Developing Integrated Data Services for Cray Systems with a Gemini Interconnect

Ron A. Oldfield Todd Kordenbrock Jay Lofstead

May 1, 2012

Abstract

Over the past several years, there has been increasing interest in injecting a layer of compute resources between a high-performance computing application and the end storage devices. For some projects, the objective is to present the parallel file system with a reduced set of clients, making it easier for file-system vendors to support extreme-scale systems. In other cases, the objective is to use these resources as "staging areas" to aggregate data or cache bursts of I/O operations. Still others use these staging areas for "in-situ" analysis on data in-transit between the application and the storage system. To simplify our discussion, we adopt the general term "Integrated Data Services" to represent these use-cases. This paper describes how we provide user-level, integrated data services for Cray systems that use the Gemini Interconnect. In particular, we describe our implementation and performance results on the Cray XE6, Cielo, at Los Alamos National Laboratory.

1 Introduction

In our quest toward exascale systems and applications, one topic that is frequently discussed is the need for more flexible execution models. Current models for capability class high-performance computing (HPC) systems are essentially static, requiring applications and analysis to execute independently storing intermediate results on a persistent, globally accessible parallel file system. For example, in fusion science, simulation of the edge of the plasma [17] and the interior of the plasma [39] are currently separate simulations. To have a more complete, accurate model for a fusion reactor, these components need to be tightly coupled to share the effects between the two models. The CESM climate model [24] is similar in that it incorporates atmosphere, ocean, land surface, sea ice, and land ice through a coupling engine to manage the interactions between each of these different systems yielding a more accurate model of global climate. In most cases, these and other scientific applications are part of larger offline workflows that process the output written to storage in phases that ultimately yield insights into the phenomena being studied.

For exascale systems, there is a general belief that systems will need more flexible execution models that allow the coupling of simulations and/or analysis. This coupling promises to reduce the I/O burden on the file system and potentially improve overall efficency of the application workflow. Current work to enable these coupling and workflow scenarios are focused on the data issues to resolve resolution and mesh mismatches, time scale mismatches, and make data available through data staging techniques [20, 34, 21, 11, 2, 40, 10].

In this paper, we describe R&D efforts and challenges of coupling simulation codes and analysis on Cray systems with Gemini Networks. In particular, we describe how to create and use *Integrated Data Services* on the Cray XE6 platform using Sandia's *Network Scalable Service Interface* (Nessie) [20]. An integrated data service, illustrated in Figure 1, is a separate (possibly parallel) application that performs operations on behalf of an actively running scientific application. Nessie is a framework for developing data services on HPC platforms. It provides portable interfaces for inter-application communication across RDMA-based networks, an RPC-like abstraction for rapidly developing client and server stubs, and a portable model for defining serializable data structures for data transfer.

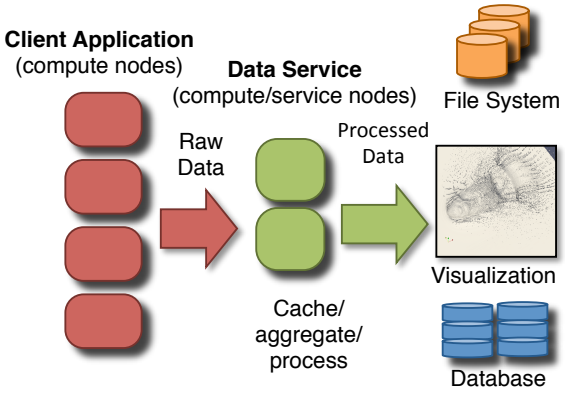


Figure 1: Data services uses additional compute resources to perform operations on behalf of an HPC application.

2 Background and Related Work

There are a number of ongoing efforts to integrate or couple simulation, analysis, and visualization. The approaches are generally categorized into two areas: in-situ and in-transit [23]. The term *in-situ* applies to codes that perform the analysis or visualization of data with the simulation that generates the data. In-situ libraries link to the main code and execute through library calls. This concept was first mentioned in the 1987 National Science Foundation Visualization in Scientific Computing workshop report [22]; however, interest in in-situ has grown significantly in recent years and is rapidly becoming one of the most important topics in large-scale visualization [15, 4].

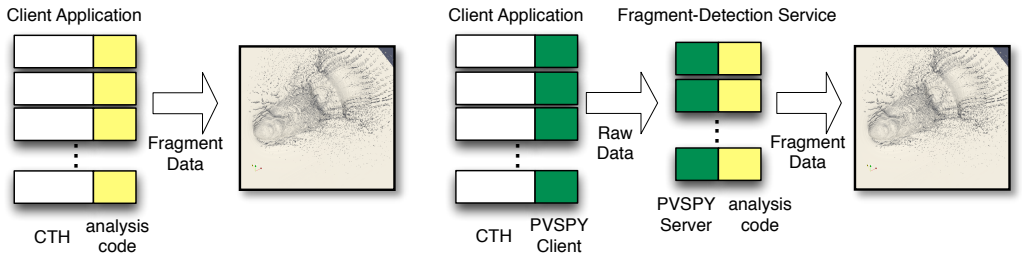
One reason in-situ analysis is attractive for petascale systems is that the cost of dedicated interactive visualization computers for petascale computing is prohibitive [9]. Developing algorithms and techniqes that work directly with the simulation code reduces, but does not eliminate, the need for specialized visualization hardware. Other studies show that the I/O cost of writing and reading data from parallel file systems is beginning to dominate the time spent in both the simulation and visualization [36, 33]. In-situ visualization eliminates this I/O cost by performing the visualization in the memory of the scientific code.

One downside of in-situ analysis is that the algorithms to perform analysis may not scale as well as the scientific code, creating a significant bottleneck for the overall runtime. This is partly because the algorithms and codes for visualization were not designed for large-scale HPC systems– we expect some of these issues to be resolved as visualization experts become more accustomed to large-scale parallel programming. In some cases, however, the communication requirements of an analysis algorithm are not appropriate for extreme-scale, and thus in-situ for capability-class applications.

In transit analysis (also known as *staged analysis*) is similar to in-situ analysis in that the analysis code runs concurrently with the simulation code. The difference is that the analysis takes place on different compute resources than the simulation. Figure 2 illustrates this by looking at in-situ and in-transit analysis for Sandia's CTH shock physics code. A physical partitioning of the simulation and analysis codes allows the analysis to execute in a pipeline-parallel fashion with minimal interference on the parallel simulation. In some cases, the staging nodes simply provide data caching in the network to provide a buffer for bursty I/O operations [29, 34, 2, 25]. In this case, the staaging area captures data from the application, then writes it to storage while the application continues to compute, effectively trading the cost of writing to a storage system with the cost of writing memory-to-memory through the high-speed interconnect [29]. There are also a number of examples of using staging areas for statistical analysis, indexing, feature extraction [23], FFTs [32], and data permutations [26].

Some operations can be performed either in-situ or in-transit based on both the resource availability (memory and computation) and communication requirements. ADIOS [19] introduced the idea of Data Characteristics [21] as a way to represent a local portion of statistics intended to either be used to gain knowledge about that portion of the data or to used in aggregate to learn about the data set as a whole. The placement of these operators was examined in PreDatA [40]. Other efforts to accelerate the use of data include FastBit [14] to generate a bitmap-based index for data values. This approach yielded a relatively compact index with fast access to elements. The ADIOS data characteristics have also been extended from just the minimum and maximum value to include count, sum, sum of squares, histogram, and is it an infinity or not [12]. Another use of in-transit processing is for data compression, such as is done in ISABELA [18].

Another primary difference between in-situ and in-transit analysis is that in-transit analysis requires the ability to transfer data from the scientific code to the "staging" area for analysis. In techniques such as I/O Delegation[25] the applications uses MPI to communicate this data. For I/O Delegation, the user allocates an additional set of staging processors when it launches the application. A separate MPI com-



(a) In-situ analysis

(b) In-transit analysis

Figure 2: Comparison of in-situ (a) and in-transit (b) fragment detection for the CTH shock physics code.

municator allows the staging processors to perform analysis without interfering with the primary application. This approach was first demonstrated for high-performance computing in a seismic imaging application called Salvo[32]. In Salvo, the user allocated an "I/O Partition" for staging outgoing data and also performing proprocessing (i.e., FFTs) on incoming data. I/O delegation is perhaps the most portable approach for in transit computation, but it requires a tight coupling of analysis with application and it is difficult to share the service with multiple applications.

A second approach for in transit analysis is to create the staging area as a separate application that communicates with the client application This approach is extremely flexible because it allows for the potential "chaining" of application services, coupling of applications, and application sharing. This is the approach we take with Nessie. One problem with this approach is that it requires the transport mechanism to use the low-level network transport of the host machine for performance and efficiency. For example, on Cray XT systems, the transport uses Portals; on Cray XE systems, the transport uses uGNI; on IBM BlueGene systems, the transport uses the *Deep Computing Message Facility* (DCMF), and on generic HPC clusters, the transport uses InfiniBand. As we discuss in Section 4.1, Nessie provides a portable abstraction layer that supports all these low-level transports.

Another challenge for in-transit approaches is managing allocation and placement of services with respect to the application. On InfiniBand systems, such as Sandia's RedSky cluster, the user-developed launch script (i.e., batch script) must be explicit about placement to avoid sharing nodes with the application, thus losing the ability to manage large buffers for staging. In addition, placement matters, as shown by an earlier study of data staging for checkpoints [28]. As we discuss in Section 4.2, inter-application communication is also restricted by the security model on some systems, such as the Cray XE6.

Data services, as defined in this paper, are a general form of in-transit computing. They leverage additional resources for analysis, management, and staging of data. For example, the Nessie service developed for the Lightweight file system (LWFS) provided authentication, authorization, and storage [30]. The Dataspaces work is also more than just "in-transit" analysis [10]. They leverage memory in unused compute nodes to manage a distributed shared cache for coupled applications.

The two projects most similar to Nessie are DataStager [2] and GLEAN [38]. All provide a mechanism for data transport and serialization, and all have been demonstrated to enable data staging and analysis capabilities. Nessie, however, is the only one with support for all the major HPC platforms. GLEAN is exclusively used on BG/P systems, and DataStager has ports for SeaStar (Portals3) and InfiniBand, but they are planning to use Nessie's RDMA abstraction library NNTI (see Section 4.1) for access to Gemini networks.

Finally, there are some interesting approaches that are effectively a hybrid of in-situ, in-transit. These projects use logic and system characterizations to decide where data-analysis code should execute. Some early work, for example, represented data-flow computations for computational grid applications as a series-parallel graph that could be optimized in two phases: the first to increase parallelism, the second to decide placement based on data flow and locality [27]. More recent work is using a combination of scheduling, dynamic code generation, and fast serialization technology to deploy "just in time" analysis capabilities that execute in-situ or in-transit, depending on a number of different parameters [1]

3 Nessie

The NEtwork Scalable Service Interface, or Nessie, is a framework for developing parallel client-server data services for large-scale HPC systems [20, 31]. Nessie was originally developed out of necessity for the Lightweight File Systems (LWFS) project [30], a joint effort between researchers at Sandia National Laboratories and the University of New Mexico. The LWFS project followed the basic philosophy of "simplicity enables scalability", the foundation of earlier work on lightweight operating system kernels at Sandia [35]. 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 heterogeneous architectures. Features of particular note include

1. asynchronous methods for most of the interface to prevent client blocking while the service processes a request;

2. a server-directed approach to efficiently manage network bandwidth between the client and servers;

3. separate channels for control and data traffic; and

4. XDR encoding for the control messages (i.e., requests and results) to support heterogeneous 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 (SNL), the Cray XT4/5 systems at Oak Ridge National Laboratory, and a large InfiniBand cluster at SNL. The Nessie RPC layer has direct support of Cray's SeaStar interconnect [7], through the Portals API [8]; Cray's Gemini interconnect [5]; and InfiniBand [6].

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.

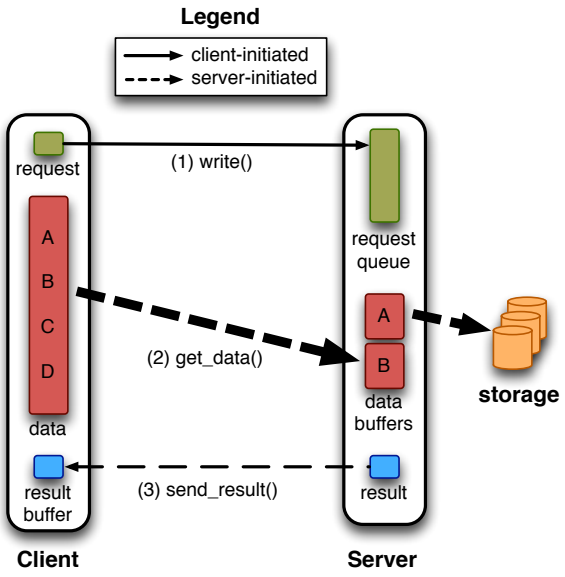


Figure 3: Conceptual network protocol for a Nessie storage server executing a write request. The initial request tells the server the operation and the location of the client buffers. The server fetches the data through RDMA get commands until it has satisfied the request. After completing the data transfers, the server sends a small "result" object back to the client indicating success or failure.

To address the issue of efficient transport for bulk data, Nessie provides separate communication channels for control and data messages. In this model, a "control" message, also known as a request, is typically small. It identifies the operation to perform, where to get arguments, the structure of the arguments, and perhaps the data itself (if the data is small enough to fit in the fixed-sized request). 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 3 shows the transport protocol for an I/O server executing a write request.

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 [16, 37]. 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, 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 [16].

While it is not strictly necessary on systems that have homogenous clients and servers, we use XDR encoding to provide portable serialization of arguments for the "control" messages and arguments. This was a design decision made early in the project that allow the client to send arbitrary C-like data structures to the server with minimal development effort. At the time, we were implementing file services for a system where the service nodes were a different architecture (and had different endienness) than the compute nodes. In this case, byte-swaps were necessary for the control structures. Since *rpcgen*, the function that generates the serialization code is pervasive in Unix environments and has been in use for more than a decade, it was the logical choice for argument marshaling. In addition, as will be shown in the Section 6, the overhead of XDR is minimal for implementations that make extensive use of the data channel for bulk data.

4 Cray XE6 Implementation

The Nessie implementation uses remote direct memory access (RDMA) to transport data for all operations. To portably support the RDMA methods used by Nessie, the implementation is built on an abstract RDMA network interface called the *Nessie Network Transport Interface* (NNTI). Each supported network transport has a custom implementation of NNTI. We currently have implementations for the Seastar (based on Portals3), InfiniBand, and Gemini interconnects. This section details the implementation of NNTI for the Gemini interconnect.

4.1 Gemini implementation of NNTI

The NNTI interface has four primary roles: initialize the interface, connect to a remote node, register/unregister memory, and transport data. The functions that support these roles and the details of their implementation of the Gemini interconnect are shown below.

4.1.1 Initialize the interface

Every low-level interconnect has an interface for initializing and cleaning up the library that provides access to the transport methods. The `NNTI_init` and `NNTI_fini` functions provide wrappers for these functions.

```
NNTI_result_t NNTI_init (
        const NNTI_transport_id_t  trans_id,
        const char                 *my_url,
        NNTI_transport_t           *trans_hdl);

NNTI_result_t NNTI_get_url (
        const NNTI_transport_t *trans_hdl,
        char                   *url,
        const uint64_t          maxlen);

NNTI_result_t NNTI_fini (
        const NNTI_transport_t *trans_hdl);
```

On Gemini systems, the `NNTI_init` function must perform four steps to intialize.

First, `NNTI_init` must collect the parameters necessary to the initialize uGNI. The ALPS job launch system assigns Gemini network and security parameters during launch. The ALPS low-level interface client library (libalpslli.a) provides a simple request/reply API to retrieve these Gemini parameters. The parameters include the network device ID, the local NIC address, the application's cookie and protection tag (pTag). In addition to the application's cookie and pTag, each process requires a unique instance ID to identify itself within a communication domain. `NNTI_init` creates the instance ID using the NIC address from `GNI_CdmGetNicAddress` and the core assigned by the scheduler.

The second step is to create a communication domain (`GNI_CdmCreate`) and attach it to the network device (`GNI_CdmAttach`).

The third step is to create a connection listener thread. NNTI uses TCP to bootstrap communications over the Gemini interconnect. Each NNTI process creates a socket that is monitored for connection requests by a background thread.

The fourth step is to create a peer identifier that uniquely identifies this process to all other uGNI processes.

`NNTI_get_url` is a convenience function that returns the string representation of a peer. The string is a URL of the form `gni://hostname:port/?ptag=<ALPSptag>&cookie=<ALPScookie>`. An NNTI process that wishes to be contacted by other NNTI processes should publish the URL in a globally accessible location (a file in a global filesystem, namespace service, etc).

On Gemini systems, the `NNTI_fini` function releases

resources that are allocated during GNI_init. All open connections are closed, the listener thread is terminated and the communication domain is destroyed (GNI_CdmDestroy).

4.1.2 Connect to a remote node

Each transport has methods used to verify that we can transfer data to/from a remote node. Some, like Portals3 [8], do not require persistent connections, but instead "ping" the remote node to make sure it can responde to PUT and GET requests. The NNTI wrappers for these functions are the NNTI_connect and NNTI_disconnect functions.

```
NNTI_result_t NNTI_connect (
        const NNTI_transport_t *trans_hdl,
        const char             *url,
        const int               timeout,
        NNTI_peer_t            *peer_hdl);

NNTI_result_t NNTI_disconnect (
        const NNTI_transport_t *trans_hdl,
        NNTI_peer_t            *peer_hdl);
```

The NNTI_connect function bootstraps communication between two NNTI processes. The URL is parsed to determine the peer's location on the network, a TCP connection is established and the processes exchange TCP and ALPS parameters. Both processes inspect the ALPS parameters to ensure that they have the same cookie and protection tag (pTag). If the cookie and pTag do not match, the processes will be in different communication domains and uGNI will not allow them to communicate. Next the server sends it's request queue parameters to the client and the client responds with flow control parameters. Finally, both sides close the TCP connection and transition the NNTI connection to the ready state.

The NNTI_disconnect function releases all resources allocated while establishing the connection. After disconnecting, this process will be unable to communicate with the peer.

4.1.3 Register memory

In general, RDMA transports require that memory be registered before it can be operated on directly by the NIC. When memory is registered, the pages are pinned in RAM so that they do not get swapped out. Registration also gives the process the opportunity to place restrictions on the types of operations allowed. The NNTI wrappers for these functions are the NNTI_register_memory and NNTI_unregister_memory functions.

```
NNTI_result_t NNTI_register_memory (
        const NNTI_transport_t *trans_hdl,
        char                   *buffer,
        const uint64_t          element_size,
        const uint64_t          num_elements,
        const NNTI_buf_ops_t    ops,
        const NNTI_peer_t      *peer,
        NNTI_buffer_t          *reg_buf);

NNTI_result_t NNTI_unregister_memory (
        NNTI_buffer_t    *reg_buf);
```

The NNTI_register_memory function goes beyond simple registration with GNI_MemRegister. NNTI_register_memory uses the ops parameter to determine the buffer's purpose and builds an NNTI_buffer_t around the memory region. Each registered buffer has an associated work request queue that keeps an ordered list of work requests. Given the asynchronous nature of Nessie and the adaptive routing feature of Gemini, work requests may not comlpete in the order they were issued.

NNTI_register_memory separates memory regions into 3 categories - initiators, targets and receive queues.

Initiator buffers are local to the initiator of the RDMA operation. Initiator buffers are NNTI_SEND_SRC, NNTI_PUT_SRC and NNTI_GET_DST. When the operation is initiated, a work request will be pushed on to the work request queue.

Target buffers are local to the target of the RDMA operation. Target buffers are NNTI_RECV_DST, NNTI_PUT_DST and NNTI_GET_SRC. At the time of registration, a work request is pushed on to the work request queue in anticipation of a future operation by an initiator.

A receive queue (NNTI_RECV_QUEUE) is a specialization of a target buffer. The memory region is divided into elements of equal size. Each send to the queue causes the offset to be automatically incremented by the element size. The offset wraps around to zero when the last element is written. At the time of registration, a work request is added to the work request queue for each element of the queue.

The NNTI_unregister_memory function cancels any outstanding work requests and deregisters the memory region (GNI_MemDeregister).

4.1.4 Transport data

All RDMA transports support one-sided PUT/GET operations and some mechanism for checking for completion of an operation. The NNTI API supports these operations with the NNTI_put, NNTI_get and NNTI_wait functions.

```
NNTI_result_t NNTI_send (
```

```
        const NNTI_peer_t    *peer_hdl,
        const NNTI_buffer_t  *msg_hdl,
        const NNTI_buffer_t  *dest_hdl);

NNTI_result_t NNTI_put (
        const NNTI_buffer_t  *src_buffer_hdl,
        const uint64_t        src_offset,
        const uint64_t        src_length,
        const NNTI_buffer_t  *dest_buffer_hdl,
        const uint64_t        dest_offset);

NNTI_result_t NNTI_get (
        const NNTI_buffer_t  *src_buffer_hdl,
        const uint64_t        src_offset,
        const uint64_t        src_length,
        const NNTI_buffer_t  *dest_buffer_hdl,
        const uint64_t        dest_offset);

NNTI_result_t NNTI_wait (
        const NNTI_buffer_t   *reg_buf,
        const NNTI_buf_ops_t   remote_op,
        const int              timeout,
        NNTI_status_t         *status);
```

The `NNTI_send` function transfers the entire `msg_hdl` buffer to `dest_hdl`. It is assumed that `dest_hdl` is at least as large as `msg_hdl`. If `dest_hdl` is not specified, the data is sent to the peer's receive queue. `NNTI_send` is different from `NNTI_put`, because `NNTI_send` always generates events on both the initator buffer and the target buffer. `NNTI_send` is intended for small buffers, so the FMA Put mechanism is used for all `NNTI_send` operations.

The `NNTI_put` and `NNTI_get` functions operate in essentially the same way with the only difference being the direction of the transfer. Each function transfers some portion of the `src_buffer_hdl` buffer into the `dest_buffer_hdl` buffer. `NNTI_put` transfers data from a local source into a remote destination. `NNTI_get` transfers data from a remote source into a local destination. The size of the transfer as well as the source and destination offsets are specified by the caller. The Gemini transport uses the Fast Memory Access (FMA) mechanism for small transfers and switches to Block Transfer Engine (BTE) mechanism for transfers larger than 4096 bytes. `NNTI_put` and `NNTI_get` always generate events on the initiator buffer. In NNTI's default configuration, events are also generated on the target buffer. For truely one-sided operation, NNTI can be configured to suppress events on the target buffer.

The completion queue events that uGNI generates for memory handles do not include sufficient information for target processes to determine the exact operation that occured or the parameters of that operation. In those cases where the target needs this additional information, NNTI sends a work completion message that specifies the instance ID of the initiator, the NNTI operation, the number of bytes transfered, the source offset and the destination offset.

The `NNTI_wait` function waits for the completion of the oldest operation on `reg_buf` and returns the status of the operation. NNTI maintains a queue of work requests for each `NNTI_buffer_t`. `NNTI_wait` processes uGNI completion events as they occur regardless of the associated buffer. `NNTI_wait` only returns when the oldest `reg_buf` work request completes or the timeout expires. The status includes the result code, the starting address of the memory region, the offset at which the operation occurred and the number of bytes affected by the operation.

There are also functions to wait for events on "any" (`NNTI_waitany`) and "all" (`NNTI_waitall`) buffers. Details of these functions are not necessary for this paper.

4.2 Enabling Inter-Application Communication on Gemini Networks

Cray uses a *Communication Domain* (CDM) construct to prevent arbitrary RDMA access from other applications [13]. Each process in a job shares an agreed upon *protection tag* (ptag) that is assigned by the ALPS job launch system, and peers with different ptags are not allowed to communicate. On production systems, the ptag assigned to each job is unique, preventing user-space applications from communicating with each other, even when the applications are owned by the same user. This is a big problem for deploying data services and coupling codes that are instantiated as independent jobs. 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.

The *CommSplitter* library was designed 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 "splits" 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.

5 A Simple Data-Transfer Service

This section 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.

5.1 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 through the procedure arguments, forcing the client to encode the array before sending and the server to decode the array when receiving. We use this method to evaluate 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_data` function. This method evaluates the RDMA transfer performance for the `nssi_get_data` function.

```
/* 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 4: Portion of the XDR file used for a data-transfer service.

`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 has to define these data structures in an XDR file. Figure 4 shows a portion of the XDR file used for the data-transfer example. XDR data structure 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

```
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_WRITE_RDMA_OP,
    &args, (char *)buf, nbytes,
    NULL, req);
}
```

Figure 5: Client stub for the `xfer_write_rdma` method of the transfer service.

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

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.

5.2 Implementing the client stubs

The client stubs provide an interface between the client application and the remote service. In most cases, the client 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 transfer operation. The details of converting the buffer pointers to memory descriptors for a specific data transport (e.g., InfiniBand, Portals, Gemini) are hidden from the user.

Figure 5 shows the client stub for the `xfer_write_rdma` method. Since the `nssi_call_rpc` method is asynchronous, the client has to check for completion of the operation by calling the `nssi_wait` or `nssi_test` method with the `nssi_request` as an argument.

5.3 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 6. 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.

6 Evaluation

As mentioned earlier in the text, we often use the transfer service as a microbenchmark to evaluate the correctness and performance of the Nessie network protocols. In this section, we specifically focus on performance on Gemini-based sytems, particularly the Cray XE6 Cielo system shared by Los Alamos National Laboratory (LANL) and Sandia National Laboratories (SNL).

The first result evaluates the overhead of the XDR encoding scheme used by Nessie by comparing the performance of `xfer_write_encode` with `xfer_write_rdma`. As we discuss, Nessie provides a data path for bulk transport of raw data, but it is possible to pass all

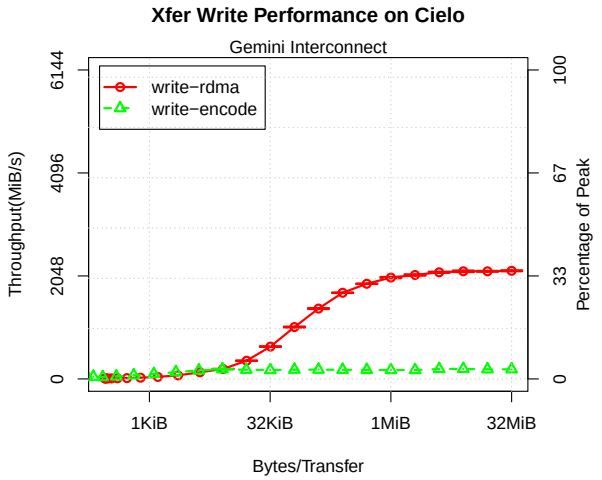


Figure 7: Comparison of `xfer-write-encode` and `xfer-write-rdma` on the Cray XE6 platform using the Gemini network transport.

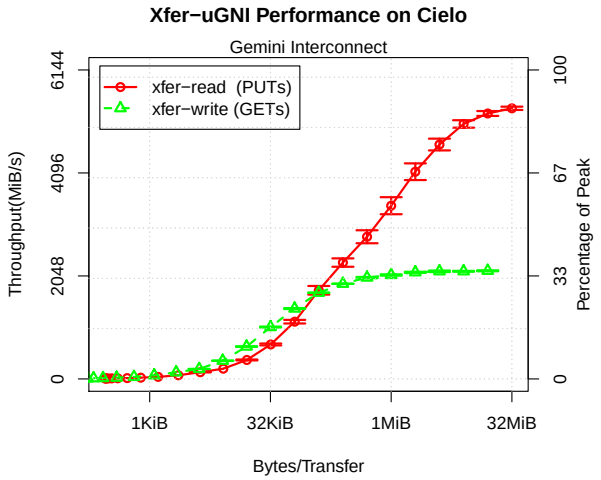


Figure 8: Comparison of `xfer_write_rdma` and `xfer_read_rdma` on the Cray XE6 platform.

data through the request arguments, forcing the XDR encoding of all transmitted data. As Figure 7 shows, the overhead of encoding is quite large, which is why we recommend passing as much data as possible through the data port. The `xfer_write_rdma` method, should have very little encoding overhead–just the cost of encoding the request, so we were surprised that it did not achieve better performance.

To further evaluate issues with the `xfer_write_rdma` performance, Figure 8 shows the performance of the `xfer_read_rdma` compared with `xfer_write_rdma`. This experiment essentially evaluates the performance difference between the uGNI implementation of the one-sided RDMA operations for PUT and GET. Since all bulk data-transfer operations are server directed, a `xfer_read_rdma` executes PUT rdma request from the server to the client. Similarly, the `xfer_write_rdma` executes an GET operation, pulling the data from the client to the server.

The results of Figure 8 show that while reads scale fairly well, the writes only achieve about 1/3 of the peak performance of the network. While the Gemini port still requires some tuning, we did not expect to see such a large difference between reads (PUTS) and writes (GETs). We did not see this same difference in the InfiniBand and SeaStar ports for example, illustrated in Figures 9(a)-and-9(b).

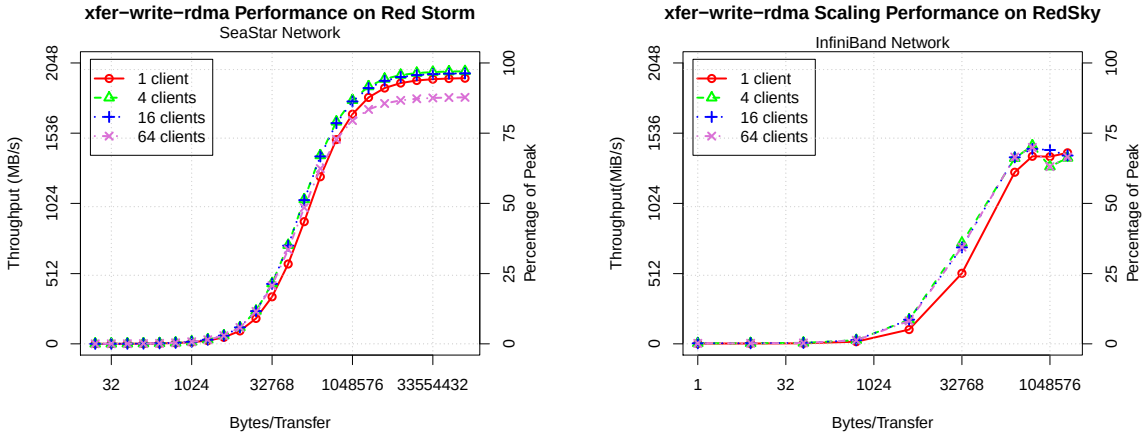
As another point of reference, we also implemented a pure MPI version of transfer tests that uses `MPI_Send`, and the one-sided `MPI_Put`, and `MPI_Get` functions. Since we expect the MPI functions to be highly tuned for the system (at least the two-sided functions), a comparison to MPI should give us an idea about peak achievable performance. Figure 10 shows these results. As expected, the implementation that uses `MPI_Send` performs fairly well, but the one-sided functions perform poorly. With all the synchronization mechanisms implicit in the use of MPI one-sided calls, and the fact that they are not highly used–inferring that tuning the one-sided functions may not be a high priority–we are not particularly surprised by these results.

It is clear that we need to further evaluate the reason behind the poor performance in our use of the GET operation on Gemini systems. The `xfer_write_rdma` operation represents an important use case for our applications, as writes are extremely important in large-scale applications, particularly for resilience (e.g., checkpoints). If we cannot improve the GET performance, we may have to resort to rewriting the NTTI implementation to use two-pass protocol where the first operation tells the server to prepare the buffers. In the second operation, the server would tell the client when to PUT the data to the server-side buffers.

7 Future Work

While current systems are able to support data services, there are a number of improvements to system software, programming models, and resilience that could dramatically improve their utility.

The largest system-software change that would improve the use of data services is that processor



(a) RedStorm (Portals)

(b) RedSky (InfiniBand)

Figure 9: Performance of `xfer_write_rdma` on Seastar (Portals) and InfiniBand interconnects. In these experiments, we evaluated performance as the number of clients-per-server ranges from 1–64.

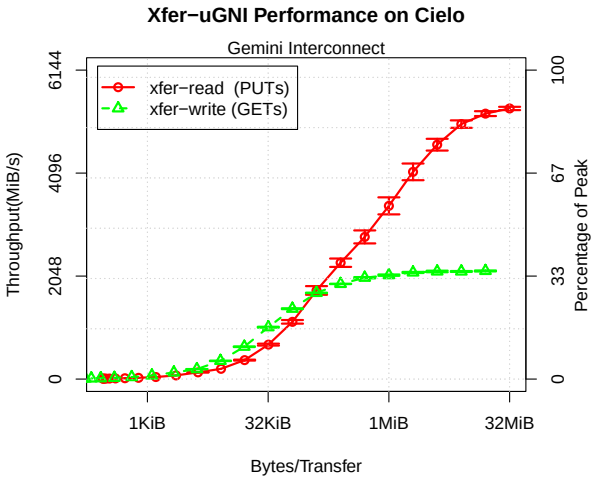


Figure 10: Comparison of `xfer_write_rdma` and `xfer_read_rdma` on the Cray XE6 platform.

scheduling mechanism. Large-scale HPC systems primarily use a static approach to resource scheduling. Nodes are typically allocated using a batch scheduler and the algorithms for placement within those nodes are often tuned for application communication patterns. A more flexible, dynamic, scheduling approach that allows for dynamic allocation and reconfiguration would allow the application to deploy services on demand and tune the service size to balance the resources. In addition, placement of services is important to avoid network contention between the data service and the application [28, 3].

There is also opportunity for improvements in programming models to support data services. First, there are no *standard* interfaces for RDMA-based inter-application communication among the HPC interconnect vendors. Much of our development work with NTTI is to provide this capability. In addition, there are no standard programming approaches for services themselves. Each research project has their own ad-hoc approach that works, but it seems that a more standard approach could improve deployment in production environments.

Finally, the use of data services creates a resilience issue that needs to be addressed. We are promoting the change from appliation work flows that store intermediate results in storage to a data service model where intermediate data is stored in memory of other nodes. This creates consistency and resilience issues when there is a failure in the data service. Sandia is actively involved in addressing this issue.

8 Summary

The increasing interest in using data services, data staging, and in-situ analysis is clearly a sign that these types of approaches are expected to be common on next-generation, and even current-generation systems. This paper presents one approach, Nessie, and describes its performance on the Cray XE6 system.

Our use of MPMD mode to enable data services is unique among data-service and data staging projects. In this case, it was necessary to enable us to overcome the security-model constraints inherent in the Gemini network. However, MPMD may turn out to be a convenient and portable way to deploy data services in production environments. Our CommSplitter library allows legacy applications to use data services with almost no change to the source, and we are working on a pure MPI implementation of NTTI that would enable us to take advantage of highly tuned MPI implementations.

We also identified some performance issues related to the uGNI GET operation. As the GET operation is critical for efficient writes using a server-directed approach, we expect to work with Cray to figure out the issue, and possibly put sufficient time toward rewriting NTTI in a way that avoids the GET operation.

9 Acknowledgements

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This work of authorship was prepared as an account of work sponsored by an agency of the United States Government. Accordingly, the United States Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so for United States Government purposes. Neither Sandia Corporation, the United States Government, nor any agency thereof, nor any of their employees makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by Sandia Corporation, the United States Government, or any agency thereof. The views and opinions expressed herein do not necessarily state or reflect those of Sandia Corporation, the United States Government or any agency thereof.

References

[1] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, Karsten Schwan, and Scott Klasky. Just in time: adding value to the io pipelines of high performance applications with jitstaging. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 27–36, New York, NY, USA, 2011. ACM.

[2] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. In *Proceedings of the 18th IEEE International Symposium on High Performance Distributed Computing*, pages 39–48, Garching, Germany, 2009. ACM Press.

[3] Mohammad H. Abbasi. *Data Services: Bringing I/O Processing to Petascale*. PhD thesis, Georgia Institute of Technology, May 2011.

[4] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, et al. Scientific discovery at the exascale. Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, February 2011.

[5] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *Proceedings of the 18th Annual Symposium on High Performance Interconnects (HOTI)*, pages 83–87, Mountain View, CA, August 2010. IEEE Computer Society Press.

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

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

[8] 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*, page 268, Fort Lauderdale, FL, April 2002. IEEE Computer Society Press.

[9] Hank Childs. Architectural challenges and solutions for petascale postprocessing. *Journal of Physics: Conference Series*, 78(1):012012, 2007.

[10] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th IEEE International Symposium on High Performance Distributed Computing*, pages 25–36, Chicago, IL, June 2010.

[11] Ciprian Docan, Fan Zhang, Manish Parashar, Julian Cummings, Norbert Podhorszki, and Scott Klasky. Experiments with memory-to-memory coupling for end-to-end fusion simulation workflows. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 293–301, May 2010.

[12] National Center for Computational Science. Adios: The adaptible io system.

[13] Howard Pritchardand Igor Gorodetsky. A uGNI-based MPICH2 nemesis network module for Cray XE computer systems. In *Cray User Group Meeting*, Fairbanks, Alaska, May 2011.

[14] Luke Gosink, John Shalf, Kurt Stockinger, Kesheng Wu, and Wes Bethel. HDF5-Fastquery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *In SSDBM*, pages 149–158, 2006.

[15] Chris Johnson, Robert Ross, et al. Visualization and knowledge discovery. Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale, October 2007.

[16] 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, New York, NY, 2001.

[17] S. Ku, C. S. Chang, M. Adams, E. D Azevedo, Y. Chen, P. Diamond, L. Greengard, T. S. Hahm, Z. Lin, S. Parker, H. Weitzner, P. Worley, and D. Zorin. Core and edge full-f ITG turbulence with self-consistent neoclassical and mean flow dynamics using a real geometry particle code XGC1. In *Proceedings of the 22th International Conference on Plasma Physics and Controlled Nuclear Fusion Research*, Geneva, Switzerland, 2008.

[18] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. Compressing the incompressible with isabela: in-situ reduction of spatio-temporal data. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 366–379, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Jay Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *CLADE 2008 at HPDC*, Boston, Massachusetts, June 2008. ACM.

[20] 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*, Seattle, WA, November 2011.

[21] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.

[22] Bruce H. McCormick, Thomas A. DeFanti, and Maxine D. Brown, editors. *Visualization in Scientific Computing (special issue of Computer Graphics)*, volume 21. ACM Press, 1987.

[23] 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*, Seattle, WA, November 2011.

[24] NCAR and UCAR. Community earth system model.

[25] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press.

[26] Ron Oldfield. *Efficient I/O for Computational Grid Applications*. PhD thesis, Dept. of Computer Science, Dartmouth College, May 2003.

Available as Dartmouth Computer Science Technical Report TR2003-459.

[27] Ron Oldfield and David Kotz. Improving data access for computational grid applications. *Cluster Computing, The Journal of Networks, Software Tools and Applications*, 9(1):79–99, January 2006.

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

[29] 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.

[30] 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.

[31] 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.

[32] Ron A. Oldfield, David E. Womble, and Curtis C. Ober. Efficient parallel I/O in seismic imaging. *The International Journal of High Performance Computing Applications*, 12(3):333–344, Fall 1998.

[33] Tom Peterka, Hongfeng Yu, Robert Ross, Kwan-Liu Ma, and R. Latham. End-to-end study of parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of the International Conference on Parallel Processing*, pages 566–573, Vienna, Austria, April 2009.

[34] 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.

[35] 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.

[36] R B Ross, T Peterka, H-W Shen, Y Hong, K-L Ma, H Yu, and K Moreland. Visualization and parallel I/O at extreme scale. *Journal of Physics: Conference Series*, 125(1):012099, 2008.

[37] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings ofSupercomputing '95*, page 57, San Diego, CA, December 1995. IEEE Computer Society Press.

[38] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E. Papka. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.

[39] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam. Gyro-kinetic simulation of global turbulent transport properties in tokamak experiments. *Physics of Plasmas*, 13:092505, September 2006.

[40] 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, Atlanta, GA, April 2010.