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
  - Zero-copy, user-level, os-bypass, fuzz-bang
PRODUCER-CONSUMER RELATIONS

- Most important communication idiom
  - Some examples:

- Perfectly supported by MPI-1 Message Passing
  - But how does this actually work over RDMA?
MPI-1 MESSAGE PASSING – SIMPLE EAGER

MPI-1 MESSAGE PASSING – SIMPLE EAGER

Producer

Consumer

Send

1. Data transfer to intermediate buffer

Mailbox

**MPI-1 MESSAGE PASSING – SIMPLE EAGER**

[Diagram showing the process of message passing between a producer and a consumer.]

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

- : target aware of completion
- : 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

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

Send

Producer

Consumer

Recv

Mailbox

: target aware of completion

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...
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
  - Put
- 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)
MPI-3 RMA COMMUNICATION OVERVIEW

Process A (passive)

Memory

Put

Non-atomic communication calls (put, get)

MPI window

Atomic

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

Get

Process B (active)

Memory

MPI window

Process C (active)

Process D (active)
MPI-3 RMA COMMUNICATION OVERVIEW

Process A (passive)

Memory

Process B (active)

Memory

Process C (active)

Process D (active)

Put

Non-atomic communication calls (put, get)

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

Process A (passive)

Memory

Put

Non-atomic communication calls (put, get)

Atomic

Process B (active)

Memory

MPI window

Get

Process C (active)

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

Process D (active)
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)

- **Atomic**
  - Put
  - Get
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
MPI-3 RMA SYNCHRONIZATION OVERVIEW

Active Target Mode
- Active process
- Passive process

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

Fence
- Post/Start/Complete/Wait
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
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**

- Fence
- Post/Start/Complete/Wait

- Lock
- Lock All
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
IN CASE YOU WANT TO LEARN MORE

How to implement producer/consumer in passive mode?

Using Advanced MPI
Modern Features of the Message-Passing Interface

William Gropp
Torsten Hoefler
Rajeev Thakur
Ewing Lusk
ONE SIDED – PUT + SYNCHRONIZATION

Producer

Consumer
ONE SIDED – PUT + SYNCHRONIZATION

Producer

Put

Consumer

1. Data transfer
ONE SIDED – PUT + SYNCHRONIZATION

Producer

Put

Flush

Consumer

1. Data transfer

2. Producer waits for remote completion

diamond: 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
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);

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

![Graph showing latency for different numbers of transferred bytes for different methods: Notified Access, MPI Message Passing, MPI One Sided, and No Synchronization. Lower values are better.]
**COMPUTATION/COMMUNICATION OVERLAP**

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

![Graph showing computation/communication overlap](image)

- **Notified Access**
- **MPI Message Passing**
- **MPI One Sided**

*Uses communication progression thread*

*Protocol Switch FMA > BTE*

*Protocol Switch Eager > Rendezvous*

*(lower is better)*

(Number of Transferred Bytes: 32, 1024, 32768, 1048576)
**PIPELINE — ONE-TO-ONE SYNCHRONIZATION**

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

---

![Graph](lower is better)

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

(Chart showing completion time in microseconds vs. number of processes, with Notified Access, MPI Message Passing, MPI One Sided PSCW, and 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

Number of Processes

GMOPS

49

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

- **Simple and fast solution**
  - The interface lies between RMA and Message Passing
  - Similarity to MPI-1 eases adoption of NA
  - Richer semantics then current notification systems
  - Maintains benefits of RDMA for producer/consumer

- **Effect on other RMA operations needs to be defined**
  - Either synchronizing [1] or no effect
  - Currently discussed in the MPI Forum

- **Fully parameterized LogGP-like performance model**

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

---

[1]: Kourosh Gharachorloo, et al.. "Memory consistency and event ordering in scalable shared-memory multiprocessors"., ISCA'90
ACKNOWLEDGMENTS

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre
Thank you for your attention
BACKUP SLIDES
NOTIFIED ACCESS - EXAMPLE

```c
MPI_Win win;
MPI_Request notification_request;
MPI_Status notification_status;
int win_size = 2 * MAX_SIZE * sizeof(double);
double *buf; int my_rank;
MPI_Win_allocate(win_size, sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &buf, &win);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/* initialize notification request */
int customTag = 99; int expected_count = 1; int assert = 0;
MPI_Notify_init(win, partner_rank, customTag, expected_count, &notification_request);
MPI_Win_lock_all(assert, win);
for(size=8; size<MAX_SIZE; size++) {
    if (my_rank==client_rank) {
        /* send ping */
        MPI_Put_notify(buf, size, MPI_DOUBLE, partner_rank, 0, size, MPI_DOUBLE, win, customTag);
        MPI_Win_flush(partner_rank, win);
        /* wait for pong */
        MPI_Start(&notification_request);
        MPI_Wait(&notification_request, &notification_status);
    } else { /* server */
        /* wait for ping */
        MPI_Start(&notification_request);
        MPI_Wait(&notification_request, &notification_status);
        /* send pong */
        MPI_Put_notify(buf, size, MPI_DOUBLE, partner_rank, MAX_SIZE, size, MPI_DOUBLE, win, customTag);
        MPI_Win_flush(partner_rank, win);
    }
} /* end of iterations */
MPI_Win_unlock_all(win);
MPI_Request_free(&notification_request);
MPI_Win_free(&win);
```
PERFORMANCE: APPLICATIONS

Annotations represent performance gain of foMPI over Cray MPI-1.

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
PERFORMANCE: MOTIF APPLICATIONS

Key/Value Store: Random Inserts per Second

<table>
<thead>
<tr>
<th>Transport Layer</th>
<th>FOMPI MPI-3.0</th>
<th>Cray UPC</th>
<th>Cray MPI-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra-node</td>
<td>0.025</td>
<td>0.100</td>
<td>1.000</td>
</tr>
<tr>
<td>inter-node</td>
<td>1.000</td>
<td>10.000</td>
<td>100.000</td>
</tr>
</tbody>
</table>

Dynamic Sparse Data Exchange (DSDE) with 6 neighbors

<table>
<thead>
<tr>
<th>Transport Layer</th>
<th>FOMPI MPI-3.0</th>
<th>LibNBC</th>
<th>Cray MPI-2.2</th>
<th>Cray Reduce_scatter</th>
<th>Cray Alltoall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [us]</td>
<td>25</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>
COMPARING APPROACHES – EXAMPLE

Overriding Interface

Flag location
Neighbor sets the flag

Polling

PE E
PE S
PE W

Counting Interface

Counter location
Neighbor increment counter

Polling

4

PE E
PE S
PE W

Notified Access

Neighbor sends tag

Head of the queue

Polling

PE E
PE W
PE S
ONE SIDED – GET + SYNCHRONIZATION
ONE SIDED – GET + SYNCHRONIZATION

Producer

Consumer

1. Data Transfer

Get

: origin aware of completion
ONE SIDED – GET + SYNCHRONIZATION

Critical Path: 3 Messages
COMPARING APPROACHES

**Eager**
- Producer sends to intermediate buffer.
- Consumer receives.
- Mailbox is used.

**Rendezvous**
- Producer sends.
- Consumer receives.
- Meta data transfer.
- Message matching.
- Request.
- Data transfer.
- Acknowledgement.

**Put + Synch**
- Producer puts.
- Consumer receives.
- Data transfer.
- Producer waits for remote completion.
- Producer reports completion to consumer.

**Get + Synch**
- Producer gets.
- Consumer receives.
- Data transfer.
- Consumer reports completion to producer.

**Notified Put**
- Producer puts.
- Consumer waits.
- Data transfer + notification.
- Acknowledgement.

**Notified Get**
- Producer gets.
- Data transfer + notification.
- Wait notification.
- Get.

*: target aware of completion
*: origin aware of completion