PDS/PIO: Lightweight Libraries for Collective Parallel I/O

Judy Sturtevant, Mark Christon, Philip D. Heermann
Sandia National Laboratories
Albuquerque, NM 87185
{jesturt, machris, pdheerm}@sandia.gov
Pang-Chieh Chen
pchen@CS.Stanford.EDU

Abstract:

PDS/PIO is a lightweight, parallel interface designed to support efficient transfers of massive, grid-based, simulation data among memory, disk, and tape subsystems. The higher-level PDS (Parallel Data Set) interface manages data with tensor and unstructured grid abstractions, while the lower-level PIO (Parallel Input/Output) interface accesses data arrays with arbitrary permutation, and provides communication and collective I/O operations. Higher-level data abstraction for finite element applications is provided by PXI (Parallel Exodus Interface), which supports, in parallel, functionality of Exodus II, a finite element data model developed at Sandia National Laboratories. The entire interface is implemented in C with Fortran-callable PDS and PXI wrappers.

Keywords:

I/O, Parallel I/O, Scalable I/O, Collective I/O

Introduction

Historically, scientists and engineers have depended on experimental results to validate their theories and designs. Experimentation has become prohibitively expensive, both from the financial and environmental points of view. Therefore, experimentation is increasingly being replaced by computer modeling and simulation. Increased reliance on high-resolution simulation forces the scope and complexity of those applications to increase dramatically. Computational resources have kept up with the increased demand, but I/O (Input/Output) subsystems have not.

I/O "bottlenecks" slow modeling and simulation application output as well as visualization application input. The I/O "bottleneck" appears at different levels of software/hardware abstractions, e.g., in data format, application read/write, file systems, and disk. Much research has demonstrated the efficiencies of data organization and collective I/O [8][9][10]. Panda [11] is an early system that made use of server-directed, collective I/O. PASSION (Parallel and Scalable Software for Input-Output) provides software support for loosely synchronous applications with data pre-fetching and data-sieving [12].

MPI-IO [4] is the first standard interface designed for portable, parallel I/O. It provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between memory and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on disk. Currently, Sandia's unstructured grid applications would not utilize MPI-IO's functionality to its fullest, but would have to deal with the additional overhead of having that generality. MPI-IO is a promising interface that may become more useful to PDS/PIO in the future. ROMIO [13] is the Argonne National Laboratory implementation of MPI-IO that will be evaluated on Sandia's TFLOP system.

A data model currently in use at SNL is Exodus II: A Finite Element Data Model [3], developed to store and retrieve data for finite element analyses. The Exodus II data file is written using NETCDF and HDF. Its most obvious limitations include lack of parallelism, file size restricted to 2 gigabytes (GBytes), lack of compression, and poor performance. It is no longer a practical tool for the largest and most complex problems.

PDS/PIO addresses the I/O problem at the level of data format and parallel application read/write operations. Data is organized so that it may be presented to the disk subsystem in a near-optimal manner. The PXI/PDS/PIO libraries are based on the Single Program Multiple Data (SPMD) programming model. Data structures are decomposed into sub-units and distributed among the computational nodes (processors). The program executes on each compute node, and messages are passed between the nodes using the MPI message-passing protocol [2].

All PXI and PDS functions are collective in that all processors have to participate in function calls, albeit with processor-specific arguments. PDS maintains light-weight meta-data information about all of the data files within the dataset. To avoid burdening the file system, the meta-data is accessed by only one processor, which then broadcasts global data when necessary. At the PIO level, the data files are accessed only by a selected set of processors acting as I/O servers, which communicate with processors needing the actual data transfer.

Preliminary results demonstrate an order of magnitude increase in performance. A test application writing data from the Intel TFLOP to the Intel PFS (Parallel File System) [1] using NETCDF demonstrates a peak bandwidth of less than 10 megabytes (MBytes)/second. An unstructured grid-based simulation application writing PDS datasets with PXI from the Intel TFLOP to the Intel PFS demonstrates a peak bandwidth of greater than 100 MBytes/second.

Background

Environment

The Accelerated Strategic Computing Initiative (ASCI) is focused upon problems requiring three-dimensional, full-physics calculations. At Sandia National Laboratories, simulation applications include electromechanics (ALEGRA [14]), large deformation transient dynamics (PRONTO), low-Mach transient fluid dynamics (GILA [15]), and shock physics (CTH [16]), to name a few. It is estimated that high-resolution grid-based simulations with adequate temporal resolution will require 10^6 to 10^9 mesh nodes. The problem complexity is stressing the limits of current infrastructure. Achieving a balance between compute power, I/O bandwidth, disk and memory capacity has proven difficult for massively parallel systems [6].

Figure 1 illustrates the major hardware components of the TFLOP system. The major functional components are the analyst's workstation, the TFLOP computer, the TFLOP local disk storage system, the Visualization Server, and the HPSS (High Performance Storage System). The Intel TFLOP system is Sandia's MPP compute engine. It consists of 4536 compute nodes, each of which contains two Pentium Pro processors with 128 MBytes of memory per node. The complete system has 1.8 teraflops peak computational rate with 608 GBytes of system memory.

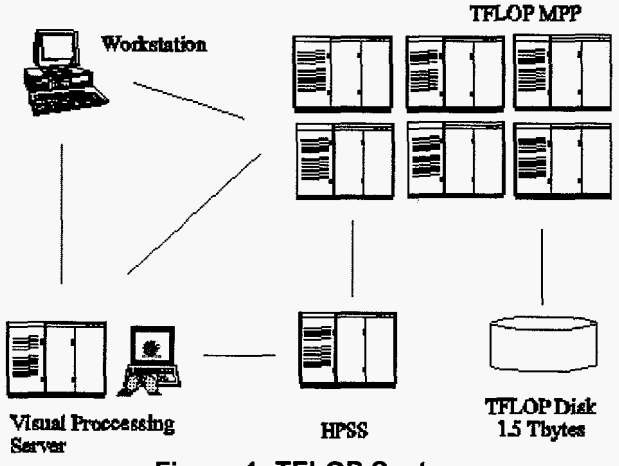

Figure 1: TFLOP System

The TFLOP I/O subsystem is comprised of 18 specialized nodes that process I/O requests and 36 RAID (Redundant Array of Independent Disk) devices. Each specialized node, or I/O node, consists of 2 Pentium Pro processors sharing 256 MBytes of memory on an Intel "Eagle" board. I/O nodes run the "Tflops Operating System" (TOS), a distributed OSF UNIX operating system. Two RAID controllers, attached to the I/O node, control two RAID devices, each of which has a 512 kilobyte (KByte) cache. Maximum bandwidth of each RAID device is 32 MBytes/second. Peak aggregate bandwidth to disk is 1 GByte/second, achievable only in a very carefully tuned test configuration.

This is in sharp contrast to the 51 GBytes/second total potential message traffic capacity between all nodes in the

DISCLAIMER

system. The inter-processor communication network of the TFLOP is configured in a 2-dimensional, 2-plane, mesh topology with peak bi-directional bandwidth of 800 MBytes/second and peak directional bandwidth of 400 MBytes/second. Measured directional bandwidth is 330 MBytes/second. Thus, the cross-sectional bandwidth of the inter-processor communication network is 51 GBytes/second.

As shown in Figure 2, the Intel Parallel File System (PFS) is built on one or more UNIX file systems, *striped* over multiple RAID devices. The 512 KByte blocksize was selected to match the RAID device cache size. The default stripe size of 1 MByte was selected to match the total RAID device cache available per I/O node.

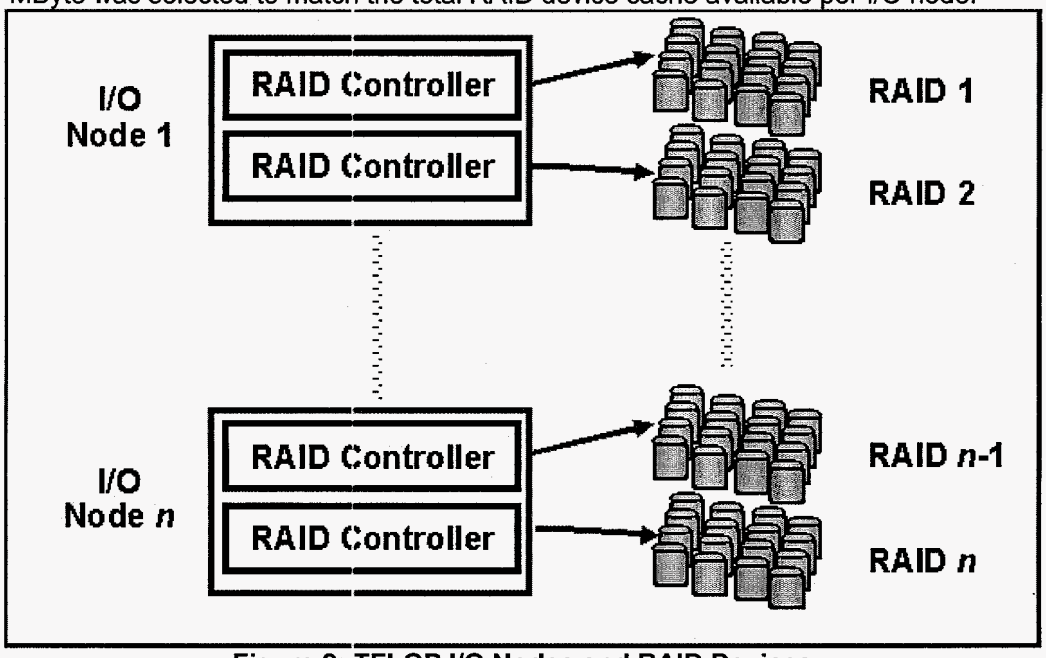

Figure 2: TFLOP I/O Nodes and RAID Devices

Motivation

User-level applications gain flexibility by using data modeling and data management tools. Such tools provide a common database for multiple application codes, such as mesh generators, analysis codes, and visualization software. Increased problem size and complexity is forcing better performance from these tools. PDS/PIO addresses the performance issues by providing a lightweight, parallel interface that optimizes data organization and disk I/O.

Currently, the ASCI Data Models and Formats (DMF) group is developing a portable data model interface that is generalized to solve all user needs. However, the time-critical needs of Sandia's ASCI applications required an immediate solution and could not wait for delivery of DMF.

The Intel TFLOP supports a Parallel File System (PFS) to give applications high-speed access to a large amount of disk storage. PFS file systems are optimized for simultaneous access by multiple nodes. File size is not limited to 2 GBytes. Access to PFS file systems use an I/O technique called *fast path I/O*, which gives better performance for large I/O operations (64K bytes or more per read or write). The Intel Cougar operating system on the compute nodes cannot directly access the file system. As shown in Figure 3, I/O requests from the application on the compute nodes are made via Remote Procedure Calls (RPCs) to an I/O Service Process executing in the Service Partition. The Service Process sends RPCs to the required I/O nodes, which then transfer the data directly from compute node memory to disk [7]. Very careful use of this system is required to prevent extremely bad I/O performance. It is quite possible for an application to compute for minutes, then write data to PFS for hours. PDS/PIO was developed to present data to PFS in a near-optimal manner.

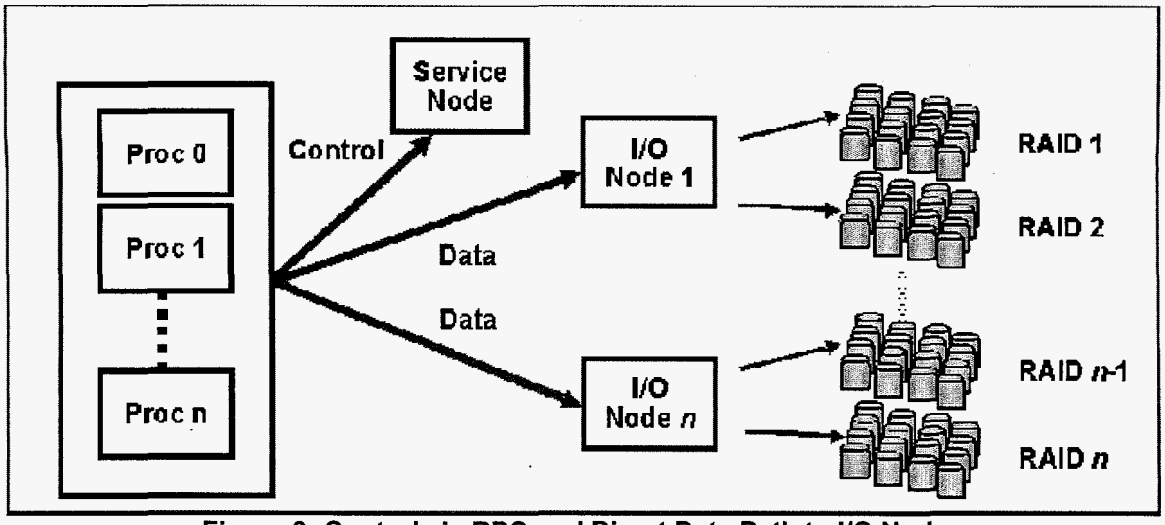

Figure 3: Control via RPC and Direct Data Path to I/O Nodes

PDS/PIO and its higher level interface, PXI, were designed to address Sandia's computer modeling and simulation I/O problems from computation to visualization. Requirements from the areas of computation data I/O and visualization data I/O differ in some areas. Simulation applications usually write data as it is computed, for an entire problem topology. Reading is limited to restarting applications from some previously written data (checkpoint). Visualization applications may read data for subsets of the topology, such as 'slices', or follow one particular variable over time. PDS/PIO addresses issues of data access patterns by organizing data to optimize access for computer modeling and simulation applications and visualization applications.

Approach

PDS/PIO is a lightweight, parallel interface designed to support efficient transfers of massive, grid-based, simulation data among memory, disk and tape subsystems. The higher-level PDS (Parallel Data Set) interface manages data with tensor and unstructured grid abstractions, while the lower-level PIO (Parallel Input/Output) interface accesses data arrays with permutation. User-level applications may access the interface at the PDS level. Higher-level abstraction for finite element applications is provided by PXI (Parallel Exodus Interface), which supports, in parallel, the functionality of Exodus II, a finite element data model. The entire interface is implemented in C with Fortran-callable PDS and PXI wrappers.

Parallel I/O performance is critical! Other requirements, while secondary, are, none the less, important. Additional requirements include:

- Portability
- Ease of use, i.e., a clean API
- "Gather/scatter" functionality
- Capability to interface with vendor-specific parallel file systems and HPSS
- Synchronous and asynchronous I/O
- Coordination of I/O requests (I/O servers)
- Thread-safe for SMP clusters

PXI/PDS/PIO was developed to address these requirements. Each library is described, beginning with the low-level PIO library.

PIO

PIO (Parallel Input/Output) reads and writes data arrays with arbitrary permutation. It is the software component responsible for organization and presentation of data to the disk subsystem in a near-optimal manner. PIO data permutation may include any or all of the following:

- Byte swapping
- Data type conversions
- Remapping based on Index and/or Value

To meet the PXI/PDS/PIO user requirements described previously, implementation of the PIO library was driven by the design criteria described in the following sections.

Parallel I/O

The single most important design requirement for our new I/O libraries was parallel I/O. Driving this requirement was the structure of the TFLOP I/O subsystem. The I/O control path is capable of handling multiple requests. More importantly, there are separate, direct data paths from compute node memory to I/O node, each of which has two attached RAID devices. Therefore, parallel I/O requests should be serviced concurrently, dramatically increasing aggregate bandwidth. PIO issues parallel I/O requests via the selected set of processors acting as I/O Servers.

To write a data array, all processors execute the algorithm in Figure 4. Writes to separate locations in the file are processed by separate I/O nodes, based on careful selection of data buffer size.

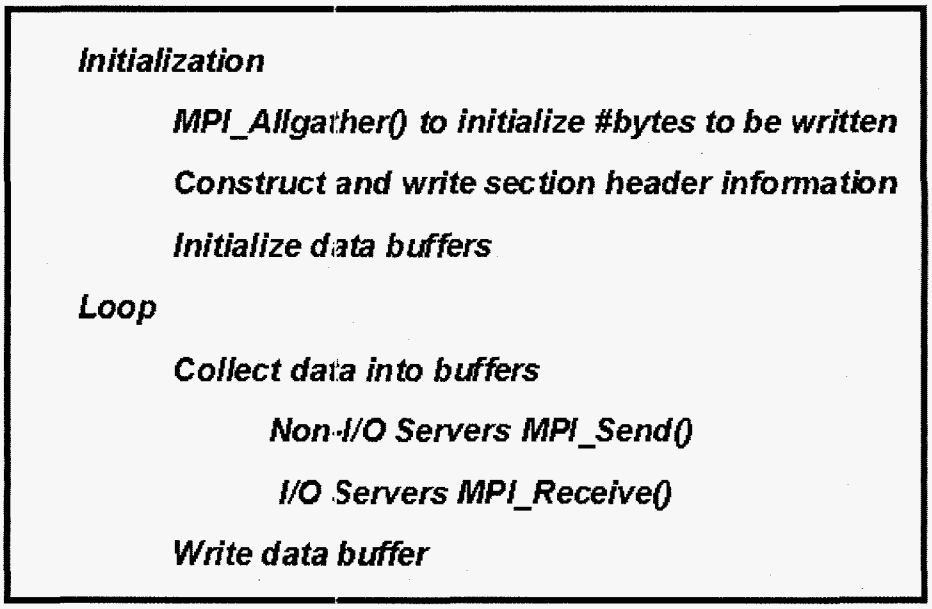

Figure 4: Algorithm to Write Data

Collective I/O

A second important design requirement for PIO was collective I/O. All compute nodes of a massively parallel system issuing I/O requests simultaneously would likely overwhelm the capacity of the I/O subsystem. Such was our experience on the TFLOP, where approximately 4500 compute nodes must issue I/O requests to a limited set of 18 I/O nodes. In the past, many Sandia applications attempted to "throttle" this I/O bottleneck by allowing only subsets of processors to issue I/O requests, while other processors remained idle. A better solution is collective I/O whereby data is "collected" from many processors to a few processors who issue the I/O requests.

The selected set of processors acting as I/O Servers communicates with processors needing actual data transfer via message passing. Communication via message passing is many times faster than disk I/O, resulting in greatly increased data throughput. Each I/O Server communicates with a subset of the application's compute processors. Figure 5 illustrates PIO using two I/O Servers.

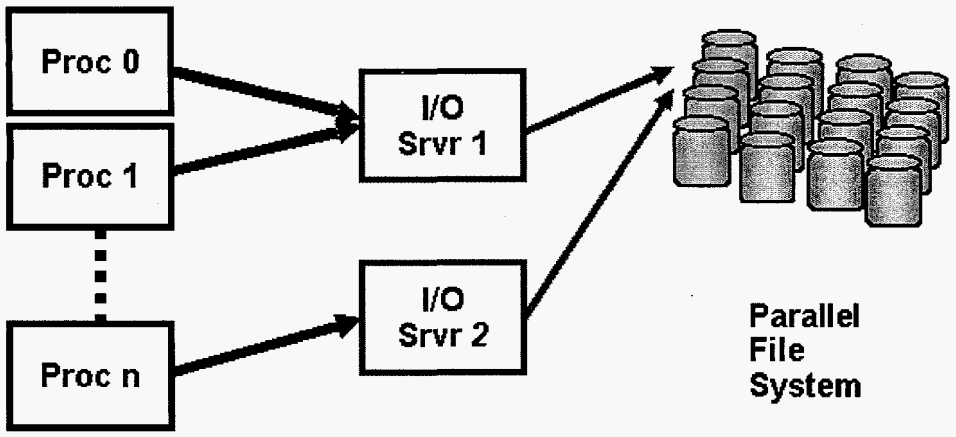

Figure 5: PIO Communications

Data Buffering

A third important design requirement for PIO was data buffering. Most, if not all, I/O subsystems operate more efficiently with few large data transfers, than with many small ones. Data buffering alone can improve I/O performance for a file system striped across multiple disks. A parallel file system should be "tuned" for optimum performance by selecting appropriate file system block size, ***stripe unit***, and ***stripe file*** size, based on disk speed and cache size. Optimum PIO buffer size will vary based on the optimum data transfer size.

Data is buffered by the I/O Server into 1 MByte blocks, by default, for efficient reads and writes to TFLOP PFS. Peak performance should be achieved with a 14 MByte data buffer. However, buffers larger than one or two MBytes may not offer enough performance improvement to justify the memory cost to the application. It becomes very important on distributed memory systems to provide a good balance of application and PIO memory requirements.

Asynchronous I/O

It is important to provide an asynchronous I/O capability, for greatest performance benefit. As described earlier, control for multiple I/O requests can be handled by the TFLOP Service process, and multiple data paths exist to the PFS I/O nodes. Therefore, it should be possible to achieve some degree of concurrency by issuing blocking I/O requests from multiple processors to multiple I/O nodes. A greater degree of concurrency will be achieved by issuing non-blocking I/O requests from multiple processors to multiple I/O nodes. PIO is currently limited to the use of blocking reads/writes. Multiple, non-blocking I/O requests have been problematic for our initial target architecture, the TFLOP PFS.

PDS

PDS provides data management at a level above data arrays, in a data abstraction context. All PDS functions are collective in that all processors have to participate in function calls, albeit with processor-specific arguments. To support flexible migration of data, PDS maintains light-weight meta-data information about all of the data files within the dataset. Meta-data is stored at the end of the initial data file. To avoid burdening the file system, the meta-data is accessed by only one processor which then broadcasts global data when necessary. For example, meta-data may contain processor-independent information that must be broadcast to all processors.

Before operating on a specific dataset, the meta-data is copied from the initial data file and accessed as a separate file (**meta-file**) with name extended by "MF", and a backup copy (**meta-file-copy**) with name extended by "MFC". PDS accesses and updates the meta-file and meta-file-copy. All other data files within the dataset are accessed through PIO. Access to the meta-file is normally buffered for efficiency. The size of the meta-file is expected to be no more than a few MBytes. At the end of PDS operations on a dataset or, at the user's discretion, the meta-data is appended to the initial data file. Figure 6 illustrates example PDS datasets.

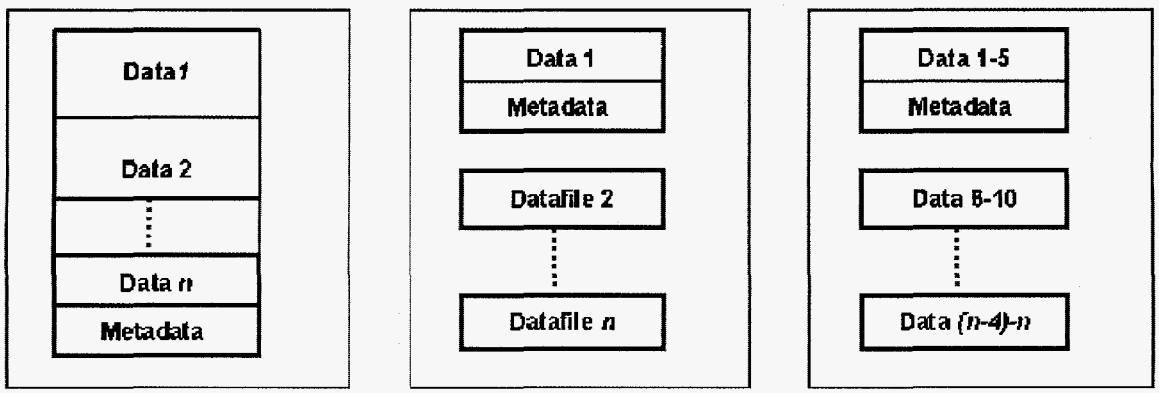

Figure 6: Example PDS Datasets

There are three sets of entities that PDS maintains: segments, variables, and infos.

PDS Segment

A segment is a contiguous piece of a file that contains data associated with a time index. It can either be a topology or a state segment, recording topological or state information at time associated with the time index. Segments must be written in increasing time-index order. A topology and a state segment can have the same time index; however, no two topology or two state segments can have the same time index.

Every segment must be entirely contained within one data file. Multiple segments may be written to a single data file. There can be more than one data file, each containing multiple segments. The segments have internal ID's that are ordered linearly: first in increasing-time-index order, second in topology-followed-by-state order. In normal use, datasets can be grown only by appending a new segment, and modified only by erasing all segments following a selected segment. Figure 7 illustrates PDS topology and state segments.

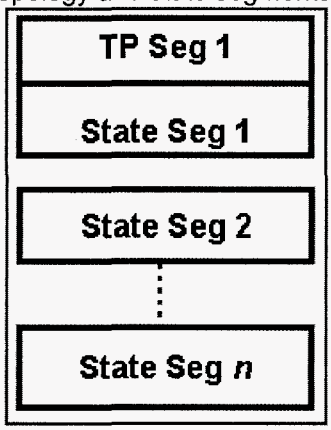

Figure 7: PDS Topology and State Segments

PDS Variable

A variable is a handle to a single, distributed, parallel data array of homogeneous types. It must be registered, to associate the variable with an ID and set its type, before the corresponding data can be read or written for each segment. Variable data are stored strictly in the data files and their accesses are collective and processor-dependent. A variable data corresponding to a particular segment is called a section. Thus, every segment is composed of a number of sections, each corresponding to a different variable. It is not necessary that every segment contain the same number of sections. Figure 8 illustrates PDS variables within a segment.

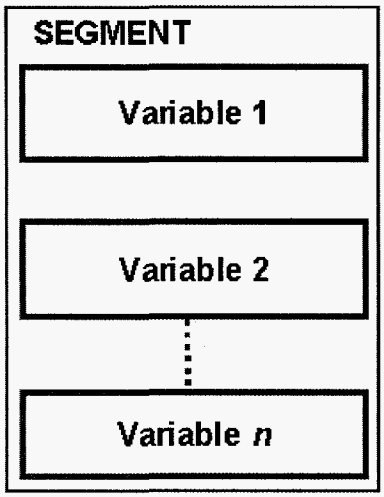

Figure 8: PDS Segment

A section is a set of variable data written in a processor-ranked, concatenated form, prepended by a header. Information in the 56-byte header includes an integer reserved for future use, number of processors contributing data, sum and max information, and a list of data sizes (number of bytes) for each processor. A section containing data written by ***n*** processors will consist of the header followed by data for P_0, P_1, P_2, ... P_{n-1}.

Variables are read and written by specifying the ID returned by its registration. Variable type is a sum of "item" type, such as PDS_SCALAR or PDS_VECTOR, plus "word" type, such as PIO_FLOAT or PIO_DOUBLE. Variable type may be reset before writing data. For a write operation, data type conversion is performed based on the "Put Type" of the variable versus the "compute word size" of the application performing the write. For a read operation, data type conversion is performed based on the "Get Type" of the variable versus the "compute word size" of the application performing the read.

PDS Info

An info is a processor-independent (name, value)-pair stored as meta-data. An info name is a string of a maximum length, and an info value is a homogeneous array whose size and datatype are user-specified. An info may be attached to the dataset or any variable or segment within the dataset. Using these info objects, one can introduce auxiliary user-supplied functions to interpret and process the data in special ways not provided through the standard PDS interface.

PXI

PXI provides data management at a level above PDS/PIO. It supports a finite element data model consisting of nodes, elements, element blocks, node sets, side sets (faces), etc. The initial version supports multi-processor applications reading/writing many-to-many, and visualization applications reading many-to-one.

The user controls the number of data files in a particular dataset in one of the following modes of operation:

- Single data file contains problem topology, all data, and meta-data.
- Initial data file contains problem topology, data for first time index, and meta-data. Remaining data files each contain data for one time index.
- Initial data file contains problem topology, data for first *n* time indices, and meta-data. Remaining data files each contain data for *n* time indices.

PXI also supports a subset of Nemesis I functionality (Set of Functions for Describing Unstructured Finite-Element Data on Parallel Computers) [5]. Nemesis I was developed at Sandia as a parallel extension to Exodus II.

During the initial design phase, the requirement for a higher-level, finite-element model, interface was not anticipated. The required "clean", easy-to-use Application Programming Interface (API) also needed to provide an easy migration path for the applications.

Results

Peak performance of the Intel PFS (Parallel File System) on Sandia's Intel TFLOP system is 1 GByte/second. It was achieved by a very finely tuned performance test conducted by Intel, using all 18 I/O nodes. Expected performance of PFS is 32 MBytes/second for each RAID device, or 64 MBytes/second for each I/O node. For 18 I/O nodes, the expected aggregate bandwidth would be 540 MBytes/second. Actual observed performance for 14 I/O nodes was 100 MBytes/second aggregate bandwidth.

On the TFLOP 28-way striped PFS, an easy way to increase performance was to increase the amount of data being read or written, i.e. PIO buffer size. Theoretically, one should be able to achieve close to the peak 32 MBytes/second bandwidth for each of the 28 RAID devices. In practice, performance grows very slowly for PIO buffer sizes greater than 2 MBytes. Table 1 shows the increase in performance (Max Bandwidth) as buffer size increases.

Buffer Size (MB)	Max BW (MB/sec)
1	15.5
2	26.1
4	47.0
8	80.0
10	93.5
14	112.5
28	150.2

Table 1: PDS/PIO Writes for 1 I/O Server

A data model currently in use at SNL is Exodus II, which uses NETCDF and HDF V for its I/O. A test application writing directly to NETCDF achieved a peak bandwidth of less than 10 MBytes/second using 1 I/O node. The same test application writing directly to HDF V achieved a peak bandwidth of less than 4 MBytes/second from one I/O node.

Preliminary PXI/PDS/PIO results demonstrate more than an order of magnitude increase in performance. A 64-processor simulation application (ALEGRA) writing PDS datasets with PXI from the Intel TFLOP to the Intel PFS demonstrated a peak bandwidth of 85 MBytes/second from one I/O node with a PIO buffer of 1 MByte. The same simulation application writing PDS datasets with PXI demonstrated a peak bandwidth of greater than 100 MBytes/second from one I/O node with a PIO buffer of 2 MBytes. The simulation application was updating meta-data following each time step.

As shown in Table 2, good performance improvement is also possible for smaller problems. A simulation application (GILA) writing PDS datasets demonstrated a peak bandwidth of 75.6 MBytes/second from one I/O node with a PIO buffer of 1 MByte. In the following table, bandwidth is reported in MBytes/second.

Dim	Elem	Node	Proc	Write	Max BW	Min BW	Avg BW
2	1760	1852	8	81	18.3	6.16	12.31
2	31320	31791	32	59	71.9	1.23	39.10
2	31320	31791	32	59	73.8	1.14	38.50
3	44250	49584	32	81	75.6	0.02	38.63

Table 2: PDS/PIO Using 1 I/O Server

One of the accomplishments not to be forgotten is the significant improvement made toward generating and visualizing LARGE DATA using the PXI/PDS/PIO libraries. Applications no longer create thousands of files requiring a long, memory-intensive recombination step. There is no limit on file size.

Testing of functionality at the PIO level has uncovered issues that must be investigated further. On a system with normal user load, increasing the number of I/O servers is not producing the expected increases in aggregate bandwidth. Multiple blocking write requests to a single PFS file do not appear to be executing in parallel, as expected. Requests are made using appropriate *seek* and PIO buffer size of 1 MB, which should guarantee concurrent writes to multiple I/O nodes.

Another issue is that multiple non-blocking write requests to a single PFS file appear to produce incorrect behavior and 'hangs' of the Service process. Further investigation is required to verify the behavior.

An issue which needs further investigation is the potential conflict for an I/O Server, that contributes data to an I/O Server other than itself. The current algorithm that determines which processors contribute data to which I/O Servers is based on the ID of a data block, not based on processor ID. One solution would be to designate a set of processors as I/O servers that are not also compute servers. This solution would probably not be acceptable to the applications. A better solution might be to separate the communication step, i.e. the "data collection" step from the I/O request and data transfer.

An order of magnitude increase in I/O performance is a significant step toward the goal of parallel, scalable I/O libraries. However, we believe that our initial version of the libraries is simply the first step toward even better I/O performance.

Summary/Future

The scope and complexity of computer modeling and simulation applications will increase as computational power increases. ASCI is already planning for future systems with goals of 10, 30, and eventually 100 teraflops of peak computing performance. It is quite possible that I/O subsystems will continue to lag behind computational systems in performance. It is imperative that we continue to develop new strategies for improving I/O performance.

I/O "bottlenecks" slow modeling and simulation application output as well as visualization application input. PDS/PIO addresses the I/O problem at the level of data format and application read/write. Data is organized so that it may be presented to the disk subsystem in a near-optimal manner. PDS/PIO is a lightweight parallel interface designed to support efficient transfers of massive, grid-based, simulation data among memory, disk and tape subsystems.

Future directions include support for the following:

- Compression
- Adaptive meshing
- Application-directed archival storage/retrieval
- Read/Write using different number of processors than previous read/write
- More Nemesis functionality
- New data types
- Partial read/write operations

Parallel I/O performance is critical! Preliminary results are very promising. We must continue to improve performance of PDS/PIO and its higher level interface, PXI.

Acknowledgments

The authors wish to acknowledge Daniel Sands for his contributions to the manuscript. This work was supported, in part, by the United States Department of Energy under Contract DE-ACO4-94AL85000. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company.

References

1 Intel Corporation Paragon System Administrator's Guide. *Chapter 10: Managing PFS File Systems*. April 1996.
2 Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 1.1, June 1995. On the World-Wide Web at **http://www.mpi-forum.org/docs/docs.html**.
3 L. Schoof, V. Yarberry. EXODUS II: A Finite Element Data Model. *Sandia Report: SAND92-2137*, September 1996.
4 Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 1997. On the World-Wide Web at **http://www.mpi-forum.org/docs/docs.html**.
5 G. Hennigan, J. Shadid. NEMESIS I: A Set of Functions for Describing Unstructured Finite-Element Data on Parallel Computers. *Sandia Report,* July 1997.
6 M. Christon, D. Crawford, E. Hertel, J. Peery, A. Robinson. ASCI Red - Experiences and Lessons Learned with a Massively Parallel TeraFLOP Supercomputer. *Supercomputer 1997,* Hans-Werner Meuer, K. G. Saur, Munich, Germany, 1997.
7 B. Cole, P. Fay, B. Godley, G. Henry, D. Robboy, P. Work. Getting I/O Performance on the ASCI Red Platform. Intel Corporation, August, 1998. In preparation.

8 J. del Rosario, R. Bordawekar, A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93,* pages 56-70, April 1993. Also published in Computer Architecture News, 21(5):31-38, December, 1993.
9 D. Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems,* 15(1):41-74, February, 1997.
10 R. Thakur, A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming,* 5(4):301-317, Winter, 1996.
11 K. Seamons, Y. Chen, P. Jones, J. Jozwiak, M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95.* ACM Press, December, 1995.
12 R. Thakur, A. Choudhary, R. Bordawekar, S. More, S. Kuditipudi. Passion: Optimized I/O for Parallel Applications. *Computer,* 29(6):70-78, June, 1996.
13 R. Thakur, E. Lusk, W. Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October, 1997.
14 ALEGRA - A Three-Dimensional, Multi-Material, Arbitrary-Lagrangian-Eulerian Code for Solid Dynamics, **http://www.sandia.gov/1431/ALEGRAw.html**
15 GILA, **http://www.cs.sandia.gov/~machris**
16 CTH 3D Eulerian Shock Code, **http://www.sandia.gov/1431/CTHw.html**