TORSTEN HOEFLER

Remote Memory Access Programming: Faster Parallel Computing Without Messages
with S. Ramos, R. Gerstenberger, M. Besta, R. Belli @ SPCL
presented at Tsinghua University, Beijing, China, November 2015

Founded 1855 - to support the industrialization of Switzerland

Top-20 general, Top-10 Computer Science

HPC at CSCS - part of ETH

Piz Daint
5272 compute nodes
Xeon E5 + K20X
7.78 PF peak

18k students, 466 professors, 1.5 B budget

Swiss HPC Community - PASC
Scientific Benchmarking of Parallel Computing Systems, SC15
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?
Example: Message Passing, Log(G)P

A Practical Model of Parallel Computation

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-world parallel computer.

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


Broadcast Problem

Optimal Solution

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

CACM 1996
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:**
  - $a = 0\text{ns}$
  - $b = 320\text{ns}$
  - $c = 56.2\text{ns}$

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

- Assume single cache line → 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: \( d=2, k_1=3, k_2=2 \):

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

\( c = \) DTD contention

\( b = \) transmit latency

\[
\begin{align*}
t_0 &= 0 \\
t_1 &= k_1^*c + b \\
t_2 &= t_1 + k_2^*c + b
\end{align*}
\]
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

---

*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

---

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

---

Small Broadcast (8 Bytes)

- Broadcast
- Min–Max Model
- Intel MPI

Latency (us)

Number of Threads

Barrier

Latency (us)

Number of threads

- Barrier
- Min–Max Model
- Intel MPI
- Intel OpenMP

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
MPI-1 MESSAGE PASSING – SIMPLE EAGER

Producer

Consumer

MPI-1 MESSAGE PASSING – SIMPLE EAGER

MPI-1 MESSAGE PASSING – SIMPLE EAGER


Producer

Send

1. Data transfer to intermediate buffer

2. Acknowledgement

Consumer

Mailbox

: origin aware of completion
Critical path: 1 latency + 1 copy

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

Producer

Consumer

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

1. Transfer of communication parameters

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

1. Transfer of communication parameters
2. Message matching

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

1. Transfer of communication parameters
2. Message matching
3. Request
4. Data transfer

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

Critical path: 3 latencies

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 reborn 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

http://www.hpcwire.com/2006/08/18/a_critique_of_rdma-1/
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

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 COMMUNICATION OVERVIEW

**Process A (passive)**

**Memory**

**Process B (active)**

**Memory**

**MPI window**

**Get**

**Atomic communication calls (put, get)**

**Process C (active)**

**MPI window**

**Put**

**Atomic communication calls**

**Process D (active)**

**Atomic (Acc, Get & Acc, CAS, FAO)**

Gropp, Hoefler, Thakur, Lusk: Using Advanced MPI
**MPI-3 RMA Communication Overview**

Process A (passive)

- Memory
  - MPI window

Process B (active)

- Memory
  - MPI window
  - Non-atomic communication calls (put, get)

Process C (active)

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

Process D (active)

- Atomic
- Get

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

Process A (passive)

Memory

Put

Process B (active)

Memory

Non-atomic communication calls (put, get)

MPI window

Process C (active)

Atomic

Get

Process D (active)

Atomic communication calls (Acc, Get & Acc, CAS, FAO)
MPI-3 RMA COMMUNICATION OVERVIEW

Process A (passive)

Memory

Put

Non-atomic communication calls (put, get)

Process C (active)

Process B (active)

Memory

Get

MPI window

Atomic

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

Process D (active)
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

- 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
- Active process
- Passive process

Passive Target Mode
- Lock
- Lock All

Fence
- Post/Start/Complete/Wait
- Communication
- Synchronization
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, Hoefler, Thakur, Lusk: Using Advanced MPI
MPI-3 RMA SYNCHRONIZATION OVERVIEW

Active Target Mode

- Active process
- Passive process

Passive Target Mode

- Synchronization
- Communication

Fence

Post/Start/Complete/Wait

Lock

Lock All

Gropp, Hoefler, 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

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

- Graph showing latency (in us) vs. size (in Bytes) for various transport layers.
- Highlighted that DMAPP protocol change results in a 80% faster performance.

**Get Inter-Node**

- Graph showing latency (in us) vs. size (in Bytes) for various transport layers.
- Highlighted that DMAPP protocol change results in a 20% faster performance.

**Half ping-pong**

- Diagram illustrating communication between Proc 0 and Proc 1:
  - "put" operation
  - "sync memory" operation

---

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

Latency [μs]

Size [Bytes]

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

Half ping-pong
Proc 0
put
sync memory
Proc 1

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

**Inter-Node**

- DMAPP protocol change
- Transport Layer: FOMPI MPI-3.0, Cray UPC, Cray MPI-2.2, Cray MPI-1, Cray CAF

**Intra-Node**

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

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

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

![Graph showing scalability and performance comparison](image)

- **Time bound**: $O(\log p)$
- **Memory bound**: $O(1)$

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

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

Performance functions for communication protocols

| Put/get | \( P_{\text{put}} = 0.16\text{ns} \cdot s + 1\mu s \)  
\( P_{\text{get}} = 0.17\text{ns} \cdot s + 1.9\mu s \) |
|---------|-------------------------------------------------|
| Atomics | \( P_{\text{acc,sum}} = 28\text{ns} \cdot s + 2.4\mu s \)  
\( P_{\text{acc,min}} = 0.8\text{ns} \cdot s + 7.3\mu s \) |
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

- **Graph:**
  - Billion Inserts per Second
  - Number of Processes: 2, 8, 32, 128, 512, 2048, 8192, 32768
  - **intra-node**
  - **inter-node**

---

**Dynamic Sparse Data Exchange (DSDE) with 6 neighbors**

- **Graph:**
  - Time [μs]: 25, 100, 1000, 10000
  - Number of Processes: 8, 32, 128, 512, 2048, 8192, 32768
  - Transport Layer:
    - FOMPI MPI–3.0
    - LibNBC
    - Cray MPI–2.2
    - Cray Reduce_scatter
    - Cray Alltoall
**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
Available in most MPI libraries today
Some are even fast!

IN CASE YOU WANT TO LEARN MORE

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

Consumer

Put

1. Data transfer

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

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

![Diagram of RMA Notifications](image-url)
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 $\rightarrow$ 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

---

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);
```

```c
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
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

**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 completion times for different processes](image)

**Completion Time (us)**

- Notified Access
- MPI Message Passing
- MPI One Sided PSCW
- MPI Reduce

*(lower is better)*

**Number of Processes**

- 4
- 8
- 16
- 32
- 64
- 128
CHOLESKY – MANY-TO-MANY SYNCHRONIZATION

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

(Higher is better)

[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>Shared Memory</th>
<th>uGNI FMA</th>
<th>uGNI BTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>0.25µs</td>
<td>1.02µs</td>
</tr>
<tr>
<td>G</td>
<td>0.08ns</td>
<td>0.105ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{init}$</td>
<td>0.07µs</td>
</tr>
<tr>
<td>$t_{free}$</td>
<td>0.04µs</td>
</tr>
<tr>
<td>$t_{start}$</td>
<td>0.008µs</td>
</tr>
<tr>
<td>$t_{na}$</td>
<td>0.29µs</td>
</tr>
</tbody>
</table>
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, RDTS timer
- 95% confidence interval always within 10% of median

PERFORMANCE MODELING
Performance functions for synchronization protocols

Fence
- \( P_{\text{Fence}} = 2.9\mu s \cdot \log_2(N) \)

PSCW
- \( P_{\text{Fence}} = 0.7\mu s, P_{\text{Fence}} = 1.8\mu s \)
- \( P_{\text{Fence}} = 3.0\mu s \cdot N \)

Locks
- \( P_{\text{Lock}} = 5.6\mu s \)
- \( P_{\text{Lock}} = 2.7\mu s \)
- \( P_{\text{Lock}} = 6.8\mu s \)
- \( P_{\text{Lock}} = 76\mu s \)
- \( P_{\text{Lock}} = 17\mu s \)

Performance functions for communication protocols

Put/Get
- \( P_{\text{Put}} = 0.16\mu s \cdot z + 1.1\mu s \)
- \( P_{\text{Get}} = 0.17\mu s \cdot z + 3.3\mu s \)

Atomic
- \( P_{\text{Atomic}} = 28\mu s \cdot z + 2.4\mu s \)
- \( P_{\text{Atomic}} = 8.9\mu s \cdot z + 7.3\mu s \)

PERFORMANCE INTER-NODE: LATENCY

Put Inter-Node
- 80% faster

Get Inter-Node
- 20% faster

PERFORMANCE: APPLICATIONS

Annotions represent performance gain of fMpi over Cray MPI-1 [3]

NAS 3D FFT [1] Performance
MilC [2] Application Execution Time