Introduction to the features of the MPI-3 standard

Fabio Affinito

Summary

- Introduction
- Nonblocking collectives
- Neighborhood collectives
- RMA and one-sided communications

Message Passing Interface

- MPI is an open standard interface for message passing on memory distributed systems
- Version 1.0 was released in 1994, 2.2 in 2009 and 3.0 in 2012
- MPI 3 contains many enhancements wrt MPI 2.2 in many areas
- MPI 3 is already implemented in many different versions by several vendors
- It tackles problems originating when the number of tasks increase as it happens on modern supercomputers

	MPICH	MVAPICH	Open MPI	Cray MPI	Tianhe MPI	Intel MPI	IBM BG/Q MPI[1]	IBM PE MPICH	IBM Platform	SGI MPI	Fujitsu MPI	MS MPI
NBC	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*
Nbrhood collectives	✔	✔	✔	✔	✔	✔	✔	✔	Q3 ‘15	✔	Q3 ‘15	
RMA	✔	✔	✔	✔	✔	✔	✔	✔	Q3 ‘15	✔	Q3 ‘15	
Shared memory	✔	✔	✔	✔	✔	✔	✔	✔	Q3 ‘15	✔	Q3 ‘15	✔
Tools Interface	✔	✔	✔	✔	✔	✔	✔[2]	✔	Q3 ‘15	✔	Q3 ‘15	*
Comm-creat group	✔	✔	✔	✔	✔	✔	✔	✔	Q3 ‘15	✔	Q3 ‘15	
F08 Bindings	✔	✔	✔	✔	✔	Q3 ‘15	✔		Q3 ‘15	✔	Q3 ‘15	
New Datatypes	✔	✔	✔	✔	✔	✔	✔	✔	Q3 ‘15	✔	Q3 ‘15	*
Large Counts	✔	✔	✔	✔	✔	✔	✔	✔	Q3 ‘15	✔	Q3 ‘15	*
Matched Probe	✔	✔	✔	✔	✔	✔	✔	✔	Q3 ‘15	✔	✔	*
NBC I/O	✔	Q3 ‘15	Q3 ‘15	Q4 ‘15						Q2 ‘16		

Summary

- Introduction
- Nonblocking collectives
- Neighborhood collectives
- RMA and one-sided communications

Communication modes

- Blocking (MPI_*)
  - it does not return until the message data and envelope have been safely away
  - the sender is free to modify the data send buffer
  - depending on the implementation the data can be buffered
- Buffered (MPI_B*)
  - can be started whether or not a matching receive was posted
  - this operation is local and its completion does not depend on the occurrence of a matching receive
- Synchronous mode (MPI_S*)
  - the send will complete successfully only if a matching receive is posted and the receive operation has started to receive the message sent by the synchronous send
  - a send executed in this mode is non-local
- Ready mode (MPI_R*)
  - a send in ready mode may be started only if the matching receive is already posted, otherwise the operation is erroneous and its outcome is undefined
- Non blocking (MPI_I*)
  - a non blocking call initiates the operation but it does not complete it

Non blocking collective communications

- Non blocking point to point benefits:
  - avoid deadlocks
  - overlap communication with computation
- Collective communications benefits:
  - Optimized routines for one-to-all or all-to-all communications
- Non blocking collective communications:
  - Sum of the benefits of both
  - Avoid bottlenecks when large number of MPI tasks
  - Semantic advantages

Non blocking collective communications

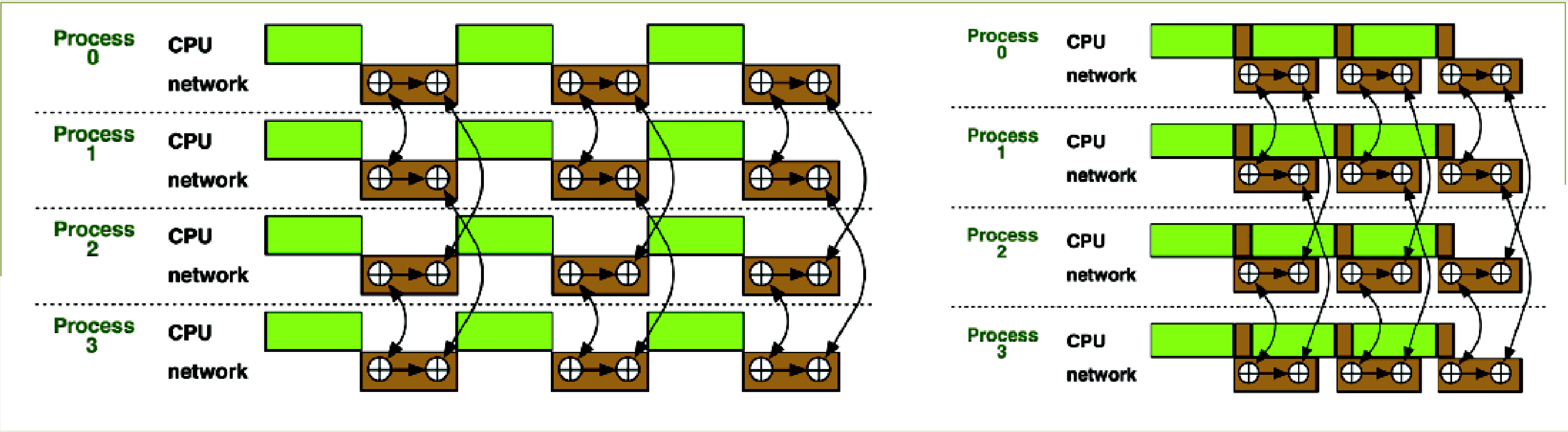
- MPI 3 adds a non blocking variant to all the collective communications
  - example: MPI_Ibcast(<bcasts args>, MPI_Request *req)
- Semantics:
  - it returns no matter what
  - no guaranteed progress
  - usual completion call
  - out of order completion
- Restrictions
  - no tags, no ordering, no in-order
  - no matching with blocking collectives

Non blocking collective communications

- Semantic advantages:
  - Enable asynchronous progression
  - Pipelining
  - Decouples data transfer and synchronization
  - It removes bottlenecks due to a large number of MPI tasks far apart

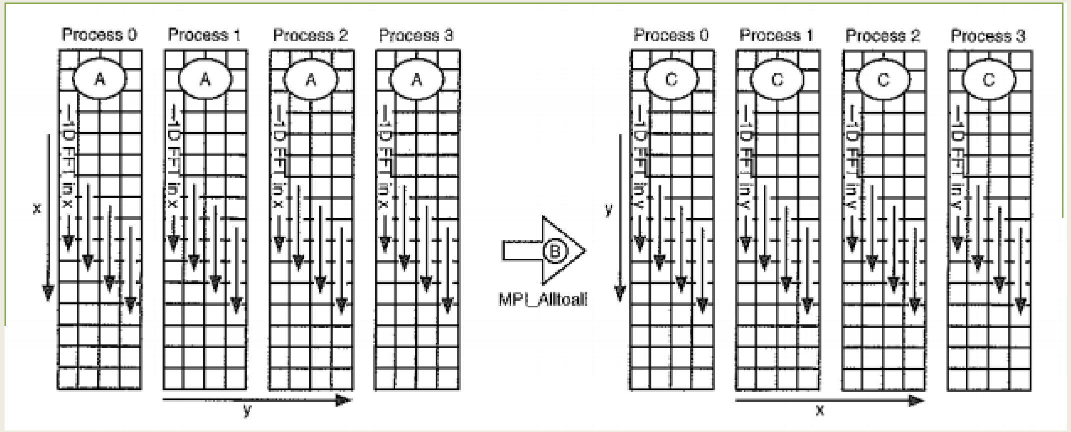
Non blocking collective communications

- Software pipelining



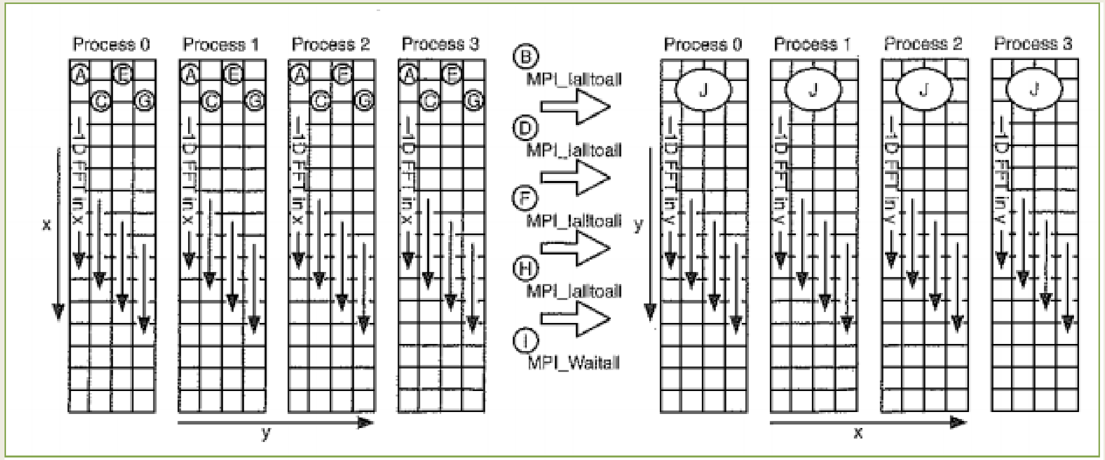
2-D FFT

- 2D-FFT can be distributed among different processes each of them executes a 1D-FFT
- After the 1D-FFT along, for example, the x-direction is completed, an MPI_Alltoall is performed.
- Now, each process can execute a 1D-FFT along the y-direction and a final MPI_Alltoall permits to obtain the complete 2D-FFT



2D-FFT

- The performance can be improved if we start the transposition and, without waiting for the completion, we start working on the FFT along the second direction



Non blocking barrier

- Non blocking barrier? Is it a *contradiction*?
- Not really...
  - Decouple the moment in which processes enter the barrier from the moment in which the synchronization actually happens!
- Example:
  - People are called to a meeting. There's a *late guy*. People is waiting for him, but while they're waiting they can keep doing other things. When the *late guy* arrives, they all can start having the meeting.

DSDE

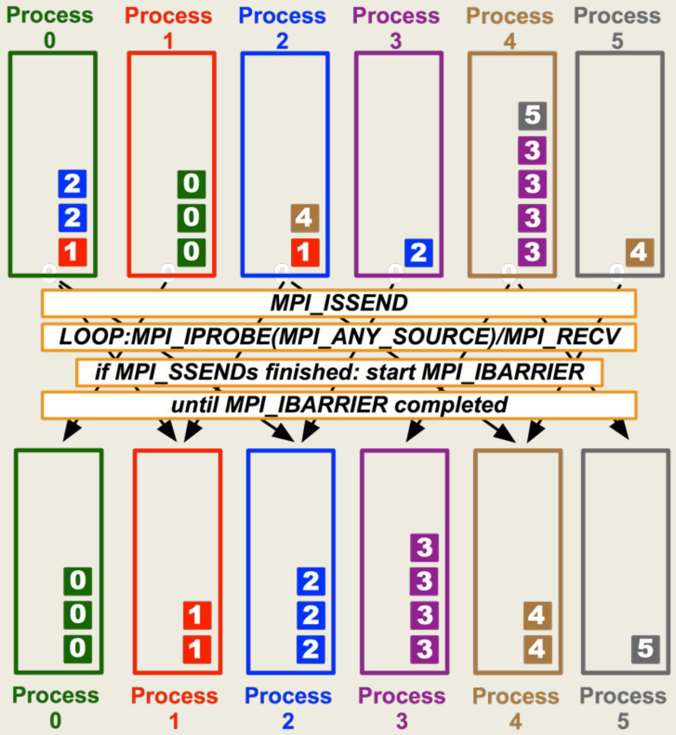
- N-body codes distribute the physical domain across different PEs.
- Computation is divided in two phase: computation of forces and particle movement
- Particles may move from one process to another
- Only the originating process knows which particles are leaving its area and where they are going
- The destination processes typically don't know how much they will receive from the other processes
- This problem is called: Dynamic Sparse Data Exchange

DSDE

- A trivial solution to the DSDE problem is:
  - to exchange data sizes with an MPI_Alltoall
  - that sets up an MPI_Alltoallv for the communication of the actual data
- This simple solution sends p^2 data items for a communicator of size p
- Using other approaches, for example using MPI_Any_Source/MPI_Probe and then MPI_Scatter/Reduce will always require p^2 data items to communicate all the metadata

DSDE with nonblocking barrier

- First phase:
  - each process sends its messages using MPI_Issend
- Second phase:
  - each process checks the completion of the send (with MPI_Iprobe)
- If all the sends are complete the process starts a nonblocking barrier and then continues to receive messages from the other processes in the loop
- The processes exit the loop once the nonblocking barrier completes



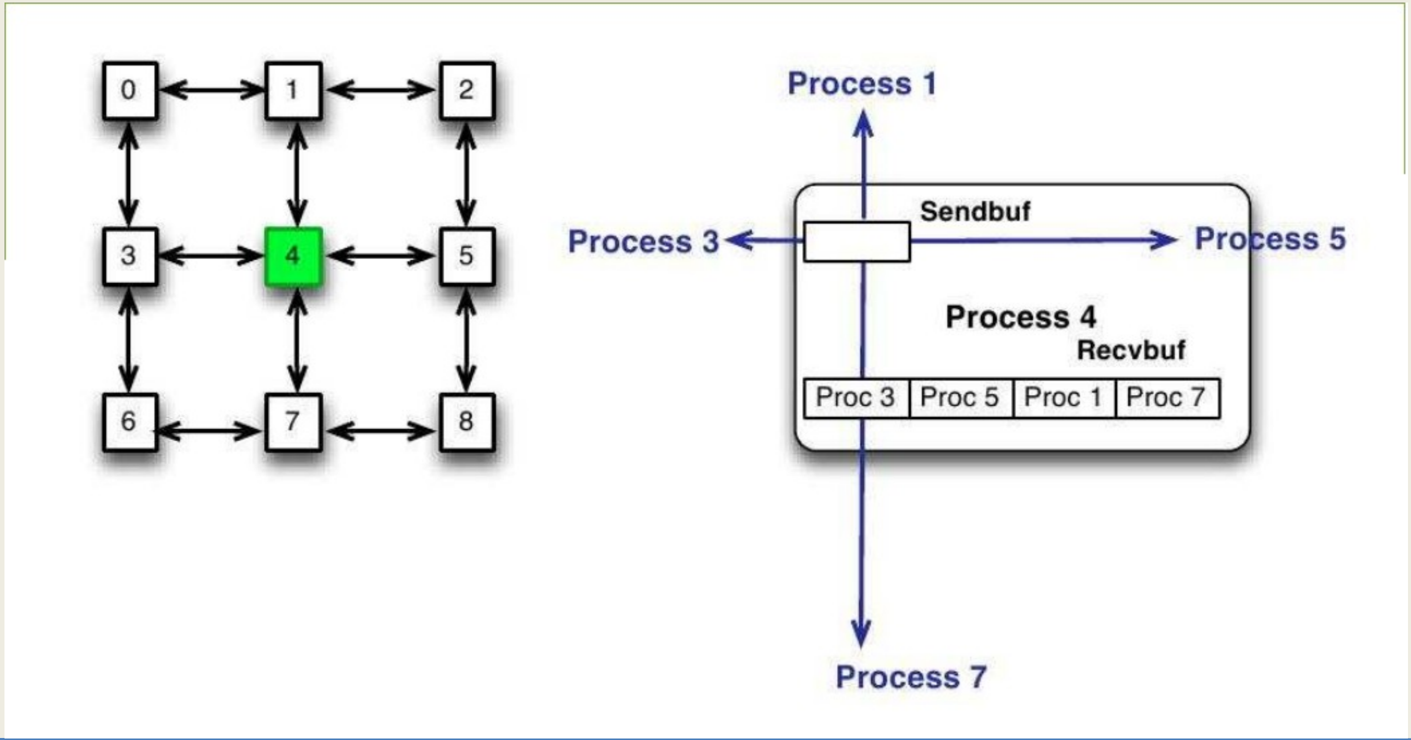
Summary

- Introduction
- Nonblocking collectives
- Neighborhood collectives
- RMA and one-sided communications

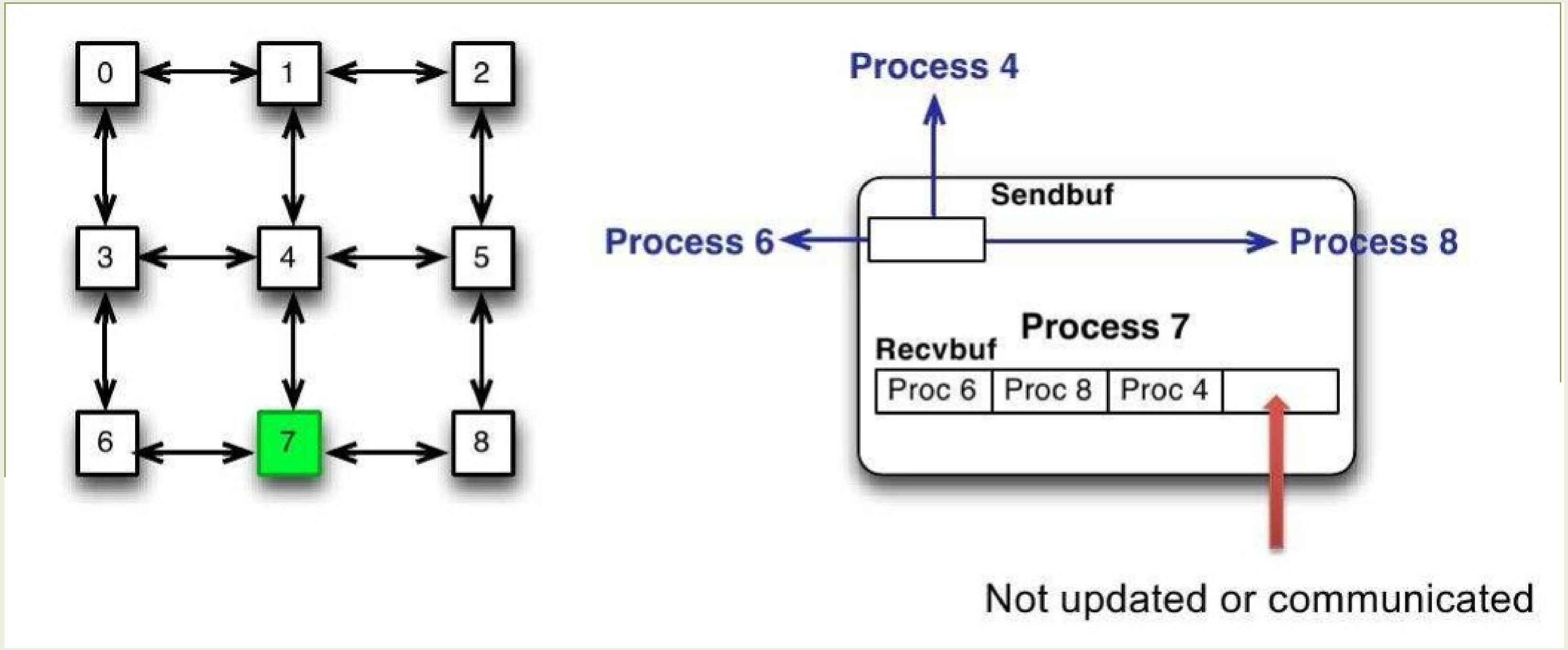
Neighborhood collective

- New functions (and their variable buffer and nonblocking variants) define collective operations among a process and its neighbors:
  - MPI_Neighbor_allgather
  - MPI_Neighbor_alltoall
- Neighbors are defined by an MPI cartesian or graph virtual topology that must be previously set
- These functions are useful, for example, in stencil computations that require nearest-neighbor exchanges

Example: MPI_Neighbor_allgather



Example: MPI_Neighbor_allgather

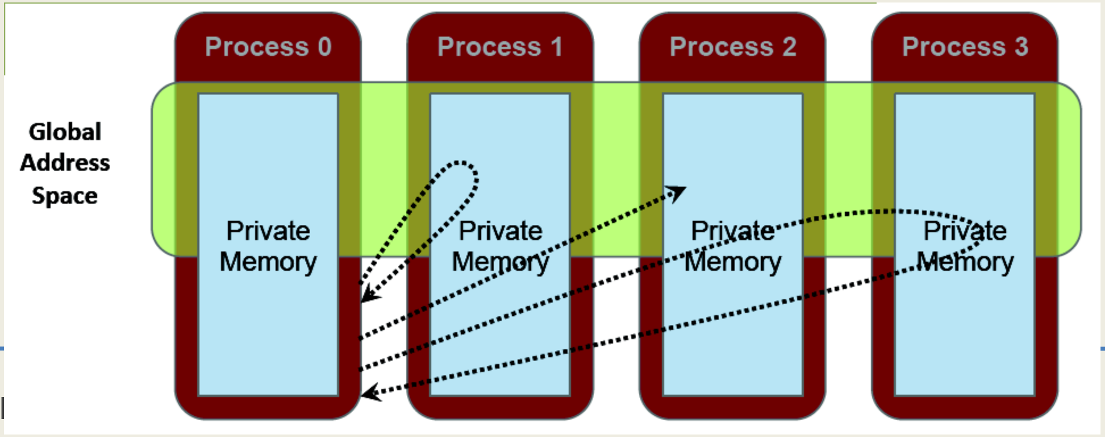


Summary

- Introduction
- Nonblocking collectives
- Neighborhood collectives
- RMA and one-sided communications

MPI-2 and one-sided communication

- One-sided communication was introduced in MPI-2 with the basic idea of decouple data movement with process synchronization
- It makes possible to move data without requiring that the remote process synchronize
- Each process exposes a part of its memory to the other processes
- Other processes can read or write in the exposed part of the memory (window)



Window creation

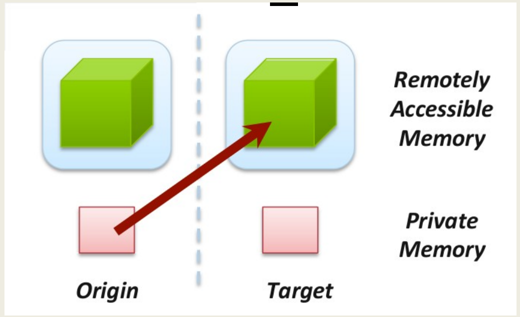
- MPI_WIN_ALLOCATE
  - You want to create a buffer and directly make it remotely accessible
- MPI_WIN_CREATE
  - You already have an allocated buffer that you would like to make remotely accessible
- MPI_WIN_CREATE_DYNAMIC
  - You don't have a buffer yet, but will have one in the future
  - You may want to dynamically add/remove buffers to/from the window
- MPI_WIN_ALLOCATE_SHARED
  - You may want multiple processes on the same node share a buffer

Example MPI_WIN_ALLOCATE

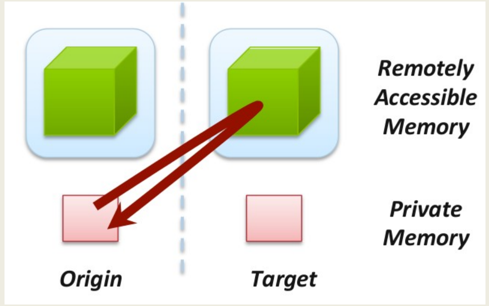
```
int main(int argc, char ** argv)
{
  int *a;
  MPI_Win win;
  MPI_Init(&argc, &argv);
/* collectively create remote accessible memory in a window */
  MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &a, &win);
/* Array ‘a’ is now accessible from all processes in MPI_COMM_WORLD */
  MPI_Win_free(&win);
  MPI_Finalize();
return 0;
}
```

Data movement

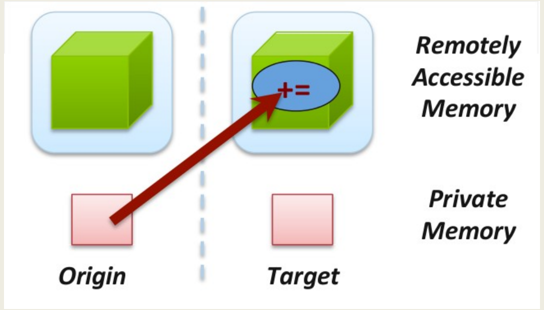
MPI_Put



MPI_Get

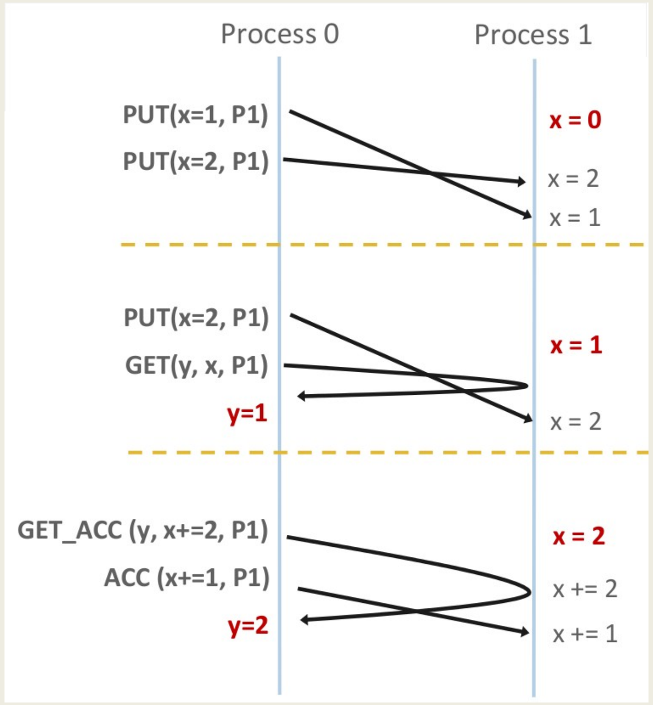


MPI_Accumulate



Plus many others... (Get_accumulate, Fetch_and_op, Compare_and_swap...)

Ordering



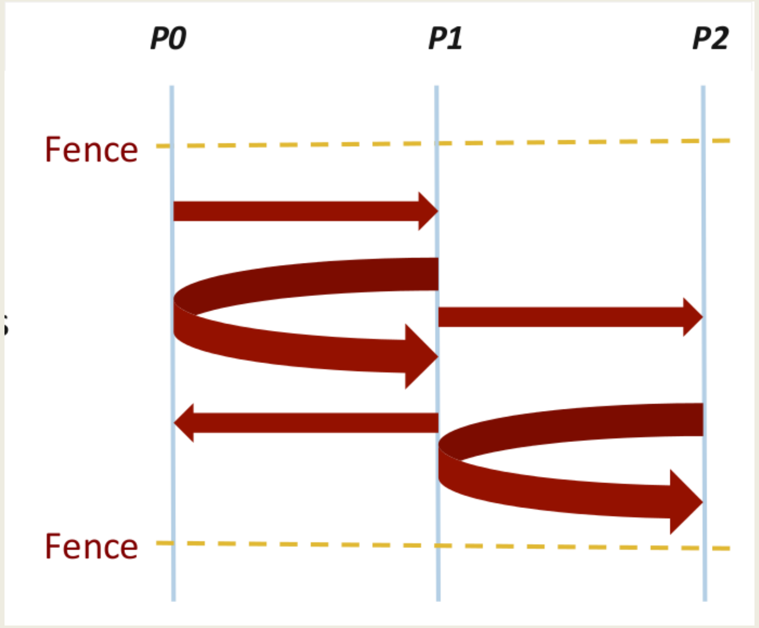
- No guaranteed ordering for put/get
- Results for concurrent put/accumulate are undefined
- For concurrent accumulate operations to the same location ordering is guaranteed

RMA synchronization models

- MPI provides three synchronization models:
  - Fence (active target)
  - Post-start-complete-wait (generalized active target)
  - Lock/Unlock (passive target)
- Data accesses occur within “epochs”
  - access epoch: contain a set of operation issued by an origin process
  - exposure epoch: enable remote processes to update a target window
  - Synchronization models provide mechanisms for establishing epochs

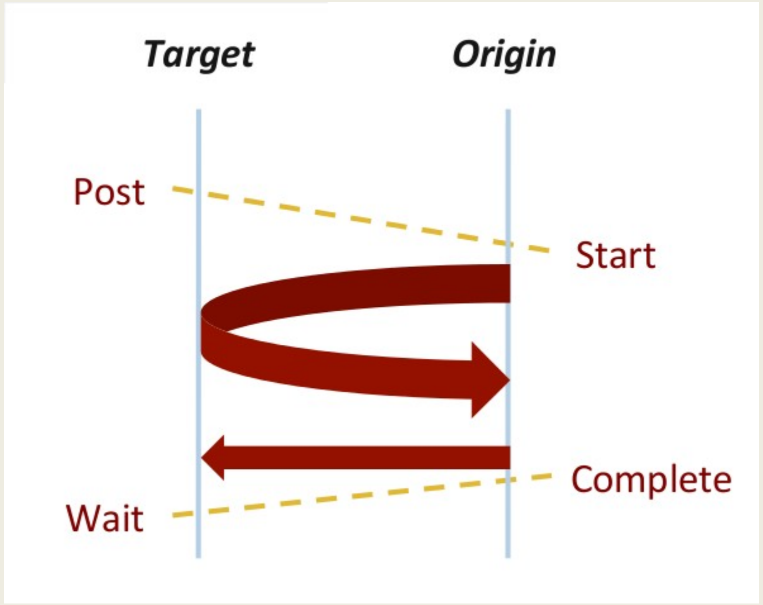
Fence: active target synchronization

- MPI_WIN_FENCE starts and ends access and exposure epoch of all processes in the window
- Collective synchronization model
- All operations complete at the second fence synchronization



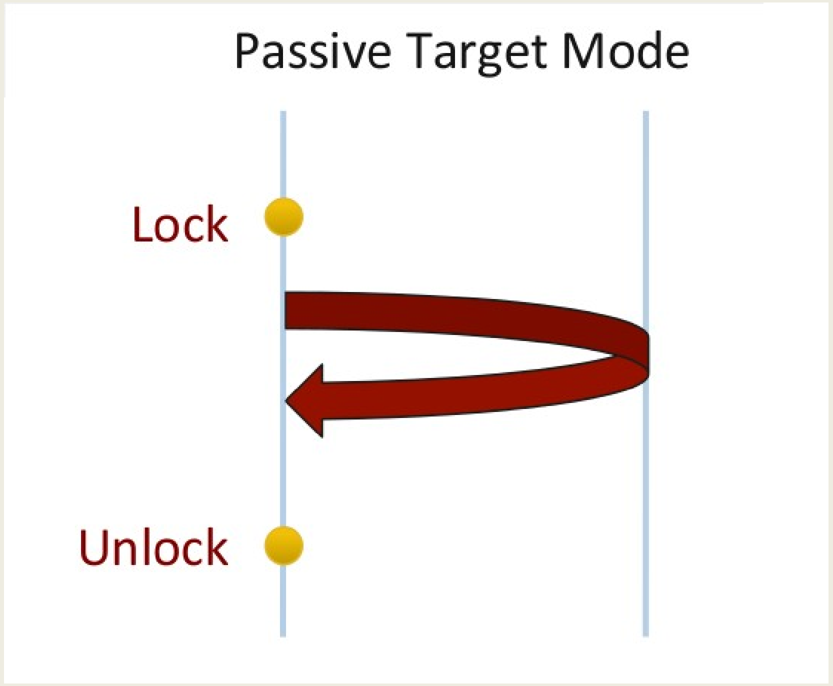
Post/start/complete/wait: generalized active target synchronization

- More flexible wrt MPI_WIN_FENCE
  - origin and target may specify who they communicate with, through the definition of an MPI_Group
- target side: exposure epoch
  - opened with MPI_WIN_Post
  - closed with MPI_WIN_Wait
- origin side: access epoch
  - opened with MPI_WIN_Start
  - closed with MPI_WIN_Complete



Lock/unlock: passive target synchronization

- One sided asynchronous communication
- Target does not participate in communication operation
- Shared memory-like model



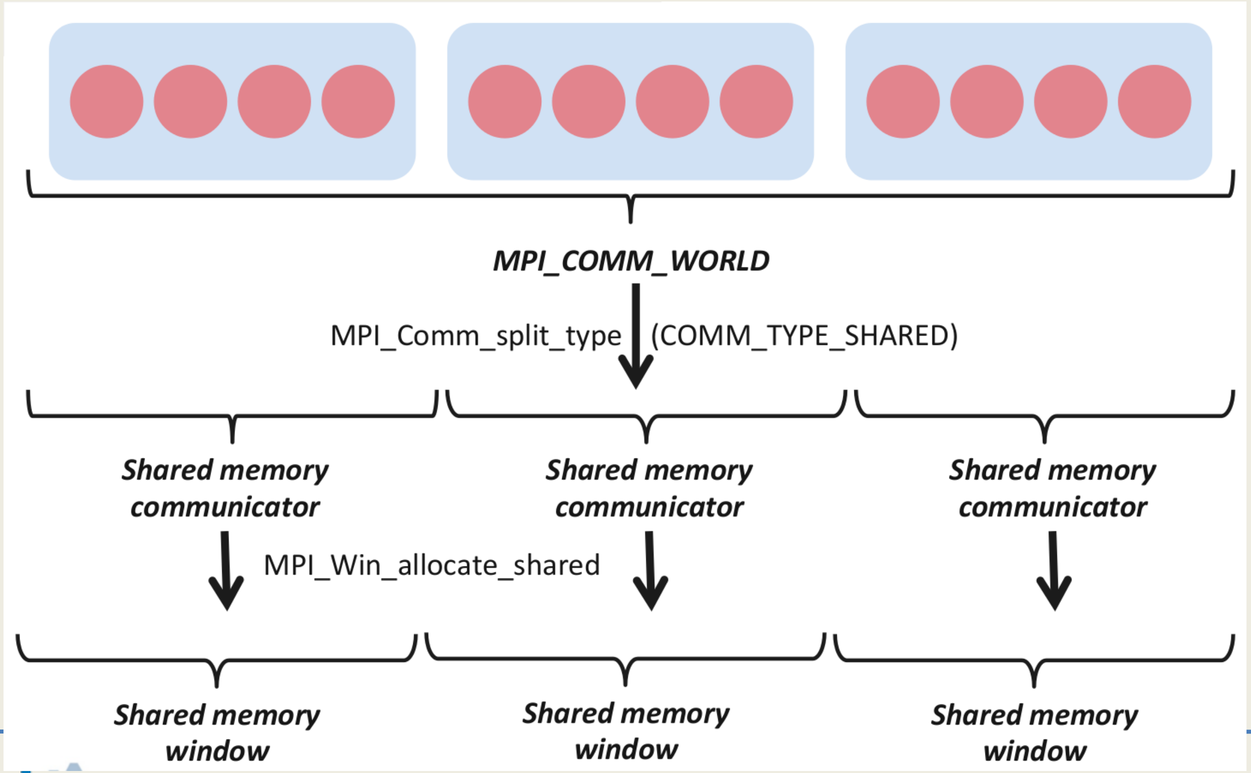
MPI RMA Memory model

- MPI-3 provides two memory models: separate and unified
- MPI-2 separate model
  - Logical public and private copies
  - MPI provides software coherency between window copies
  - extremely portable to systems that don't provide hardware coherence
- MPI-3 unified model (new!)
  - single copy for the window
  - system should provide a mechanism for coherency
  - it allows concurrent local/remote access

Shared memory with MPI

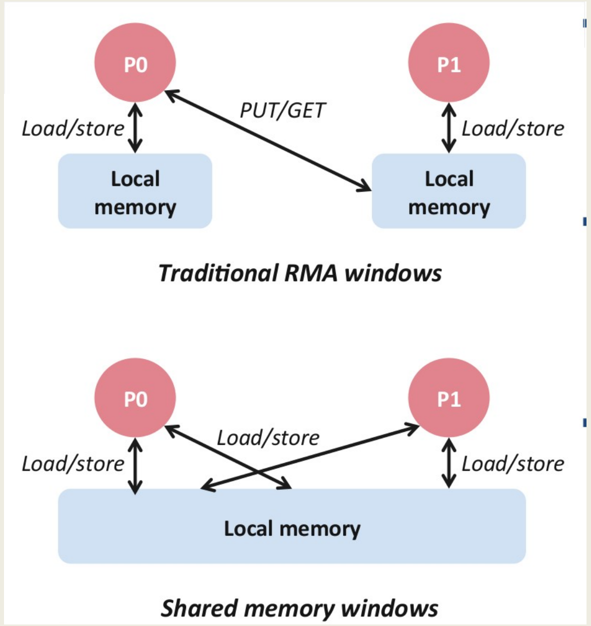
- MPI-3 permits to manage shared memory access to different processes
- It uses many of the concepts of one-sided communication
- Can be simpler to implement wrt OpenMP threads
- It can live together with other different MPI parallelization layers

MPI and shared memory



RMA windows vs shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - e.g. x[100]=10



Memory allocation and placement

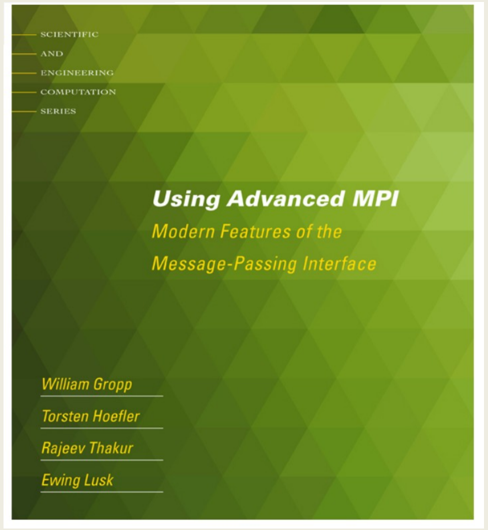
- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed (e.g. which physical memory it will be pinned to)
- The total allocated shared memory on a communicator is contiguous by default
  - Users can pass an info hint called “noncontig” that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement

MPI Shared memory example

```
int main(int argc, char ** argv)
{
  int buf[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
  MPI_Win_allocate_shared(comm, ..., &win);
  MPI_Win_lockall(win);

  /* copy data to local part of shared memory */

  MPI_Win_sync(win);

  /* use shared memory */

  MPI_Win_unlock_all(win);
  MPI_Win_free(&win);
  MPI_Finalize();
  return 0;
}
```

Resources and acknowledgements

- Online resources:
  - www.mcs.anl.gov/~thakur/sc15-mpi-tutorial
  - http://www.mpi-forum.org/docs/docs.html
  - http://www.mpi-forum.org/
  - http://software.intel.com/en-us/intel-mpi-library/
  - http://www.open-mpi.org/
- Books:
  - Gropp *et al.* Using advanced MPI - MIT Press, 2014



- Acknowledgements:
  - G.F. Marras (Cineca), H. Bockhorst (Intel) for revision and contribution to these slides

Backup slides

Intel NBC benchmarks (IMB)

- Results based on Intel MPI Library 5.0 Beta3 and Intel MPI Benchmarks 4.0 Beta (IMB), both of which support the MPI-3 standard.
- In a nutshell, the benchmark flow looks as follows:

  1. Measure the time needed for a pure communication call (e.g., MPI_Ibcast() followed by MPI_Wait())

  2. Start communication (e.g., call MPI_Ibcast())

  3. Start computation with duration equal to the time measured in step 1. Thus we ensure that communication and computation parts consume approximately the same amount of time.

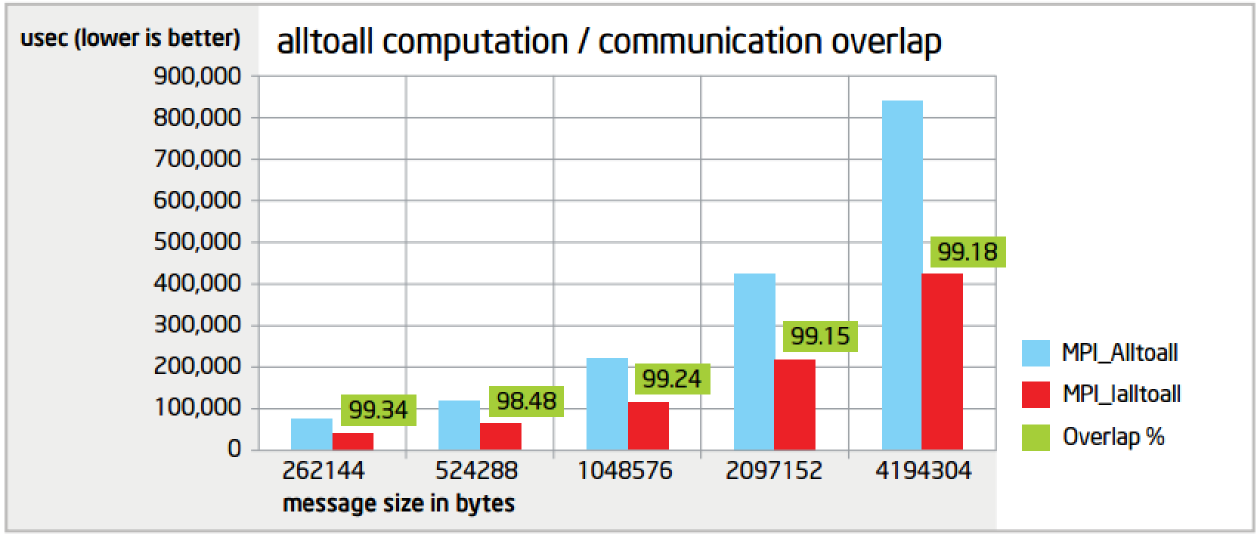
  4. Wait for communication to finish (i.e., call MPI_Wait())

Intel NBC benchmarks (IMB)

- Given the description above, the IMB-NBC benchmarks output four timings:
  - time_pure is the time for nonblocking operation, which is executed without any concurrent CPU activity;
  - time_CPU is the time of the test CPU activity (which is supposed to be close to the time_pure value);
  - time_ovrlp is the time for nonblocking operations concurrent with CPU activity;
  - overlap – the estimated overlap percentage obtained by the following formula:

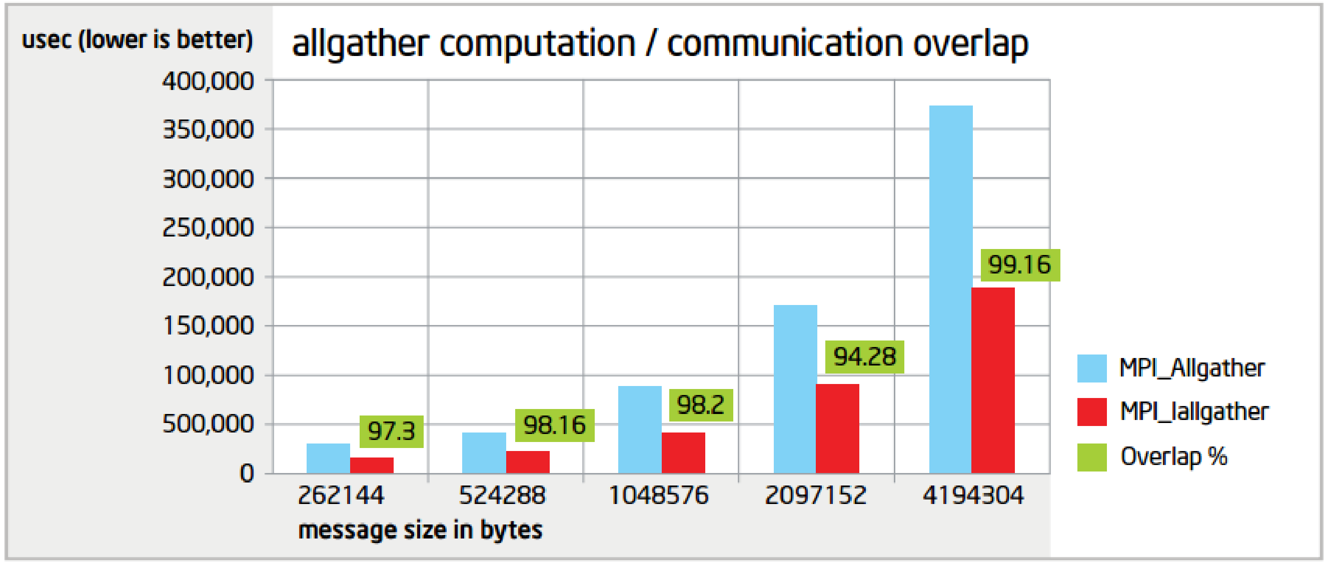
overlap = 100 * max(0, min(1, (time_pure + time_CPU - time_ovrlp) / max(time_pure, time_CPU)))

Results



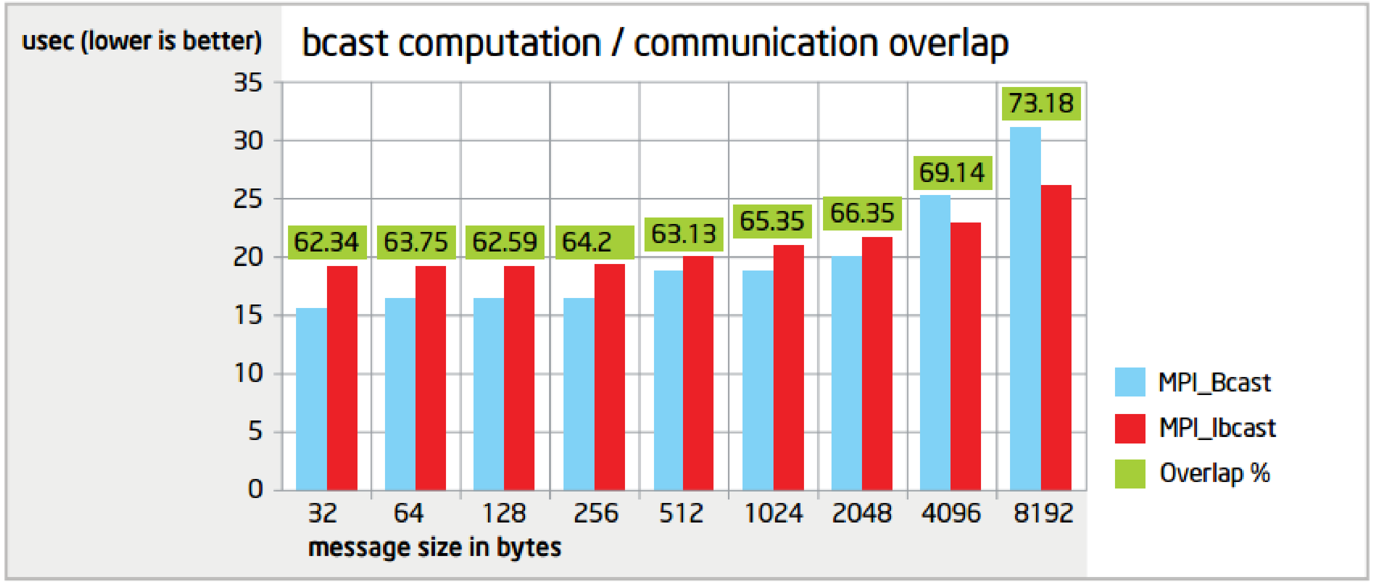
1 MPI_Ialltoall overlapped with computation vs. MPI_Alltoall followed by computation.

Results



2 MPI_Iallgather overlapped with computation vs. MPI_Allgather followed by computation.

Results



3 MPI_Ibcast overlapped with computation vs. MPI_Bcast followed by computation.