TORSTEN HOEFLER

Remote Memory Access Programming: Faster Parallel Computing Without Messages

with S. Ramos, R. Gerstenberger, M. Besta, R. Belli @ SPCL

presented at San Diego Supercomputing Center/UCSD, San Diego, CA, June 2015
Motivation & Goals

- My dream: provably optimal performance (time and energy)
  - From problem to machine code
  - How to get there?

- Model-based Performance Engineering!
  1. Design a system model
  2. Define your problem
  3. Find (close-to) optimal solution in model → prove
  4. Implement, test, refine if necessary

- Will demonstrate techniques & insights
  - And obstacles 😊
  - RMA as a solution?
A Practical Model of Parallel Computation

Example: Message Passing, Log(G)P

Our goal is to develop a model of parallel computation that will serve as a basis for the design and analysis of fast, portable parallel algorithms, such as algorithms that can be implemented effectively on a wide variety of current and future parallel machines. If we look at the body of parallel algorithms developed under current parallel models, many are impractical because they exploit artificial factors not present in any real parallel computer.

LogP

Process P-1

P0: MPI_Recv(1); MPI_Recv(1)
P1: MPI_Send(0); MPI_Send(0)

Process 1

CPU

NET

Process 0

CPU

NET

Broadcast Problem

Optimal Solution

PRAM consists of a collection of processors which compute synchronously in parallel and communicate with a global random access memory.

Hardware Reality

Interlagos, 8/16 cores, source: AMD

POWER 7, 8 cores, source: IBM

Xeon Phi, 64 cores, source: Intel
Hardware Reality

Interlagos, 8/16 cores, source: AMD

POWER 7, 8 cores, source: IBM

Xeon Phi, 64 cores, source: Intel

Kepler GPU, source: NVIDIA

InfiniBand, sources: Intel, Mellanox

BG/Q, Cray Aries, sources: IBM, Cray
Hardware Reality

- **Interlagos**, 8/16 cores, source: AMD
- **POWER 7**, 8 cores, source: IBM
- **Xeon Phi**, 64 cores, source: Intel
- **Kepler GPU**, source: NVIDIA
- **InfiniBand**, sources: Intel, Mellanox
- **BG/Q**, Cray Aries, sources: IBM, Cray
Example: Cache-Coherent Communication

Xeon Phi (Rough) Architecture
Invalid read $R_I = 278$ ns
Local read: $R_L = 8.6$ ns
Remote read $R_R = 235$ ns

*Inspired by Molka et al.: “Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system”*
**DTD Contention 😞**

- **E state:**
  - $a = 0$ ns
  - $b = 320$ ns
  - $c = 56.2$ ns

$$T_C(n_{th}) = c \cdot n_{th} + b - \frac{a}{n_{th}}$$
Designing Broadcast Algorithms

- Assume single cache line \(\rightarrow\) forms a Tree
  - We choose \(d\) levels and \(k_j\) children in level \(j\)
  - Reachable threads: \(n_{th} \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j\)
  - Example: \(d=2, k_1=3, k_2=2:\)

\[
T_{tree} = \sum_{i=1}^{d} T_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b) \\
= \sum_{i=1}^{d} (R_R + R_L + c \cdot (k_i - 1))
\]

\(c = DTD\) contention
\(b =\) transmit latency

\[
t_0 = 0 \\
t_1 = k_1^*c + b \\
t_2 = t_1 + k_2^*c + b
\]
Finding the Optimal Broadcast Algorithm

- Broadcast example:

\[ T_{tree} = \sum_{i=1}^{d} T_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b) \]

\[ = \sum_{i=1}^{d} (R_R + R_L + c \cdot (k_i - 1)) \]

\[ T_{sbcast} = \min_{d,k_i} \left( T_{fw} + \sum_{i=1}^{d} (c \cdot k_i + b) + \sum_{i=1}^{d} T_{nb}(k_i + 1) \right) \]

\[ N \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j, \quad \forall i < j, k_i \leq k_j \]

Min-Max Modeling

- Example:
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion
Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

Min-Max Modeling

Example:
- $T_0 + T_1$ write CL
- $T_2$ polls for completion

Min-Max Modeling

- Example:
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

Min-Max Modeling

- Example:
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

*Ramos, Hoefler: “Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi”, HPDC’13*
Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

Min-Max Modeling

- Example:
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

Min-Max Modeling

- Example:
  - \( T_0 + T_1 \) write CL
  - \( T_2 \) polls for completion
Min-Max Modeling

- Example:
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

Min-Max Modeling

- Example:
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

**Diagram Description**

- $T_0$ writes to memory.
- $T_1$ reads from memory.
- $T_2$ reads from memory.
- Diagram shows the sequence of reads and writes, indicating communication between tasks.

---

*Ramos, Hoefler: “Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi“, HPDC’13*
Min-Max Modeling

- **Example:**
  - \( T_0 + T_1 \) write CL
  - \( T_2 \) polls for completion

---

*Ramos, Hoefler: “Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi”, HPDC'13*
Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

---

Min-Max Modeling

- **Example:**
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

Min-Max Modeling

- Example:
  - $T_0 + T_1$ write CL
  - $T_2$ polls for completion

Small Broadcast (8 Bytes)

- Broadcast
- Min–Max Model
- Intel MPI

Barrier

Small Reduction

Lessons learned

- Rigorous modeling has large potential
  - Coming with great cost (working on tool support [1])

- Understanding cache-coherent communication performance is incredibly complex (but fun)!
  - Many states, min-max modeling, NUMA, …
  - Have models for Sandy Bridge now (QPI, worse!)

- Cache coherence really gets in our way here 😞

- Obvious question: why do we need cache coherence?
  - Answer: well, we don’t, if we program right!

[1]: Calotoiu et al.: Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes, SC13
[2]: Gerstenberger et al.: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC13, Best Paper
COMMUNICATION IN TODAY’S HPC SYSTEMS

- The de-facto programming model: MPI-1
  - Using send/recv messages and collectives

- The de-facto network standard: RDMA, SHM
  - Zero-copy, user-level, os-bypass, fuzz-bang

Random datacenter picture copyrighted by Reuters (yes, they go after academics with claims for 10 year old images)
MPI-1 MESSAGE PASSING – SIMPLE EAGER

Producer

Consumer

MPI-1 MESSAGE PASSING – SIMPLE EAGER

Producer

Send

1. Data transfer to intermediate buffer

Consumer

Mailbox

MPI-1 MESSAGE PASSING – SIMPLE EAGER

Producer

Send

Consumer

1. Data transfer to intermediate buffer

2. Acknowledgement

: origin aware of completion

Critical path: 1 latency + 1 copy

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

Critical path: 3 latencies

August 18, 2006

A Critique of RDMA

by Patrick Geoffray, Ph.D.

Do you remember VIA, the Virtual Interface Architecture? I do. In 1998, according to its promoters — Intel, Compaq, and Microsoft — VIA was supposed to change the face of high-performance networking. VIA was a buzzword at the time; Venture Capital was flowing, and startups multiplying. Many HPC pundits were rallying behind this low-level programming interface, which promised scalable, low-overhead, high-throughput communication, initially for HPC and eventually for the data center. The hype was on and doom was spelled for the non-believers.

It turned out that VIA, based on RDMA (Remote Direct Memory Access, or Remote DMA), was not an improvement on existing APIs to support widely used application-software interfaces such as MPI and Sockets. After a while, VIA faded away, overtaken by other developments.

VIA was eventually re-born into the RDMA programming model that is the basis of various InfiniBand Verbs implementations, as well as DAPL (Direct Access Provider Library) and iWARP (Internet Wide Area RDMA Protocol). The pundits have returned, VCs are spending their money, and RDMA is touted as an ideal solution for the efficiency of high-performance networks.

However, the evidence I'll present here shows that the revamped RDMA model is more a problem than a solution. What's more, the objective that RDMA pretends to address of efficient user-level communication between computing nodes is already solved by the two-sided Send/Recv model in products such as Quadrics QsNet, Cray SeaStar (implementing Sandia Portals), Qlogic InfiniPath, and Myricom's Myrinet Express (MX).

Send/Recv versus RDMA

The difference between these two paradigms, Send/Receive (Send/Recv) and RDMA, resides essentially in the
REMOTE MEMORY ACCESS PROGRAMMING

- Why not use these RDMA features more directly?
  - A global address space may simplify programming
  - … and accelerate communication
  - … and there could be a widely accepted standard

- MPI-3 RMA (“MPI One Sided”) was born
  - Just one among many others (UPC, CAF, …)
  - Designed to react to hardware trends, learn from others
  - Direct (hardware-supported) remote access
  - New way of thinking for programmers

MPI-3 RMA Summary

- MPI-3 updates RMA ("MPI One Sided")
  - Significant change from MPI-2
- Communication is "one sided" (no involvement of destination)
  - Utilize direct memory access
- RMA decouples communication & synchronization
  - Fundamentally different from message passing

Two sided communication involves synchronous operations at both the sender and receiver, while one-sided communication allows the sender to perform operations independently without synchronization at the destination.

MPI-3 RMA Communication Overview

- Process A (passive)
  - Memory
  - MPI window

- Process B (active)
  - Memory
  - MPI window

- Process C (active)
  - Atomic communication calls (put, get)

- Process D (active)
  - Atomic communication calls (Acc, Get & Acc, CAS, FAO)

Non-atomic communication calls (put, get)

Gropp, Hoefler, Thakur, Lusk: Using Advanced MPI
MPI-3 RMA COMMUNICATION OVERVIEW

Process A (passive)

Memory

MPI window

Put

Non-atomic communication calls (put, get)

Atomic

Atomic communication calls (Acc, Get & Acc, CAS, FAO)

Process B (active)

Memory

MPI window

Get

Process C (active)

Process D (active)
MPI-3 RMA COMMUNICATION OVERVIEW

Process A (passive)

Memory

Put

Non-atomic communication calls (put, get)

Atomic communication calls (Acc, Get & Acc, CAS, FAO)

Process B (active)

Memory

MPI window

Process C (active)

Atomic

Process D (active)

Gropp, Hoefler, Thakur, Lusk: Using Advanced MPI
MPI-3 RMA COMMUNICATION OVERVIEW

- Process A (passive)
  - Memory

- Process B (active)
  - Memory
  - MPI window
  - Put

- Process C (active)
  - Get

- Process D (active)
  - Atomic communication calls (Acc, Get & Acc, CAS, FAO)

Atomic communication calls (Acc, Get & Acc, CAS, FAO)

Non-atomic communication calls (put, get)
MPI-3 RMA Communication Overview

- Process A (passive)
  - Memory
  - MPI window

- Process B (active)
  - Memory
  - MPI window

- Process C (active)
  - Atomic communication calls (put, get)

- Process D (active)
  - Atomic communication calls (Acc, Get & Acc, CAS, FAO)

Non-atomic communication calls (put, get)
MPI-3 RMA SYNCHRONIZATION OVERVIEW

Active Target Mode
- Fence
- Post/Start/Complete/Wait

Passive Target Mode
- Lock
- Lock All

Active process
Passive process

Synchronization
Communication
MPI-3 RMA Synchronization Overview

Active Target Mode
- Fence
- Post/Start/Complete/Wait
- Active process
- Passive process

Passive Target Mode
- Lock
- Lock All
- Synchronization
- Communication

Gropp, Hoefler, Thakur, Lusk: Using Advanced MPI
MPI-3 RMA SYNCHRONIZATION OVERVIEW

Active Target Mode

- Active process
- Passive process
- Fence
- Post/Start/Complete/Wait

Passive Target Mode

- Synchronization
- Communication
- Lock
- Lock All
- Lock All

Gropp, Hoelfer, Thakur, Lusk: Using Advanced MPI
MPI-3 RMA SYNCHRONIZATION OVERVIEW

Active Target Mode
- Fence
- Post/Start/Complete/Wait
- Active process
- Passive process
- Synchronization
- Communication

Passive Target Mode
- Lock
- Lock All
- Active process
- Passive process
- Synchronization
- Communication

Gropp, Hoelfer, Thakur, Lusk: Using Advanced MPI
MPI-3 RMA SYNCHRONIZATION OVERVIEW

Active Target Mode

- Fence
- Post/Start/Complete/Wait

Passive Target Mode

- Lock
- Lock All

Active process
Passive process
Synchronization
Communication

Gropp, Hoelfer, Thakur, Lusk: Using Advanced MPI
SCALABLE PROTOCOLS & REFERENCE IMPLEMENTATION

- Scalable & generic protocols
  - Can be used on any RDMA network (e.g., OFED/IB)
  - Window creation, communication and synchronization

Window creation

Communication

Synchronization
**SCALABLE PROTOCOLS & REFERENCE IMPLEMENTATION**

- Scalable & generic protocols
  - Can be used on any RDMA network (e.g., OFED/IB)
  - Window creation, communication and synchronization

- foMPI, a fully functional MPI-3 RMA implementation
  - DMAPP: lowest-level networking API for Cray Gemini/Aries systems
  - XPMEM, a portable Linux kernel module
SCALABLE PROTOCOLS & REFERENCE IMPLEMENTATION

- Scalable & generic protocols
  - Can be used on any RDMA network (e.g., OFED/IB)
  - Window creation, communication and synchronization

- foMPI, a fully functional MPI-3 RMA implementation
  - DMAPP: lowest-level networking API for Cray Gemini/Aries systems
  - XPMEM, a portable Linux kernel module

http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI
**Scalable Protocols & Reference Implementation**

- Scalable & generic protocols
  - Can be used on any RDMA network (e.g., OFED/IB)
  - Window creation, communication and synchronization

- foMPI, a fully functional MPI-3 RMA implementation
  - DMAPP: lowest-level networking API for Cray Gemini/Aries systems
  - XPMEM: a portable Linux kernel module

http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI
**Performance Inter-node: Latency**

**Put Inter-Node**

- **80% faster**
- Size [Bytes]
- Latency [us]
- Transport Layer:
  - FOMPI MPI-3.0
  - Cray UPC
  - Cray MPI-2.2
  - Cray MPI-1
  - Cray CAF
- DMAPP protocol change

**Get Inter-Node**

- **20% faster**
- Size [Bytes]
- Latency [us]
- Transport Layer:
  - FOMPI MPI-3.0
  - Cray UPC
  - Cray MPI-2.2
  - Cray MPI-1
  - Cray CAF
- DMAPP protocol change

Half ping-pong

Proc 0 \[\text{put}\] \[\text{sync memory}\] \[\text{Proc 1}\]

Gerstenberger, Besta, Hoefler: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC13
PERFORMANCE INTRA-NODE: LATENCY

Put/Get Intra-Node

3x faster

Transport Layer
- FOMPI MPI-3.0
- Cray UPC
- Cray MPI-2.2
- Cray MPI-1
- Cray CAF

Latency [μs]

Size [Bytes]

Gerstenberger, Besta, Hoefler: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC13
PERFORMANCE: MESSAGE RATE

Inter-Node

Intra-Node

Transport Layer
- FOMPI MPI-3.0
- Cray UPC
- Cray MPI-2.2
- Cray MPI-1
- Cray CAF

Message Size [Bytes]

Message Rate [Million Mes./Sec.]

Gerstenberger, Besta, Hoefler: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC13
PART 3: SYNCHRONIZATION

**Active Target Mode**
- Fence
- Post/Start/Complete/Wait

**Passive Target Mode**
- Lock
- Lock All

**Communication**
- Active process
- Passive process
- Synchronization
- Communication
SCALABLE FENCE PERFORMANCE

<table>
<thead>
<tr>
<th>Time bound</th>
<th>$\mathcal{O}(\log p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bound</td>
<td>$\mathcal{O}(1)$</td>
</tr>
</tbody>
</table>

90% faster
**Flush Synchronization**

- Guarantees remote completion
- Performs a remote bulk synchronization and an x86 mfence
- One of the most performance critical functions, we add only 78 x86 CPU instructions to the critical path

<table>
<thead>
<tr>
<th>Time bound</th>
<th>$O(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bound</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Gerstenberger, Besta, Hoefler: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC13
Flush Synchronization

- Guarantees remote completion
- Performs a remote bulk synchronization and an x86 mfence
- One of the most performance critical functions, we add only 78 x86 CPU instructions to the critical path
### Performance Modeling

Performance functions for synchronization protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Performance Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fence</td>
<td>$P_{fence} = 2.9\mu s \cdot \log_2(p)$</td>
</tr>
</tbody>
</table>
| PSCW       | $P_{start} = 0.7\mu s, P_{wait} = 1.8\mu s$
|            | $P_{post} = P_{complete} = 350\text{ns} \cdot k$ |
| Locks      | $P_{lock,excl} = 5.4\mu s$
|            | $P_{lock,shrd} = P_{lock_all} = 2.7\mu s$
|            | $P_{unlock} = P_{unlock_all} = 0.4\mu s$
|            | $P_{flush} = 76\text{ns}$
|            | $P_{sync} = 17\text{ns}$ |

Performance functions for communication protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Performance Functions</th>
</tr>
</thead>
</table>
| Put/get    | $P_{put} = 0.16\text{ns} \cdot s + 1\mu s$
|            | $P_{get} = 0.17\text{ns} \cdot s + 1.9\mu s$ |
| Atomics    | $P_{acc,sum} = 28\text{ns} \cdot s + 2.4\mu s$
|            | $P_{acc,min} = 0.8\text{ns} \cdot s + 7.3\mu s$ |
APPLICATION PERFORMANCE

- Evaluation on Blue Waters System
  - 22,640 computing Cray XE6 nodes
  - 724,480 schedulable cores
- All microbenchmarks
- 4 applications
- One nearly full-scale run 😊
PERFORMANCE: MOTIF APPLICATIONS

Key/Value Store: Random Inserts per Second

- Transport Layer
  - FOMPI MPI-3.0
  - Cray UPC
  - Cray MPI-1

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Billion Inserts per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.025</td>
</tr>
<tr>
<td>8</td>
<td>0.100</td>
</tr>
<tr>
<td>32</td>
<td>1.000</td>
</tr>
<tr>
<td>128</td>
<td>10.000</td>
</tr>
<tr>
<td>512</td>
<td>100.000</td>
</tr>
<tr>
<td>2048</td>
<td>1000.000</td>
</tr>
<tr>
<td>8192</td>
<td>10000.000</td>
</tr>
<tr>
<td>32768</td>
<td>100000.000</td>
</tr>
</tbody>
</table>

Dynamic Sparse Data Exchange (DSDE) with 6 neighbors

- Transport Layer
  - FOMPI MPI-3.0
  - LibNBC
  - Cray MPI-2.2
  - Cray Reduce_scatter
  - Cray Alltoall

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Time [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
</tr>
<tr>
<td>512</td>
<td>10000</td>
</tr>
<tr>
<td>2048</td>
<td>100000</td>
</tr>
<tr>
<td>8192</td>
<td>1000000</td>
</tr>
<tr>
<td>32768</td>
<td>10000000</td>
</tr>
</tbody>
</table>
PERFORMANCE: APPLICATIONS


**NAS 3D FFT [1] Performance**

**MILC [2] Application Execution Time**

[1] Nishtala et al.: Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. IPDPS’09
[2] Shan et al.: Accelerating applications at scale using one-sided communication. PGAS’12
In case you want to learn more

- Available in most MPI libraries today
- Some are even fast!

Using Advanced MPI
Modern Features of the Message-Passing Interface

How to implement producer/consumer in passive mode?

William Gropp
Torsten Hoefler
Rajeev Thakur
Ewing Lusk
**PRODUCER-CONSUMER RELATIONS**

- Most important communication idiom
  - Some examples:

- Perfectly supported by MPI-1 Message Passing
  - But how does this actually work over RDMA?

ONE SIDED – PUT + SYNCHRONIZATION

Producer

Consumer
ONE SIDED – PUT + SYNCHRONIZATION

Producer

Put

1. Data transfer

Consumer

ONE SIDED – PUT + SYNCHRONIZATION

Producer

Put

Flush

Consumer

1. Data transfer

2. Producer waits for remote completion

: origin aware of completion

**ONE SIDED – PUT + SYNCHRONIZATION**

**Producer**
- Put
- Flush
- Explicit Synch

**Consumer**
- Explicit Synch

1. Data transfer
2. Producer waits for remote completion
3. Producer reports completion to consumer

☆: target aware of completion
☆: origin aware of completion

**Critical path: 3 latencies**

COMPARING APPROACHES

Message Passing:
1 latency + copy / 3 latencies

One Sided:
3 latencies

IDEA: RMA Notifications

- First seen in Split-C (1992)
- Combine communication and synchronization using RDMA
- RDMA networks can provide various notifications
  - Flags
  - Counters
  - Event Queues
COMPARING APPROACHES

Message Passing
1 latency + copy / 3 latencies

One Sided
3 latencies

Notified Access
1 latency

Comparing Approaches

Message Passing
1 latency + copy / 3 latencies

One Sided
3 latencies

Notified Access
1 latency

But how to notify?
PREVIOUS WORK: OVERWRITING INTERFACE

- Flags (polling at the remote side)
  - Used in GASPI, DMAPP, NEON

Disadvantages
- Location of the flag chosen at the sender side
- Consumer needs at least one flag for every process
- Polling a high number of flags is inefficient
**Previous Work: Counting Interface**

- **Atomic counters (accumulate notifications → scalable)**
  - Used in Split-C, LAPI, SHMEM - Counting Puts, …

**Disadvantages**
- Dataflow applications may require many counters
- High polling overhead to identify accesses
- Does not preserve order (may not be linearizable)
WHAT IS A GOOD NOTIFICATION INTERFACE?

- **Scalable to yotta-scale**
  - Does memory or polling overhead grow with # of processes?

- **Computation/communication overlap**
  - Do we support maximum asynchrony? (better than MPI-1)

- **Complex data flow graphs**
  - Can we distinguish between different accesses locally?
  - Can we avoid starvation?
  - What about load balancing?

- **Ease-of-use**
  - Does it use standard mechanisms?
**OUR APPROACH: NOTIFIED ACCESS**

- Notifications with MPI-1 (queue-based) matching
  - Retains benefits of previous notification schemes
  - Poll only head of queue
  - Provides linearizable semantics

![Diagram showing process A, B, and C with PUT + tag X and PUT + tag Y interactions.](image)
NOTIFIED ACCESS – AN MPI INTERFACE

- Minor interface evolution
  - Leverages MPI two sided <source, tag> matching
  - Wildcards matching with FIFO semantics

Example Communication Primitives

```c
int MPI_Put (void *origin_addr, int origin_count, MPI_Datatype origin_type, int target_rank,
             MPI_Aint target_disp, int target_count, MPI_Datatype target_type, MPI_Win win);

int MPI_Get (void *origin_addr, int origin_count, MPI_Datatype origin_type, int target_rank,
             MPI_Aint target_disp, int target_count, MPI_Datatype target_type, MPI_Win win);
```

Example Synchronization Primitives

```c
/*Functions already available in MPI*/
int MPI_Start(MPI_Request *request);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```
NOTIFIED ACCESS – AN MPI INTERFACE

- Minor interface evolution
  - Leverages MPI two sided <source, tag> matching
  - Wildcards matching with FIFO semantics

Example Communication Primitives

```c
#include <mpi.h>

int MPI_Put_notify(void *origin_addr, int origin_count, MPI_Datatype origin_type, int target_rank, 
                   MPI_Aint target_disp, int target_count, MPI_Datatype target_type, MPI_Win win, 
                   int tag);

int MPI_Get_notify(void *origin_addr, int origin_count, MPI_Datatype origin_type, int target_rank, 
                   MPI_Aint target_disp, int target_count, MPI_Datatype target_type, MPI_Win win, 
                   int tag);
```

Example Synchronization Primitives

```c
int MPI_Notify_init(MPI_Win win, int src_rank, int tag, int expected_count, MPI_Request *request);
/*Functions already available in MPI*/
int MPI_Start(MPI_Request *request);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```
**NOTIFIED ACCESS - IMPLEMENTATION**

- **foMPI** – a fully functional MPI-3 RMA implementation
  - Runs on newer Cray machines (Aries, Gemini)
  - DMAPP: low-level networking API for Cray systems
  - XPMEM: a portable Linux kernel module
- **Implementation of Notified Access via uGNI [1]**
  - Leverages uGNI queue semantics
  - Adds unexpected queue
  - Uses 32-bit immediate value to encode source and tag

**EXPERIMENTAL SETTING**

- **Piz Daint**
  - Cray XC30, Aries interconnect
  - 5'272 computing nodes (Intel Xeon E5-2670 + NVIDIA Tesla K20X)
  - Theoretical Peak Performance 7.787 Petaflops
  - Peak Network Bisection Bandwidth 33 TB/s

PING PONG PERFORMANCE (INTER-NODE)

- 1000 repetitions, each timed separately, RDTSC timer
- 95% confidence interval always within 1% of median

PING PONG PERFORMANCE (INTRA-NODE)

- 1000 repetitions, each timed separately, RDTSC timer
- 95% confidence interval always within 1% of median

COMPUTATION/COMMUNICATION OVERLAP

- 1000 repetitions, each timed separately, RDTSC timer
- 95% confidence interval always within 1% of median

PIPELINE – ONE-TO-ONE SYNCHRONIZATION

- 1000 repetitions, each timed separately, RDTSC timer
- 95% confidence interval always within 1% of median

MPI Message Passing
MPI One Sided
Notified Access

(lower is better)

Number of Processes
1.1x 1.32x 1.43x 1.52x 1.58x 2.17x

Normalized Completion Time

[1] https://github.com/intelsg/PRK2
**Reduce – One-to-Many Synchronization**

- Reduce as an example (same for FMM, BH, etc.)
  - Small data (8 Bytes), 16-ary tree
  - 1000 repetitions, each timed separately with RDTSC

![Graph showing performance of different operations (Notified Access, MPI Message Passing, MPI One Sided PSCW, MPI Reduce) with respect to number of processes (4 to 128). The lower the time, the better.](image)

*(lower is better)*
**CHOLESKY – MANY-TO-MANY SYNCHRONIZATION**

- 1000 repetitions, each timed separately, RDTSC timer
- 95% confidence interval always within 10% of median

![Graph](image-url)

(Higher is better)

- Notified Access
- MPI Message Passing
- MPI One Sided

[1]: J. Kurzak, H. Ltaief, J. Dongarra, R. Badia: "Scheduling dense linear algebra operations on multicore processors“, CCPE 2010
DISCUSSION AND CONCLUSIONS

- Performance of cache-coherency is hard to model
  - Min/max models
- RDMA+SHM are de-facto hardware mechanisms
  - Gives rise to RMA programming
- MPI-3 RMA standardizes clear semantics
  - Builds on existing practice (UPC, CAF, ARMCI etc.)
  - Rich set of synchronization mechanisms
- Notified Access can support producer/consumer
  - Maintains benefits of RDMA
- Fully parameterized LogGP-like performance model
  - Aids algorithm development and reasoning

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Notify_init</td>
<td>$t_{init} = 0.07\mu s$</td>
</tr>
<tr>
<td>MPI_Request_free</td>
<td>$t_{free} = 0.04\mu s$</td>
</tr>
<tr>
<td>MPI_Start</td>
<td>$t_{start} = 0.008\mu s$</td>
</tr>
<tr>
<td>MPI_{Put</td>
<td>Get}_notify</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS
ACKNOWLEDGMENTS

IDEA: RMA NOTIFICATIONS
- First seen in Split-C (1992)
- Combine communication and synchronization using RDMA
- RDMA networks can provide various notifications
  - Flags
  - Counters
  - Event Queues

CHOLESKY – MANY-TO-MANY SYNCHRONIZATION
- 1000 repetitions, each timed separately, RDTSC timer
- 95% confidence interval always within 10% of median

PERFORMANCE MODELING
- Performance functions for synchronization protocols
  - Fence
  - PSCGW
  - Locks
- Performance functions for communication protocols
  - Put/Get
  - Atomics

PERFORMANCE INTER-NODE: LATENCY
- Put Inter-Node
- Get Inter-Node

PERFORMANCE: APPLICATIONS
- Annotations represent performance gain of fMpi over Cray MPI-1 [3]