

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

**2016**  
**SC**Platform for Advanced Scientific Computing  
Conference

Lausanne Switzerland | 08-10 June 2016

- CLIMATE & WEATHER
- SOLID EARTH
- LIFE SCIENCE
- CHEMISTRY & MATERIALS
- PHYSICS
- COMPUTER SCIENCE & MATHEMATICS
- ENGINEERING
- EMERGING DOMAINS

# 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 new parallel machine model reflects the critical technology trends underlying parallel computers

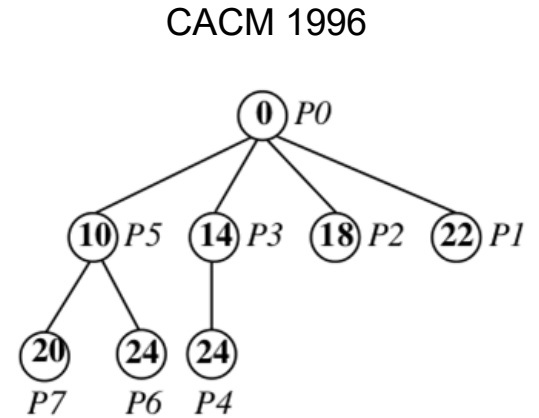
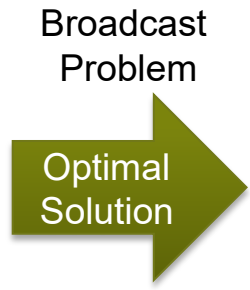
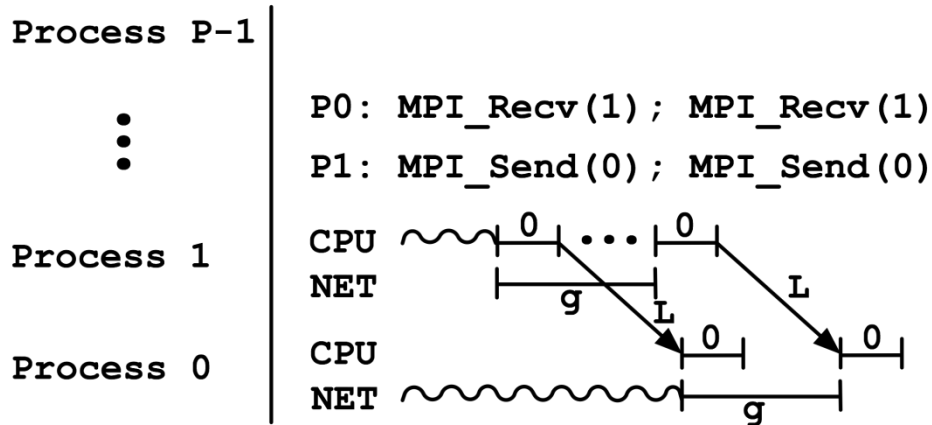
**LogP**

## A PRACTICAL MODEL of PARALLEL COMPUTATION

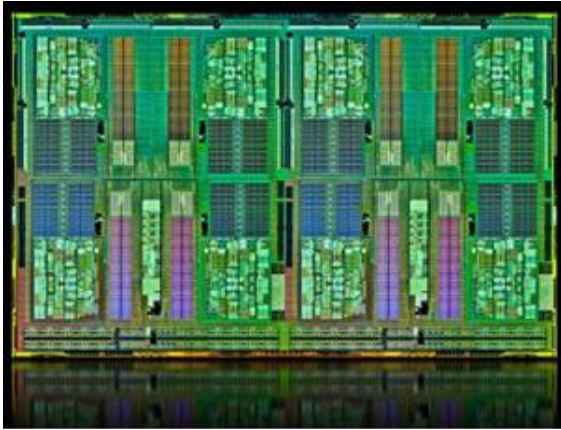
OUR GOAL IS TO DEVELOP A MODEL OF PARALLEL COMPUTATION THAT WILL serve as a basis for the design and analysis of fast, portable parallel algorithms, such as algorithms that can be implemented effectively on a wide variety of current and future parallel machines. If we look at the body of parallel algorithms developed under current parallel models, many are impractical because they exploit artificial factors not present in any rea-

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

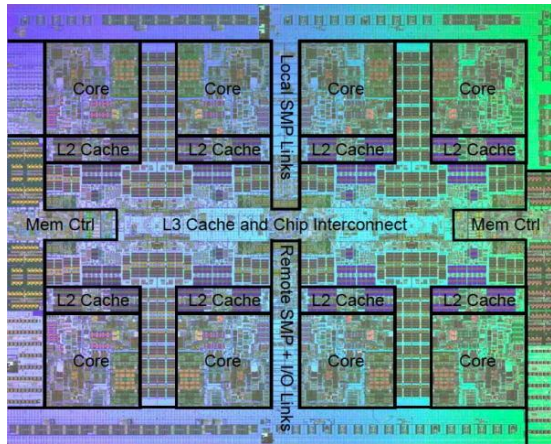
David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauer, Ramesh Subramonian, and Thorsten von Eicken



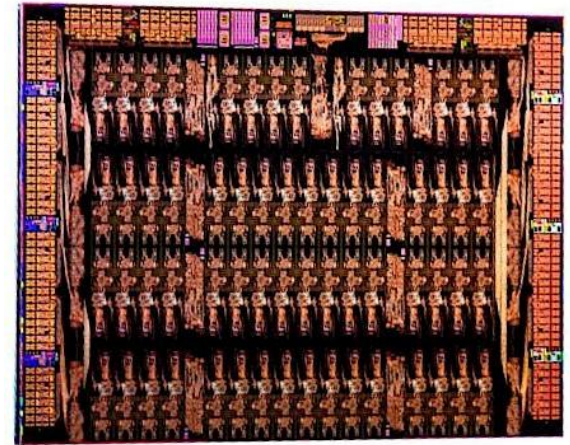
# Hardware Reality



Interlagos, 8/16 cores, source: AMD

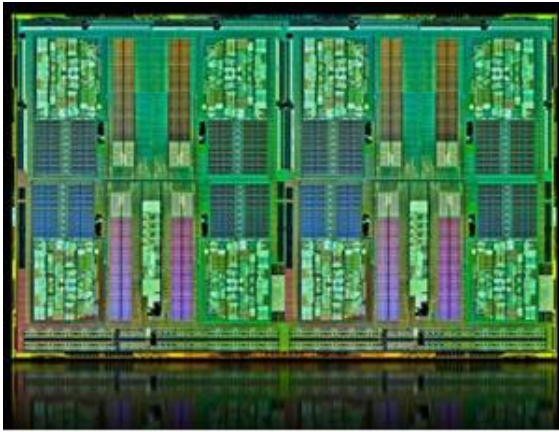


POWER 7, 8 cores, source: IBM

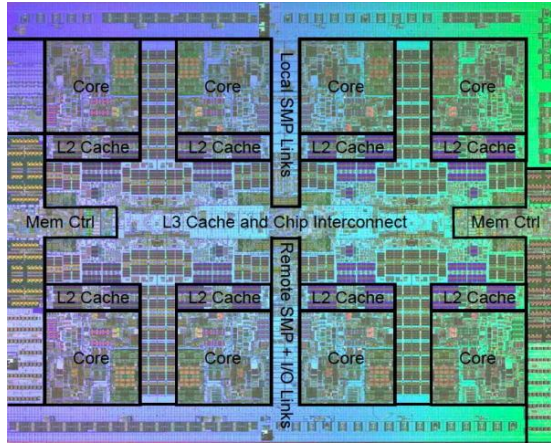


Xeon Phi, 64 cores, source: Intel

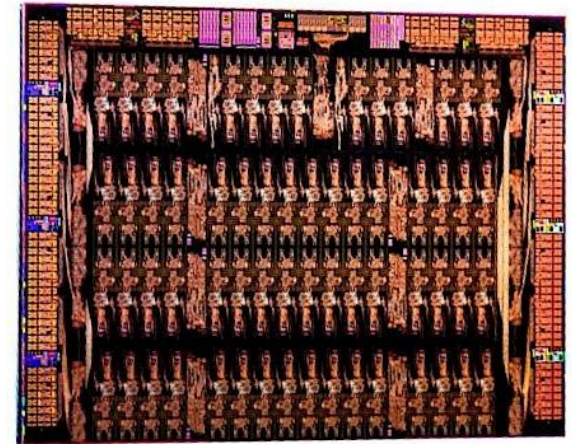
# Hardware Reality



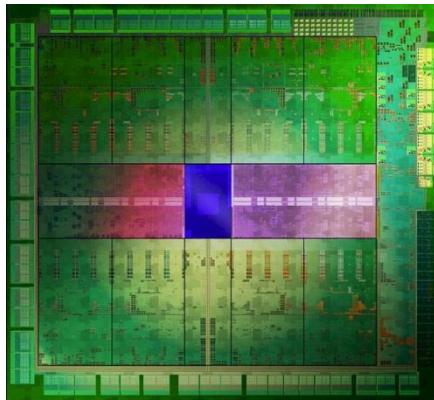
Interlagos, 8/16 cores, source: AMD



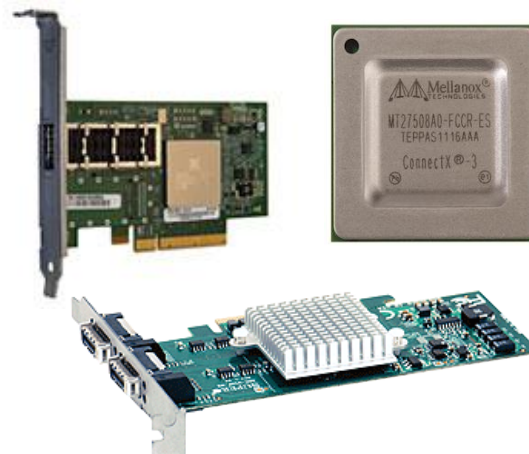
POWER 7, 8 cores, source: IBM



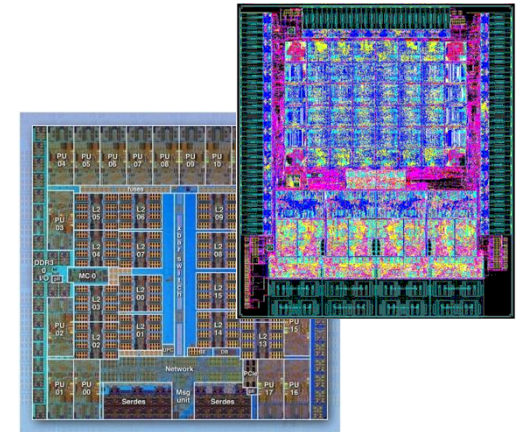
Xeon Phi, 64 cores, source: Intel



Kepler GPU, source: NVIDIA

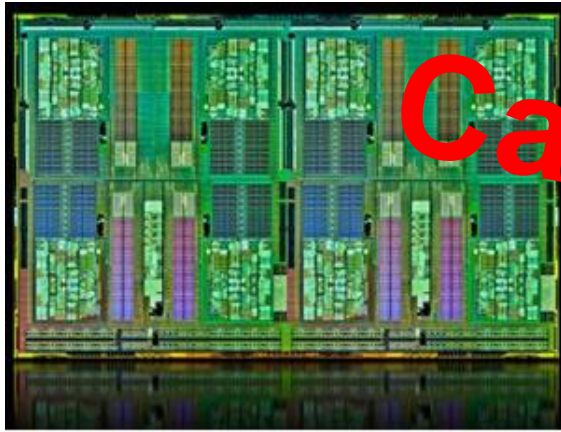


InfiniBand, sources: Intel, Mellanox

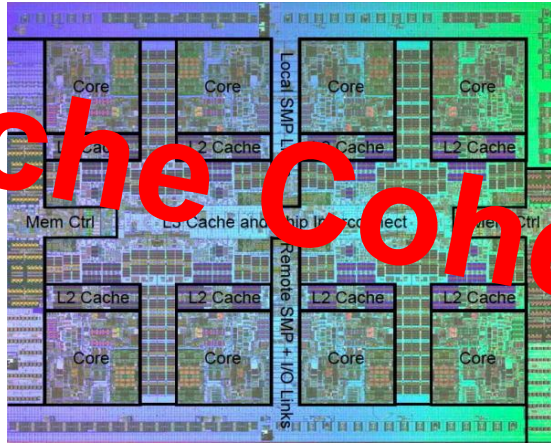


BG/Q, Cray Aries, sources: IBM, Cray

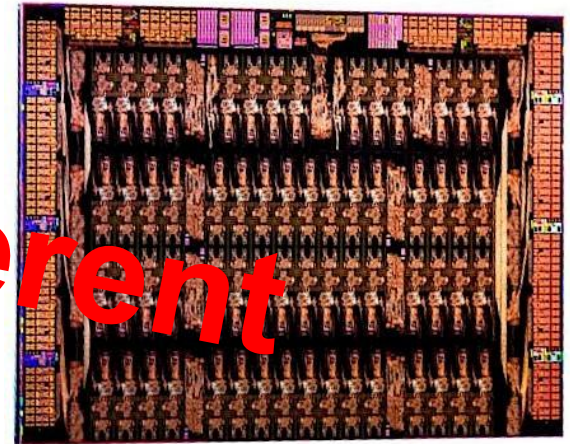
# Hardware Reality



Interlagos, 8/16 cores, source: AMD

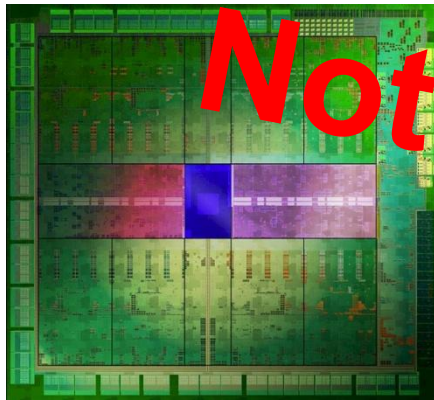


POWER 7, 8 cores, source: IBM



Xeon Phi, 64 cores, source: Intel

Cache Coherent

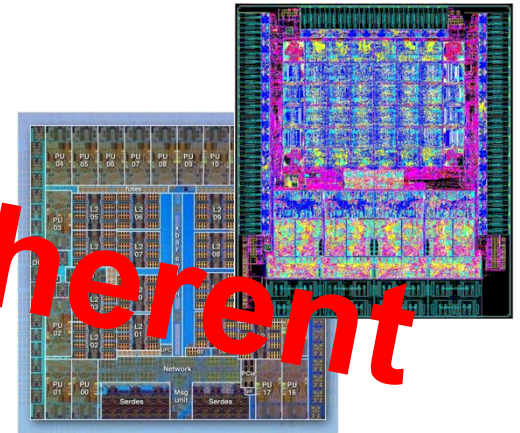


Kepler GPU, source: NVIDIA

Not Cache Coherent

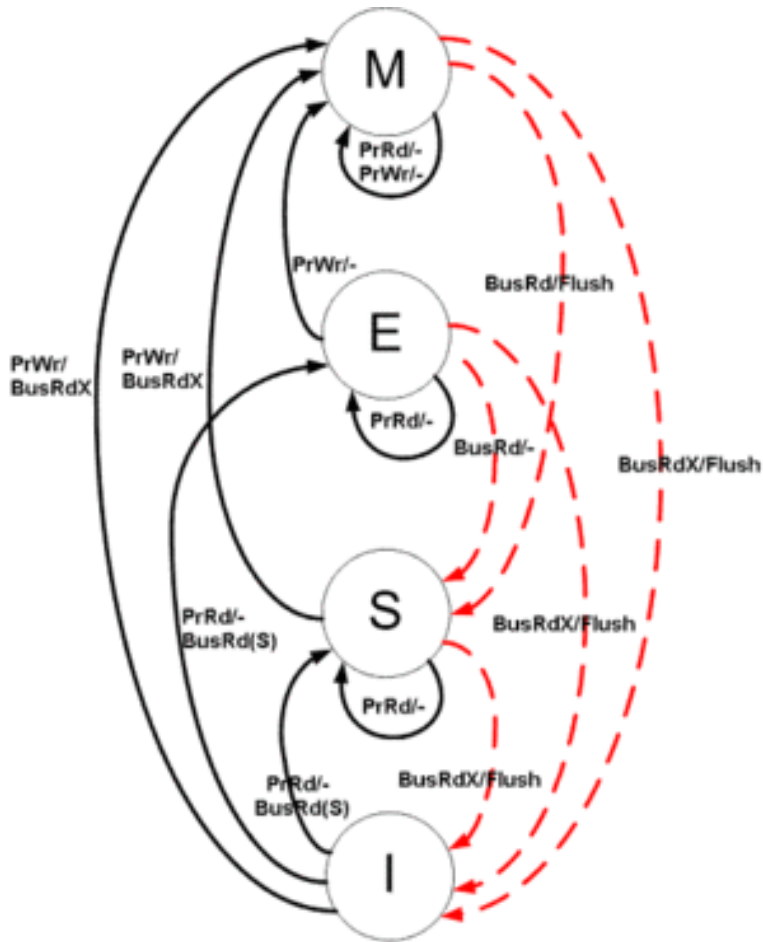


InfiniBand, sources: Intel, Mellanox

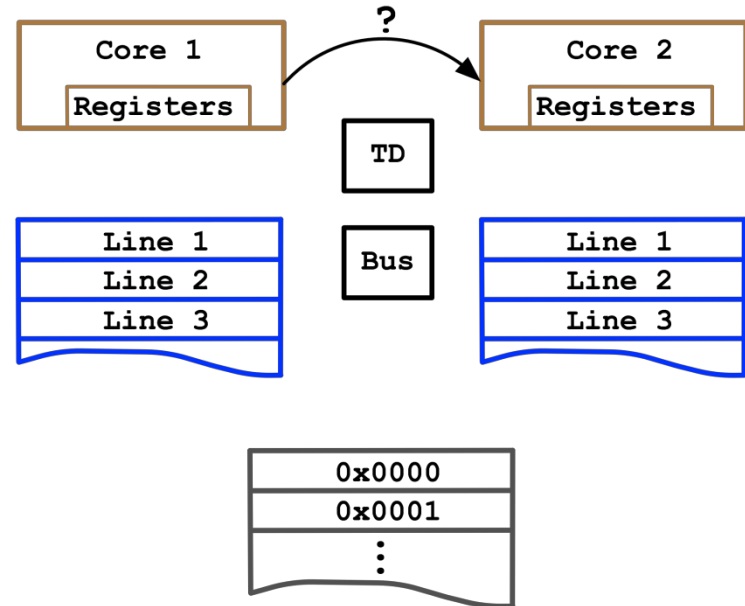


BG/Q, Cray Aries, sources: IBM, Cray

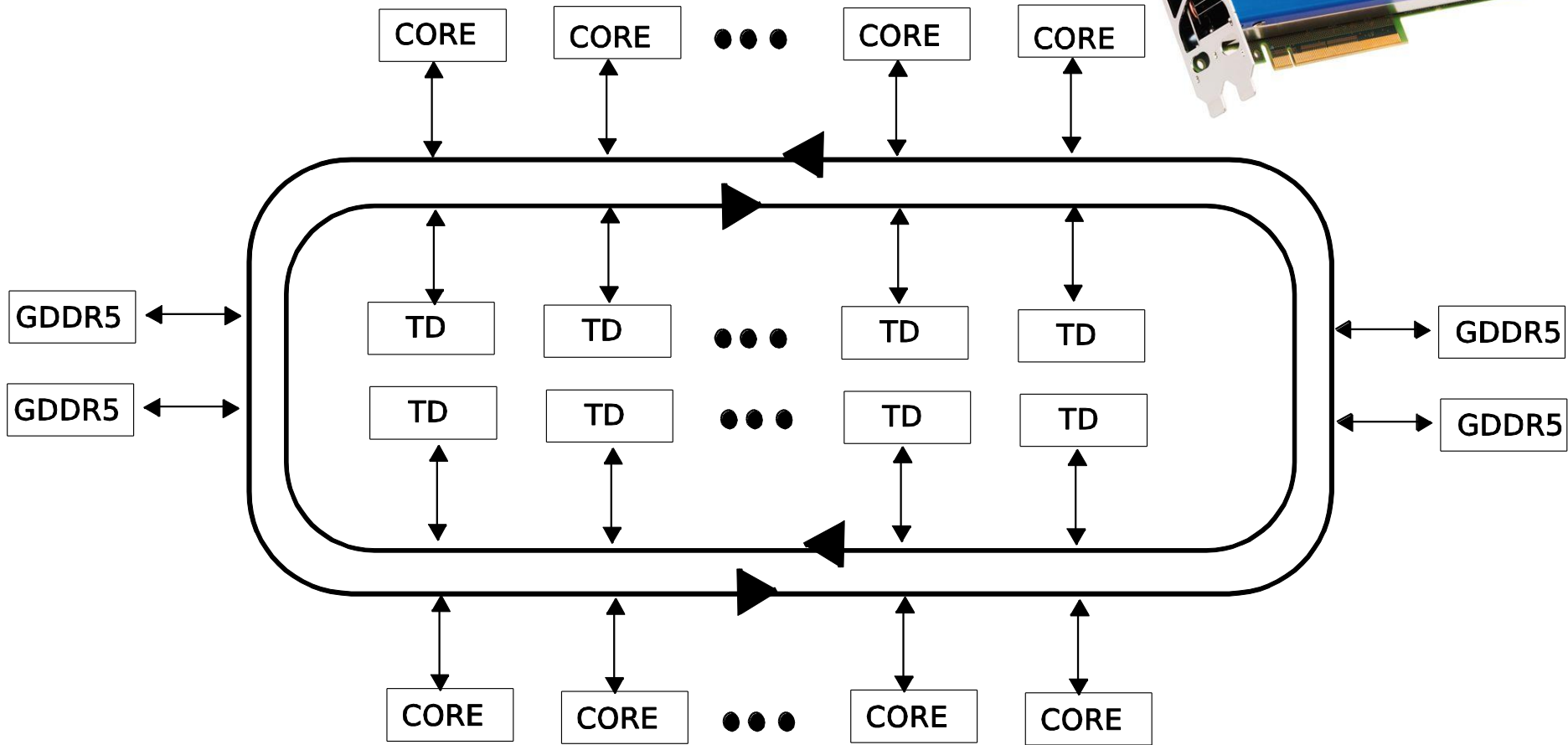
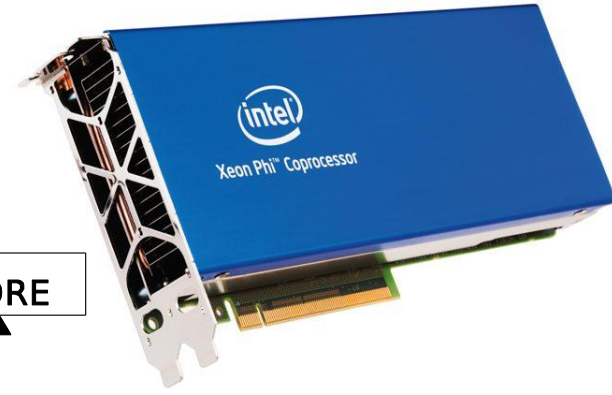
# Example: Cache-Coherent Communication



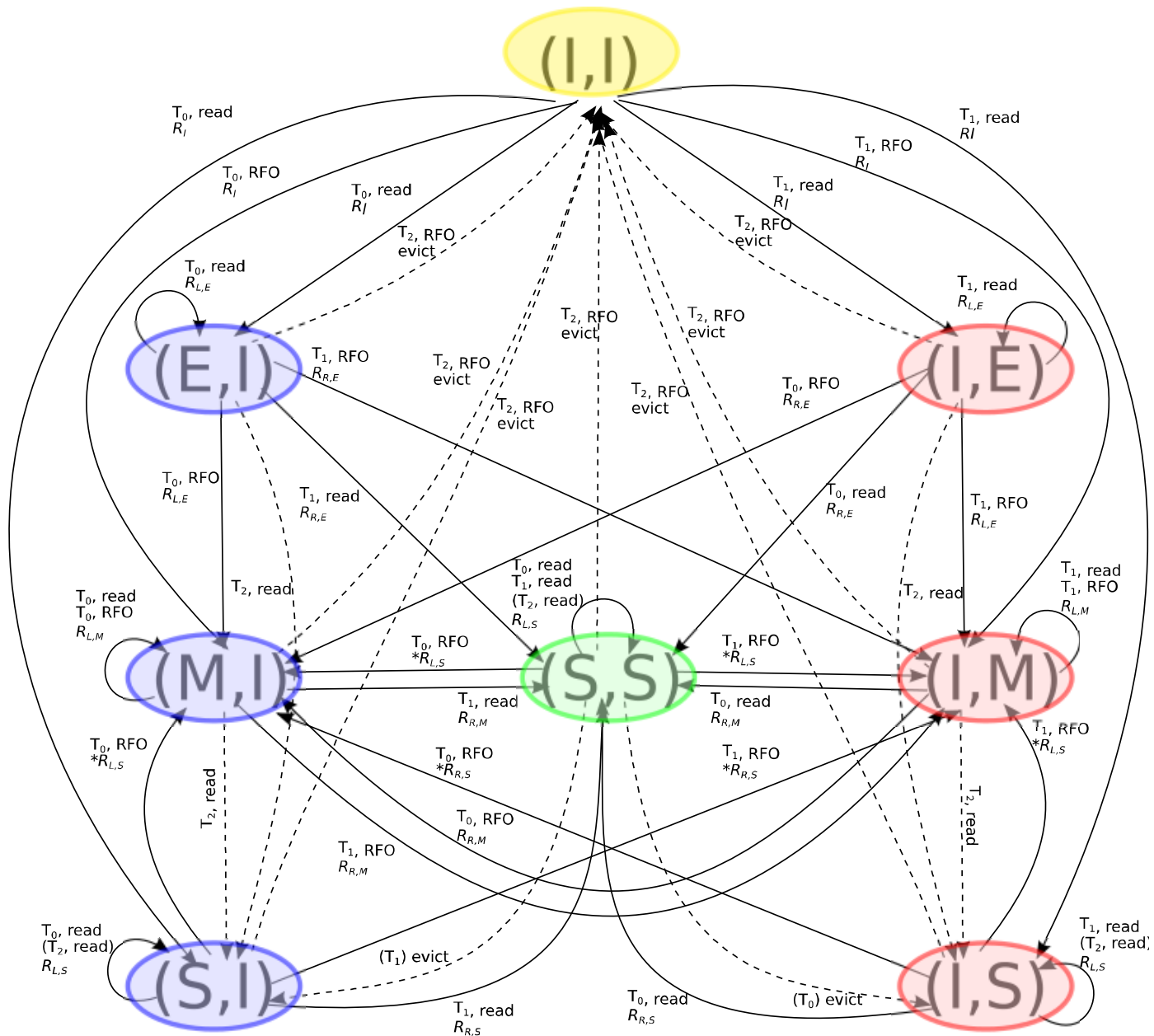
Source: Wikipedia

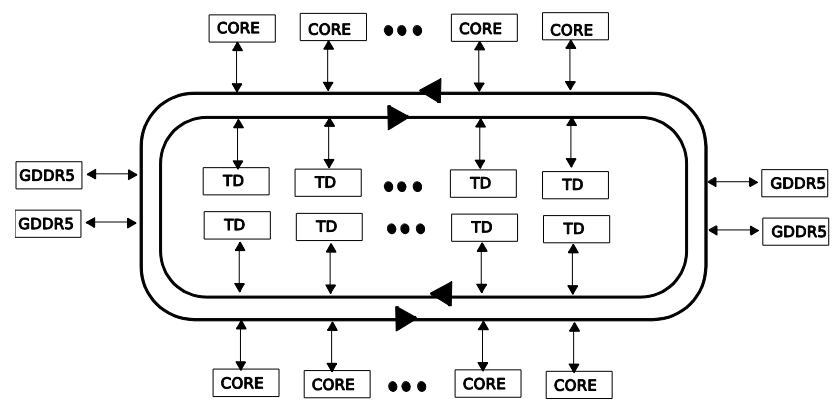
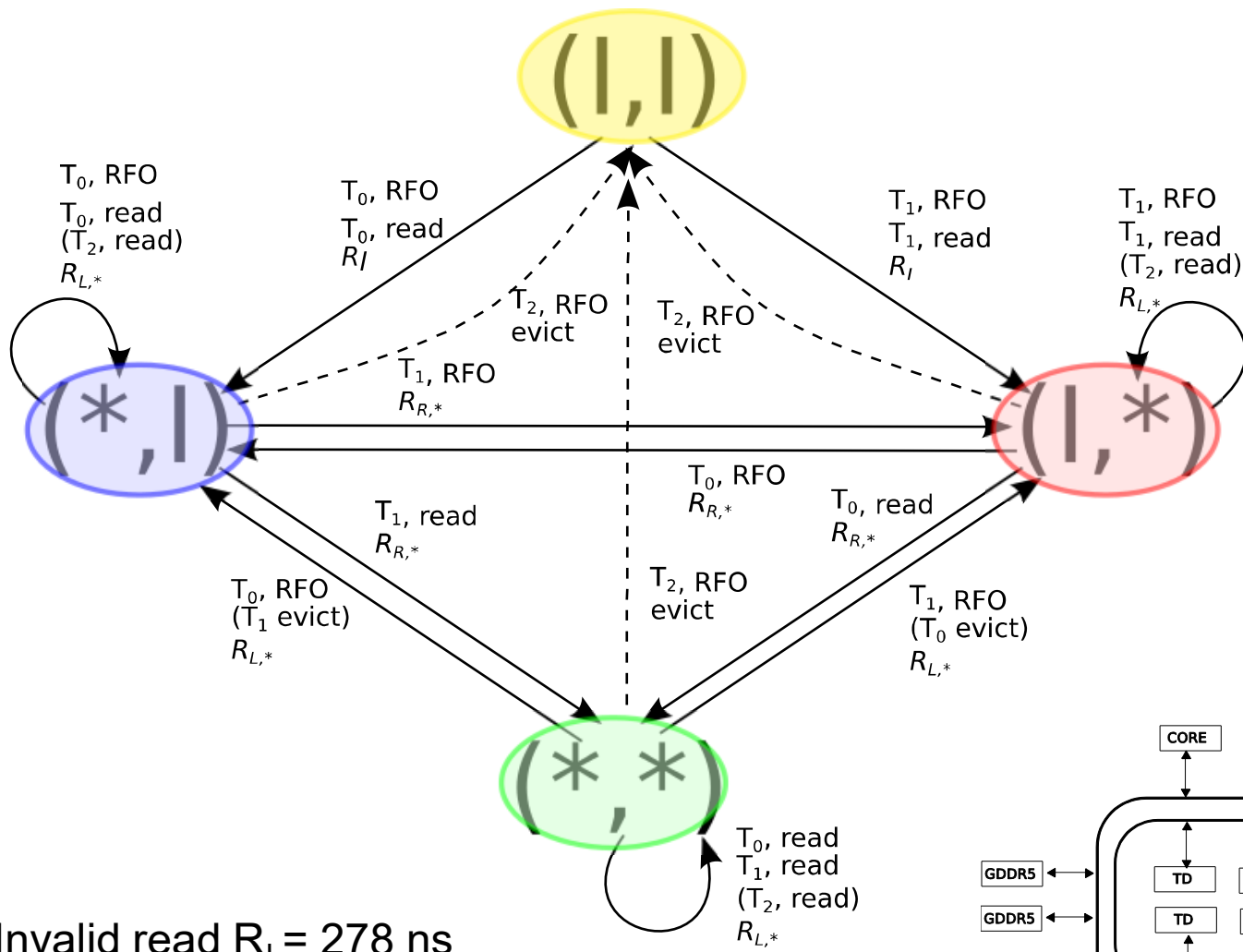


# Xeon Phi (Rough) Architecture





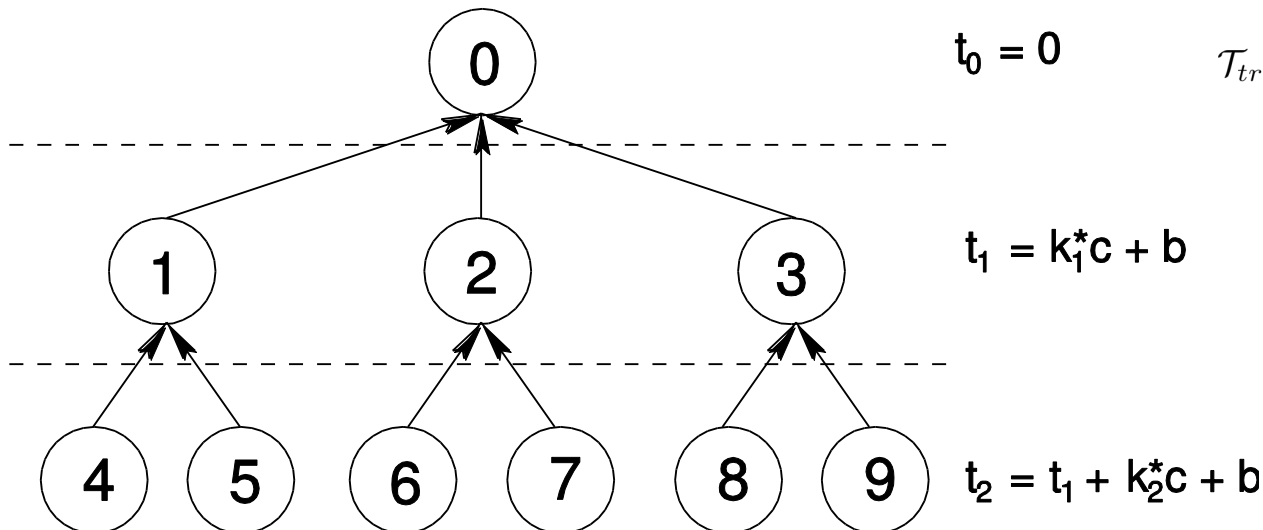




Invalid read  $R_I = 278$  ns  
 Local read:  $R_L = 8.6$  ns  
 Remote read  $R_R = 235$  ns

# 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:  $d=2$ ,  $k_1=3$ ,  $k_2=2$ :



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

$c$  = DTD contention  
 $b$  = transmit latency

# Finding the Optimal Broadcast Algorithm

- Broadcast example:

Bcast cost

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

Number of levels

$$\mathcal{T}_{sbcast} = \min_{d, k_i} \left( \mathcal{T}_{fw} + \sum_{i=1}^d (c \cdot k_i + b) + \sum_{i=1}^d \mathcal{T}_{nb}(k_i + 1) \right)$$

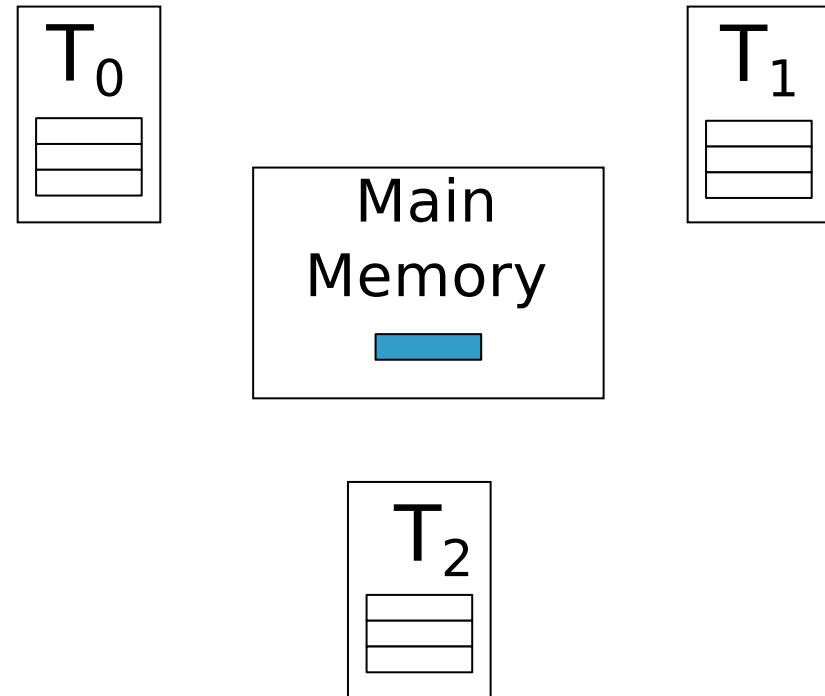
$$N \leq 1 + \sum_{i=1}^d \prod_{j=1}^i k_j, \quad \forall i < j, k_i \leq k_j$$

Reached threads

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

# Min-Max Modeling

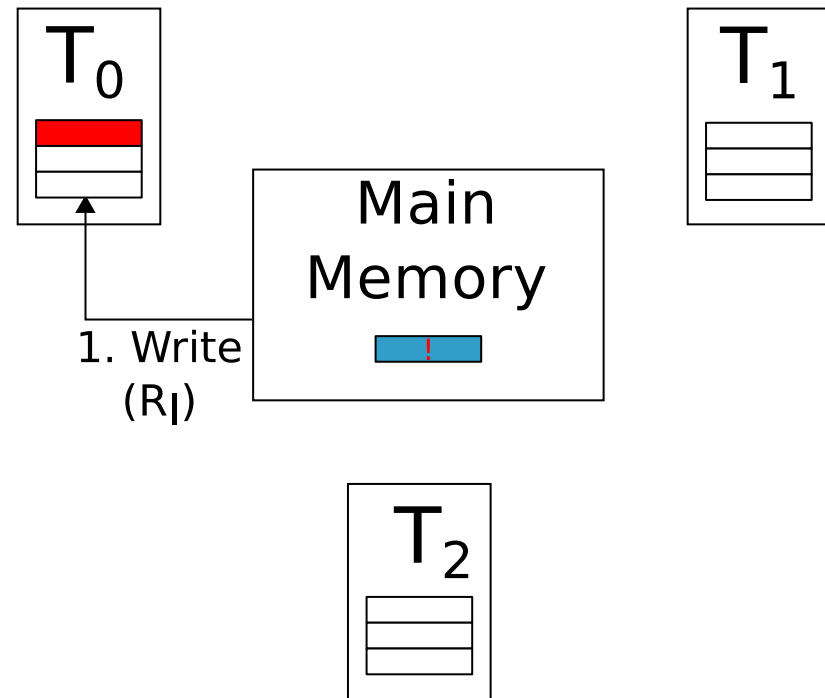
- **Example:**
  - $T_0 + T_1$  write CL
  - $T_2$  polls for completion



# Min-Max Modeling

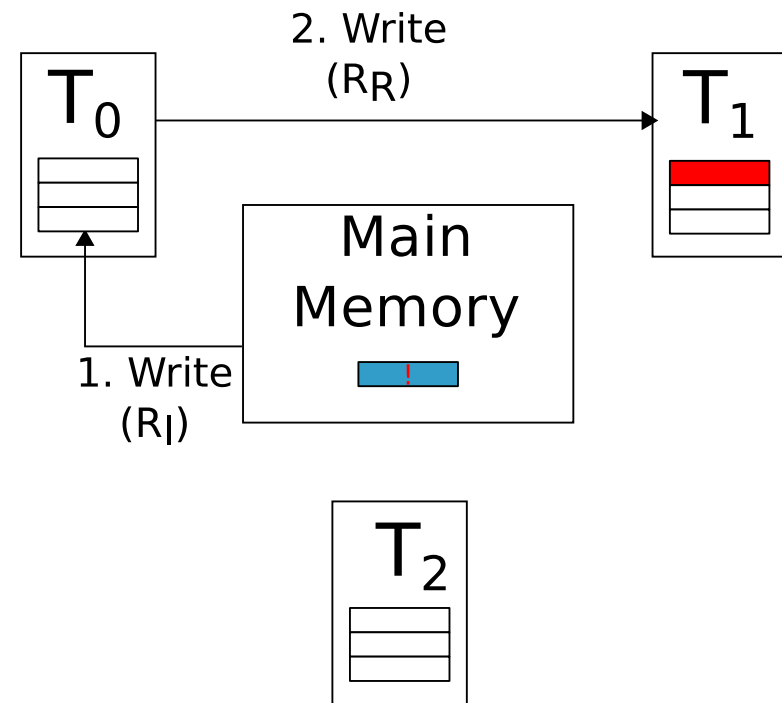
## ■ Example:

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



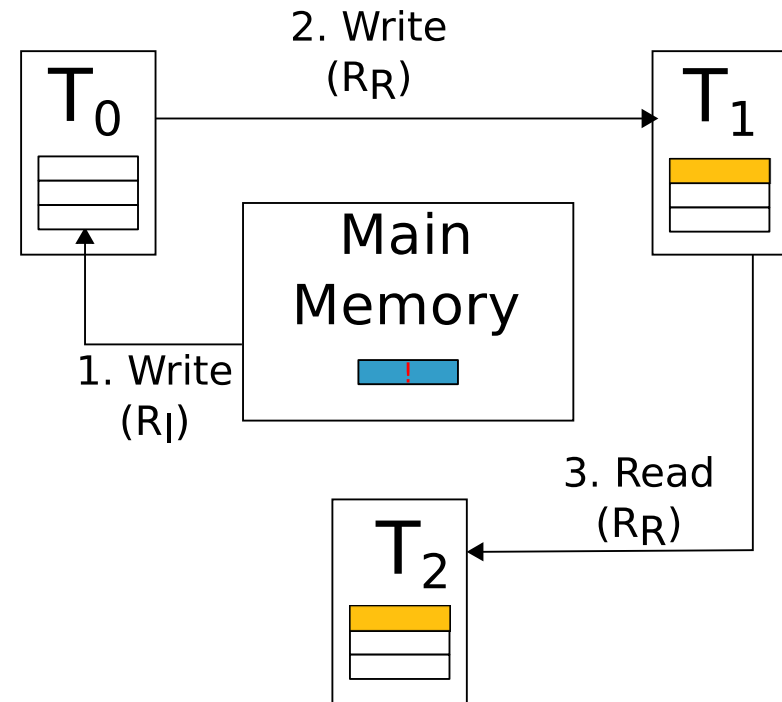
# Min-Max Modeling

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



# Min-Max Modeling

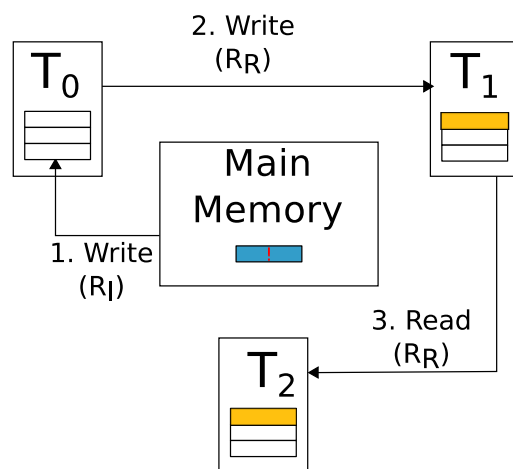
- **Example:**
  - $T_0 + T_1$  write CL
  - $T_2$  polls for completion





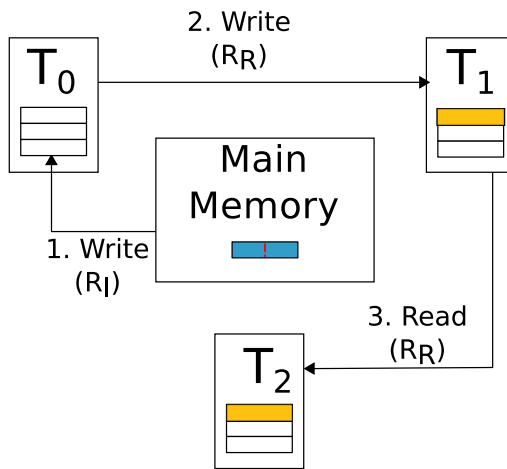
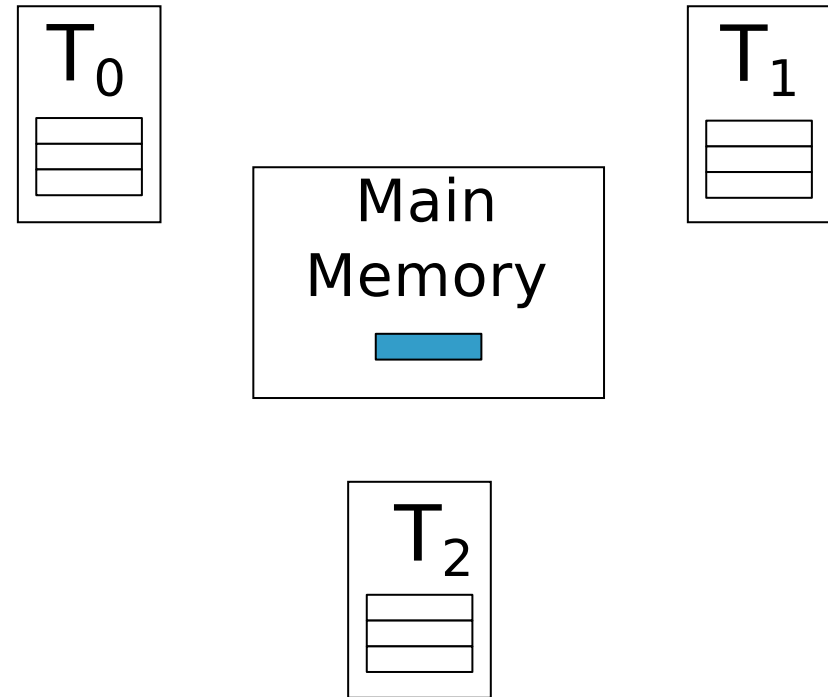
# Min-Max Modeling

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



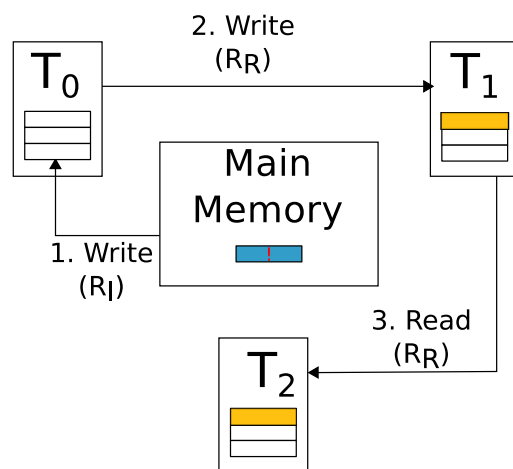
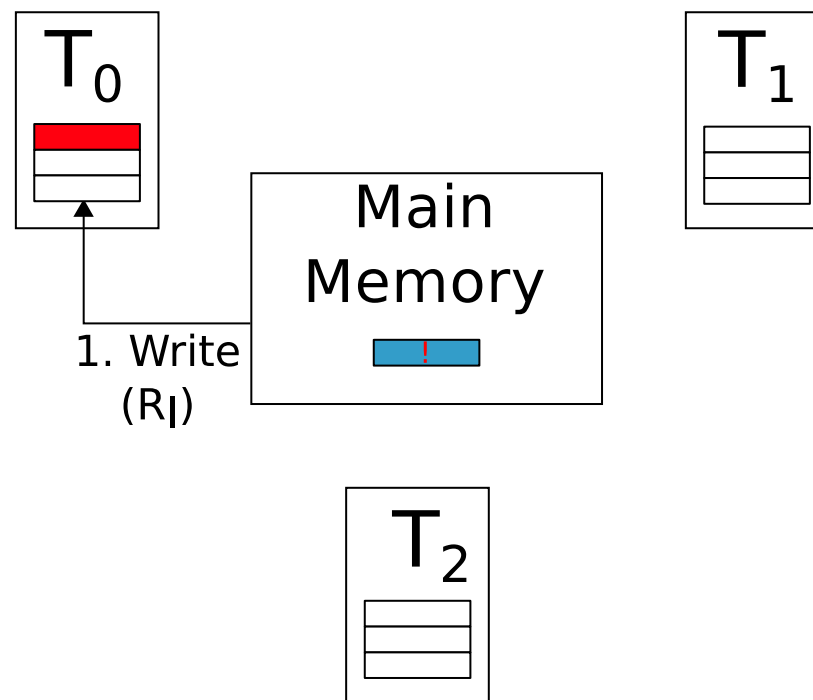
# Min-Max Modeling

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



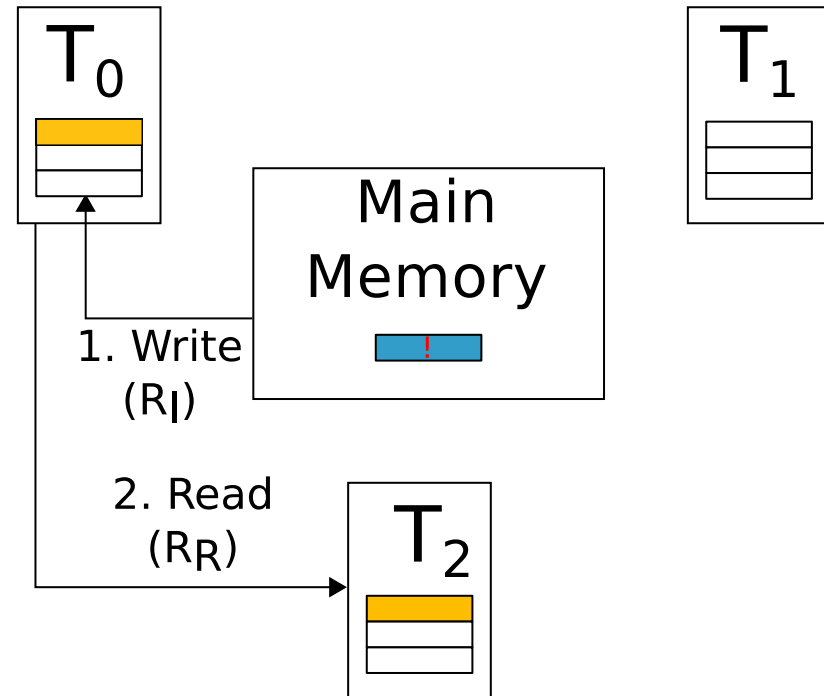
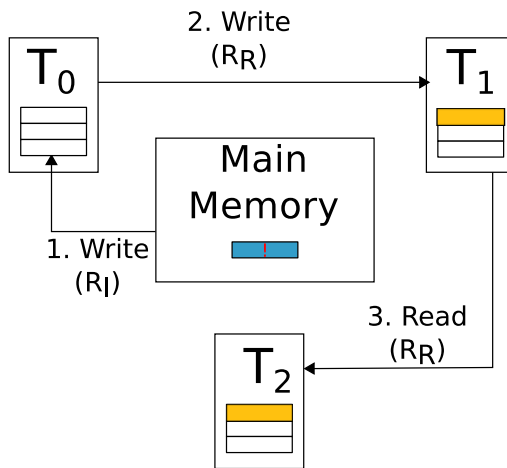
# Min-Max Modeling

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



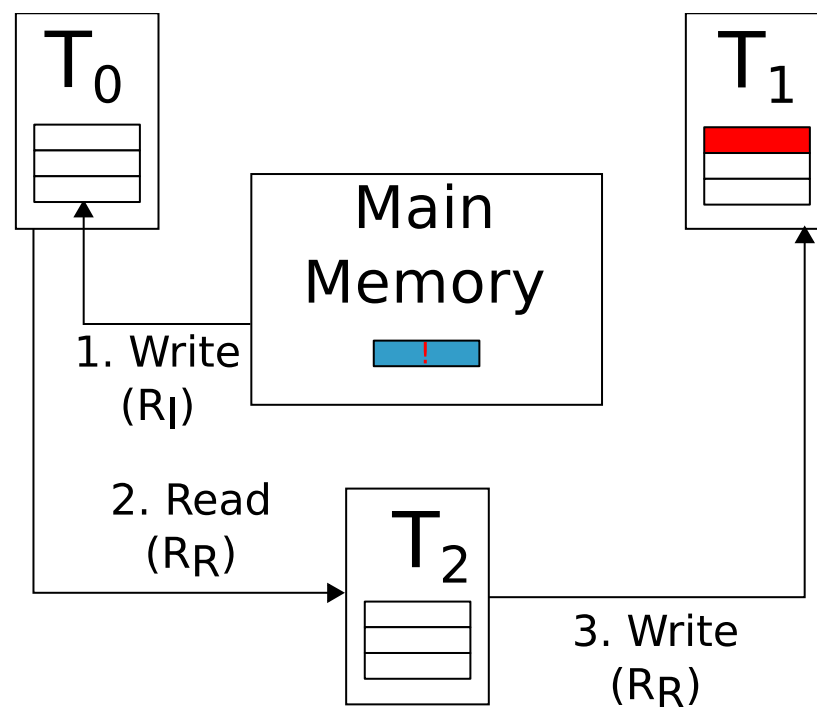
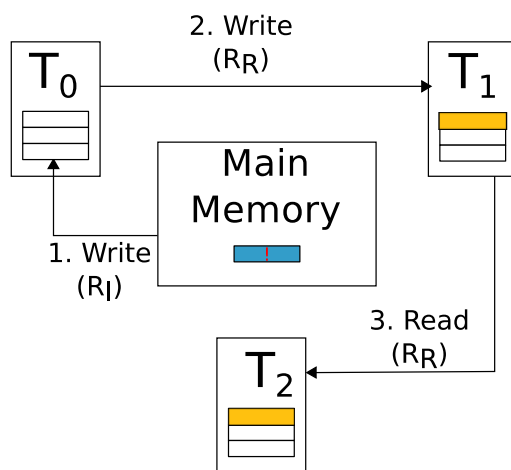
# Min-Max Modeling

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



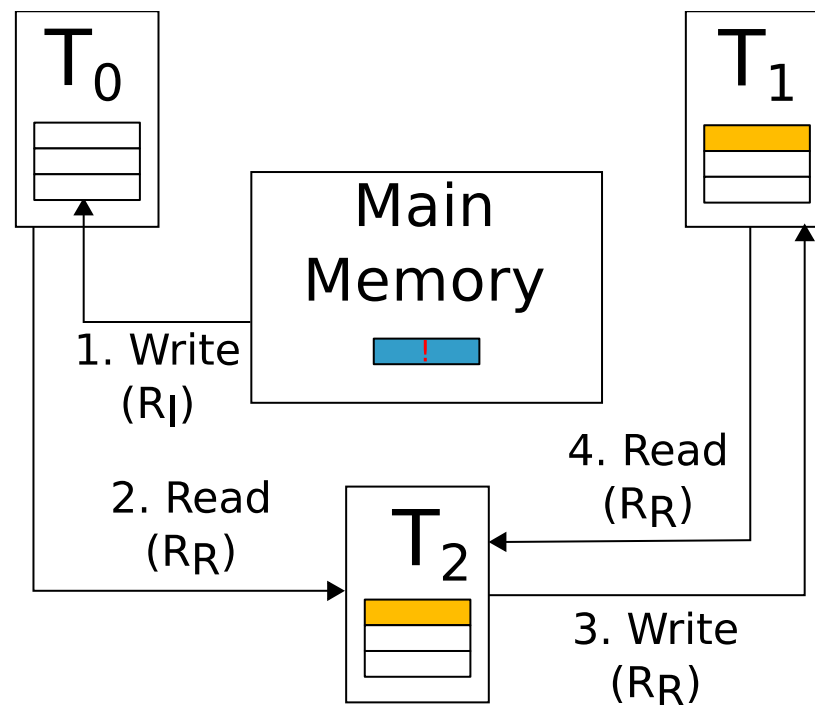
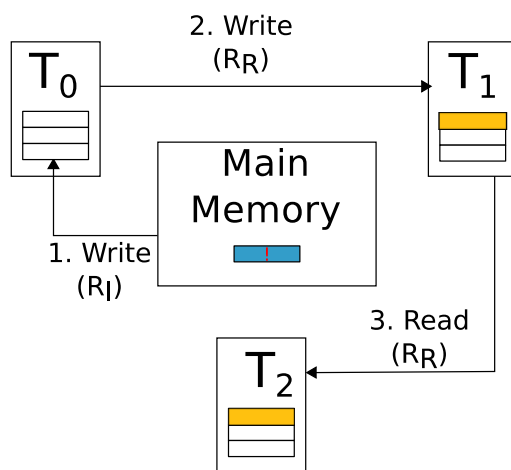
# Min-Max Modeling

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



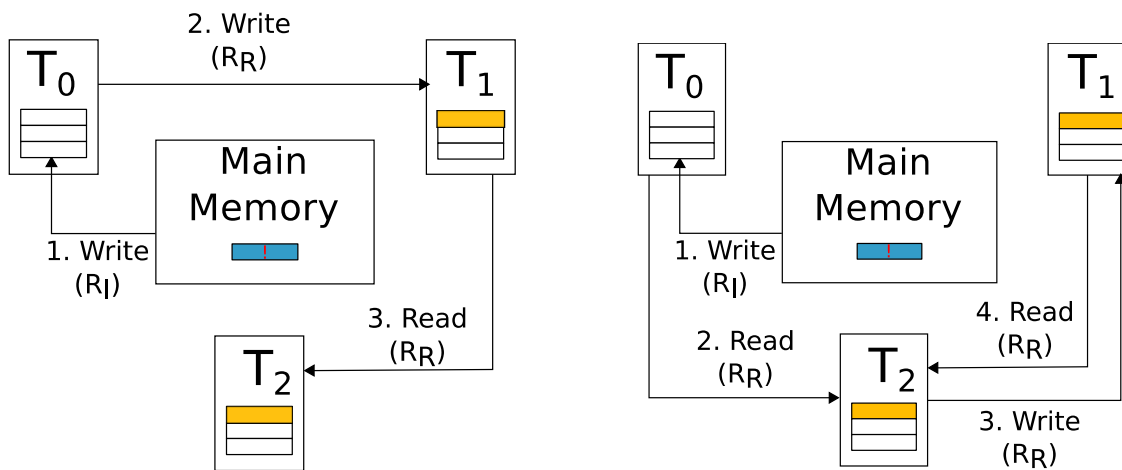
# Min-Max Modeling

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



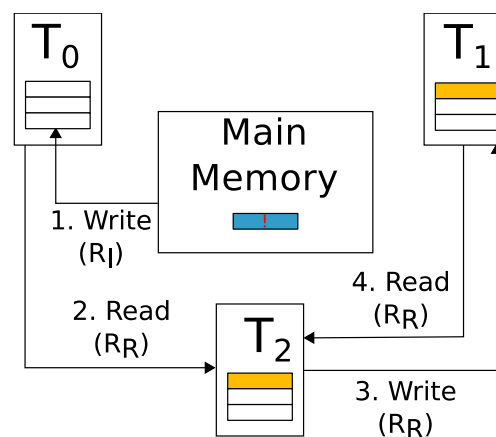
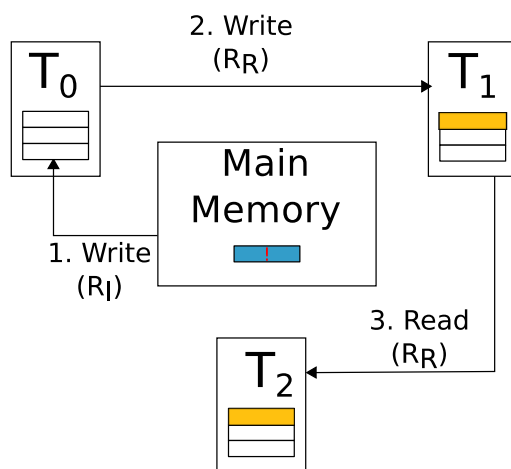
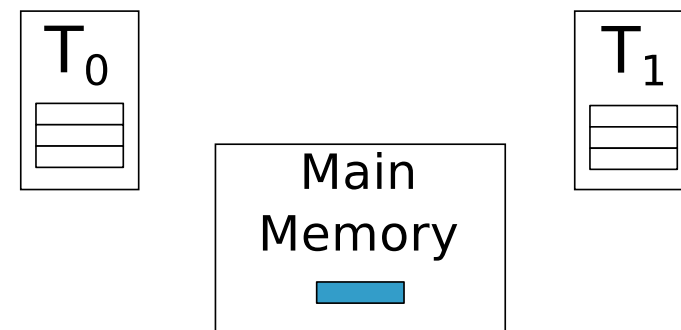
# Min-Max Modeling

- **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

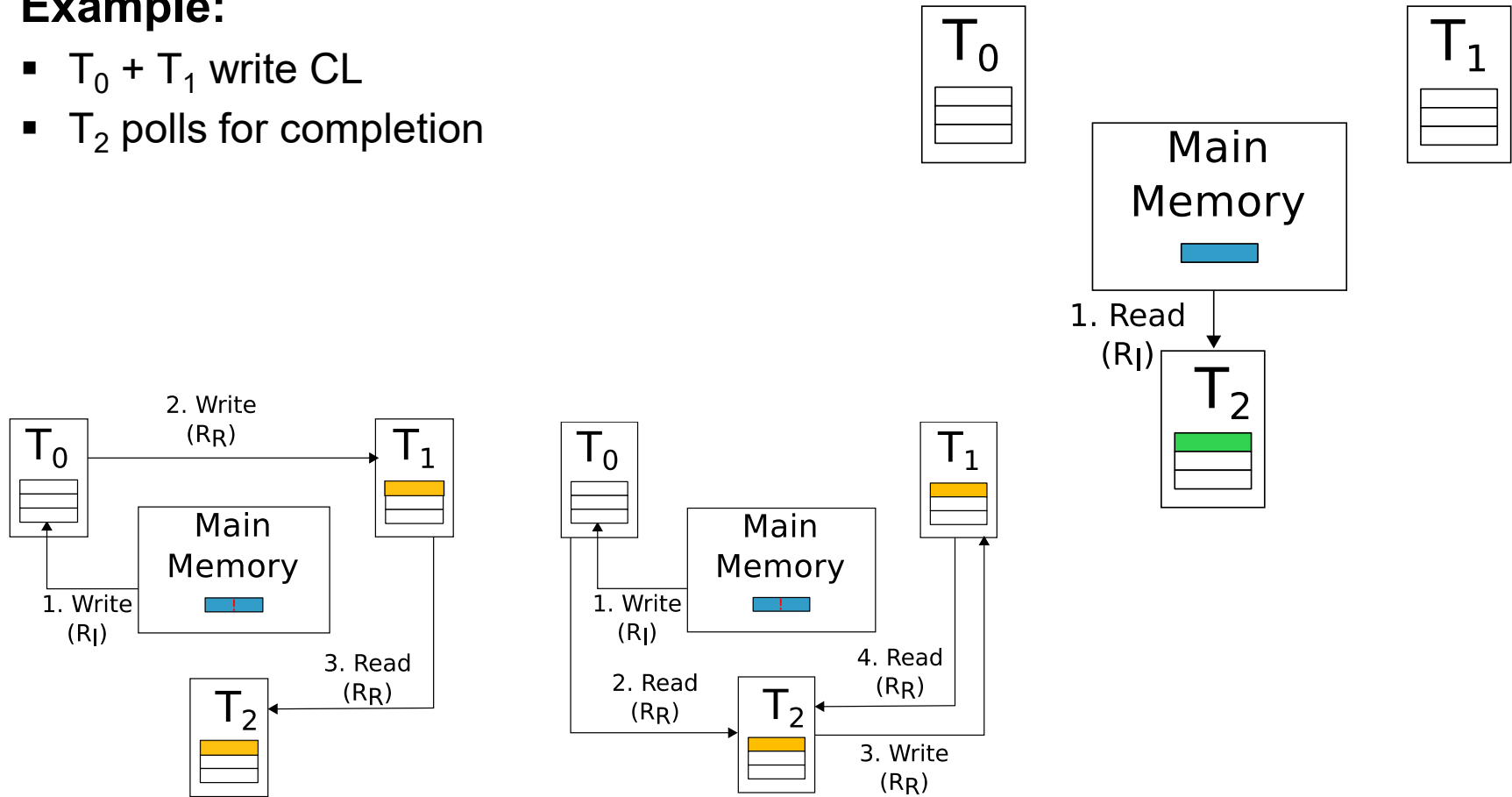




# Min-Max Modeling

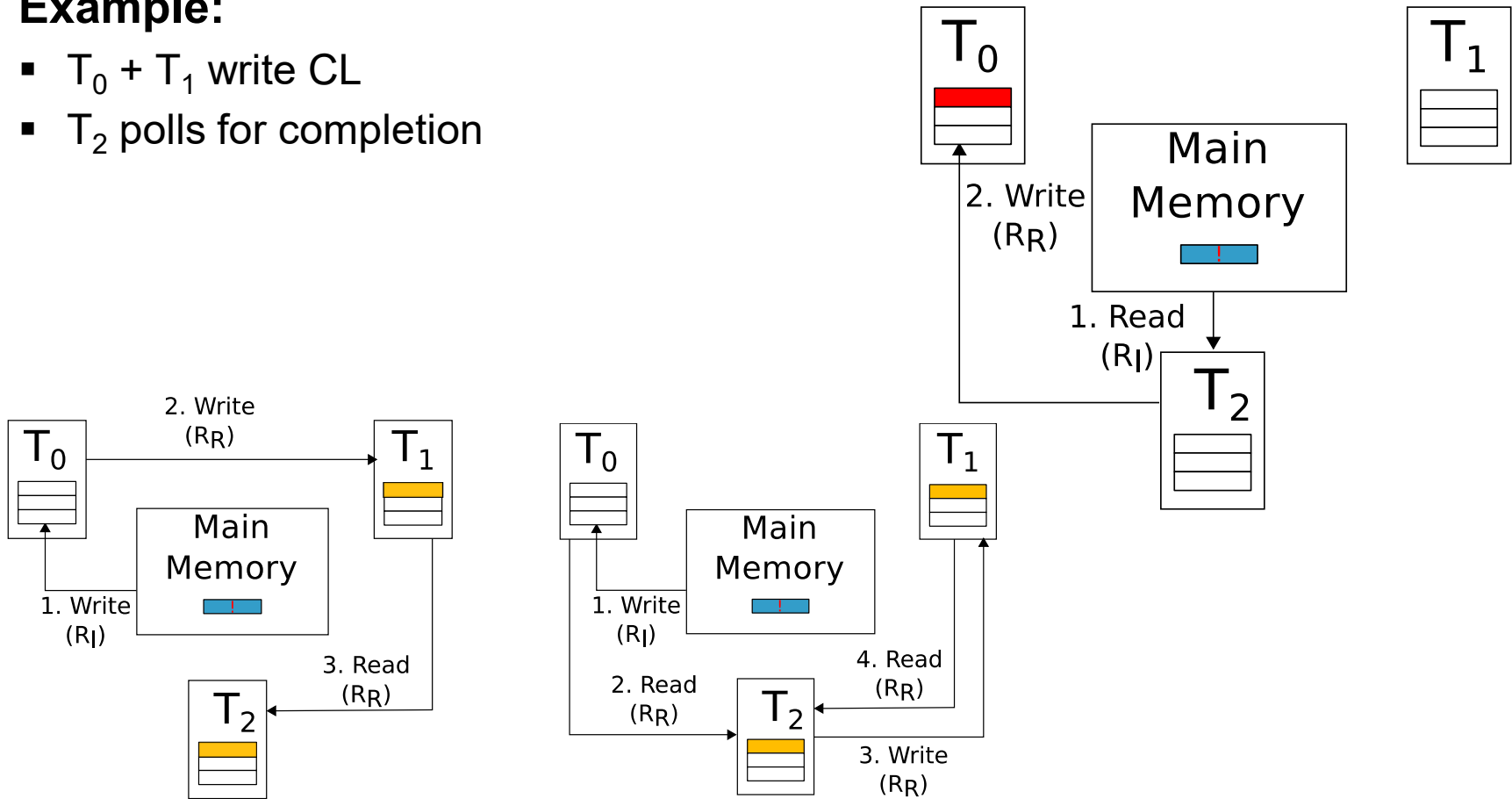
## Example:

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



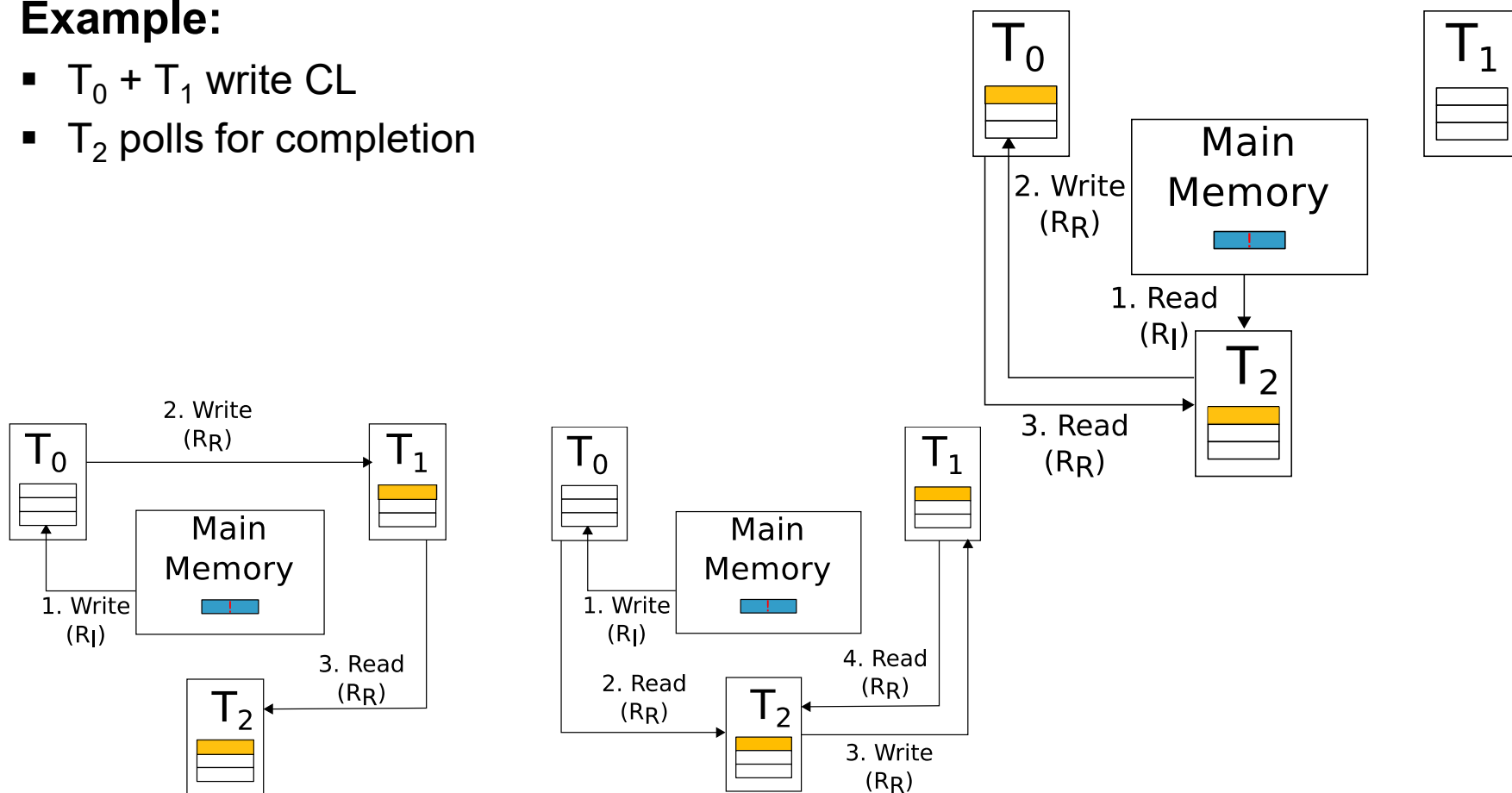
# Min-Max Modeling

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



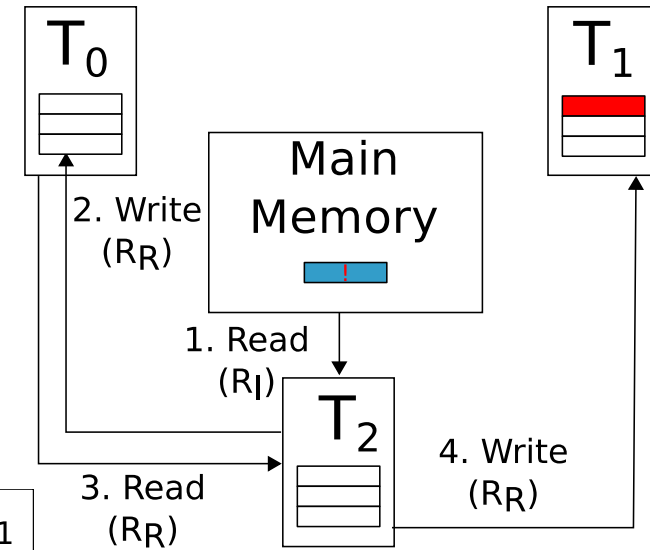
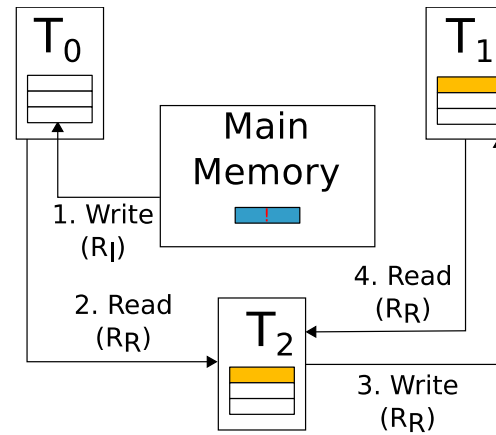
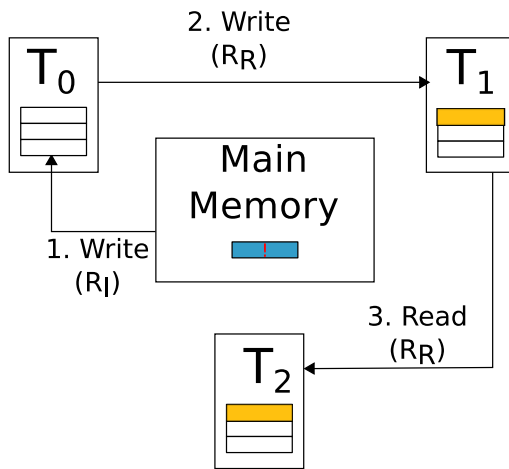
# Min-Max Modeling

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



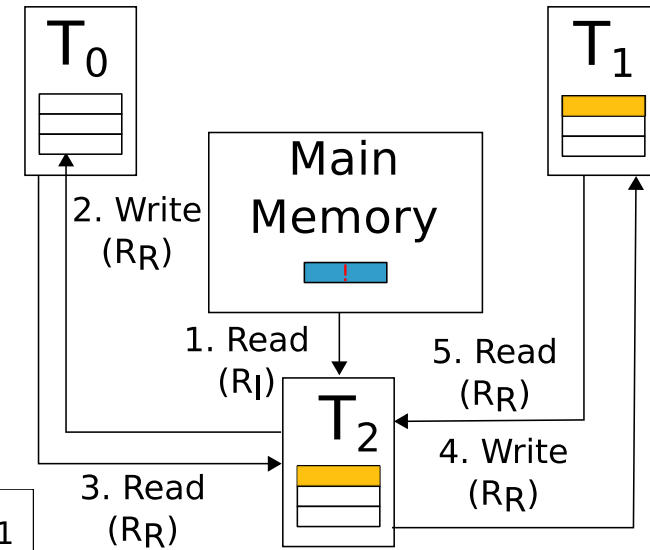
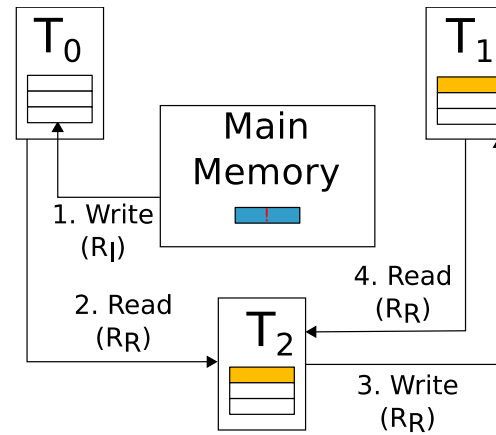
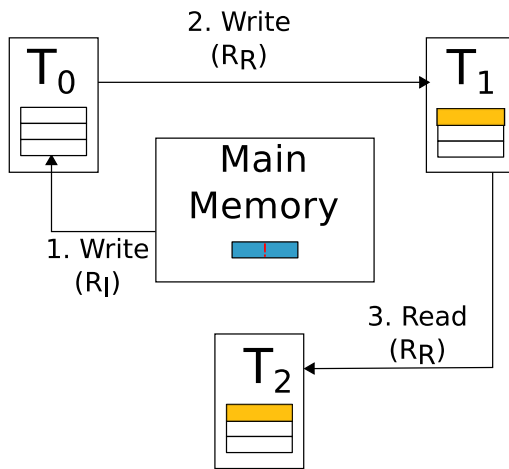
# Min-Max Modeling

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



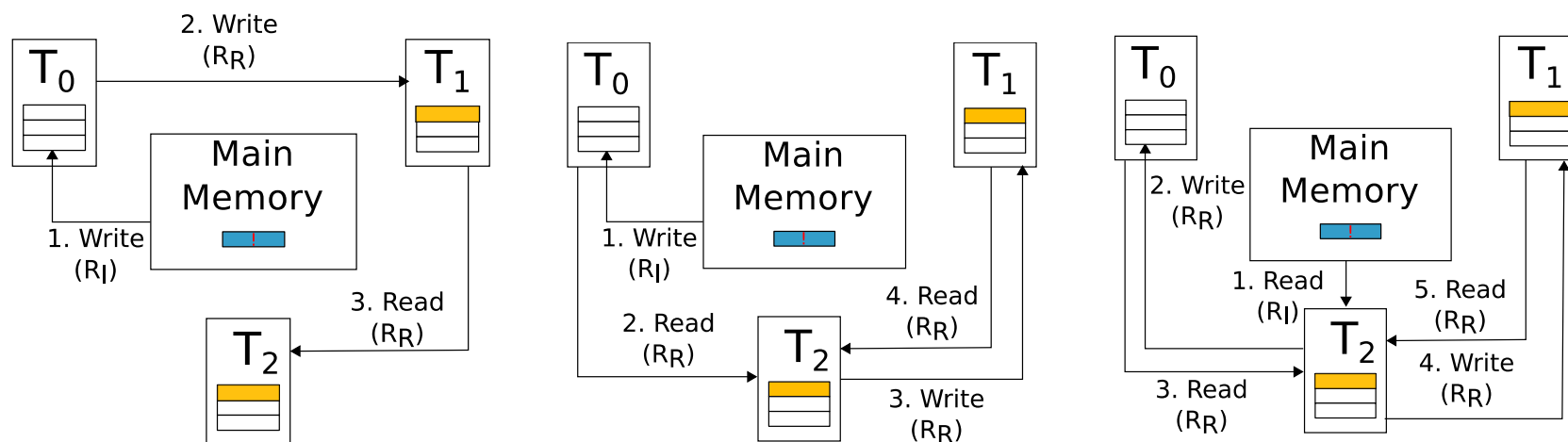
# Min-Max Modeling

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

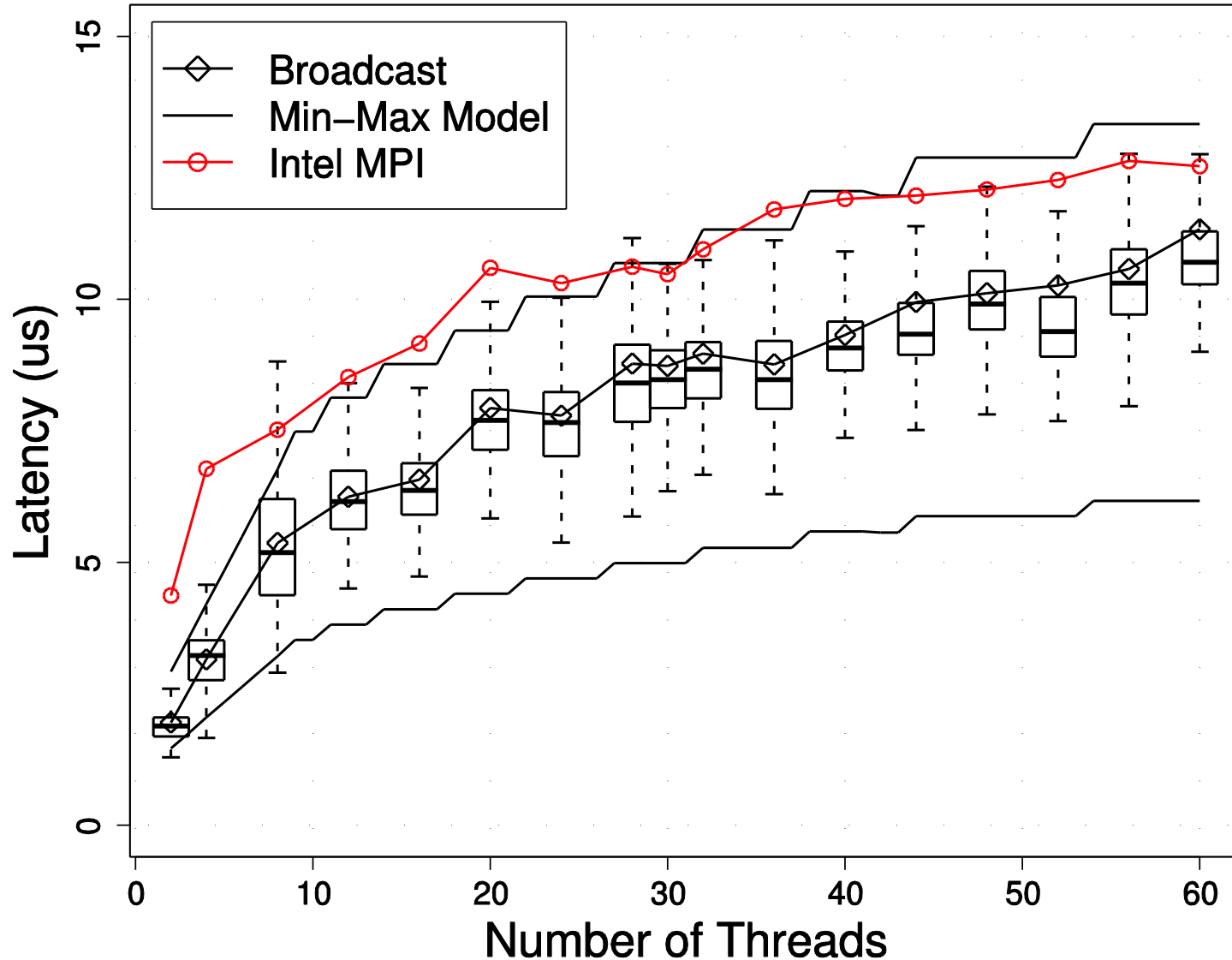


# Min-Max Modeling

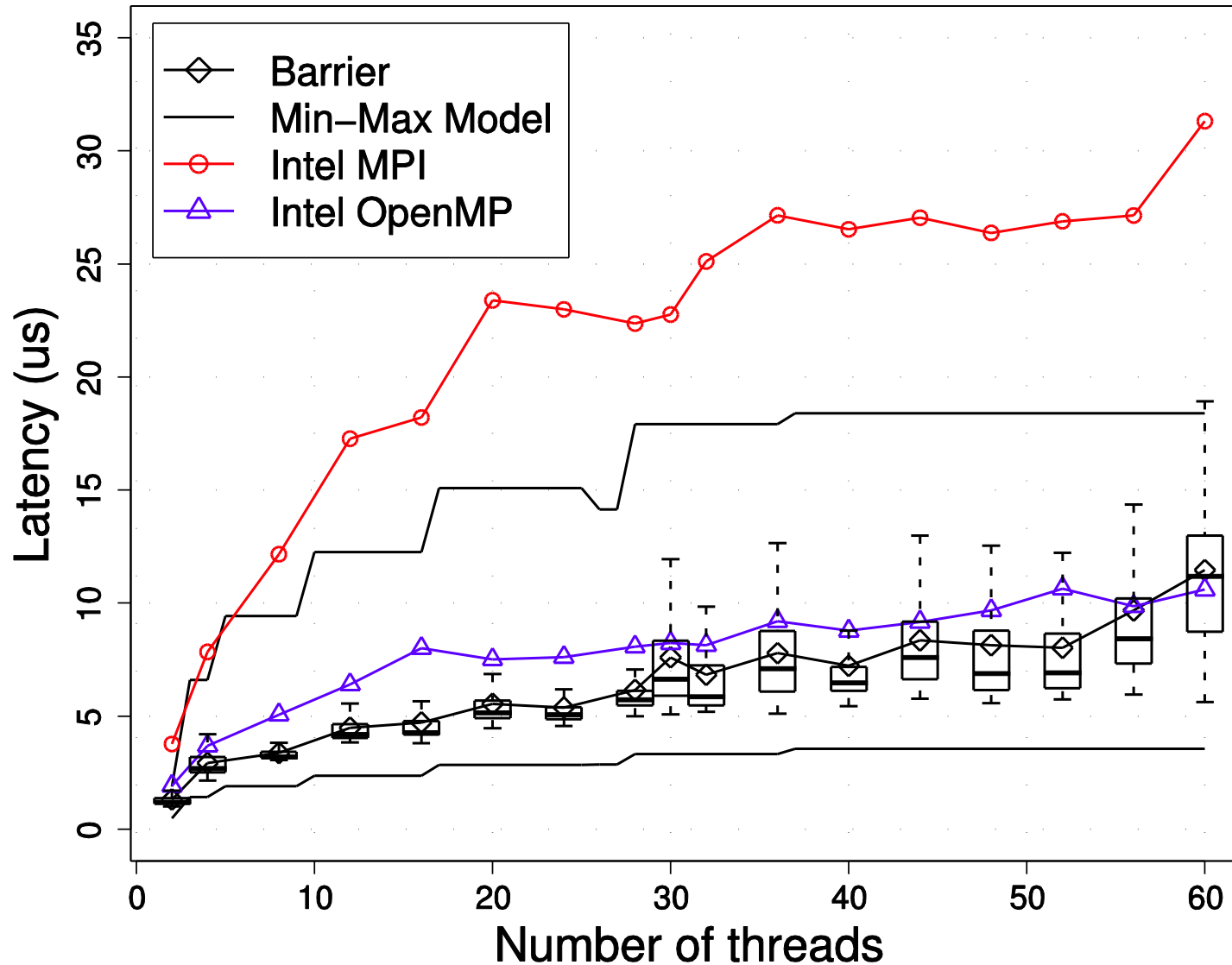
- **Example:**
  - $T_0 + T_1$  write CL
  - $T_2$  polls for completion



# Small Broadcast (8 Bytes)

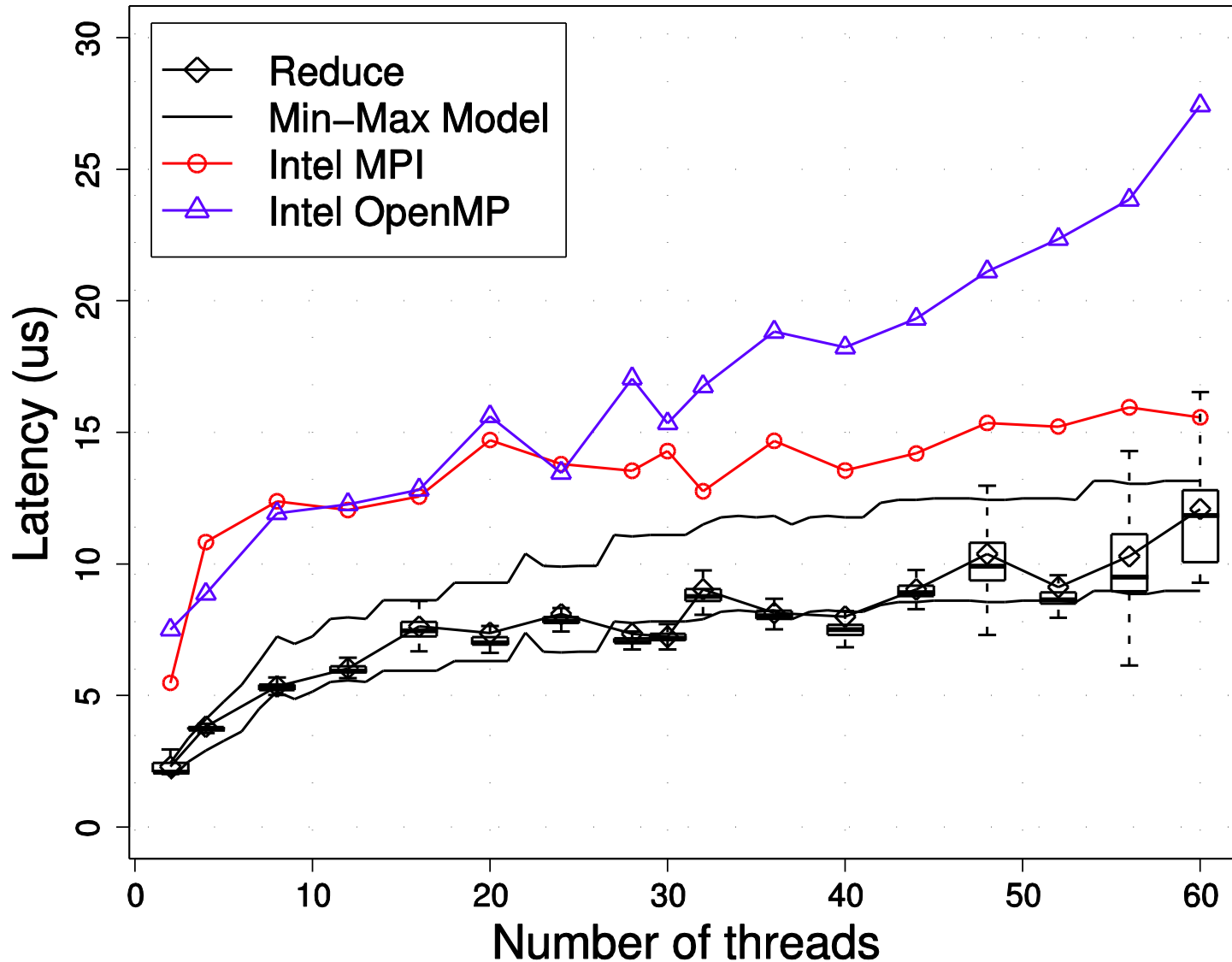


# Barrier



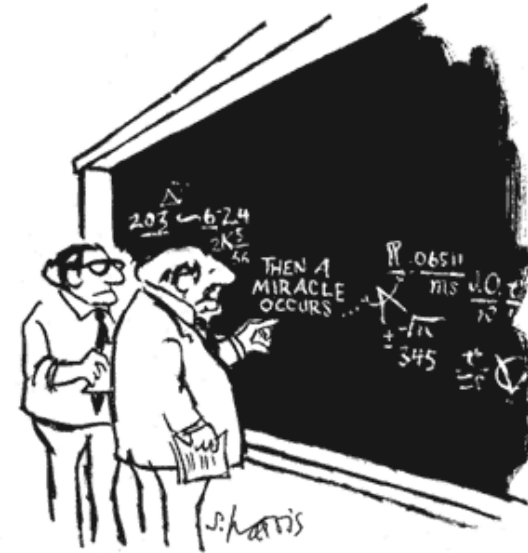


# Small Reduction



# 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 ☹**
  - Complicates modeling and is expensive
- **Obvious question: why do we need cache coherence?**
  - Answer: well, we don't, **if we program right!**



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

[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

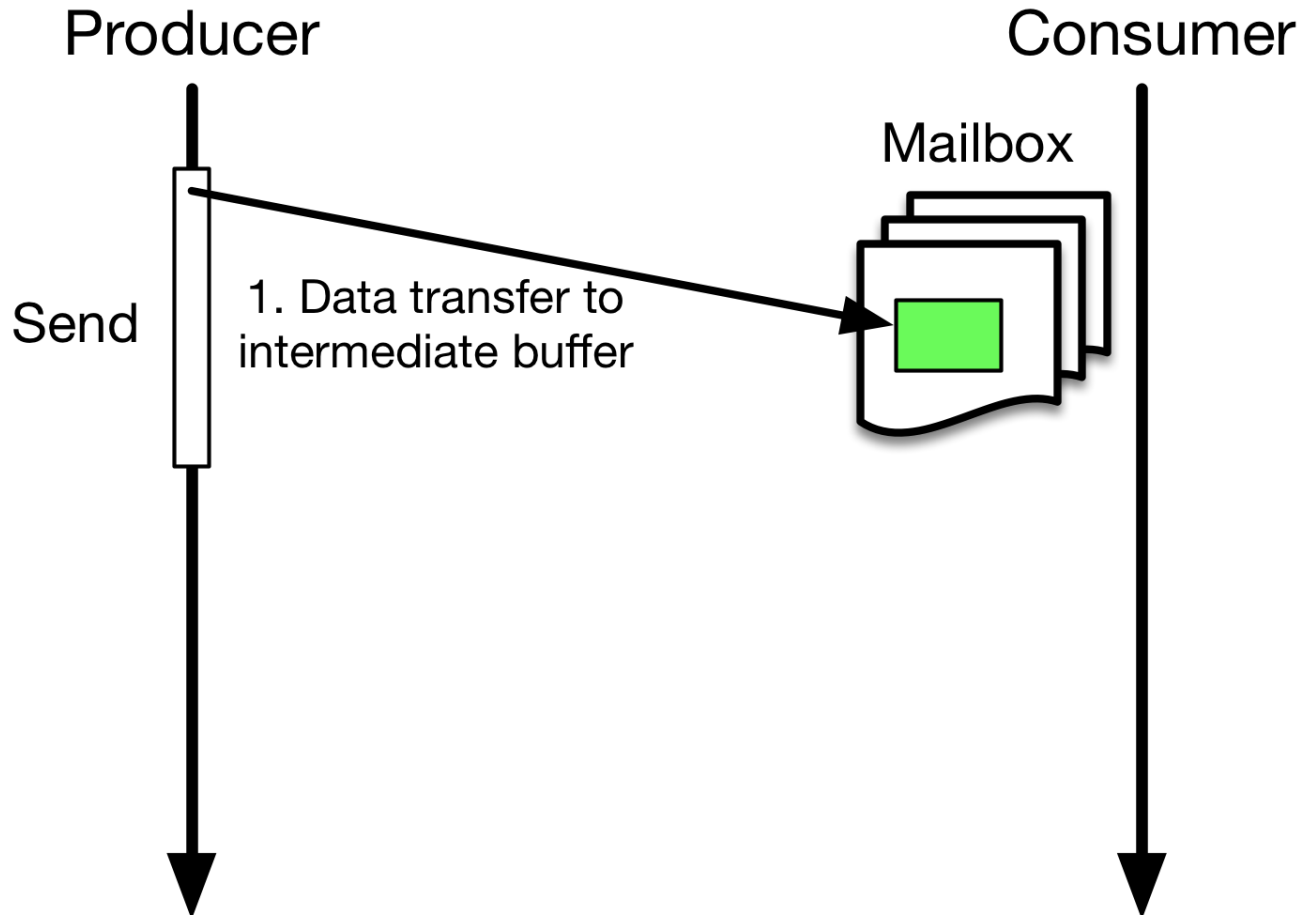
Producer



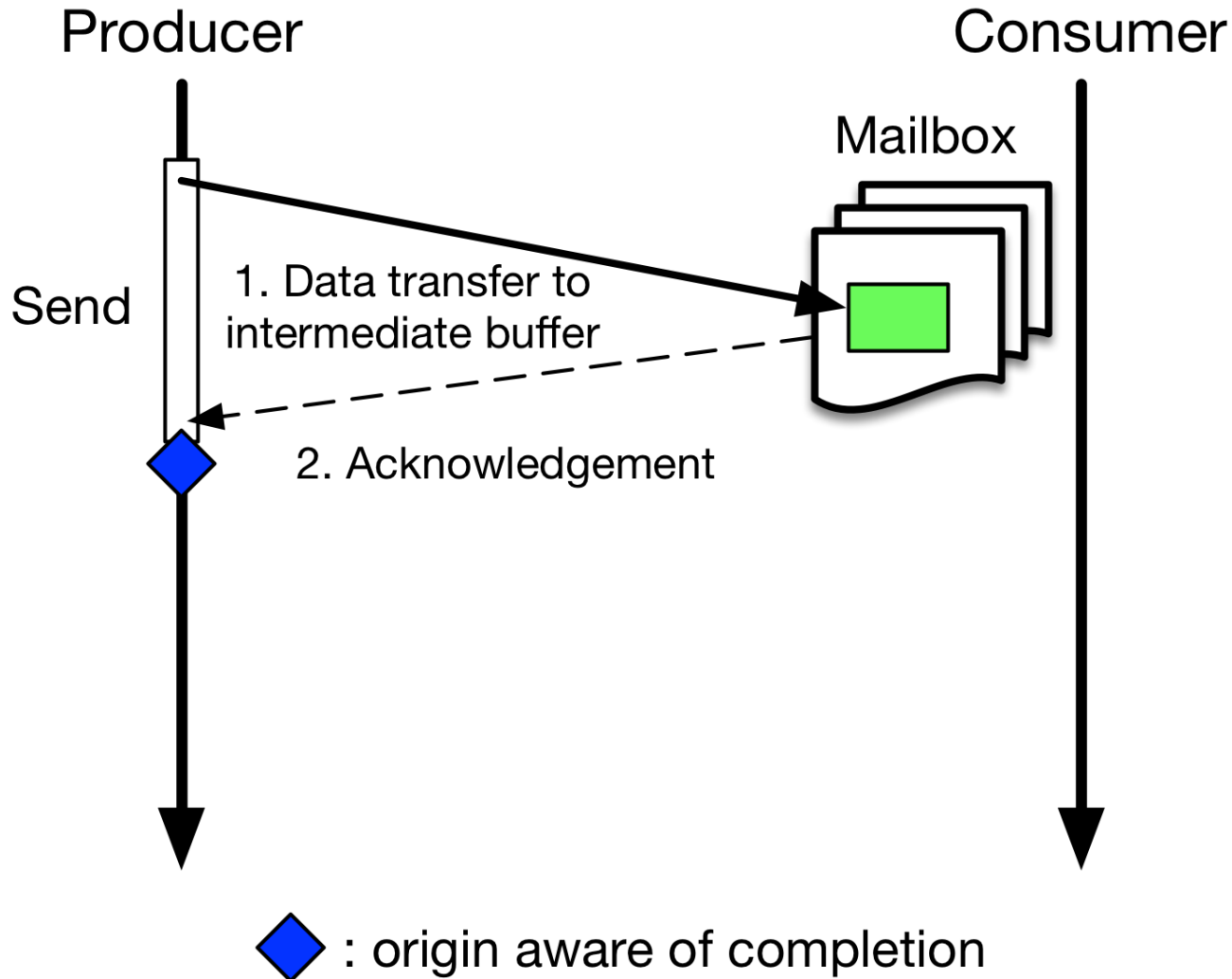
Consumer



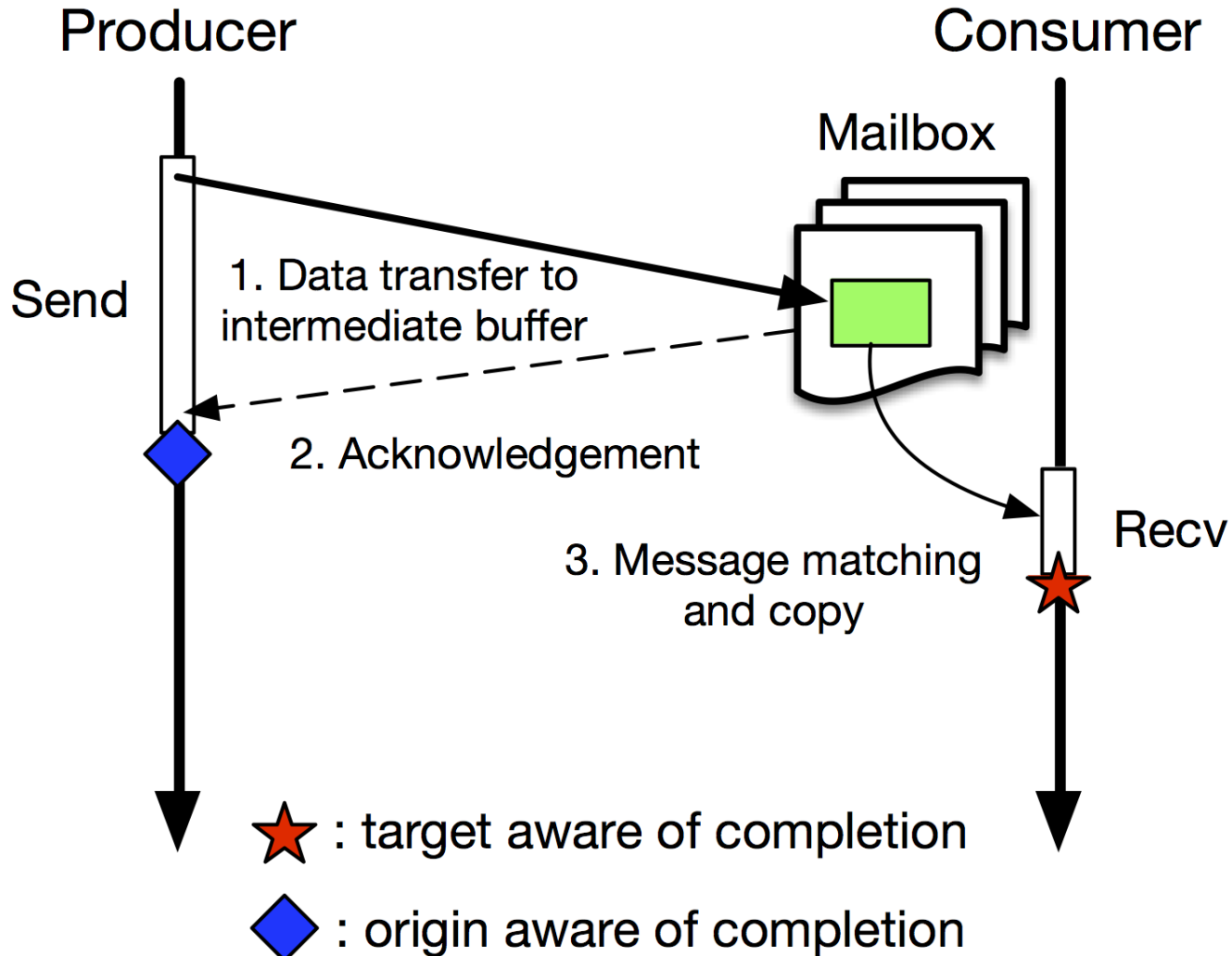
# MPI-1 MESSAGE PASSING – SIMPLE EAGER



# MPI-1 MESSAGE PASSING – SIMPLE EAGER



# MPI-1 MESSAGE PASSING – SIMPLE EAGER



**Critical path: 1 latency + 1 copy**

# MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS

Producer

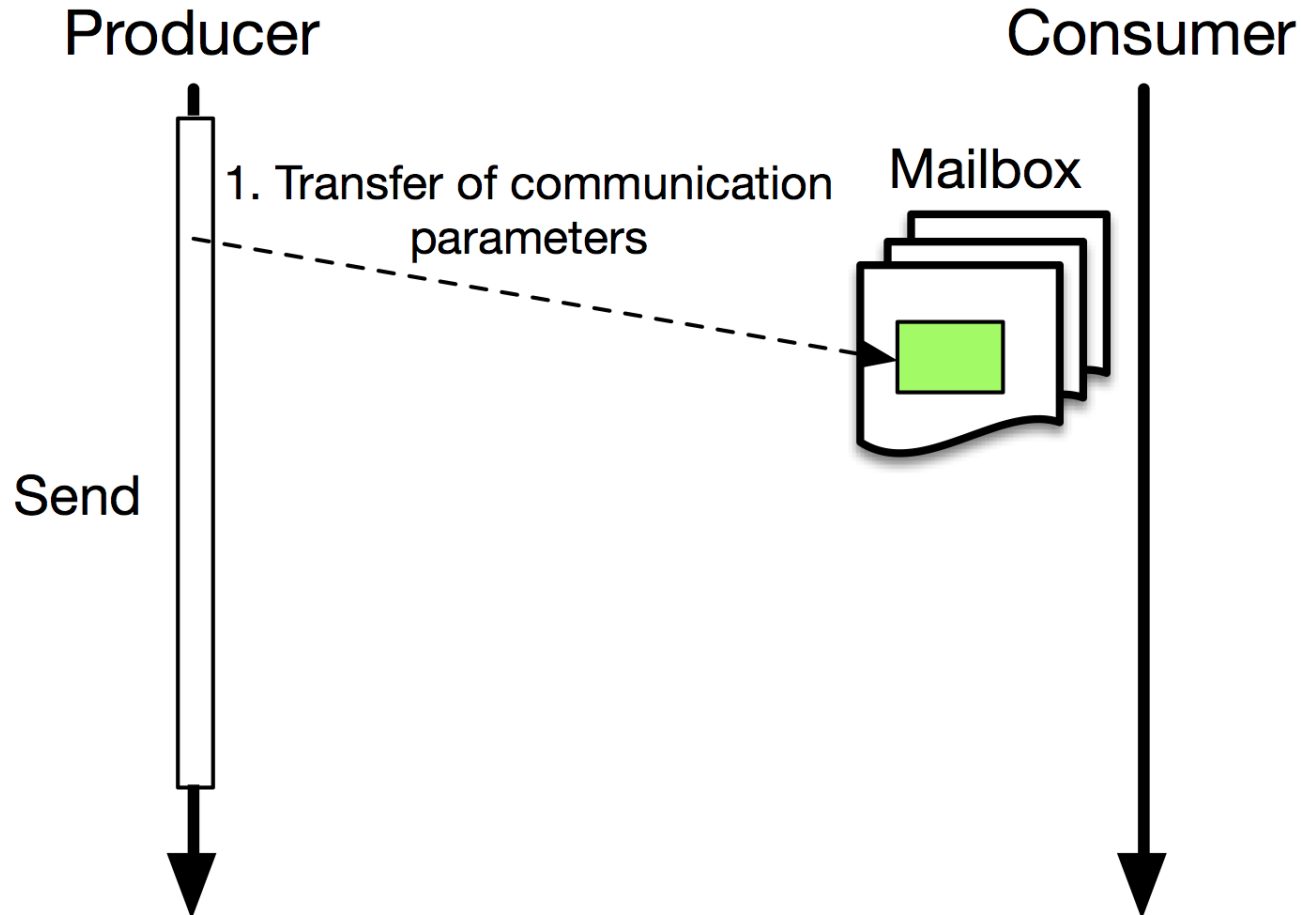


Consumer

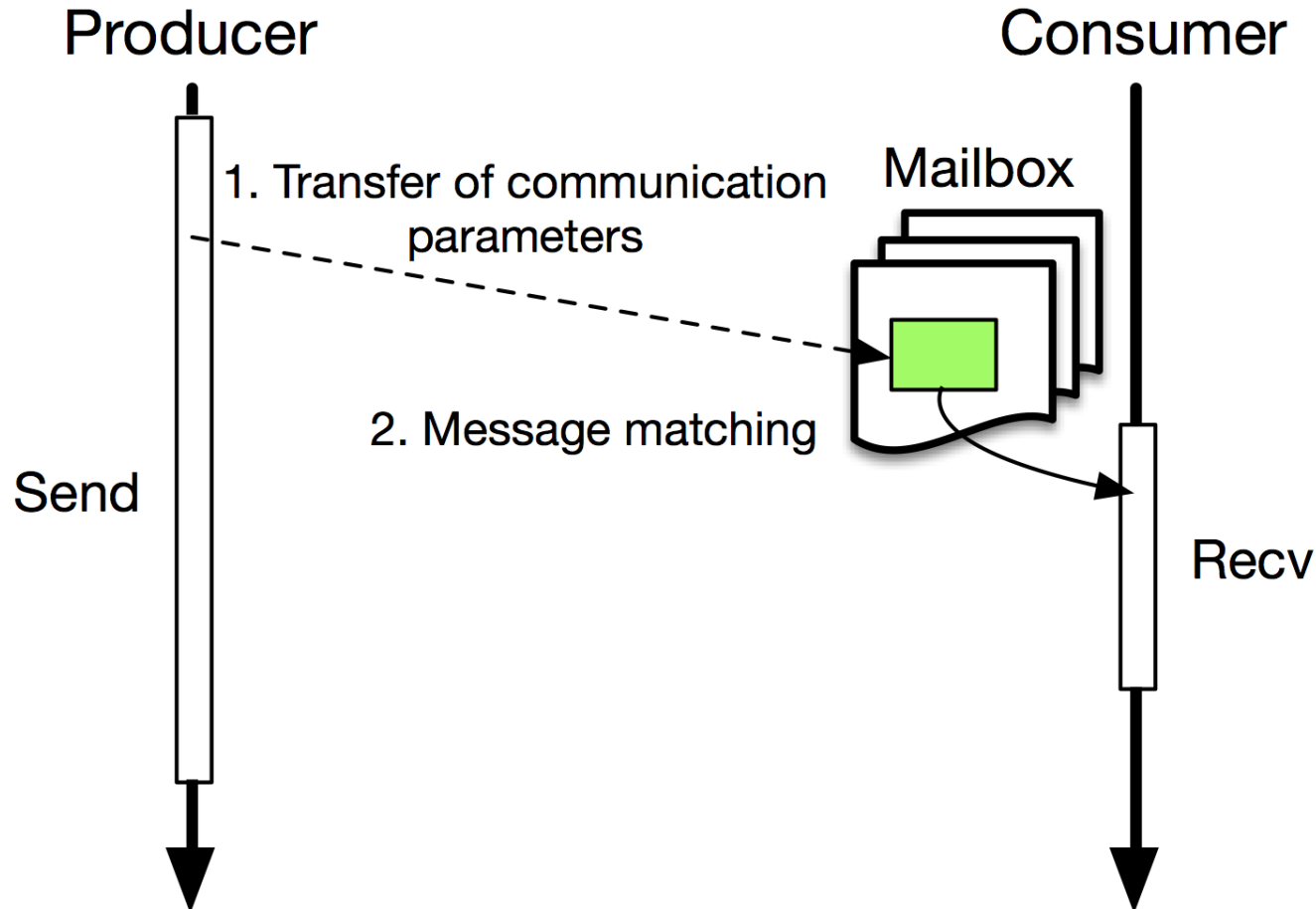




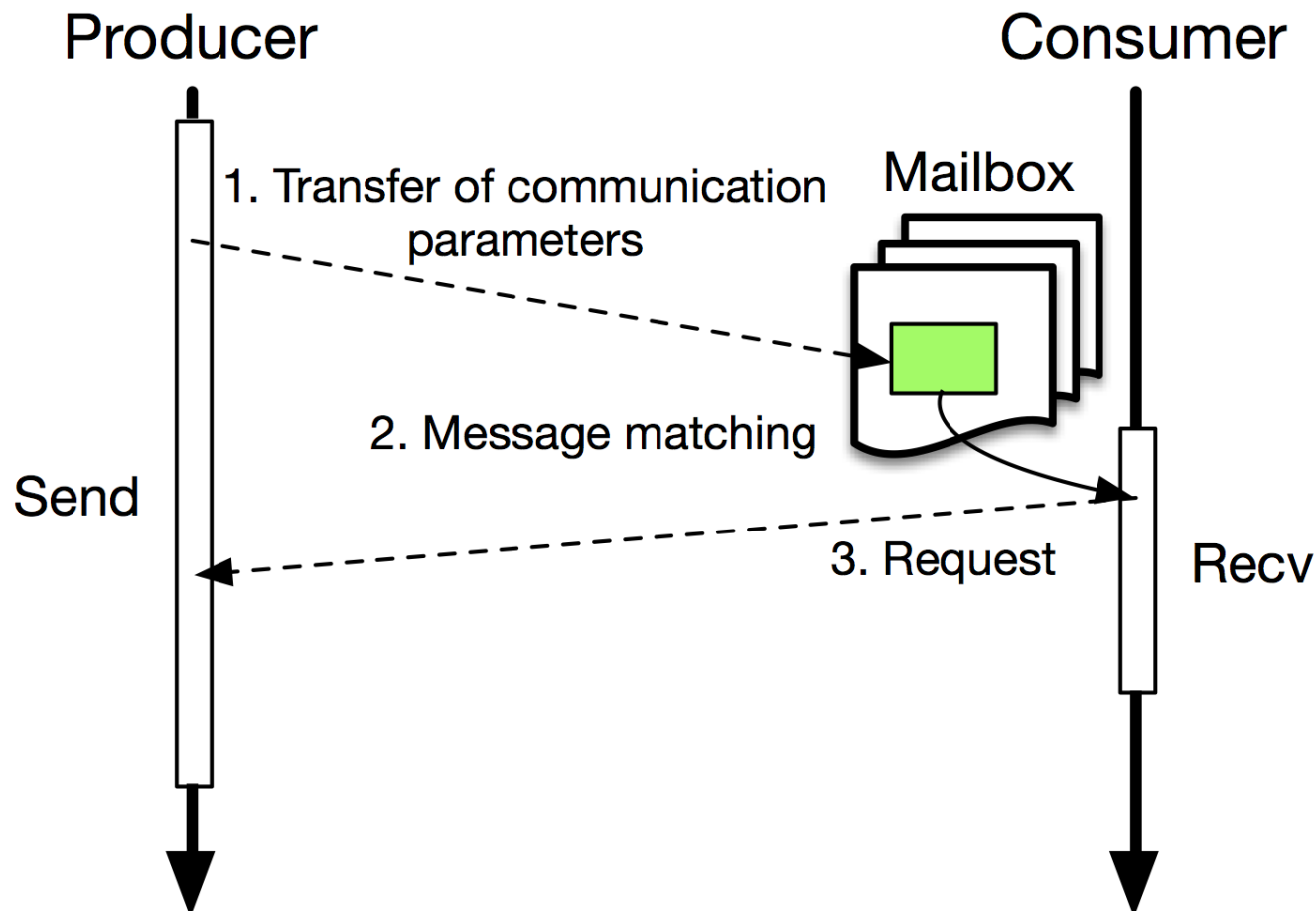
# MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS



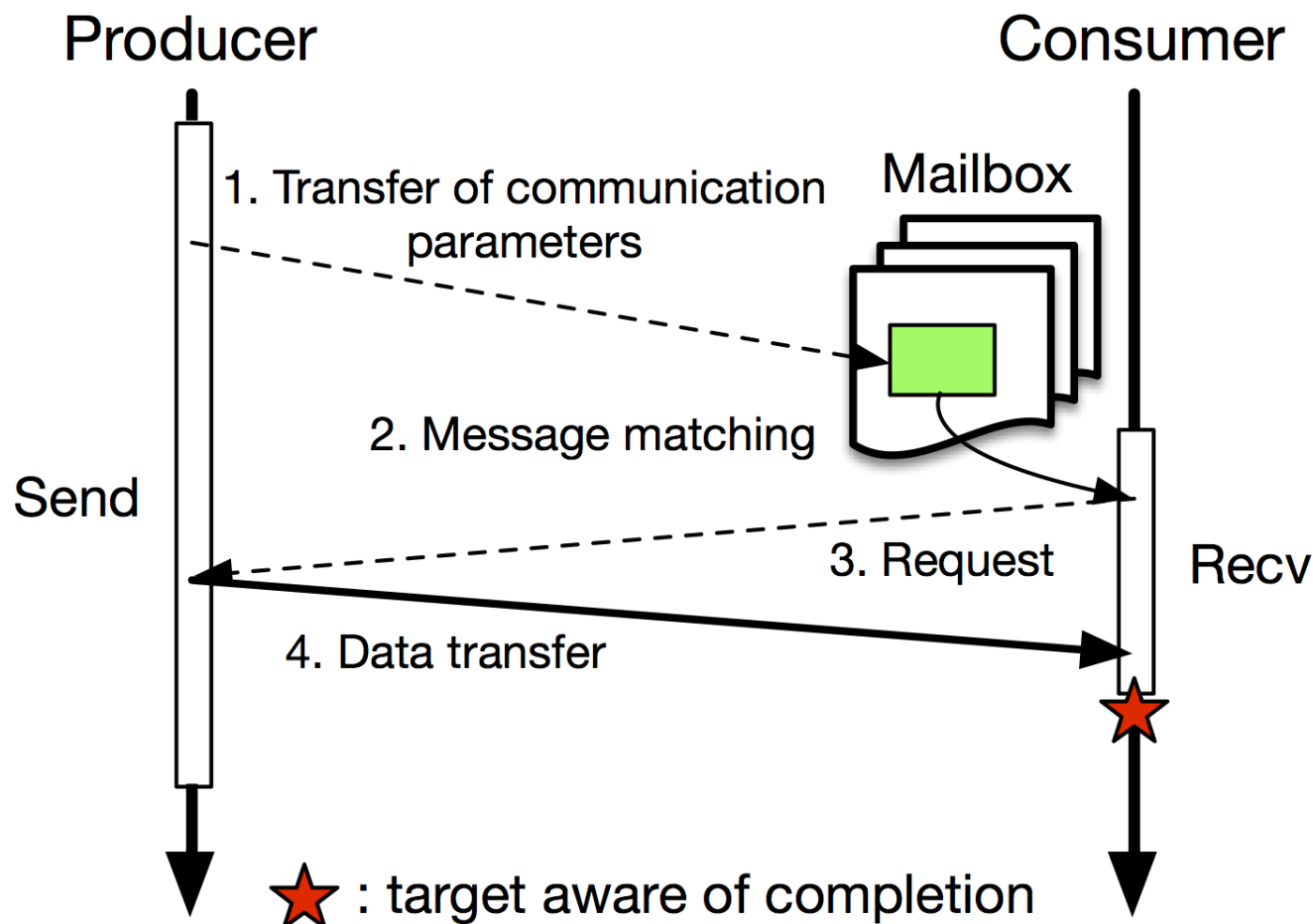
# MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS



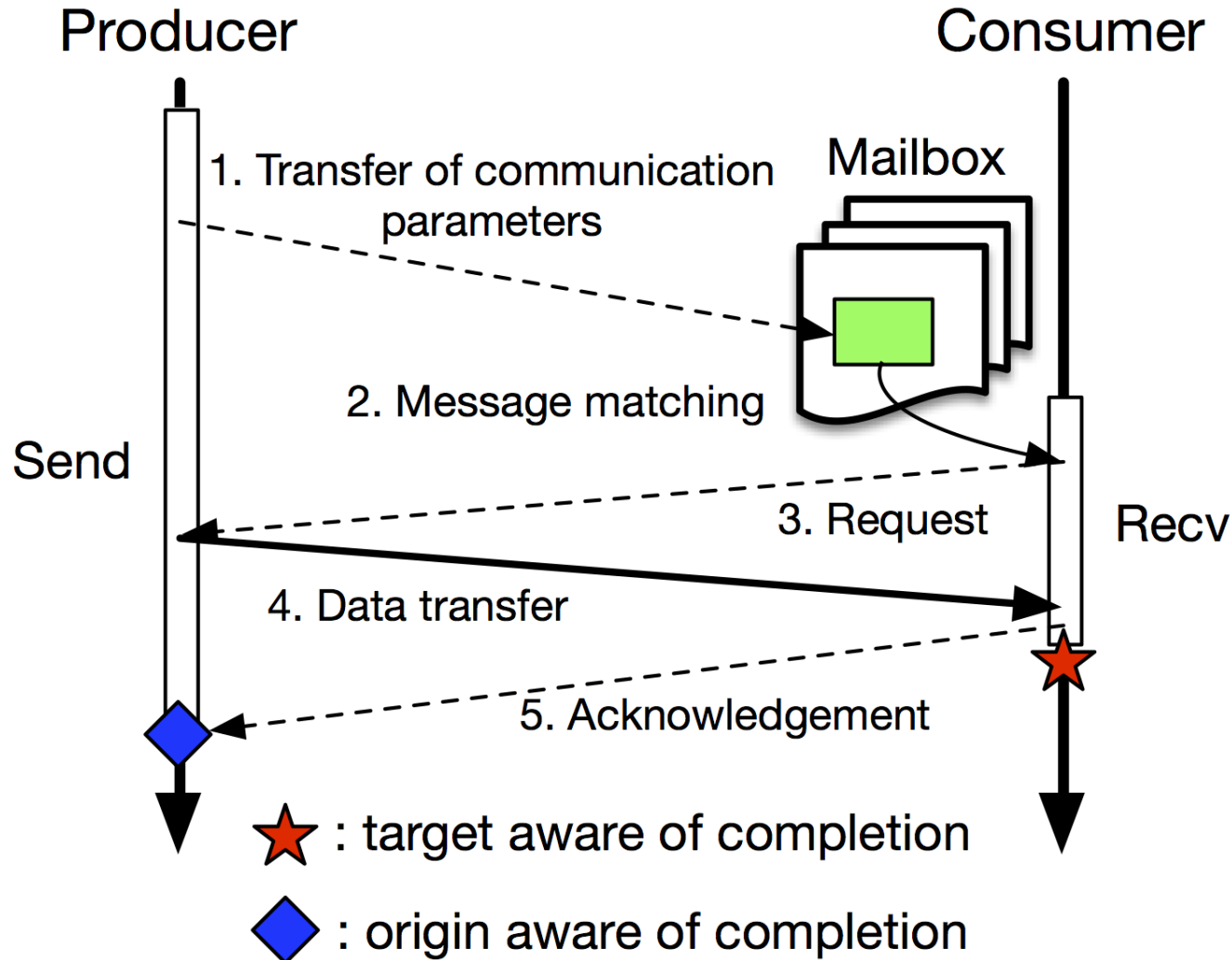
# MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS



# MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS



# MPI-1 MESSAGE PASSING – SIMPLE RENDEZVOUS



**Critical path: 3 latencies**

# COMMUNICATION IN TODAY'S

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.

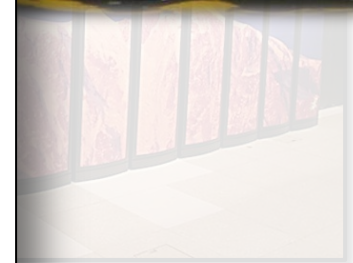
It turned out that VIA, based on RDMA (Remote Direct Memory Access, or Remote DMA), was not an improvement on existing APIs to support widely used application-software interfaces such as MPI and Sockets. After a while, VIA faded away, overtaken by other developments.

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



# 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



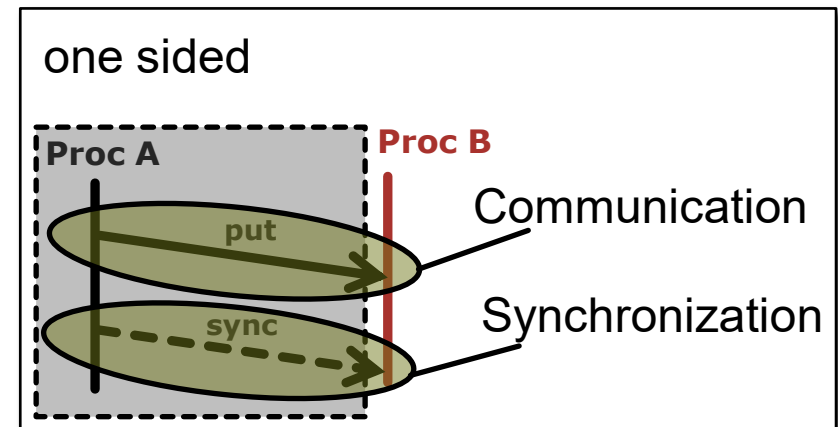
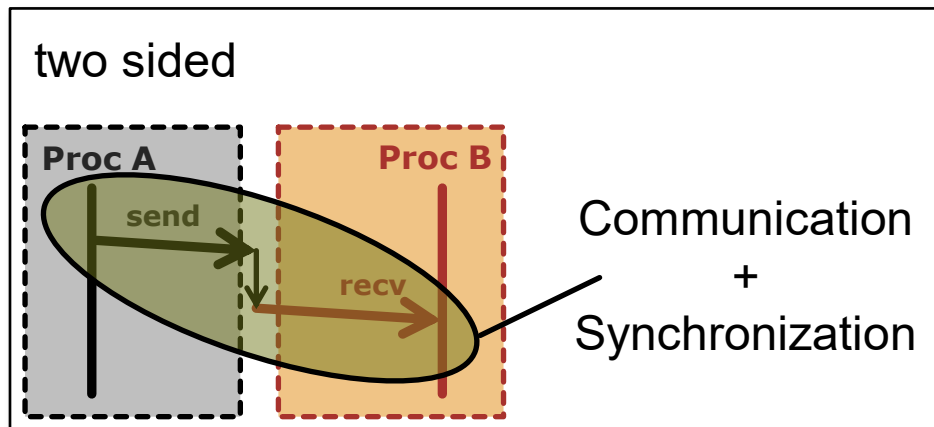
“Traditionally, HPC programming models are following hardware developments” (IPDPS’15)



# MPI-3 RMA SUMMARY

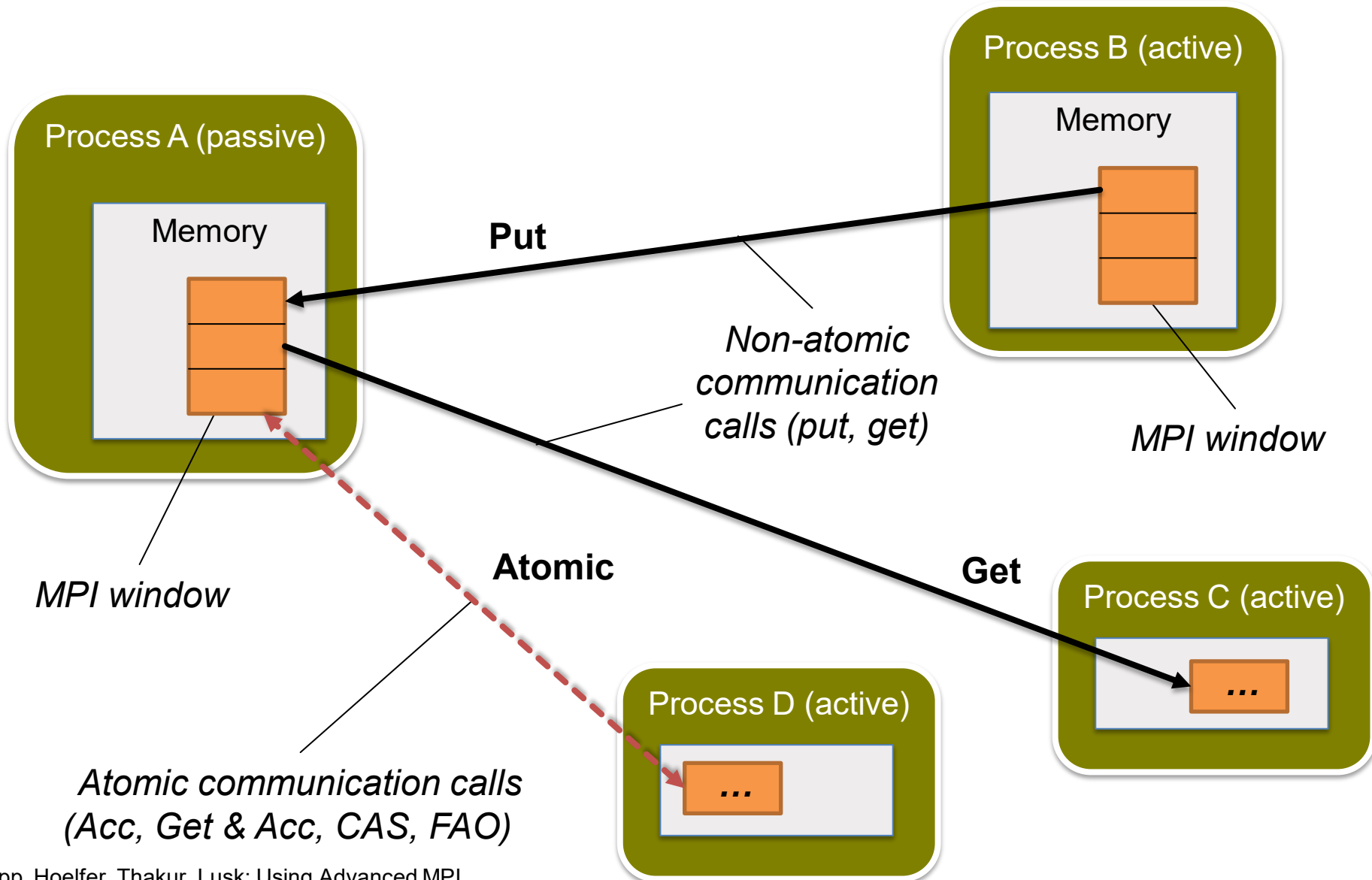


- **MPI-3 updates RMA (“MPI One Sided”)**
  - Significant change from MPI-2
- **Communication is „one sided” (no involvement of destination)**
  - Utilize direct memory access
- **RMA decouples communication & synchronization**
  - Fundamentally different from message passing

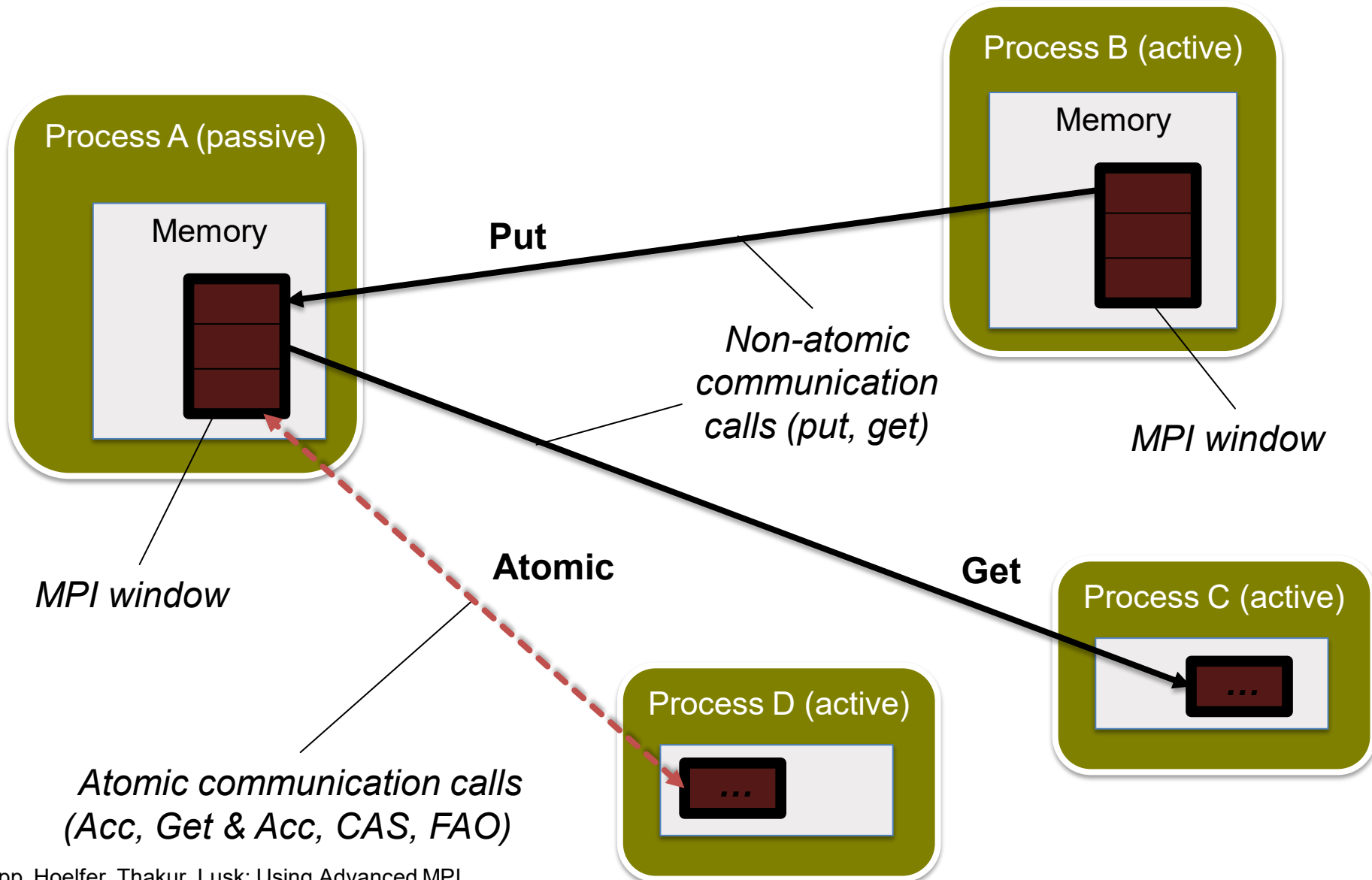




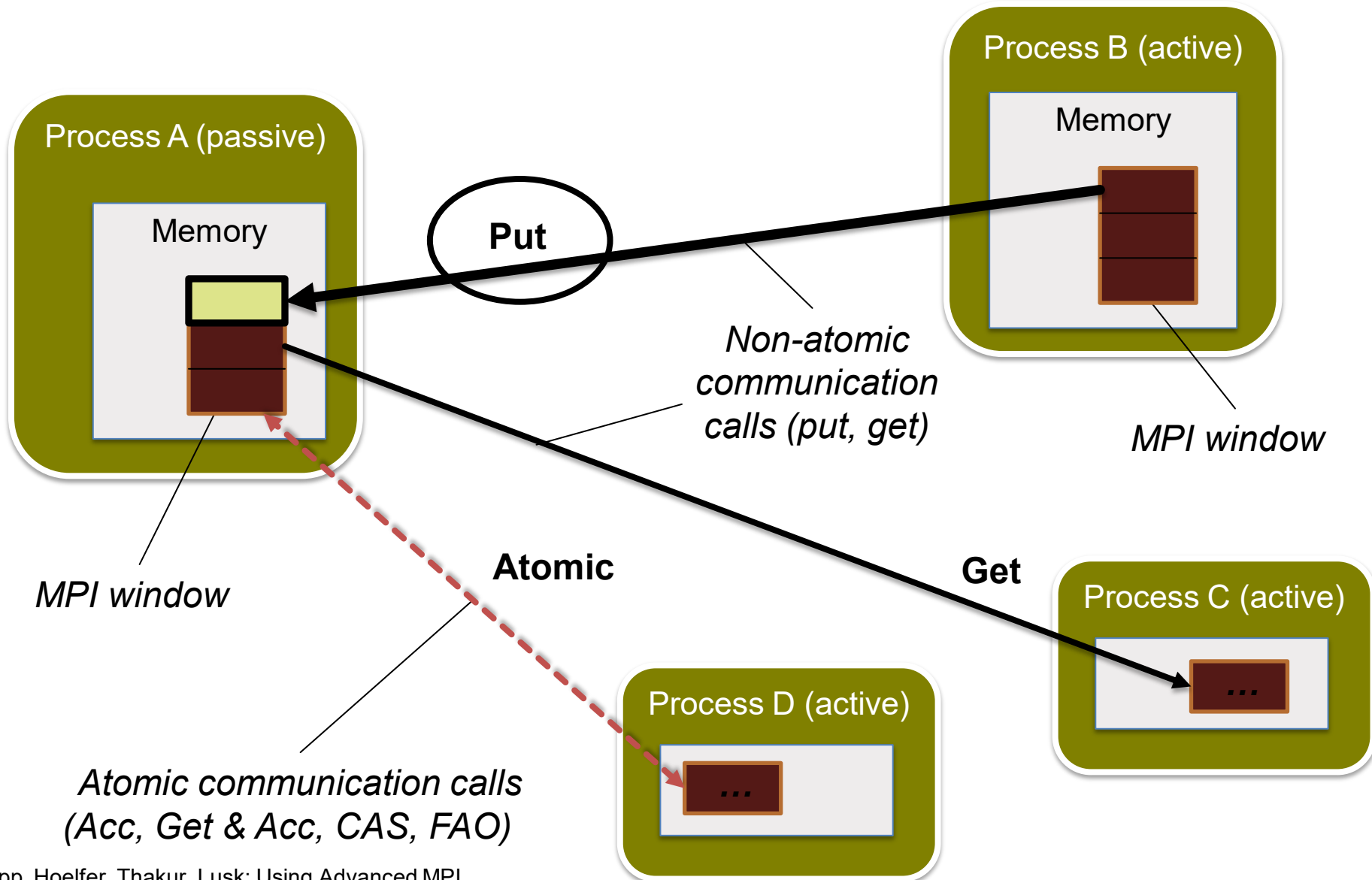
# MPI-3 RMA COMMUNICATION OVERVIEW



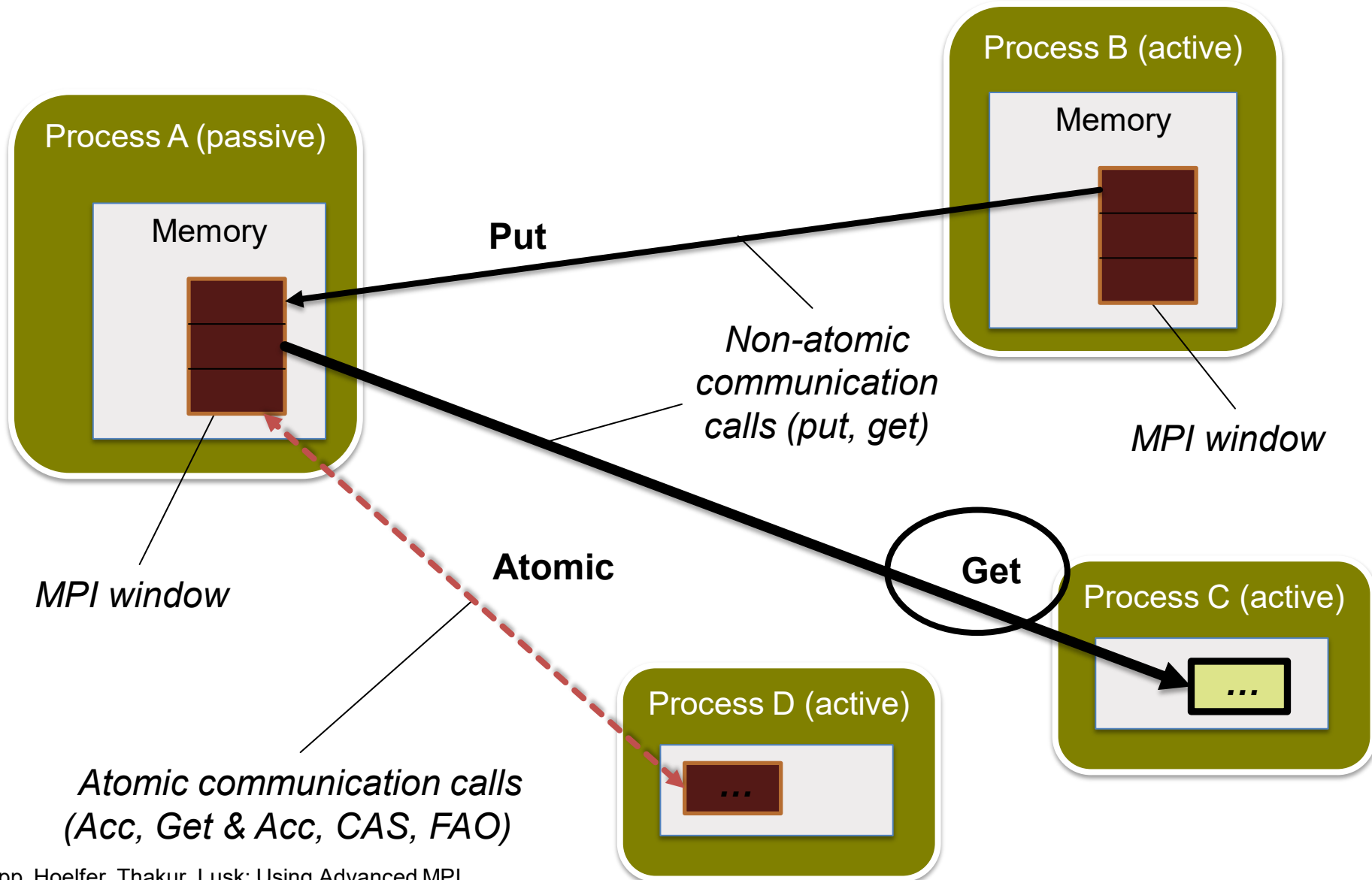
# MPI-3 RMA COMMUNICATION OVERVIEW



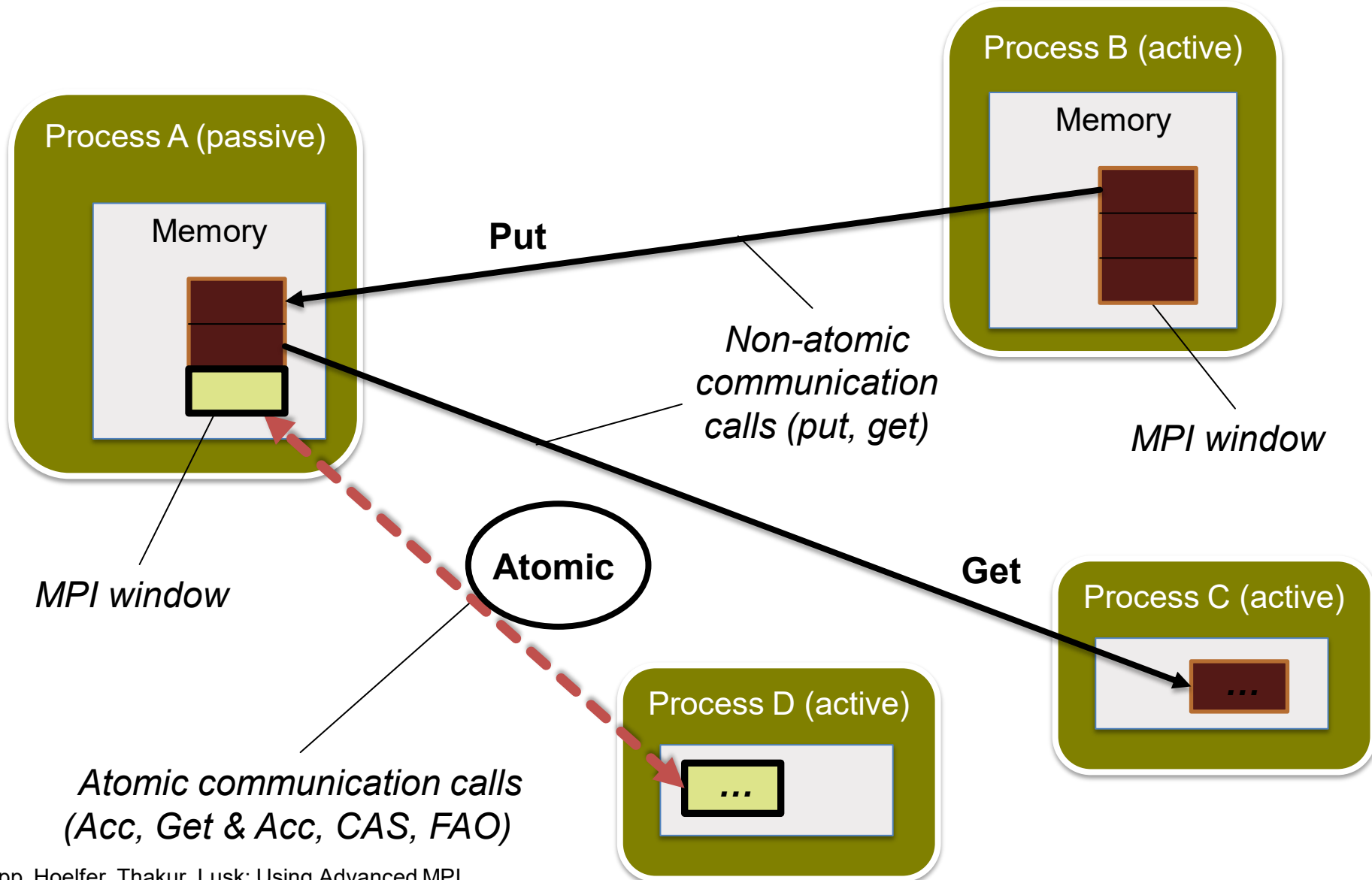
# MPI-3 RMA COMMUNICATION OVERVIEW



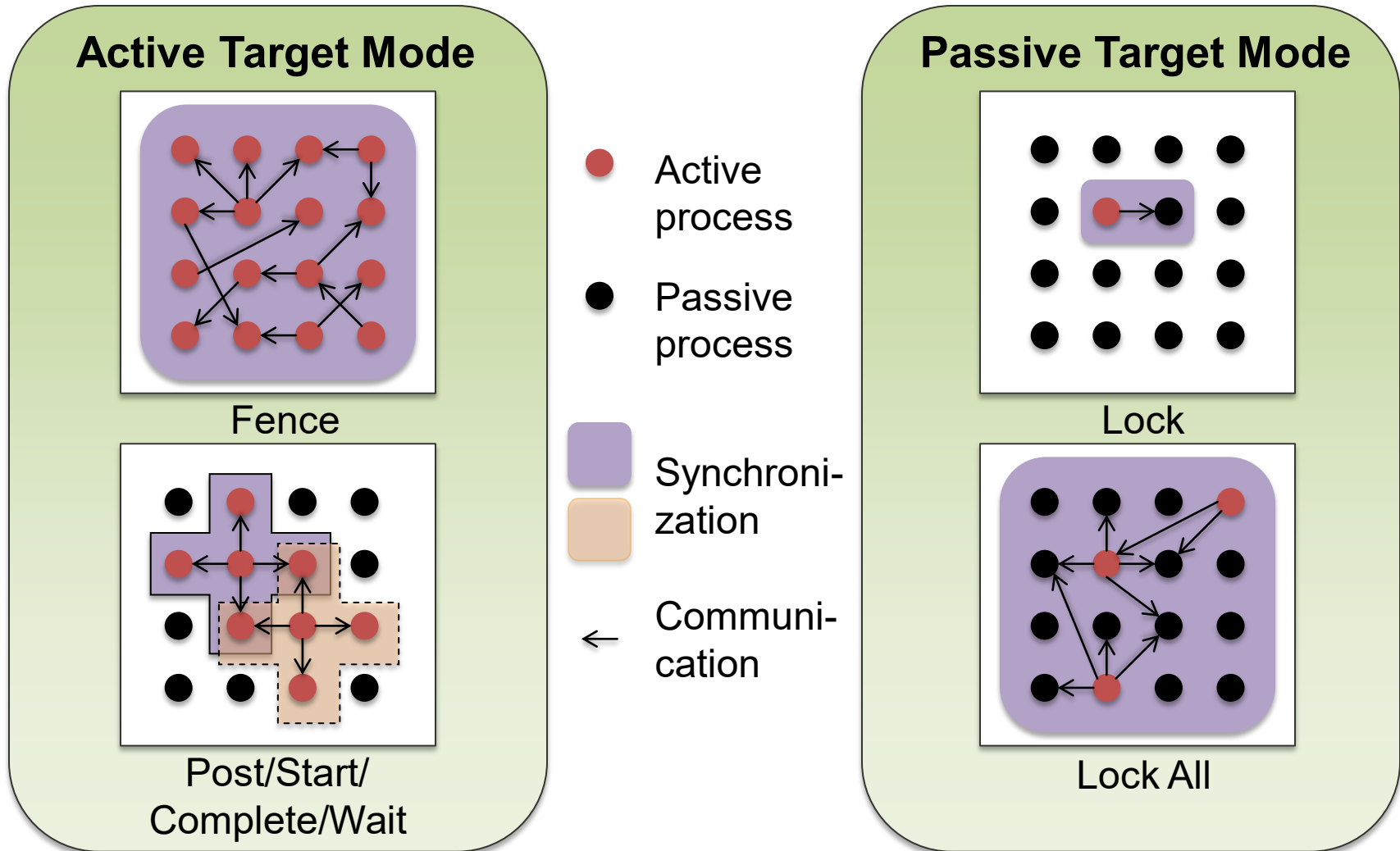
# MPI-3 RMA COMMUNICATION OVERVIEW



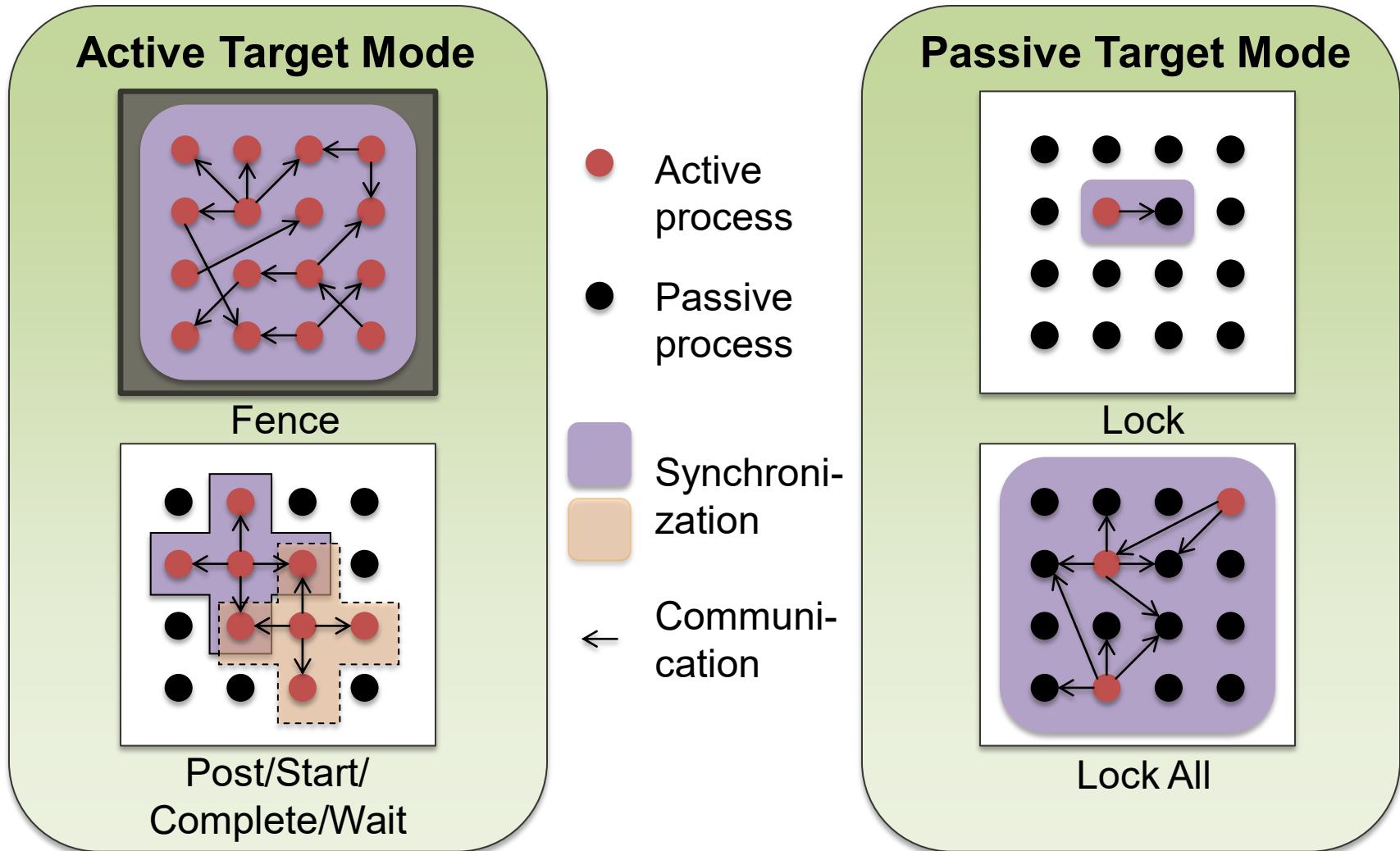
# MPI-3 RMA COMMUNICATION OVERVIEW



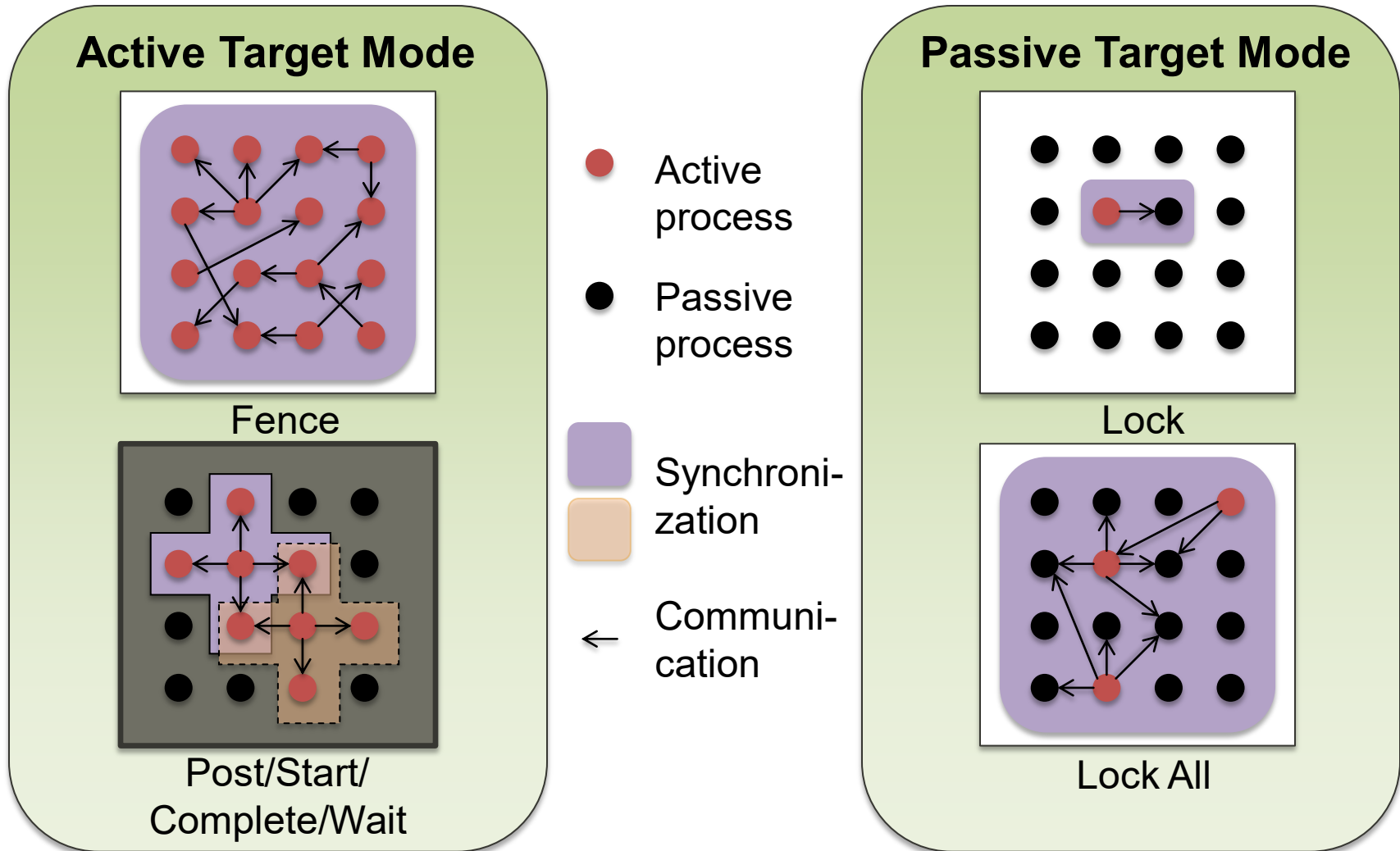
# MPI-3 RMA SYNCHRONIZATION OVERVIEW



# MPI-3 RMA SYNCHRONIZATION OVERVIEW



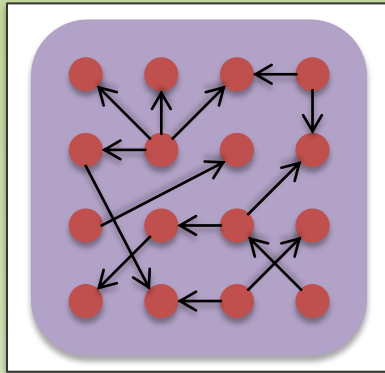
# MPI-3 RMA SYNCHRONIZATION OVERVIEW



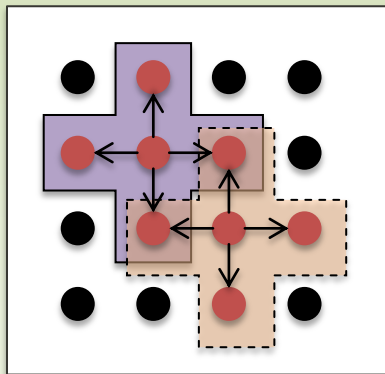


# MPI-3 RMA SYNCHRONIZATION OVERVIEW

## Active Target Mode



Fence



Post/Start/  
Complete/Wait

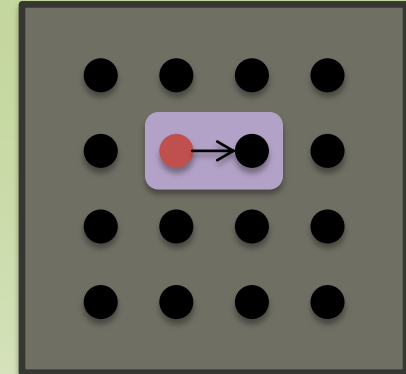
● Active process

● Passive process

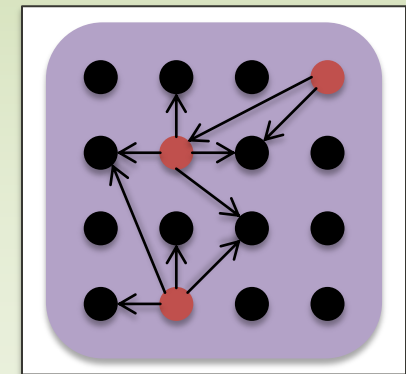
■ Synchroni-  
zation

← Communi-  
cation

## Passive Target Mode

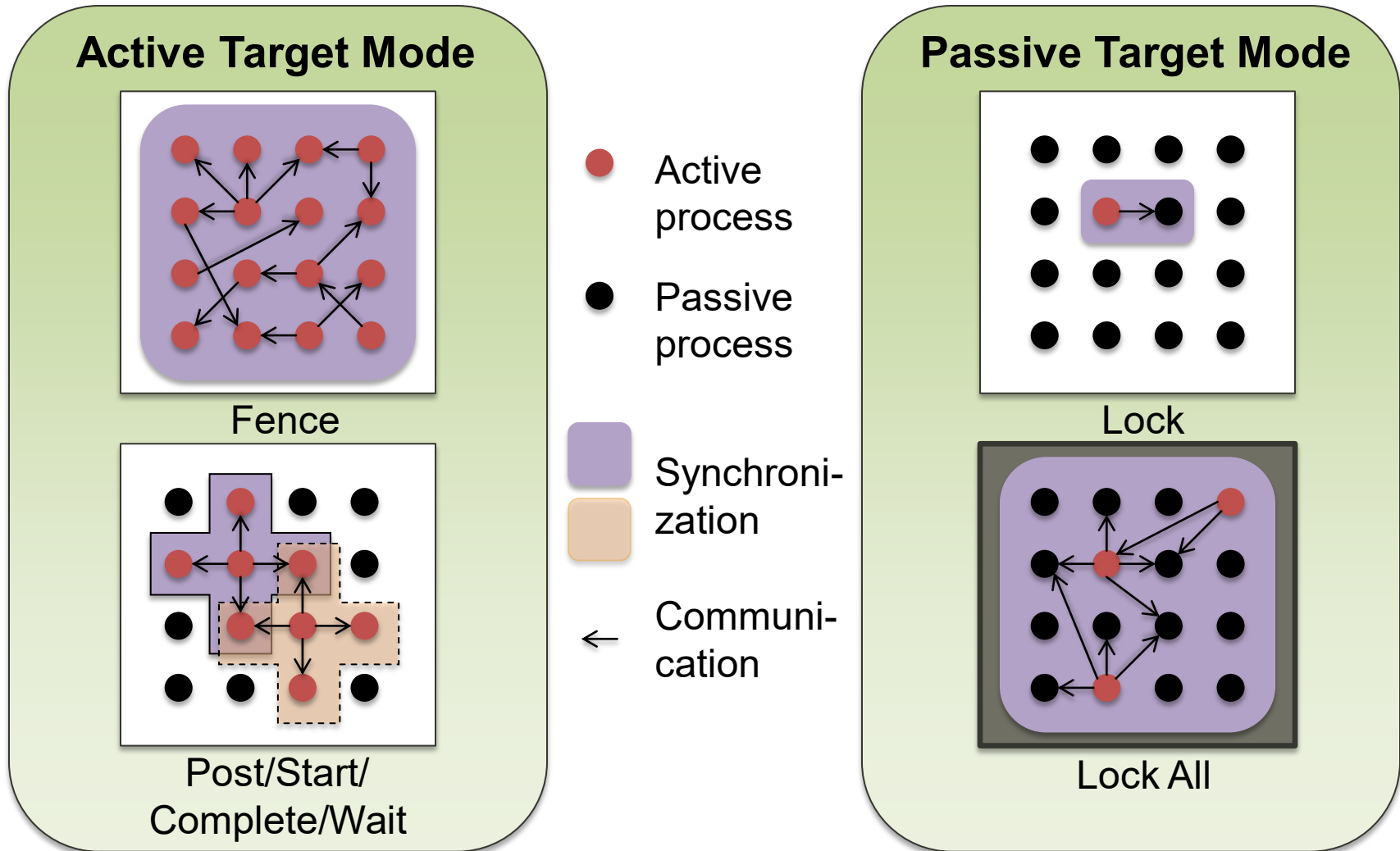


Lock



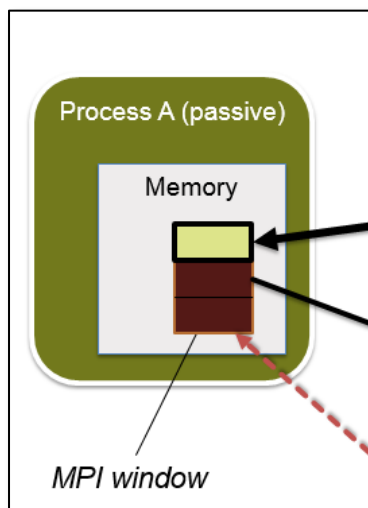
Lock All

# MPI-3 RMA SYNCHRONIZATION OVERVIEW

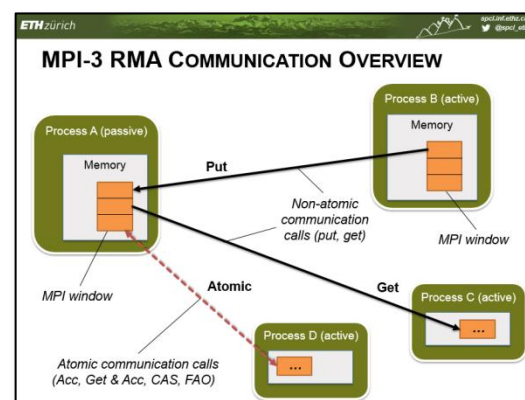


# SCALABLE PROTOCOLS & REFERENCE IMPLEMENTATION

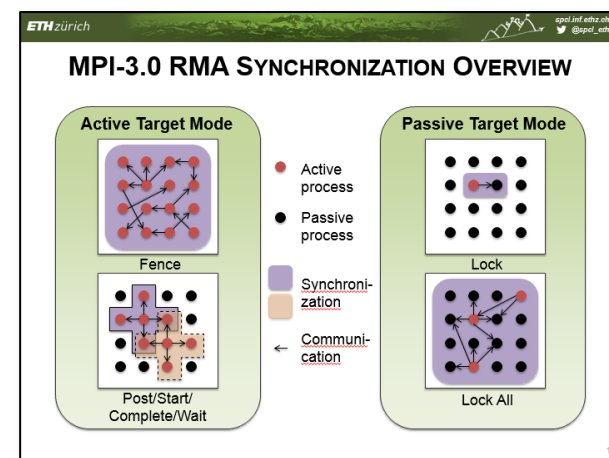
- Scalable & generic protocols
  - Can be used on any RDMA network (e.g., OFED/IB)
  - Window creation, communication and synchronization



Window creation



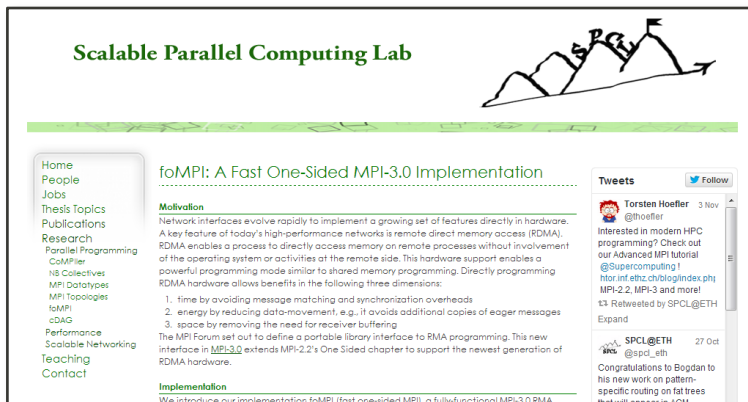
Communication



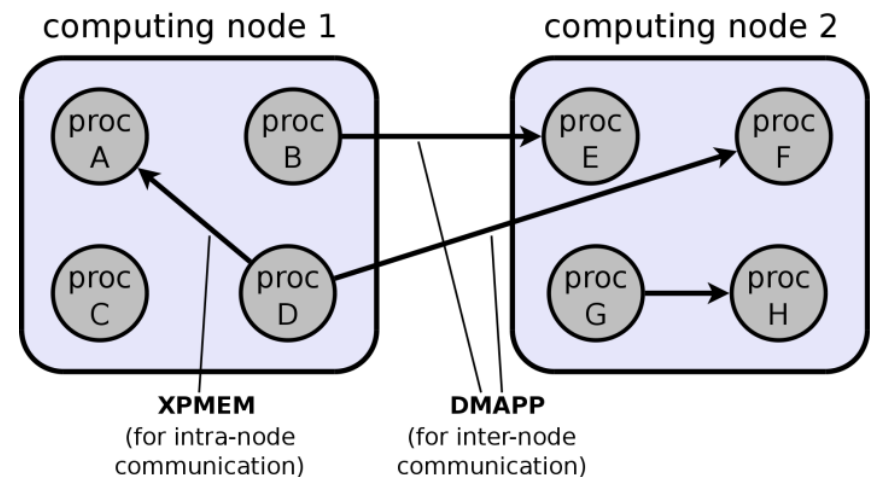
Synchronization

# 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

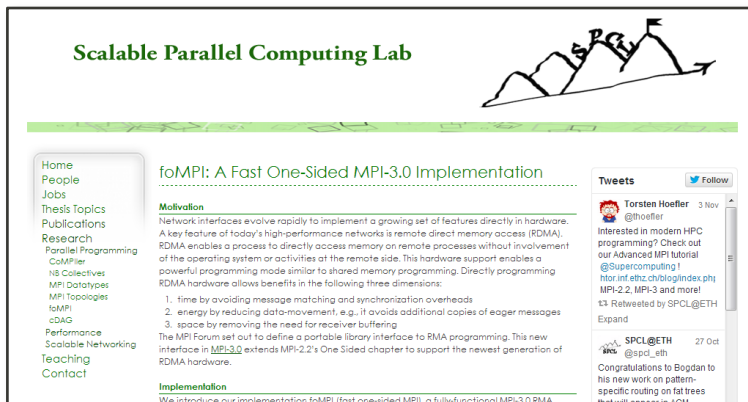


The screenshot shows the website for the Scalable Parallel Computing Lab. The main heading is "foMPI: A Fast One-Sided MPI-3.0 Implementation". Below the heading, there is a "Motivation" section and a "Tweets" section. The "Motivation" section discusses the benefits of RDMA hardware, such as avoiding message matching and synchronization overheads, and reducing data movement. The "Tweets" section shows a tweet from Torsten Hoefler (@thoefler) dated 3 Nov, expressing interest in modern HPC programming and mentioning the foMPI project. Another tweet from SPCL@ETH dated 27 Oct congratulates Bogdan for his work on pattern-specific routing on fat trees.

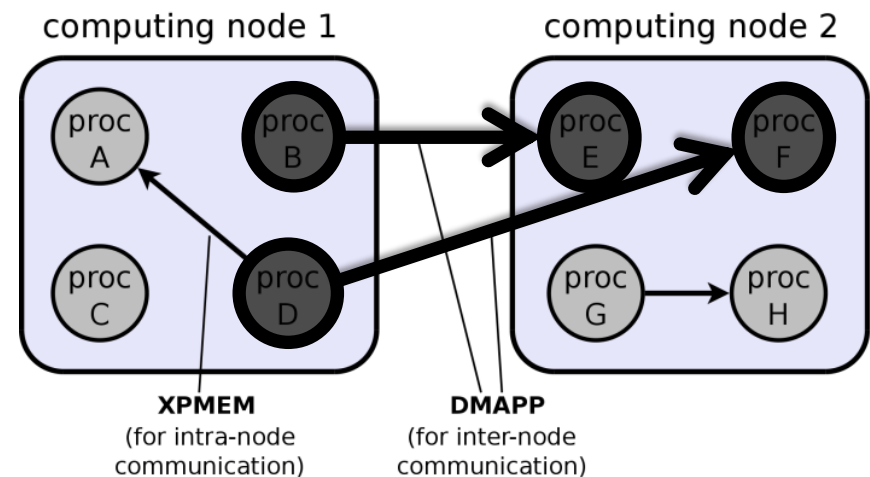


# SCALABLE PROTOCOLS & REFERENCE IMPLEMENTATION

- 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

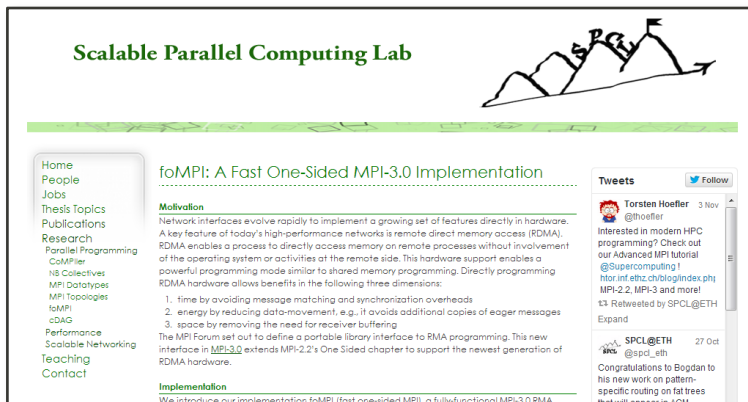


The screenshot shows the website for the Scalable Parallel Computing Lab. The main heading is "foMPI: A Fast One-Sided MPI-3.0 Implementation". Below the heading, there is a "Motivation" section and a "Tweets" section. The "Motivation" section discusses the benefits of RDMA hardware, such as avoiding message matching and synchronization overheads, and reducing data movement. The "Tweets" section shows a tweet from Torsten Hoefler (@thoefler) dated 3 Nov, expressing interest in modern HPC programming and mentioning the foMPI project. Another tweet from SPCL@ETH dated 27 Oct congratulates Bogdan for his work on pattern-specific routing on fat trees.



# 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 Parallel Computing Lab**

**foMPI: A Fast One-Sided MPI-3.0 Implementation**

**Motivation**

Network interfaces evolve rapidly to implement a growing set of features directly in hardware. A key feature of today's high-performance networks is remote direct memory access (RDMA). RDMA enables a process to directly access memory on remote processes without involvement of the operating system or activities at the remote side. This hardware support enables a powerful programming mode similar to shared memory programming. Directly programming RDMA hardware allows benefits in the following three dimensions:

1. time by avoiding message matching and synchronization overheads
2. energy by reducing data-movement, e.g., if avoids additional copies of eager messages
3. space by removing the need for receiver buffering

The MPI Forum set out to define a portable library interface to RMA programming. This new interface in [MPI-3.0](#) extends MPI-2.2's One Sided chapter to support the newest generation of RDMA hardware.

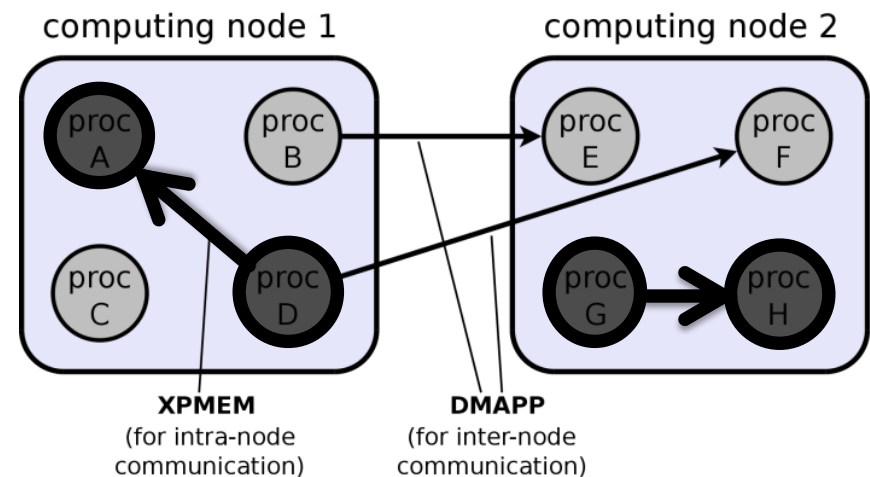
**Implementation**

We introduce our implementation foMPI (fast one-sided MPI) as fully functional MPI-3.0 RMA

**Tweets**

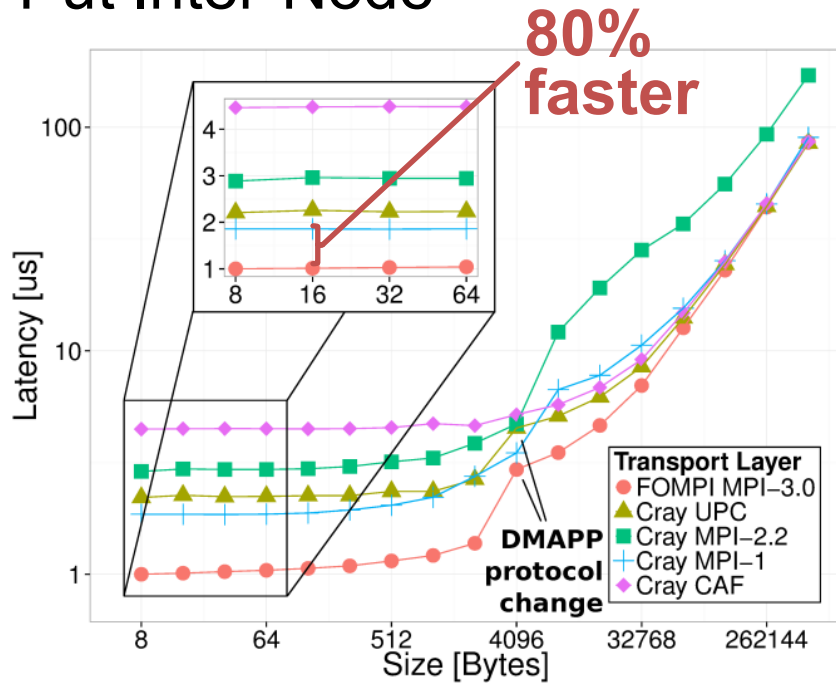
Torsten Hoefler @thoefler 3 Nov  
Interested in modern HPC programming? Check out our Advanced MPI tutorial @Supercomputing1 http://inf.ethz.ch/blog/index.php/MPI-2.2, MPI-3 and more!  
Retweeted by SPCL@ETH Expand

SPCL@ETH @spcl\_eth 27 Oct  
Congratulations to Bogdan to his new work on pattern-specific routing on fat trees that will speed up MPI

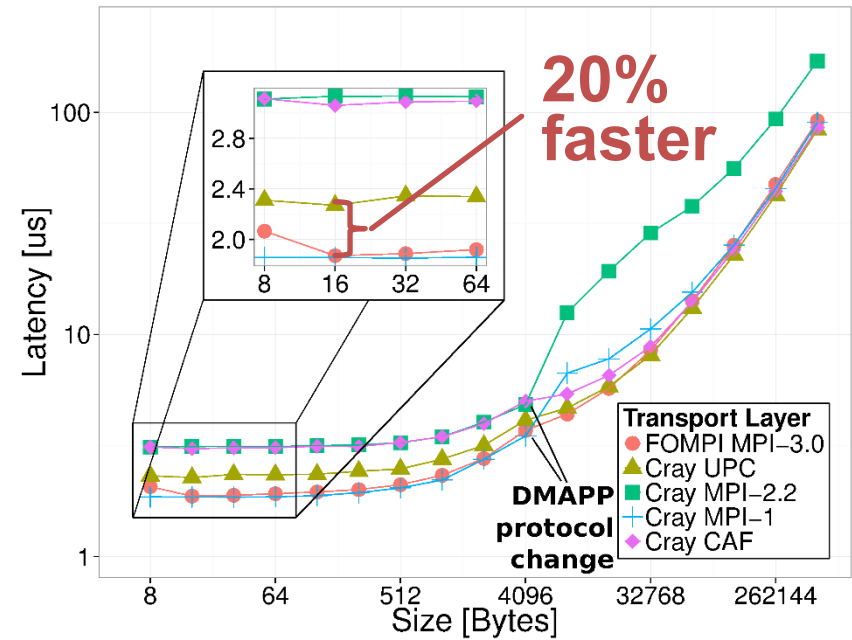


# PERFORMANCE INTER-NODE: LATENCY

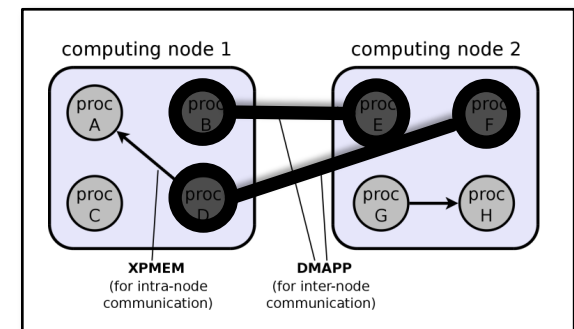
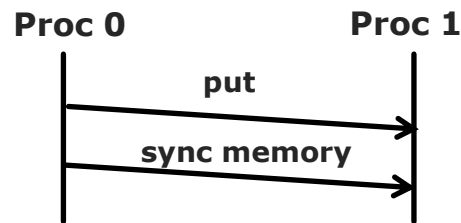
## Put Inter-Node



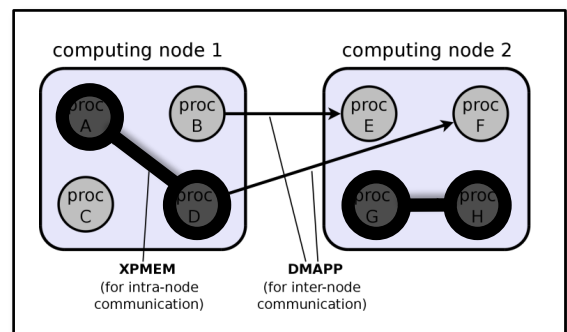
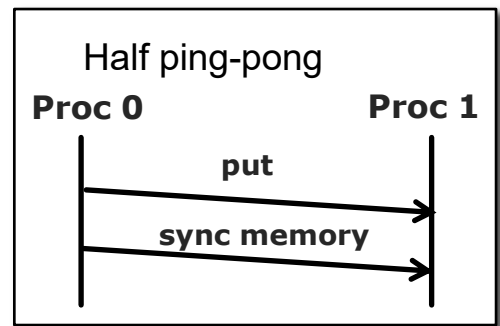
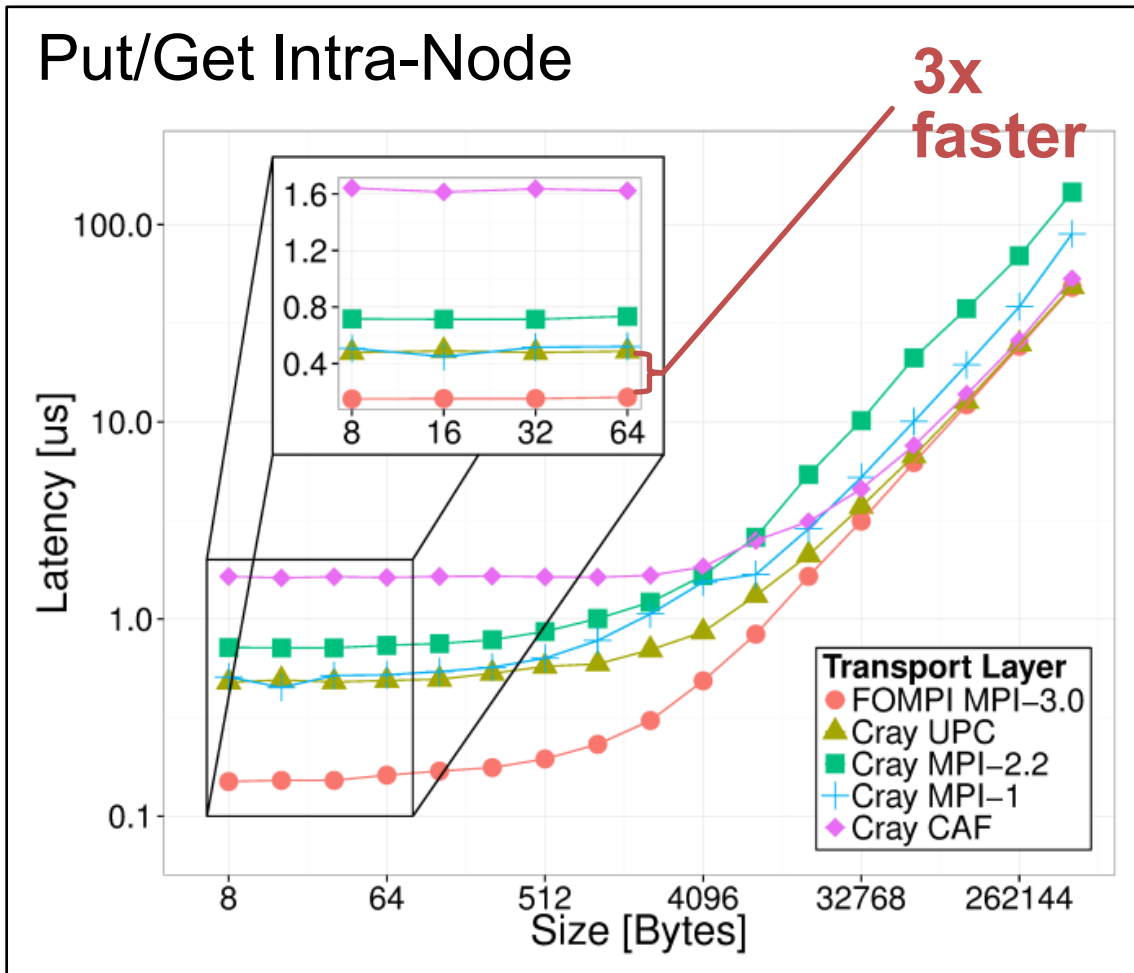
## Get Inter-Node



### Half ping-pong



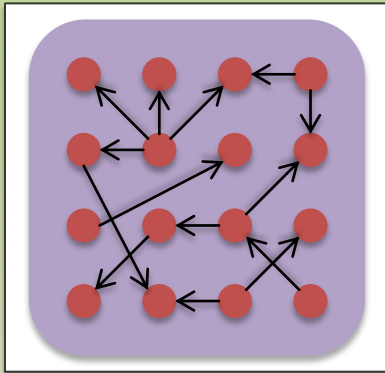
# PERFORMANCE INTRA-NODE: LATENCY



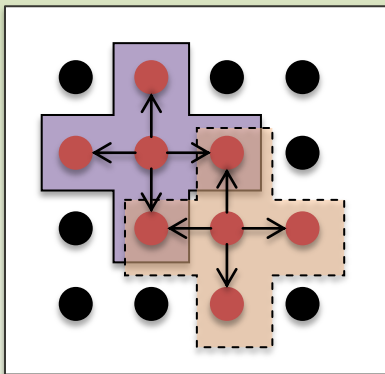


# PART 3: SYNCHRONIZATION

## Active Target Mode



Fence



Post/Start/  
Complete/Wait

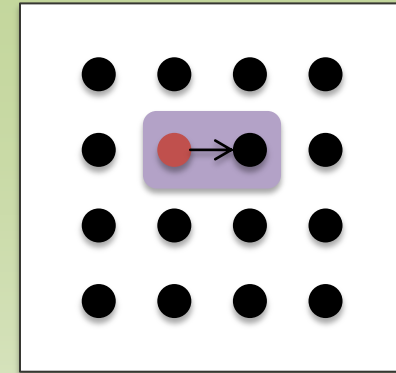
● Active process

● Passive process

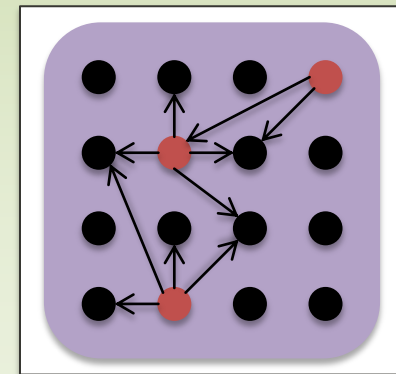
■ Synchroni-  
zation

← Communi-  
cation

## Passive Target Mode

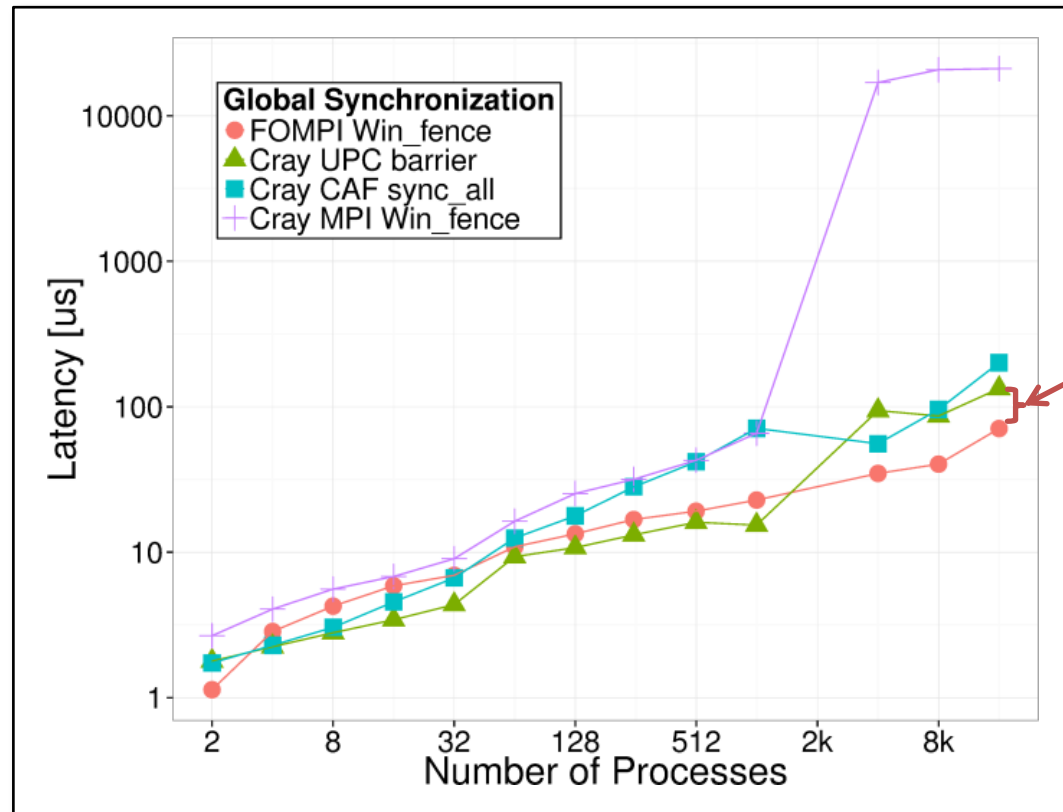


Lock



Lock All

# SCALABLE FENCE PERFORMANCE



**90%  
faster**

Time bound

$\mathcal{O}(\log p)$

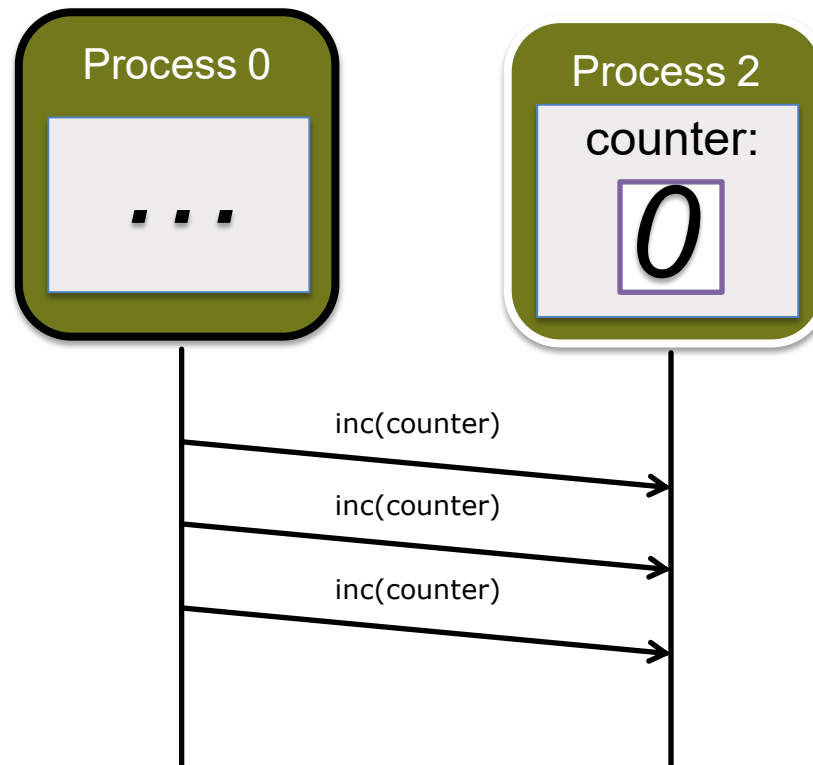
Memory bound

$\mathcal{O}(1)$

# FLUSH SYNCHRONIZATION

Time bound	$O(1)$
Memory bound	$O(1)$

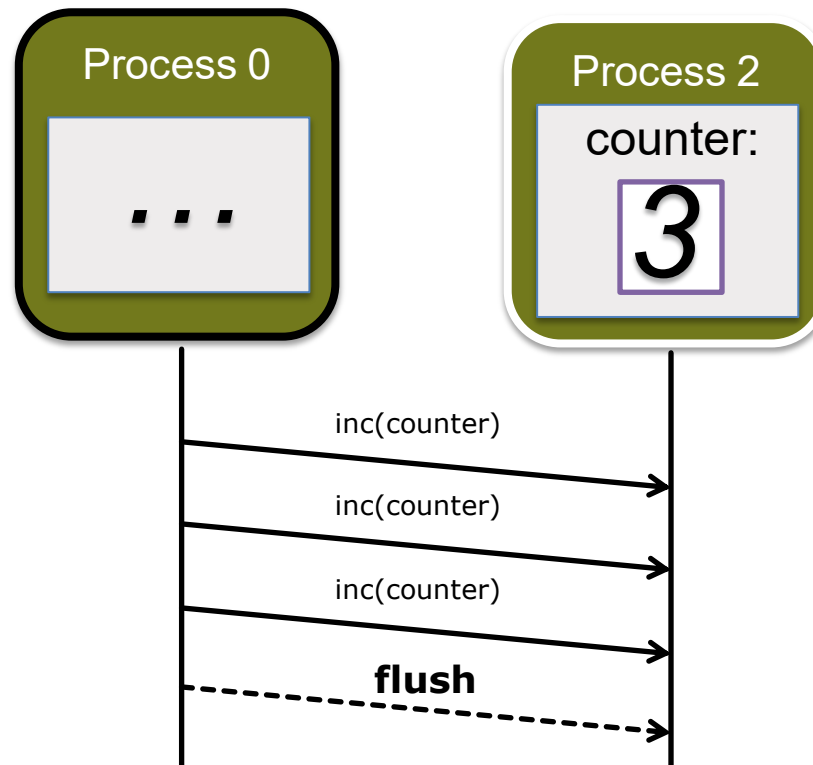
- Guarantees remote completion
- Performs a remote bulk synchronization and an x86 mfence
- One of the most performance critical functions, we add only **78 x86** CPU instructions to the critical path



# FLUSH SYNCHRONIZATION

Time bound	$O(1)$
Memory bound	$O(1)$

- 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

Fence	$\mathcal{P}_{fence} = 2.9\mu s \cdot \log_2(p)$
PSCW	$\mathcal{P}_{start} = 0.7\mu s, \mathcal{P}_{wait} = 1.8\mu s$ $\mathcal{P}_{post} = \mathcal{P}_{complete} = 350ns \cdot k$
Locks	$\mathcal{P}_{lock,excl} = 5.4\mu s$ $\mathcal{P}_{lock,shrd} = \mathcal{P}_{lock\_all} = 2.7\mu s$ $\mathcal{P}_{unlock} = \mathcal{P}_{unlock\_all} = 0.4\mu s$ $\mathcal{P}_{flush} = 76ns$ $\mathcal{P}_{sync} = 17ns$

Performance functions for communication protocols

Put/get	$\mathcal{P}_{put} = 0.16ns \cdot s + 1\mu s$ $\mathcal{P}_{get} = 0.17ns \cdot s + 1.9\mu s$
Atomics	$\mathcal{P}_{acc,sum} = 28ns \cdot s + 2.4\mu s$ $\mathcal{P}_{acc,min} = 0.8ns \cdot s + 7.3\mu s$

# APPLICATION PERFORMANCE

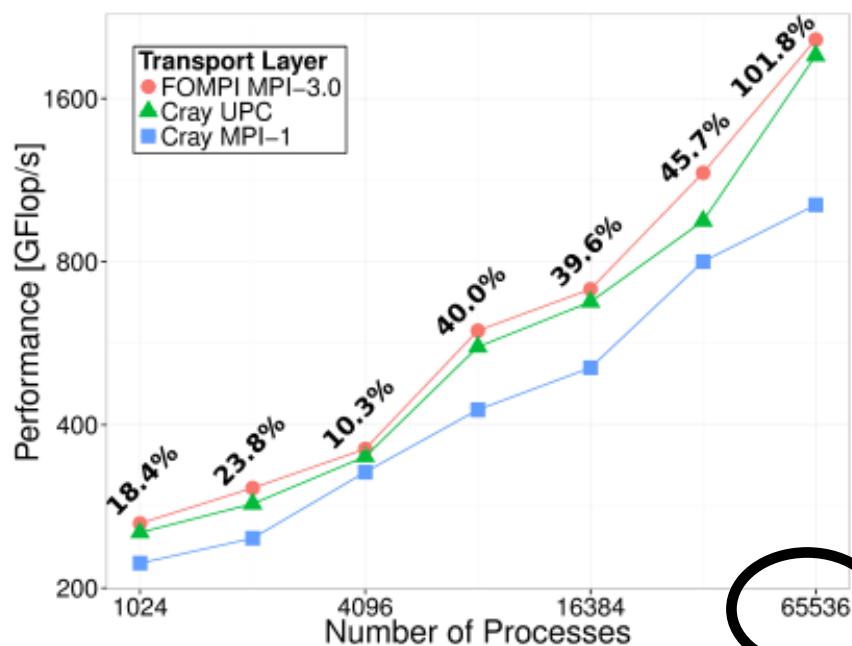
- Evaluation on Blue Waters System
  - 22,640 computing Cray XE6 nodes
  - 724,480 schedulable cores
- All microbenchmarks
- 4 applications
- One nearly full-scale run 😊



# PERFORMANCE: APPLICATIONS

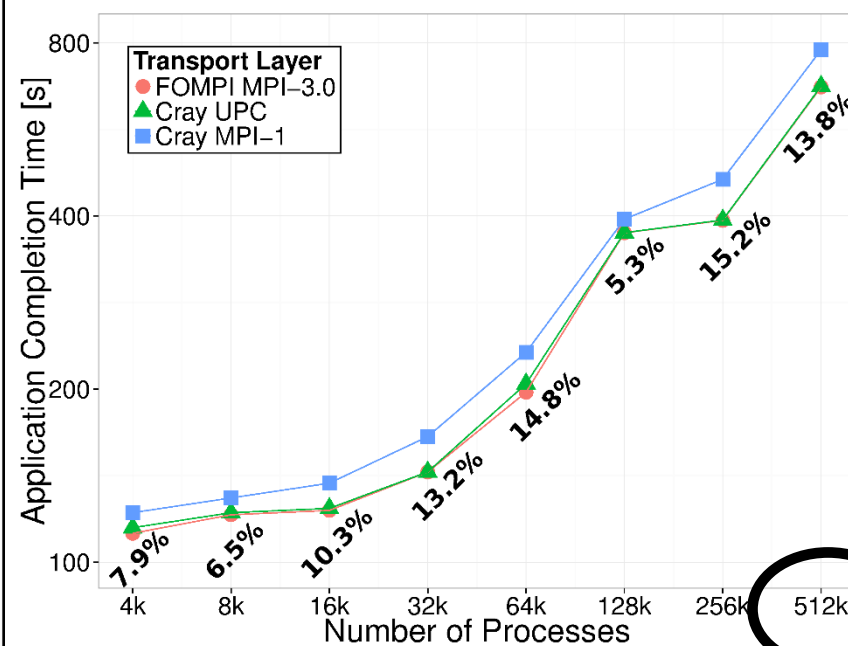
Annotations represent performance gain of foMPI [3] over Cray MPI-1.

## NAS 3D FFT [1] Performance



scale  
to 65k procs

## MILC [2] Application Execution Time



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

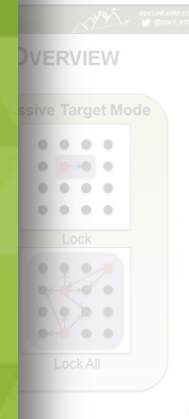
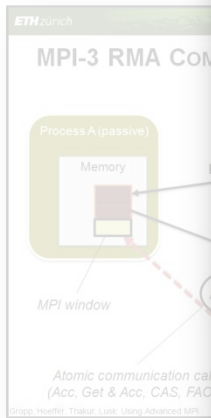
[3] Gerstenberger, Besta, Hoefler: Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC13

# IN CASE YOU WANT TO LEARN MORE

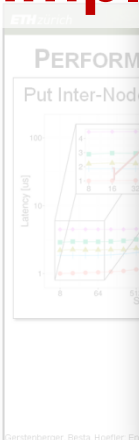
- Available
- Some ar

SCIENTIFIC  
AND  
ENGINEERING  
COMPUTATION  
SERIES

*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*

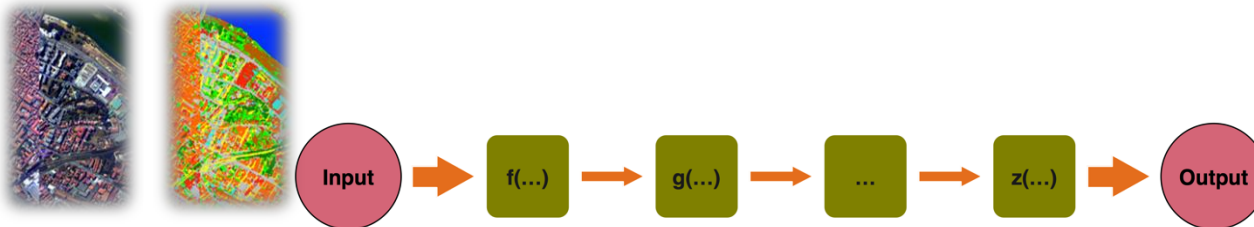
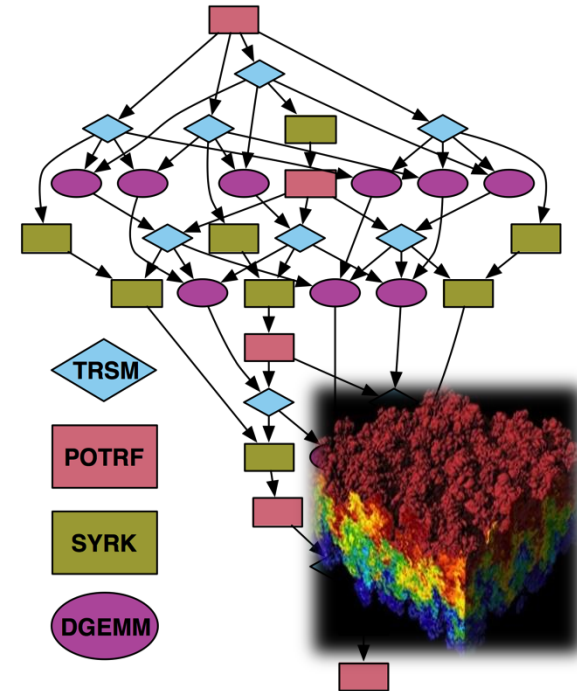
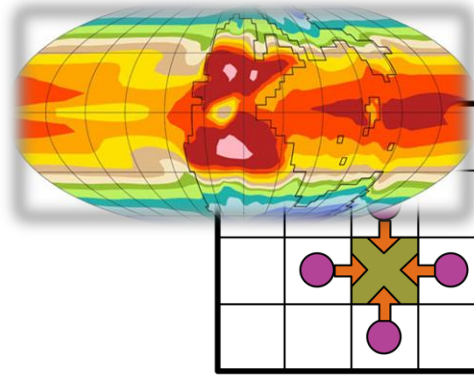
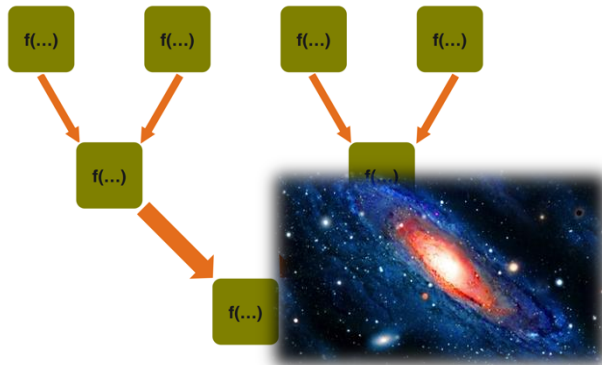




# PRODUCER-CONSUMER RELATIONS

- 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

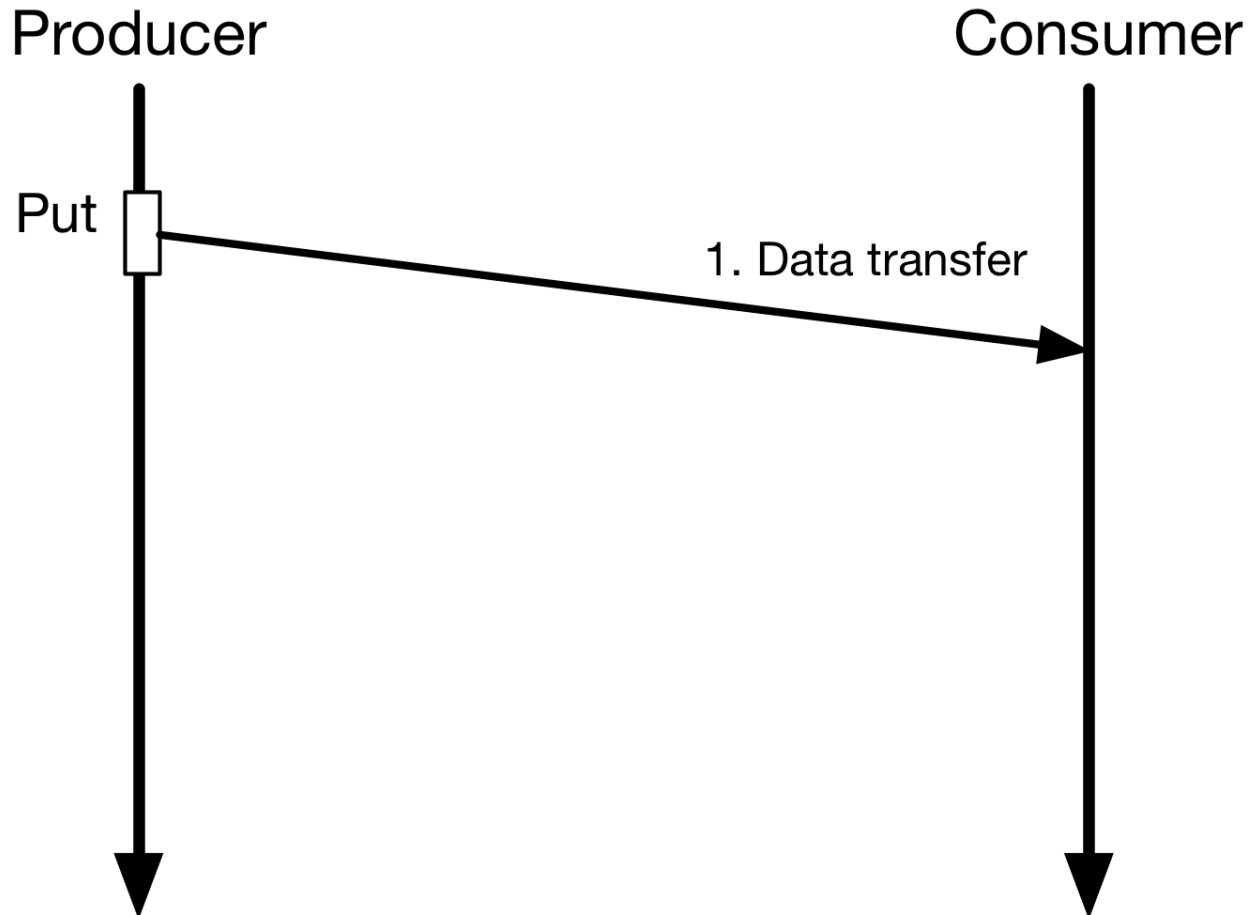
Producer



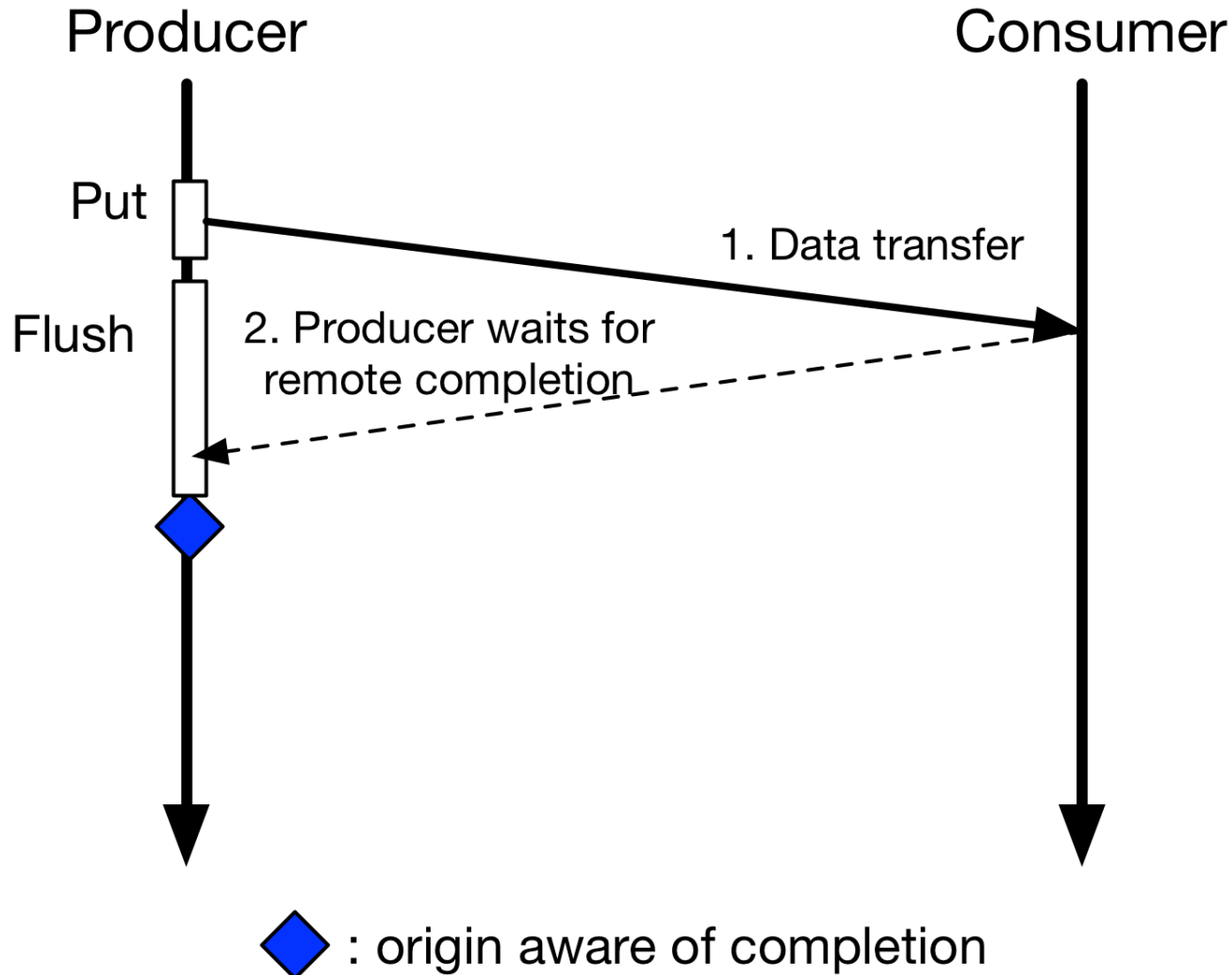
Consumer



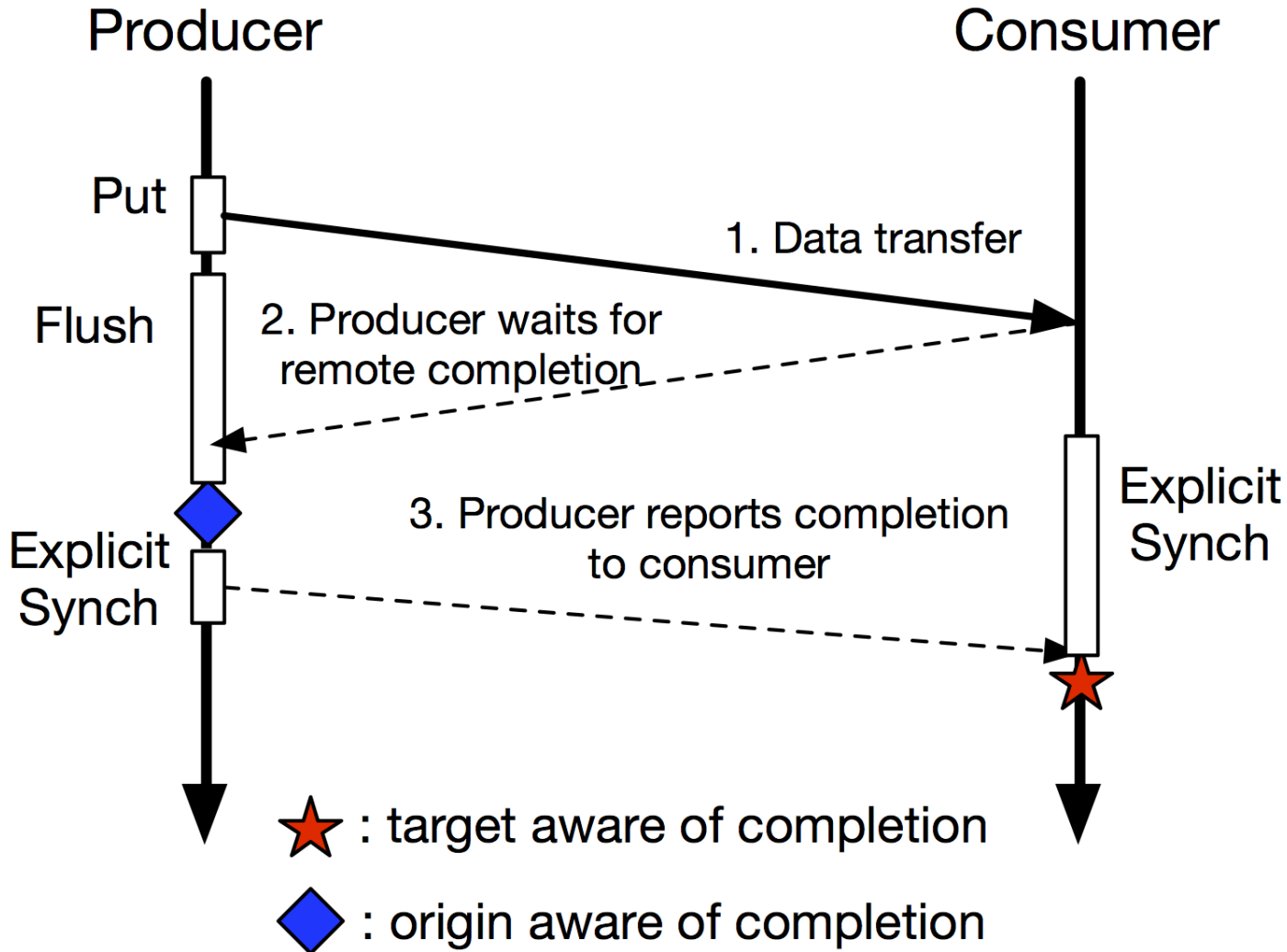
# ONE SIDED – PUT + SYNCHRONIZATION



# ONE SIDED – PUT + SYNCHRONIZATION

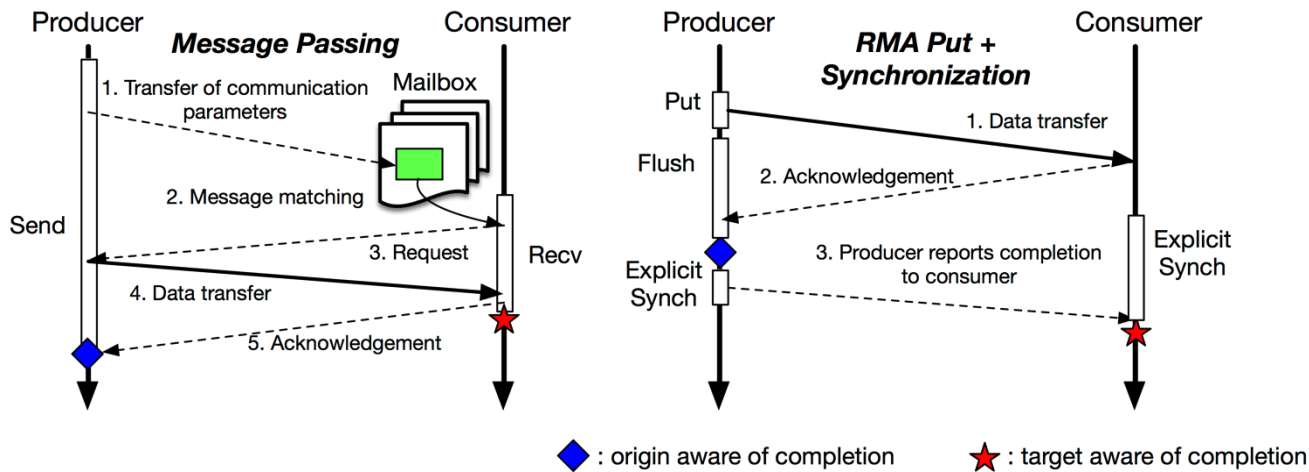


# ONE SIDED – PUT + SYNCHRONIZATION



**Critical path: 3 latencies**

# COMPARING APPROACHES

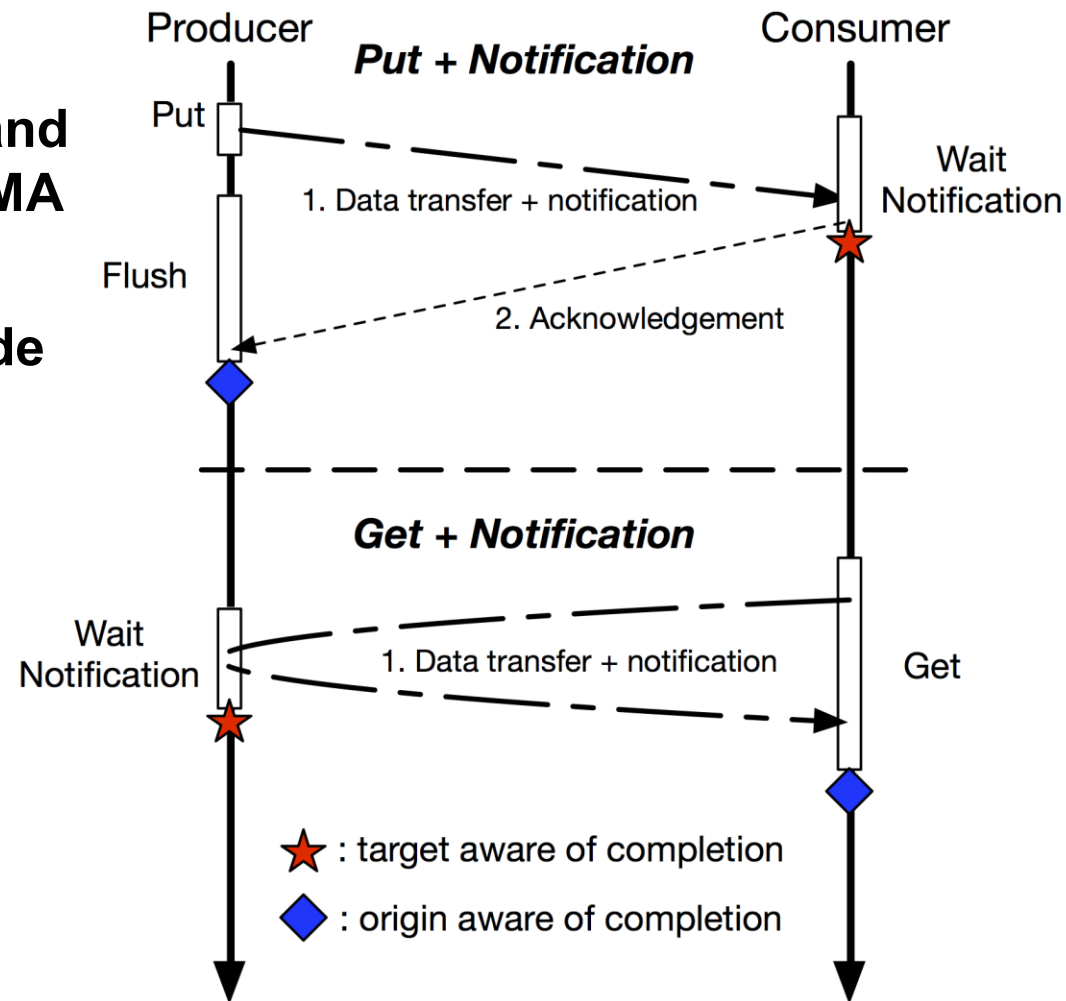


**Message Passing**  
 1 latency + copy /  
 3 latencies

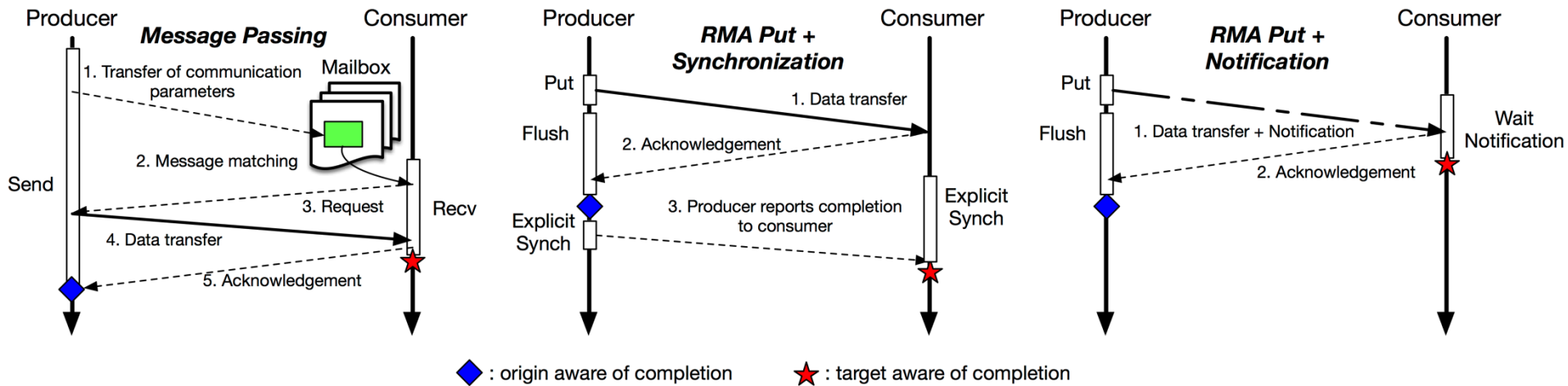
**One Sided**  
 3 latencies

# 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



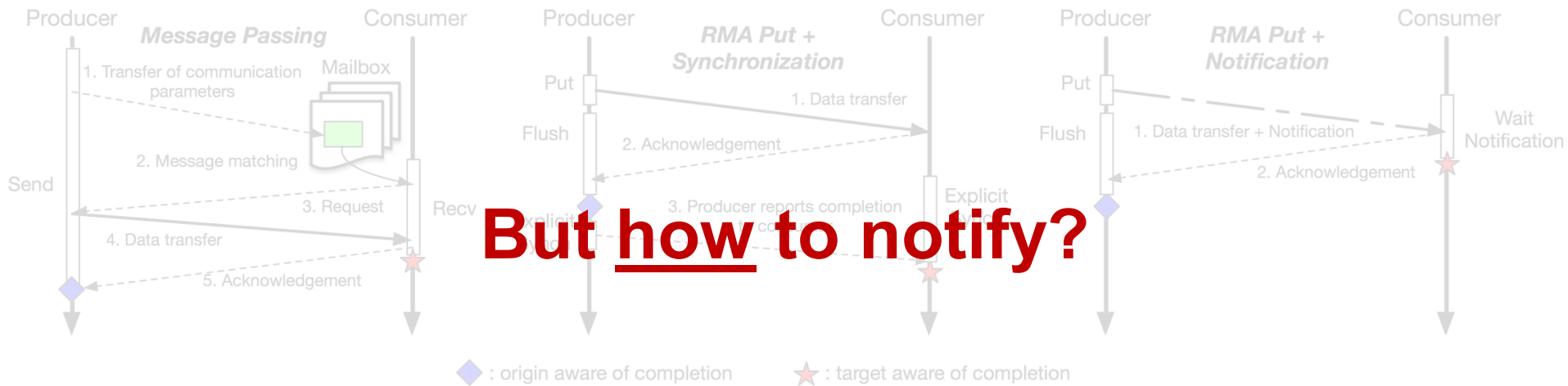
**Message Passing**  
1 latency + copy /  
3 latencies

**One Sided**  
3 latencies

**Notified Access**  
1 latency



# COMPARING APPROACHES



**But how to notify?**

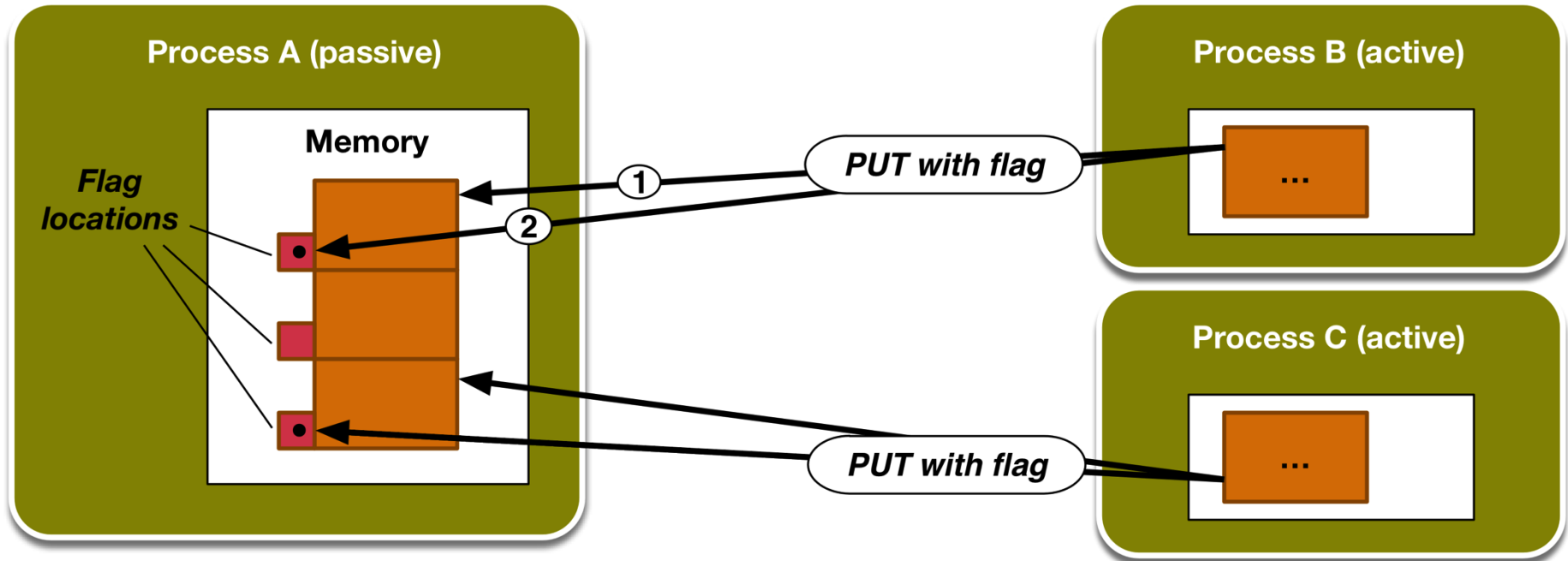
*Message Passing*  
1 latency + copy /  
3 latencies

*One Sided*  
3 latencies

*Notified Access*  
1 latency

# PREVIOUS WORK: OVERWRITING INTERFACE

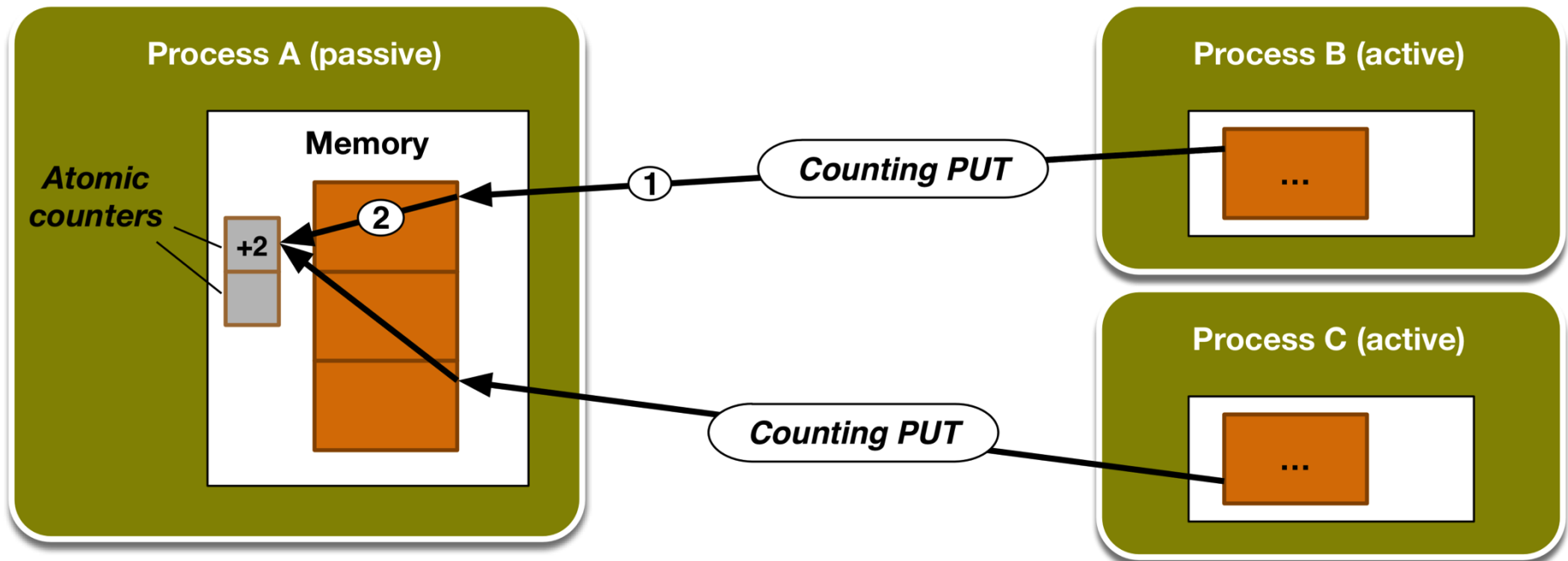
- **Flags (polling at the remote side)**
  - Used in *GASPI*, *DMAPP*, *NEON*



- **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

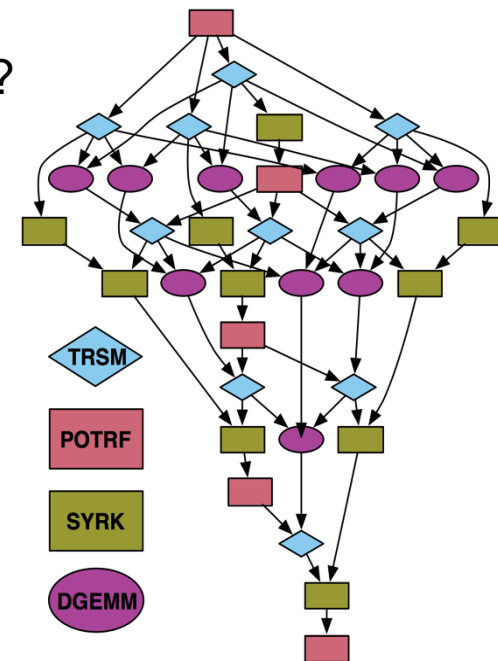
- **Atomic counters (accumulate notifications → scalable)**
  - Used in *Split-C*, *LAPI*, *SHMEM - Counting Puts*, ...



- **Disadvantages**
  - Dataflow applications may require many counters
  - High polling overhead to identify accesses
  - Does not preserve order (may not be linearizable)

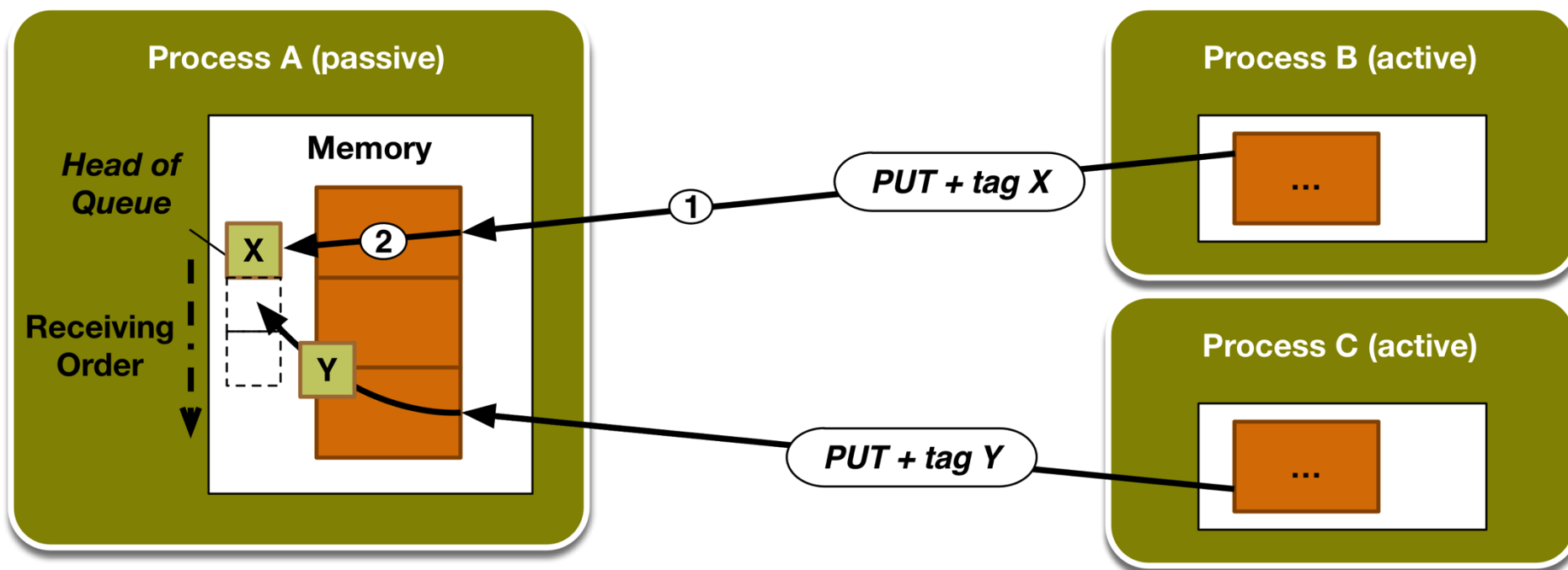
# WHAT IS A GOOD NOTIFICATION INTERFACE?

- **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

---

```
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

---

```
/*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

```
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

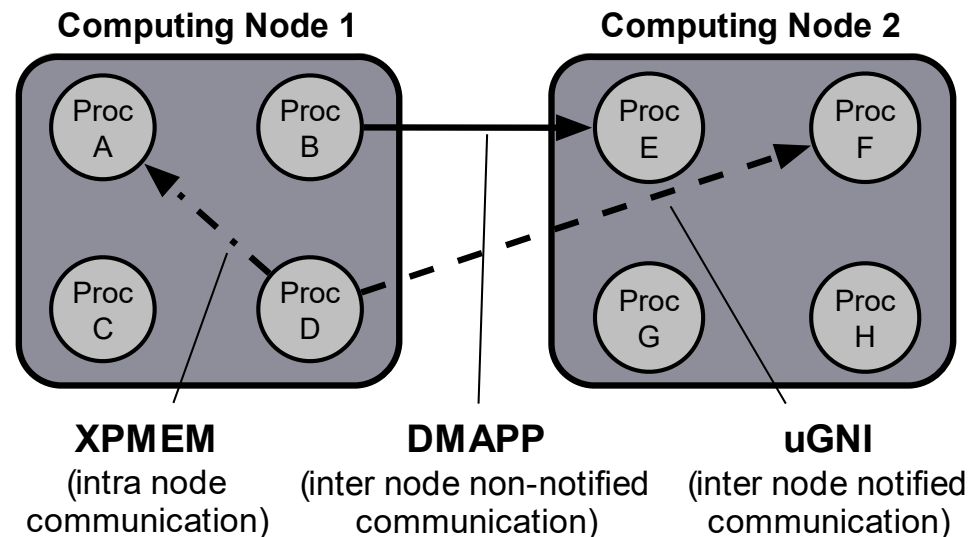
```
int MPI_Notify_init(MPI_Win win, int src_rank, int tag, int expected_count, MPI_Request *request);
/*Functions already available in MPI*/
int MPI_Start(MPI_Request *request);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);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

# 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

The screenshot shows the website for the Scalable Parallel Computing Lab. The main heading is "foMPI-NA: enabling Notified Access semantics in MPI". Below this, there is a "Motivation" section explaining that modern interconnects offer Remote Direct Memory Access (RDMA) features, which reflect the characteristics of RMA programming models. It notes that RDMA has also been used to implement fast message passing systems, but both RMA and Message Passing require at least an additional round-trip message between two processes to synchronize. The additional synchronization overhead may drastically reduce the performance of applications where synchronization is needed after the transmission of each message. For instance, this per-message synchronization is used in the well-known producer-consumer class of problems.

There is also a "Communication Schemes" section. It states that the Message Passing programming model mainly provides two calls for exchanging data between processes: the send and recv. These two calls ensure both data transfer and synchronization. In current RDMA-based implementations of Message Passing libraries, two classes of protocols are mainly used: the Eager and the Rendezvous. The first one is used to provide a very low latency on small transfers. In this case, as soon as the producer invokes the send, the message is directly transferred to a mailbox (a pre-allocated intermediate buffer) of the receiver side. This method can be used only if the message fits the mailbox, it provides low latencies but the use of these pre-allocated buffers is considered not scalable. The second protocol used in Message Passing libraries is the Rendezvous protocol. In this case, a longer synchronization phase is needed but the protocol enables zero-copy transfers. Alternatively, RMA does not provide data transfer and synchronization within the same primitive. These two phases require the invocation of different primitives. The following figures depict the typical interconnect patterns used in both communication schemes.





# 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



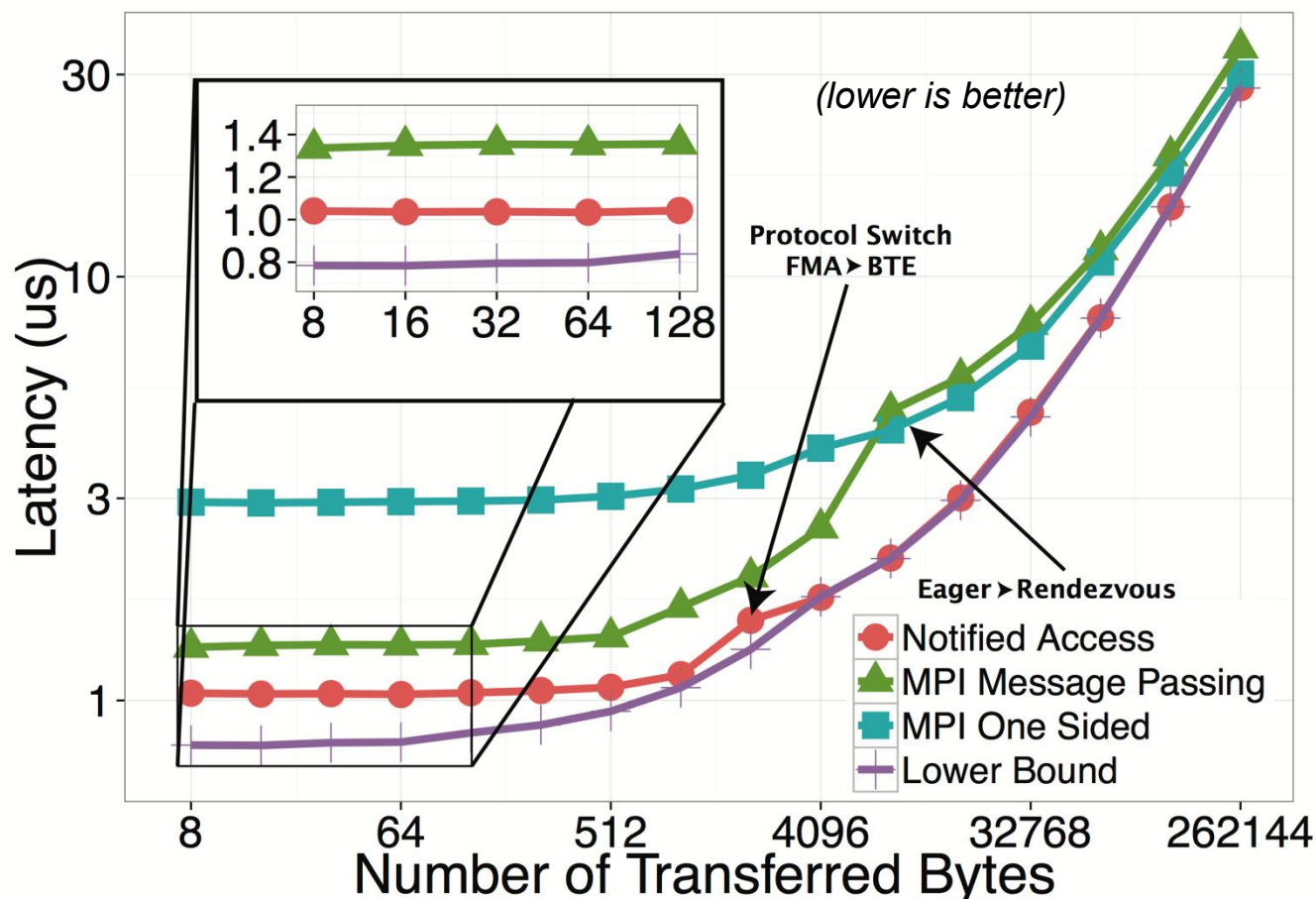
## CSCS

Centro Svizzero di Calcolo Scientifico  
Swiss National Supercomputing Centre



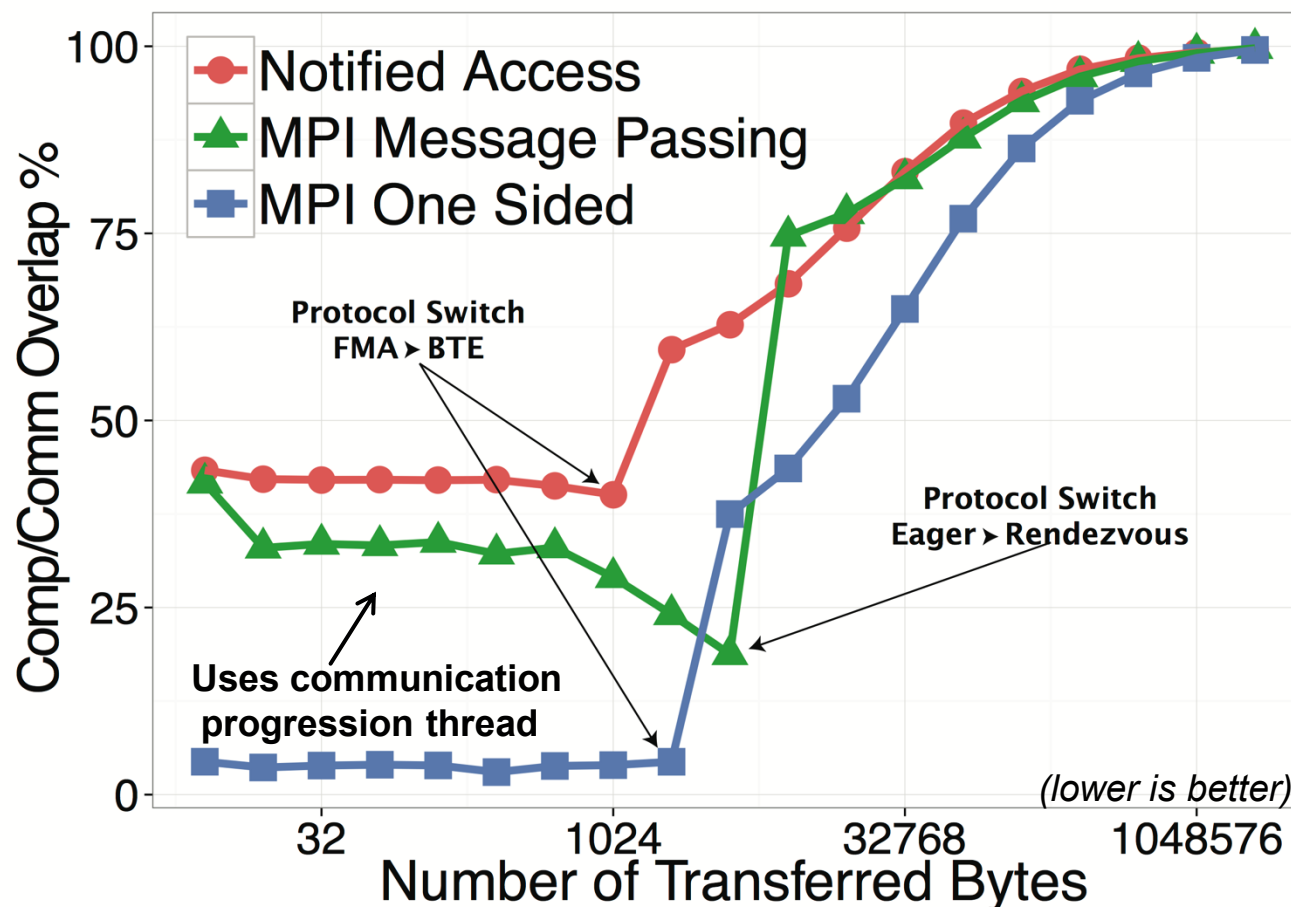
# PING PONG PERFORMANCE (INTER-NODE)

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



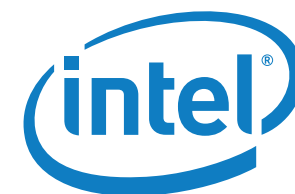
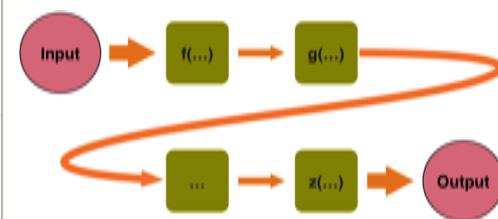
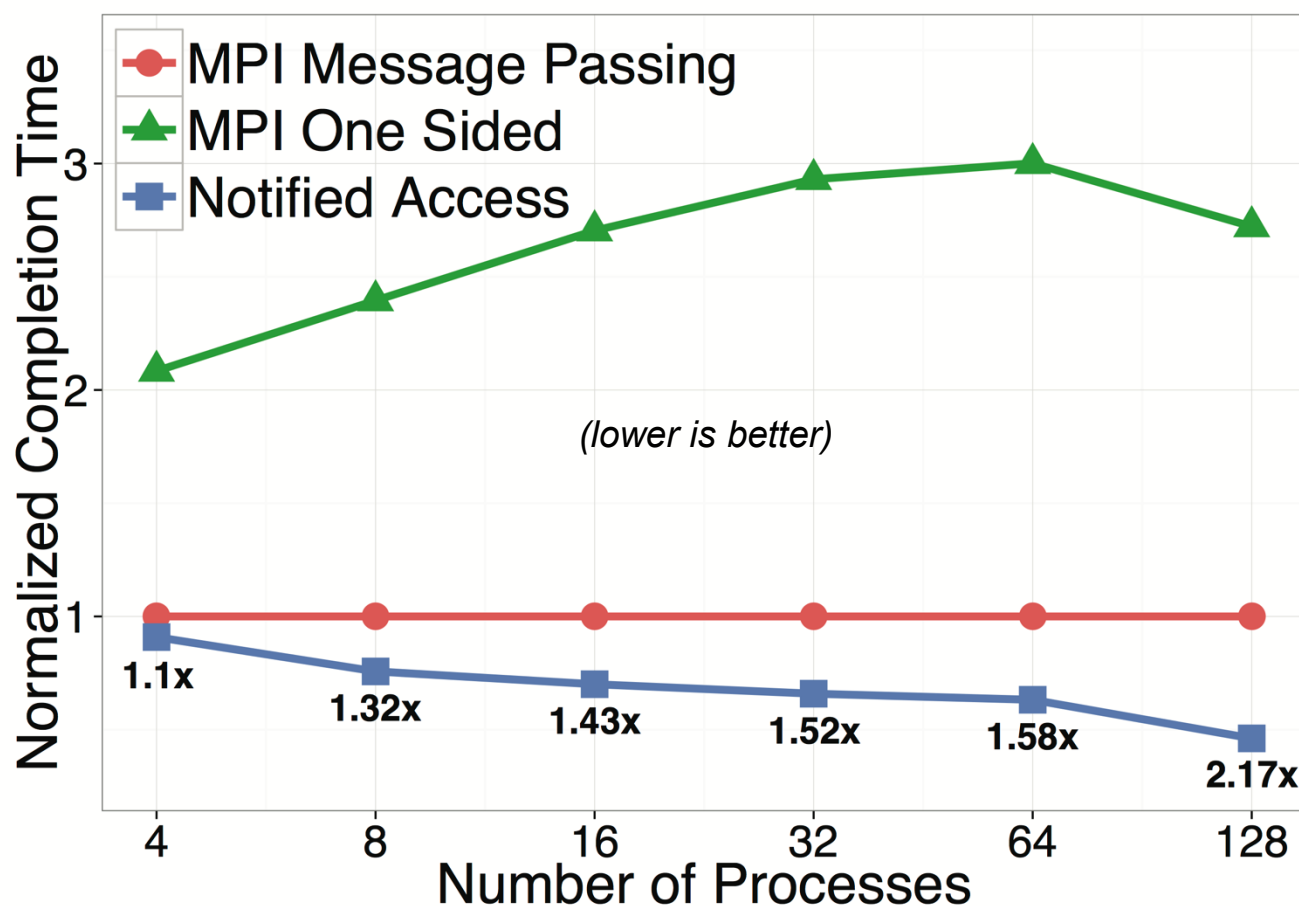
# COMPUTATION/COMMUNICATION OVERLAP

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



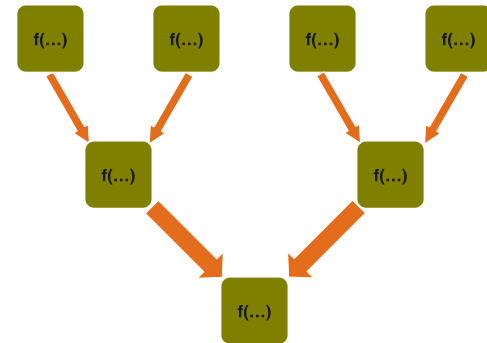
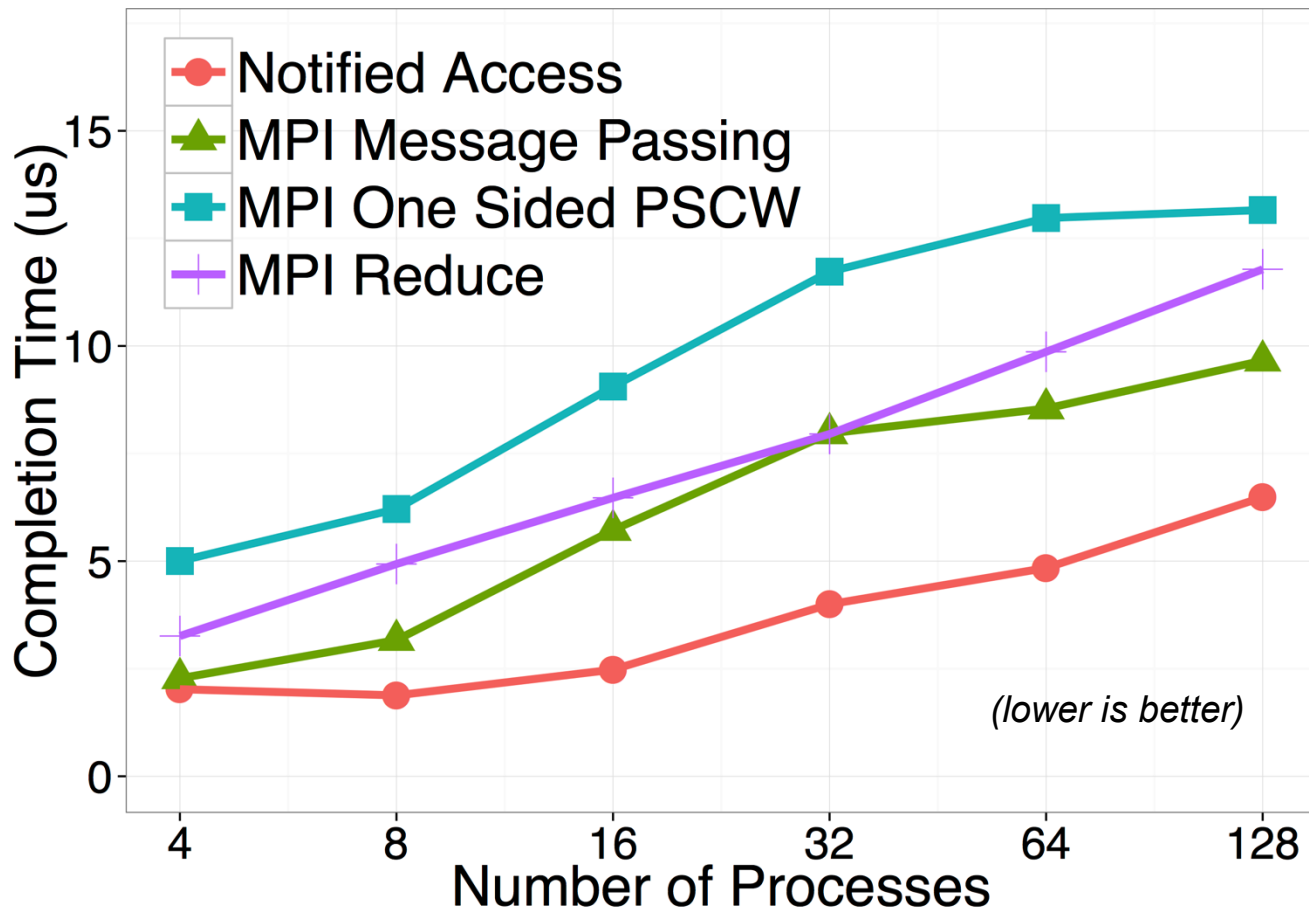
# PIPELINE – ONE-TO-ONE SYNCHRONIZATION

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



# REDUCE – ONE-TO-MANY SYNCHRONIZATION

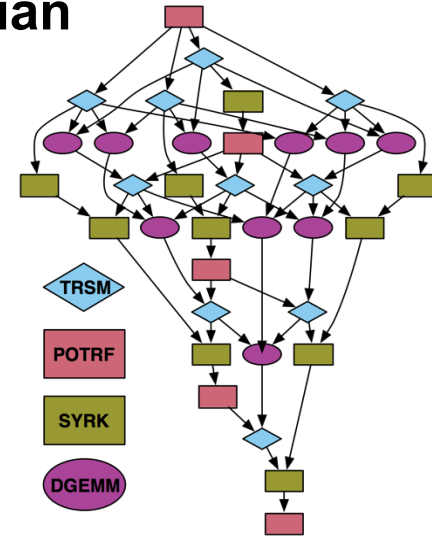
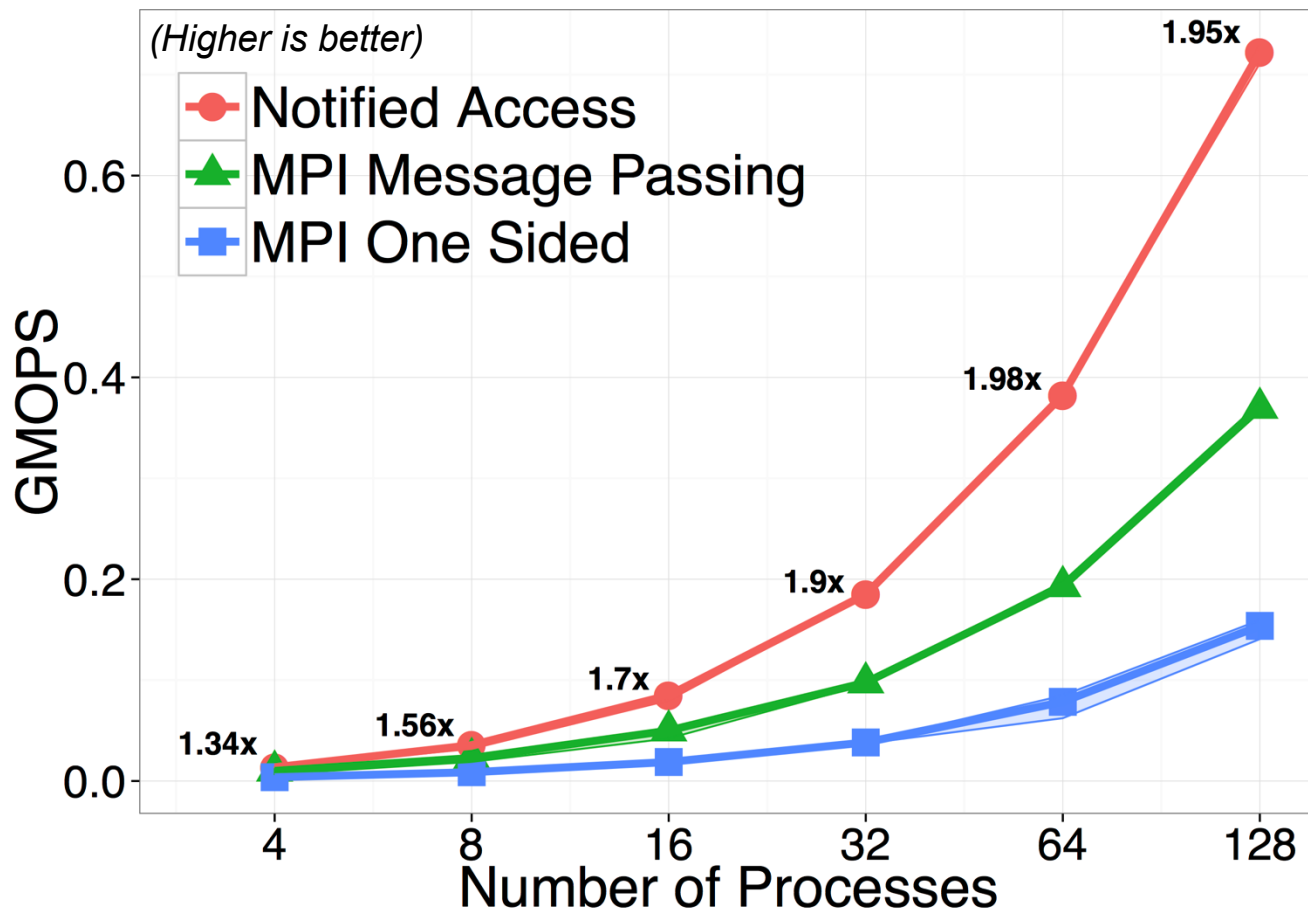
- Reduce as an example (same for FMM, BH, etc.)
  - Small data (8 Bytes), 16-ary tree
  - 1000 repetitions, each timed separately with RDTSC



**CRAY**  
Supercomputer

# CHOLESKY – MANY-TO-MANY SYNCHRONIZATION

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



# 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



applicable at least to:



OPENFABRICS  
ALLIANCE



portals



	Shared Memory	uGNI FMA	uGNI BTE
L	$0.25\mu s$	$1.02\mu s$	$1.32\mu s$
G	$0.08ns$	$0.105ns$	$0.101ns$

Function	Time
MPI_Notify_init	$t_{init} = 0.07\mu s$
MPI_Request_free	$t_{free} = 0.04\mu s$
MPI_Start	$t_{start} = 0.008\mu s$
MPI_{Put Get}_notify	$t_{na} = 0.29\mu s$

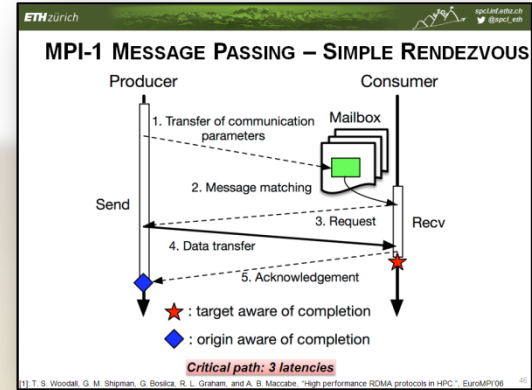
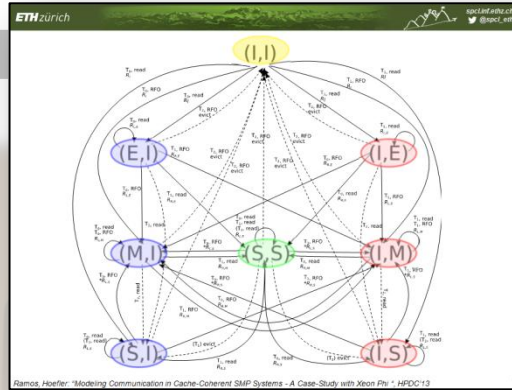
# ACKNOWLEDGMENTS





### 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



### PERFORMANCE MODELING

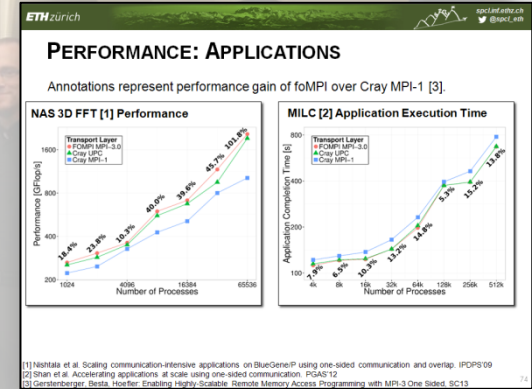
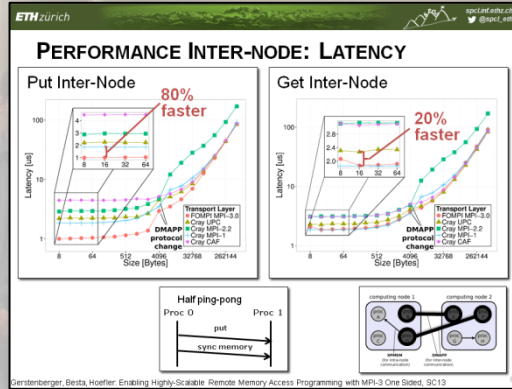
Performance functions for synchronization protocols

Fence	$P_{fence} = 2.9\mu s \cdot \log_2(p)$
PSCW	$P_{start} = 0.7\mu s, P_{wait} = 1.8\mu s$ $P_{post} = P_{complete} = 350ns \cdot k$
Locks	$P_{lock\_exc} = 5.4\mu s$ $P_{lock\_shrd} = P_{lock\_all} = 2.7\mu s$ $P_{unlock} = P_{unlock\_all} = 0.4\mu s$ $P_{flush} = 76ns$ $P_{sync} = 17ns$

Performance functions for communication protocols

Put/get	$P_{put} = 0.16ns \cdot s + 1\mu s$ $P_{get} = 0.17ns \cdot s + 1.9\mu s$
Atomics	$P_{accsum} = 2.8ns \cdot s + 2.4\mu s$ $P_{accmin} = 0.8ns \cdot s + 7.3\mu s$

Zersterberger, Besta, Hoefler, Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided, SC'13

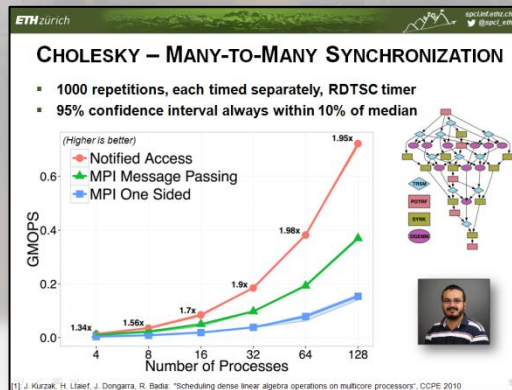


### IDEA: RMA NOTIFICATIONS

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

★ : target aware of completion  
◆ : origin aware of completion

Hoefler, "Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization", ICPDS'15



SCIENTIFIC  
AND  
ENGINEERING  
COMPUTATION  
SERIES

### Using Advanced MPI

Modern Features of the Message-Passing Interface

William Gropp  
Torsten Hoefler  
Rajeev Thakur  
Ewing Lusk