
CS 267

**Lecture 10: Advanced MPI and
Collective Communication Algorithms**

Aydin Buluc

<https://sites.google.com/lbl.gov/cs267-spr2020/>

Distributed deep learning is all about collectives

TORCH.DISTRIBUTED Backends

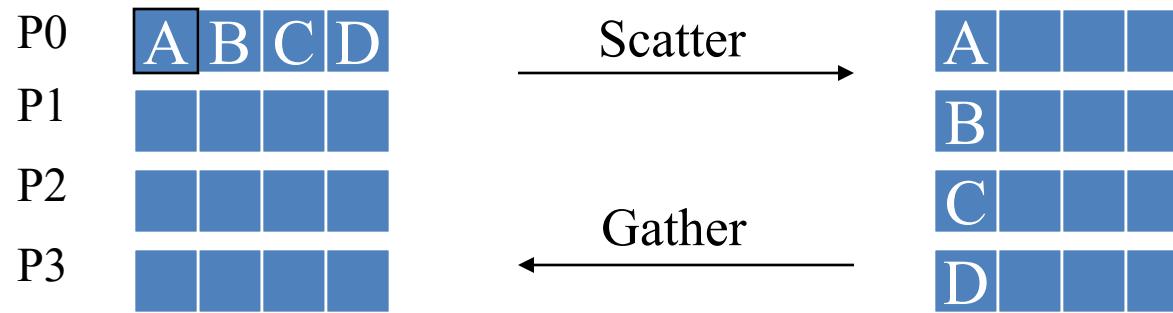
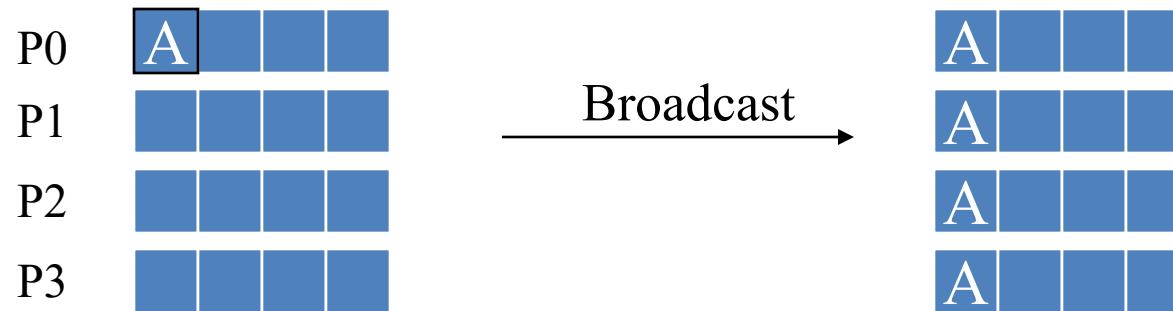
`torch.distributed` supports three backends, each with different capabilities. The table below shows which functions are available for use with CPU / CUDA tensors. MPI supports CUDA only if the implementation used to build PyTorch supports it.

Backend	gloo		mpi		nccl	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✗	✓	?	✗	✗
recv	✓	✗	✓	?	✗	✗
broadcast	✓	✓	✓	?	✗	✓
all_reduce	✓	✓	✓	?	✗	✓
reduce	✓	✗	✓	?	✗	✓
all_gather	✓	✗	✓	?	✗	✓
gather	✓	✗	✓	?	✗	✗
scatter	✓	✗	✓	?	✗	✗

Facebook's gloo: “**Collective** communications library with various primitives for multi-machine training”.

The NVIDIA
Collective
Communications
Library

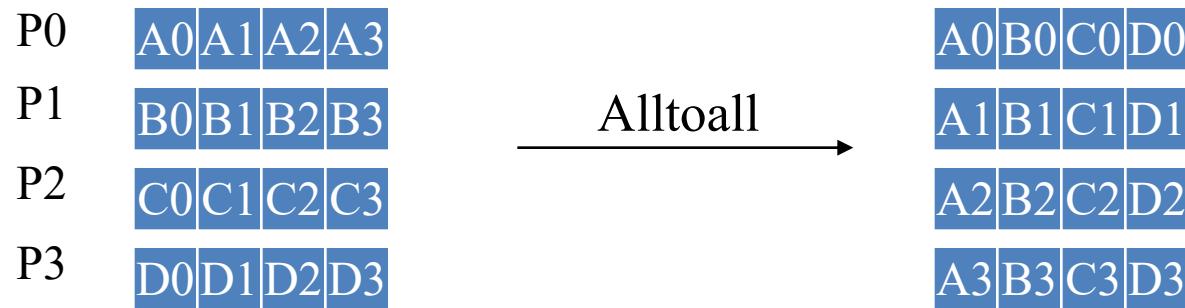
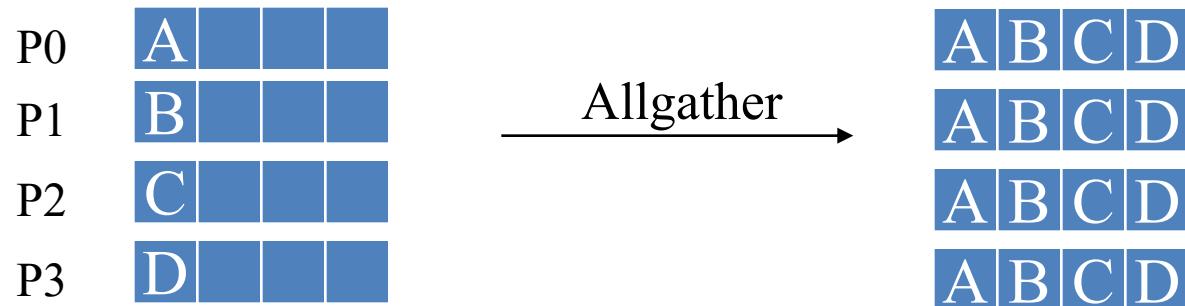
Collective Data Movement



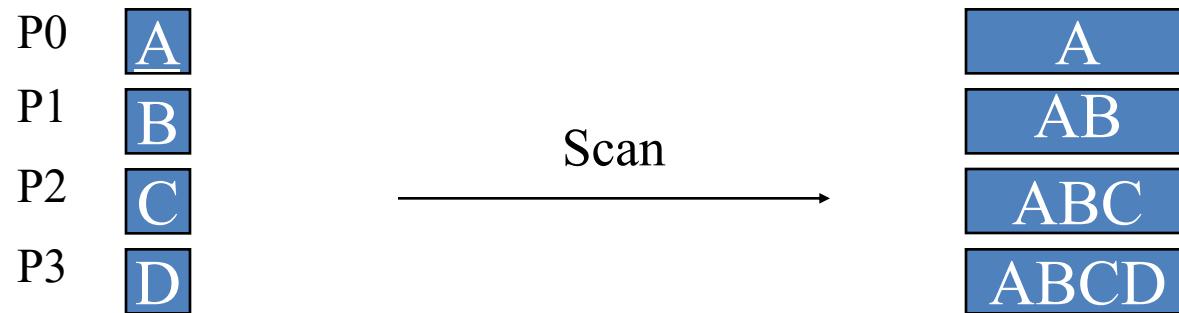
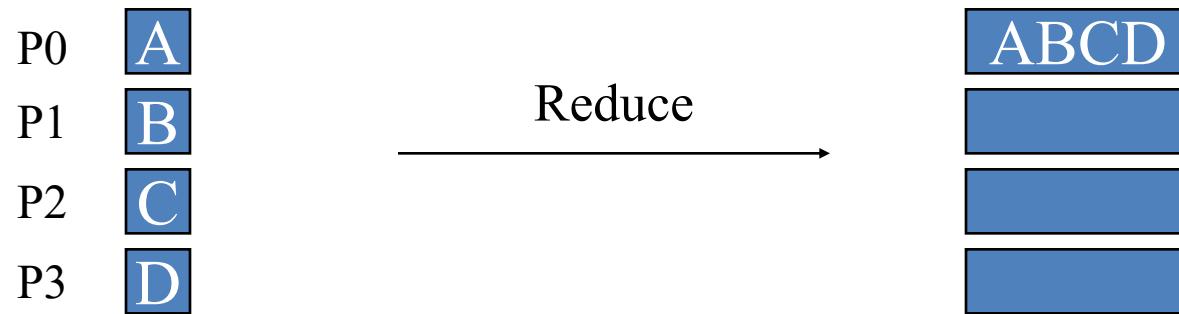
Comments on Broadcast, other Collectives

- **All collective operations must be called by *all* processes in the communicator**
- **MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast**
 - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

More Collective Data Movement



Collective Computation



MPI Collective Routines

- **Many Routines:** `Allgather`, `Allgatherv`,
`Allreduce`, `Alltoall`, `Alltoallv`, `Bcast`,
`Gather`, `Gatherv`, `Reduce`, `Reduce_scatter`,
`Scan`, `Scatter`, `Scatterv`
- **All versions deliver results to all participating processes, not just root.**
- **V versions allow the chunks to have variable sizes.**
- **Allreduce, Reduce, Reduce_scatter, and Scan take both built-in and user-defined combiner functions.**

MXX: An MPI productivity library for C++11

A few annoying redundancies:

- ° Any irregular exchange (e.g, alltoallv, allgatherv) is a multi step process: **(1) exchange counts**, **(2) copy data to buffer**, **(3) allocate space**, **(4) exchange actual data**
- ° have to create a derived data type for any non-PDO data
- ° have to map user defined functions to MPI functions

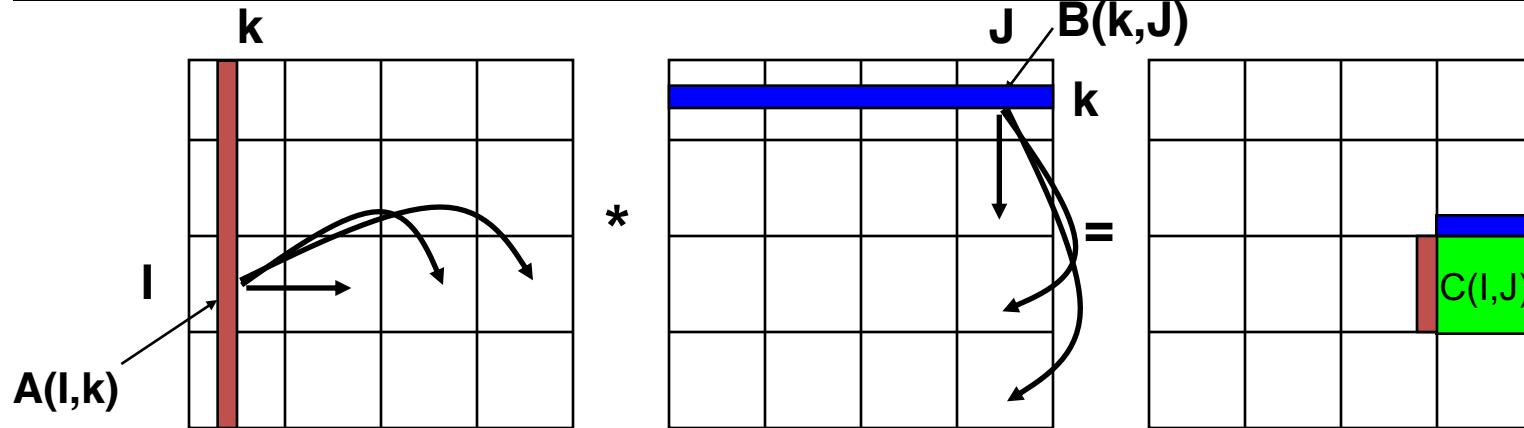
The MXX way:

```
// lets take some pairs and find the one with the max second element
std::pair<int, double> v = ...;
std::pair<int, double> min_pair = mxx::allreduce(v,
    [](const std::pair<int, double>& x,
        const std::pair<int, double>& y){
        return x.second > y.second ? x : y;
});
```

SUMMA Algorithm

- **SUMMA = Scalable Universal Matrix Multiply**
- **Slightly less efficient than Cannon**
... but simpler and easier to generalize
- **Presentation from van de Geijn and Watts**
 - www.netlib.org/lapack/lawns/lawn96.ps
 - Similar ideas appeared many times
- **Used in practice in PBLAS = Parallel BLAS**
 - www.netlib.org/lapack/lawns/lawn100.ps

SUMMA



- I, J represent all rows, columns owned by a processor
- k is a single row or column
 - or a block of b rows or columns
- $C(I, J) = C(I, J) + \sum_k A(I, k) * B(k, J)$
- Assume a p_r by p_c processor grid ($p_r = p_c = 4$ above)
 - Need not be square

MPI_Comm_split

```
int MPI_Comm_split( MPI_Comm comm,
                    int color,
                    int key,
                    MPI_Comm *newcomm)
```

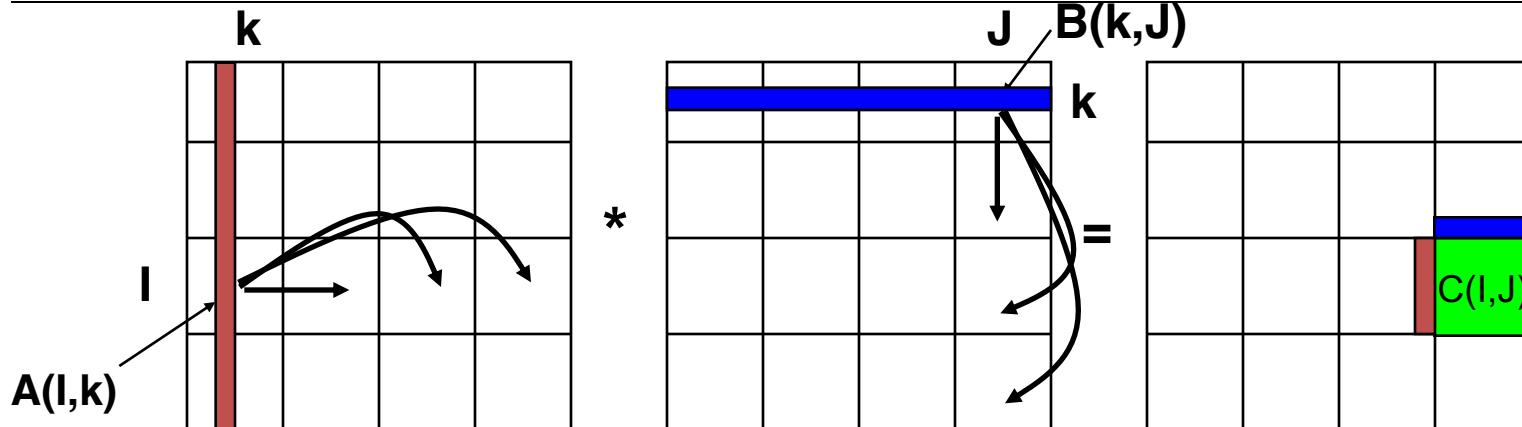
MPI's internal Algorithm:

1. Use MPI_Allgather to get the color and key from each process
2. Count the number of processes with the same color; create a communicator with that many processes. If this process has MPI_UNDEFINED as the color, create a process with a single member.
3. Use key to order the ranks

Color: controls assignment to new communicator

Key: controls rank assignment within new communicator

SUMMA



For $k=0$ to $n-1$... or $n/b-1$ where b is the block size

... = # cols in $A(l,k)$ and # rows in $B(k,J)$

for all $l = 1$ to p_r ... in parallel

owner of $A(l,k)$ broadcasts it to whole processor row

for all $J = 1$ to p_c ... in parallel

owner of $B(k,J)$ broadcasts it to whole processor column

Receive $A(l,k)$ into A_{col}

Receive $B(k,J)$ into B_{row}

$C(\text{myproc}, \text{myproc}) = C(\text{myproc}, \text{myproc}) + A_{col} * B_{row}$

(naïve) SUMMA in MPI

```
void SUMMA(double *mA, double *mB, double *mc, int p_c)
{
    int row_color = rank / p_c; // p_c = sqrt(p) for simplicity
    MPI_Comm row_comm;
    MPI_Comm_split(MPI_COMM_WORLD, row_color, rank, &row_comm);

    int col_color = rank % p_c;
    MPI_Comm col_comm;
    MPI_Comm_split(MPI_COMM_WORLD, col_color, rank, &col_comm);

    for (int k = 0; k < p_c; ++k) {
        if (col_color == k)    memcpy(Atemp, mA, size);
        if (row_color == k)    memcpy(Btemp, mB, size);

        MPI_Bcast(Atemp, size, MPI_DOUBLE, k, row_comm);
        MPI_Bcast(Btemp, size, MPI_DOUBLE, k, col_comm);

        SimpleDGEMM(Atemp, Btemp, mc, N/p, N/p, N/p);
    }
}
```

MPI Built-in Collective Computation Operations

- | | |
|---------------------|----------------------|
| ◦ MPI_MAX | Maximum |
| ◦ MPI_MIN | Minimum |
| ◦ MPI_PROD | Product |
| ◦ MPI_SUM | Sum |
| ◦ MPI LAND | Logical and |
| ◦ MPI LOR | Logical or |
| ◦ MPI LXOR | Logical exclusive or |
| ◦ MPI_BAND | Binary and |
| ◦ MPI_BOR | Binary or |
| ◦ MPI_BXOR | Binary exclusive or |
| ◦ MPI_MAXLOC | Maximum and location |
| ◦ MPI_MINLOC | Minimum and location |

How are collectives implemented in MPI?

- I specifically mention MPI as it enforces **certain semantic rules** (which also means that you can reimplement your own AllReduce if you have more relaxed semantics)
- Example: **MPI_AllReduce**
 1. All processes must receive the same result vector;
 2. Reduction must be performed in canonical order $m_0 + m_1 + \dots + m_{p-1}$ (if the operation is not commutative);
 3. The same reduction order and bracketing for all elements of the result vector is not strictly required, but should be strived for.

How are collectives implemented in MPI?

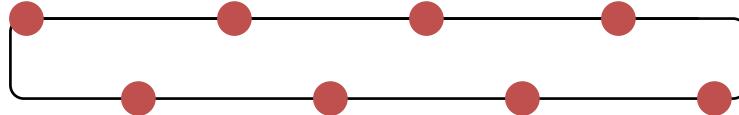
- **Lower bounds:**

Communication	Latency	Bandwidth	Computation
Broadcast	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	—
Reduce(-to-one)	$\lceil \log_2(p) \rceil \alpha$	$n\beta$	$\frac{p-1}{p}n\gamma$
Scatter	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p}n\beta$	—
Gather	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p}n\beta$	—
Allgather	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p}n\beta$	—
Reduce-scatter	$\lceil \log_2(p) \rceil \alpha$	$\frac{p-1}{p}n\beta$	$\frac{p-1}{p}n\gamma$
Allreduce	$\lceil \log_2(p) \rceil \alpha$	$2\frac{p-1}{p}n\beta$	$\frac{p-1}{p}n\gamma$

Note: Pay particular attention to the conditions for the lower bounds given in the text.

AllGather

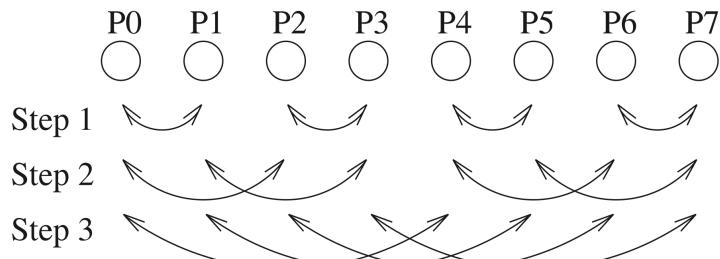
Ring Algorithm



$$T_{\text{ring}} = \alpha (p-1) + \beta n (p-1)/p$$

- **At time t:** send the data you received at time $t-1$ to your right, and receive new data from your left.
- **At time 0,** send your original data
- Optimal bandwidth, high latency

Recursive Doubling Algorithm



$$T_{\text{rec-dbl}} = \alpha \lg(p) + \beta n (p-1)/p$$

- **At time t:** process i exchanges (send/recv) all your current data (its original data plus anything received so far) with process $i \pm 2^t$
- Data exchanged at each step: $n/p, 2n/p, 4n/p, \dots, 2^{\lg(p)-1} n/p$
- Tricky for non-power-of-two

AllGather – The Bruck Algorithm

- At time t : process i receives all your current data from process $i+2^t$ and sends all of its current data to process $i-2^t$ (both modulo p)
 - This regular exchange ends after $\lfloor \lg(p) \rfloor$ steps
 - At the last communication step, instead of receiving/sending all current data, send/recv only the **top ($p-2^{\lfloor \lg(p) \rfloor}$) entries**
 - Requires a final, local shift to get data in correct order.
 - For any p : $T_{\text{block}} = \alpha \lfloor \lg(p) \rfloor + \beta n (p-1)/p$
 - By contrast, recursive doubling takes $2\lfloor \lg(p) \rfloor$ steps for non-power-of-two processor counts

AllGather Performance

- Similar ideas are used in other collectives (e.g. **recursive halving** instead of recursive doubling for **reduce-scatter**) with different local computations (e.g. for **allreduce**, perform a **local reduction** at each step instead of **concatenating** data as in allgather)

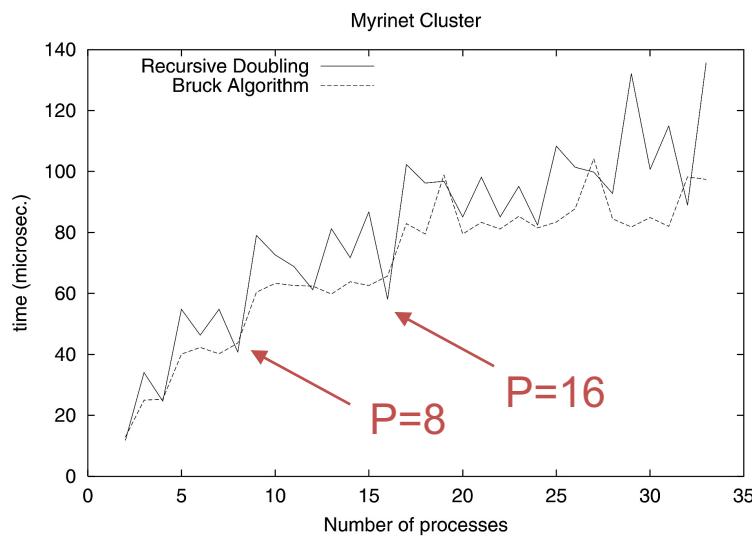


Fig. 3 Performance of recursive doubling versus Bruck allgather for power-of-two and non-power-of-two numbers of processes (message size 16 bytes per process).

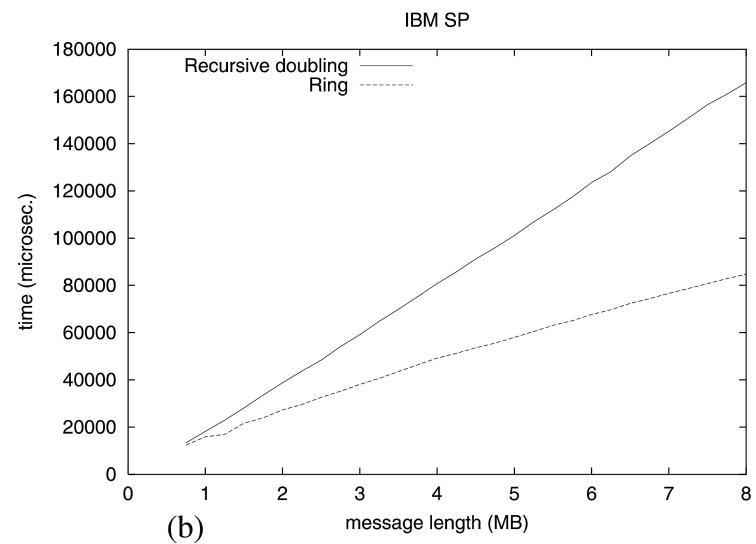


Fig. 5 Ring algorithm versus recursive doubling for long-message allgather (64 nodes). The size on the x-axis is the total amount of data gathered on each process.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH." *The International Journal of High Performance Computing Applications* 19.1 (2005): 49-66

Synchronization

- **MPI_Barrier(comm)**
- **Blocks until all processes in the group of the communicator `comm` call it.**
- **Almost never required in a parallel program**
 - Occasionally useful in measuring performance and load balancing

Nonblocking Collective Communication

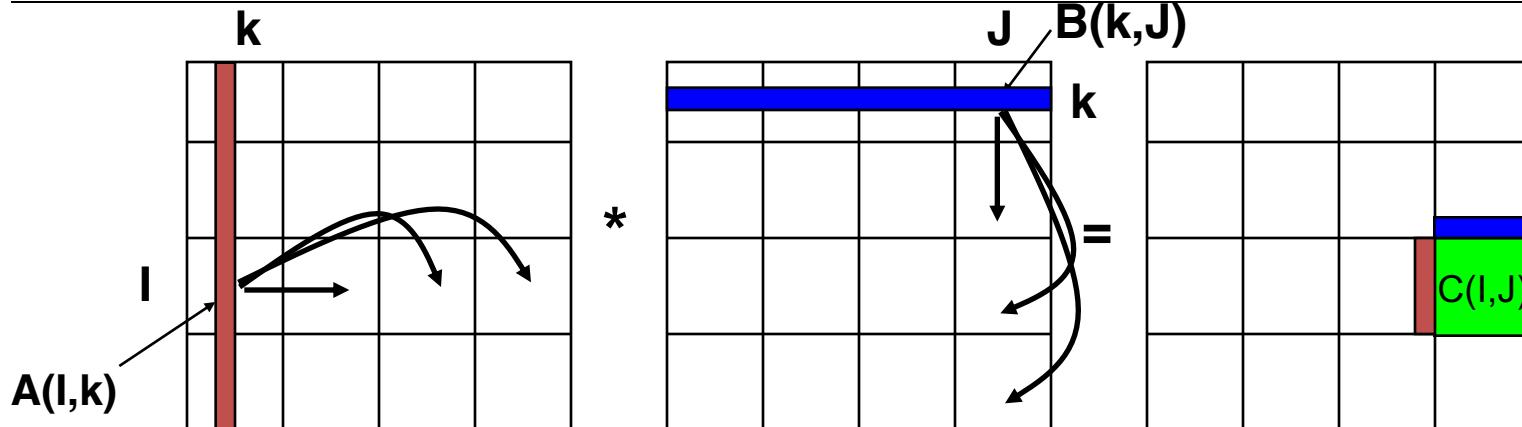
- Nonblocking variants of all collectives
 - **MPI_Ibcast(<bcast args>, MPI_Request *req);**
- Semantics:
 - Function returns no matter what
 - **No guaranteed progress (quality of implementation)**
 - Usual completion calls (wait, test) + mixing
 - Out-of order completion
- Restrictions:
 - No tags, in-order matching
 - Send and vector buffers may not be touched during operation
 - **No matching with blocking collectives**

Nonblocking Collective Communication

Semantic advantages:

- ° Enable asynchronous progression (and manual)
 - Software pipelining
- ° Decouple data transfer and synchronization
 - Noise resiliency!
- ° Allow overlapping communicators
 - See also neighborhood collectives
- ° Multiple outstanding operations at any time
 - Enables pipelining window

(recall) SUMMA



For $k=0$ to $n-1$... or $n/b-1$ where b is the block size
... = # cols in $A(I, k)$ and # rows in $B(k, J)$

for all $I = 1$ to p_r ... in parallel
owner of $A(I, k)$ broadcasts it to whole processor row

for all $J = 1$ to p_c ... in parallel
owner of $B(k, J)$ broadcasts it to whole processor column

Receive $A(I, k)$ into A_{col}

Receive $B(k, J)$ into B_{row}

$C(myproc , myproc) = C(myproc , myproc) + A_{col} * B_{row}$

(potentially overlapped) SUMMA in MPI – Part 1/3

```
void SUMMA(double *mA, double *mB, double *mc, int p_c)
{
    int row_color = rank / p_c; // p_c = sqrt(p) for simplicity
    MPI_Comm row_comm;
    MPI_Comm_split(MPI_COMM_WORLD, row_color, rank, &row_comm);

    int col_color = rank % p_c;
    MPI_Comm col_comm;
    MPI_Comm_split(MPI_COMM_WORLD, col_color, rank, &col_comm);

    double *mA1, *mA2, *mB1, *mB2;
    colsplit(mA, mA1, mA2); // split mA by the middle column
    rowsplit(mB, mB1, mB2); // split mA by the middle row

    if (col_color == 0)    memcpy(Atemp1, mA1, size)
    if (row_color == 0)    memcpy(Btemp1, mB1, size);

    MPI_Request reqs1[2];
    MPI_Request reqs2[2];
    MPI_Ibcast(Atemp1, size, MPI_DOUBLE, k, row_comm, &reqs1[0]);
    MPI_Ibcast(Btemp1, size, MPI_DOUBLE, k, col_comm, &reqs1[1]);
    ...
}
```

(potentially overlapped) SUMMA in MPI – Part 2/3

```
...  
  
for (int k = 0; k < p_c-1; ++k) {  
    if (col_color == k)  memcpy(Atemp2, mA2, size);  
    if (row_color == k)  memcpy(Btemp2, mB2, size);  
  
    MPI_Ibcast(Atemp2, size, MPI_DOUBLE, k, row_comm, &reqs2[0]);  
    MPI_Ibcast(Btemp2, size, MPI_DOUBLE, k, col_comm, &reqs2[1]);  
  
    MPI_Waitall(reqs1, MPI_STATUS_IGNORE);  
    SimpleDGEMM (Atemp1, Btemp1, mC, N/p, N/p, N/p);  
  
    if (col_color == k)  memcpy(Atemp1, mA1, size);  
    if (row_color == k)  memcpy(Btemp1, mB1, size);  
  
    MPI_Ibcast(Atemp1, size, MPI_DOUBLE, k, row_comm, &reqs1[0]);  
    MPI_Ibcast(Btemp1, size, MPI_DOUBLE, k, col_comm, &reqs1[1]);  
  
    MPI_Waitall(reqs2, MPI_STATUS_IGNORE);  
    SimpleDGEMM (Atemp2, Btemp2, mC, N/p, N/p, N/p);  
}  
...
```

(potentially overlapped) SUMMA in MPI – Part 3/3

```
...  
  
if (col_color == p-1)    memcpy(Atemp2, mA2, size);  
if (row_color == p-1)    memcpy(Btemp2, mB2, size);  
  
MPI_Ibcast(Atemp2, size, MPI_DOUBLE, k, row_comm, &reqs2[0]);  
MPI_Ibcast(Btemp2, size, MPI_DOUBLE, k, col_comm, &reqs2[1]);  
  
MPI_Waitall(reqs1, MPI_STATUS_IGNORE);  
SimpleDGEMM (Atemp1, Btemp1, mC, N/p, N/p, N/p);  
  
MPI_Waitall(reqs2, MPI_STATUS_IGNORE);  
SimpleDGEMM (Atemp2, Btemp2, mC, N/p, N/p, N/p);  
}  
...
```

Hybrid Programming with Threads

(slides by Gropp, Thakur, Balaji)

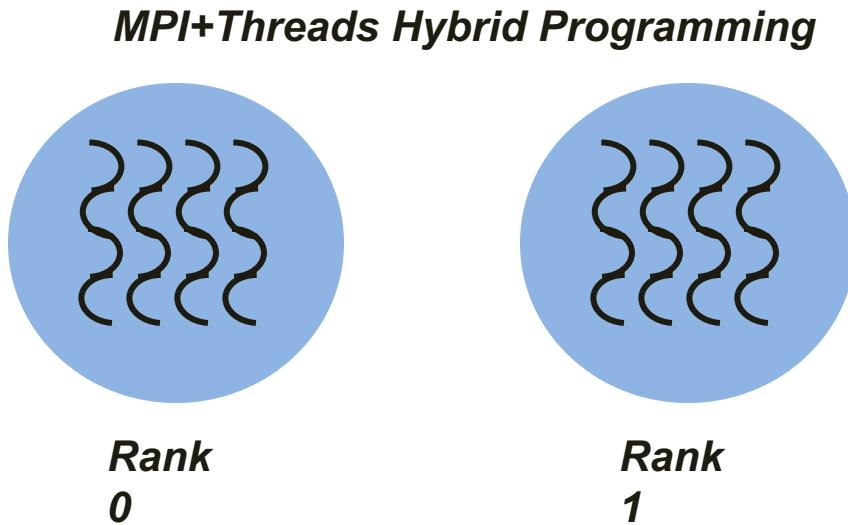
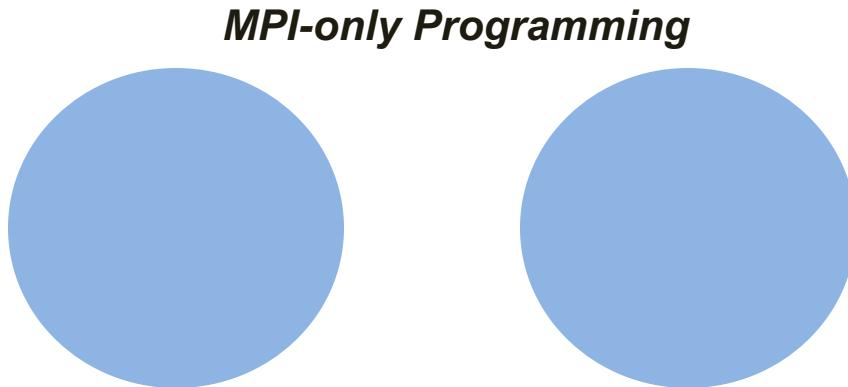
MPI and Threads

- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
 - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
 - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.

Programming for Multicore

- ° **Common options for programming multicore clusters**
 - All MPI
 - MPI between processes both within a node and across nodes
 - MPI internally uses shared memory to communicate within a node
 - MPI + OpenMP
 - Use OpenMP within a node and MPI across nodes
 - MPI + Pthreads
 - Use Pthreads within a node and MPI across nodes
- ° **The latter two approaches are known as “hybrid programming”**

Hybrid Programming with MPI+Threads



- In MPI-only programming, each MPI process has a single program counter
- In MPI+threads hybrid programming, there can be multiple threads executing simultaneously
 - All threads share all MPI objects (communicators, requests)
 - The MPI implementation might need to take precautions to make sure the state of the MPI stack is consistent

MPI's Four Levels of Thread Safety

- ° MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
 - **MPI_THREAD_SINGLE**: only one thread exists in the application
 - **MPI_THREAD_FUNNELED**: multithreaded, but only the main thread makes MPI calls (the one that called `MPI_Init_thread`)
 - **MPI_THREAD_SERIALIZED**: multithreaded, but only one thread *at a time* makes MPI calls
 - **MPI_THREAD_MULTIPLE**: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- ° Thread levels are in increasing order
 - If an application works in FUNNELED mode, it can work in SERIALIZED
- ° MPI defines an alternative to **`MPI_Init`**
 - **`MPI_Init_thread(requested, provided)`**
 - *Application gives level it needs; MPI implementation gives level it supports*

MPI_THREAD_SINGLE

- There are no threads in the system
 - E.g., there are no OpenMP parallel regions

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);

    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```

MPI_THREAD_FUNNELED

- All MPI calls are made by the master thread
 - Outside the OpenMP parallel regions
 - In OpenMP master regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD, 1);

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

    /* Do MPI stuff */

    MPI_Finalize();
    return 0;
}
```

MPI_THREAD_SERIALIZED

- ° Only one thread can make MPI calls at a time
 - Protected by OpenMP critical regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    if (provided < MPI_THREAD_SERIALIZED)
        MPI_Abort(MPI_COMM_WORLD, 1);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
#pragma omp critical
        /* Do MPI stuff */
    }

    MPI_Finalize();
    return 0;
}
```

MPI_THREAD_MULTIPLE

Any thread can make MPI calls any time (w/ restrictions)

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_MULTIPLE)
        MPI_Abort(MPI_COMM_WORLD, 1);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

Threads and MPI

- An implementation is not required to support levels higher than `MPI_THREAD_SINGLE`; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`
- A program that calls `MPI_Init` (instead of `MPI_Init_thread`) should assume that only `MPI_THREAD_SINGLE` is supported
- *A threaded MPI program that does not call `MPI_Init_thread` is an incorrect program (common user error we see)*

Specification of MPI_THREAD_MULTIPLE

- ° ***Ordering:*** When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
 - Ordering is maintained within each thread
 - User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
 - E.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator
 - It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
 - E.g., accessing an info object from one thread and freeing it from another thread
- ° ***Blocking:*** Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

Blocking Calls in MPI_THREAD_MULTIPLE:

Correct Example

	<i>Process 0</i>	<i>Process 1</i>
Thread 1	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 2	MPI_Send(dst=1)	MPI_Send(dst=0)

- ° An implementation must ensure that the above example never deadlocks for any ordering of thread execution (recall **ordering** rules in previous slide)
- ° That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress (recall **blocking** rules in previous slide)

Ordering in MPI_THREAD_MULTIPLE:

Incorrect Example with Collectives

	<i>Process 0</i>	<i>Process 1</i>
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Barrier(comm)	MPI_Barrier(comm)

- ° P0 and P1 can have different orderings of Bcast and Barrier
- ° Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- ° Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI

Ordering in MPI_THREAD_MULTIPLE:

Incorrect Example with Object Management

	<i>Process 0</i>	<i>Process 1</i>
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Comm_free(comm)	MPI_Comm_free(comm)

- ° The user has to make sure that one thread is not using an object while another thread is freeing it
 - This is an ordering issue; the object might get freed before it is used

The Current Situation

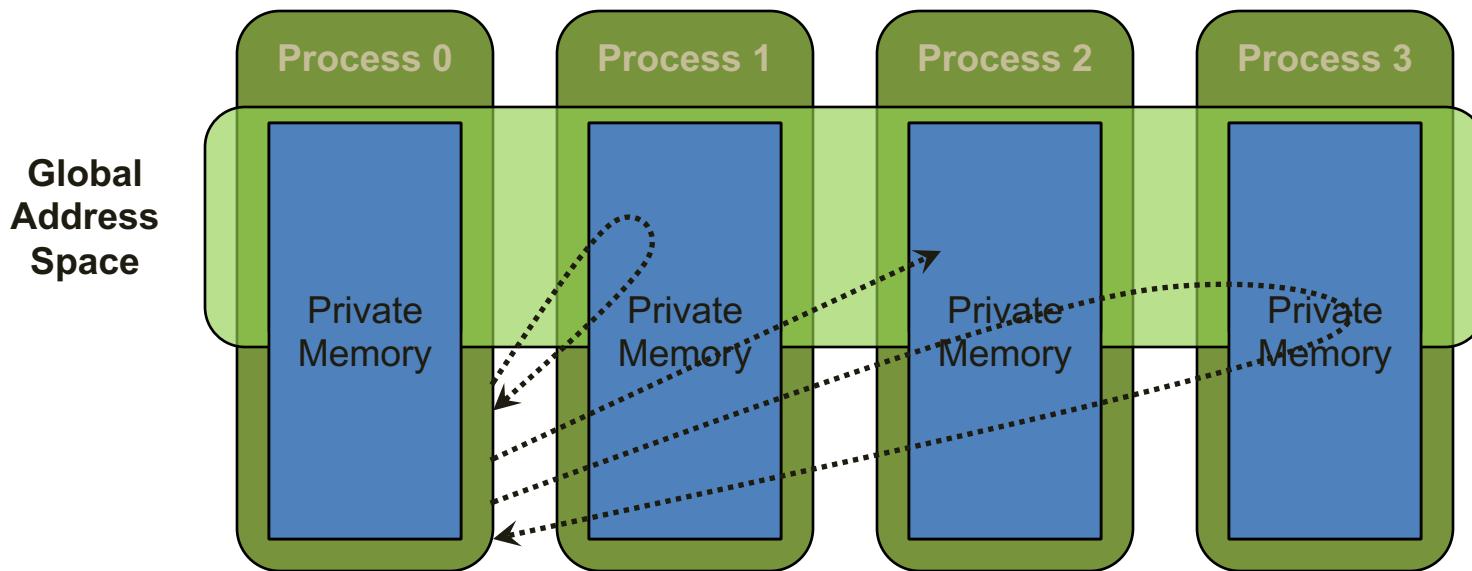
- All MPI implementations support MPI_THREAD_SINGLE
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
 - Does require thread-safe malloc
 - Probably OK in OpenMP programs
- Many (but not all) implementations support MPI_THREAD_MULTIPLE
 - Hard to implement efficiently though (lock granularity issue)
- “Easy” OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
 - So don't need “thread-safe” MPI for many hybrid programs
 - But watch out for Amdahl's Law!

One-sided Communication

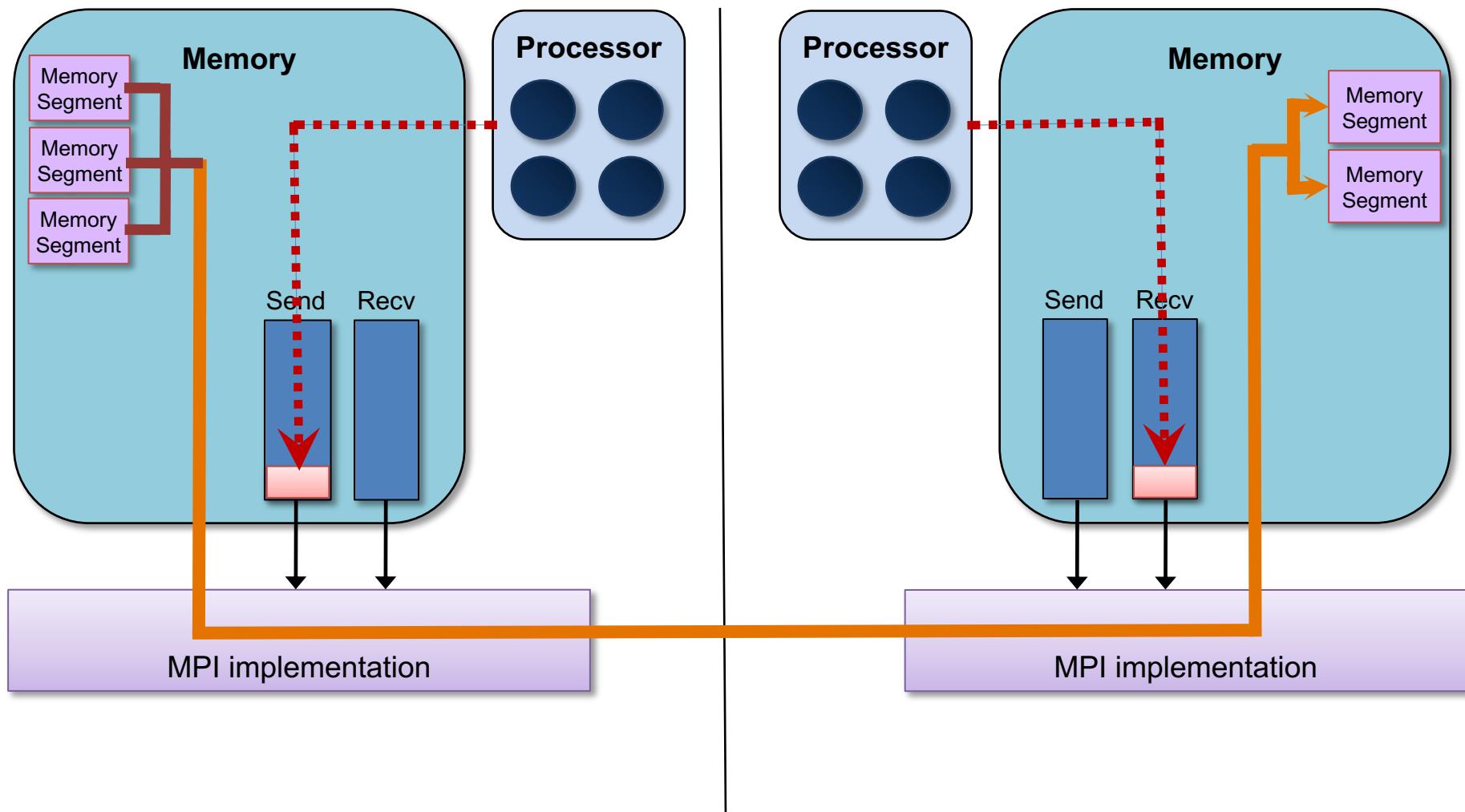
(slides by Gropp, Thakur, Balaji)

One-sided Communication

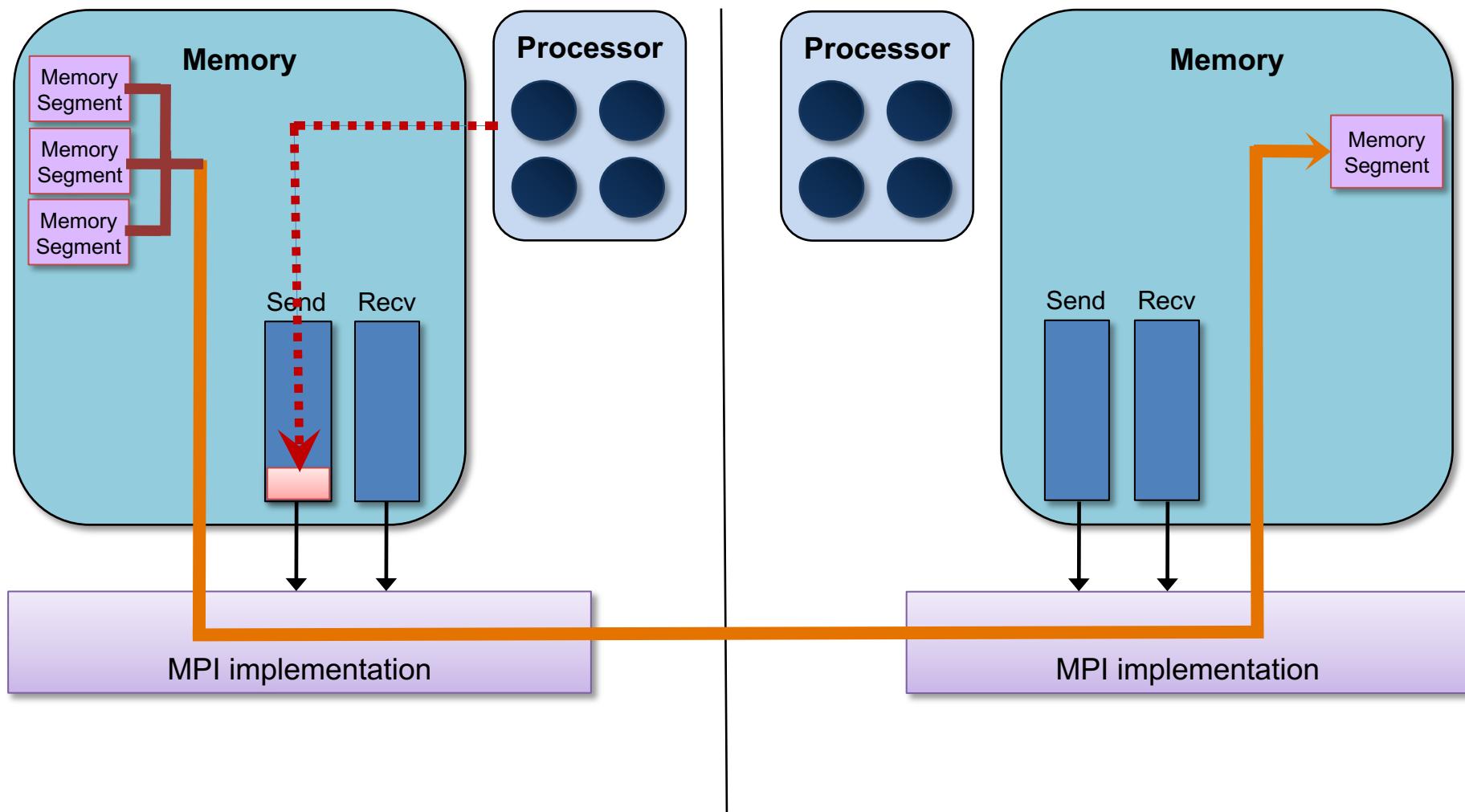
- ° The basic idea of one-sided communication models is to decouple data movement with process synchronization
 - Should be able move data without requiring that the remote process synchronize
 - Each process exposes a part of its memory to other processes
 - Other processes can directly read from or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



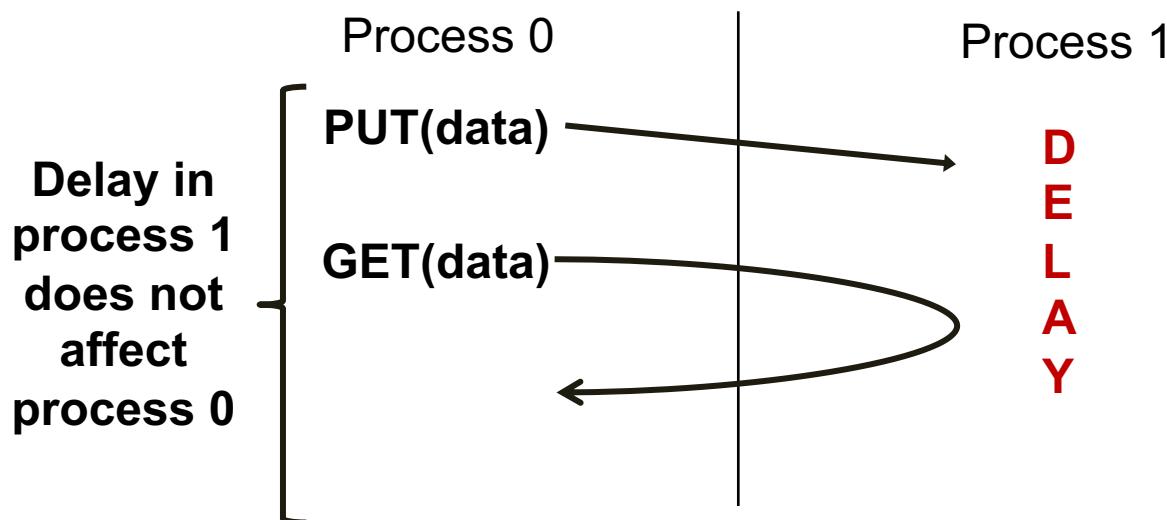
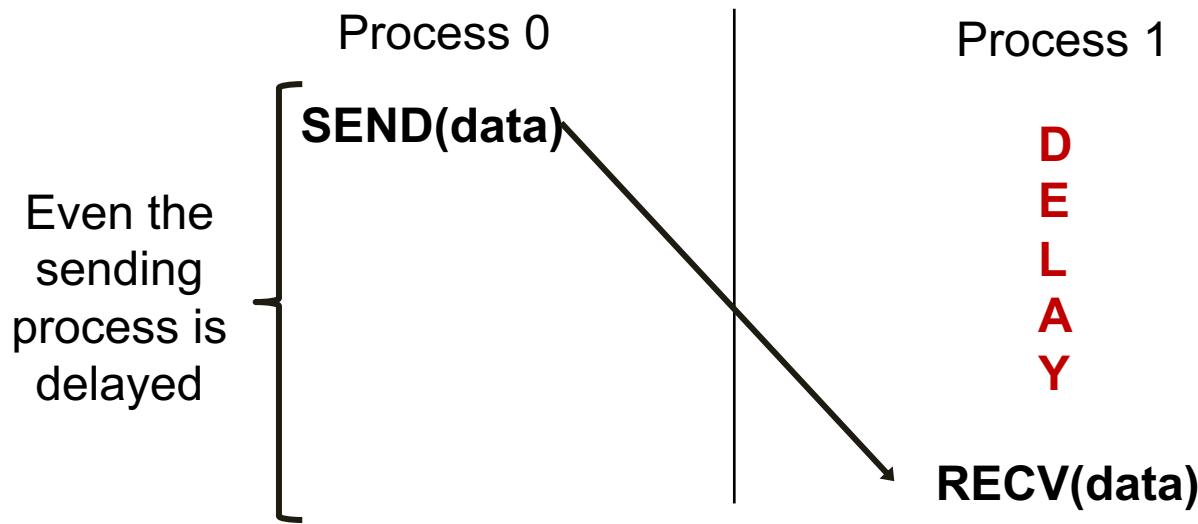
Two-sided Communication Example



One-sided Communication Example



Comparing One-sided and Two-sided Programming



What we need to know in MPI 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

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

MPI_WIN_ALLOCATE

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit,  
                     MPI_Info info, MPI_Comm comm, void *baseptr,  
                     MPI_Win *win)
```

- Create a remotely accessible memory region in an RMA window
 - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
 - size - size of local data in bytes (nonnegative integer)
 - disp_unit - local unit size for displacements, in bytes (positive integer)
 - info - info argument (handle)
 - comm - communicator (handle)
 - baseptr - pointer to exposed local data
 - win - window (handle)

Example with MPI_WIN_ALLOCATE

```
int main(int argc, char ** argv)
{
    int *a;      MPI_Win win;

    MPI_Init(&argc, &argv);

    /* collectively create remote accessible memory in a window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                     MPI_COMM_WORLD, &a, &win);

    /* Array 'a' is now accessible from all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

MPI_WIN_CREATE_DYNAMIC

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,  
                           MPI_Win *win)
```

- **Create an RMA window, to which data can later be attached**
 - Only data exposed in a window can be accessed with RMA ops
- **Initially “empty”**
 - Application can dynamically attach/detach memory to this window by calling **MPI_Win_attach/detach**
 - Application can access data on this window only after a memory region has been attached
- **Window origin is MPI_BOTTOM**
 - Displacements are segment addresses relative to **MPI_BOTTOM**
 - Must tell others the displacement after calling attach

Example with MPI_WIN_CREATE_DYNAMIC

```
int main(int argc, char ** argv)
{
    int *a;      MPI_Win win;

    MPI_Init(&argc, &argv);
    MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* create private memory */
    a = (int *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000*sizeof(int));

    /* Array 'a' is now accessible from all processes */

    /* undeclare remotely accessible memory */
    MPI_Win_detach(win, a);  free(a);
    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

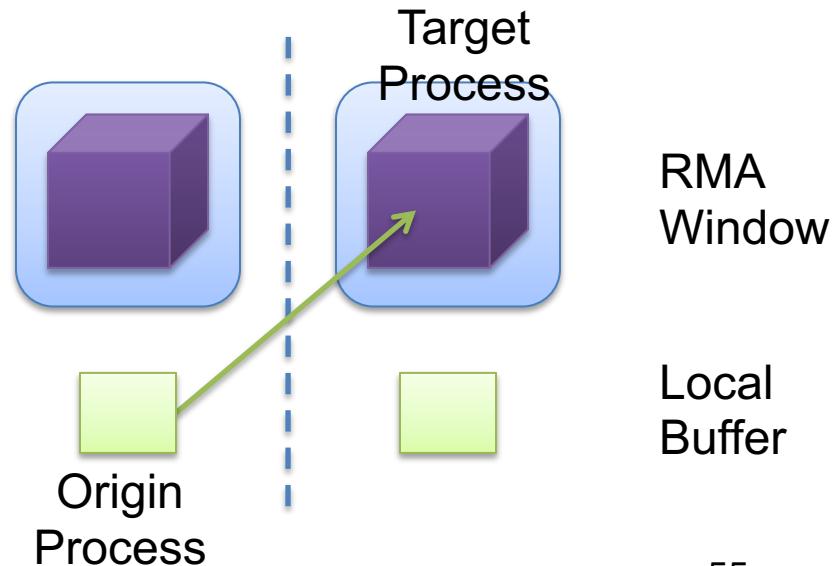
Data movement

- **MPI provides ability to read, write and atomically modify data in remotely accessible memory regions**
 - MPI_PUT
 - MPI_GET
 - MPI_ACCUMULATE
 - MPI_GET_ACCUMULATE
 - MPI_COMPARE_AND_SWAP
 - MPI_FETCH_AND_OP

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)
```

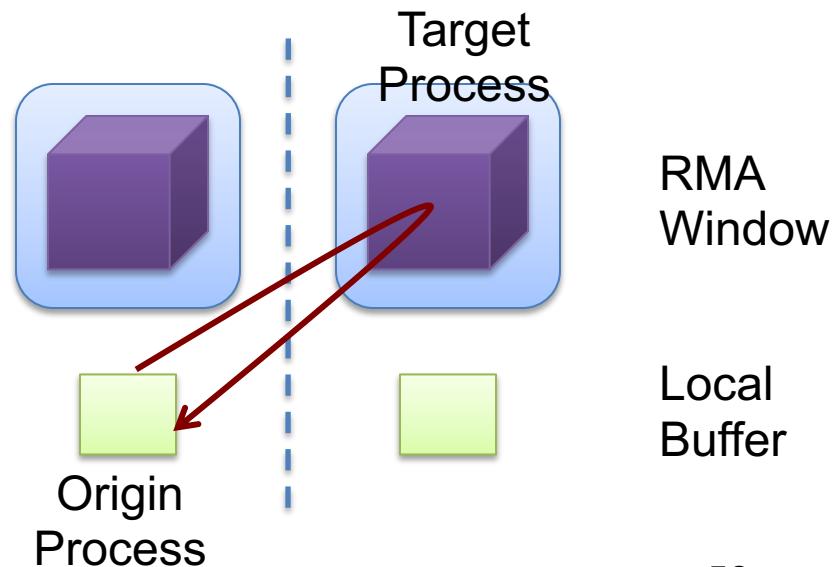
- ° Move data from origin, to target
- ° Separate data description triples for origin and target



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 to origin, from target



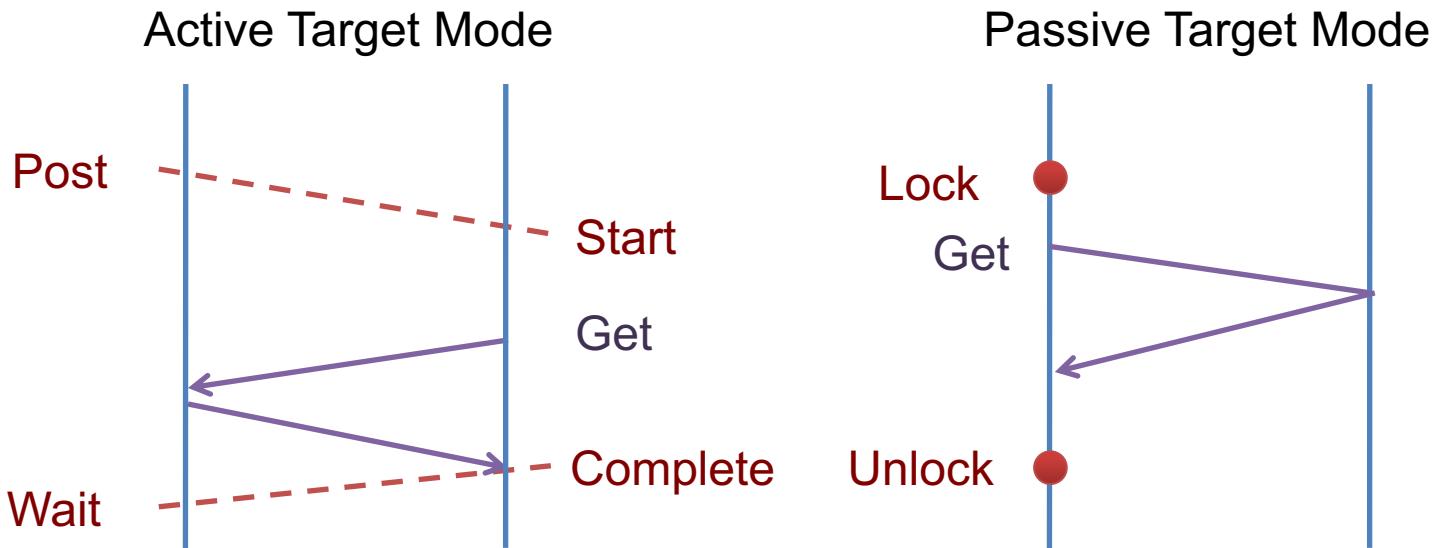
RMA Synchronization Models

- 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

Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent Put/Accumulate undefined
 - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which they occurred
 - Atomic put: Accumulate with op = MPI_REPLACE
 - Atomic get: Get_accumulate with op = MPI_NO_OP
- Accumulate operations from a given process are ordered by default
 - User can tell the MPI implementation that (s)he does not require ordering as optimization hint
 - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW

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 locktype, 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 to 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

Not Covered

- **Topologies:** map a communicator onto, say, a 3D Cartesian processor grid
 - Implementation can provide ideal logical-to-physical mapping
- **Rich set of I/O functions:** individual, collective, blocking and non-blocking
 - Collective I/O can lead to many small requests being merged for more efficient I/O
- **Task creation and destruction:** change number of tasks during a run
 - Few implementations available