

Advanced MPI

George Bosilca

Nonblocking and collective communications

- Nonblocking communication
 - Prevent deadlocks related to message ordering
 - Overlapping communication/computation
 - If communication progress is provided by the implementation/hardware
- Collective communication
 - Collection of pre-defined routines for generalist communication patterns
 - Optimized by the implementations
- Nonblocking collective communication
 - Combines both advantages
 - System noise/imbalance resiliency
 - Semantic advantages

Nonblocking communications

- Semantics are simple:
 - Function returns immediately
 - No requirement for progress (more complicated than point-to-point communications)
- E.g.: `MPI_Isend(..., MPI_Request *req);`
- Nonblocking tests:
 - Test, Testany, Testall, Testsome
- Blocking wait:
 - Wait, Waitany, Waitall, Waitsome
- Blocking vs. nonblocking communication
 - Mostly equivalent, nonblocking has constant request management overhead
 - Nonblocking may have other non-trivial overheads

Nonblocking communications

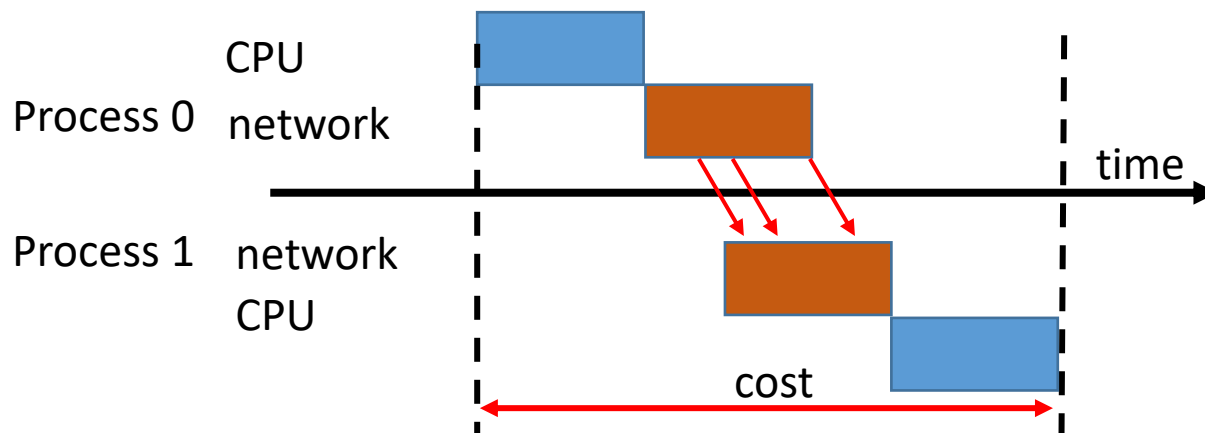
- An important technical detail
 - Eager vs. Rendezvous



- Most/All MPIs switch protocols
 - Small messages are copied to internal remote buffers
 - And then copied to user buffer
 - Frees sender immediately (cf. bsend)
 - Usually below MTU
 - Large messages divided in multiple pieces
 - wait until receiver is ready to prevent temporary memory allocations on the receiver due to unexpected communication
 - Blocks sender until receiver arrived
- Hint: in many cases you can tune these limits (for your environment) and your application
 - Not only for performance reasons but also to minimize the memory used by the MPI library (for internal storage)

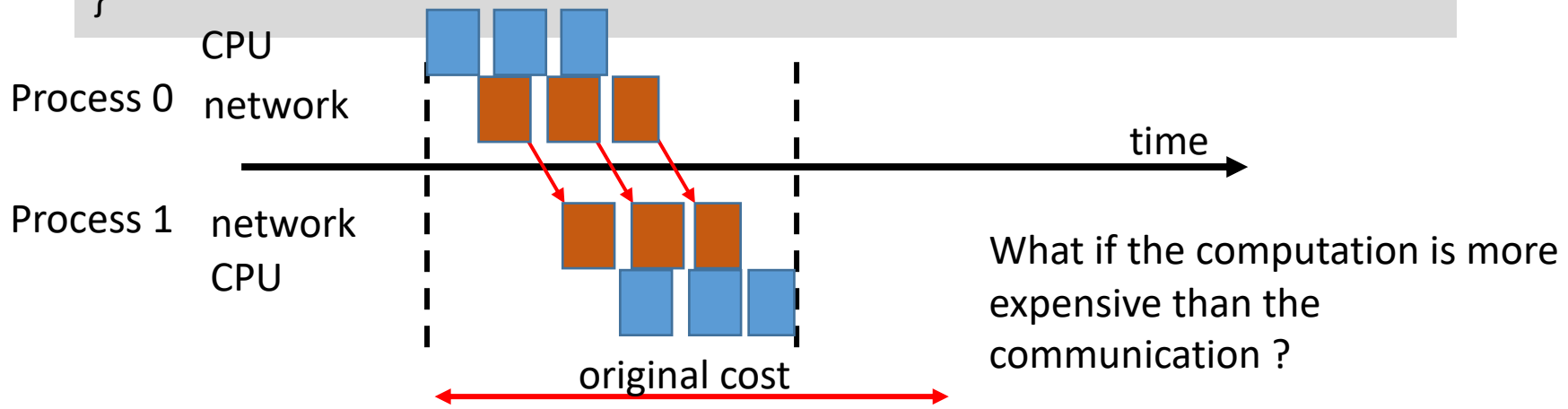
Software Pipelining - Motivation

```
if( 0 == rank ) {  
    for( int i = 0; i < MANY; i++ ) {  
        buf[i] = compute(buf, size, i);  
    }  
    MPI_Send(buf, size, MPI_DOUBLE, 1, 42, comm );  
} else {  
    MPI_Recv(buf, size, MPI_DOUBLE, 0, 42, comm, &status);  
    compute(buf, size);  
}
```



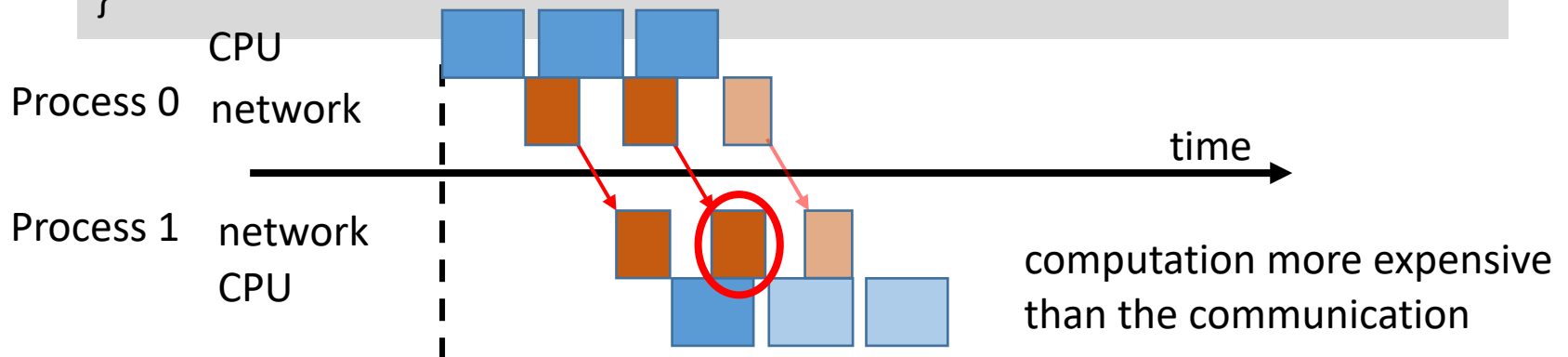
Software Pipelining - Implementation

```
MPI_Request req = MPI_REQUEST_NULL;
if( 0 == rank ) {
    for( int b = 0; b < (size / BSIZE); b++ ) {
        MPI_wait( req, &status); /* complete previous step */
        for( int i = b * BSIZE; i < ((b+1) * BSIZE); i++ )
            buf[i] = compute(buf, size, i);
        MPI_Isend(&buf[b * BSIZE], BSIZE, MPI_DOUBLE, 1, 42, comm, &req );
    }
} else {
    for( int b = 0; b < (size / BSIZE); b++ ) {
        MPI_Recv(&buf[b*BSIZE], BSIZE, MPI_DOUBLE, 0, 42, comm, &status);
        compute(&buf[b*BSIZE], BSIZE);
    }
}
```



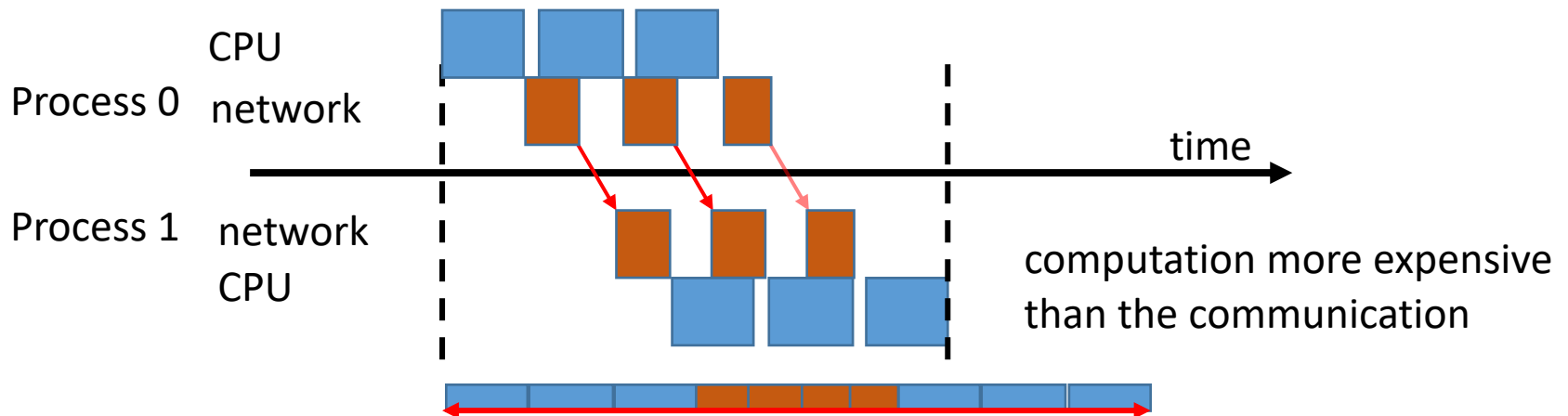
Software Pipelining - Implementation

```
MPI_Request req = MPI_REQUEST_NULL;
if( 0 == rank ) {
    for( int b = 0; b < (size / BSIZE); b++ ) {
        MPI_wait( req, &status); /* complete previous step */
        for( int i = b * BSIZE; i < ((b+1) * BSIZE); i++ )
            buf[i] = compute(buf, size, i);
        MPI_Isend(&buf[b * BSIZE], BSIZE, MPI_DOUBLE, 1, 42, comm, &req );
    }
} else {
    for( int b = 0; b < (size / BSIZE); b++ ) {
        MPI_Recv(&buf[b*BSIZE], BSIZE, MPI_DOUBLE, 0, 42, comm, &status);
        compute(&buf[b*BSIZE], BSIZE);
    }
}
```



Software Pipelining - Implementation

```
MPI_Request req[2] = {MPI_REQUEST_NULL};
if( 0 == rank ) {
    /* keep the same send code */
} else { idx = 0;
    MPI_Irecv(&buf[0*BSIZE], BSIZE, MPI_DOUBLE, 0, 42, comm, &req[idx]);
    for( int b = 0; b < (size / BSIZE); b++ ) {
        MPI_Wait(&req[idx], &status);
        if( (b+1)*BSIZE < size ) { idx = (idx + 1) % 2;
            MPI_Irecv(&buf[(b+1)*BSIZE], BSIZE, ..., comm, &req[idx]); }
        compute(&buf[b*BSIZE], BSIZE);
    }
}
```



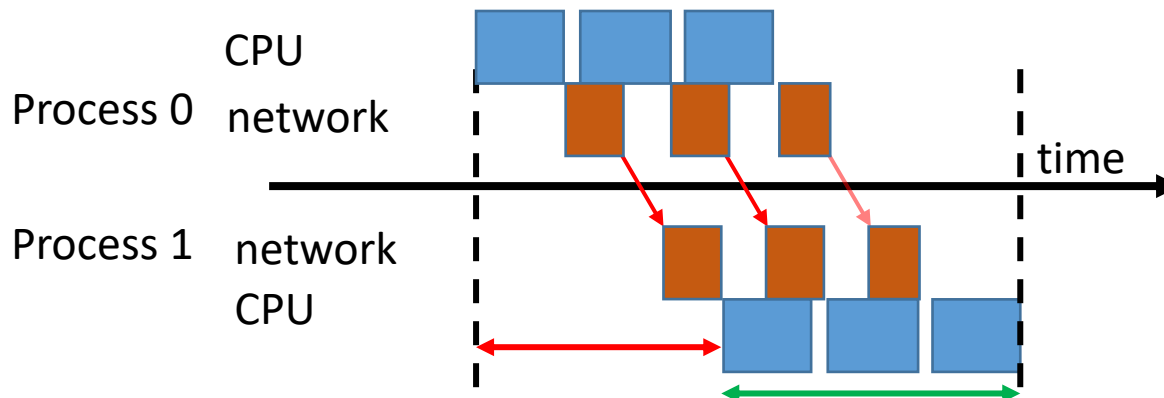
Software pipelining - modelization

- No pipeline

- $T = T_{\text{comp}}(s) + T_{\text{comm}}(s) + T_{\text{startc}}(s) + T'_{\text{comp}}(s)$

- Pipeline

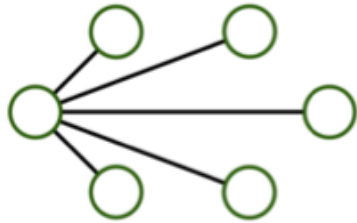
- $T = T_{\text{comp}}(bs) + T_{\text{comm}}(bs) + T_{\text{startc}}(bs) +$
 $\text{nblocks} * \max(T_{\text{comp}}(bs), T_{\text{comm}}(bs), T_{\text{startc}}(bs), T'_{\text{comp}}(bs))$



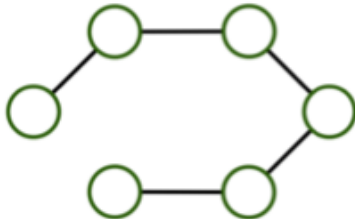
Communicators - Collectives

- Simple classification by operation class
- **One-To-All** (simplex mode)
 - One process contributes to the result. All processes receive the result.
 - MPI_Bcast
 - MPI_Scatter, MPI_Scatterv
- **All-To-One** (simplex mode)
 - All processes contribute to the result. One process receives the result.
 - MPI_Gather, MPI_Gatherv
 - MPI_Reduce
- **All-To-All** (duplex mode)
 - All processes contribute to the result. All processes receive the result.
 - MPI_Allgather, MPI_Allgatherv
 - MPI_Alltoall, MPI_Alltoallv
 - MPI_Allreduce, MPI_Reduce_scatter
- **Other**
 - Collective operations that do not fit into one of the above categories.
 - MPI_Scan
 - MPI_Barrier
- Common semantics:
 - Blocking semantics (return when complete)
 - Therefore no tags (communicators can serve as such)
 - Not necessarily synchronizing (only barrier and all*)

Collective Communications



- Most algorithms are $\log(P)$
- They classify in 3 major communication patterns
 - Scatter, Gather, Reduce
 - Barrier, AllReduce, Allgather, Alltoall
 - Scan, Exscan



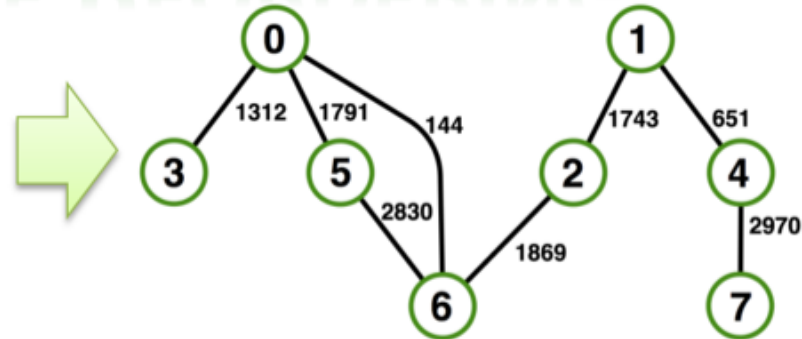
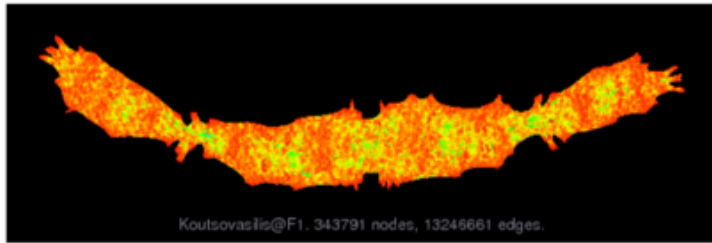
Nonblocking collectives

- Nonblocking variants of all collectives
 - `MPI_Ibcast(, 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
 - `MPI_Cancel` not supported
 - No matching with blocking collectives

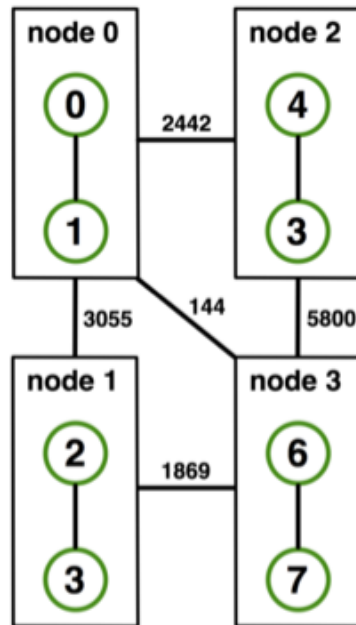
Nonblocking collectives

- 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
- Complex progression
 - MPI's global progress rule!
 - Higher CPU overhead (offloading?)
 - Differences in asymptotic behavior
 - Collective time often
 - Computation
 - Performance modeling (more complicated than for blocking)
 - One term often dominates and complicates overlap

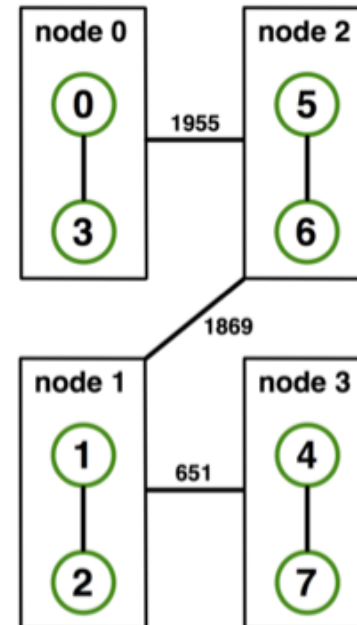
Topologies and Neighborhood



Naïve Mapping



Optimized Mapping



Topomap

Courtesy to Torsten Hoefler

- Rank reordering (transform the original, resource manager provided allocation) and map the processes on it based on the communication pattern

MPI topologies support

- MPI-1: Basic support Convenience functions
 - Create and query a graph
 - Useful especially for Cartesian topologies
 - Query neighbors in n-dimensional space
 - Non-scalable: the graph knowledge must be global as each rank must specify the full graph
- MPI-2.2: Scalable Graph topology
 - Distributed Graph: each rank specifies its neighbors or arbitrary subset of the graph
- MPI-3.0: Neighborhood collectives
 - Adding communication functions defined on graph topologies (neighborhood of distance one)

Cartesian topology creation

- Specify ndims-dimensional topology
 - Optionally periodic in each dimension (Torus)
- Some processes may return `MPI_COMM_NULL`
 - Product sum of dims must be $\leq P$
- Reorder argument allows for topology mapping
 - Each calling process may have a new rank in the created communicator
 - Application must adapt to rank changing between the old and the new communicator, i.e. data must be manually remapped
- MPI provides support for creating the dimensions array ("square" topologies via `MPI_Dims_create`)
 - Non-zero entries on the dims array will not be changed

```
MPI_Cart_create(MPI_Comm old_comm,  
                int ndims, const int*dims, const int *periods,  
                int reorder, MPI_Comm *comm)  
MPI_Dims_create(int nnodes, int ndims, int *dims)
```


Graph Creation

- `nnodes` is the total number of nodes in the graph
- `index[i]` stores the total number of neighbors for the first `i` nodes (sum)
 - Acts as offset into edges array
- `edges` stores the edge list for all processes
 - Edge list for process `j` starts at `index[j]` in `edges`
 - Process `j` has `index[j+1]-index[j]` edges
- Each process must know the entire topology
 - Not scalable

```
MPI_Graph_create(MPI_Comm comm_old, int nnodes,  
                 const int *index, const int *edges, int reorder,  
                 MPI_Comm *comm_graph)
```

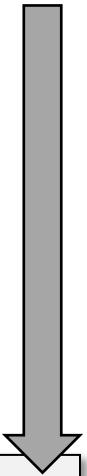
Distributed graph creation

- Scalable, allows distributed graph specification
 - Each nodes specifies either the local neighbors or any edge in the graph (knowledge is now globally distributed)
- Specify edge weights
 - Optimization opportunity for reordering despite the fact that the meaning is undefined
 - Each edge must be specified twice, once as out-edge (at the source) and once as in-edge (at the dest)
- Info arguments
 - Communicate assertions of semantics to the MPI library
 - E.g., semantics of edge weights

Distributed graph creation

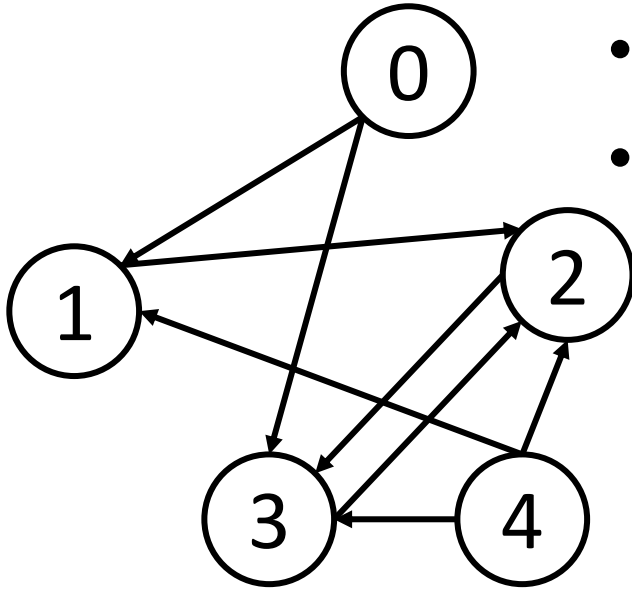
```
MPI_Dist_graph_create_adjacent(MPI_Comm old_comm  
    int indegree, const int sources[], const int sourceweights[],  
    int outdegree, const int destinations[], const int destweights[],  
    MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

- n – number of source nodes
- sources – n source nodes
- degrees – number of edges for each source
- destinations, weights – dest. processor specification
- info, reorder – as usual
- MPI_Dist_graph_create requires global communications to redistribute the information (as each process will eventually need to know it's neighbors)



```
MPI_Dist_graph_create(MPI_Comm comm_old, int n,  
    const int sources[], const int degrees[], const int destinations[],  
    const int weights[], MPI_Info info, int reorder,  
    MPI_Comm*comm_dist_graph)
```

Example: distributed graph creation



- `MPI_Dist_graph_create_adjacent`
- `MPI_Dist_graph_create`

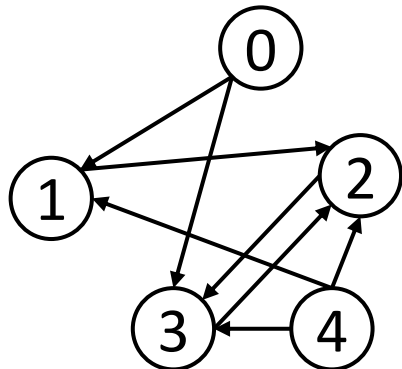
	P0	P1	P2	P3	P4
indegree	{0}	{2}	{3}	{3}	{0}
sources	{}	{0, 4}	{1, 3, 4}	{0, 2, 4}	{}
outdegree	{2}	{1}	{1}	{1}	{3}
destinations	{1, 3}	{2}	{3}	{2}	{1, 2, 3}

- The order is not important, but it must reflect on how the topology will be used
 - Define the buffers order in the neighborhood collectives
- `MPI_Dist_graph_create` can be any permutation of the same edges representation

Distributed Graph query functions

- Query information (the number of neighbors and the neighbors) about the calling process
 - MPI_Dist_graph_neighbors_count return counts for the indegree, outdegree and weight.

```
MPI_Dist_graph_neighbors_count(MPI_Comm comm,  
                               int *indegree, int *outdegree, int *weighted)  
  
MPI_Dist_graph_neighbors(MPI_Comm comm,  
                          int maxindegree, int sources[], int sourceweights[],  
                          int maxoutdegree, int destinations[], int destweights[])
```



	P0	P1	P2	P3	P4
indegree	{0}	{2}	{3}	{3}	{0}
sources	{}	{0, 4}	{1, 3, 4}	{0, 2, 4}	{}
outdegree	{2}	{1}	{1}	{1}	{3}
destinations	{1, 3}	{2}	{3}	{2}	{1, 2, 3}

MPI_Dist_graph_neighbors_count

MPI_Dist_graph_neighbors

Neighborhood Collectives

- Collective communications over topologies
 - They are still **collective** (all processes in the communicator **must** do the call, *including processes without neighbors*)
 - Buffers are accessed in the neighbors sequence
 - Order is determined by order of neighbors as returned by the corresponding query functions ([dist_]graph_neighbors).
 - Defined by order of dimensions, first negative, then positive
 - Cartesians 2*ndims sources and destinations
 - Distributed graphs are directed and may have different numbers of send/recv neighbors
 - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!
 - Every process is root in its own neighborhood (!)

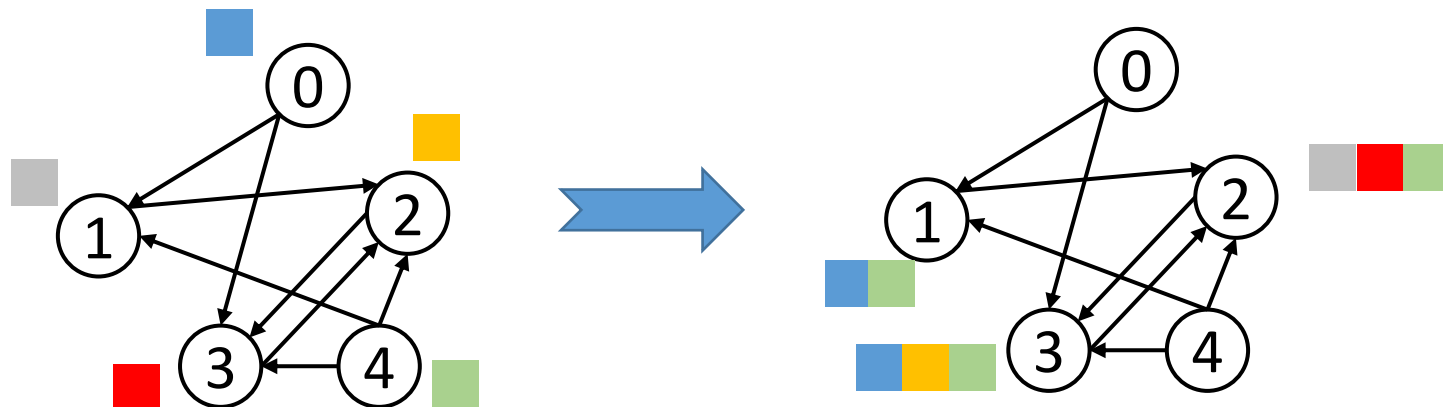
MPI_Neighbor_allgather

- Each process send the same message to all neighbors (the sendbuf)
- Each process receives indegree messages, one from each neighbors in their corresponding order from the query functions
- Similar to MPI_gather where each process is the root on the neighborhood
 - Despite the fact that name starts with all

```
MPI_Neighbor_allgather(  
    const void* sendbuf, int sendcount, MPI_Datatype sendtype,  
    void* recvbuf, int recvcount, MPI_Datatype recvtpe,  
    MPI_Comm comm)
```

MPI_Neighbor_allgather

MPI_Neighbor_allgather(
 const void* sendbuf, int sendcount, MPI_Datatype sendtype,
 void* recvbuf, int recvcount, MPI_Datatype recvtype,
 MPI_Comm comm)



	P0	P1	P2	P3	P4
indegree	{0}	{2}	{3}	{3}	{0}
sources	{}	{0, 4}	{1, 3, 4}	{0, 2, 4}	{}
outdegree	{2}	{1}	{1}	{1}	{3}
destinations	{1, 3}	{2}	{3}	{2}	{1, 2, 3}

Nonblocking versions

- Full support for all nonblocking neighborhood collectives
 - Same collective invocation requirement
 - Matching will be done in order of the collective post for each collective
 - As each communicator can only have a single topology
- Think about the Jacobi where the communications are done with neighbor collectives

One-sided communications

- In MPI we are talking about epoch: a window of memory updates
 - Somewhat similar to memory transactions
 - Everything in an epoch is visible at once on the remote peers
 - Allow to decouple data transfers and synchronizations
- Terms:
 - **Origin process**: Process with the source buffer, initiates the operation
 - **Target process**: Process with the destination buffer, does not explicitly call communication functions
 - **Epoch**: Virtual time where operations are in flight. Data is consistent after new epoch is started.
 - Access epoch: rank acts as origin for RMA calls
 - Exposure epoch: rank acts as target for RMA calls
 - **Ordering**: only for accumulate operations: order of messages between two processes (default: in order, can be relaxed)
 - **Assert**: assertions about how the one sided functions are used, “fast” optimization hints, cf. Info objects (slower)

Overview

- Window creation
 - Static
 - Expose allocated memory: MPI_Win_create
 - Allocate and expose memory: MPI_Win_allocate
 - Dynamic
 - MPI_Win_create_dynamic
- Communications
 - Data movements: Put, Rput, Get, Rget
 - Accumulate (acc, racc, get_acc, rget_acc)
 - Atomic operations (fetch&op, compare and swap)
- Synchronizations
 - Active: Collective (fence); Group
 - Passive: P2P (lock/unlock); One epoch (lock_all)

Memory Exposure

- Collective calls (attached to a communicator)
- Info
 - no_locks – user asserts to not lock win
 - accumulate_ordering – comma-separated rar, war, raw, waw
 - accumulate_ops – same_op or same_op_no_op (default) – assert used ops for related accumulates
 - same_size – if true, user asserts that size is identical on all calling processes (only for MPI_Win_allocate)
- MPI_Win_allocate is preferred, as the implementation is allowed to prepare the memory (pinning and co.)
- MPI_Win_free will free the memory allocated by the MPI library (special care for MPI_Win_allocate)

```
MPI_Win_create(void *base, MPI_Aint size, int disp_unit,  
              MPI_Info info, MPI_Comm comm, MPI_Win *win)  
MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,  
               MPI_Comm comm, void *baseptr, MPI_Win *win)  
[ MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)  
  MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)  
  MPI_Win_detach(MPI_Win win, const void *base)  
MPI_Win_free(MPI_Win *win)
```

One Sided communications

- Put and Get have symmetric behaviors
- Nonblocking, they will complete at the end of the epoch
- Conflicting accesses (for more than one byte) are allowed, but their outcome is undefined
- The request based version can be waited using any MPI completion mechanism (MPI_Test* or MPI_Wait*)
- Similarly to MPI_Send completion of the request only has a local meaning
 - GET: the data is stored in the local buffer
 - PUT: The local buffer can be safely reused (no remote completion)

```
MPI_Put(const 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)  
MPI_Rput(..., MPI_Request *request)
```

One Sided Accumulate

- Atomic update of remote memory based on a combination of the existing data and local data
 - Except if OP is MPI_REPLACE (when it is equivalent to MPI_Put)
 - Non overlapping entries at the target (because memory consistency and ordering accesses)
- MPI_Get_accumulate similar behavior to fetch_and_* operations
 - Accumulate **origin** into **target**, returns content before accumulate in **result**
 - The accumulate operation is atomic
- Order between operations can be relaxed with info (accumulate_ordering = raw, waw, rar, war) during window creation

```
MPI_Accumulate(const 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_Op op, MPI_Win win)
```

```
MPI_Get_accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,  
void *result_addr, int result_count, MPI_Datatype result_datatype,  
int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,  
MPI_Op op, MPI_Win win)
```

One Sided Atomic Operations

- Similar to the atomic operations on the processor
- Fetch_and_op common use case for single element
 - Supposed to be a faster version of the MPI_Get_accumulate because of the restriction on the datatype and count
- Compare and swap
 - Compares **compare** buffer with **target** and replaces value at **target** with **origin** if compare and target are identical. Original target value is returned in **result**.

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

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

One Sided Synchronizations

- Active / Passive

```
MPI_Win_fence(int assert, MPI_Win win)
```

- Collective Synchronization: all operations started before will complete by the time we return
 - Ends the exposure epoch for the entire window
 - Optimization possible via the MPI_MODE_NOPRECEDE assert (no local or remote operations with target the local processor exists)

```
MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_Win_complete(MPI_Win win)
```

```
MPI_Win_wait(MPI_Win win)
```

- Specification of access/exposure epochs separately:
 - Post: start exposure epoch to group, nonblocking
 - Start: start access epoch to group, may wait for post
 - Complete: finish prev. access epoch, origin completion only (not target)
 - Wait: will wait for complete, completes at (active) target
- As asynchronous as possible

One Sided Synchronizations

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

```
MPI_Win_unlock(int rank, MPI_Win win)
```

- Initiates RMA access epoch to rank
 - No concept of exposure epoch
- Unlock closes access epoch
 - Operations have completed at origin and target
- Type:
 - Exclusive: no other process may hold lock to rank
 - More like a real lock, e.g., for local accesses
 - Shared: other processes may also hold lock

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

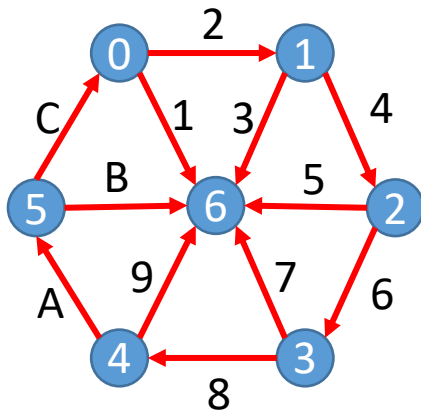
```
MPI_Win_unlock_all(MPI_Win win)
```

- Starts a shared access epoch from origin to all ranks!
 - Not collective!
- Does not really lock anything
 - Opens a different mode of use

More advanced MPI and
mixed programming topics

Extracting messages from MPI

- MPI_Recv delivers each message from a peer in the order in which these messages were send
 - No coordination between peers is possible



- Take a scenario where we have a ring of processors with (P-1) participants, and a lone process that centralize messages from all peers.
- Each processor (except 0) waits for a message from its predecessor in the ring before sending a message to the coordinator
- In which order the messages are received at the coordinator ?
- How we can implement this if each ring participant send a message of a different length ?
- What if we assume a large number of processes?

- Missing functionality: the capability to peek (but not alter) into the network to extract what message will be the next to be locally received
 - Functionality that behaves as MPI_Recv but without altering the matching queue

MPI Probe

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);  
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);  
int MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count);  
MPI_Status a structure containing the fields MPI_SOURCE, MPI_TAG and MPI_ERROR
```

- MPI_ANY_SOURCE and MPI_ANY_TAG can be used as markers for unnamed receives
- The usual usage scenario is probe, memory allocation and then receive
 - How can we use this functionality in a thread safe application when all threads work on the same communicator ?
 - Assume 2 threads (X,Y) doing the probe (P), alloc (A) and receive (R) operation each one on its own context
 - $X_P \rightarrow X_A \rightarrow X_R \rightarrow Y_P \rightarrow Y_A \rightarrow Y_R$
 - What happens if the order of the operations is $X_P \rightarrow X_A \rightarrow Y_P \rightarrow Y_A \rightarrow Y_R \rightarrow X_R$
- The access to the matching queue need to be protected for concurrent accesses

Message Probe

- Functionality that extracts the message from the matching queue but without receiving it
 - Supported by functionality to extract the content of the message into a user provided buffer
 - Any partial ordering between our threads X and Y is now correct: $X_{P'} \rightarrow X_A \rightarrow Y_{P'} \rightarrow Y_A \rightarrow Y_{R'} \rightarrow X_{R'}$

```
int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,
               MPI_Status *status);
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message,
               MPI_Status *status)
int MPI_Mrecv(void *buf, int count, MPI_Datatype type, MPI_Message *message,
               MPI_Status *status);
int MPI_Irecv(void *buf, int count, MPI_Datatype type, MPI_Message *message,
               MPI_Request *request);
```

Collective Communication with threads

- What is happening if multiple threads issue in the same communicator in same time
 - Multiple blocking collectives ?
 - Multiple non-blocking collective with the same datatype and count ?
 - Multiple non-blocking collective with the different datatype and count ?

Shared Memory

- Potential for memory reduction as initialization data can be shared between processes
 - Avoid recomputing the same initial state by multiple applications (on the same node)
 - POSIX provides shared memory regions but (1) not all OSes have support for them and (2) it does not integrate with MPI functionality
- Need functionality to split a communicator in disjoint groups with shared capabilities
 - Similar to `MPI_Comm_split` with architecture aware color (key will then be the rank in the original communicator)
 - Single info key standardized: `MPI_COMM_TYPE_SHARED`
 - Some MPI implementations provide support for different granularities of sharing ([Open MPI](#))

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info,
                        MPI_Comm *newcomm);
```


Shared Memory Window

- In non contiguous cases we need to extract the remote address in order to complete RMA operations
 - As the memory region might be mapped at different addresses in different processes each process local address has no meaning
 - Unlike in Open SHMEM where the RMA operations applied on symmetric memory (!)
 - Only works for windows of type `MPI_WIN_FLAVOR_SHARED` (aka. created via `MPI_Win_allocate_shared`)

```
int MPI_Win_shared_query (MPI_Win win, int rank, MPI_Aint *size, int *disp_unit,  
                          void *baseptr);
```

RMA and pt2pt puzzle ?

- Assuming a correctly initialized window what is the outcome of the following code ?

```
for(i = 0; i < len; a[i] = (double)(10*me+i), i++);
if (me == 0) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Send(NULL, 0, MPI_BYTE, 2, 1001, MPI_COMM_WORLD);
    MPI_Get(a, len, MPI_DOUBLE, 1, 0, len, MPI_DOUBLE, win);
    MPI_Win_unlock(1, win);
    for(i = 0; i < len; i++) printf("a[%d] = %d\n", a[i]);
} else if (me == 2) { /* this should block till 0 releases the lock. */
    MPI_Recv(NULL, 0, MPI_BYTE, 0, 1001, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Put(a, len, MPI_DOUBLE, 1, 0, len, MPI_DOUBLE, win);
    MPI_Win_unlock(1, win);
}
```