Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Lock-Free and Distributed Memory

Instructor: Torsten Hoefler & Markus Püschel

TA: Timo Schneider

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Administrivia

- **Final project presentation: Monday 12/16 during last lecture**
  - Send slides to Timo by 12/16, 11am
  - 15 minutes per team (hard limit)

  - Rough guidelines:

    Summarize your goal/task

    Related work (what exists, literature review!)

    Describe techniques/approach (details!)

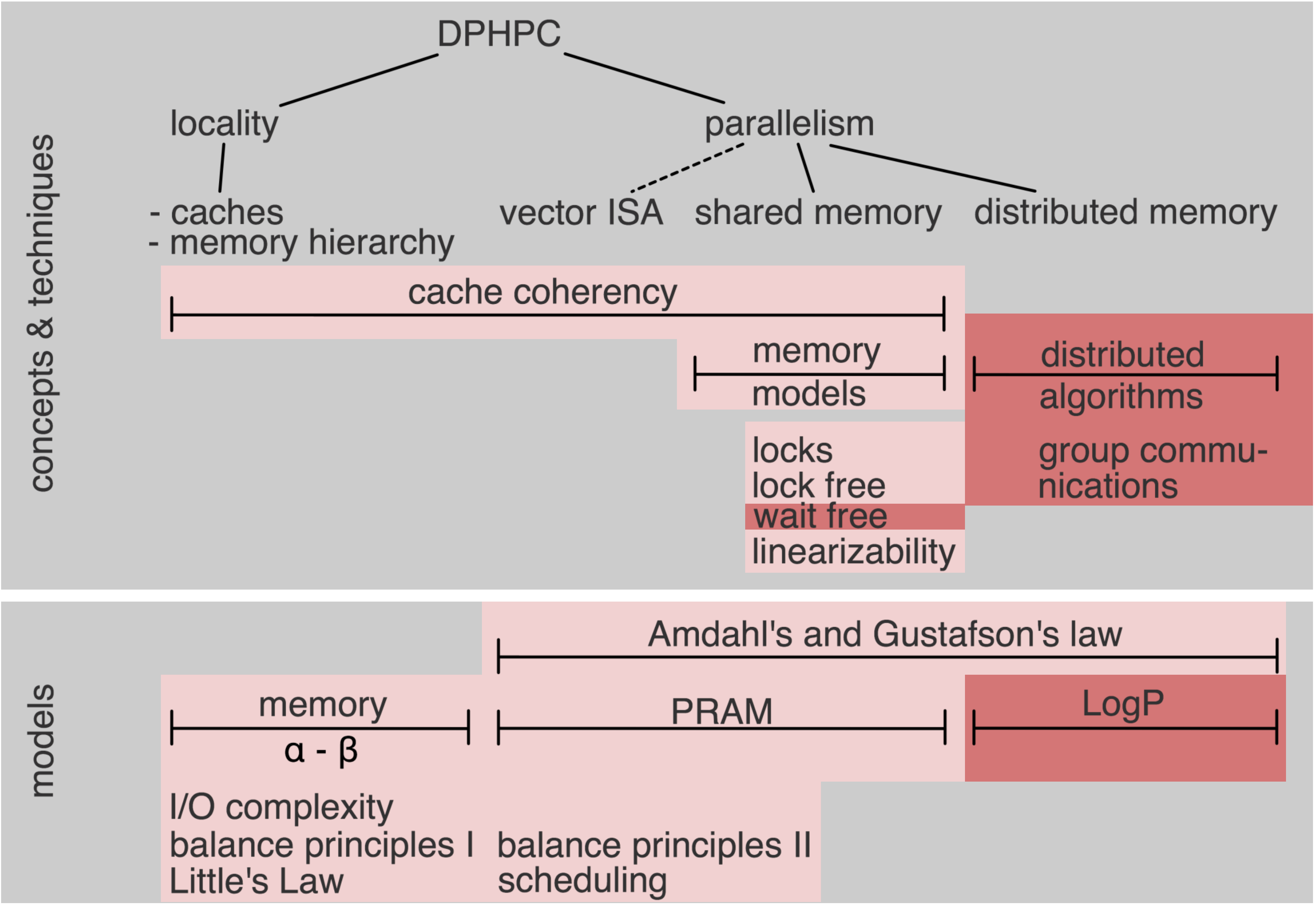
    Final results and findings (details)

    Pick one presenter (you may also switch but keep the time in mind)

Review of last lecture

- **Abstract models**
  - Amdahl's and Gustafson's Law
  - Little's Law
  - Work/depth models and Brent's theorem
  - I/O complexity and balance (Kung)
  - Balance principles
- **Scheduling**
  - Greedy
  - Random work stealing
- **Balance principles**
  - Outlook to the future
  - Memory and data-movement will be more important

DPHPC Overview



Goals of this lecture

- **Answer "Why need to lock+validate in contains of optimistic queue"?**
  - An element may be reused, assume free() is called after remove
  - Contains in A may grab pointer to element and suspend
  - B frees element and grabs location as new memory and initializes it to V
  - Resumed contains in A may now find V even though it was never in the list
- **Finish wait-free/lock-free**
  - Consensus hierarchy
  - The promised proof!
- **Distributed memory**
  - Models and concepts
  - Designing optimal communication algorithms
- **The Future!**
  - Remote Memory Access Programming

Lock-free and wait-free

- **A lock-free method**
  - guarantees that infinitely often **some** method call finishes in a finite number of steps
- **A wait-free method**
  - guarantees that **each** method call finishes in a finite number of steps (implies lock-free)
  - Was our lock-free list also wait-free?
- **Synchronization instructions are not equally powerful!**
  - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level z>x.

Concept: Consensus Number

- **Each level of the hierarchy has a "consensus number" assigned.**
  - Is the maximum number of threads for which primitives in level x can solve the consensus problem
- **The consensus problem:**
  - Has single function: decide(v)
  - Each thread calls it at most once, the function returns a value that meets two conditions:

    consistency: all threads get the same value

    valid: the value is some thread's input
  - Simplification: binary consensus (inputs in {0,1})

Understanding Consensus

- **Can a particular class solve n-thread consensus wait-free?**
  - A class C solves n-thread consensus if there exists a consensus protocol using **any number** of objects of class C and **any number** of atomic registers
  - The protocol has to be wait-free (bounded number of steps per thread)
  - The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite)
  - Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Starting simple …

- **Binary consensus with two threads (A, B)!**
  - Each threads moves until it decides on a value
  - May update shared objects
  - Protocol state = state of threads + state of shared objects
  - Initial state = state before any thread moved
  - Final state = state after all threads finished
  - States form a tree, wait-free property guarantees a finite tree

    Example with two threads and two moves each!

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
  - Really?
- **Proof outline:**
  - Assume arbitrary consensus protocol, thread A, B
  - Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)
  - Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!
    1) *Any thread reads (other thread runs solo until end)*
    2) *Threads write to different registers (order doesn't matter)*
    3) *Threads write to same register (solo thread can start after each write)*

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
- **Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers**
  - "perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)
  - → We need hardware atomics or TM!
- **Proof technique borrowed from:**

  Impossibility of **distributed consensus** with one faulty process
  MJ Fischer, NA Lynch, MS Paterson - Journal of the ACM (JACM), 1985 - dl.acm.org
  Abstract The **consensus** problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of **...**
  Cited by 3180 Related articles All 164 versions

- **Very influential paper, always worth a read!**
  - Nicely shows proof techniques that are central to parallel and distributed computing!

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)
- **CAS and TM have consensus number ∞**
  - Constructive proof!

Compare and Set/Swap Consensus

```
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
  proposed[tid] = v;
  if(CAS(thread, first, tid))
    return v; // I won!
  else
    return proposed[thread]; // thread won
}
```

- **CAS provides an infinite consensus number**
  - Machines providing CAS are **asynchronous** computation equivalents of the Turing Machine
  - I.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)

Now you know everything ☺

- **Not really … ;-)**
  - We'll argue about **performance** now!
- **But you have all the tools for:**
  - Efficient locks
  - Efficient lock-based algorithms
  - Efficient lock-free algorithms (or even wait-free)
  - Reasoning about parallelism!
- **What now?**
  - A different class of problems

    Impact on wait-free/lock-free on actual performance is not well understood
  - Relevant to HPC, applies to shared and distributed memory

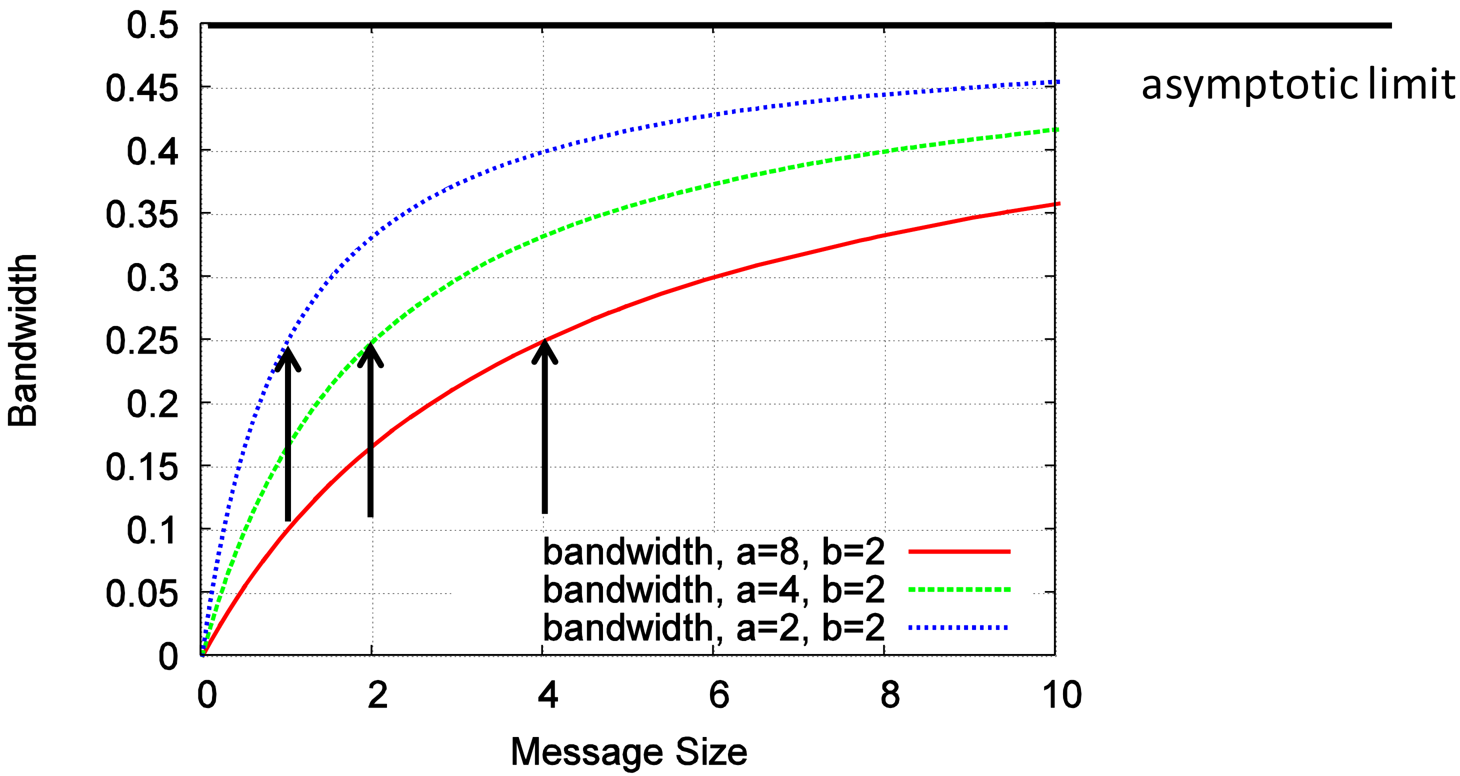
    → *Group communications*

Remember: A Simple Model for Communication

- **Transfer time $T(s) = \alpha+\beta s$**
  - α = startup time (latency)
  - β = cost per byte (bandwidth=$1/\beta$)
- **As s increases, bandwidth approaches $1/\beta$ asymptotically**
  - Convergence rate depends on α
  - $s_{1/2} = \alpha/\beta$
- **Assuming no pipelining (new messages can only be issued from a process after all arrived)**

Bandwidth vs. Latency

- **$s_{1/2} = \alpha/\beta$ often used to distinguish bandwidth- and latency-bound messages**
  - $s_{1/2}$ is in the order of kilobytes on real systems



Quick Example

- **Simplest linear broadcast**
  - One process has a data item to be distributed to all processes
- **Broadcasting s bytes among P processes:**
  - T(s) = (P-1) * (α+βs) = $\mathcal{O}(P)$
- **Class question: Do you know a faster method to accomplish the same?**

k-ary Tree Broadcast

- **Origin process is the root of the tree, passes messages to k neighbors which pass them on**
  - k=2 -> binary tree
- **Class Question: What is the broadcast time in the simple latency/bandwidth model?**
  - $T(s) \approx \lceil log_k(P) \rceil \cdot k \cdot (\alpha + \beta \cdot s) = \mathcal{O}(log(P))$ (for fixed k)
- **Class Question: What is the optimal k?**
  - $0 = \frac{ln(P) \cdot k}{ln(k)} \frac{d}{dk} = \frac{ln(P) ln(k) - ln(P)}{ln^2(k)} \rightarrow k = e = 2.71...$
  - Independent of P, α, βs? Really?

Faster Trees?

- **Class Question: Can we broadcast faster than in a ternary tree?**
  - Yes because each respective root is idle after sending three messages!
  - Those roots could keep sending!
  - Result is a k-nomial tree

    For k=2, it's a binomial tree

- **Class Question: What about the runtime?**
  - $$T(s) = \lceil log_k(P) \rceil \cdot (k-1) \cdot (\alpha + \beta \cdot s) = \mathcal{O}(log(P))$$
- **Class Question: What is the optimal k here?**
  - T(s) d/dk has monotonically increasing for k>1, thus k_{opt}=2
- **Class Question: Can we broadcast faster than in a k-nomial tree?**
  - $\mathcal{O}(log(P))$ is asymptotically optimal for s=1!
  - But what about large s?

Very Large Message Broadcast

- **Extreme case (P small, s large): simple pipeline**
  - Split message into segments of size z
  - Send segments from PE i to PE i+1
- **Class Question: What is the runtime?**
  - $T(s) = (P-2+s/z)(\alpha + \beta z)$
- **Compare 2-nomial tree with simple pipeline for $\alpha=10$, $\beta=1$, $P=4$, $s=10^6$, and $z=10^5$**
  - 2,000,020 vs. 1,200,120
- **Class Question: Can we do better for given α, β, P, s?**
  - Derive by z $z_{opt} = \sqrt{\frac{s\alpha}{(P-2)\beta}}$
- **What is the time for simple pipeline for $\alpha=10$, $\beta=1$, $P=4$, $s=10^6$, z_{opt}?**
  - 1,008,964

Lower Bounds

- **Class Question: What is a simple lower bound on the broadcast time?**
  - $T_{BC} \geq \min\{\lceil \log_2(P) \rceil \alpha, s\beta\}$
- **How close are the binomial tree for small messages and the pipeline for large messages (approximately)?**
  - Bin. tree is a factor of $\log_2(P)$ slower in bandwidth
  - Pipeline is a factor of $P/\log_2(P)$ slower in latency
- **Class Question: What can we do for intermediate message sizes?**
  - Combine pipeline and tree → pipelined tree
- **Class Question: What is the runtime of the pipelined binary tree algorithm?**
  - $T \approx \left(\frac{s}{z} + \lceil \log_2 P \rceil - 2\right) \cdot 2 \cdot (\alpha + z\beta)$
- **Class Question: What is the optimal z?**
  - $z_{opt} = \sqrt{\frac{\alpha s}{\beta(\lceil \log_2 P \rceil - 2)}}$

Towards an Optimal Algorithm

- **What is the complexity of the pipelined tree with z_{opt} for small s, large P and for large s, constant P?**
  - Small messages, large P: s=1; z=1 (s≤z), will give O(log P)
  - Large messages, constant P: assume α, β, P constant, will give asymptotically O(sβ)

    Asymptotically optimal for large P and s but bandwidth is off by a factor of 2! Why?

- **Bandwidth-optimal algorithms exist, e.g., Sanders et al. *"Full Bandwidth Broadcast, Reduction and Scan with Only Two Trees"*. 2007**
  - Intuition: in binomial tree, all leaves (P/2) only receive data and never send → wasted bandwidth
  - Send along two simultaneous binary trees where the leafs of one tree are inner nodes of the other
  - Construction needs to avoid endpoint congestion (makes it complex)

    Can be improved with linear programming and topology awareness

    (talk to me if you're interested)

Open Problems

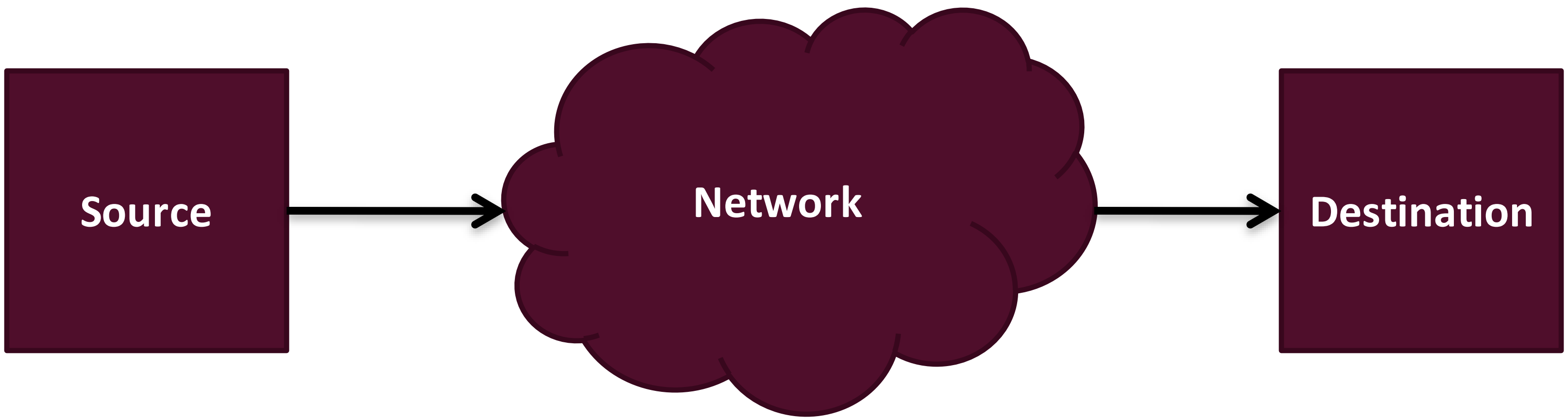
- **Look for optimal parallel algorithms (even in simple models!)**
  - And then check the more realistic models
  - Useful optimization targets are MPI collective operations

    Broadcast/Reduce, Scatter/Gather, Alltoall, Allreduce, Allgather, Scan/Exscan, …
  - Implementations of those (check current MPI libraries ☺)
  - Useful also in scientific computations

    Barnes Hut, linear algebra, FFT, …
- **Lots of work to do!**
  - Contact me for thesis ideas (or check SPCL) if you like this topic
  - Usually involve optimization (ILP/LP) and clever algorithms (algebra) combined with practical experiments on large-scale machines (10,000+ processors)

HPC Networking Basics

- **Familiar (non-HPC) network: Internet TCP/IP**
  - Common model:

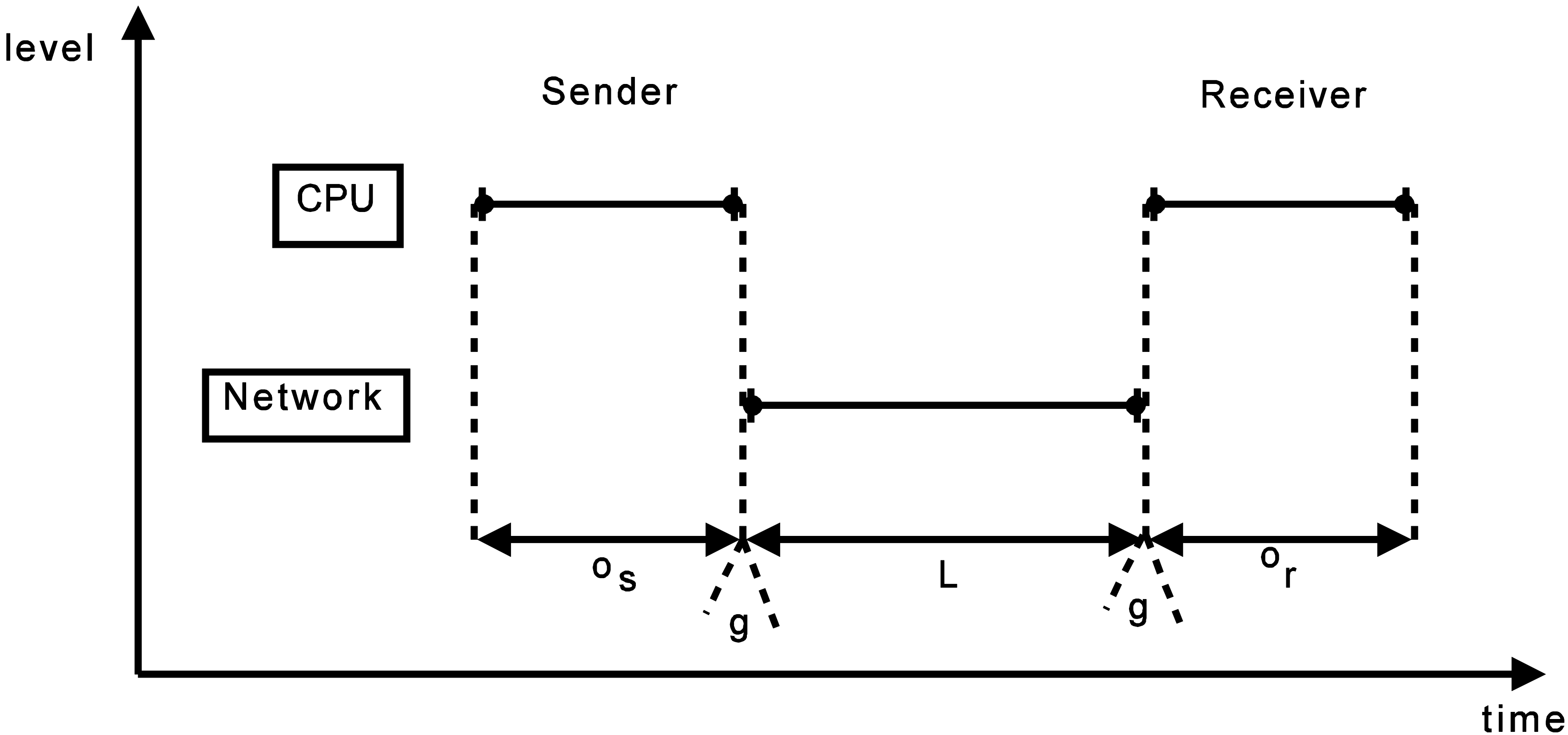


- **Class Question: What parameters are needed to model the performance (including pipelining)?**
  - Latency, Bandwidth, Injection Rate, Host Overhead

The LogP Model

- **Defined by four parameters:**
  - L: an upper bound on the latency, or delay, incurred in communicating a message containing a word (or small number of words) from its source module to its target module.
  - o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
  - g: the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.
  - P: the number of processor/memory modules. We assume unit time for local operations and call it a cycle.

The LogP Model



Simple Examples

- **Sending a single message**
  - $T = 2o+L$

- **Ping-Pong Round-Trip**
  - $T_{RTT} = 4o+2L$

- **Transmitting n messages**
  - $T(n) = L+(n-1)*\max(g, o) + 2o$

Simplifications

- **o is bigger than g on some machines**
  - g can be ignored (eliminates max() terms)
  - be careful with multicore!
- **Offloading networks might have very low o**
  - Can be ignored (not yet but hopefully soon)
- **L might be ignored for long message streams**
  - If they are pipelined
- **Account g also for the first message**
  - Eliminates "-1"

Benefits over Latency/Bandwidth Model

- **Models pipelining**
  - L/g messages can be “in flight”
  - Captures state of the art (cf. TCP windows)
- **Models computation/communication overlap**
  - Asynchronous algorithms
- **Models endpoint congestion/overload**
  - Benefits balanced algorithms

Example: Broadcasts

- **Class Question: What is the LogP running time for a linear broadcast of a single packet?**
  - $T_{lin} = L + (P-2) * \max(o,g) + 2o$
- **Class Question: Approximate the LogP runtime for a binary-tree broadcast of a single packet?**
  - $T_{bin} \leq \log_2 P * (L + \max(o,g) + 2o)$
- **Class Question: Approximate the LogP runtime for an k-ary-tree broadcast of a single packet?**
  - $T_{k\text{-}n} \leq \log_k P * (L + (k-1)\max(o,g) + 2o)$

Example: Broadcasts

- **Class Question: Approximate the LogP runtime for a binomial tree broadcast of a single packet?**
  - $T_{bin} \leq \log_2 P * (L + 2o)$ (assuming $L > g$!)
- **Class Question: Approximate the LogP runtime for a k-nomial tree broadcast of a single packet?**
  - $T_{k\text{-}n} \leq \log_k P * (L + (k-2)\max(o,g) + 2o)$
- **Class Question: What is the optimal k (assume o>g)?**
  - Derive by k: $0 = o * \ln(k_{opt}) - L/k_{opt} + o$ (solve numerically)

    For larger L, k grows and for larger o, k shrinks
  - Models pipelining capability better than simple model!

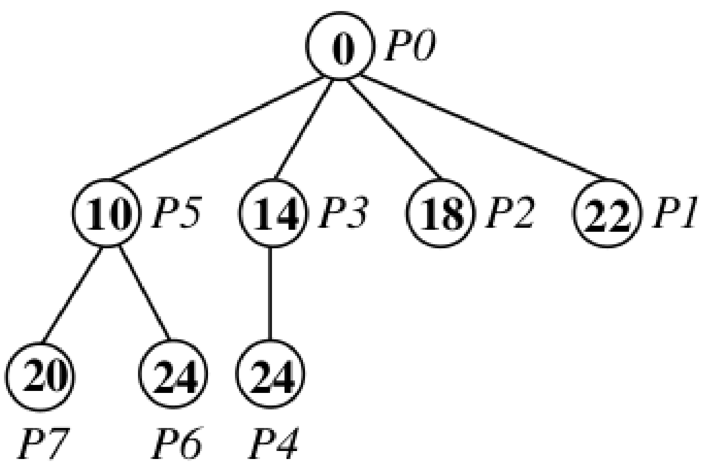
Example: Broadcasts

- **Class Question: Can we do better than k_{opt}-ary binomial broadcast?**
  - Problem: fixed k in all stages might not be optimal

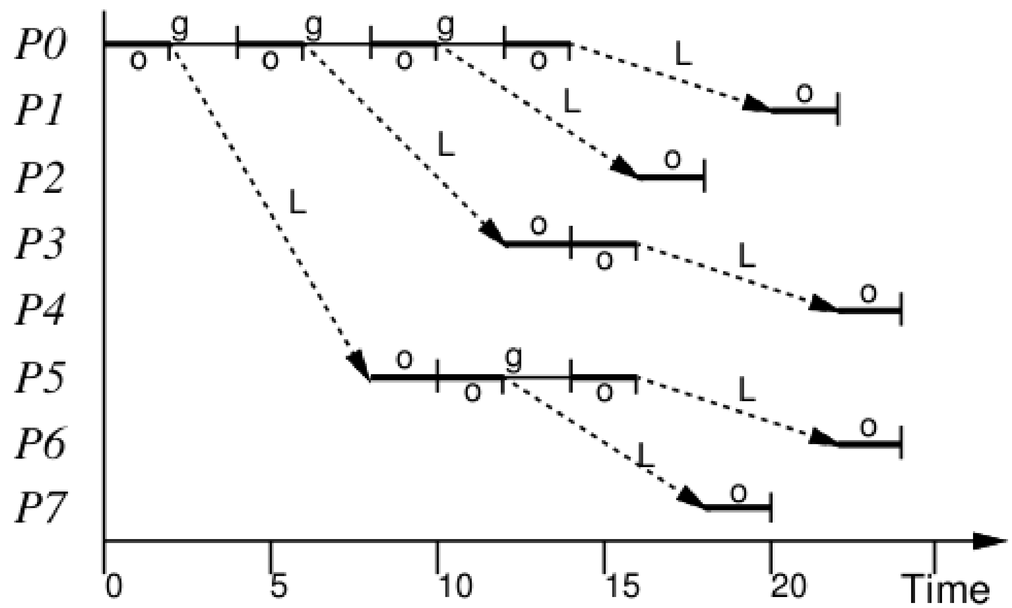
    Only a constant away from optimum
  - We can construct a schedule for the optimal broadcast in practical settings
  - First proposed by Karp et al. in “Optimal Broadcast and Summation in the LogP Model”

Example: Optimal Broadcast

- **Broadcast to P-1 processes**
  - Each process who received the value sends it on; each process receives exactly once



P=8, L=6, g=4, o=2



Optimal Broadcast Runtime

- **This determines the maximum number of PEs (P(t)) that can be reached in time t**
- **P(t) can be computed with a generalized Fibonacci recurrence (assuming o>g):**

$$P(t) = \begin{cases} 1 : & t < 2o + L \\ P(t-o) + P(t-L-2o) : & \text{otherwise.} \end{cases} \quad (1)$$

- **Which can be bounded by (see [1]):** $2^{\left\lfloor \frac{t}{L+2o} \right\rfloor} \leq P(t) \leq 2^{\left\lfloor \frac{t}{o} \right\rfloor}$
  - A closed solution is an interesting open problem!

[1]: Hoefler et al.: “Scalable Communication Protocols for Dynamic Sparse Data Exchange” (Lemma 1)

The Bigger Picture

- **We learned how to program shared memory systems**
  - Coherency & memory models & linearizability
  - Locks as examples for reasoning about correctness and performance
  - List-based sets as examples for lock-free and wait-free algorithms
  - Consensus number
- **We learned about general performance properties and parallelism**
  - Amdahl’s and Gustafson’s laws
  - Little’s law, Work-span, …
  - Balance principles & scheduling
- **We learned how to perform model-based optimizations**
  - Distributed memory broadcast example with two models
- **What next? MPI? OpenMP? UPC?**
  - Next-generation machines “merge” shared and distributed memory concepts → Partitioned Global Address Space (PGAS)

Partitioned Global Address Space

- **Two developments:**
  1. Cache coherence becomes more expensive

     May react in software! Scary for industry ;-)
  2. Novel RDMA hardware enables direct access to remote memory

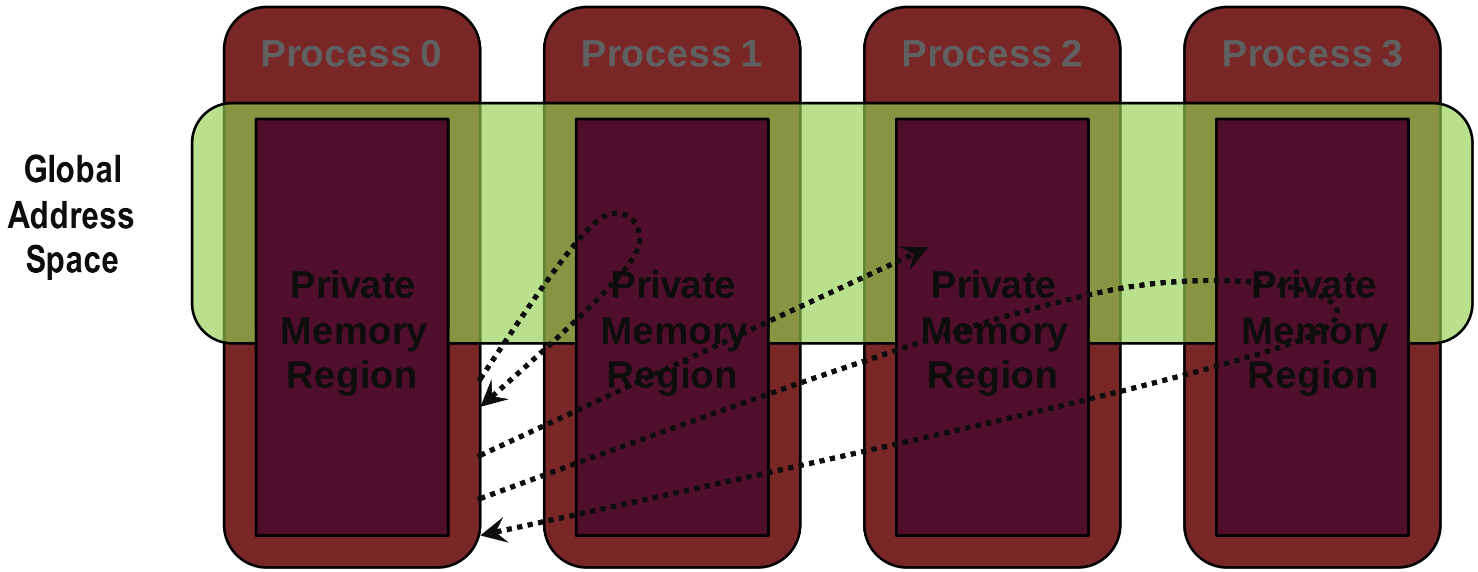
     May take advantage in software! An opportunity for HPC!

- **Still ongoing research! Take nothing for granted ☺**
  - Very interesting opportunities
  - Wide-open research field
  - Even more thesis ideas on next generation parallel programming

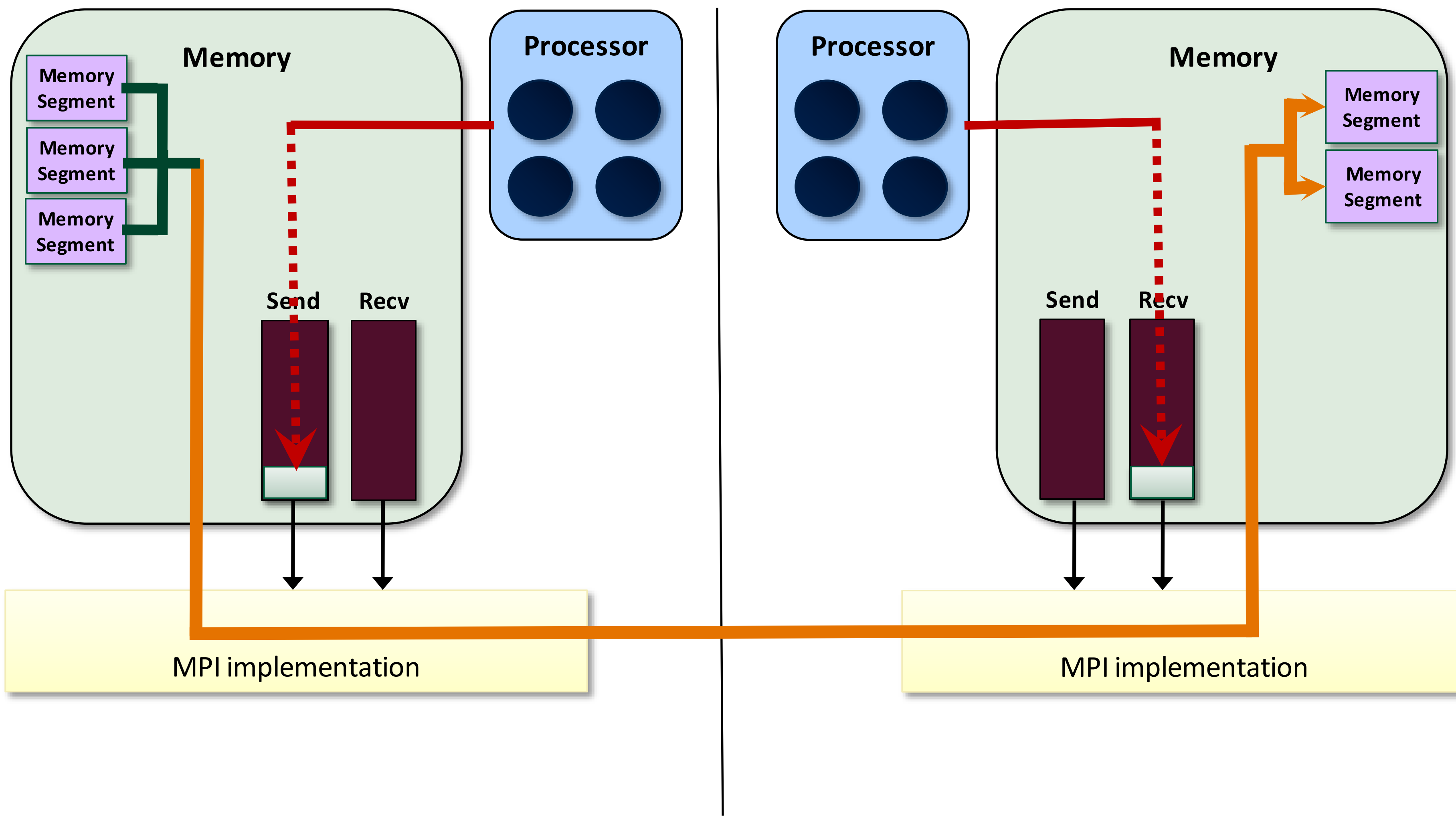
- **I will introduce the concepts behind the MPI-3.0 interface**
  - It's nearly a superset of other PGAS approaches (UPC, CAF, …)

One-sided Communication

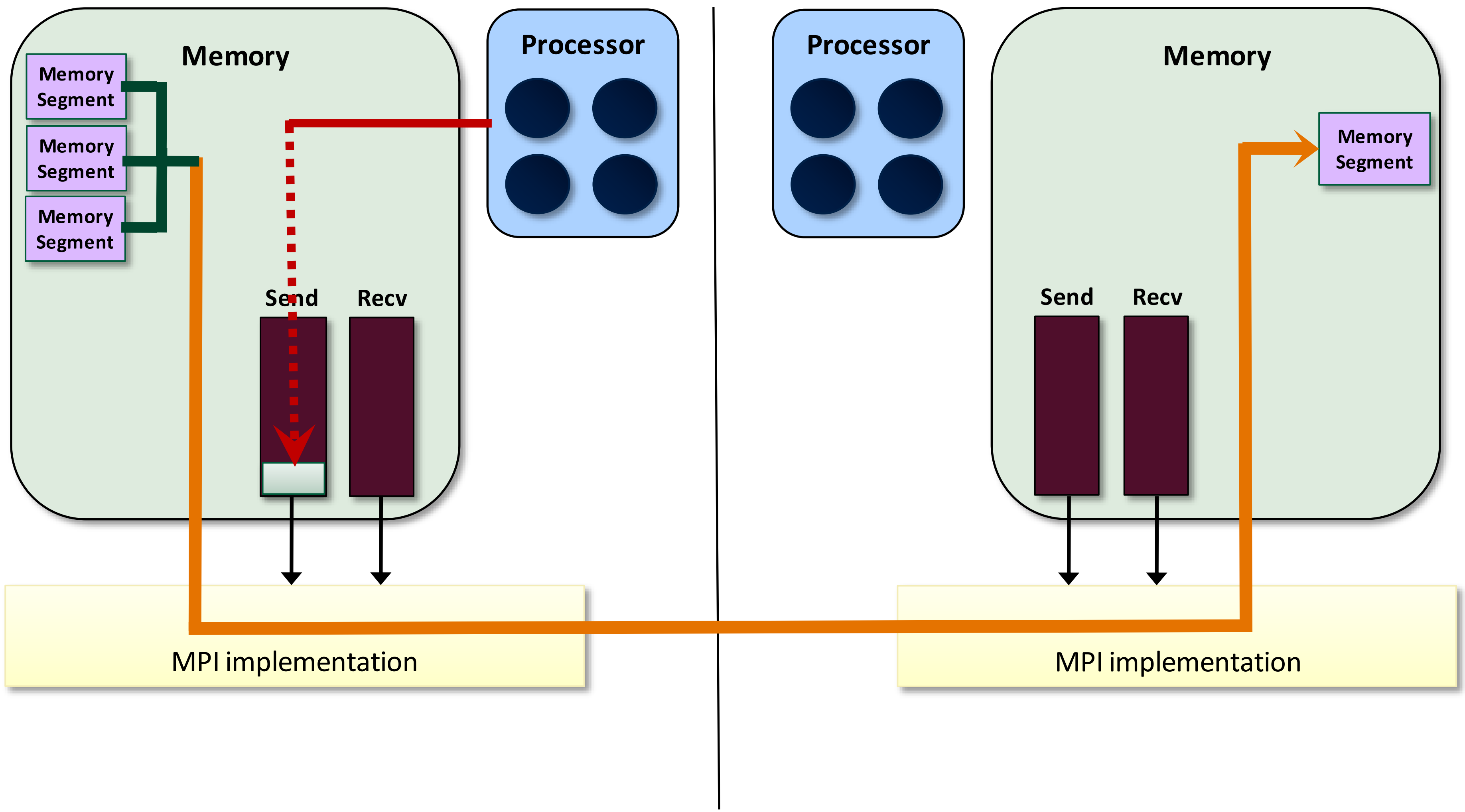
- **The basic idea of one-sided communication models is to decouple data movement with process synchronization**
  - Should be able move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory



Two-sided Communication Example



One-sided Communication Example



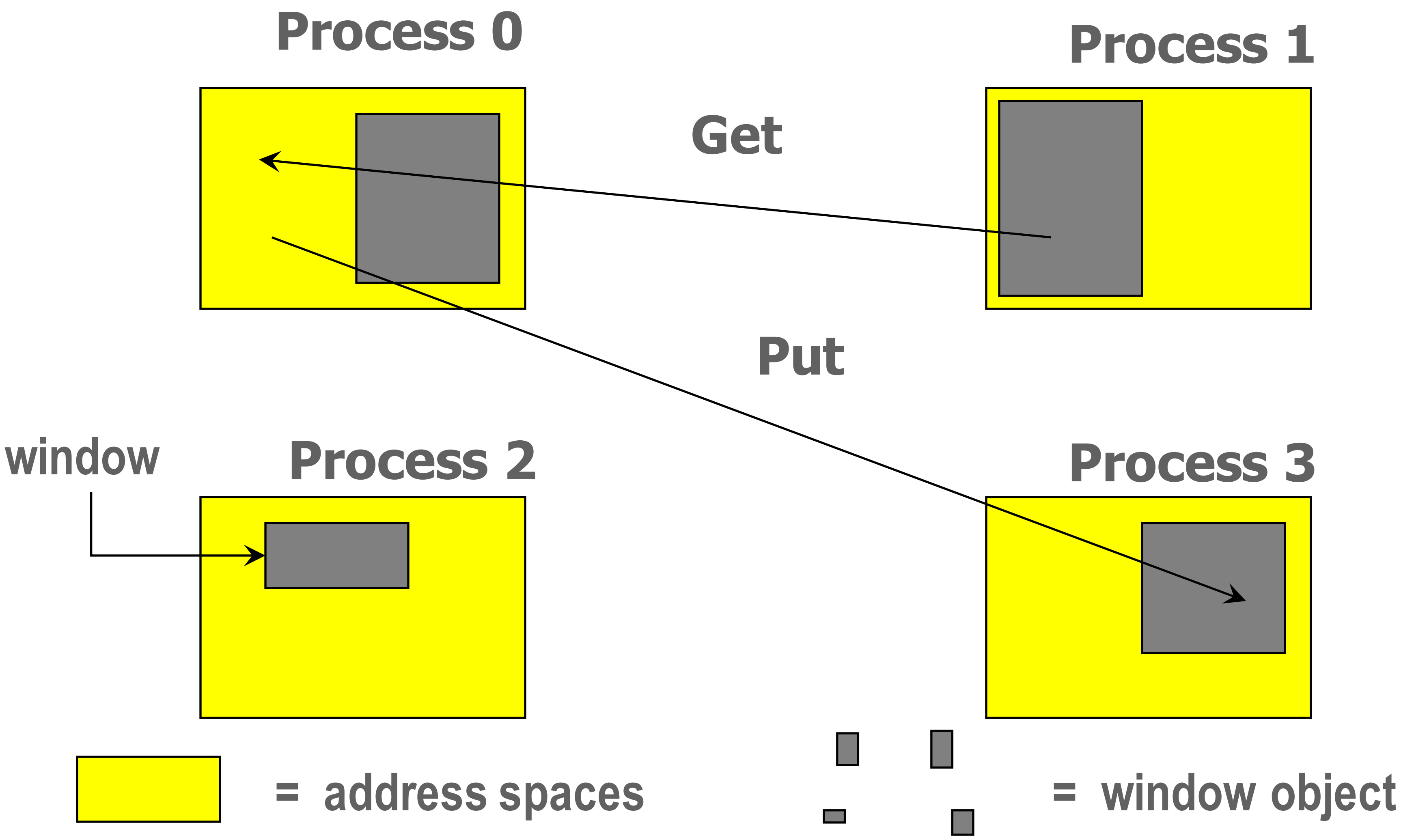
What we need to know in RMA

- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Model

Creating Public Memory

- **Any memory used by a process is, by default, only locally accessible**
  - X = malloc(100);
- **Once the memory is allocated, the user has to make an explicit MPI call to declare a memory region as remotely accessible**
  - MPI terminology for remotely accessible memory is a "window"
  - A group of processes collectively create a "window"
- **Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process**

Remote Memory Access



Basic RMA Functions

- **`MPI_Win_create` – exposes local memory to RMA operation by other processes in a communicator**
  - Collective operation
  - Creates window object
- **`MPI_Win_free` – deallocates window object**
- **`MPI_Put` – moves data from local memory to remote memory**
- **`MPI_Get` – retrieves data from remote memory into local memory**
- **`MPI_Accumulate` – atomically updates remote memory using local values**
  - Data movement operations are non-blocking
  - Data is located by a displacement relative to the start of the window
- **Subsequent synchronization on window object needed to ensure operation is complete**

Window creation models

- **Four models exist**
  - MPI_WIN_CREATE

    You already have an allocated buffer that you would like to make remotely accessible
  - MPI_WIN_ALLOCATE

    You want to create a buffer and directly make it remotely accessible
  - MPI_WIN_CREATE_DYNAMIC

    You don't have a buffer yet, but will have one in the future

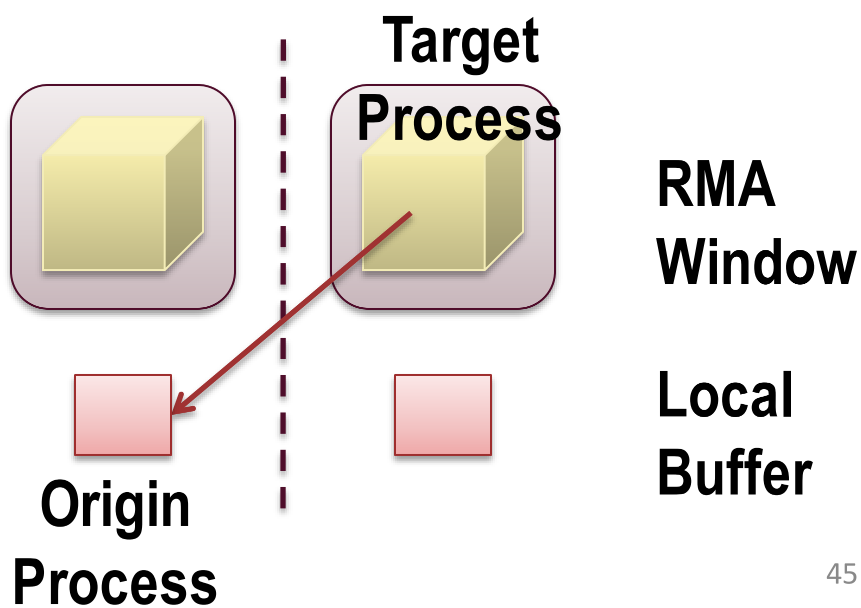
    You may want to dynamically add/remove buffers to/from the window
  - MPI_WIN_ALLOCATE_SHARED

    You want multiple processes on the same node share a buffer

Data movement: *Get*

```
MPI_Get(void * origin_addr, int origin_count,
        MPI_Datatype origin_datatype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_datatype, MPI_Win win)
```

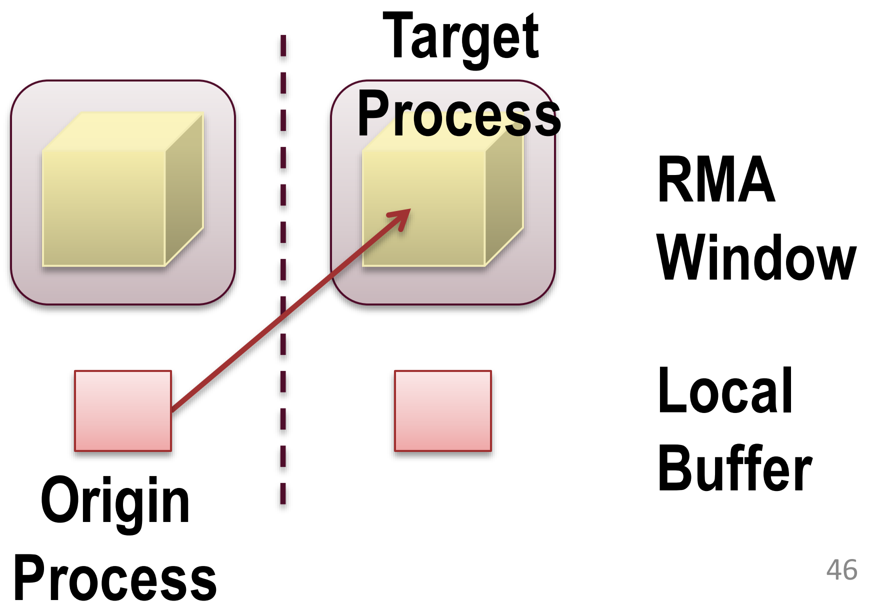
- Move data <u>to</u> origin, <u>from</u> target
- Separate data description triples for origin and target



Data movement: *Put*

```
MPI_Put(void * origin_addr, int origin_count,
        MPI_Datatype origin_datatype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_datatype, MPI_Win win)
```

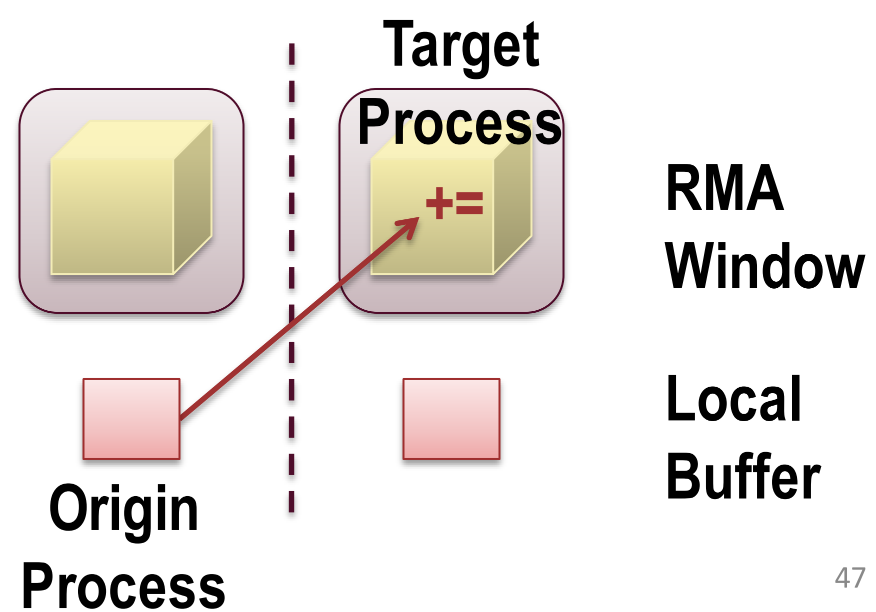
- Move data <u>from</u> origin, <u>to</u> target
- Same arguments as MPI_Get



Atomic Data Aggregation: *Accumulate*

```
MPI_Accumulate(void * origin_addr, int origin_count,
          MPI_Datatype origin_datatype, int target_rank,
          MPI_Aint target_disp, int target_count,
          MPI_Datatype target_dtype, MPI_Op op, MPI_Win win)
```

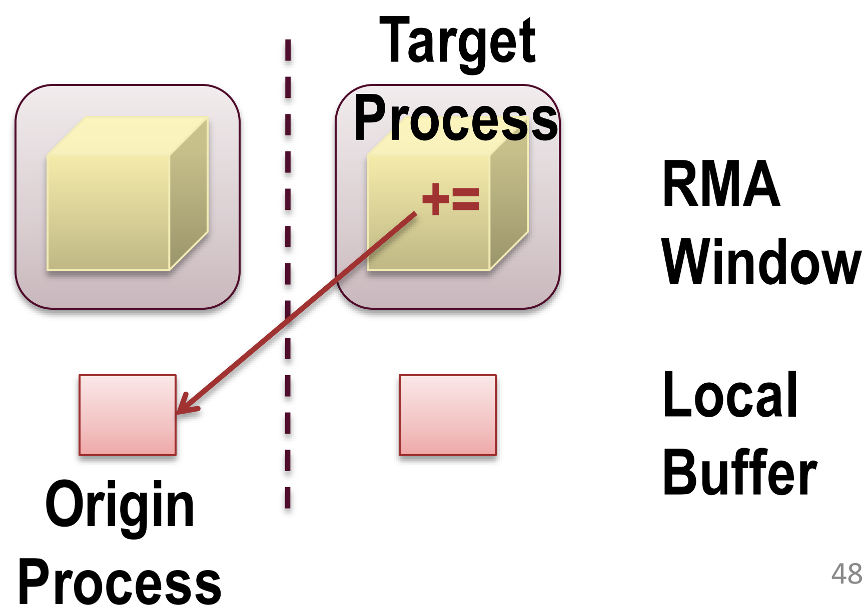
- **Atomic update operation, similar to a put**
  - Reduces origin and target data into target buffer using op argument as combiner
  - Predefined ops only, no user-defined operations
- **Different data layouts between target/origin OK**
  - Basic type elements must match
- **Op = MPI_REPLACE**
  - Implements *f(a,b)=b*
  - Atomic PUT



Atomic Data Aggregation: *Get Accumulate*

```
MPI_Get_accumulate(void *origin_addr, int origin_count,
          MPI_Datatype origin_dtype, void *result_addr,
          int result_count, MPI_Datatype result_dtype,
          int target_rank, MPI_Aint target_disp,
          int target_count, MPI_Datatype target_dype,
          MPI_Op op, MPI_Win win)
```

- **Atomic read-modify-write**
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only
- **Result stored in target buffer**
- **Original data stored in result buf**
- **Different data layouts between target/origin OK**
  - Basic type elements must match
- **Atomic get with MPI_NO_OP**
- **Atomic swap with MPI_REPLACE**



Atomic Data Aggregation: *CAS and FOP*

```
MPI_Compare_and_swap(void *origin_addr,
        void *compare_addr, void *result_addr,
        MPI_Datatype datatype, int target_rank,
        MPI_Aint target_disp, MPI_Win win)
```

- **CAS: Atomic swap if target value is equal to compare value**
- **FOP: Simpler version of MPI_Get_accumulate**
  - All buffers share a single predefined datatype
  - No count argument (it's always 1)
  - Simpler interface allows hardware optimization

```
MPI_Fetch_and_op(void *origin_addr, void *result_addr,
        MPI_Datatype datatype, int target_rank,
        MPI_Aint target_disp, MPI_Op op, MPI_Win win)
```

RMA Synchronization Models

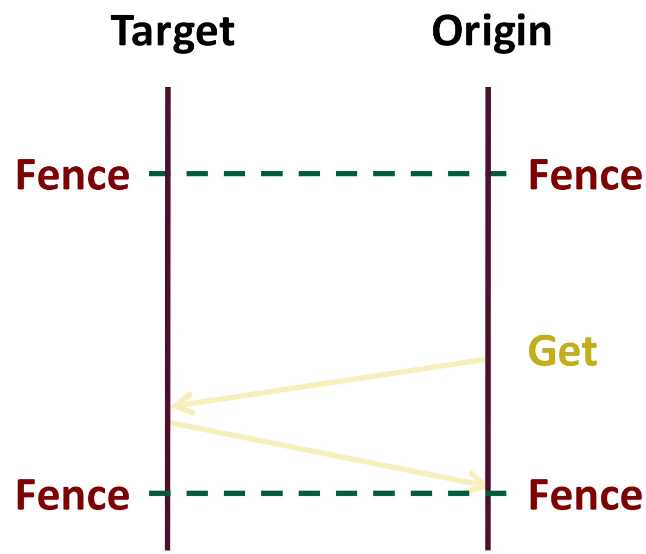
- **RMA data access model**
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X available for process Y to read?
  - RMA synchronization models define these semantics
- **Three synchronization models provided by MPI:**
  - Fence (active target)
  - Post-start-complete-wait (generalized active target)
  - Lock/Unlock (passive target)
- **Data accesses occur within "epochs"**
  - *Access epochs*: contain a set of operations issued by an origin process
  - *Exposure epochs*: enable remote processes to update a target's window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs

    E.g., starting, ending, and synchronizing epochs

Fence: Active Target Synchronization

MPI_Win_fence(int assert, MPI_Win win)

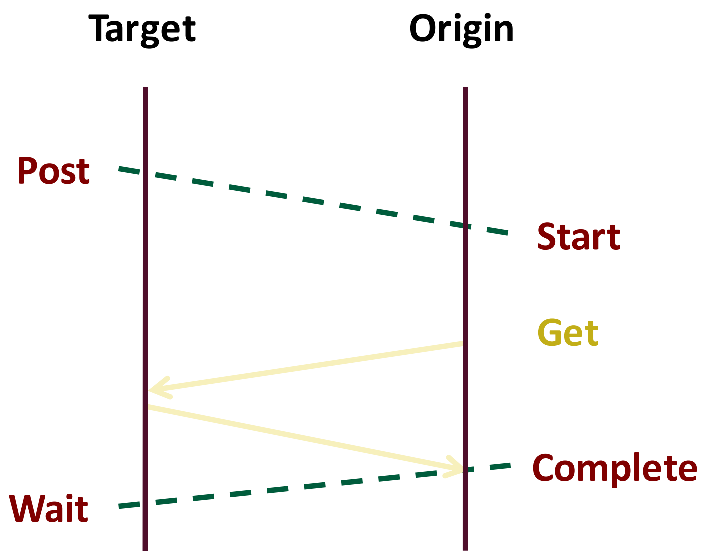
- Collective synchronization model
- Starts *and* ends access and exposure epochs on all processes in the window
- All processes in group of “win” do an MPI_WIN_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
- All operations complete at the second fence synchronization



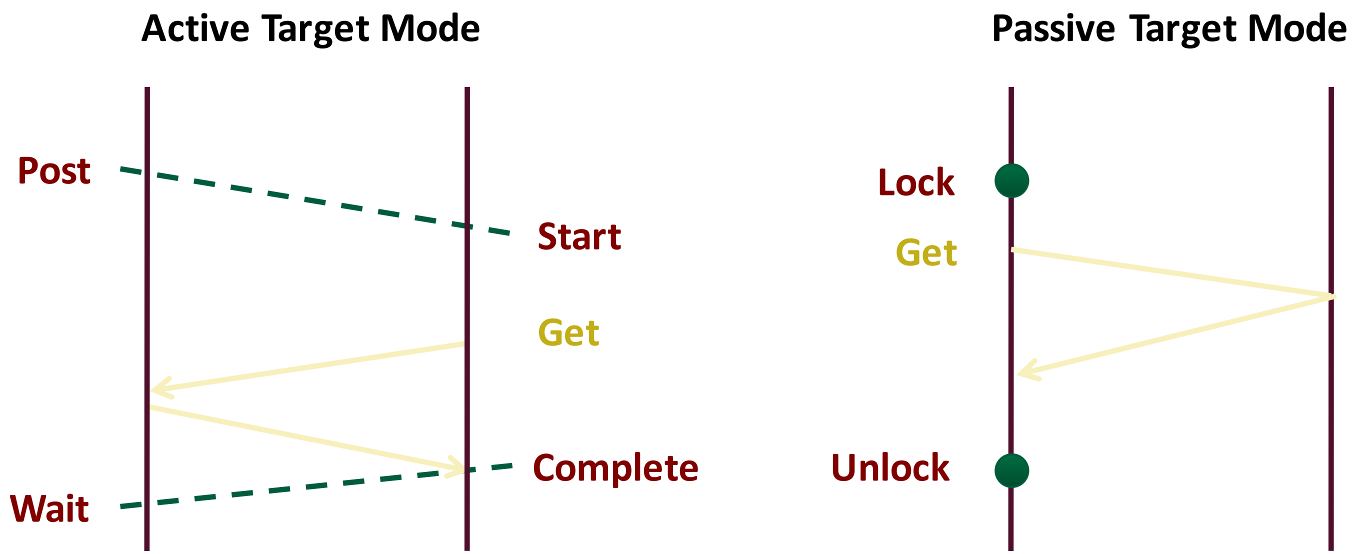
PSCW: Generalized Active Target

MPI_Win_post/start(MPI_Group, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)

- **Like FENCE, but origin and target specify who they communicate with**
- **Target: Exposure epoch**
  - Opened with MPI_Win_post
  - Closed by MPI_Win_wait
- **Origin: Access epoch**
  - Opened by MPI_Win_start
  - Closed by MPI_Win_compete
- **All synchronization operations may block, to enforce P-S/C-W ordering**
  - Processes can be both origins and targets



Lock/Unlock: Passive Target Synchronization



- **Passive mode: One-sided, *asynchronous* communication**
  - Target does **not** participate in communication operation
- **Shared memory-like model**

Passive Target Synchronization

```
MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

MPI_Win_unlock(int rank, MPI_Win win)
```

- **Begin/end passive mode epoch**
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs top different processes
  - Concurrent epochs to same process not allowed (affects threads)
- **Lock type**
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

Advanced Passive Target Synchronization

MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)

MPI_Win_flush/flush_local(int rank, MPI_Win win)
MPI_Win_flush_all/flush_local_all(MPI_Win win)

- **Lock_all: Shared lock, passive target epoch to all other processes**
  - Expected usage is long-lived: lock_all, put/get, flush, …, unlock_all
- **Flush: Remotely complete RMA operations to the target process**
  - Flush_all – remotely complete RMA operations to all processes
  - After completion, data can be read by target process or a different process
- **Flush_local: Locally complete RMA operations to the target process**
  - Flush_local_all – locally complete RMA operations to all processes

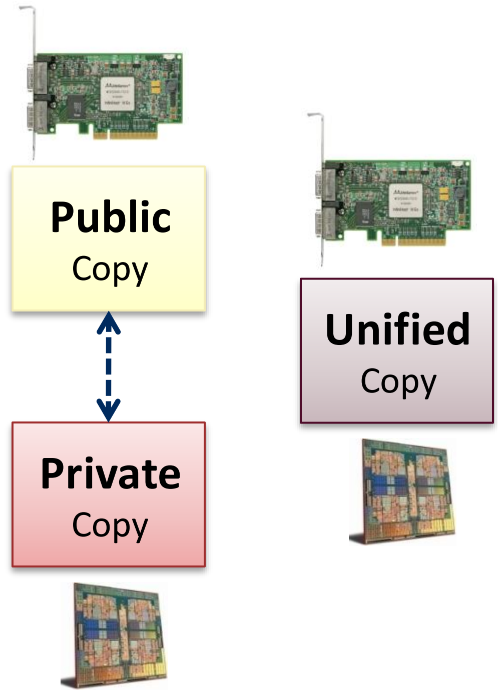
Which synchronization mode should I use, when?

- **RMA communication has low overheads versus send/recv**
  - Two-sided: Matching, queueing, buffering, unexpected receives, etc…
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- **Active mode: bulk synchronization**
  - E.g. ghost cell exchange
- **Passive mode: asynchronous data movement**
  - Useful when dataset is large, requiring memory of multiple nodes
  - Also, when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays
- **Passive target locking mode**
  - Lock/unlock – Useful when exclusive epochs are needed
  - Lock_all/unlock_all – Useful when only shared epochs are needed

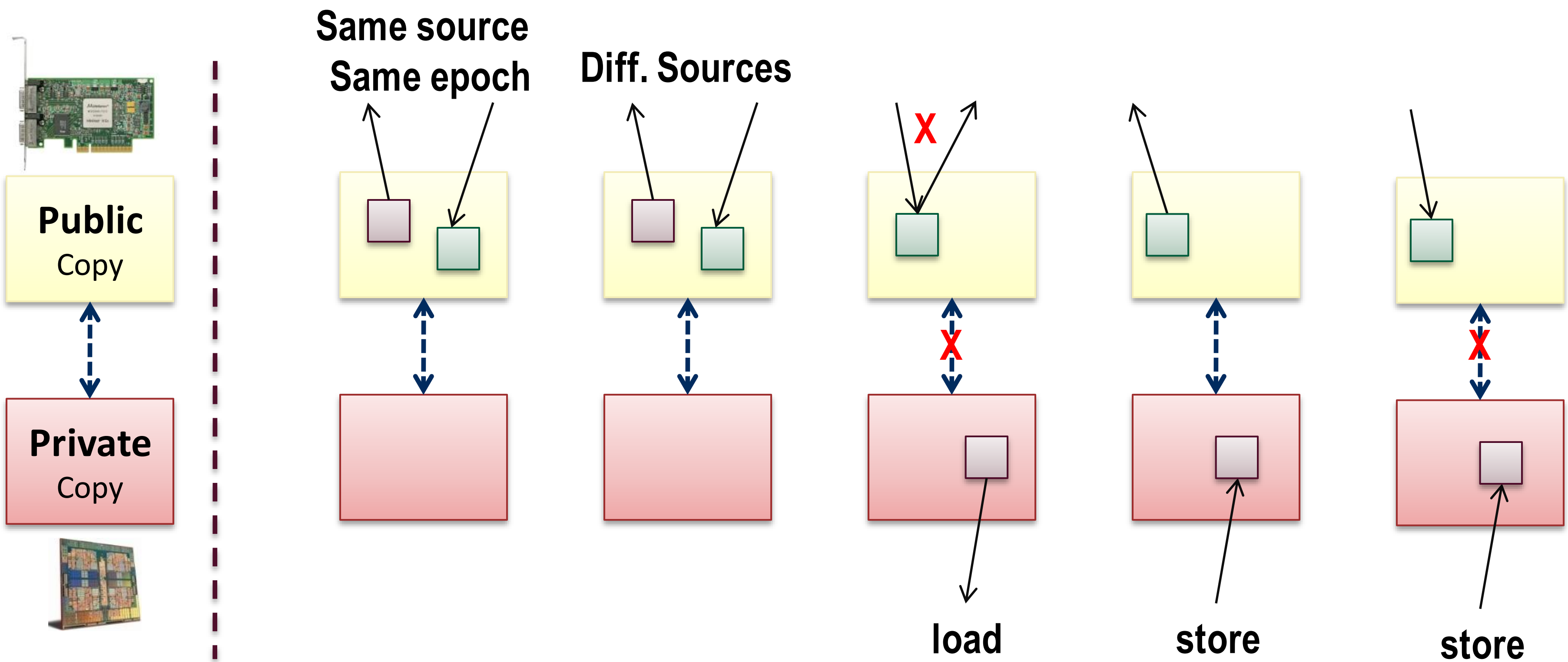
MPI RMA Memory Model

- **MPI-3 provides two memory models: separate and unified**
- **MPI-2: Separate Model**
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don't provide hardware coherence
- **MPI-3: New Unified Model**
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics

    E.g. allows concurrent local/remote access
  - Provides access to full performance potential of hardware

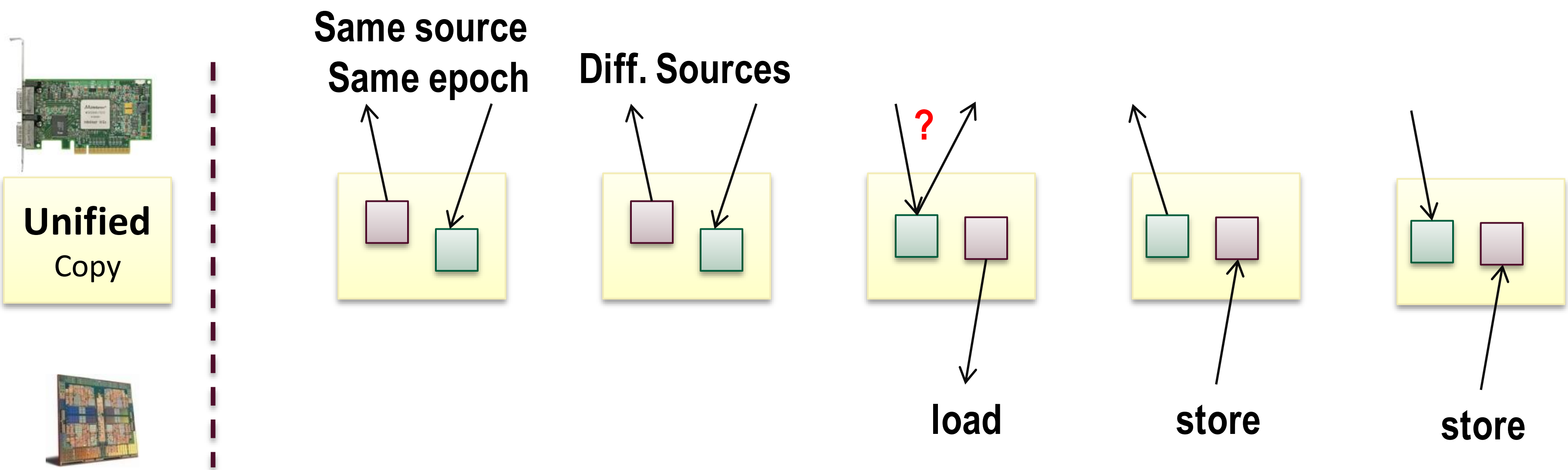


MPI RMA Memory Model (separate windows)



- **Very portable, compatible with non-coherent memory systems**
- **Limits concurrent accesses to enable software coherence**

MPI RMA Memory Model (unified windows)



- **Allows concurrent local/remote accesses**
- **Concurrent, conflicting operations don't "corrupt" the window**
  - Outcome is not defined by MPI (defined by the hardware)
- **Can enable better performance by reducing synchronization**

That’s it folks

- **Thanks for your attention and contributions to the class ☺**
- **Good luck (better: success!) with your project**
  - Don’t do it last minute!
- **Same with the final exam!**
  - Di 21.01., 09:00-11:00 (watch date and room in edoz)
- **Do you have any generic questions?**
  - Big picture?
  - Why did we learn certain concepts?
  - Why did we not learn certain concepts?
  - Anything else (comments are very welcome!)