Integrating Analysis and Computation with Trios Services

Ron A. Oldfield
Scalable System Software
Sandia National Laboratories
Albuquerque, NM, USA
raoldfi@sandia.gov

Trilinos User Group Meeting

Oct 31, 2012
Some I/O Issues for Exascale

- Storage systems are the slowest, most fragile, part of an HPC system
  - Scaling to extreme client counts is challenging
  - POSIX semantics gets in the way, ...

- Current usage models not appropriate for Petascale, much less Exascale
  - Checkpoints are a **HUGE** concern for I/O...currently primary focus of FS
  - App workflow uses storage as a communication conduit
    - Simulate, *store*, analyze, *store*, refine, *store*, ... most of the data is transient

- One way to reduce I/O pressure on the FS is to inject nodes between app and FS
  1. Reduce the “effective” I/O cost through data staging (a.k.a. *Burst Buffer*)
  2. Reduce amount of data written to storage (integrated analysis, data services)
  3. Present FS with fewer clients (IO forwarding)

  "**Trios Services**" enable application control of these nodes
Trios Data Services

**Approach**
- Leverage available compute/service node resources for I/O caching and data processing

**Application-Level I/O Services**
- PnetCDF staging service
- CTH real-time analysis
- SQL Proxy (for NGC)
- Interactive sparse-matrix visualization
- In-memory key-value (in development)

**Nessie**
- Framework for developing data services
- Portable API for inter-app comm (NNTI)
- Client and server libs, cmake macros, utilities
- Originally developed for LWFS
Example: A Simple Transfer Service

Trilinos/packages/trios/examples/xfer-service

- Used to test Nessie API
  - `xfer_write_rdma`: server pulls raw data using RDMA get
  - `xfer_read_rdma`: server transfers data to client using RDMA put

- Used for performance evaluation
  - Test low-level network protocols
  - Test overhead of XDR encoding
  - Tests async and sync performance

- Creating the Transfer Service
  - Define the XDR data structs and API arguments
  - Implement the client stubs
  - Implement the server

```c
/* 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;
};
...
```
Interface between scientific app and service

Steps for client stub

- Initialize the remote method arguments, in this case, it’s just the length of the array
- Call the rpc function. The RPC function includes method arguments \((args)\), and a pointer to the data available for RDMA \((buf)\)

The RPC is asynchronous

- The client checks for completion by calling \(\text{nssi\_wait}(&\text{req})\);

```c
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=arr\_data\_array\_t\_val;

    /* size of the RDMA buffer */
    nbytes = args\_len*\text{sizeof}(data\_t);

    /* call the remote methods */
    nssi\_call\_rpc(svc, XFER\_PULL,
        &args, (char *)buf, nbytes,
        NULL, req);
}
```
Transfer Service
Implementing the Server

- Implement server stubs
  - Using standard stub args
  - For `xfer_write_rdma_srvr`, the server pulls data from client

- Implement server executable
  - Initialize Nessie
  - Register server stubs/callbacks
  - Start the server thread(s)

```c
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);
}
```
Transfer Service Evaluation:
Put/Get Performance

Xfer-uGNI Performance on Cielo

MPi Scaling Performance on Cielo
Transfer Service Evaluation:
Put/Get Performance

Xfer–uGNI Performance on Cielo

Xfer–write–rdma Performance on Red Storm

Throughput (MB/s) vs. Bytes/Transfer

Throughput (MB/s) vs. Bytes/Transfer

Gemini Interconnect

SeaStar Network

- xfer–read (PUTs)
- xfer–write (GETs)

- 1 client
- 4 clients
- 16 clients
- 64 clients

October 31, 2012
Trilinos User Group Meeting
Motivation

- Analysis code may not scale as well as HPC code
- Direct integration may be fragile (e.g., large binaries)
- "Fat" nodes may be available on Exascale architectures for buffering and analysis

CTH fragment detection service

- Extra nodes provide in-line processing (overlap fragment detection with time step calculation)
- Only output results to storage (reduce I/O)
- Non-intrusive – Looks like in-situ (pvspy API)

Issues to Address

- Number of nodes for service
  - Based on memory requirements
  - Based on computational requirements
- Placement of nodes
- Resilience
Memory Requirements for CTH Service

- Memory Analysis of ParaView Codes
  - Current implementation of analysis codes have problems...

- Constraints given 32 GiB/node
  - One node can manage/process ~16K AMR blocks from CTH.
  - 16:1 ratio of compute nodes to service nodes (based on our input decks)

- Our goal is to use less than 10% additional resources
  - In-situ viz adds ~10% overhead

Memory Used by Visualization Service
256 clients, 8 servers (1 node), 5500 blocks

- AMR (timing only)
- Baseline (Nessie+Block Management)
- FileAMR (write VTK files)
- ImageAMR (write png)
Load Balancing for CTH Service

Ten Cycles of 128-core run (one server node)

- 2 server cores – 64:1
  - 10 cycles in 37 secs
  - Client idle waiting for server to complete (also affects transfers)

- 4 server cores – 32:1
  - 10 cycles in 23 secs

- 8 server cores – 16:1
  - 10 cycles in 19 secs
  - Less than 1% time waiting
Impact of Placement on Performance

- We know placement is important from previous study
- Goal is to place nodes within given allocation to avoid network contention
  - App-to-app (MPI), app-to-svc (NTTI), svc-to-svc (MPI), svc-to-storage (PFS)
  - Graph partitioning based on network topology and application network traffic (w/Pedretti)

![Graph showing 8:1 ratio of application to staging nodes](image)
Resilience for Trios Services

- Storage-efficient app resilience is still a problem after 20+ years of research
- Trios services use memory for transient data, how do we ensure resilience in such a model?

- We are exploring transaction-based methods
  - Goal is to provide assurances in multiple protection domains (e.g., the application, service 1, service 2,...)
  - Jay Lofstead (1423) has an LDRD to look at this issue.
Summary

- Data Services provide a new way to integrate analysis and computation
  - Particularly useful on deployments of “Burst Buffer” architectures
  - Other Labs are also looking into this type of approach (ANL:Gleam, ORNL:ADIOS, ...)
  - Scheduling, programming models, security, and resilience need to better support data services. Lots of work to do!

- Nessie provides an effective framework for developing services
  - Client and server API, macros for XDR processing, utils for managing svcs
  - Supports most HPC interconnects (Seastar, Gemini, InfiniBand, IBM)

- Trilinos provides a great research vehicle
  - Common repository, testing support, broad distribution

- Trios Data Services Development Team (and current assignment)
  - Ron Oldfield: PI, CTH data service, Nessie development
  - Todd Kordenbrock: Nessie development, performance analysis
  - Gerald Lofstead: PnetCDF/Exodus, transaction-based resilience
  - Craig Ulmer: Data-service APIs for accelerators (GPU, FPGA)
  - Shyamali Mukherjee: Protocol performance evaluations, Nessie BG/P support
Integrating Analysis and Computation with Trios Services

Ron A. Oldfield
Scalable System Software
Sandia National Laboratories
Albuquerque, NM, USA
raoldfi@sandia.gov

Trilinos User Group Meeting

Oct 31, 2012
Nessie Network Transport Interface (NNTI)
An abstract API for RDMA-based interconnects

- Portable methods for inter-application communication

- Basic functionality
  - Initialize/close the interface
  - Connect/disconnect to a peer
  - Register/deregister memory
  - Transport data (put, get, wait)

- Supported Interconnects
  - Seastar (Cray XT), InfiniBand, Gemini (Cray XE, XK), DCMF (IBM BG/P)

- Users
  - Nessie, ADIOS/DataStager (new), HDF5?, Ceph?
Motivation
- Synchronous I/O libraries require app to wait until data is on storage device
- Not enough cache on compute nodes to handle “I/O bursts”
- NetCDF is basis of important I/O libs at Sandia (Exodus)

PnetCDF Caching Service
- Service aggregates/caches data and pushes data to storage
- Async I/O allows overlap of I/O and computation

Status
- First described in MSST 2006 paper
- Implemented in 2007
- Presented at PDSW’11