Trilinos I/O Support (Trios)

Ron A. Oldfield, Gregory D. Sjaardema, Gerald F. Lofstead II
Sandia National Laboratories
{raoldfi,gdsjaar,gflofst}@sandia.gov

Todd Kordenbrock
Hewlett-Packard Company
thkorde@sandia.gov

November 3, 2011

Abstract

Trilinos I/O Support (Trios) is a new capability area in Trilinos that serves two important roles: 1) it provides and supports I/O libraries used by in-production scientific codes; 2) it provides a research vehicle for the evaluation and distribution of new techniques to improve I/O on advanced platforms. This paper provides a brief overview of the production-grade I/O libraries in Trios as well as some of the ongoing research efforts that contribute to the experimental libraries in Trios.

1 Introduction

The Trilinos project was started as an effort to "facilitate the design, development, and ongoing support" of mathematical libraries for scientific codes [12]. Initially, that involved developing parallel solver algorithms and libraries for large-scale multi-physics applications. As the project evolved, it became evident that support of scientific codes on high-performance computing (HPC) platforms required more than efficient parallel solvers. One identified gap in Trilinos was I/O support. In late 2010, the Trilinos project added the Trilinos I/O Support (Trios) capability area to address this gap.

To address current and future needs, the Trios capability area consists of two efforts. The first effort is to support the current needs of active users through providing standard, extensible I/O APIs. Second, an active I/O research platform for experimenting with techniques and architectures on new and evolving platforms. Developments made through the research platform are available for users willing to try newer techniques that are less mature. As these techniques mature, they will evolve into options for the users requiring a more proven, widely supported technology set.

1.1 Trios Software Components

Trios began with two primary objectives: provide I/O support for existing production scientific codes and provide a common repository and evaluation framework for experimental I/O software for next-generation platforms.

In early 2011, Trios was granted copyright approval to release the well-established Sandia National Laboratories Engineering Analysis Code Access System (SEACAS) [29]. Some of these libraries have been in use at Sandia for more than a decade. Incorporating the SEACAS libraries into Trios serves multiple purposes: it allows the SEACAS development team to leverage the stringent testing framework of Trilinos to ensure robustness, it provides a single point of access to existing Sandia customers, and it enables a broader distribution of SEACAS to potential external users. Section 2 provides a description of the SEACAS I/O libraries.

To address the research objective of Trios, the Trios team added the in-situ and in-transit data-services work that evolved from the Lightweight File Systems project at Sandia [21, 22]. The data-services software allows large-scale scientific applications to leverage additional computational resources for real-time data-staging [8, 16, 24, 30] or integrated data analysis [18]. Putting the data-services software in Trios simplifies development by providing a unified

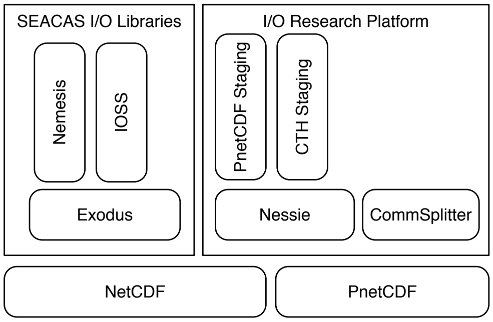


Figure 1: Trios Software Componets and Supporting Technology

software repository for researchers at different institutions and it provides an opportunity for co-design through increased access to application code teams and external users of Trilinos. In Section 3, we describe the Trios libraries used to support data services along with three examples of data-services currently in use.

1.2 Supported Platforms

As with the other capability areas in Trilinos, Trios provides an enabling technology that is “robust” and “efficient” on parallel computing platforms. Some of the experimental libraries in the Trios package are designed specifically for capability class supercomputers with low-level support for RDMA, such as the CrayXT, CrayXE, and large InfiniBand clusters. While there are third-party libraries, like the Portals3 reference implementation [4], that enable this code to execute on traditional TCP/IP based clusters, performance and robustness is not guaranteed or supported on all platforms.

2 SEACAS I/O Libraries

SEACAS includes applications and libraries that support a wide range of functionality including preprocessing and postprocessing (mesh generation, visualization); libraries (including I/O), FORTRAN extensions (memory management, parsing, and system services), visualization, and domain decomposition; and Exodus database manipulation (combination, parallel decomposition, concatenation, translation, differencing, and merging). In the context of this paper, we only discuss the I/O libraries Exodus, Nemesis, and IOSS.

2.1 Exodus

Exodus [27] is a library and data-model used for finite element analysis. It provides a common database for multiple application codes (e.g., mesh generators, analysis codes, and visualization software) rather than code-specific utilities. A common database gives flexibility and robustness for both the application code developer and the application code user. The use of the Exodus data model gives the user access to the large selection of application codes (including vendor-supplied codes) that read and/or write the Exodus format either directly or via translators.

The Exodus data model design was steered by finite-element application developers to meet the following requirements:

- *Random read/write access.*
- *Portability* - The data should be readable and writable on many systems from large HPC clusters down to small personal computers without translation.
- *Robustness* - Any data written to the file should not be corrupted if the application crashes or aborts later.
- *Support multiple languages* - Application programming interfaces (API) exist for FORTRAN, C, C++, and Python.
- *Efficiency* - It should be efficient, both in file space and time, to store and retrieve data to/from the database.
- *Real-time access during analysis* - Allow read access to the data in a file while the file is being created.
- *Extensibility* - Allow new data objects to be added without modifying the application programs that use the file format.

To address these requirements, the Exodus designers chose to layer the API on top of the Network Common Data Form (NetCDF) library [25]. NetCDF provides a portable, well-supported, self-describing data format with APIs in C, FORTRAN, C++, Python, Java, and Perl; The data sets structure is also easily extendible without copying or modifying the structure of the file–thus satisfying the final requirement of Exodus users.

Because an Exodus file is a netCDF file, an application program can access data via the Exodus API or the netCDF API directly. This functionality is illustrated in Figure 2. Although accessing the data

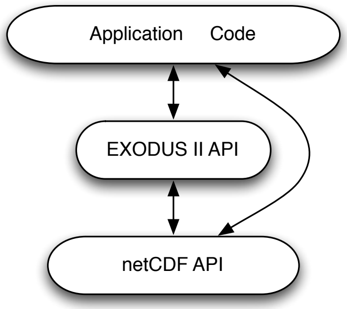


Figure 2: Exodus Software Stack

directly via the netCDF API requires more in-depth understanding of netCDF, this capability is a powerful feature that allows the development of auxiliary libraries of special purpose functions not offered in the standard Exodus library. For example, if an application required specialized data access not provided by the Exodus API, a function could be written that calls netCDF routines directly to read the data of interest. This feature can also be used if an application needs to store data that is not supported by Exodus. The application can write the data directly at the netCDF level. However, the disadvantages of this direct access is that: 1) other applications will not know about this data, 2) changes to the Exodus datastructure may result in the direct netCDF calls to fail, and 3) if a different data format were chosen in the future to replace netCDF, these calls would have to be modified before using the newer version of Exodus.

The Exodus file can contain nodes, edges, faces, and elements grouped in "blocks" or "sets". A block is a collection of homogeneous entities and all entities must be in one, and only one, block. A set is a collection of possibly heterogeneous entities of a single entity type and are optional. An additional entity group is a sideset which is a collection of "element - local element side" pairs. A sideset is typically used to specify a surface of the model where a boundary condition is applied. Each set and block can have optional named attribute data, results data, and map data.

Initialization data includes sizing parameters (e.g., number of nodes and number of elements), optional quality assurance information (names of codes that have operated on the data), and optional informational text.

The model data is static (does not change through time). This data includes block and set definitions; nodal coordinates; element, face, and/or edge connectivity[1]; attributes; and maps[2].

The results data are optional and include several types of variables – block and set data on nodes, edges, faces, and elements; sideset; and global – each of which is stored through time. Variables are output at each time step for all entities in the specific set or block. For example, the "node block" consists of all nodes in the model so a node block result variable would be output for all nodes in the model. Examples of a node block variable include displacement in the X direction; an element block variable example is element stress for all "hexahedral" elements in an element block. Another use of element variables is to record element status, a binary flag indicating whether each element is "active" or "inactive", through time. Global results are output at each time step for a single element or node or for a single property. Kinetic energy of a structure and the acceleration at a particular point are both examples of global variables. Although these examples correspond to typical finite element applications, the data format is flexible enough to accommodate a spectrum of uses.

Exodus files can be written and read by application codes written in C, C++, or Fortran via calls to functions in the application programming interface (API). Functions within the API are categorized as data file utilities, model description functions, or results data functions.

In general, the following pattern is followed for writing data objects to a file using the C API.

1. create the file with `ex_create`;

2. define global parameters using `ex_put_init`;

3. write out specific data object parameters; for example, define element block parameters with `ex_put_block`;

4. write out the data object; for example, output the connectivity for an element block with `ex_put_conn`;

5. close the file with `ex_close`.

Steps 3 and 4 are repeated within this pattern for each data object (e.g,, nodal coordinates, element blocks, node sets, side sets, and results variables). For some data object types, steps 3 and 4 are combined in a single call. During the database writing process, there are a few order dependencies (e.g., an element block must be defined before element variables for

[1] node lists for each element, face, and/or edge

[2] used to assign an arbitrary integer value to an entity, for example, a global id

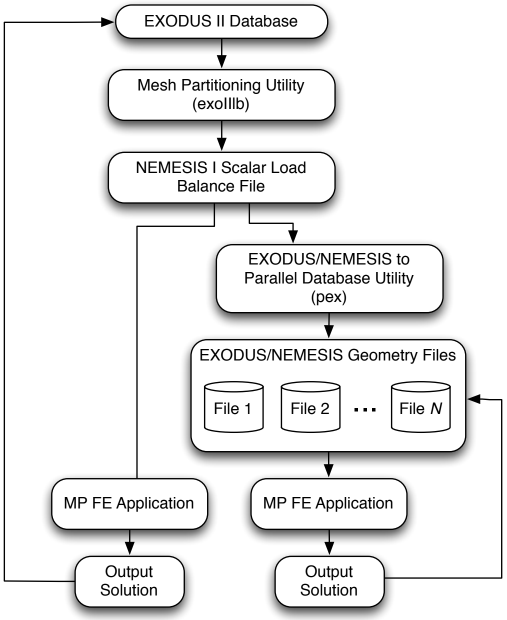

EXODUS II Database
Mesh Partitioning Utility (exoIIIb)
NEMESIS I Scalar Load Balance File
EXODUS/NEMESIS to Parallel Database Utility (pex)
EXODUS/NEMESIS Geometry Files
File 1
File 2
...
File N
MP FE Application
MP FE Application
Output Solution
Output Solution

Figure 3: Conceptual description of how EXODUS/Nemesis files are generated and used by a parallel finite element application

that element block are written) that are documented in the description of each library function.

For more details on the APIs and the Exodus data model, as well as application examples, see [27].

2.2 Nemesis

The analysis process for most application codes using Exodus mesh and results data on multi-processor parallel systems is that the original mesh database is "spread" into multiple databases–one per-process. The application code on each processor reads and writes its individual file and then the files are joined back together at the end of execution.

Nemesis [11] is an addition to the Exodus finite element database model that adds communication and partitioning information to the Exodus data model to facility this parallel analysis process. The SEACAS package includes applications that read Exodus data defining the model topology and then create a database containing structures that facilitate the partitioning of a single, scalar Exodus file into a set of files, read independently by each process in a parallel job. Nemesis takes advantage of the extensibility of Exodus to add additional information to an existing Exodus database, thus, any existing software that reads Exodus files can also read files that contain Nemesis information.

A Nemesis data set consists of a scalar "load-balance" file and a set of N "parallel geometry" files that contain the partition information for the parallel execution on each of the N processes used by the finite element code. The load-balance file contains information about the association of elements to processes and how processes exchange data with each other to obtain required boundary information. The load-balance file does not generally contain geometry information, such as element connectivity, nodal coordinates, or boundary-condition information. This information remains in the original Exodus database. The SEACAS `nem_slice` application uses the CHACO [10][3] and Zoltan [7] graph-partitioning libraries to create the Nemesis load-balance file

Given the original Exodus file and the load-balance file, an application has all the information required to execute a parallel finite element code. However, as mentioned previously, typically an additional application `nem_spread` is used to read the load-balance file and the original Exodus database and to create the N individual geometry files, each containing the portion of the original model for analysis by a specific processes. The geometry files are basically a standard Exodus file plus some additional datastructures that indicate which nodes are shared with other processes, which element boundaries are shared with other processes, and for which process this file is intended. Each process in a parallel analysis then reads the mesh information from the specific Exodus database for the process that contains the mesh geometry and topology for that process and the communication information specifying with which process(es) this process communicates. The output results file(s) are treated similarly with each process writing data to its own Exodus database. At the end of the analysis, the individual databases can be joined together using the SEACAS application `epu` while some visualization packages can handle the multiple files without joining.

For a more complete description of the Nemesis C and C++ APIs, see [11].

2.3 Sierra IO System

The Sierra IO system is a collection of C++ classes designed to provide an abstract interface to multiple finite-element database formats. Currently, Exodus, XDMF [6], embedded visualization, heartbeat, and history database formats are supported. The application accesses data at the abstract `Ioss::DatabaseIO` level, which is independent of the database format. Concrete `DatabaseIO` classes provide access to the

[3] CHACO is also provided in the Trilinos SEACAS package.

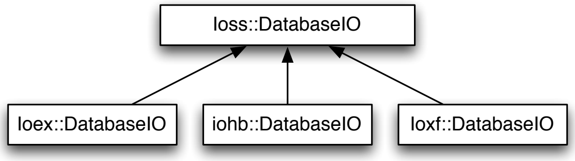


Figure 4: Ioss::DatabaseIO inheritance diagram.

data for each database type. In the context of this paper, we discuss the Ioss::DatabaseIO class.

Figure 4 shows the inheritance structure of Ioss::DatabaseIO. There are currently three concrete databases implemented: Exodus, in class Ioex:DatabaseIO; the eXtensible Data Model Format (XDMF) in class Iohb::DatabaseIO; and Heartbeat in Iohb::DatabaseIO, a simple text output of global data at each timestep.

The Ioss::DatabaseIO class has a pointer to an Ioss::Region, the root of the generic model representation. The Ioss:Region is an Ioss::GroupingEntity representing a portion of the finite-element model that can be read from or written to a database. Specific GroupingEntity classes include (all in the Ioss namespace): ElementBlock, FaceBlock, EdgeBlock, NodeBlock, ElementSet, FaceSet, EdgeSet, and NodeSet. There are also a Field class that is used to represent model, attribute, and transient field data; a Property class that is used to store properties of a GroupingEntity, for example, node count of a node block; and element topology for an element block.

While there are a number of details missing from this description, these classes form the basis of the finite-element database I/O capabilities in the Sierra system and are also used in several of the SEACAS database manipulation applications. The Ioss library is emerging as a viable C++-based API for the Exodus library.

3 Data Services in Trios

The reserach platform portion of Trios includes emerging I/O techniques. One such techique is providing data services. Simply put, a data service is a separate (possibly parallel) application that performs operations on behalf of an actively running scientific application.

This data service architecture uses remote direct-memory access (RDMA) to move data from memory to memory between the application and the service(s). Figure 5 illustrates the organization of an application using data services. On current capability-class HPC systems, services execute on compute nodes or service nodes and provide the application the ability to "offload" operations that present scalability challenges for the scientific code. One commonly used example for data services is data staging, or caching data between the application and the storage system [19, 20, 24]. Section 3.3 describes such a service. Other examples include proxies for database operations [23] and in-situ data analysis [9, 16, 30].

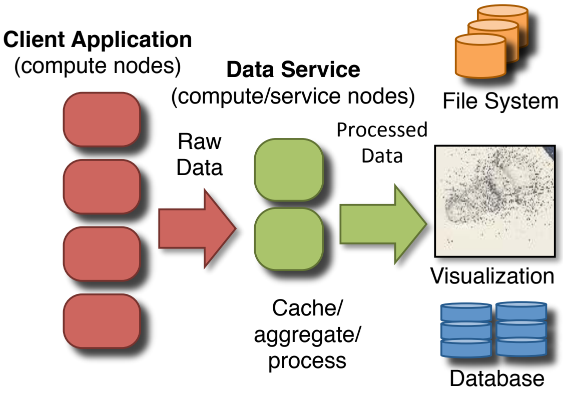


Figure 5: A data service uses additional compute resources to perform operations on behalf of an HPC application.

This section provides descriptions of the data service support libraries as well as examples of data services currently in use or in development. The data-transfer service, described in Section 3.2, is the canonical example on how to develop a data service using Nessie, the PnetCDF service from Section 3.3 is an example of link-time replacement I/O library that performs data-staging for bursty I/O operations, and the CTH in-transit analysis service in Section 3.4 demonstrates how we use a data service to perform real-time fragment detection for the CTH shock physics code.

3.1 Data Service Support Libraries

The primary library to support data services is the Network Scalable Service Interface (Nessie). It is used on all platforms and provides a basic framework for developing new services. Trios also includes a support library called "CommSplitter" used to enable data on the Cray XE6 platform.

3.1.1 Nessie

The NEtwork Scalable Service Interface, or Nessie, is a framework for developing parallel client-server data services for large-scale HPC systems [16, 22].

Nessie was originally developed out of necessity for the Lightweight File Systems (LWFS) project [21], a joint effort between researchers at Sandia National Laboratories and the University of New Mexico. The LWFS project followed the same philosophy of "simplicity enables scalability", the foundation of earlier work on lightweight operating system kernels at Sandia [26]. The LWFS approach was to provide a core set of fundamental capabilities for security, data movement, and storage and afford extensibility through the development of additional services. For example, systems that require data consistency and persistence might create services for transactional semantics and naming to satisfy these requirements. The Nessie framework was designed to be the vehicle to enable the rapid development of such services.

Because Nessie was originally designed for I/O systems, it includes a number of features that address scalability, efficient data movement, and support for heterogenous architectures. Features of particular note include 1) using asynchronous methods for most of the interface to prevent client blocking while the service processes a request; 2) using a server-directed approach to efficiently manage network bandwidth between the client and servers; 3) using separate channels for control and data traffic; and 4) using XDR encoding for the control messages (i.e., requests and results) to support heterogenous systems of compute and service nodes.

A Nessie service consists of one or more processes that execute as a serial or parallel job on the compute nodes or service nodes of an HPC system. We have demonstrated Nessie services on the Cray XT3 at Sandia National Laboratories, the Cray XT4/5 systems at ORNL, and a large InfiniBand cluster at SNL. The Nessie RPC layer has direct support of Cray's SeaStar interconnect [3], through the Portals API [4]; Cray's Gemini interconnect [1]; and InfiniBand [2].

The Nessie API follows a remote procedure call (RPC) model, where the client (i.e., the scientific application) tells the server(s) to execute a function on its behalf. Nessie relies on client and server stub functions to encode/decode (i.e., marshal) procedure call parameters to/from a machine-independent format. This approach is portable because it allows access to services on heterogeneous systems, but it is not efficient for I/O requests that contain raw buffers that do not need encoding. It also employs a 'push' model for data transport that puts tremendous stress on servers when the requests are large and unexpected, as is the case for most I/O requests.

To address the issue of efficient transport for bulk data, Nessie uses separate communication channels for control and data messages. In this model, a control message is typically small. It identifies the operation to perform, where to get arguments, the structure of the arguments, and so forth. In contrast, a data message is typically large and consists of "raw" bytes that, in most cases, do not need to be encoded/decoded by the server. For example, Figure 6 shows the transport protocol for an I/O server executing a write request.

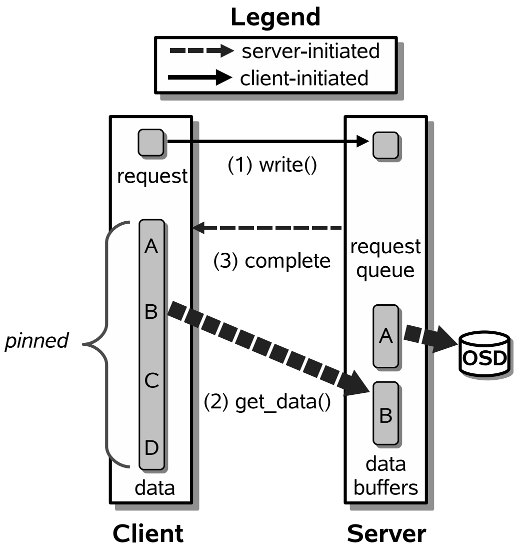


Figure 6: Network protocol for a Nessie storage server executing a write request. The initial request tells the server the operation and the location of the client buffers. The server fetches the data through RDMA get commands until it has satisfied the request. After completing the data transfers, the server sends a small "result" object back to the client indicating success or failure.

The Nessie client uses the RPC-like interface to push control messages to the servers, but the Nessie server uses a different, one-sided API to push or pull data to/from the client. This protocol allows interactions with heterogeneous servers and benefits from allowing the server to control the transport of bulk data [15, 28]. The server can thus manage large volumes of requests with minimal resource requirements. Furthermore, since servers are expected to be a critical bottleneck in the system (recall the high proportion of compute nodes to I/O nodes in MPPs), a server directed approach affords the server optimizing request processing for efficient use of underlying network and storage devices – for example, re-ordering requests to a storage device [15].

3.1.2 CommSplitter

The CommSplitter library was designed to overcome a security model limitation in the Gemini interconnect. On Gemini systems, multiple user space applications are not allowed to communicate[4]. We overcame that limitation by launching our jobs in Multiple Program, Multiple Data (MPMD) mode. MPMD mode enables a set of applications to execute concurrently, sharing a single MPI Communicator. The problem with this approach is that legacy applications were not designed to share a communicator with other applications. In fact, most HPC codes assume they have exclusive use of the `MPI_COMM_WORLD` communicator. When this is not the case, a global barrier, such as an `MPI_Barrier` function will hang because the other applications did not call the `MPI_Barrier` function.

To address this issue, we developed the CommSplitter library to allow applications to run in MPMD mode while still maintaining exclusive access to a virtual `MPI_COMM_WORLD` global communicator.

The CommSplitter library identifies the processes that belong to each application, then "split" the real `MPI_COMM_WORLD` into separate communicators. The library then uses the MPI profiling interface to intercept MPI operations, enforcing the appropriate use of communicators for collective operations.

No changes are required to the application source code to enable this functionality. The user simply links the CommSplitter library to the executable before launching the job. The library has no effect on applications that are not run in MPMD mode.

3.2 A Simple Data-Transfer Service

The data-transfer service is included in the "examples/xfer-service/" directory of the Trios package. This example demonstrates how to construct a simple client and server that transfer an array of 16-byte data structures from a parallel application to a set of servers. The code serves three purposes: it is the primary example for how to develop a data service, it is used to test correctness of the Nessie APIs, and we use it to evaluate network performance of the Nessie protocols.

Creating the transfer-service requires the following three steps:

1. Define the functions and their arguments.

2. Implement the client stubs.

3. Implement the server.

3.2.1 Defining the Service API

To properly evaluate the correctness of Nessie, we created procedures to transfer data to/from a remote server using both the control channel (through the function arguments or the result structure) and the data channel (using the RDMA put/get commands). We defined client and server stubs for the following procedures:

`xfer_write_encode` Transfer an array of data structures to the server using the control channel. This method sends the data through the procedure parameters, forcing the client to encode the array before sending and the server to decode the array when receiving. This procedure evaluates the performance of the encoding/decoding the arguments. For large arrays, this method also tests our two-phase transfer protocol in which the client pushes a small header of arguments and lets the server pull the remaining arguments on demand.

`xfer_write_rdma` Transfer an array of data structures to the server using the data channel. This procedure passes the length of the array in the arguments. The server then "pulls" the unencoded data from the client using the `nssi_get` function. This method evaluates the RDMA transfer performance for the `nssi_get_data` function.

`xfer_read_encode` Transfer an array of data structures to the client using the control channel. This method tells the server to send the data array to the client through the result data structure, forcing the server to encode the array before sending and the client to decode the array when receiving. This procedure evaluates the performance of the encoding/decoding the arguments. For large arrays, this method also tests our two-phase transfer protocol for the result structure in which the server pushes a small header of the result and lets the client pull the remaining result on demand (at the `nssi_wait` function).

`xfer_read_rdma` Transfer an array of data structures to the client using the data channel. This procedure passes the length of the array in the arguments. The server then "puts" the unencoded data into the client memory using the `nssi_put_data` function. This method evaluates the RDMA transfer performance for the `nssi_put_data` function.

Since the service needs to encode and decode remote procedure arguments, the service-developer

[4] Cray is currently addressing this issue to better support data services in future versions of Gemini

```
/* Data structure to transfer */
struct data_t {
   int int_val;        /* 4 bytes */
   float float_val;    /* 4 bytes */
   double double_val; /* 8 bytes */
};

/* Array of data structures */
typedef data_t data_array_t<>;

/* Arguments for xfer_write_encode */
struct xfer_write_encode_args {
   data_array_t array;
};

/* Arguments for xfer_write_rdma */
struct xfer_write_rdma_args {
   int len;
};

...
```

Figure 7: Portion of the XDR file used for a data-transfer service.

has to define these data structures in an XDR file. Figure 7 shows a portion of the XDR file used for the data-transfer example. XDR data structures definitions are very similar to C data structure definitions. During build time, a macro called "`TriosProcessXDR`" converts the xdr file into a header and source file that call the XDR library to encode the defined data structures. `TriosProcessXDR` executes the UNIX tool "rpcgen" the remote procedure call protocol compiler to generate the source and header files.

3.2.2 Implementing the client stubs

The client stubs provide the interface between the client application and the remote service. The stubs do nothing more than initialize the RPC arguments, and call the `nssi_call_rpc` method. For RDMA operations, the client also has to provide pointers to the appropriate data buffers so the RDMA operations know where to put or get the data for the tranfer operation.

Figure 8 shows the client stub for the `xfer_write_rdma` method. Since the `nssi_call_rpc` method is asynchronous. The client checks for completion of the operation by calling the `nssi_wait` method with the `nssi_request` as an argument.

```
int xfer_write_rdma(
  const nssi_service *svc,
  const data_array_t *arr,
  nssi_request *req)
{
  xfer_write_rdma_args args;
  int nbytes;

  /* the only arg is size of array */
  args.len = arr->data_array_t_len;

  /* the RDMA buffer */
  const data_t *buf=array->data_array_t_val;

  /* size of the RDMA buffer */
  nbytes = args.len*sizeof(data_t);

  /* call the remote methods */
  nssi_call_rpc(svc, XFER_PULL,
    &args, (char *)buf, nbytes,
    NULL, req);
}
```

Figure 8: Client stub for the `xfer_write_rdma` method of the transfer service.

3.2.3 Implementing the server

The server consists of some initialization code along with the server-side API stubs for any expected requests. Each server-side stub has the form described in Figure 9. The API includes a request identifier, a peer identifier for the caller, decoded arguments for the method, and RDMA addresses for the data and result. The RDMA addresses allow the server stub to write to or read from the memory on the client. In the case of the `xfer_write_rdma_srvr`, the stub has to pull the data from the client using the `data_addr` parameter and send a result (success or failure) back to the client using the `res_addr` parameter.

For complete details on how to create the transfer service code, refer to the online documentation or the source code in the trios/examples directory.

3.2.4 Performance of the transfer service

As mentioned earlier in the text, the

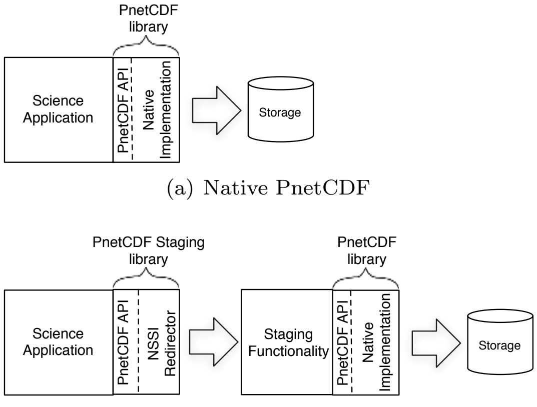
3.3 PnetCDF staging service

Demonstrating the performance and functionality advantages Nessie provides, the NetCDF/PnetCDF link-time replacement library offers a transparent way to use a staging area with hosted data services without disturbing the application source code and not impacting the ultimate data storage format. At a simple level, the library is inserted into the I/O path

```
int xfer_write_rdma_srvr(
        const unsigned long request_id,
        const NNTI_peer_t *caller,
        const xfer_pull_args *args,
        const NNTI_buffer_t *data_addr,
        const NNTI_buffer_t *res_addr)
{
  const int len = args->len;
  int nbytes = len*sizeof(data_t);

  /* allocate space for the buffer */
  data_t *buf = (data_t *)malloc(nbytes);

  /* fetch the data from the client */
  nssi_get_data(caller,buf,nbytes,
       data_addr);

  /* send the result to the client */
  rc = nssi_send_result(caller,request_id,
         NSSI_OK, NULL, res_addr);

  /* free buffer */
  free(buf);
}
```

Figure 9: Server stub for the `xfer_write_rdma` method of the transfer service.

affording redirecting the NetCDF API calls into the staging area for further processing prior to calling the native NetCDF APIs for the ultimate movement of data to storage. This structure is illustrated in Figure 10.



(a) Native PnetCDF

(b) PnetCDF Staging

Figure 10: System Architecture

At a minimum, this architecture affords reducing the number of processes participating in collective coordination operations enhancing scalability [16]. Overall, it affords changing or processing the data prior to writing to storage without impacting the application source code.

The staging functionality can be hosted over any number of processes and nodes as memory and processing capabilities demand. The initial results shown in the Parallel Data Storage Workshop 2011 paper uses a single staging node, but with 12 staing processes on that node. Those processes are capable of coordinating among themselves in order to manipulate the data. Currently there are five data processing modes for the data staging area:

1. *direct* - immediately use the PnetCDF library to execute the request synchronously with the file system

2. *caching independent* - caches the write calls in the staging area until either no more buffer space is available or the file close call is made. At that time, the data is written using an independent IO mode rather than collective IO. This avoids both coordination among the staging processes and any additional data rearrangement prior to movement to storage.

3. *aggregate independent* - similar to caching independent except that the data is aggregated into larger, contiguous chunks as much as possible within all of the server processes on a single compute node prior to writing to storage. That is, to optimize the data rearrangement performance, the movement is restricted to stay within the same node avoiding any network communication overhead.

4. *caching collective* - works like the *chaching indpendent* mode, except that it attempts to use as many collective I/O calls as possible to write the data to storage. If the data payloads are not evenly distributed across all of the staging processes, a number of collective calls corresponding to the number of smallest number of data payloads in any staging process followed by a series of independent calls to complete writing the data.

5. *aggregate collective* - operates as a blend of the *caching collective* in that it tries to use as many collective I/O calls as possible to write the data, but uses the aggregation data pre-processing steps to reduce the number of data packets written.

Unlike many aschronous staging approaches, the PnetCDF staging service ultimately performs synchronously. The call to the file close function blocks until the data has been flushed to storage.

Using the staging service at run time is a 4 step process. First, the staging area is launched generating a list of contact strings. Each string contains the information necessary to reach a single staging process. The client (science application) can choose which client process communicates with which staging service process. Second, these strings are processed to generate a standard XML-based format making client processing simpler and environment variables are set exposing the contact file filename in a standard way. Third, the science application is launched. Finally, as part of the PnetCDF initialization, the re-implementation of the PnetCDF reads the environment variable to determine the connection information file filename, reads the file, and broadcasts the connection information to all of the client processes. These processes select one of the server processes with which to communicate based on a load-balancing calculation.

The current functionality of increasing the performance of PnetCDF collective operations is just a first step. The current architecture offer the ability to have any parallel or serial processing engine installed in the staging area application. The scaling of this application is independent of scaling of the science application. This decoupling of concerns simplifies programming of the integrated workflow of the simulation generating raw data and the analysis routines distilling the data into the desired processed form.

Ultimately, this technique of reimplementing the API for accessing staging offers a way to enhance the functionality of online scientific data processing without requiring changing the application source code. As in the case of the PnetCDF service, these analysis or other data processing routines can be inserted as part of the I/O path with the data ultimately hitting the storage in the format prescribed by the original API.

3.3.1 PnetCDF staging service performance analysis

Evaluating the performance of the service is performed in two parts. First, an examination of IOR [13] performance is evaluated followed by an I/O kernel for Sandia's S3D [5] combustion code.

IOR Performance To evaluate the potential of PnetCDF staging, we measured the performance of our PnetCDF staging library when used by the IOR benchmark code. IOR (Interleave-or-random) [13] is a highly configurable benchmark code from LLNL that IOR is often used to find the peak measurable throughput of an I/O system. In this case, IOR provides a tool for evaluating the impact of offloading the management overhead of the netCDF and PnetCDF libraries onto staging nodes.

Figure 11 shows measured throughput of three different experiments: writing a single shared file using PnetCDF directly, writing a file-per-process using standard netCDF3, and writing a single shared file using the PnetCDF staging service. In every experiment, each client wrote 25% of its compute-node memory, so we allocated one staging node for each four compute nodes to provide enough memory in the staging area to handle an I/O “dump”.

Results on Thunderbird show terrible performance for both the PnetCDF and netCDF file-per-process case when using the library directly. The PnetCDF experiments maxed out at 217 MiB/s and reached the peak almost immediately. The PnetCDF shared file did not do much better, achieving a peak throughput of 3.3 GiB/s after only 10s of clients. The PnetCDF staging service, however, achieved an “effective” I/O rate of 28 GiB/s to a single shared file. This is the rate observed by the application as the time to transfer the data from the application to the set of staging nodes. The staging nodes still have to write the data to storage, but for applications with “bursty” IO patterns, staging is very effective.

S3D Performance In the final set of experiments, we evaluate the performance of the PnetCDF staging library when used by Sandia's S3D simulation code [5], a flow solver for performing direct numerical simulation of turbulent combustion.

All experiments take place on the JaguarPF system at Oak Ridge National Laboratories. JaguarPF is a Cray XT5 with 18,688 compute nodes in addition to dedicated login and service nodes. Each compute node has dual hex-core AMD Opteron 2435 processors running at 2.6GHz, 16 GB RAM, and a SeaStar 2+ router. The PnetCDF version is 1.2.0 and uses the default Cray MPT MPI implementation. The file system, called Spider, is a Lustre 1.6 system with 672 object storage targets and a total of 5 PB of disk space. It has a demonstrated maximum bandwidth of 120 GB/sec. We configured the file system to stripe using the default 1 MB stripe size across 160 storage targets for each file for all tests.

In our test configuration, we use ten, 32 cubes (32×32×32) of doubles per process across a shared, global space. The data size is 2.7 GB per 1024 processes. We write the whole dataset at a single time and measure the time from the file open through the file close. We use five tests for each process count and show the best performance for each size. In this set of tests, we use a single node for staging. To max-

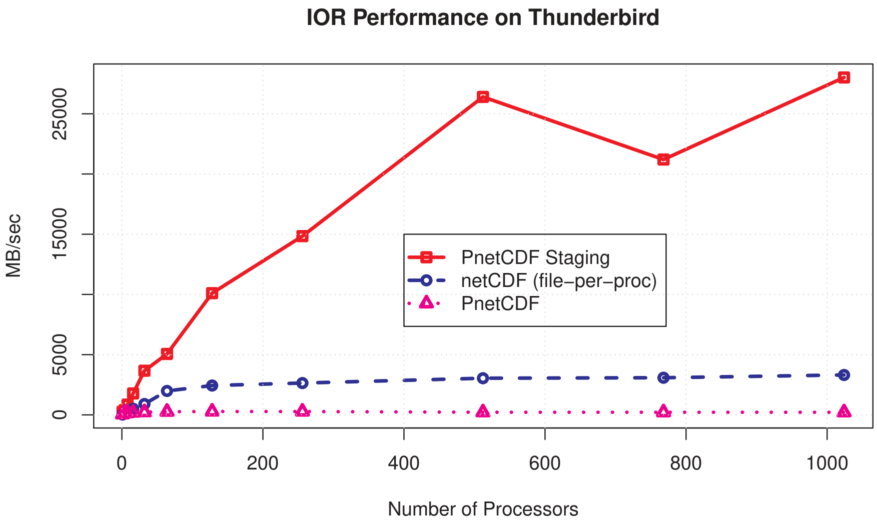


Figure 11: Measured throughput of the IOR benchmark code on Thunderbird

imize the parallel bandwidth to the storage system, one staging process per core is used (12 staging processes). Additional testing with a single staging process did not show significant performance differences. The client processes are split as evenly as possible across the staging processes in an attempt to balance the load.

Figure 12 shows the results of S3D using the PnetCDF library directly with the four different configurations of our PnetCDF staging library described in Section 3.3. In all cases measured, the base PnetCDF performance was no better than any other technique at any process count. The biggest difference between the base performance and one of the techniques is for 1024 processes using the caching independent mode at only 32% as much time spent performing IO. The direct technique starts at about 50% less time spent and steadily increases until it reached parity at 7168 processes. Both cache independent and aggregate independent advantages steadily decrease as the scale increases, but still have a 20% advantage at 8192 processes.

In spite of there only being 12 staging processes with a total gross of 16 GB of RAM, the performance improvement is still significant. The lesser performance of the direct writing method is not very surprising. By making the broadly distributed calls synchronous through just 12 processes, the calling application must wait for the staging area to complete the write call before the next process will attempt to write. The advantage shown for smaller scales shows the disadvantage of the communication to rearrange the data compared to just writing the data. Ultimately, the advantage is overwhelmed by the number of requests being performed synchronously through the limited resources.

The advantage of the caching and aggregating over the direct and base techniques shows that by queueing all of the requests and letting them execute without interruption and delay of returning back to the compute area offers a non-trivial advantage over the synchronous approach. Somewhat surprisingly, the aggregation approach that reduces the number of IO calls via data aggregation did not yield performance advantages over just caching the requests. This suggests that for the configuration of the Spider file system at least, reducing the number of concurrent clients to the IO system is the advantageous approach. Additional efforts to reduce the number of IO calls do not yield benefits.

3.4 CTH in-transit analysis

As an example of using Nessie for in-transit analysis, we implemented an in-transit analysis capability for the CTH shock physics code [14]. For export-control issues, the code is not available in the Trilinos repository. It is included in this document merely as an example.

Much like the PnetCDF staging service, the in-transit CTH analysis service is a drop-in replacement for an already used library. In this case, we implemented client and server stubs for the PVSPY library – an API for performing in-situ analysis using the

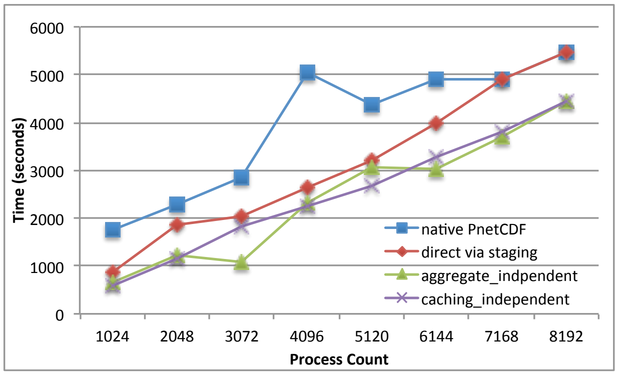


Figure 12: Writing performance on JaguarPF one staging node (12 processes)

ParaView coProcessing libraries [17]. The difference between the in-situ approach and the in-transit approach is that in-situ, meaning "in place", executes the analysis on the same compute nodes as the scientific code. Instead of performing the analysis on the CTH compute nodes, our PVSPY client marshals requests, sends data to the staging nodes, and performs the analysis on the staging nodes. Figure 13 illustrates this process for analysis that does fragment detection.

There are a couple of trade-offs to consider when deciding whether to perform the analysis in-transit or in-situ. First, the in-transit approach allows fragment detection to execute in parallel with CTH, unlike the in-situ approach that requires CTH to wait for the analysis to complete. If the time to execute the analysis code is substantially larger than the time to transfer the raw data to the service, there is a performance advantage to using the in-transit approach.

A second consideration is library scalability. While significant effort has gone into making the CTH code scale to extremely large core counts, not as much effort has gone into scalability of the analysis code. For example, the ParaView coProcessing libraries have not successfully run on more than 32 thoousand cores. Linking CTH to ParaView for in-situ analysis also limits the scalability of the ParaView run. In contrast, the data service will likely use a much smaller number of cores, putting no limitation on the scale of CTH.

Another often overlooked consideration is the memory required to link a large analysis library into a production scientific code. In the in-situ case, CTH has to link ParaView. Since many HPC systems do not efficiently support dynamic libraries[5], the entire static ParaView library has to be linked. On the Cray XE6, the in-situ binary for CTH is 330 MiB, where the in-transit binary for CTH is 30 MiB. That is a substantial difference, especially on systems that are memory limited – as is the case for most multi-core HPC platforms.

For efficiency reasons, our PVSPY client implementation does not simply forward all the functions to the service. In many cases, the client maintains metadata to avoid unnecessary data transfers. For example, the PVSPY API includes "setup" functions for initializing data structures, assigning cell and material field names, and setting cell and material fields pointers. Not all of these functions require an immediate interaction with the data service. In fact, the only operation that requires a bulk data transport is the function to initiate the analysis.

Development and testing of the CTH in-transit service is ongoing. We expect to publish a more complete description along with performance results in the near future.

4 Future Work

4.1 Exodus

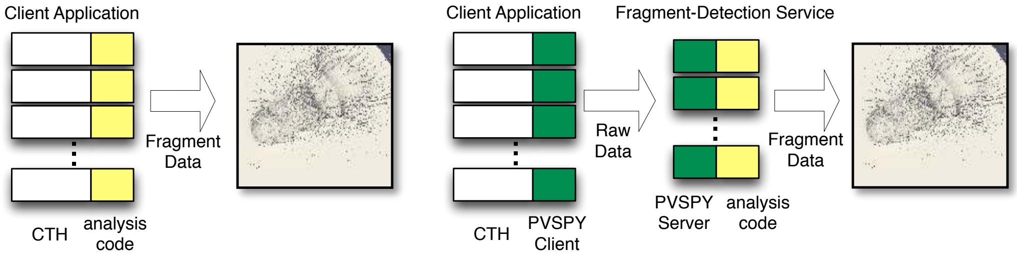
The current Exodus database format has some limitations that will be addressed in the near future.

- The Exodus data model uses 32-bit integers for all ids and offsets which limits the model size to approximately 2.1 billion entities of each type. This limitation is planned to be eliminated soon in a backwardly-compatible manner which will allow existing databases to be accessed by applications using the new API.

- The Exodus data model currently only stores scalar data (X-displacement at a node) and higher-order data structures (vector, tensor, quaternion) are implied via naming conventions. For example, the variables d_x, d_y, d_z would be interpreted by some applications as a 3D vector "d". Native support for higher-order vector, tensor, and quaternion data is planned.

- Store the model hierarchy/part structure in the Exodus datamodel and permit storing of transient data on the parts and assemblies. The current Exodus model is a flat array of named element blocks, sidesets, and nodesets. As models become more complicated, it is necessary to reflect the geometric model assembly structure in the mesh to facilitate visualization and analysis.

[5] Support for dynamic libraries is currently being evaluated for the Cray XE6.



(a) In-situ analysis

(b) In-transit analysis

Figure 13: Comparison of in-situ (a) and in-transit (b) fragment detection for the CTH shock physics code.

- Support for changing topologies in Exodus. The current Exodus requires generating a completely new file every time the model topology changes. This can result in hundreds of "topology-change" files during a routine analysis which can overwhelm filesystems, and more importantly, the analyst. Need to modify the Exodus format to be able to efficiently handle changing model topology. In the short-term, need to develop tools to make the handling of lots of files more efficient for the analyst.
- Better support for Parallel I/O using the parallel capabilities of NetCDF and/or Parallel NetCDF.
- Additional API language support including C++ and Python and improvements to the C and Fortran API.

The Exodus library is expected to evolve to support the ever-increasing data demands of finite element analysis models and codes.

4.2 Comparing in-transit with in-situ

Since there are a number of research projects investigating both in-situ and in-transit approaches, we are interested in doing a thorough performance evaluation between the two approaches. Our decision to develop drop-in replacements for existing libraries makes this type of investigation relatively easy, particularly for the CTH example. In the next year, we expect to perform a detailed performance comparison of CTH in-transit verses in-situ.

5 Summary

This paper describes the new capability area for Trilinos called *Trios*. By providing two sets of functionality, both production quality and experimental, Trios addresses both immediate needs of the Trilinos community and provides a platform for experimentation with new I/O techniques and technologies in a harmonious form with the Trilinos pacakages.

The inclusion of the Exodus foundational API, the Nemesis extensions, and the Sierra C++ wrappers, a variety of interfaces to a standardized NetCDF file format are offered. Much of this technology has been in productive use for a decade or longer proving it is a mature and useful product.

The more recent developments of the Nessie framework affords experimentation with new I/O techniques including easier access to staging as well as a transparent way to incorporate ‘in flight’ data processing between the science application and storage.

In combination, these technologies provide both a mature, proven API and file format in use by many science codes as well as interesting technology that is proving to provide ways to enhance the scalabilty and richness of the I/O path.

Continuing developments in both the mature tools and the experimental platforms will continue to enhance both the usability and usefulness of Trios to the greater Trilinos community.

References

[1] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *Proceedings of the 18th Annual Symposium on High Performance Interconnects (HOTI)*. IEEE Computer Society Press, August 2010.

[2] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2, October 2004.

[3] Ron Brightwell, Kevin Pedretti, Keith Underwood, and Trammell Hudson. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[4] Ron Brightwell, Rolf Riesen, Bill Lawry, and Arther B. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2002.

[5] J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor-Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):31pp, 2009.

[6] Jerry A. Clarke and Eric R. Mark. Enhancements to the eXtensible Data Model and format (XDMF). In *Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference*, pages 322–327, June 2007.

[7] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, March/April 2002.

[8] Ian Foster, David Kohr, Jr., Rakesh Krishnaiyer, and Jace Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25, San Jose, CA, November 1997. ACM Press.

[9] Jing Fu, Ning Liu, O. Sahni, K.E. Jansen, M.S. Shephard, and C.D. Carothers. Scalable parallel I/O alternatives for massively parallel partitioned solver systems. In *International Parallel and Distributed Processing Symposium, Workshops and PhD Forum*, Atlanta, GA, April 2010.

[10] Bruce Hendrickson and Robert Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1994.

[11] Gary L. Hennigan and John N. Shadid. NEMESIS I: A set of functions for describing unstructured finite-element data on parallel computers. Technical report, Sandia National Laboratories, December 1998.

[12] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[13] IOR interleaved or random HPC benchmark. http://sourceforge.net/projects/ior-sio/.

[14] E. S. Hertel Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In R. Brun and L.D. Dumitrescu, editors, *Proceedings of the 19'th International Symposium on Shock Physics*, volume 1, pages 377–382, Marseille, France, July 1993.

[15] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 35, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, 2001.

[16] Jay Lofstead, Ron Oldfield, Todd Kordenbrock, and Charles Reiss. Extending scalability of collective I/O through nessie and staging. In *Proceedings of the 6th Parallel Data Storage Workshop*, November 2011.

[17] Kenneth Moreland, Nathan Fabian, Pat Marion, and Berk Geveci. Visualization on supercomputing platform level II ASC milestone (3537-1b) results from Sandia. Technical Report SAND2010-6118, Sandia National Laboratories, September 2010.

[18] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Joudain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, Michael E. Papka, and Scott Klasky. Examples of in transit visualization. In *Proceedings of the PDAC 2011 : 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, November 2011. Submitted.

[19] Ron A. Oldfield. Lightweight storage and overlay networks for fault tolerance. Technical Report SAND2010-0040, Sandia National Laboratories, Albuquerque, NM, January 2010.

[20] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Rolf Riesen, Maria Ruiz Varela, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, San Diego, CA, September 2007.

[21] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[22] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight I/O. In *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*, Barcelona, Spain, September 2006.

[23] Ron A. Oldfield, Andrew Wilson, George Davidson, and Craig Ulmer. Access to external resources using service-node proxies. In *Proceedings of the Cray User Group Meeting*, Atlanta, GA, May 2009.

[24] Charles Reiss, Gerald Lofstead, and Ron Oldfield. Implementation and evaluation of a staging proxy for checkpoint I/O. Technical report, Sandia National Laboratories, Albuquerque, NM, August 2008.

[25] Russ Rew, Glenn Davis, Steve Emmerson, and Harvey Davies. *The NetCDF Users Guide: Data Model, Programming Interfaces, and Format for Self-Describing, Portable Data.* Unidata Program Center, version 4.1.3 edition, June 2011.

[26] Rolf Riesen, Ron Brightwell, Patrick Bridges, Trammell Hudson, Arthur Maccabe, Patrick Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, August 2008.

[27] Larry A. Schoof and Victor R. Yarberry. EXODUS II: A finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, Albuquerque, New Mexico 87185, 1992.

[28] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings ofSupercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[29] Gregory D. Sjaardema. Overview of the Sandia National Laboratories engineering analysis code access system (seacas). Technical Report SAND92-2292, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, January 1993.

[30] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Scott Klasky, Qing Liu, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. PreDatA - preparatory data analytics on Peta-Scale machines. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–12, April 2010.