Portals 4 Exascale

Ron Brightwell
March 26, 2012
Outline

• Recap from last year
• Portals overview
• Portals 4.0 implementations
• OpenSHMEM on Portals 4.0
• Portals 4.0 triggered operations
  – MPI collective operations
  – MPI rendezvous protocol
• Linux XPMEM
Recap From Last Year

- Overview of Portals 4.0 API
- Research vehicle for NIC architecture co-design
- Description of IB reference implementation
Portals Network Programming Interface

- Network API developed by Sandia, U. New Mexico, Intel
- Previous generations of Portals deployed on several production massively parallel systems
  - 1993: 1800-node Intel Paragon (SUNMOS)
  - 1997: 10,000-node Intel ASCI Red (Puma/Cougar)
  - 1999: 1800-node Cplant cluster (Linux)
  - 2005: 10,000-node Cray Sandia Red Storm (Catamount)
  - 2009: 18,688-node Cray XT5 – ORNL Jaguar (Linux)
- Focused on providing
  - Lightweight “connectionless” model for massively parallel systems
  - Low latency, high bandwidth
  - Independent progress
  - Overlap of computation and communication
  - Scalable buffering semantics
  - Protocol building blocks to support higher-level protocols
- Supports MPI, SHMEM, ARMCI, GASNet, Lustre, etc.
Portals 4.0 Implementations

- **OpenFabrics Verbs**
  - Provided by System Fabric Works
  - Provides a high-performance reference implementation for experimentation
  - Help identify issues with API, semantics, performance, etc.
  - Independent analysis of the specification

- **Shared memory**
  - Offers consistent and understandable performance characteristics
  - Provides ability to accurately measure instruction count for Portals operations
  - Better characterization of operations that impact latency and message rate
  - Evaluation of single-core onloading performance limits

- **Structural Simulation Toolkit (SST)**
  - Partial implementation for exploring NIC structures for offload
OpenSHMEM on Portals 4.0

OpenSHMEM

- Proposed community standard for SHMEM
- Partitioned Global Address Space library
- Put, get, atomic operations, plus collective communication
- Encourages asynchronous, small message patterns
OpenSHMEM

- Communication calls
  - Elemental put, block put, strided put
  - Elemental get, block get, strided get
  - Atomic operations
  - All operations provide local completion
  - Read and read-write operations imply remote completion

- Operations must target symmetric memory
  - Global data: Global and static variables in C, Common block
  - Symmetric Heap: global dynamic memory

- Ordering / completion functions
  - Fence/quiet
  - Address wait
Portals 4.0

- Communication Calls
  - Non-blocking Put, Get, Atomic
  - Matching or non-matching receive interfaces
- Completion Semantics
  - Completion events for local and remote completion
  - Counters of events / bytes for light-weight messaging
- Memory Model
  - Generally, no atomicity / data ordering guarantees
  - Maximum message size for atomic operations
  - Maximum size for single-byte write-after-write ordering
  - May provide more general write-after-write ordering
  - Maximum size for local completion of put/atomic operations
# Memory Layout

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Text</td>
</tr>
</tbody>
</table>

![Diagram of memory layout with stack, heap, data, and text sections](image-url)
Memory Layout

stack

heap

Sym. heap

data

text
Memory Layout

<table>
<thead>
<tr>
<th>stack</th>
<th>Put MD</th>
<th>Get MD</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sym. heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Heap LE
- Data LE
Portals Data Structures

- Portal Table
  - Heap PT
  - Data PT
  - Heap LE
  - Data LE
  - Err EQ
  - Put MD
  - Put CT
  - Get MD
  - Get CT
  - Target CT
void shmem_long_p(long *addr, long value, int pe) {
    ptl_process_t peer;
    ptl_pt_index_t pt;
    long offset;
    peer.rank = pe;
    GET_REMOTE_ACCESS(target, pt, offset);

    PtlPut(shmem_internal_put_md_h,
    (ptl_size_t) &value,
    sizeof(value),
    PTL_CT_ACK_REQ,
    peer,
    pt,
    0,
    offset,
    NULL,
    0);
}
void shmem_double_get(double *target, const double *source,
    size_t len, int pe) {

    ...

    ptl_ct_event_t ct;
    peer.rank = pe;
    GET_REMOTE_ACCESS(source, pt, offset);

    PtlGet(shmem_internal_get_md_h,
        (ptl_size_t) target,
        len * sizeof(double),
        peer,
        pt,
        0,
        offset,
        0);
    shmem_internal_pending_get_counter++;
    PtlCTWait(shmem_internal_get_ct_h,
        shmem_internal_pending_get_counter,
        &ct);

}
Results

- Running SHMEM codes over both shared memory and InfiniBand
- Results shown for shared memory
- 3.33 GHz Westmere EP w/ 1333 DDR3 Linux system

- ½ Round Trip Latency:
  - Raw Portals: 0.43 μs
  - OpenSHMEM: 0.39 μs

  - Numbers misleading; slightly more work on receive side for Raw Portals
Message Rate

Portals LE/CT
SHMEM

Message size (bytes)
Message Rate (messages/second) vs. Message size (bytes)
NetPIPE Bandwidth

![Graph showing bandwidth vs. message size for Portals LE/CT and SHMEM.]
Triggered Operations for Collective Communication

Motivation

- Collectives are important to a broad array of applications
  - As node counts grow, it becomes hard to keep collective time low
- Offload provides a mechanism to reduce collective time
  - Eliminates portion of Host-to-NIC latency from the critical path
  - Relatively complex collective algorithms are constantly refined and tuned
- Building blocks provide a better
  - Allow algorithm research and implementation to occur on the host
  - Provides a simple set of hardware mechanisms to implement
- A general purpose API is needed to express the building blocks
Triggered Operations

- Lightweight events are counters of various network transactions
  - One counter can be attached to multiple different operations or even types of operations
  - Fine grained control of what you count is provided
- Portals operation is “triggered” when a counter reaches a threshold specified in the operations
  - Various types of operations can be triggered
  - Triggered counter update allows chaining of local operations
Generality of Triggered Operations

- Numerous collectives have been implemented so far
  - Allreduce
  - Bcast
  - Barrier

- Numerous algorithms have been implemented for multiple collectives
  - Binary tree
  - k-nomial tree
  - Pipelined broadcast
  - Dissemination barrier
  - Recursive doubling
Simulation Methodology

• Utilized SST simulator developed at Sandia
• Modeled processor and NIC as separate state machines
  – Fixed delays between states to model delays and overhead
  – Single state machine for processor, multiple for NIC to model concurrent hardware blocks
• Modeled several combinations of parameters defined by latency and message rate
  – Allocated delay to various units that were modeled
High-Level NIC Architecture
Simulation Settings

(a) simulation parameters

<table>
<thead>
<tr>
<th>Property</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msg Latency</td>
<td>500 ns, 1000 ns, 1500 ns</td>
</tr>
<tr>
<td>Msg Rate</td>
<td>5 Mmsgs/s, 10 Mmsgs/s</td>
</tr>
<tr>
<td>Overhead</td>
<td>$\frac{1}{\text{MsgRate}}$</td>
</tr>
<tr>
<td>NIC Msg Rate</td>
<td>62.5 Mmsgs/s</td>
</tr>
<tr>
<td>Rtr Latency</td>
<td>50 ns</td>
</tr>
<tr>
<td>Setup Time</td>
<td>200 ns</td>
</tr>
<tr>
<td>Cache Line</td>
<td>64 Bytes</td>
</tr>
<tr>
<td>Miss Latency</td>
<td>100 ns</td>
</tr>
<tr>
<td>Noise</td>
<td>250 ns @ 100KHz, 25 $\mu$s @ 1KHz, 2.5 ms @ 10Hz</td>
</tr>
</tbody>
</table>

(b) simulation configurations

<table>
<thead>
<tr>
<th></th>
<th>500 ns</th>
<th>1000 ns</th>
<th>1500 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Mmsgs/s</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>10 Mmsgs/s</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Allreduce
500ns, 10 Mmsgs/s
Noise Simulations

• Three noise profiles were simulated (2.5% noise for each)
  – 250 ns @ 100KHz
  – 25 μs @ 1KHz
  – 2.5 ms @ 10Hz

• Noise events were randomly distributed
  – Stopped all host processing during a noise event
  – NIC processing continued

• Timed individual collective operations (first entry to last exit)
Allreduce With Noise

25 us @ 1 KHz

![Graph showing Allreduce performance with noise across different node counts and tree structures.]
Noise Simulation Results

- Recursive doubling has poor noise tolerance
- Offload gives significant improvement in noise tolerance
  - Partly from reduced time
  - Partly from reduced host participation
  - Synchronizing operation still cannot complete until everyone contributes a value
- Interesting shape of curves in middle noise case
  - Host based latency continues to grow with node count
  - NIC based latency plateaus
Interesting Things We Learned

- Time to initiate a transaction from the host to the NIC makes things difficult
  - Even with a high NIC rate, can be rate limited by the host
  - Limitation of using host to initiate all operations instead of offloading algorithm
  - If transactions are posted in correct order, limitation is effectively mitigated

- Proper message scheduling is important
  - Time between message initiations on the host (gap) matches network hop latency: send the far away ones first!

- $k$-nomial trees are better, but the work at the root limits the maximum value of $k$

- You can have speed or reproducibility, but…
Triggered Collectives Summary

- Triggered operations provide a general set of building blocks
  - Supports a variety of collective operations
  - Supports a variety of algorithms
  - Has usage beyond just collectives offload
- Collective offload has limited performance upside versus idealized host implementation
  - 2x performance improvement due to improved latency and improved message rate
  - Performance could be improved somewhat by having host “push” data
- Noise sensitivity substantially reduced when operations are offloaded
Triggered Operations for a Rendezvous Protocol

Ping-Pong Bandwidth
Ping-Pong Bandwidth

![Graph showing bandwidth vs message size for different protocols.](Image)
Linux XPMEM for Progress

• Based on SGI IRIX sproc lightweight process
  – sprocs were able to attach segments of other sprocs to their address space
  – Segments of other sprocs mapped at an offset in virtual address space
  – Used to implement SHMEM and MPI on SGI systems

• SGI Altix
  – Ran separate Linux images or “partitions”
  – “Cross-partition” memory module allowed sharing address space between processes in separate partitions (with hardware help)
  – Also works for processes in the same OS partition
  – XPMEM user-level API
    • xpmem_make()
      – Returns a unique handle representing a segment of the address space
    • xpmem_get()
      – Returns handle that can be used to map the segment of another process
    • Xpmem_attach()
      – Returns the starting virtual address of a mapping for a given handle

• Exploring using XPMEM for progress rather than threads
• http://code.google.com/p/xpmem
Acknowledgments

• Sandia
  – Brian Barrett
  – Scott Hemmert
  – Kevin Pedretti
  – Mike Levenhagen

• Intel
  – Keith Underwood
  – Jerrie Coffman
  – Roy Larsen

• System Fabric Works
Portals4 over IB+SM

Work by Frank Zago & Bob Pearson
Supported by Sandia and Intel
Progress Report

• 2010 base
  – Focus was on supporting the spec not performance
  – IB only

• 2011 changes
  – Merged IB and SM (KNEM) implementations
  – Implemented correct overflow behavior
  – Implemented late MR mapping
  – Performance tuning, focused on latency
  – All shmem and MPI tests passing, tests up to 32 nodes (Stan Smith @intel)
DATA IN PHASE (bcopy, RDMA or KNEM)

DATA OUT PHASE (bcopy, RDMA or KNEM)

send1

IB ack

send2

ack/rep

req

IB ack

ack or rep

put/get

Progress thread

P4

RDMA

Wire

RDMA

P4
# Performance

<table>
<thead>
<tr>
<th>Test</th>
<th>SM Transport</th>
<th>IB Transport</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE/CT short message latency</td>
<td>610 nsec</td>
<td>3.28 usec</td>
</tr>
<tr>
<td>ME/EQ short message latency</td>
<td>690 nsec</td>
<td>3.18 usec</td>
</tr>
<tr>
<td>Short message rate</td>
<td>3.26 M msg/sec</td>
<td>1.07 M msg/sec</td>
</tr>
</tbody>
</table>

Measurements made on 2GHz Magny-cours nodes with CX2 QDR HCA and 1 QDR switch i.e. old and slow
2012 Work Plan

• Maintenance
  – Track spec changes (mostly minor)
  – Continue testing and shoot bugs

• Code refactoring
  – Reduce lines of code and cleanup three implementations: ib, sm, mc

• Performance tuning
  – Reduce CPU utilization is the focus
  – Implement shared progress engine (XPMEM based)
Where to find the code

  - Then source->browse->svn->trunk
  - Follow the directions in README, or

- `% svn checkout` [http://portals4.googlecode.com/src/trunk/portals4-read-only](http://portals4.googlecode.com/src/trunk/portals4-read-only)