TORSTEN HOEFLER

Remote Memory Access Programming: Faster Parallel Computing Without Messages

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

presented at the School of Comp. Science and Engineering, Georgia Tech, 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?
**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-world parallel computers.

PRAM consists of a collection of processors which compute synchronously in parallel and communicate with a global random access memory. A new parallel machine model reflects the critical technology trends underlying parallel computers.

---

**A Practical Model of Parallel Computation**

**Optimal Solution**

**Broadcast Problem**

---

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 $R_l = 278$ 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"
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: \( d=2, k_1=3, k_2=2 \):

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

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

\( c = \text{DTD contention} \)

\( b = \text{transmit latency} \)
Finding the Optimal Broadcast Algorithm

- Broadcast example:

Broadcast cost

\[
\mathcal{T}_{\text{tree}} = \sum_{i=1}^{d} \mathcal{T}_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b)
\]

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

Number of levels

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

Reached threads

\[
n_{th} \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j
\]
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

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

---

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 😡
  - Complicates modeling and is expensive

- 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

MPI-1 MESSAGE PASSING – SIMPLE EAGER

Producer

Send

1. Data transfer to intermediate buffer

Consumer

Mailbox

MPI-1 MESSAGE PASSING – SIMPLE EAGER

Producer

Consumer

Send

1. Data transfer to intermediate buffer

2. Acknowledgement

: origin aware of completion

MPI-1 MESSAGE PASSING – SIMPLE EAGER

1. Data transfer to intermediate buffer
2. Acknowledgement
3. Message matching and copy

Critical path: 1 latency + 1 copy

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

Producer

Consumer

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

Producer

Send

Consumer

1. Transfer of communication parameters

Mailbox

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

1. Transfer of communication parameters
2. Message matching
3. Request

Producer

Send

Consumer

Mailbox

Recv

MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

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

*: target aware of completion

**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 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
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

“Traditionally, HPC programming models are following hardware developments” (IPDPS’15)

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
  - Non-atomic communication calls (put, get)

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

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

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

Process A (passive)

Memory

MPI window

Put

Process B (active)

Memory

MPI window

Non-atomic communication calls (put, get)

Atomic

Process C (active)

Get

Process D (active)

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

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

Process A (passive)
Memory

Process B (active)
Memory

Process C (active)

Process D (active)

Non-atomic communication calls (put, get)

Atomic communication calls (Acc, Get & Acc, CAS, FAO)
**MPI-3 RMA Communication Overview**

- **Process A (passive)**
  - Memory

- **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
  - MPI window

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

- **Process C (active)**
  - Non-atomic communication calls (put, get)

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

**Diagram Overview**
- **Put**
- **Get**
- **Atomic**

---

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

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

Passive Target Mode
- Lock
- Lock All

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
- Lock
- Lock All

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, Hoefler, Thakur, Lusk: Using Advanced MPI
MPI-3 RMA Synchronization Overview

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

- Active process
- Passive process

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

- Communication
- Synchronization

Gropp, Hoefler, Thakur, Lusk: Using Advanced MPI
Scalable & generic protocols
- Can be used on any RDMA network (e.g., OFED/IB)
- Window creation, communication and synchronization

Window creation

MPI-3 RMA COMMUNICATION OVERVIEW

Active Target Mode
- Active process
- Passive process

Passive Target Mode
- Lock
- Lock All

Synchronization
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 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

**Get Inter-Node**
- 20% faster

Half ping-pong

Proc 0  
<table>
<thead>
<tr>
<th>put</th>
<th>sync memory</th>
</tr>
</thead>
</table>

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**

![Graph showing latency vs size for different transport layers.](attachment:plot.png)

- **Half ping-pong**
  - Proc 0
  - Proc 1
  - `put`
  - `sync memory`

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

Active Target Mode

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

![Graph showing latency vs. number of processes](graph.png)

- **Global Synchronization**
  - FOMPI Win_fence
  - Cray UPC barrier
  - Cray CAF sync_all
  - Cray MPI Win_fence

- **Latency [us]**
  - 2
  - 10
  - 100
  - 1000
  - 10000

- **Number of Processes**
  - 2
  - 8
  - 32
  - 128
  - 512
  - 2k
  - 8k

- **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

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

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

Performance functions for synchronization protocols

<table>
<thead>
<tr>
<th>Fence</th>
<th>( P_{fence} = 2.9\mu s \cdot \log_2(p) )</th>
</tr>
</thead>
</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

| 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: APPLICATIONS**


---

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

- ![Graph showing NAS 3D FFT performance](image)

**MILC [2] Application Execution Time**

- ![Graph showing MILC application execution time](image)
- Scale to 65k procs
- Scale to 512k procs

---

[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

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

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

Message Passing

1 latency + copy / 3 latencies

One Sided
3 latencies

Put
Flush

Explicit Synch

1. Data transfer
2. Acknowledgement
3. Producer reports completion to consumer

Explicit Synch

1: origin aware of completion
2: target aware of completion

**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

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
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
**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

---

![Graph showing normalized completion time for different synchronization methods.](lower_is_better.png)

**[1] https://github.com/inteleqs/PRK2**
Reduce as an example (same for FMM, BH, etc.)
- Small data (8 Bytes), 16-ary tree
- 1000 repetitions, each timed separately with RDTSC

Diagram showing completion time in microseconds for different processes and methods:
- Notified Access
- MPI Message Passing
- MPI One Sided PSCW
- MPI Reduce

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

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

(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>Shared Memory</th>
<th>uGNI FMA</th>
<th>uGNI BTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>0.25(\mu s)</td>
<td>1.02(\mu s)</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>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
Hardware Reality

Performance Modeling
Performance functions for synchronization protocols:

- Fence
  - \( P_{\text{fence}} = 2.9 \times \log_2(D) \)
- PSCW
  - \( P_{\text{scw}} = 0.7 D \times \log_2(D) \)
  - \( P_{\text{scw}} = 1 \times D \)
- Latches
  - \( P_{\text{latch}} = 5 \mu s \)
  - \( P_{\text{latch}} = 2.7 \mu s \)
- Locks
  - \( P_{\text{lock}} = 2.7 \mu s \)
  - \( P_{\text{lock}} = 2.7 \mu s \)
- Performance functions for communication protocols:
  - Put/Get
    - \( P_{\text{put}} = 0.16 \times D + 1 \mu s \)
    - \( P_{\text{put}} = 0.17 \times D + 1.3 \mu s \)
  - Atomics
    - \( P_{\text{atomic}} = 28 \mu s \)
    - \( P_{\text{atomic}} = 2.4 \mu s \)
    - \( P_{\text{atomic}} = 0.8 \mu s \)

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

Using Advanced MPI
Modern Features of the Message-Passing Interface

William Gropp
Tucstea Hartier
Raviv Thakur
Eewen Laure