SOFTWARE COMBINING TO MITIGATE MULTITHREADED MPI CONTENTION

Halim Amer, *Yanfei Guo, Kenneth J Raffenetti, Min Si, Pavan Balaji*
Argonne National Laboratory

Charles Archer, Michael Blocksome, Chongxiao Cao, Michael Chuvelev, Hajime Fujita, Jeff R Hammond, Mikhail Shiryaev, Sagar Thapaliya, Maria Garzaran
Intel Corporation

Shintaro Iwasaki, Kenjiro Taura
The University of Tokyo

HYBRID MPI + THREADS PROGRAMMING

Fundamentals and Scope

CONTEXT: THREADS --→ MPI --→ NETWORK INTERFACE

Application

Processes

Threads

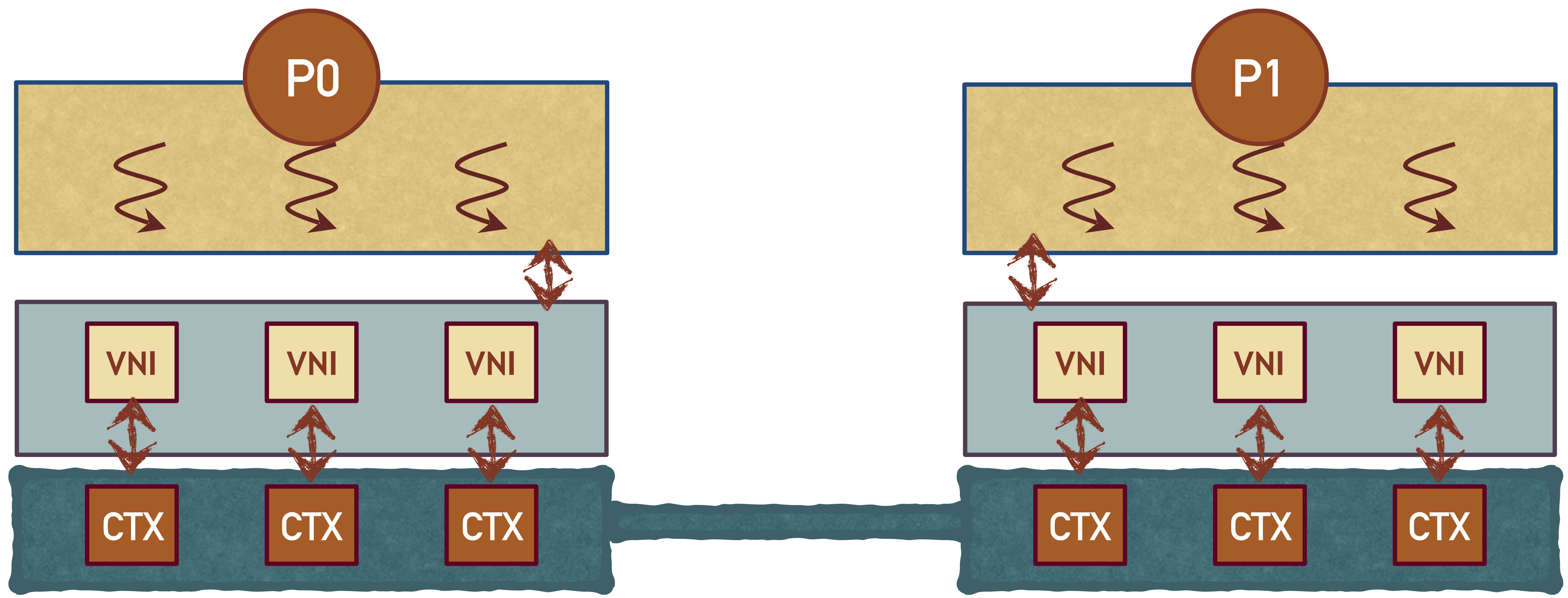
MPI

Virtual Network Interfaces (VNIs)

Network Hardware

Parallel Network Contexts (CTXs)

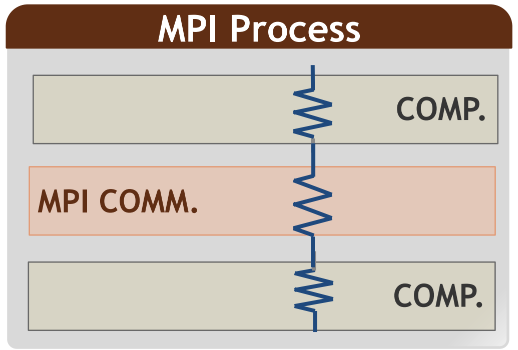
- TCP sockets
- Libfabric endpoints
- UCX workers



APPLICATION THREADS-MPI INTERACTION

MPI_THREAD_SINGLE

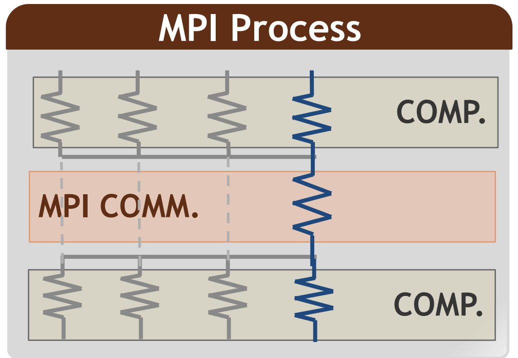
```
for (i=0; i<100; i++)
{
    compute(buf[i]);
    MPI_Send(&buf[i],…);
}
```



MPI_THREAD_FUNNELED

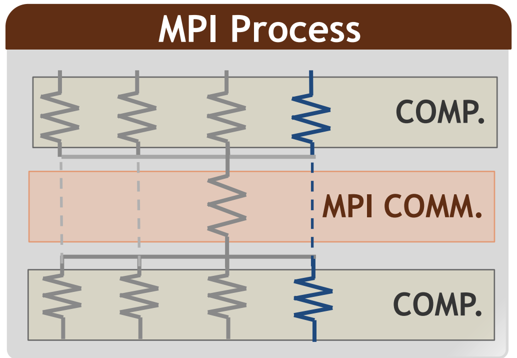
```
#pragma omp parallel for
for (i=0; i<100; i++)
    compute(buf[i]);

MPI_Send(buf,…);
```



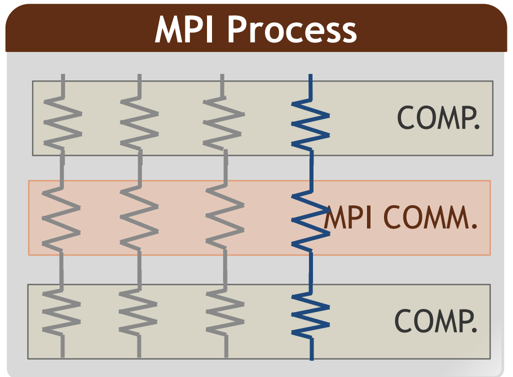
MPI_THREAD_SERIALIZED

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    #pragma omp critical
    MPI_Send(&buf[i],…);
}
```



MPI_THREAD_MULTIPLE

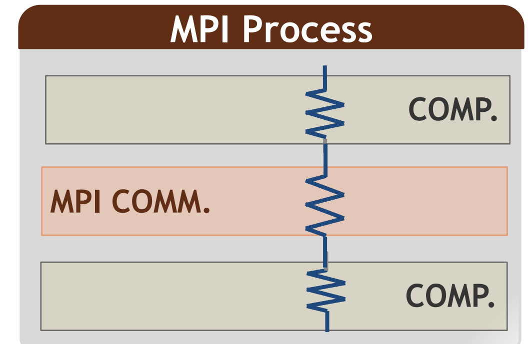
```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Send(&buf[i],…);
}
```



APPLICATION THREADS–MPI INTERACTION

MPI_THREAD_SINGLE

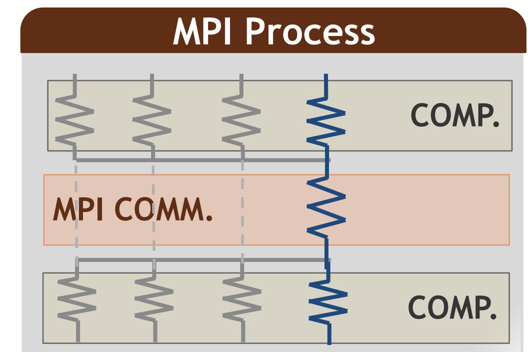
```
for (i=0; i<100; i++)
{
    compute(buf[i]);
    MPI_Send(&buf[i],…);
}
```



MPI_THREAD_FUNNELED

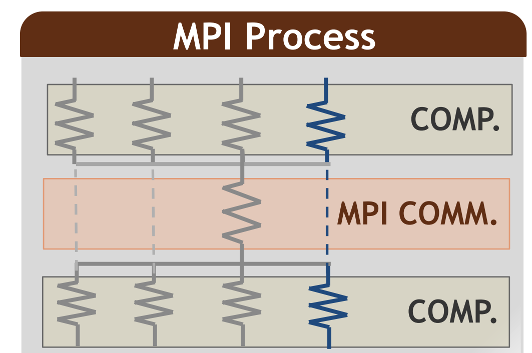
```
#pragma omp parallel for
for (i=0; i<100; i++)
    compute(buf[i]);

MPI_Send(buf,…);
```



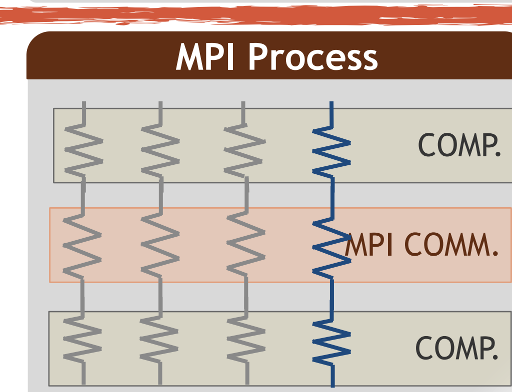
MPI 3.1 requirement
- Thread safety (mutual exclusion)
- Progress (blocking calls only block caller thread)

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    #pragma omp critical
    MPI_Send(&buf[i],…);
}
```

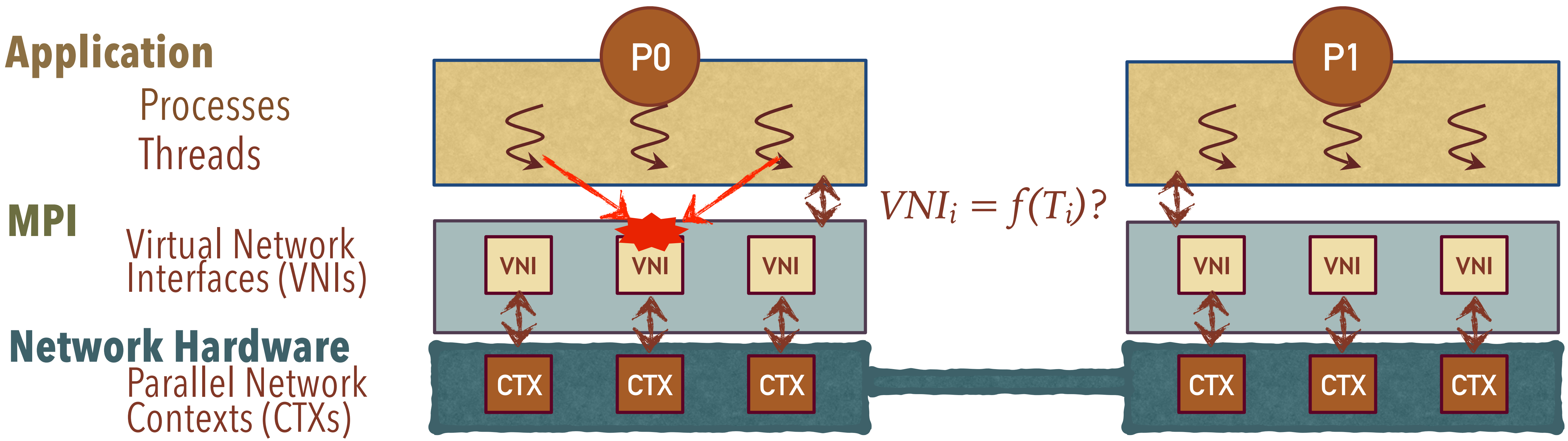


MPI_THREAD_MULTIPLE

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Send(&buf[i],…);
}
```

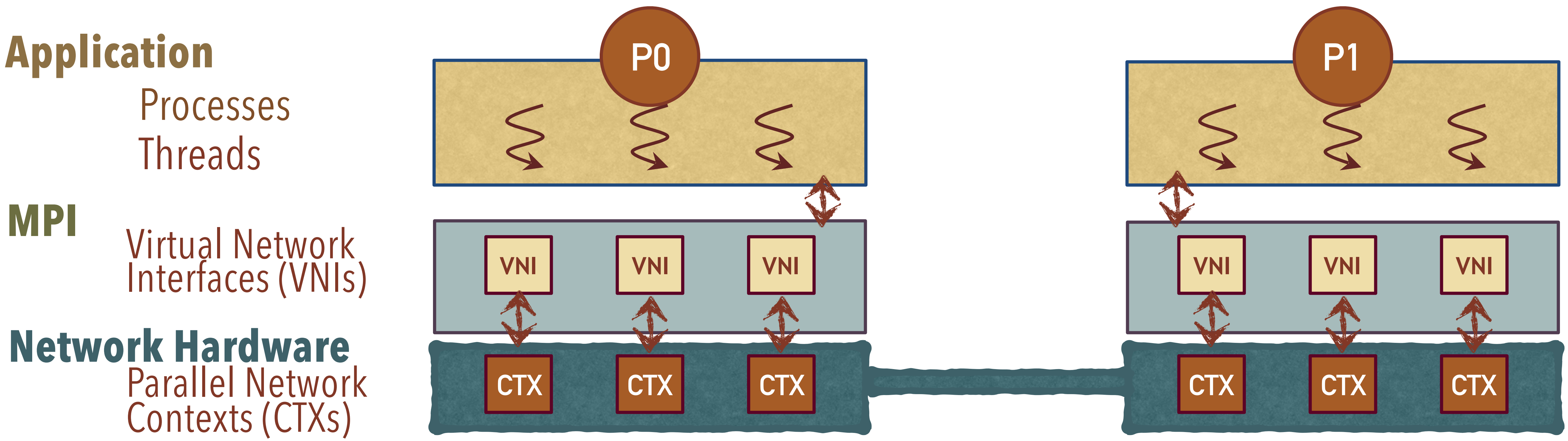


NETWORK RESOURCES: MAJOR HOT SPOT



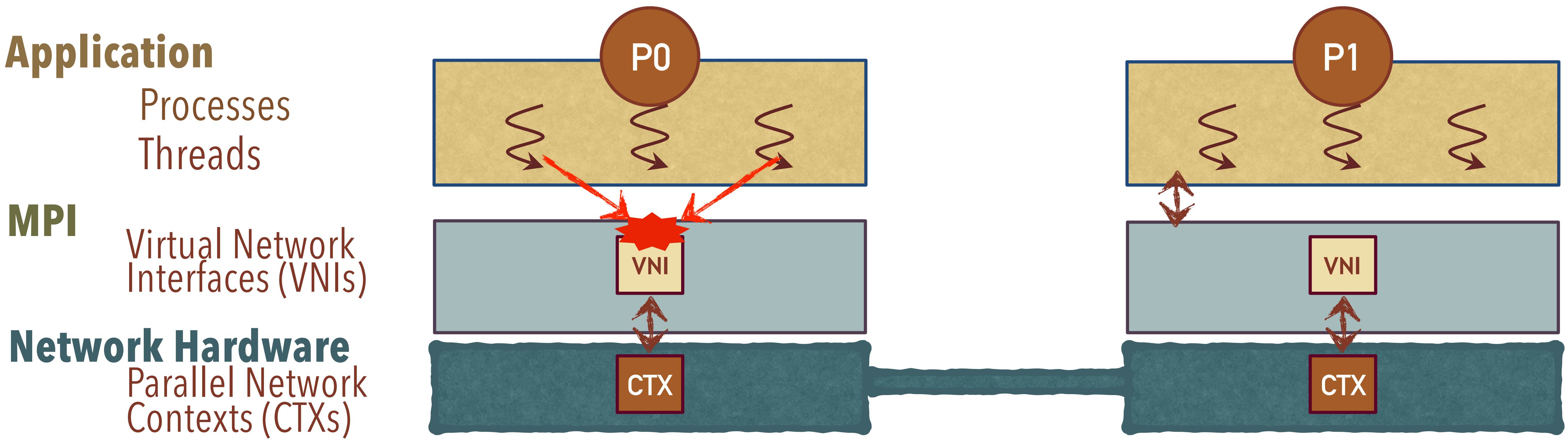
- Not always possible to guarantee independent VNIs for threads
  - Insufficient network resources (multiplexing necessary)
  - Application constraints (e.g., load balancing network traffic across communicators, tags, etc.)
  - Lack of user control over thread-VNI mapping
    - Current MPI libraries best effort mapping (blindly mapping comms/tags/wins to VNIs)
    - MPI Endpoints still not standard

CHALLENGE: NO CONTROL OVER CONCURRENCY



- Threads belong to the user application, not MPI
- Synchronization algorithms that assume N threads won't work

SIMPLIFICATION: SINGLE VNI FOR ALL THREADS



HISTORY

MPI Thread Safety Models

THREAD SAFETY MODELS

- Abstract models
- Capture how thread safety is managed
  - Critical section granularity
  - Access order (e.g., fairness)
  - Wait on completion synchronization model (e.g., spin on flag, lock acquisition, condition variable, etc.)
- Ideally void of technical details (e.g., locking algorithm, atomic operations, etc.)
- Examples of models
  - Global lock
  - Per-object locking
  - Lockless offloading

MOST BASIC THREAD_MULTIPLE MODEL

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Isend(&buf[i],…, &req[i]);
}
#pragma omp parallel for
for (i=0; i<100; i++)
   MPI_Wait(…, &req[i]);
```

APPLICATION

```
MPI_Isend (…,*req) {

   request_create(req);
   network_isend(…,req);

}
```

NETWORK HARDWARE

```
MPI_Wait (…,*req) {

   while (!completed(req)) {
      network_progress();

   }
   free(req);
   req = REQUEST_NULL;

}
```

MPI

MOST BASIC THREAD_MULTIPLE MODEL

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Isend(&buf[i],…, &req[i]);
}
#pragma omp parallel for
for (i=0; i<100; i++)
   MPI_Wait(…, &req[i]);
```

```
MPI_Isend (…,*req) {

   request_create(req);
   network_isend(…,req);

}
```

```
MPI_Wait (…,*req) {

   while (!completed(req)) {
      network_progress();

   }
   free(req);
   req = REQUEST_NULL;

}
```

- **Not thread-safe. Threads can corrupt**
  1. Hardware network state
  2. User buffers
  3. Request objects
  4. …

MOST BASIC THREAD_MULTIPLE MODEL

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Isend(&buf[i],…, &req[i]);
}
#pragma omp parallel for
for (i=0; i<100; i++)
   MPI_Wait(…, &req[i]);
```

```
MPI_Isend (…,*req) {
   lock_acquire(L);
   request_create(req);
   network_isend(…,req);
   lock_release(L);
}
```

- **Simplest MPI-compliant design**
  - Single API level lock (**L**)
  - Release lock in blocking calls to let other threads progress

```
MPI_Wait (…,*req) {
   lock_acquire(L);
   while (!completed(req)) {
      network_progress();
      if (!completed(req)) {
         lock_release(L);
         /*pause/yield*/
         lock_acquire(L);
      }
   }
   free(req);
   req = REQUEST_NULL;
   lock_acquire(L);
}
```

MOST BASIC THREAD_MULTIPLE MODEL

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Isend(&buf[i],…, &req[i]);
}
#pragma omp parallel for
for (i=0; i<100; i++)
   MPI_Wait(…, &req[i]);
```

```
MPI_Isend (…,*req) {
   lock_acquire(L);
   request_create(req);
   network_isend(…,req);
   lock_release(L);
}
```

- **Simplest MPI-compliant design**
  - Single API level lock (**L**)
  - Release lock in blocking calls to let other threads progress
- **Value**
  - Simplicity (less error prone, easy to maintain)
  - Low overheads under zero contention
- **Drawbacks**
  - No internal concurrency
  - Prone to serialization and contention
  - **Lack of asynchrony due to lock acquisitions**

```
MPI_Wait (…,*req) {
   lock_acquire(L);
   while (!completed(req)) {
      network_progress();
      if (!completed(req)) {
         lock_release(L);
         /*pause/yield*/
         lock_acquire(L);
      }
   }
   free(req);
   req = REQUEST_NULL;
   lock_acquire(L);
}
```

FINE-GRAINED LOCKING MODELS

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Isend(&buf[i],…, &req[i]);
}
#pragma omp parallel for
for (i=0; i<100; i++)
   MPI_Wait(…, &req[i]);
```

- **Eliminate the coarse-grained global lock**
  - Independent objects ⇢ separate critical sections
  - Single lock, per-object locks, locks per class of objects, etc.
- **Value**
  - More internal concurrency ⇢ less serialization/contention
- **Drawbacks**
  - Complexity and overheads grow with the number of critical sections
  - Hot spots still possible (all threads may funnel traffic through same VNI)
  - **Lack of asynchrony due to lock acquisitions**
- **Instances:** Dózsa et al. [1], Balaji et al. [2], Kandalla et al. [3]

```
MPI_Isend (…,*req) {
   lock_acquire(req_L);
   request_create(req);
   lock_release(req_L);
   lock_acquire(net_L);
   network_isend(…,req);
   lock_release(net_L);
}
```

```
MPI_Wait (…,*req) {
   while (!completed(req)) {
      lock_acquire(net_L);
      network_progress();
      lock_release(net_L);
      /*pause/yield*/
   }
   lock_acquire(req_L);
   free(req);
   req = REQUEST_NULL;
   lock_acquire(req_L);
}
```

[1] Gábor Dózsa et al. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. (EuroMPI'10)
[2] Pavan Balaji et al. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. IJHPCA (2010)
[3] Krishna Kandalla et al. Optimizing Cray MPI and SHMEM Software Stacks for Cray-XC Supercomputers based on Intel KNL Processors. Cray User Group (2016).

CONTENTION MANAGEMENT MODELS

```
#pragma omp parallel for
for (i=0; i<100; i++) {
    compute(buf[i]);
    MPI_Isend(&buf[i],…, &req[i]);
}
#pragma omp parallel for
for (i=0; i<100; i++)
   MPI_Wait(…, &req[i]);
```

- **Advanced critical section management on contention**
  - Orthogonal to critical section granularity
  - Goal: maximize work inside critical sections
  - Example: O(1) instead of O(N) blind wakeup
- **Value**
  - No added complexity
  - Demonstrated high performance even with coarse-grained locking
- **Drawbacks**
  - Serialization and lack of concurrency
  - **Lack of asynchrony due to lock acquisitions**
- **Instances:** Dang et al [1] and Amer et al. [2]

```
MPI_Isend (…,*req) {
   lock_acquire(L);
   request_create(req);
   network_isend(…,req);
   lock_release(L);
}
```

Just for illustration purposes. Incomplete and incorrect. See Dang et al. [1] for complete algorithm.

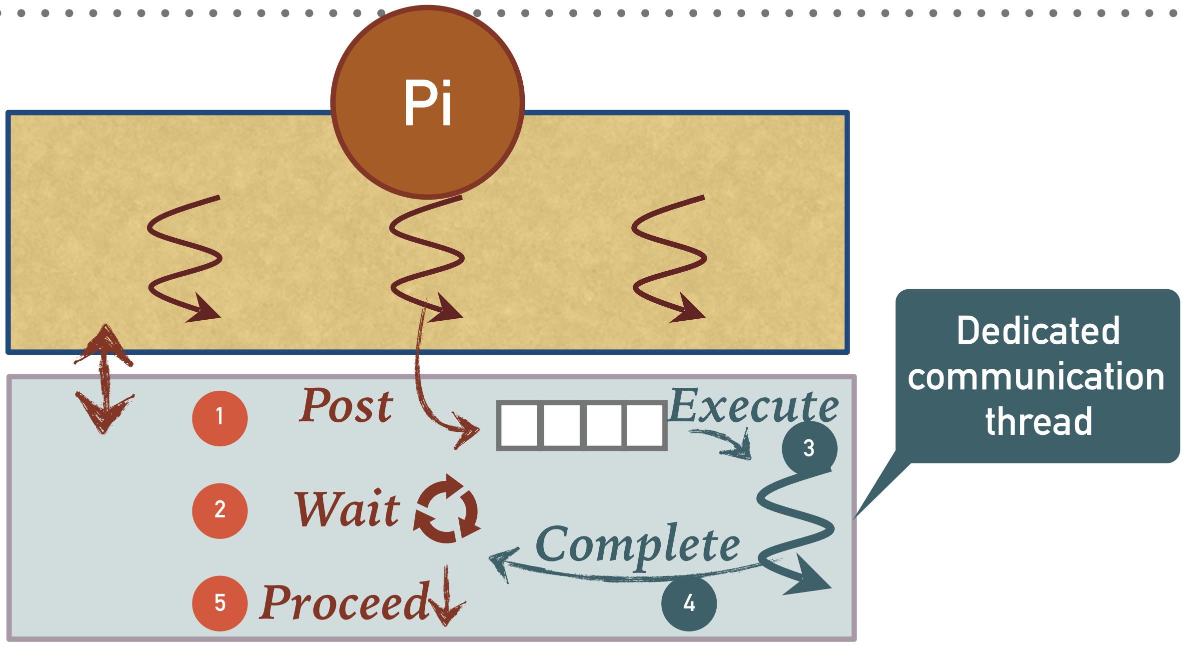
```
MPI_Wait (…,*req) {
   lock_acquire(L);
   if (!completed(req))
      cond_wait(req.c, L);

   while (!completed(req)) {
      req2 = network_cq_poll();
      cond_signal(req2.c);
   }
   free(req);
   req = REQUEST_NULL;
   lock_acquire(L);
}
```

O(1) wakeup

[1] Vu Dang et al. Advanced Thread Synchronization for Multithreaded MPI Implementations. (CCGRID'17)
[2] Abdelhalim Amer et al. Lock Contention Management in Multithreaded MPI. ACM Transactions on Parallel Computing (TOPC) 2019

LOCKLESS OFFLOADING MODEL




```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    descr_create(…,req,&d);
    network_post(…,d);
}
```

- **Offloading to dedicated communication threads**
  - Application threads offload operations to communication threads
  - Lockless: 1) post network operations, 2) wait on a flag on synchronization
  - **Asynchronous progress (more than just a thread safety model)**
- **Value:** Highest possible concurrency
- **Drawbacks**
  - Must sacrifice CPU resources
  - Forces enqueue operation even with zero contention
- **Instances:** Kumar et al. [1], Vaidyanathan et al. [2]

Used as upper bound on performance under contention

```
MPI_Wait (…,*req) {

    while (!completed(req)) {
        /* spin on the
           Request local flag*/
    }
    lock_acquire(req_L);
    free(req);
    req = REQUEST_NULL;
    lock_acquire(req_L);
}
```

[1] Sameer Kumar et al. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. (IPDPS '12).
[2] Karthikeyan Vaidyanathan et al. Improving Concurrency and Asynchrony in Multithreaded MPI Applications Using Software Offloading. (SC'15)

HISTORICAL SUMMARY

		Fine-Grained Locking	Lock Contention Management	Offloading
Nonblocking Operations	No Contention	Overhead and complexity **grows with the number of critical sections**	**Simplest and lowest overhead**	**High offloading overhead**
	High Contention	Performance improvements from **increased concurrency**	High performance from **high throughput locks**	**High performance proportional to queue efficiency**
Waiting in Blocking Operations	No Contention	Overhead and complexity **grows with the number of critical sections**	**Low overhead**	**Lowest overhead** (only check local flag)
	High Contention	Bad performance from **blind lock ownership passing**	High performance **O(1) wakeup**. Overhead of **progress calls**	**Lowest overhead** (only check local flag, no progress calls)
Asynchrony of Nonblocking Calls		**May block on lock acquisition**	**May block on lock acquisition**	**Asynchronous**
CPU Resource Consumption		**Nothing special**	**Nothing special**	**CPU resources grow with the number of dedicated threads**
Hardware Awareness		**Can be Agnostic**	**Necessary for high throughput locks**	**Can be agnostic**

NEW SOFTWARE COMBINING MODELS (1/2)

Software Combining

		Fine-Grained Locking	Lock Contention Management	Offloading	CSync
Nonblocking Operations	No Contention	Overhead and complexity **grows with the number of critical sections**	**Simplest and lowest overhead**	**High offloading overhead**	**High offloading overhead**
	High Contention	Performance improvements from **increased concurrency**	High performance from **high throughput locks**	**High performance proportional to queue efficiency**	**High performance proportional to queue efficiency**
Waiting in Blocking Operations	No Contention	Overhead and complexity **grows with the number of critical sections**	**Low overhead**	**Lowest overhead** (only check local flag)	**Low overhead**
	High Contention	Bad performance from **blind lock ownership passing**	High performance **O(1) wakeup**. Overhead of **progress calls**	**Lowest overhead** (only check local flag, no progress calls)	**Wasteful**
Asynchrony of Nonblocking Calls		**May block on lock acquisition**	**May block on lock acquisition**	**Asynchronous**	**May block on pending operation or lock acquisition**
CPU Resource Consumption		**Nothing special**	**Nothing special**	**CPU resources grow with the number of dedicated threads**	**Nothing special**
Hardware Awareness		**Can be Agnostic**	**Necessary for high throughput locks**	**Can be agnostic**	**Can be agnostic**

NEW SOFTWARE COMBINING MODELS (2/2)

Software Combining

		Fine-Grained Locking	Lock Contention Management	Offloading	CSync	LockQ
Nonblocking Operations	No Contention	Overhead and complexity **grows with the number of critical sections**	**Simplest and lowest overhead**	**High offloading overhead**	**High offloading overhead**	**Simple and low overhead**
	High Contention	Performance improvements from **increased concurrency**	High performance from **high throughput locks**	**High performance proportional to queue efficiency**	**High performance proportional to queue efficiency**	**High performance proportional to queue efficiency**
Waiting in Blocking Operations	No Contention	Overhead and complexity **grows with the number of critical sections**	**Low overhead**	**Lowest overhead** (only check local flag)	**Low overhead**	**Low overhead**
	High Contention	Bad performance from **blind lock ownership passing**	High performance **O(1) wakeup**. Overhead of **progress calls**	**Lowest overhead** (only check local flag, no progress calls)	**Wasteful**	**Not wasteful but unsatisfactory**
Asynchrony of Nonblocking Calls		**May block on lock acquisition**	**May block on lock acquisition**	**Asynchronous**	**May block on pending operation or lock acquisition**	**Asynchronous**
CPU Resource Consumption		**Nothing special**	**Nothing special**	**CPU resources grow with the number of dedicated threads**	**Nothing special**	**Nothing special**
Hardware Awareness		**Can be Agnostic**	**Necessary for high throughput locks**	**Can be agnostic**	**Can be agnostic**	**Can be agnostic**

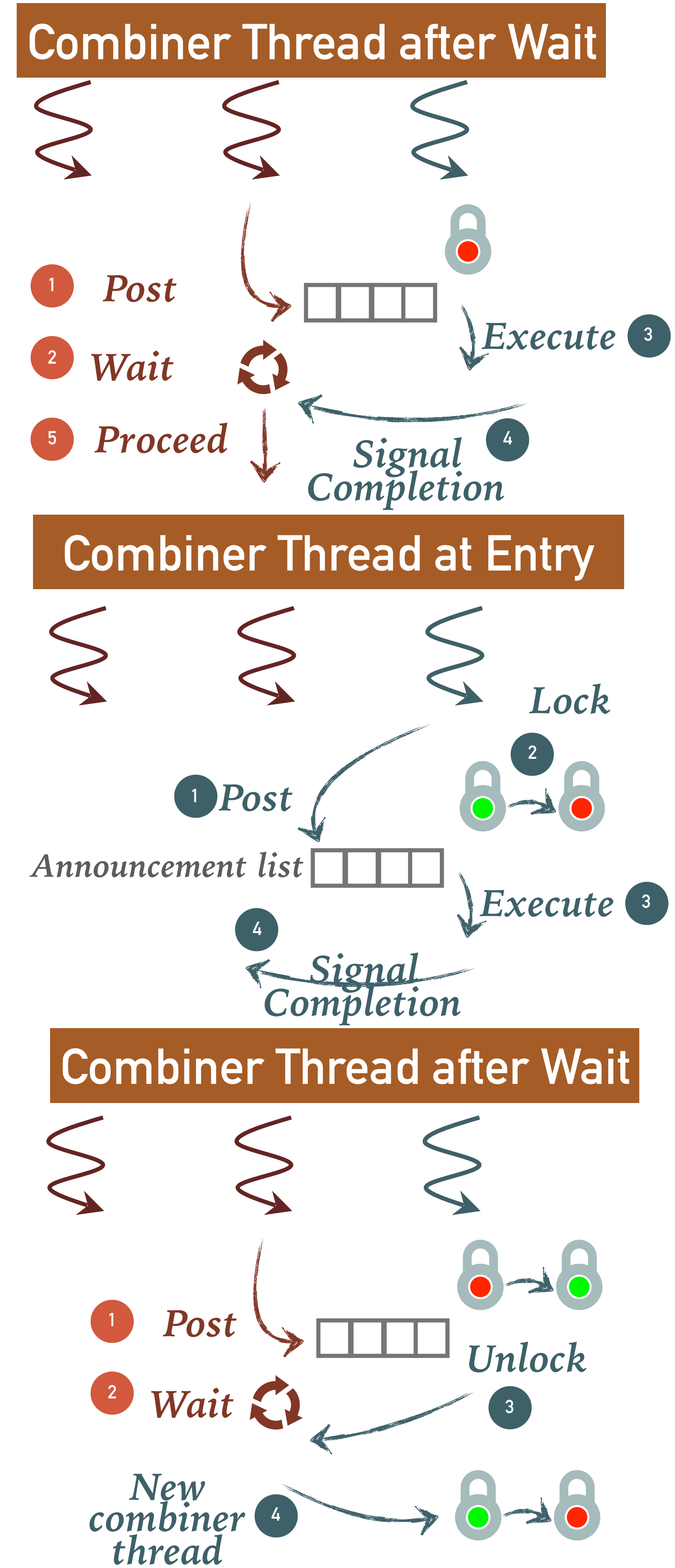
SOFTWARE COMBINING

Description and Example

SOFTWARE COMBINING

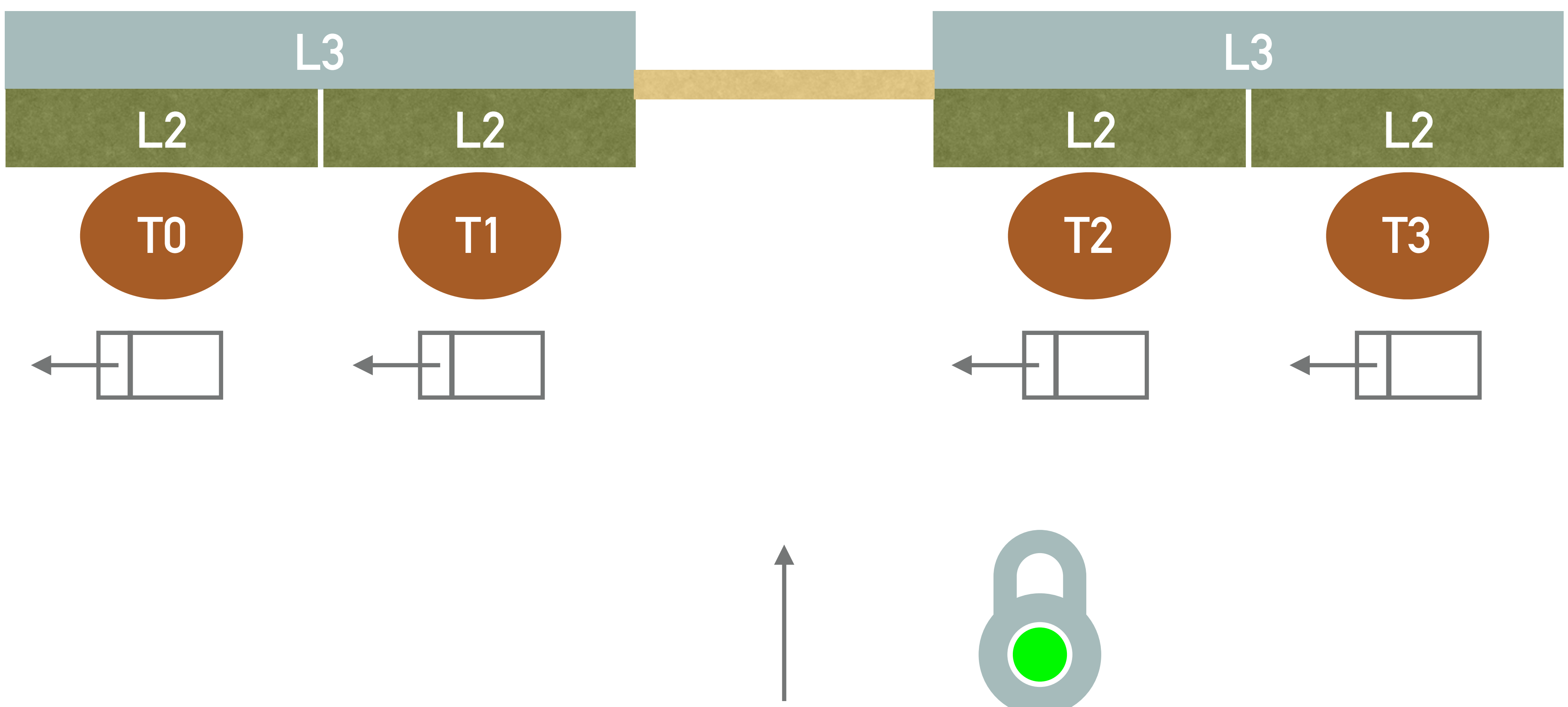
- Goal: **scalability**
- Principle

  - Lock + announcement list
  - Waiters announce their work requests
  - Lock owner combines them (executes on behalf of waiters)
- Most implementations are **hardware agnostic**
- Several implementations
- Many scalable applications especially for concurrent data structures
  - Lists, queues, stacks, etc.



DSM–SYNCH: EXTENDING MCS WITH COMBINING

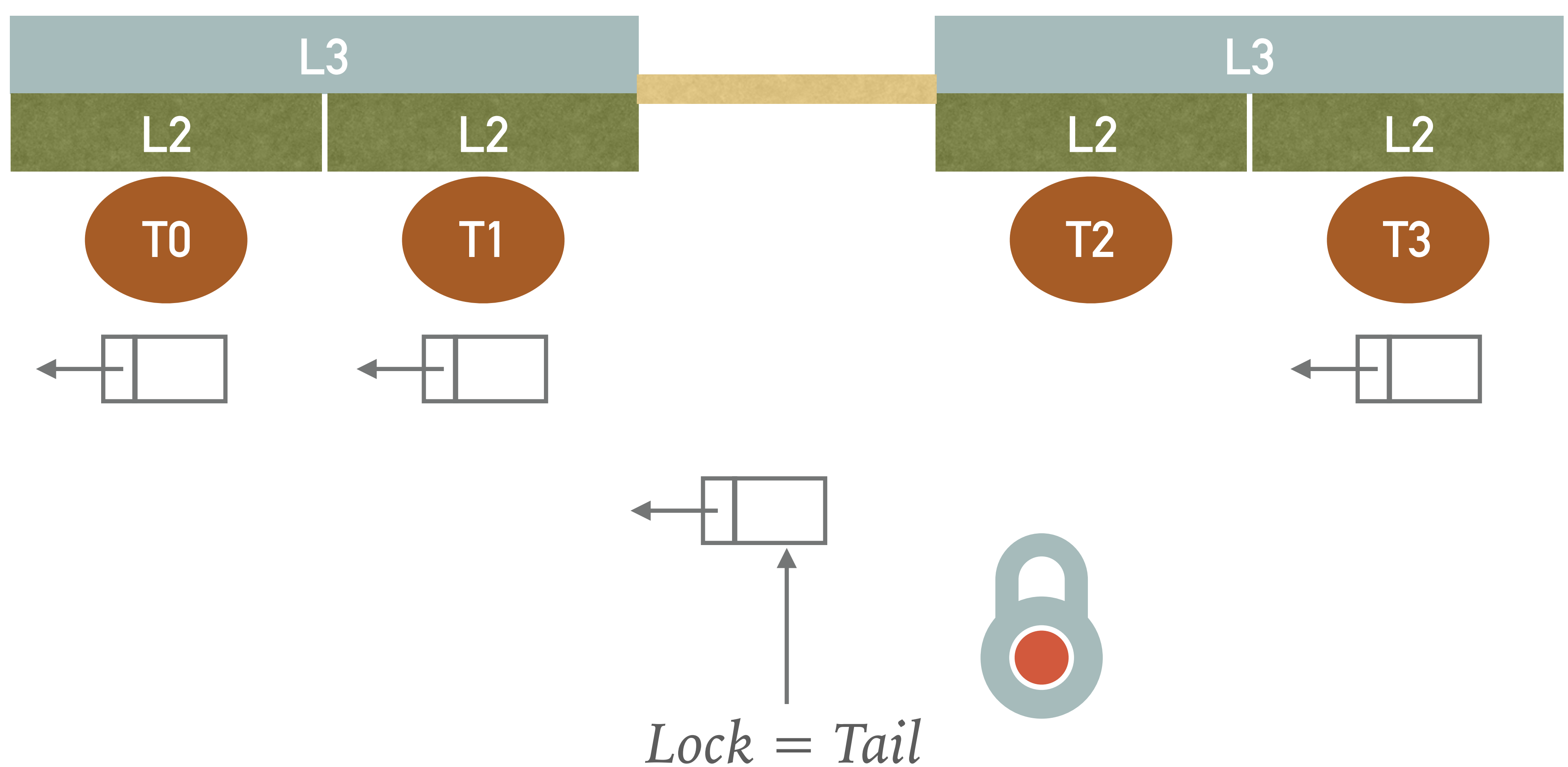
- Probably the most popular mutual exclusion algorithm (over 1.5K citations!)
- Mellor-Crummey & Scott (1991): "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems.*
- Queue-based lock algorithm



$Lock = Tail$

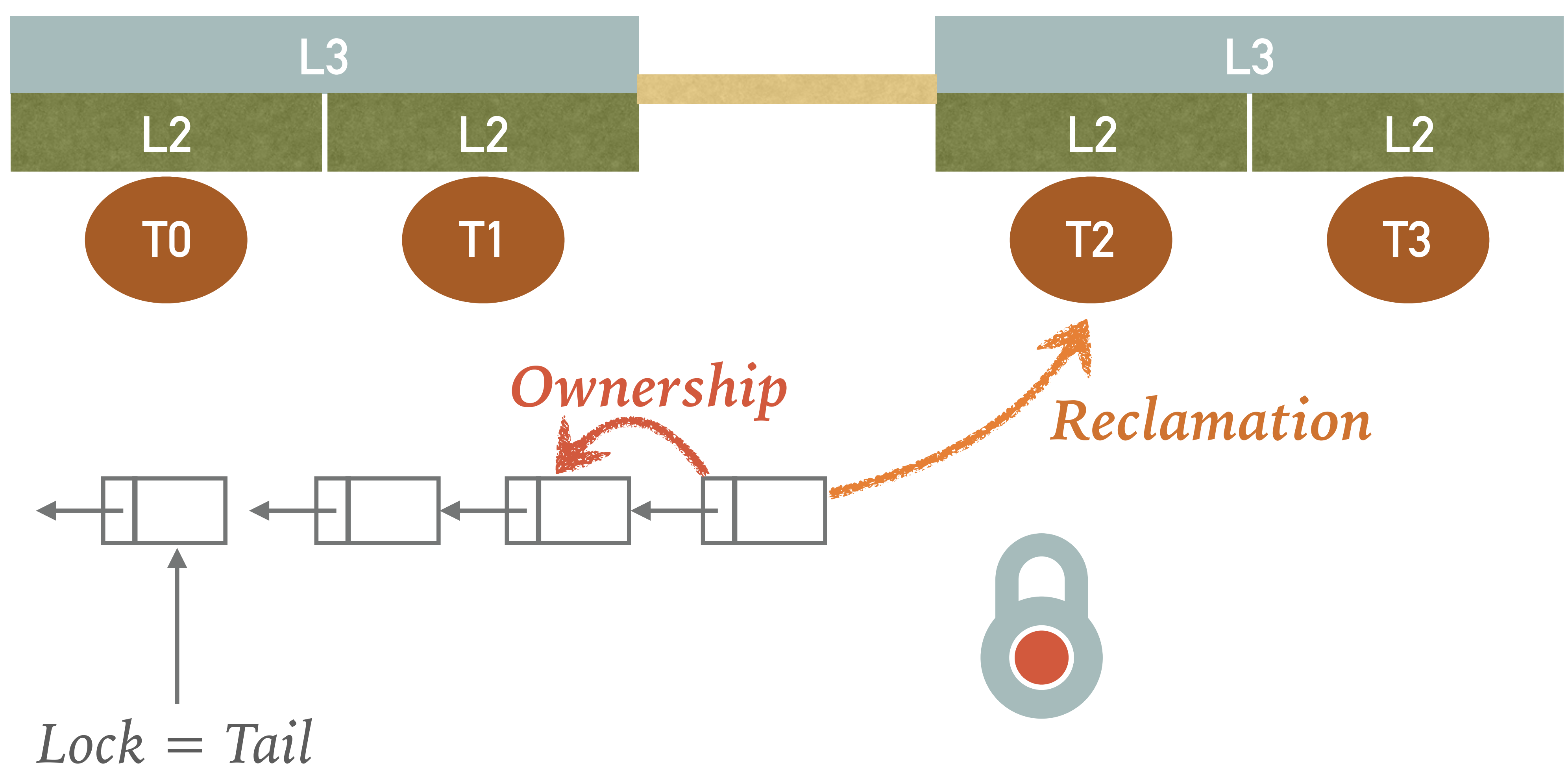
DSM-SYNCH: EXTENDING MCS WITH COMBINING

- Probably the most popular mutual exclusion algorithm (over 1.5K citations!)
- Mellor-Crummey & Scott (1991): "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems.*
- Queue-based lock algorithm



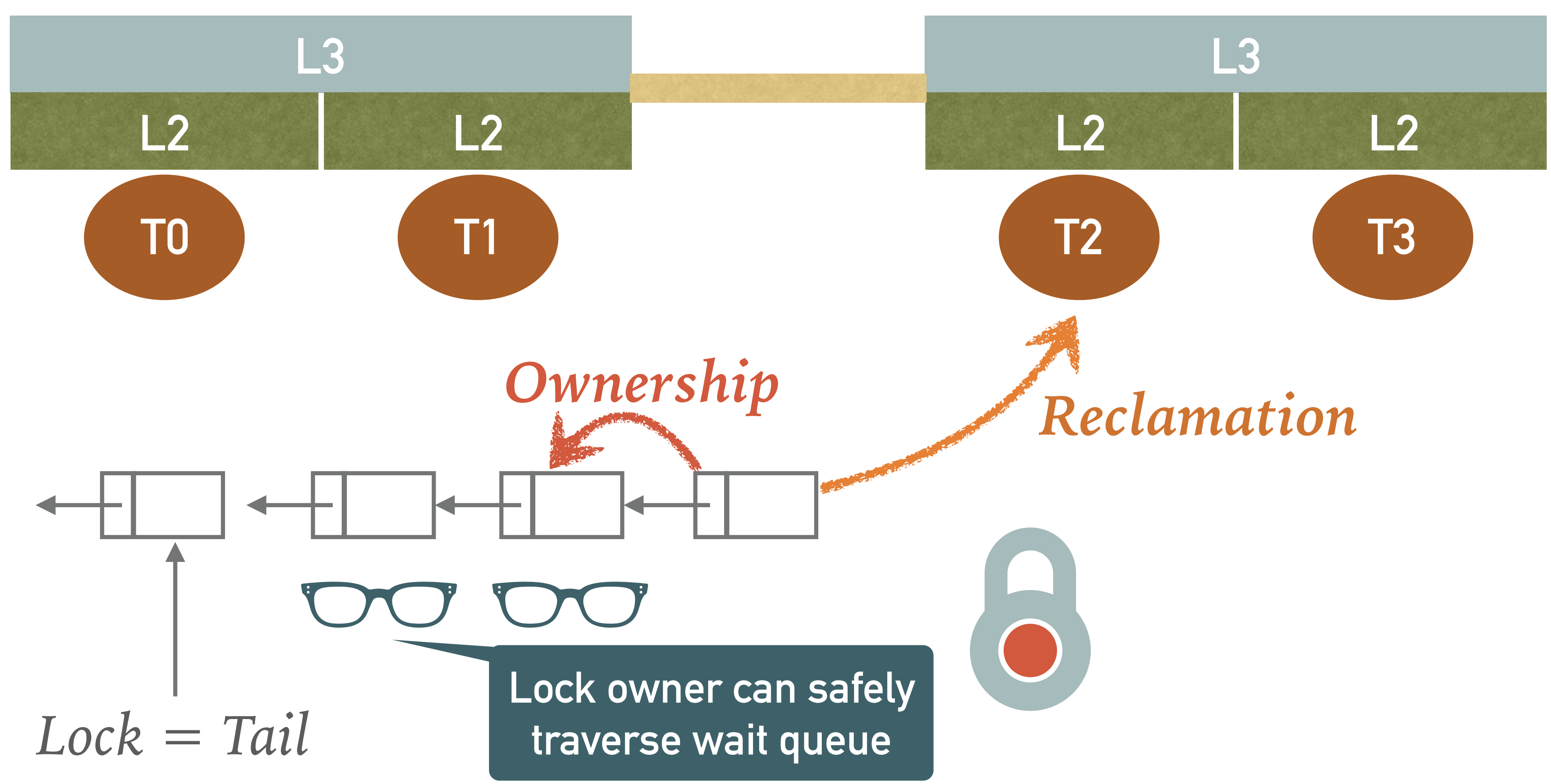
DSM–SYNCH: EXTENDING MCS WITH COMBINING

- Probably the most popular mutual exclusion algorithm (over 1.5K citations!)
- Mellor-Crummey & Scott (1991): "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems.*
- Queue-based lock algorithm



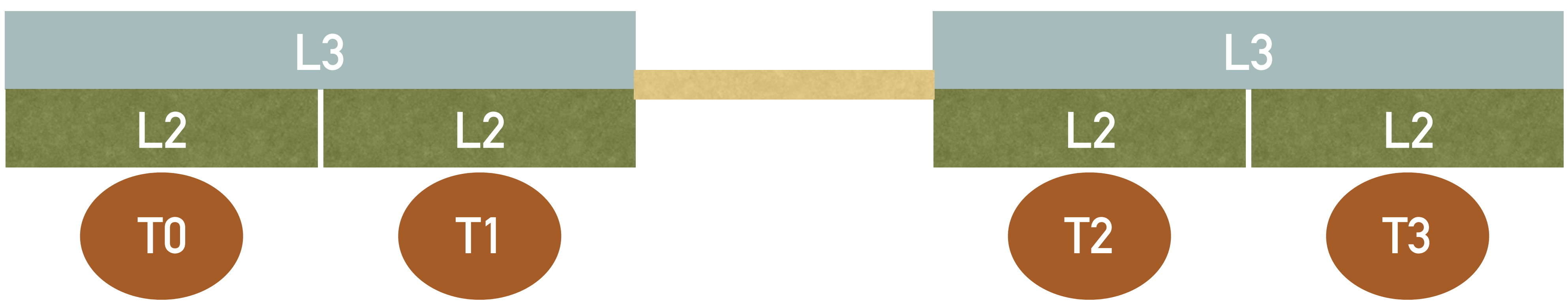
DSM–SYNCH: EXTENDING MCS WITH COMBINING

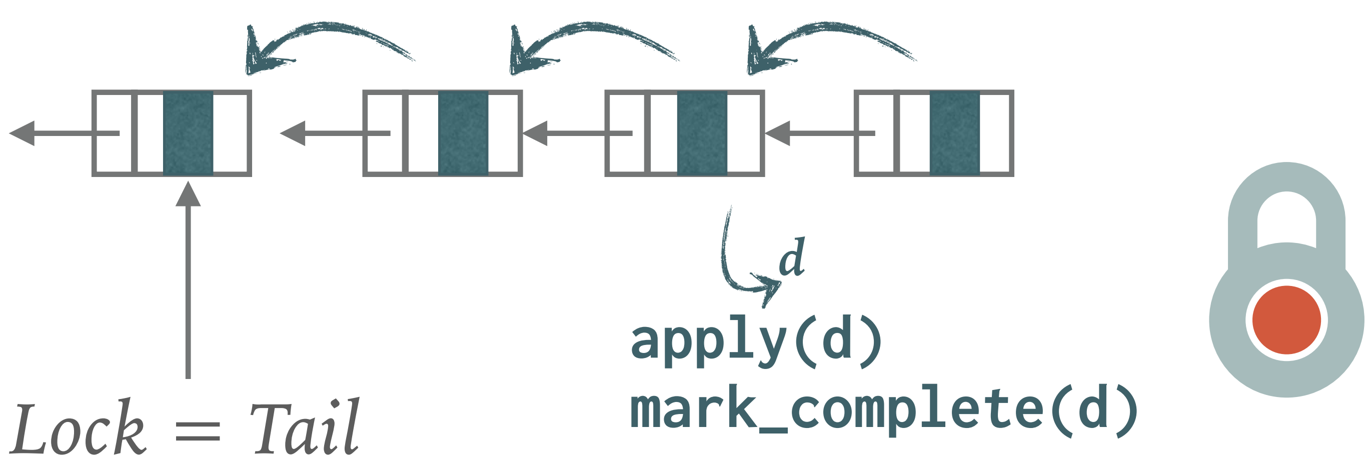
- Probably the most popular mutual exclusion algorithm (over 1.5K citations!)
- Mellor-Crummey & Scott (1991): "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems.*
- Queue-based lock algorithm



DSM–SYNCH: EXTENDING MCS WITH COMBINING

- Extend an MCS queue node with a **request** or **work descriptor**
- Wait queue has two purposes
  1. Holds waiting thread nodes as in MCS
  2. Works as an **announcement list** to publish work descriptors
- Lock owner executes operations found in the queue

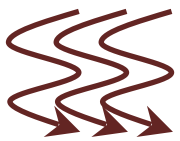
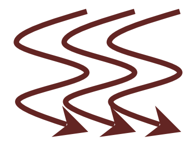




Fatourou, Panagiota, and Nikolaos D. Kallimanis. "Revisiting the combining synchronization technique." In *ACM SIGPLAN Notices (PPoPP'12)*

PERFORMANCE EXAMPLE WITH A FIFO QUEUE

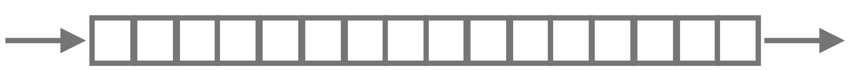
Enqueue *Dequeue*

`mcs_acquire()` `mcs_acquire()`

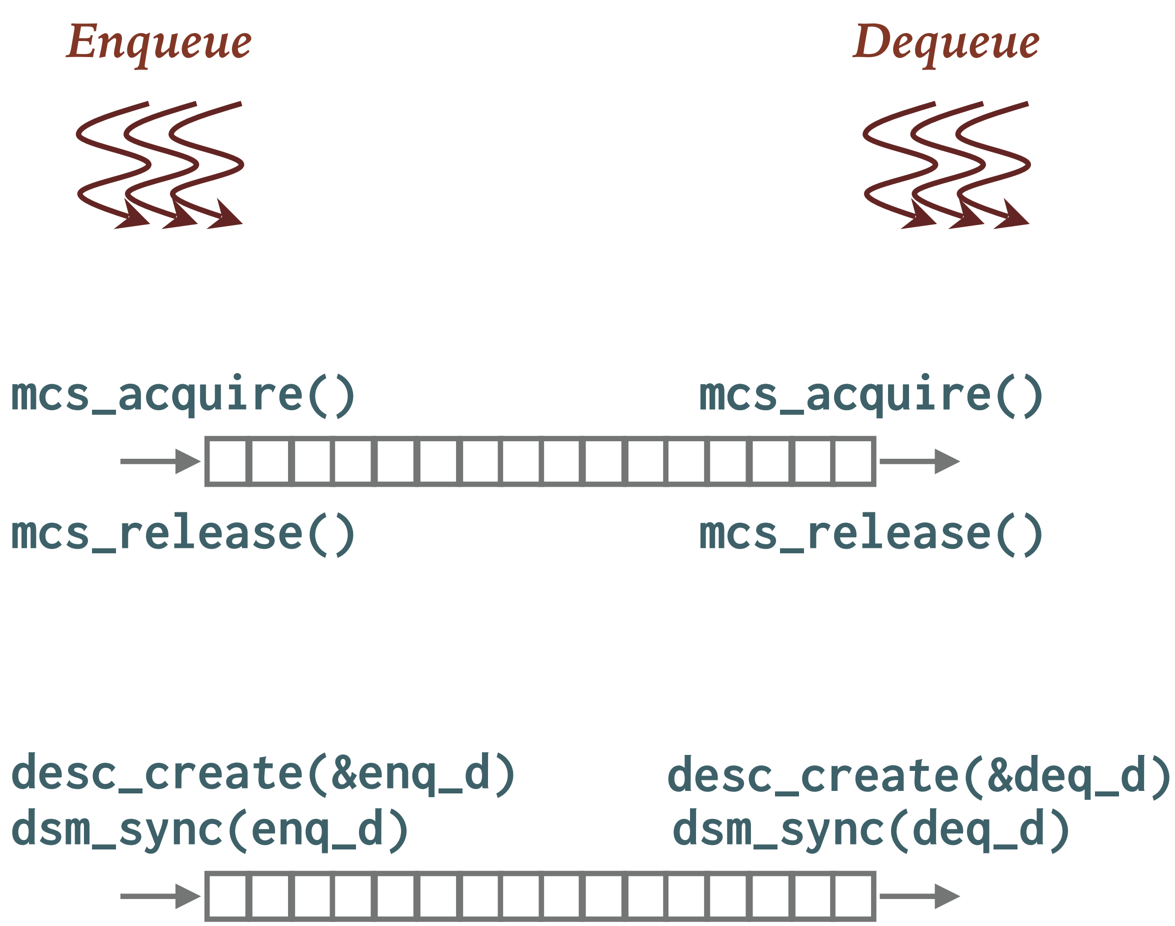
`mcs_release()` `mcs_release()`

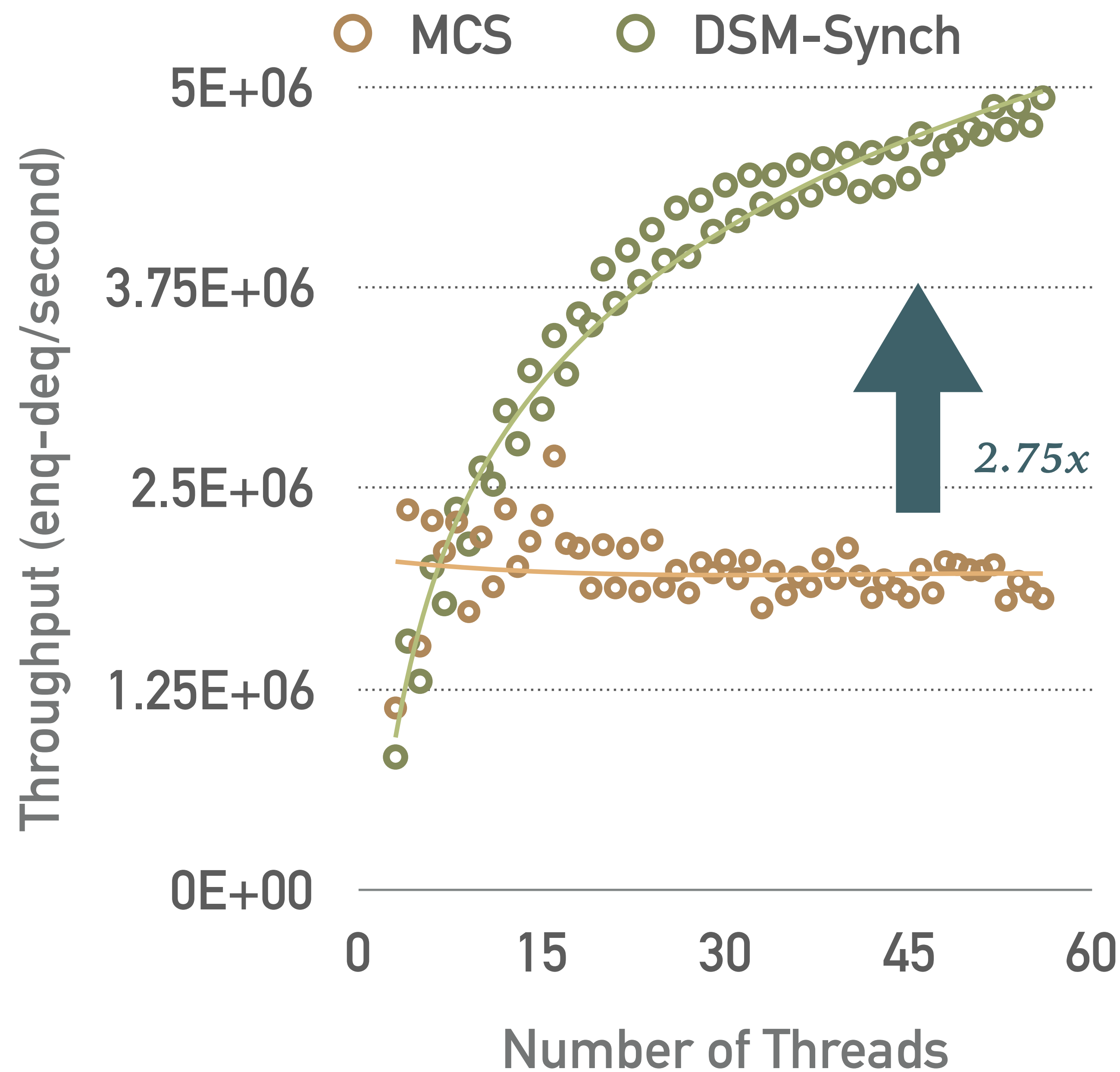
`desc_create(&enq_d)` `desc_create(&deq_d)`
`dsm_sync(enq_d)` `dsm_sync(deq_d)`



Fatourou, Panagiota, and Nikolaos D. Kallimanis. "Revisiting the combining synchronization technique." In *ACM SIGPLAN Notices (PPoPP'12)*

PERFORMANCE EXAMPLE WITH A FIFO QUEUE





Enq/Deq throughput on 56-Core Intel Skylake at 2.5GHz

Fatourou, Panagiota, and Nikolaos D. Kallimanis. "Revisiting the combining synchronization technique." In *ACM SIGPLAN Notices (PPoPP'12)*

CSYNC AND LOCKQ MODELS

Software Combining at the Rescue

CSYNC: DIRECT SOFTWARE COMBINING APPLICATION

```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    lock_acquire(net_L);
    network_isend(…,req);
    lock_release(net_L);
}
```

```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    descr_create(ISEND,…,&d);
    dsm_synch(net_L, d);
}
```

```
MPI_Wait (…,*req) {
    while (!completed(req))
    {
        lock_acquire(net_L);
        network_progress();
        lock_release(net_L);
        /*pause/yield*/
    }
    lock_acquire(req_L);
    free(req);
    req = REQUEST_NULL;
    lock_acquire(req_L);
}
```

```
MPI_Wait (…,*req) {
    while (!completed(req)) {
        descr_create(PROGRESS,&d);
        dsm_synch(net_L, d);
        /*pause/yield*/
    }
    lock_acquire(req_L);
    free(req);
    req = REQUEST_NULL;
    lock_acquire(req_L);
}
```

CSYNC: DIRECT SOFTWARE COMBINING APPLICATION

```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    lock_acquire(net_L);
    network_isend(…,req);
    lock_release(net_L);
}

MPI_Wait (…,*req) {
    while (!completed(req))
    {
        lock_acquire(net_L);
        network_progress();
        lock_release(net_L);
        /*pause/yield*/
    }
    lock_acquire(req_L);
    free(req);
    req = REQUEST_NULL;
    lock_acquire(req_L);
}
```

```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    descr_create(ISEND,…,&d);
    dsm_synch(net_L, d);
}

MPI_Wait (…,*req) {
    while (!completed(req)) {
        descr_create(PROGRESS,&d);
        dsm_synch(net_L, d);
        /*pause/yield*/
    }
    lock_acquire(req_L);
    free(req);
    req = REQUEST_NULL;
    lock_acquire(req_L);
}
```

```
apply (*d) {
    switch(d->op) {
    case ISEND:
        network_isend(…);
    case PROGRESS:
        network_progress(…);
    }
}
```

Combiner thread internal call

LIMITATIONS OF CSYNC

2 *Unbounded waiting wastes asynchrony of nonblocking calls*

1 *Software offloading even under no contention (descriptor creation + enq/deq overhead)*

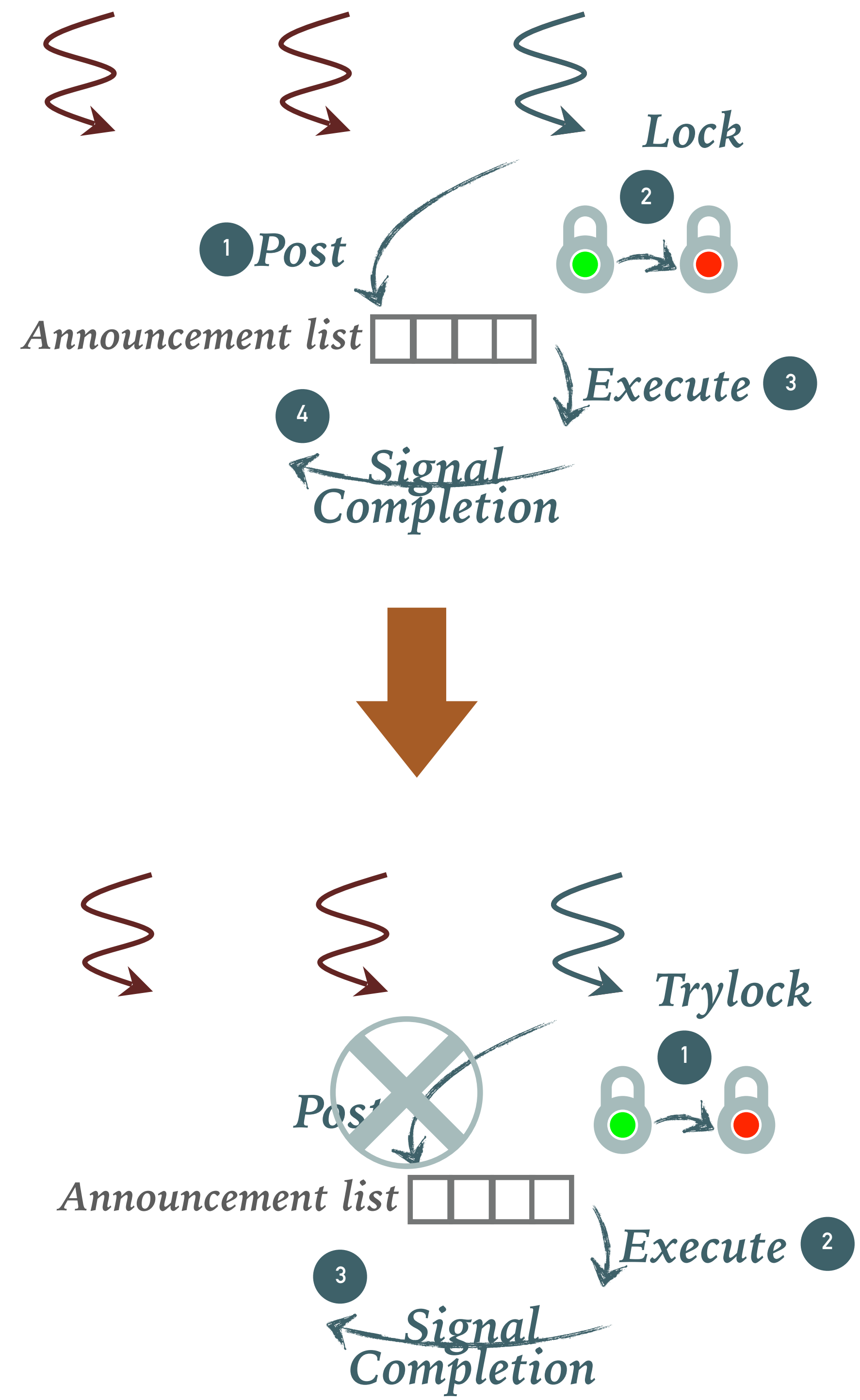
```
MPI_Isend (…,*req) {
   lock_acquire(req_L);
   request_create(req);
   lock_release(req_L);
   descr_create(ISEND,…,&d);
   dsm_synch(net_L, d);
}
```

3 *Combining queue polluted by progress calls (low priority operation)*

```
MPI_Wait (…,*req) {
   while (!completed(req)) {
      descr_create(PROGRESS,&d);
      dsm_synch(net_L, d);
      /*pause/yield*/
   }
   lock_acquire(req_L);
   free(req);
   req = REQUEST_NULL;
   lock_acquire(req_L);
}
```

ELIMINATING UNNECESSARY OFFLOADING

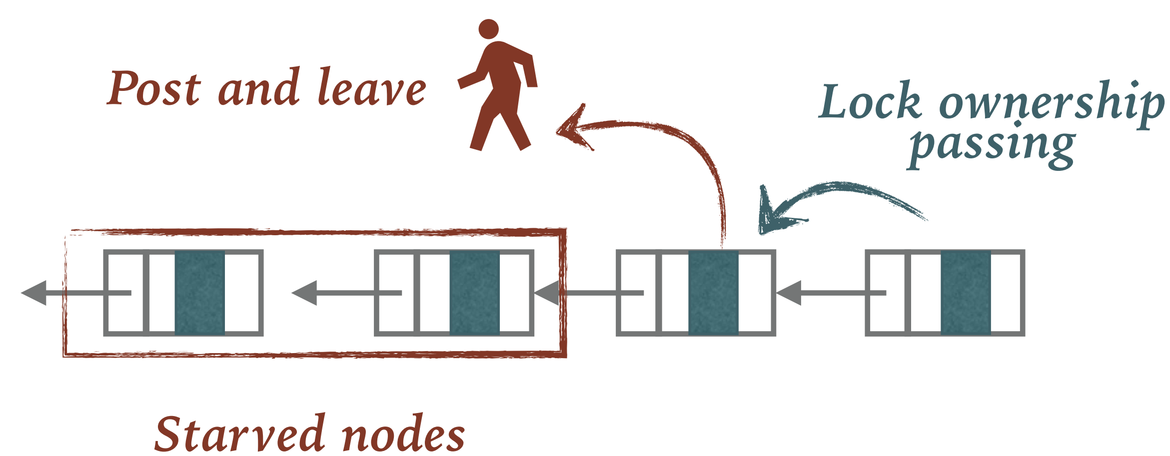
- Don't post a work descriptor unconditionally
- Use an empty node (no descriptor creation)
- Try to acquire the lock
  - If successful
    - Combine operations
    - If threshold of combining reached, enqueue my operation
    - Else execute operation and then leave
  - Lock acquisition failure
    - Post work descriptor
    - Wait



POST AND LEAVE BREAKS THE SYSTEM

- Keeping nonblocking calls asynchronous improves latency hiding and overlapping opportunities
- Only way is to leave after posting a work descriptor on lock acquisition failure
    - Thread gives up lock ownership passing and combining responsibilities
    - Work descriptors and threads may starve in the queue

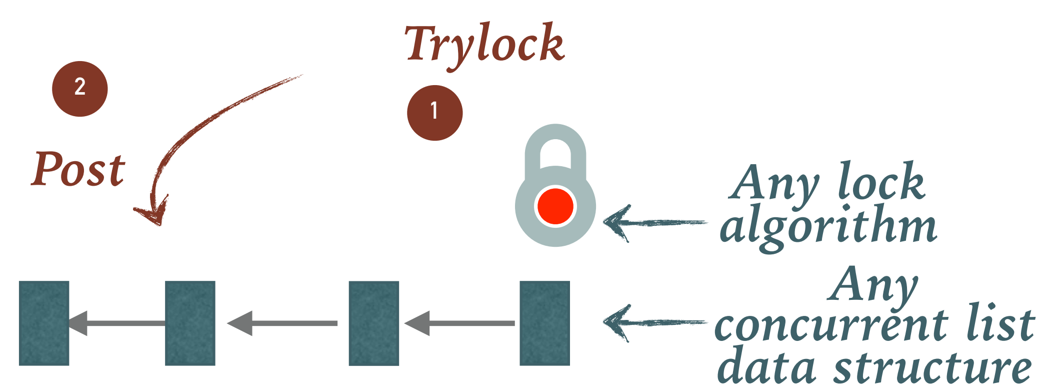
```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    descr_create(ISEND,…,&d);
    dsm_synch(net_L, d);
}
```



SOLUTION: DECOUPLED LOCK-LIST STRUCTURE

- Fundamental issue is coupled lock-list data structure
- Decoupling data structure
  - No waiting necessary in a nonblocking call
  - **Post and leave**, and thus **keep nonblocking calls asynchronous**
- Flexibility
  - Any lock algorithm can be used
  - Any concurrent list data structure can be used

```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    if (trylock(net_L)) {
        combine();
        network_isend(…);
        lock_release(net_L);
    } else {
        descr_create(ISEND,…,&d);
        post(d);
    }
}
```



RACES AND MPI COMPLETION SEMANTICS

- Work descriptors may never be executed due to races
- Solution: rely on MPI completion semantics as last resort
  - Request completion: MPI_Wait and MPI_Test family
  - RMA: synchronization calls (e.g., MPI_Win_flush, MPI_Win_unlock, etc.)

T1 *T2*

```
                                 trylock()        ← Success
                                 combine()
                                 network_isend()
Fail → trylock()
                                 lock_release()
       desc_create(d)            /* Leave */
       post(d)
       /* Leave */
```

d may never be executed

```
MPI_Wait (…,*req) {
    while (!completed(req))
    {
        lock_acquire(net_L);
        combine();
        network_progress();
        lock_release(net_L);
        /*pause/yield*/
    }
    …
}
```

PUTTING THEM TOGETHER: LOCKQ AND DETAILS

- **LockQ**
  - Avoids unnecessary offloading under no contention
  - Keeps nonblocking asynchronous
  - Combing queue is not polluted by progress calls
- **Combining thread doing too much?**
  - User-controllable combining threshold
  - Combining responsibility changes over time
- **How about nonblocking progress calls like MPI_Test?**
  - Asynchronous with trylock
  - Exponential backoff to reduce contention
- **Are nonblocking calls made blocking with last resort combining?**
  - Yes, but rare in practice

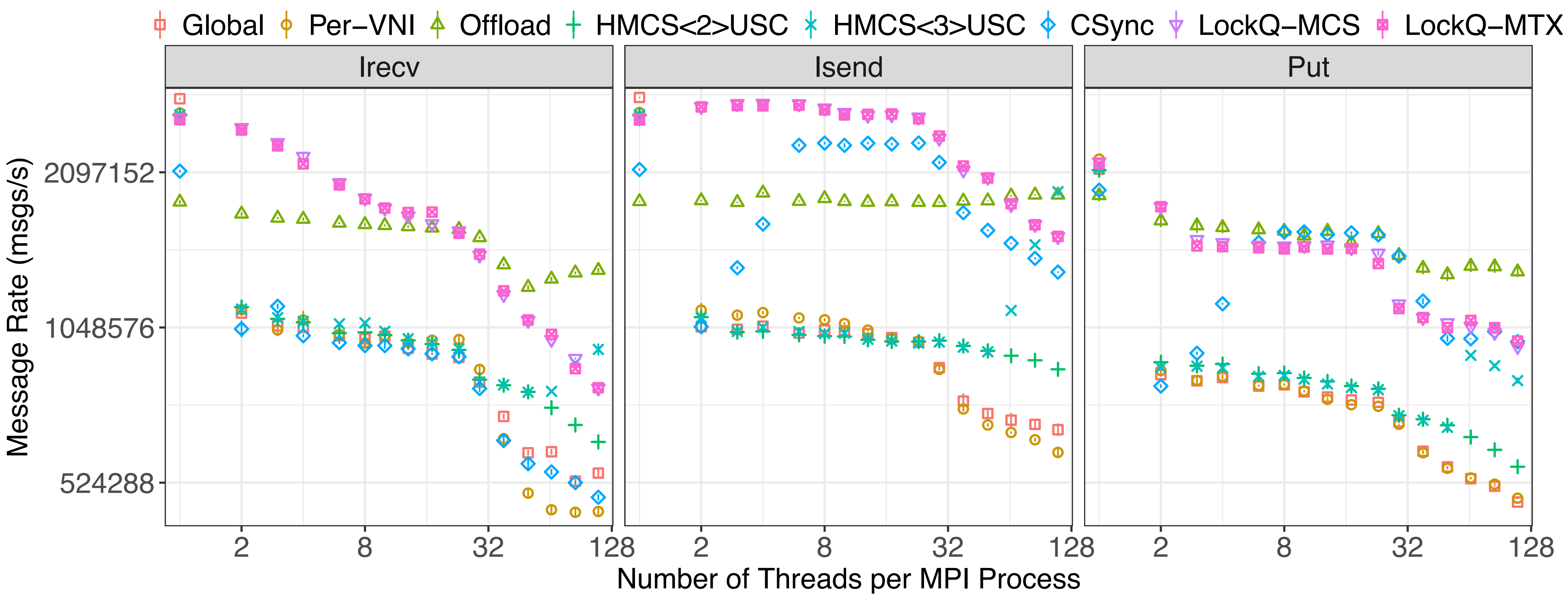
```
MPI_Isend (…,*req) {
    lock_acquire(req_L);
    request_create(req);
    lock_release(req_L);
    if (trylock(net_L)) {
        combine();
        network_isend(…);
        lock_release(net_L);
    } else {
        descr_create(ISEND,…,&d);
        post(d);
    }
}
```

```
MPI_Wait (…,*req) {
    while (!completed(req))
    {
        lock_acquire(net_L);
        combine();
        network_progress();
        lock_release(net_L);
        /*pause/yield*/
    }
    …
}
```

EVALUATION

MESSAGE RATE VS. STATE-OF-THE-ART

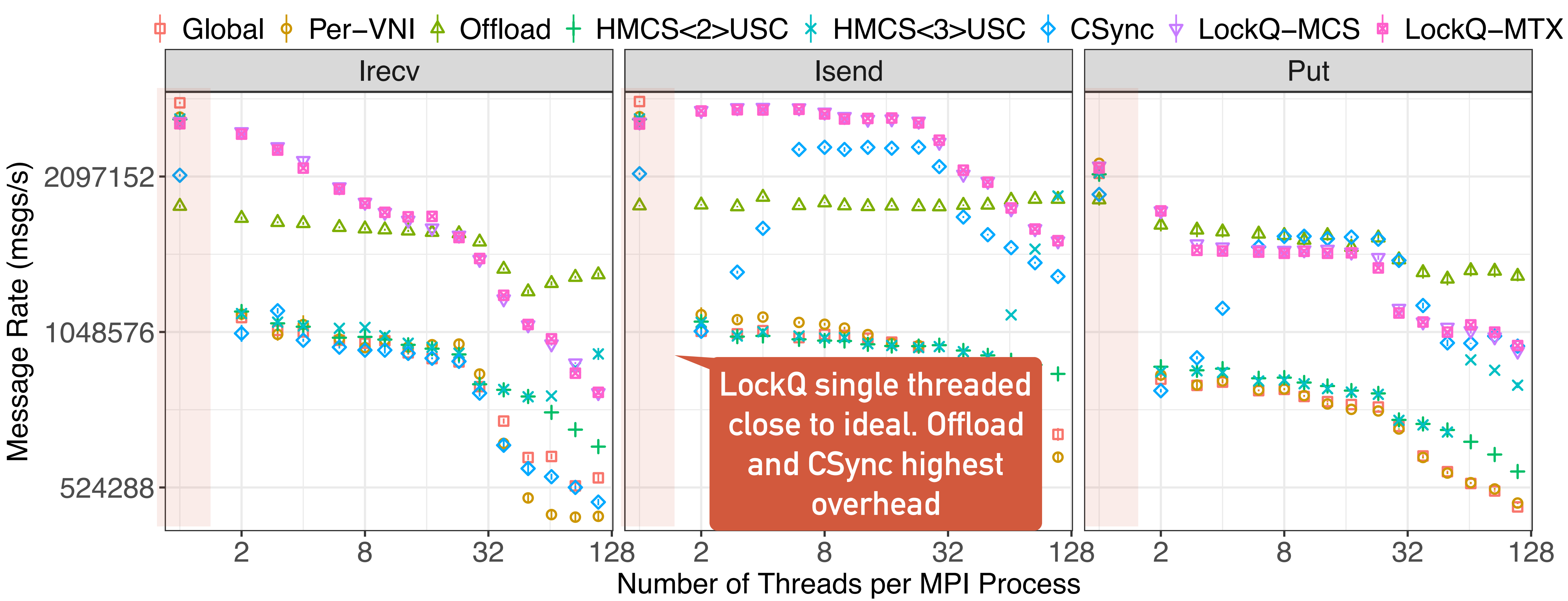
Label	Global	Per-VNI	Offload	HMCS<N>USC	CSync	LockQ-MCS	LockQ-MTX
Description	Global MCS lock	VNI fine-grained locking with MCS	Lockless Software offloading	HMCS lock + O(1) wakeup	VNI granularity with DSM-Synch	VNI granularity with MCS-based LcokQ	VNI granularity with Pthread mutex-based LockQ



64B message rate between two 56-Core Intel Skylake nodes at 2.5GHz over Intel Omnipath

MESSAGE RATE VS. STATE-OF-THE-ART

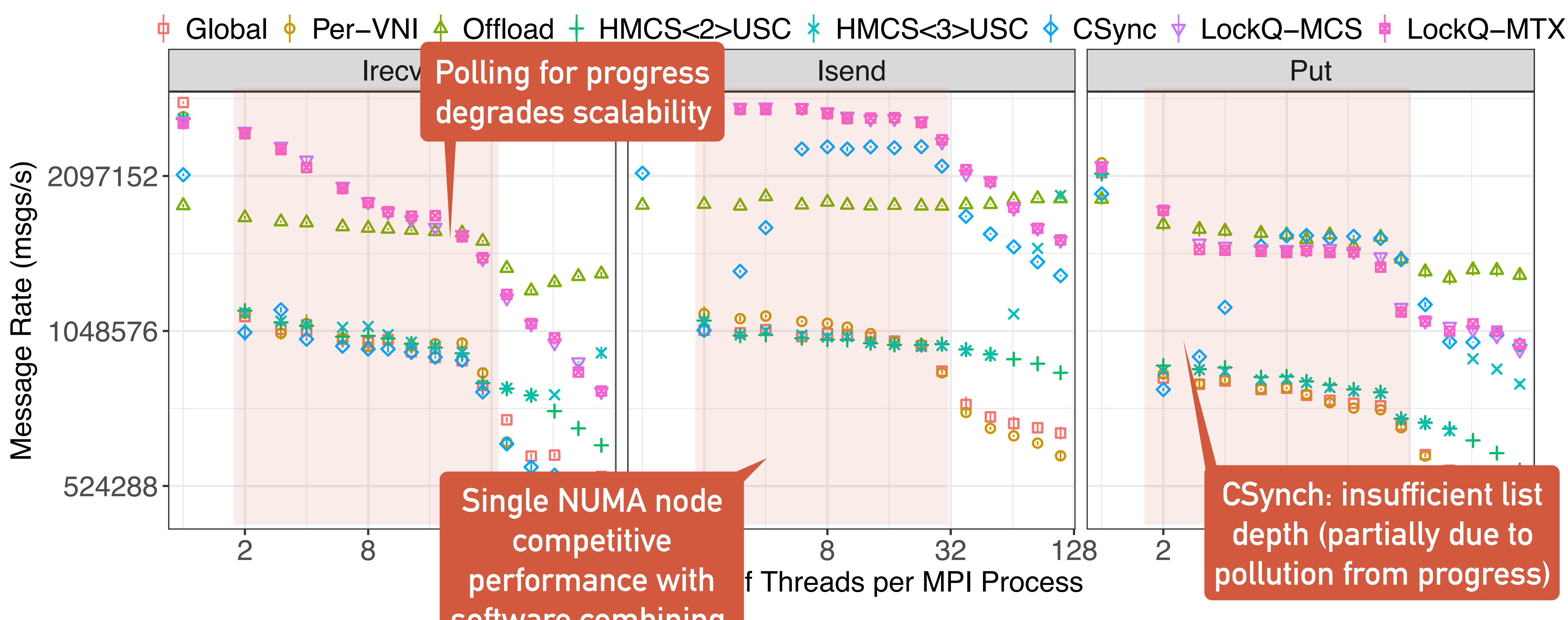
Label	Global	Per-VNI	Offload	HMCS<N>USC	CSync	LockQ-MCS	LockQ-MTX
Description	Global MCS lock	VNI fine-grained locking with MCS	Lockless Software offloading	HMCS lock + O(1) wakeup	VNI granularity with DSM-Synch	VNI granularity with MCS-based LcokQ	VNI granularity with Pthread mutex-based LockQ



64B message rate between two 56-Core Intel Skylake nodes at 2.5GHz over Intel Omnipath

MESSAGE RATE VS. STATE-OF-THE-ART

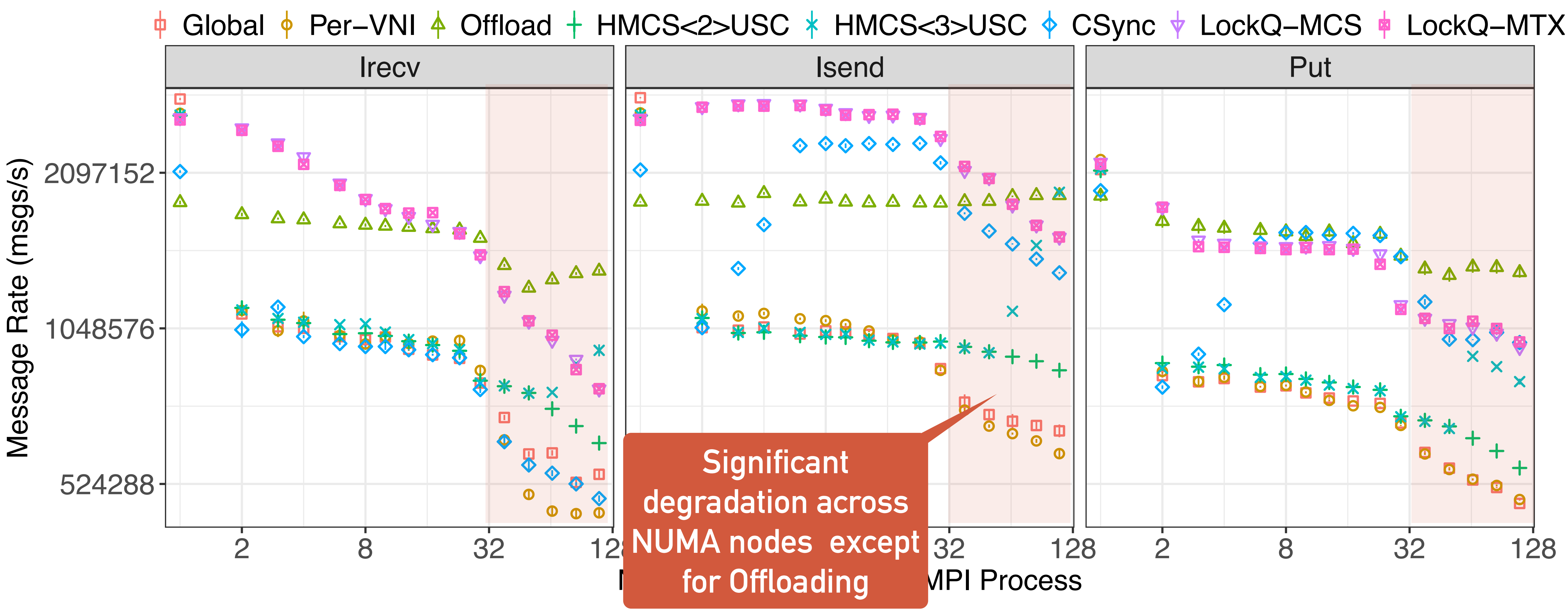
Label	Global	Per-VNI	Offload	HMCS<N>USC	CSync	LockQ-MCS	LockQ-MTX
Description	Global MCS lock	VNI fine-grained locking with MCS	Lockless Software offloading	HMCS lock + O(1) wakeup	VNI granularity with DSM-Synch	VNI granularity with MCS-based LcokQ	VNI granularity with Pthread mutex-based LockQ



64B message rate betw… ntel Skylake nodes at 2.5GHz over Intel Omnipath

MESSAGE RATE VS. STATE-OF-THE-ART

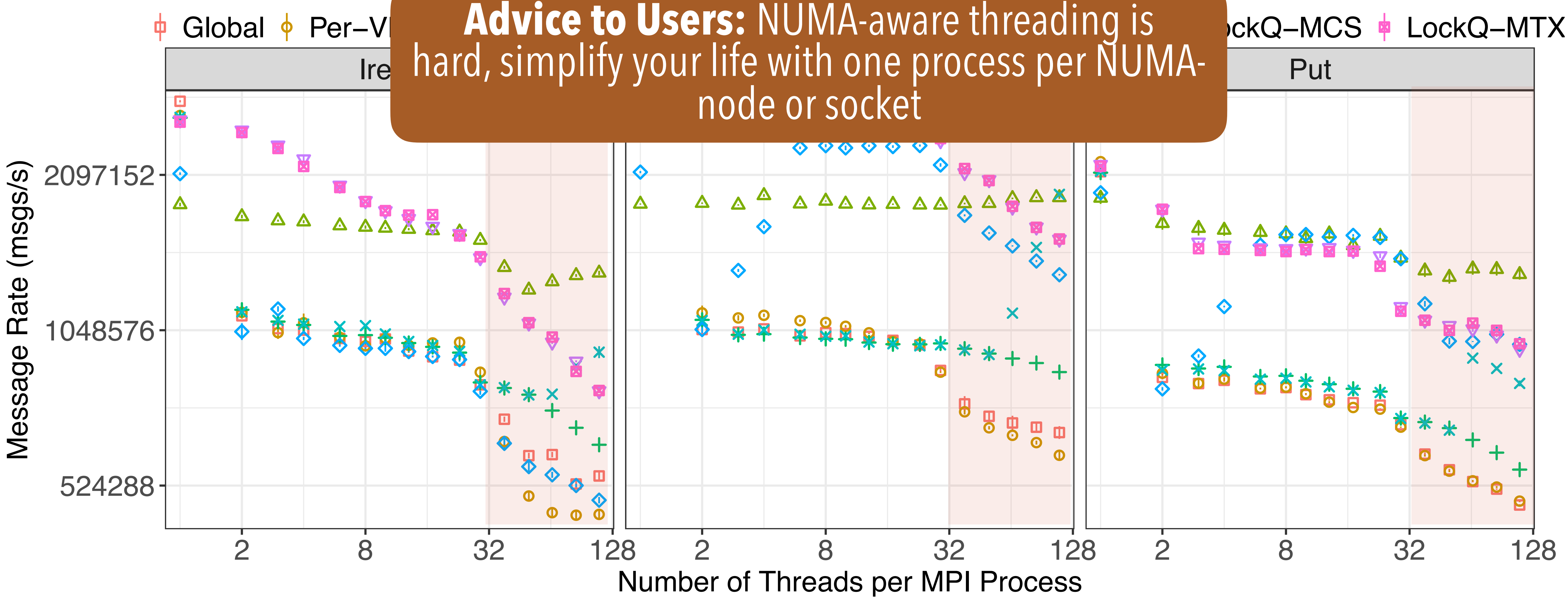
Label	Global	Per-VNI	Offload	HMCS<N>USC	CSync	LockQ-MCS	LockQ-MTX
Description	Global MCS lock	VNI fine-grained locking with MCS	Lockless Software offloading	HMCS lock + O(1) wakeup	VNI granularity with DSM-Synch	VNI granularity with MCS-based LcokQ	VNI granularity with Pthread mutex-based LockQ



64B message rate between two 56-Core Intel Skylake nodes at 2.5GHz over Intel Omnipath

MESSAGE RATE VS. STATE-OF-THE-ART

Label	Global	Per-VNI	Offload	HMCS<N>USC	CSync	LockQ-MCS	LockQ-MTX
Description	Global MCS lock	VNI fine-grained locking with MCS	Lockless Software offloading	HMCS lock + O(1) wakeup	VNI granularity with DSM-Synch	VNI granularity with MCS-based LcokQ	VNI granularity with Pthread mutex-based LockQ



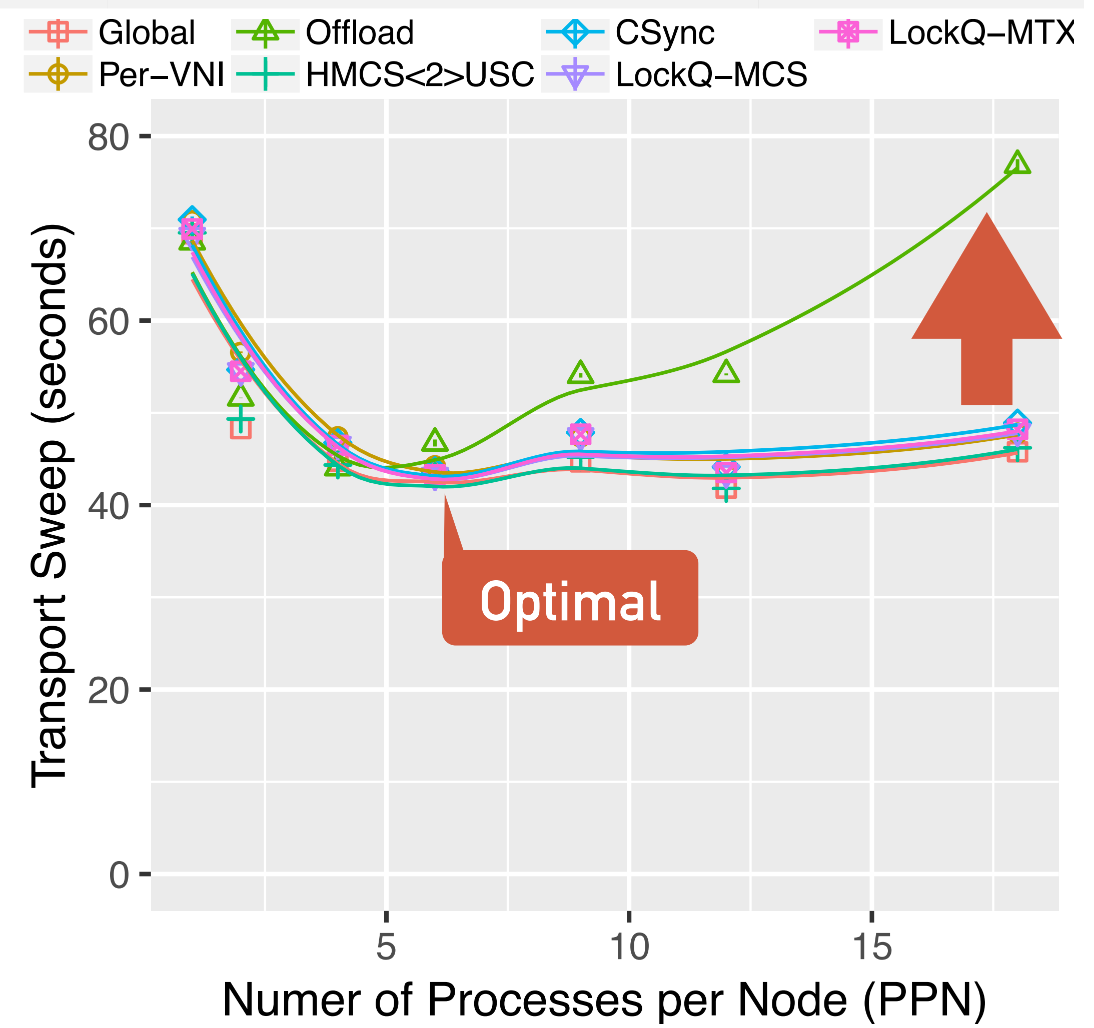
Advice to Users: NUMA-aware threading is hard, simplify your life with one process per NUMA-node or socket

64B message rate between two 56-Core Intel Skylake nodes at 2.5GHz over Intel Omnipath

CPU SACRIFICES TEST WITH THE SNAP PROXY-APP

Label	Global	Per-VNI	Offload	HMCS<N>USC	CSync	LockQ-MCS	LockQ-MTX
Description	Global MCS lock	VNI fine-grained locking with MCS	Lockless Software offloading	HMCS lock + O(1) wakeup	VNI granularity with DSM-Synch	VNI granularity with MCS-based LcokQ	VNI granularity with Pthread mutex-based LockQ

- SNAP (https://github.com/losalamos/snap) proxy application
- Models the PARTISN particle transport application
- Wavefront communication pattern with two-sided communication
- Tuning Processes per Node (PPN)
  - Standard practice
  - Reduces inter-NUMA-node cache traffic
  - All methods perform best at PPN=6
  - **Offload up to 2x degradation from over sacrificing CPU resources**



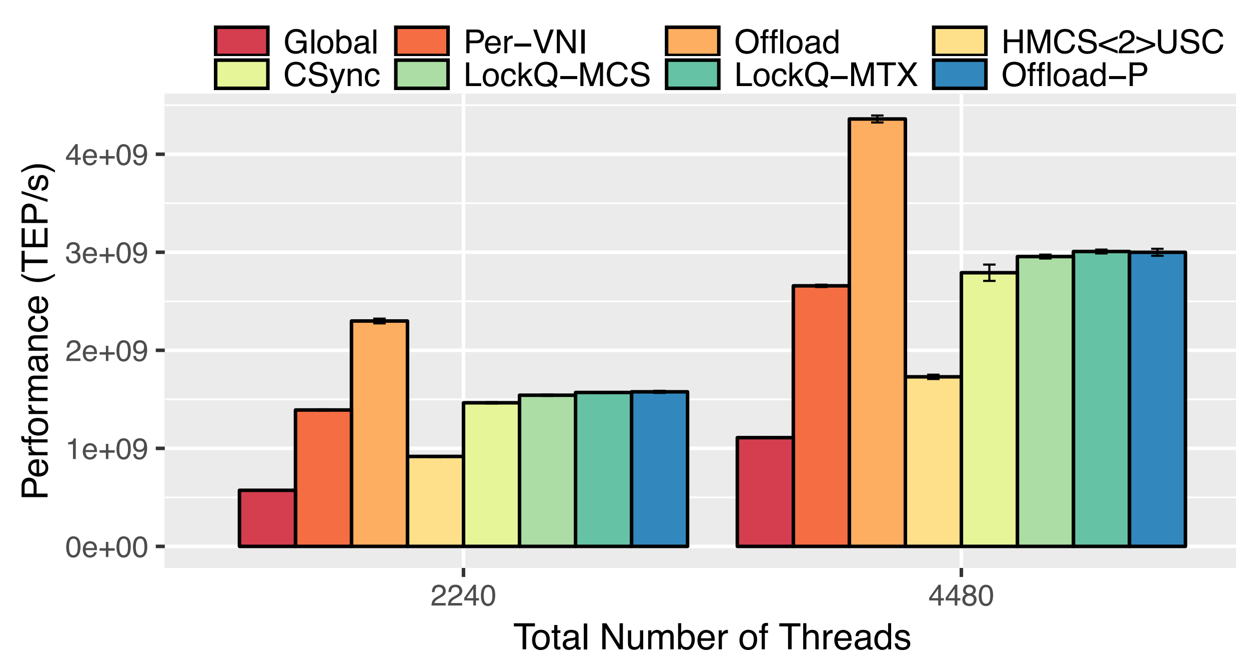
Performance with the "Transport Sweep" stage of SNAP on 16 Broadwell nodes over Intel OmniPath with problem size {nx,ny,nz} = {128,72,64} with respect to PPN

WORST–CASE CONTENTION WITH GRAPH500

Label	Global	Per-VNI	Offload	HMCS<N>USC	CSync	LockQ-MCS	LockQ-MTX
Description	Global MCS lock	VNI fine-grained locking with MCS	Lockless Software offloading	HMCS lock + O(1) wakeup	VNI granularity with DSM-Synch	VNI granularity with MCS-based LcokQ	VNI granularity with Pthread mutex-based LockQ

- Graph500 Benchmark (graph500.org)
- Core kernel: breadth-first search
- Updated to perform computation and communication concurrently by threads [1]
- Communication initiation: nonblocking point-to-point
- Completion detection: MPI_Test
- LockQ outperforms CSync and existing lock-based methods
- Offload significantly outperforms every other method
- Bottlneck: progress in MPI_Test
  - Impossible to beat Offload (only check local flag)
  - Still nonblocking progress + exponential backoff in offload needs improvements

Offload-P
Lockless Software offloading + application threads call progress



Graph500 strong-scaling results on the Broadwell-OmniPath cluster with 35 threads per MPI process with respect to the total number of threads.

Amer et al. "Characterizing MPI and hybrid MPI+ Threads applications at scale: case study with BFS." CCGrid'15

SUMMARY

- LockQ takes advantage of software combining for scalability
- Leverages MPI semantics to relax synchronization
- Results
  - High throughput without hardwire knowledge
  - Asynchronous nonblocking calls for latency hiding and communication overlapping
- LockQ already released in MPICH 3.3 (if you want to try it out)
- Nonblocking progress management insufficient
  - Make MPI_Test family of calls scale is still an open problem
- Evaluation with multiple VNIs for further insight

ACKNOWLEDGMENT

- **ICS Organization and Program committees**
- **Funding**
  - This research was supported by the **Exascale Computing Project** (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357
- **Resources**
  - We gratefully for the computing resources provided and operated by the Laboratory Computing Resource Center (**LCRC**) and by the Joint Laboratory for System Evaluation (**JLSE**) at Argonne National Laboratory.