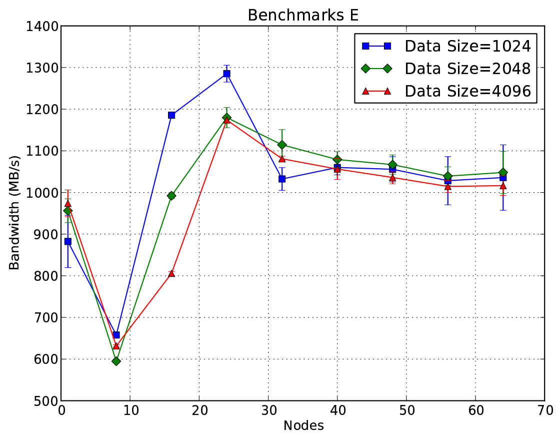
References

[1] Poirier, D., Allmaras, S. R., McCarthy, D. R., Smith, M. F., and Enomoto, F. Y., “The CGNS System,” *AIAA Paper 98-3007*, 1998. 1

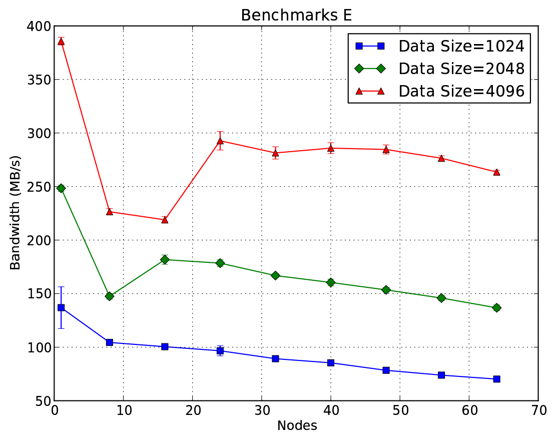
[2] Poirier, D. M. A., Bush, R. H., Cosner, R. R., Rumsey, C. L., and McCarthy, D. R., “Advances in the CGNS Database Standard for Aerodynamics and CFD,” *AIAA Paper 2000-0681*, 2000. 1

[3] Legensky, S. M., Edwards, D. E., R. H. Bush, D. M. A. P., Rumsey, C. L., Cosner, R. R., and Towne, C. E., “CFD General Notation System (CGNS): Status and Future Directions,” *AIAA Paper 2002-0752*, 2002. 1

[4] Jespersen, D., Pulliam, T., and Buning, P., “Recent Enhancements to OVERFLOW,” *InAIAA Proceedings, Paper No. 970644*, 1997, pp. 97–0644. 2



(a) Read bandwidth



(b) Write bandwidth

Figure 11: I/O performance for two partitioned zones per process

[5] Weller, H., Tabor, G., Jasak, H., and Fureby, C., "A tensorial approach to computational continuum mechanics using object orientied techniques," *Computers in Physics*, Vol. 12, No. 6, 1998, pp. 620 – 631. 2

[6] Interface, M. P., "MPI-2: Extensions to the Message-Passing Interface," 1996. 2

[7] Hauser, T., "Parallel I/O for the CGNS system," *AMERICAN INSTITUTE OF AERONAUTICS AND ASTRONAUTICS*, , No. 1090, 2004. 4

[8] Parimala D. Pakalapati, T. H., "Benchmarking Parallel I/O Performance for Computational Fluid Dynamics Applications," *AIAA Aerospace Sciences Meeting and Exhibit*, Vol. 43, 2005. 4

[9] Thakur, R., Ross, R., Lusk, E., and Gropp, W., "Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation," Tech. Rep. Technical Memorandum No. 234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised January 2002. 5

[10] Thakur, R., Gropp, W., , and Lusk, E., "An Abstract-Device interface for Implementing Portable Parallel-I/O Interfaces(ADIO)," *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996, pp. 180–187. 5

[11] Thakur, R., Gropp, W., , and Lusk, E., "On Implementing MPI-I/O Portably and with High Performance," *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, May 1999, pp. 23–32. 5

[12] Rosario, J., Bordawekar, R., , and Choudhary, A., "Improved Parallel I/O via a Two-Phase Run-time Access Strategy," *IPPS '93 Parallel I/O Workshop*, February 1993. 5

[13] Thakur, R., Bordawekar, R., Choudhary, A., and Ponnusamy, R., "PASSION Runtime Library for Parallel I/O," *Scalable Parallel Libraries Conference*, October 1994. 5

[14] Thakur, R. and Choudhary, A., "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming*, Vol. 5, No. 4, 1996, pp. 301–317. 5

[15] Thakur, R., Gropp, W., and Lusk, E., "Data Sieving and Collective I/O in ROMIO," *Proceeding of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999, pp. 182–189. 5

[16] "HDF4 Home Page," The National Center for Supercomputing Applications. http://hdf.ncsa.uiuc.edu/hdf4.html. 5

[17] "HDF5 Home Page, The National Center for Supercomputing Applications," http://hdf.ncsa.uiuc.edu/HDF5/. 5

[18] Li, J., Liao, W., Choudhary, A., and Taylor, V., "I/O Analysis and Optimization for an AMR Cosmology Application," *Proceedings of IEEE Cluster 2002*, Chicago, IL, September 2002. 5

[19] Ross, R., Nurmi, D., Cheng, A., and Zingale, M., "A Case Study in Application I/O on Linux Clusters," *Proceedings of SC2001*, Denver, CO, November 2001. 5

[20] Li, J., keng Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., and Siegel, A., "Parallel netCDF: A High-Performance Scientific I/O Interface," *Proceedings of the Supercomputering 2003 Conference*, Phoenix, AZ, November 2003. 5

[21] Rew, R. and Davis, G., "The Unidata netCDF: Software for Scientific Data Access," *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, Anaheim, CA, February 1990. 5

[22] "IOR HPC Benchmark," Soureforge http://sourceforge.net/projects/ior-sio/. 6

An Efficient and Flexible Parallel I/O implementation for the CFD General Notation System

Kyle Horne, Nate Benson
Center for High Performance Computing, Utah State University
Thomas Hauser
Associate Director Research Computing
Northwestern University

UtahState University
CENTER FOR HIGH PERFORMANCE COMPUTING

Outline

- CGNS
- Parallel I/O for CGNS
- Benchmarking system
- Benchmarking results
  - IOR
  - Parallel CGNS

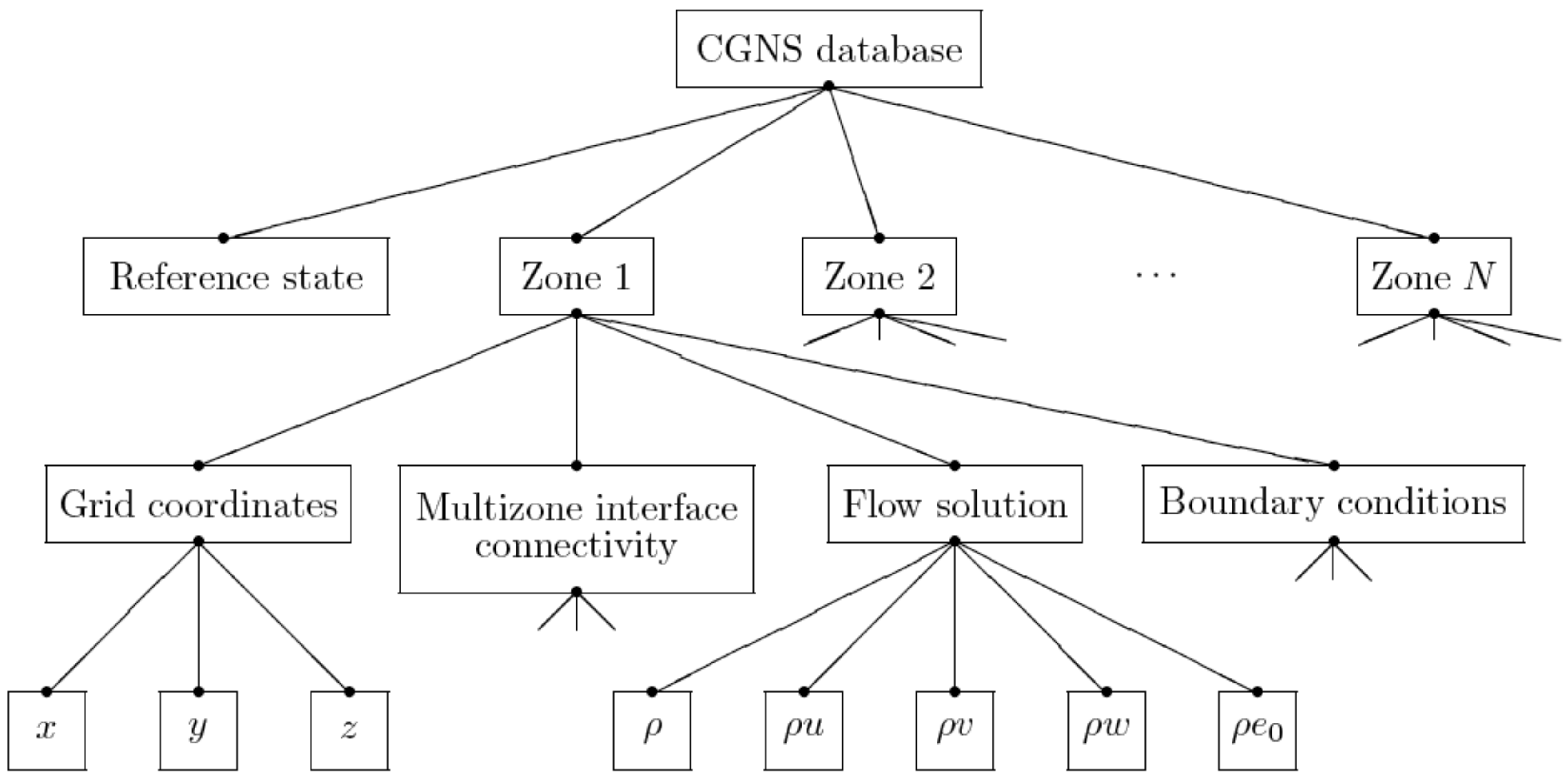
What is CGNS ?

- CFD General Notation System
  - Principal target is the data normally associated with compressible viscous flow (i.e. Navier-Stokes)
  - Applicable to computational field physics in general with augmentation of the data definitions and storage conventions
- Objectives
  - Provide a general, portable and extensible standard for the storing and retrieval of CFD analysis data
  - Offer seamless communication of CFD analysis data between sites, applications and system architectures
  - Eliminate the overhead costs due to file translation and multiplicity of data sets in various formats
  - Provide free, open software – GNU Lesser General Public License

What is CGNS ?

- Standard Interface Data Structures (SIDS)
  - Collection of conventions and definitions that defines the intellectual content of CFD-related data.
  - Independent of the physical file format
- SIDS to ADF Mapping
  - Advanced Data Format
- SIDS to HDF5 Mapping
  - Defines how the SIDS is represented in HDF5
- CGNS Mid-Level Library (MLL)
  - High level Application Programming Interface (API) which conforms closely to the SIDS
  - Built on top of ADF/HDF5 and does not perform any direct I/O operation

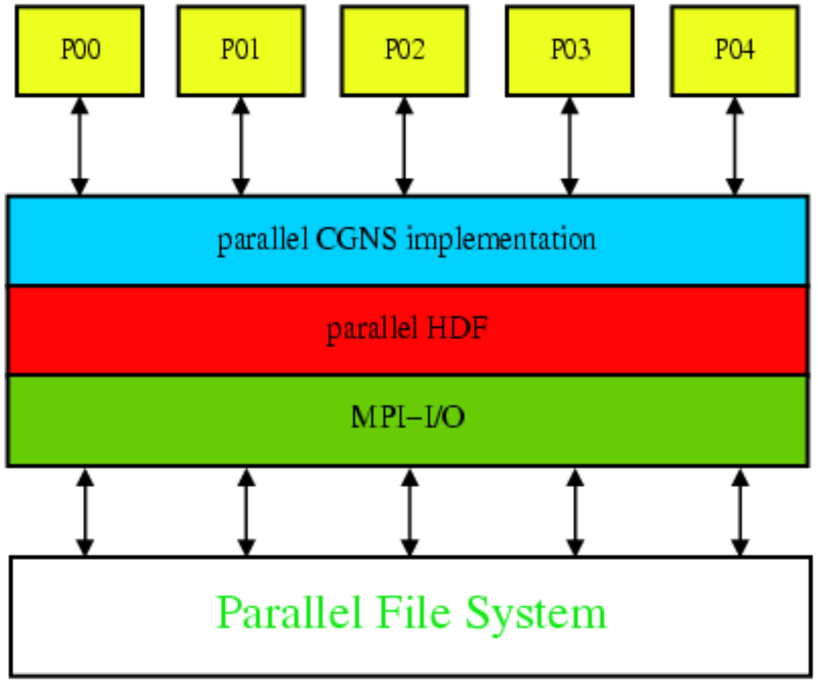
CGNS


CGNS database
Reference state
Zone 1
Zone 2
· · ·
Zone N
Grid coordinates
Multizone interface connectivity
Flow solution
Boundary conditions
x
y
z
ρ
ρu
ρv
ρw
ρe_0

I/O Needs on Parallel Computers

- High Performance
  - Take advantage of parallel I/O paths (when available)
  - Support for application-level tuning parameters
- Data Integrity
  - Deal with hardware and power failures sanely
- Single System Image
  - All nodes "see" the same file systems
  - Equal access from anywhere on the machine
- Ease of Use
  - Accessible in exactly the same ways as a traditional UNIX-style file system

Parallel CGNS I/O

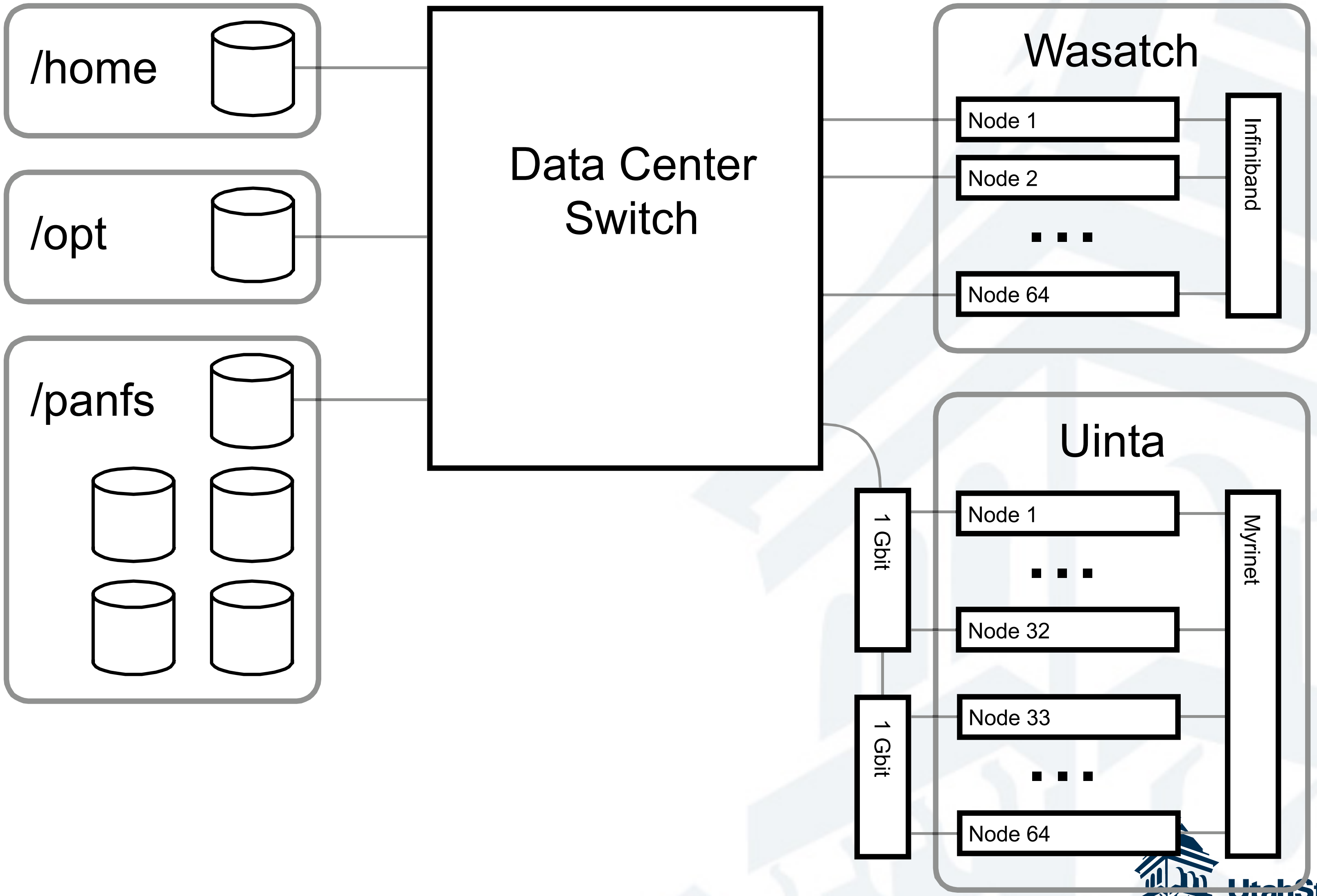


- New parallel interface
- Perform I/O collectively
- Potential I/O optimizations for better performance
- CGNS integration
- Overcome HDF5 limitation because of multi block data partitioning
  - Introduce I/O queue
  - Flush queue and write to disk

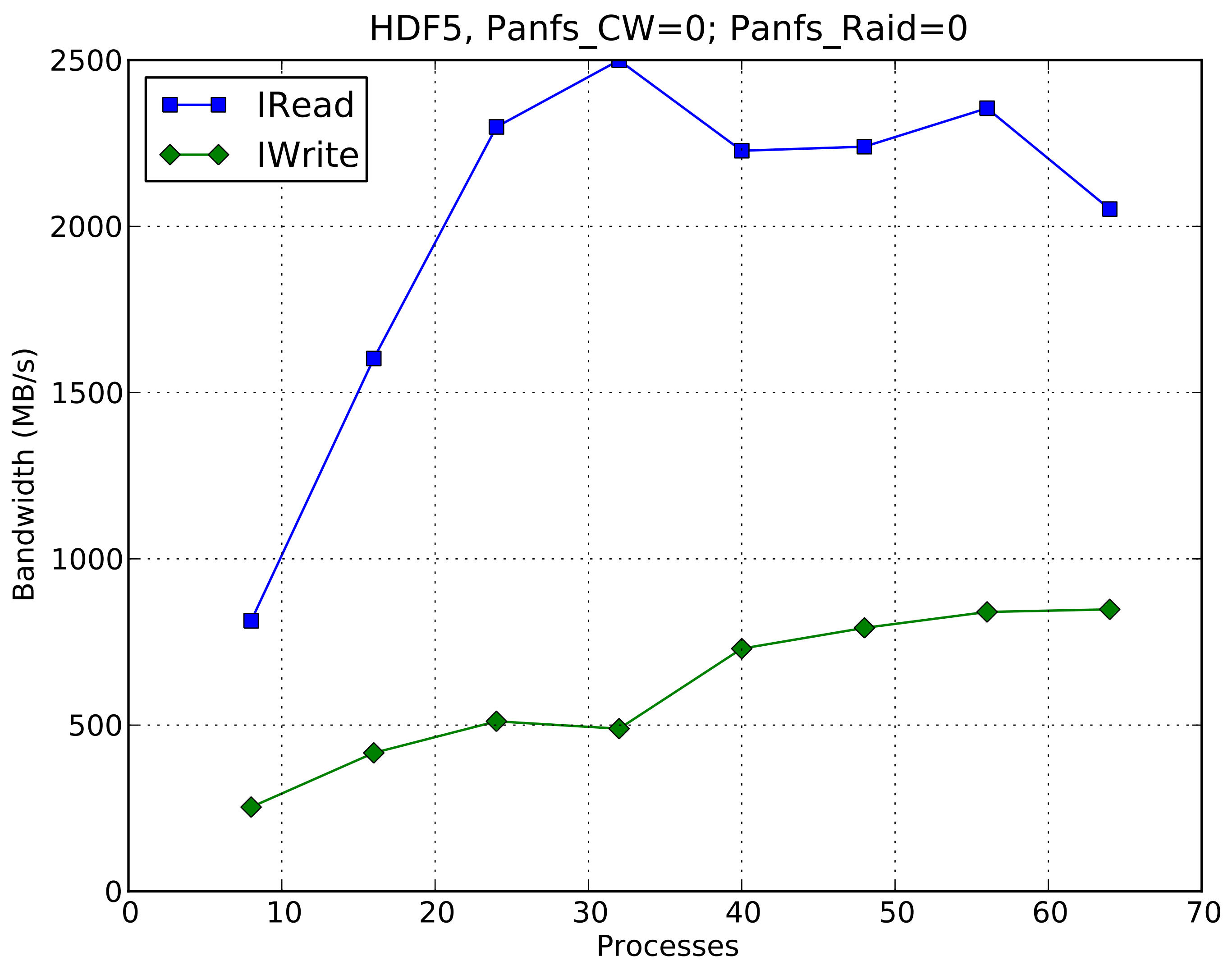
Parallel CGNS API

General File Operations	
cgp_open	Open a new file in parallel
cgp_base_read	Read the details of a base in the file
cgp_base_write	Write a new base to the file
cgp_nbases	Return the number of bases in the file
cgp_zone_read	Read the details of a zone in the base
cgp_zone_type	Read the type of a zone in the base
cgp_zone_write	Write a zone to the base
cgp_nzones	Return the number of zones in the base
Coordinate Data Operations	
cgp_coord_write	Create the node and empty array to store coordinate data
cgp_coord_write_data	Write coordinate data to the zone in parallel
Unstructured Grid Connectivity Operations	
cgp_section_write	Create the nodes and empty array to store grid connectivity for an unstructured mesh
cgp_section_write_data	Write the grid connectivity to the zone in parallel for an unstructured mesh
Solution Data Operations	
cgp_sol_write	Create the node and empty array to store solution data
cgp_sol_write_data	Write solution data to the zone in parallel
General Array Operations	
cgp_array_write	Create the node and empty array to store general array data
cgp_array_write_data	Write general array data to the zone in parallel
Queued I/O Operations	
queue_slice_write	Queue a write operation to be executed later
queue_flush	Execute queued write operations

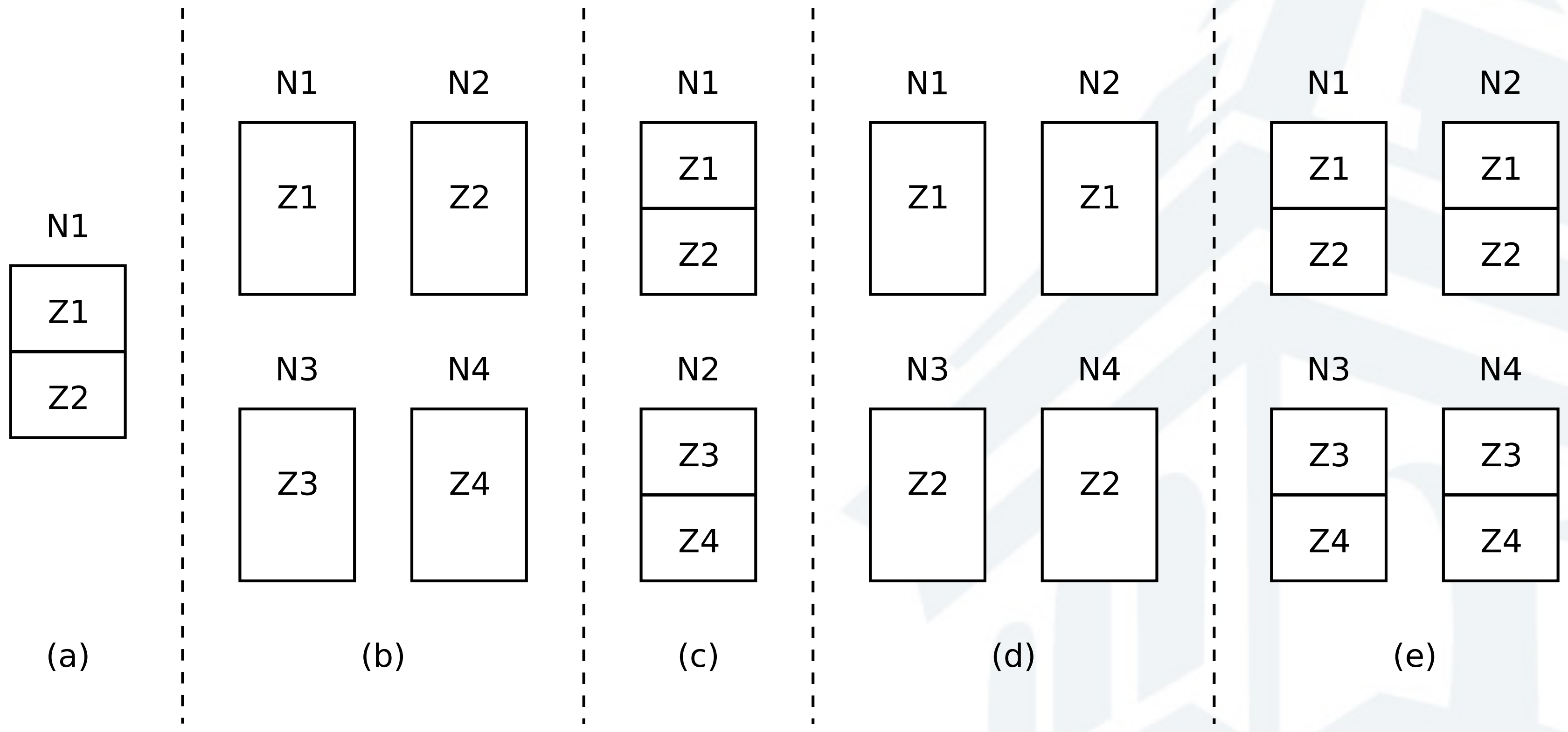
Benchmarking environment



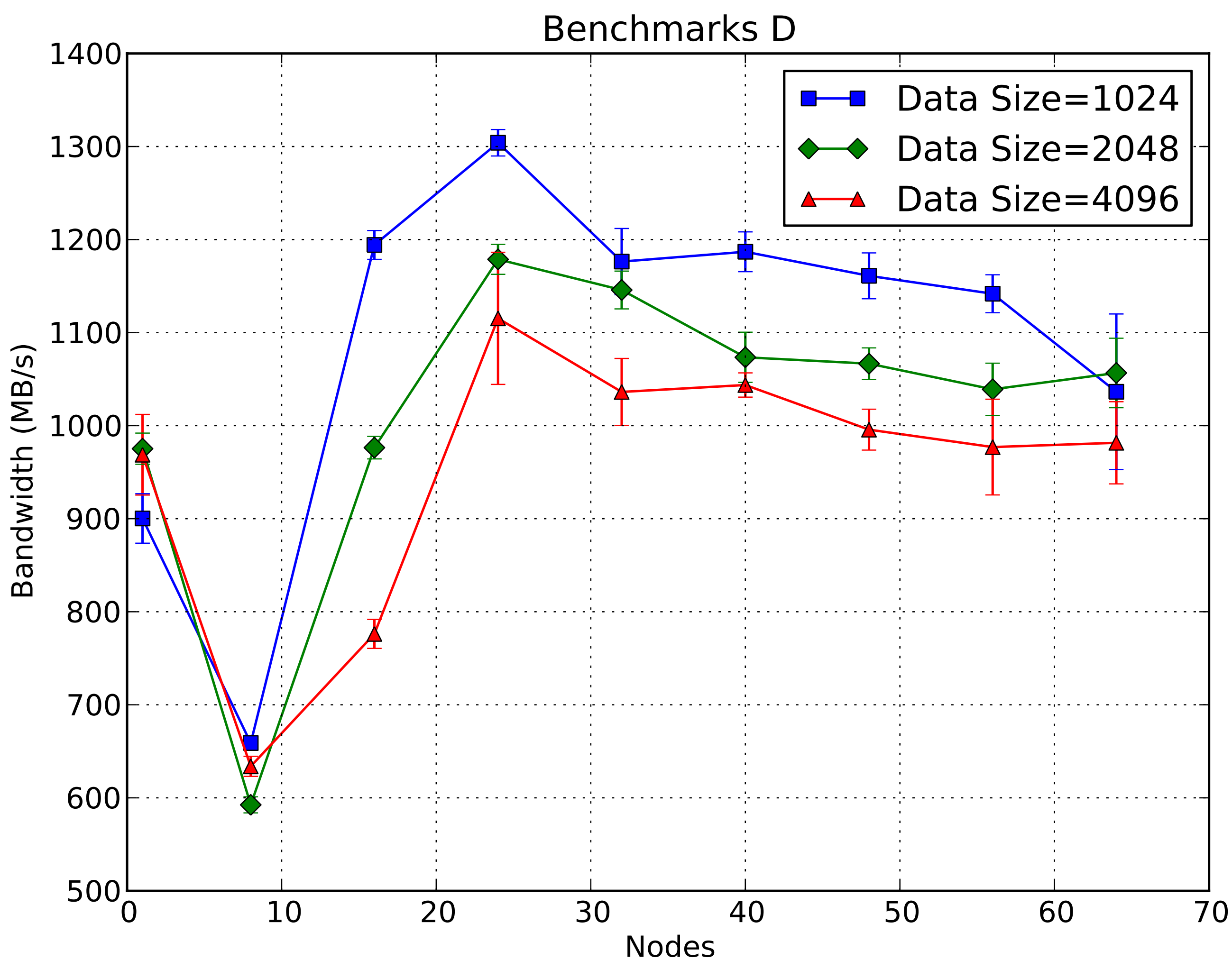
IOR results


HDF5, Panfs_CW=0; Panfs_Raid=0
Bandwidth (MB/s)
Processes
IRead
IWrite
0
500
1000
1500
2000
2500
0
10
20
30
40
50
60
70

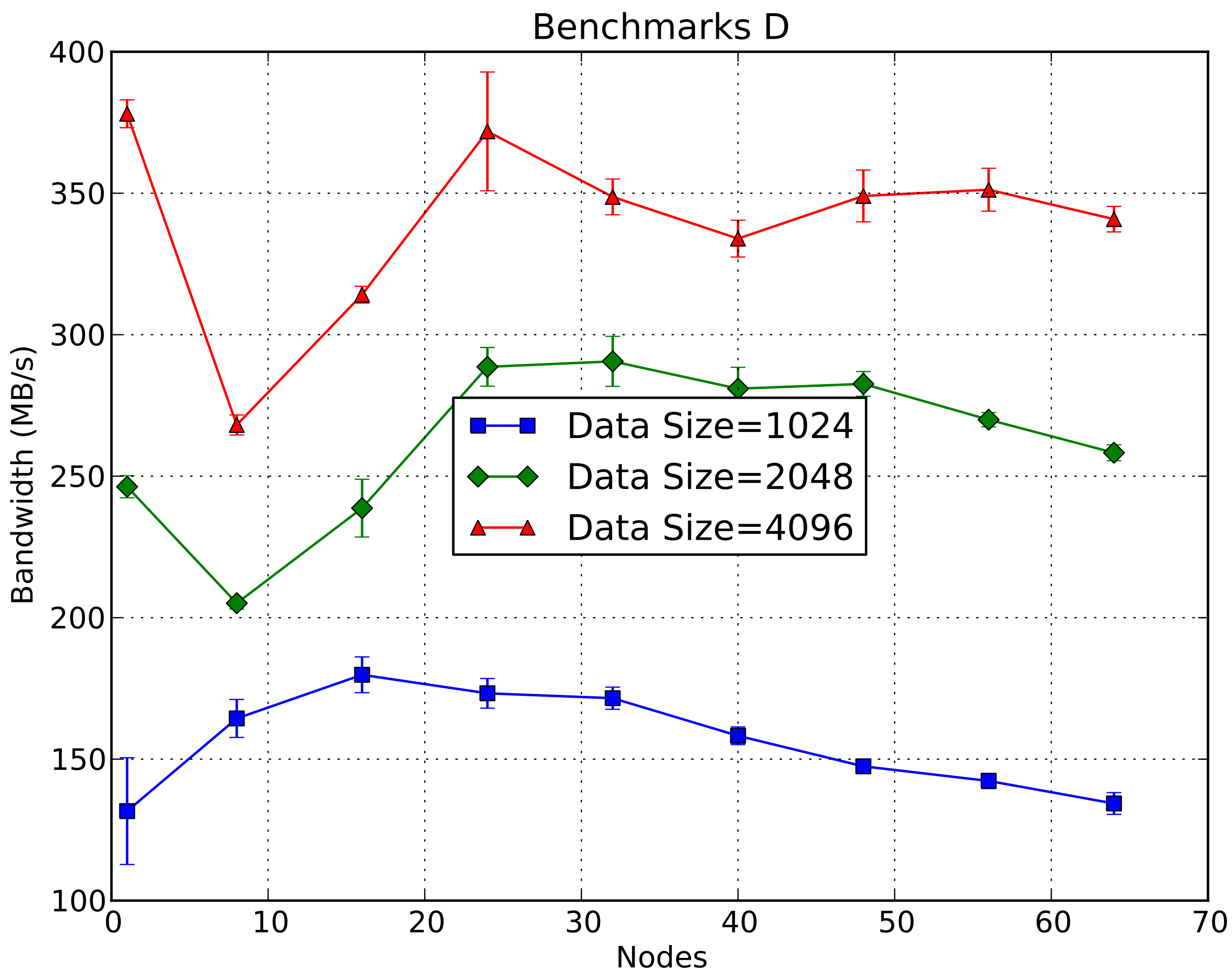
Benchmarks for pCGNS



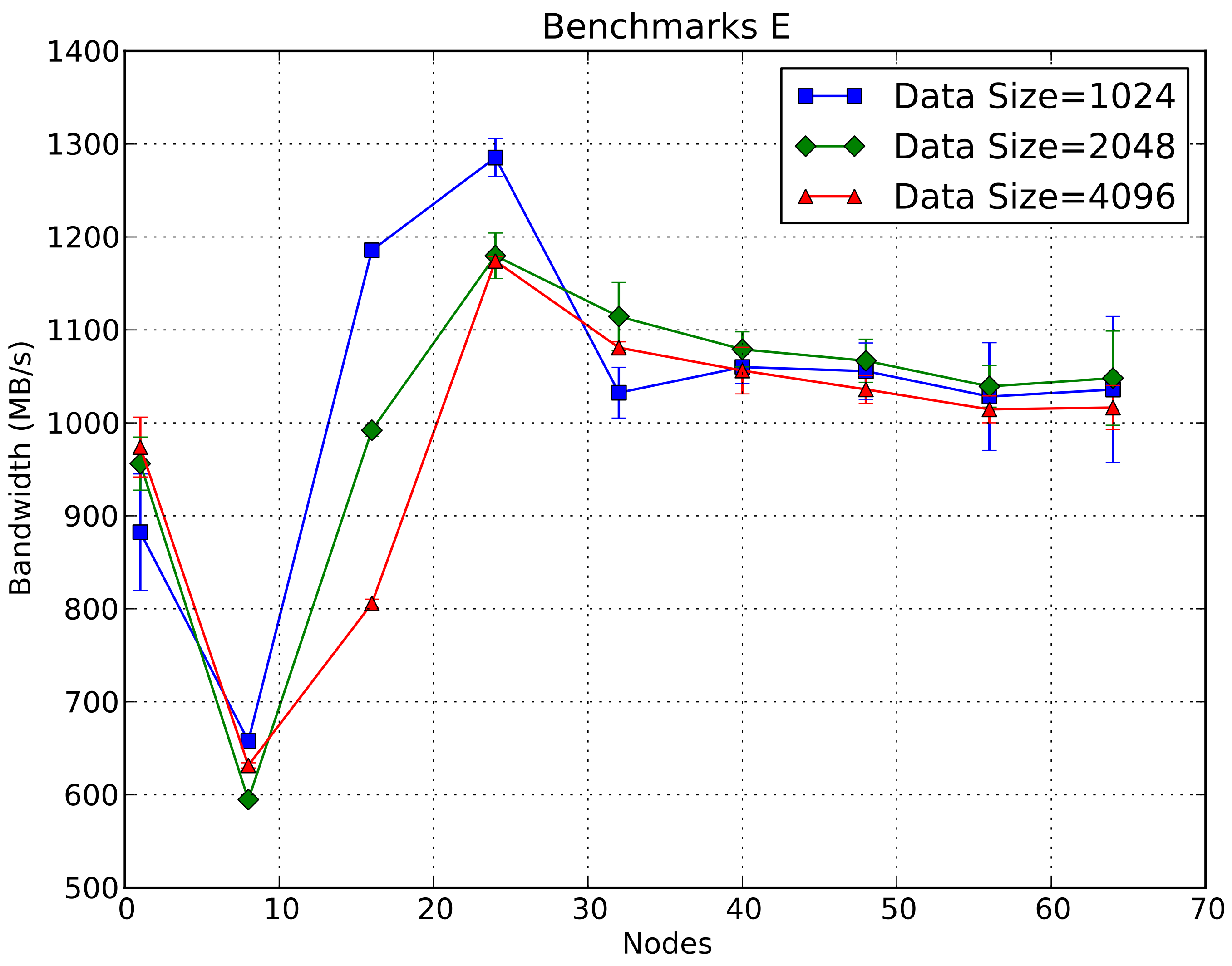
Reading one partitioned zone per process with multiple zones



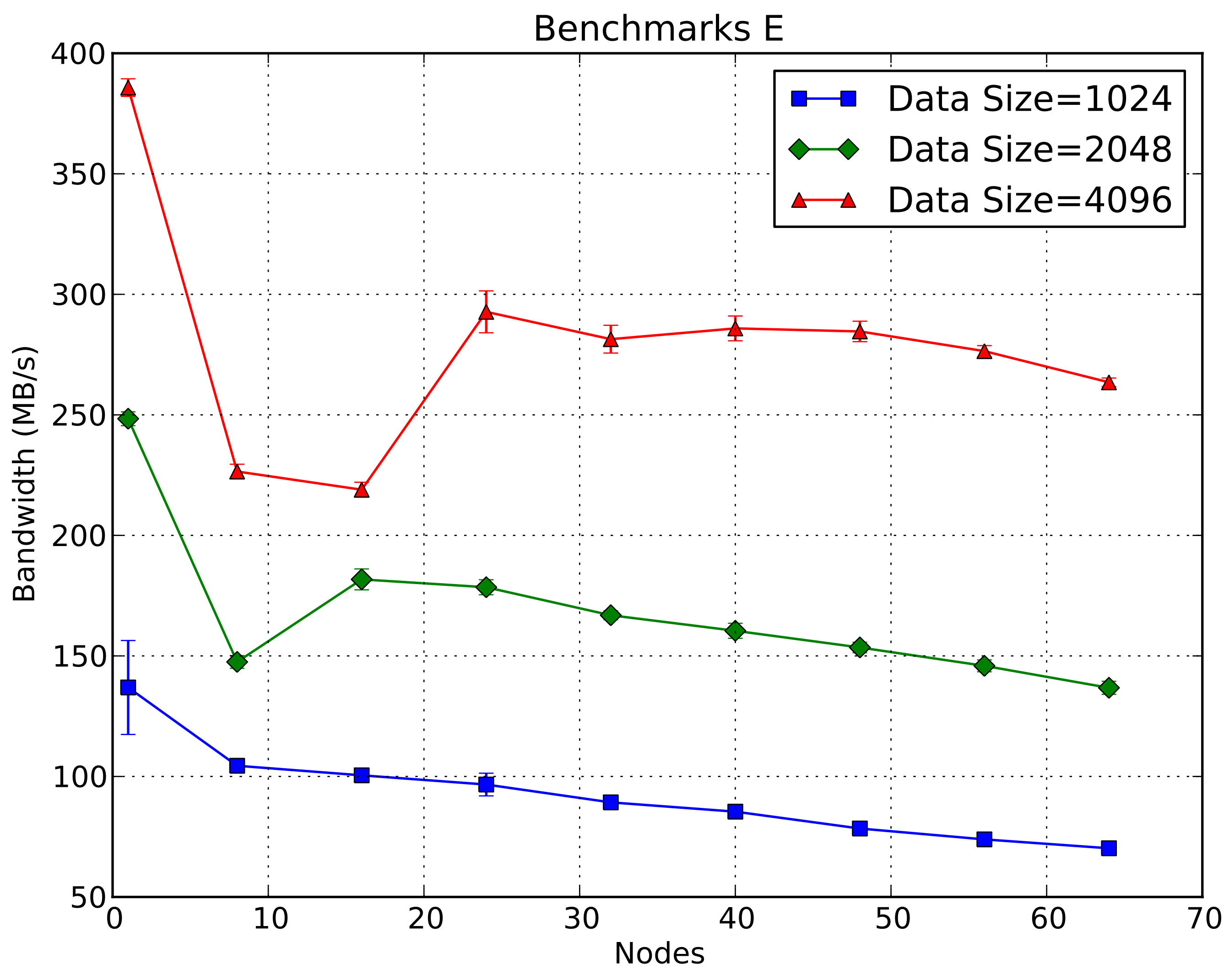
Writing one partitioned zone per process with multiple zones



Reading multiple partitioned zone per process with multiple zones



Writing multiple partitioned zone per process with multiple zones



Conclusion

- Created a parallel library for CGNS data I/O
- Cover all I/O patterns supported for multi-block structured and unstructured performance
- Implemented a queuing approach for I/O request since HDF5 can write only in one array at a time
- Achieved good performance for a high level library compared to IOR on our benchmarking system

Appendix

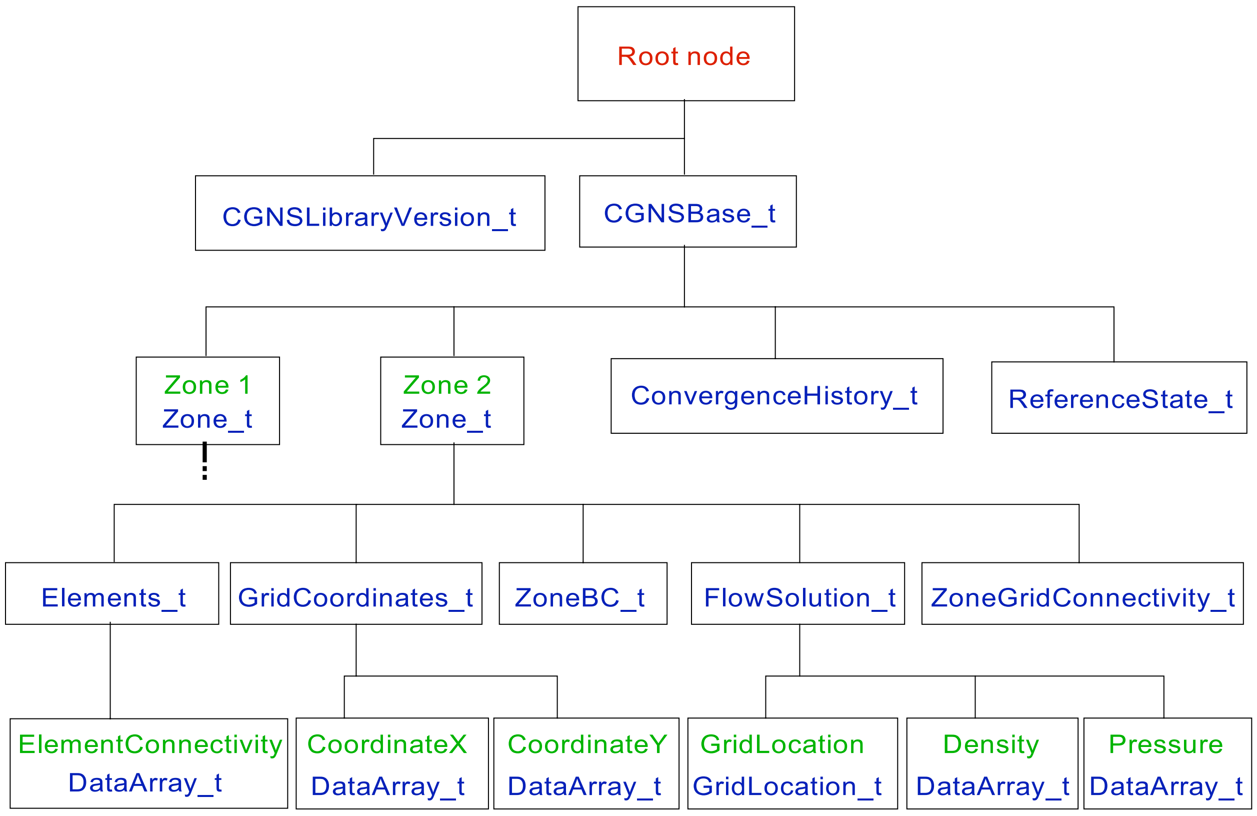


Figure 12: Example CGNS file with zone being the node containing grid and solution information.

Table I: API of the pCGNS library as implemented in *pcgnslib.c* and declared in *pcgnslib.h*. Software using the library to access CGNS files in parallel must use the routines listed here.

General File Operations	
cgp_open	Open a new file in parallel
cgp_base_read	Read the details of a base in the file
cgp_base_write	Write a new base to the file
cgp_nbases	Return the number of bases in the file
cgp_zone_read	Read the details of a zone in the base
cgp_zone_type	Read the type of a zone in the base
cgp_zone_write	Write a zone to the base
cgp_nzones	Return the number of zones in the base
Coordinate Data Operations	
cgp_coord_write	Create the node and empty array to store coordinate data
cgp_coord_write_data	Write coordinate data to the zone in parallel
Unstructured Grid Connectivity Operations	
cgp_section_write	Create the nodes and empty array to store grid connectivity for an unstructured mesh
cgp_section_write_data	Write the grid connectivity to the zone in parallel for an unstructured mesh
Solution Data Operations	
cgp_sol_write	Create the node and empty array to store solution data
cgp_sol_write_data	Write solution data to the zone in parallel
General Array Operations	
cgp_array_write	Create the node and empty array to store general array data
cgp_array_write_data	Write general array data to the zone in parallel
Queued I/O Operations	
queue_slice_write	Queue a write operation to be executed later
queue_flush	Execute queued write operations

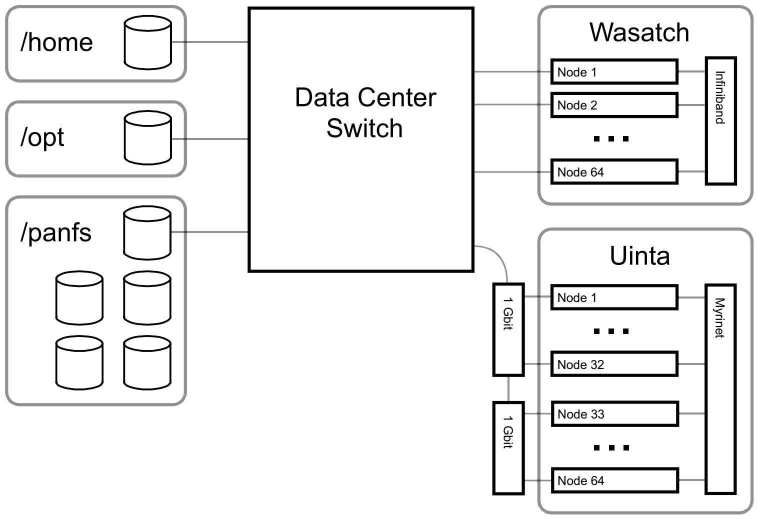


Figure 13: Topology of cluster's network attached storage at Utah State University. The older cluster Uinta is connected to the root switch through either one or two gigabit switches, whereas all nodes in the new cluster Wasatch are connected directly to the root switch. All the storage is connected directly to the root switch, including a parallel storage solution from Panasas mounted on /panfs.