Towards scalable RDMA locking on a NIC

TORSTEN HOEFLER

with support of Patrick Schmid, Maciej Besta, Salvatore di Girolamo @ SPCL
presented at HP Labs, Palo Alto, CA, USA
NEED FOR EFFICIENT LARGE-SCALE SYNCHRONIZATION
Locks

Inuitive semantics

An example structure

Various performance penalties
LOCKS: CHALLENGES
LOCKS: CHALLENGES

We need intra- and inter-node topology-awareness

We need to cover arbitrary topologies
We need to distinguish between readers and writers

We need flexible performance for both types of processes

What will we use in the design?
WHAT WE WILL USE
MCS Locks

Proc

Cannot enter

Next proc

Proc

Cannot enter

Next proc

Proc

Can enter

Next proc

Proc

Can enter

Next proc

...
WHAT WE WILL USE
Reader-Writer Locks
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
**REMOTE MEMORY ACCESS (RMA) PROGRAMMING**

Diagram showing the interaction between two processes, `p` and `q`, with memory access operations `A`, `B`, `put`, `get`, and `flush`. The diagram includes a Cray BlueWaters supercomputer image.
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks support RDMA
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.

Readers and writers synchronize with a distributed counter (DC).

MCS queues form a distributed tree (DT).

Modular design.
How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

DC: a parameter for the latency of readers vs writers

DT: a parameter for the throughput of readers vs writers

A tradeoff parameter for every structure
DISTRIBUTED MCS QUEUES (DQs)
Throughput vs Fairness

- Larger $T_{L,i}$: more throughput at level $i$.
- Smaller $T_{L,i}$: more fairness at level $i$.

Each DQ: The maximum number of lock passings within a DQ at level $i$, before it is passed to another DQ at $i$. $T_{L,i}$
DISTRIBUTED TREE OF QUEUES (DT)
Throughput of readers vs writers

DT: The maximum number of consecutive lock passings within readers ($T_R$).
**DISTRIBUTED COUNTER (DC)**

Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS

$$T_{DC} = 1$$

$$T_{DC} = 2$$
THE SPACE OF DESIGNS

Higher throughput of writers vs readers

Design A

Design B

Locality vs fairness (for writers)

Lower latency of writers vs readers

Higher throughput of writers vs readers
LOCK ACQUIRE BY READERS

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS
**Lock Acquire by Writers**

Acquire the main lock

Acquire MCS

Acquire the main MCS lock
EVALUATION

- CSCS Piz Daint (Cray XC30)
- 5272 compute nodes
- 8 cores per node
- 169TB memory
EVALUATION
CONSIDERED BENCHMARKS

The latency benchmark

Throughput benchmarks:
- Empty-critical-section
- Single-operation
- Wait-after-release
- Workload-critical-section

DHT
Distributed hashtable evaluation
EVALUATION
DISTRIBUTED COUNTER ANALYSIS

Throughput, 2% writers
Single-operation benchmark

![Graph showing throughput vs. MPI processes (P)]

- Throughput [mlln locks/s]
- MPI processes (P)

<table>
<thead>
<tr>
<th>T_{DC}</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
EVALUATION

READER THRESHOLD ANALYSIS

Throughput, 0.2% writers,
Empty-critical-section benchmark

Throughput [mln locks/s]

\[ T_R \]

- 6000
- 5000
- 4000
- 3000
- 2000
- 1000

MPI processes (P)
EVALUATION
COMPARISON TO THE STATE-OF-THE-ART

EVALUATION

COMPARISON TO THE STATE-OF-THE-ART

Throughput, single-operation benchmark

EVALUATION
DISTRIBUTED HASHTABLE

20% writers

10% writers

EVALUATION
DISTRIBUTED HASHTABLE

2% of writers

0% of writers

OTHER ANALYSES
But why stop at RDMA -- A brief history

- **1980’s**
  - Lossy Networks
  - Ethernet

- **2000’s**
  - Lossless Networks
  - RDMA

- **2020’s**
  - Device Programming
  - Offload

**Questions:**
- How to program QsNet?
- How to offload in Portals 4?
- How to offload in libfabric?
Computations

L0: recv a from P1;
L1: b = compute f(buff, a);
L2: send b to P1;
L0 and CPU→ L1
L1 → L2
Performance Model

Fully Offloaded Collectives

Collective communication: A communication that involves a group of processes
Non-blocking collective: Once initiated the operation may progress independently of any computation or other communication at participating processes
Fully Offloaded Collectives

Collective communication: A communication that involves a group of processes
Non-blocking collective: Once initiated the operation may progress independently of any computation or other communication at participating processes

Fully Offloading:
1. *No synchronization* is required in order to start the collective operation
2. Once a collective operation is started, *no further CPU intervention* is required in order to progress or complete it.
A Case Study: Portals 4

- Based on the one-sided communication model
- Matching/Non-Matching semantics can be adopted

[2] "The Portal 4.0.2 Network Programming Interface"
A Case Study: Portals 4

Communication primitives
- Put/Get operations are natively supported by Portals 4
- One-sided + matching semantic

Atomic operations
- Operands are the data specified by the MD at the initiator and by the ME at the target
- Available operators: min, max, sum, prod, swap, and, or, ...

Counters
- Associated with MDs or MEs
- Count specific events (e.g., operation completion)

Triggered operations
- Put/Get/Atomic associated with a counter
- Executed when the associated counter reaches the specified threshold
FFlib: An Example

Proof of concept library implemented on top of Portals 4

```c
ff_schedule_h sched = ff_schedule_create(...);

ff_op_h r1 = ff_op_create_recv(tmp + blocksize, blocksize, child1, tag);
ff_op_h r2 = ff_op_create_recv(tmp + 2*blocksize, blocksize, child2, tag);

ff_op_h c1 = ff_op_create_computation(rbuff, blocksize, tmp + blocksize, blocksize, operator, datatype, tag)
ff_op_h c2 = ff_op_create_computation(rbuff, blocksize, tmp + 2*blocksize, blocksize, operator, datatype, tag)

ff_op_h s = ff_op_create_send(rbuff, blocksize, parent, tag)

ff_op_hb(r1, c1)
ff_op_hb(r2, c2)
ff_op_hb(c1, s)
ff_op_hb(c2, s)

ff_schedule_add(sched, r1)
ff_schedule_add(sched, r2)
ff_schedule_add(sched, c1)
ff_schedule_add(sched, c2)
ff_schedule_add(sched, s)
```
Experimental Results

**Target machine: Curie**
- 5,040 nodes
- 2 eight-core Intel Sandy Bridge processors
- Full fat-tree Infiniband QDR

**OMPI/P4: Open MPI 1.8.4 + Portals 4 RL**

**FFLIB: proof of concept library**

Experimental Results: Latency/Overhead

**Broadcast**

- **OMPI/P4**: Open MPI 1.8.4 + Portals 4 RL
- **FFLIB**: proof of concept library

**Allreduce**

- **OMPI/P4**: Open MPI 1.8.4 + Portals 4 RL
- **FFLIB**: proof of concept library

Target machine: Curie
- 5,040 nodes
- 2 eight-core Intel Sandy Bridge processors
- Full fat-tree Infiniband QDR

Experimental Results: Latency/Overhead

Target machine: Curie
5,040 nodes
2 eight-core Intel Sandy Bridge processors
Full fat-tree Infiniband QDR

OMPI/P4: Open MPI 1.8.4 + Portals 4 RL
FFLIB: proof of concept library

More about FFLIB at: http://spcl.inf.ethz.ch/Research/Parallel_Programming/FFlib/
Experimental Results: Micro-Benchmarks

Target machine: Curie
- 5,040 nodes
- 2 eight-core Intel Sandy Bridge processors
- Full fat-tree Infiniband QDR

OMPI/P4: Open MPI 1.8.4 + Portals 4 RL
FFLIB: proof of concept library

More about FFLIB at: http://spcl.inf.ethz.ch/Research/Parallel_Programming/FFlib/
Simulations

- **Why?** To study offloaded collectives at large scale
- **How?** Extending the LogGOPSim to simulate Portals 4 functionalities

### Broadcast

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>P4-SW</th>
<th>P4-HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>128</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>2048</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>8192</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

### Allreduce

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>P4-SW</th>
<th>P4-HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>128</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>2048</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>8192</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

### Table

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>o</th>
<th>g</th>
<th>G</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4-SW</td>
<td>5μs</td>
<td>6μs</td>
<td>6μs</td>
<td>0.4ns</td>
<td>0.9ns</td>
</tr>
<tr>
<td>P4-HW</td>
<td>2.7μs</td>
<td>1.2μs</td>
<td>0.5μs</td>
<td>0.4ns</td>
<td>0.3ns [4]</td>
</tr>
</tbody>
</table>

---


Offloading Collectives

Solo Collectives

Mapping to Portals 4

Abstract Machine Model

Results

FFlib
CONCLUSIONS

Modular distributed RMA lock, correctness with SPIN
Parameter-based design, feasible with various RMA libs/languages
Improves latency and throughput over state-of-the-art
Enables high-performance distributed hashtabled

Thank you for your attention