# Myths and reality of communication/computation overlap in MPI applications

**Alessandro Fanfarillo**
National Center for Atmospheric Research
Boulder, Colorado, USA

elfanfa@ucar.edu

Oct 12th, 2017

# Disclaimer

- This presentation is not meant to provide official performance results.
- This presentation is not meant to judge/test/blame any compiler, MPI implementation, network interconnect, MPI Forum decision, or NCAR staff.
- This presentation is only meant to clarify some of the misconceptions behind the communication/computation overlap mechanism in MPI applications.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of NCAR, UCAR and NSF.

# Short Historical Introduction to MPI

Early 80s: Message Passing model was dominant. Every machine/network producer had its own message passing protocol.

1992: A group of researchers from academia and industry, decided to design a standardized and portable message-passing system called MPI, Message Passing Interface.

The MPI specification provides over 200 API functions with a specific and defined behavior to which an implementation must conform.

There are many cases where standard is (intentionally ?) ambiguous and the MPI implementer has to make a decision.

The *asynchronous message progress* is one of them.

# Common Myths on Communication/Computation Overlap

My application overlaps communication with computation because...

- ...it uses non-blocking MPI routines
- ...it uses one-sided (Remote Memory Access) MPI routines
- ...it runs on a network with native support for Remote Direct Memory Access (e.g. Mellanox Infiniband)

MPI courses usually explain "how to get correct results and hoping to get communication/computation overlap".

We need a better definition of what overlap actually means...

# Overlap and MPI Progress

**Overlap** is a characteristic of the **network layer**; it consists of the NIC capability to take care of data transfer(s) without direct involvement of the host processor, thus allowing the CPU to be dedicated to computation.

**Progress** is a characteristic related to MPI, which is the software stack that resides above the network layer.

Remote Direct Memory Access can provide overlap.

The MPI standard defines a Progress Rule for asynchronous communication operations.

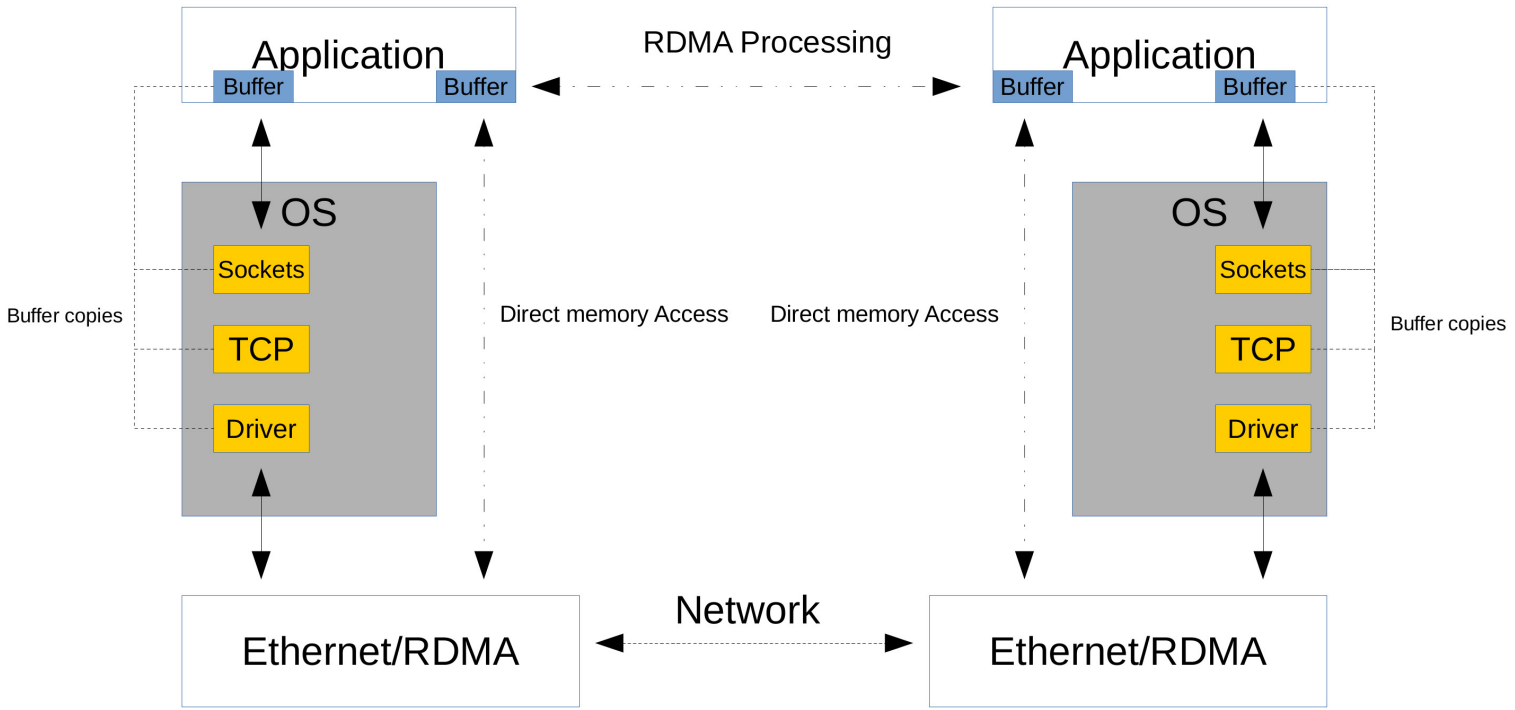
Two different interpretations leading to different behaviors (both compliant with the standard).

# Remote Direct Memory Access (1)

RDMA is a way (protocol) of moving buffers between two applications across a network.

RDMA is composed by three main components:

1. Zero-copy (send/recv data without copies)
2. Kernel/OS bypass (direct access hw network without OS intervention)
3. Protocol offload (RDMA and transport layer implemented in hw) !!!



# Remote Direct Memory Access (2)

In theory the CPU is not involved in the data transfer and the cache in the remote CPU won't be polluted.

In practice, vendors decide how much protocol processing is implemented in the NIC.

QLogic (Intel True Scale) encourages protocol onloading.

Mellanox Infiniband encourages protocol offloading.

Both Yellowstone and Cheyenne have Mellanox InfiniBand.

# MPI Progress Rule

- Strictest interpretation: it mandates non-local progress semantics for all non-blocking communication operations once they have been enabled.

- Weak interpretation: it allows a compliant implementation to require the application to make further library calls in order to achieve progress on other pending communication operations.

It is possible either to support overlap without supporting independent MPI progress or have independent MPI progress without overlap.

Some networks (e.g. Portals, Quadrics) provide MPI-like interfaces in order to support MPI functions in hardware, others do not.

MPI libraries should implement in software what is needed and not provided by the hardware.

# Strategies for Asynchronous Progress

Three possible strategies for asynchronous progress:

1. Manual progress
2. Thread-based progress (polling-based vs. interrupt-based)
3. Communication offload

Manual progress is the most portable but requires the user to call MPI functions (e.g. `MPI_Test`).

Using a dedicated thread to poke MPI requires a thread-safe implementation.

Offloading may become a bottleneck because of the low performance of embedded processor on NIC.

# Thread-based progress: the silver bullet ?

Often stated as the silver bullet but not widely used.

The MPI implementation must be thread-safe (MPI_THREAD_MULTIPLE).

Thread-safety in MPI implementations penalizes performance (mostly latency).

Two alternatives: polling-based and interrupt-based.

Polling-based is beneficial if separate computation cores are available for the progression threads.

Interrupt-based might also be helpful in case of oversubscribed node.

For more info: T. Hoefler and A. Lumsdaine - *Message Progression in Parallel Computing - To Thread or not to Thread?*

# MPI Non-blocking Example

```
  call MPI_COMM_RANK(MPI_COMM_WORLD,me,ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,np,ierr)

  if(me == np-1) then
  call MPI_Irecv(y, n, MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, req, ierr)
  endif

  call MPI_Barrier(MPI_COMM_WORLD,ierr)

  if(me == 0) then
  call MPI_Isend(x, n, MPI_DOUBLE, np-1, 0,
                 MPI_COMM_WORLD, req, ierr)
   ! Some long computation here
   call MPI_Wait(req,status,ierr)
else if(me == np-1) then
   call MPI_Wait(req,status,ierr)
   ! Data on y correctly received
endif
```

# Modules and Flags for IntelMPI

In the demo, two MPI implementations have been used: OpenMPI/3.0.0 and IntelMPI/2017.1.134.

For compiling and running the benchmarks with IntelMPI:

module swap gnu intel

module load intel/17.0.1

module load impi

mpiicc your_benchmark.c -o your_benchmark_intel

To turn on the helper thread for asynchronous progress in IntelMPI:

export I_MPI_ASYNC_PROGRESS=1

# Modules and Flags for Open-MPI

For compiling and running the benchmarks with Open-MPI on Cheyenne:

module load gnu/7.1.0

module load openmpi/3.0.0

mpif90 your_benchmark.c -o your_benchmark_openmpi

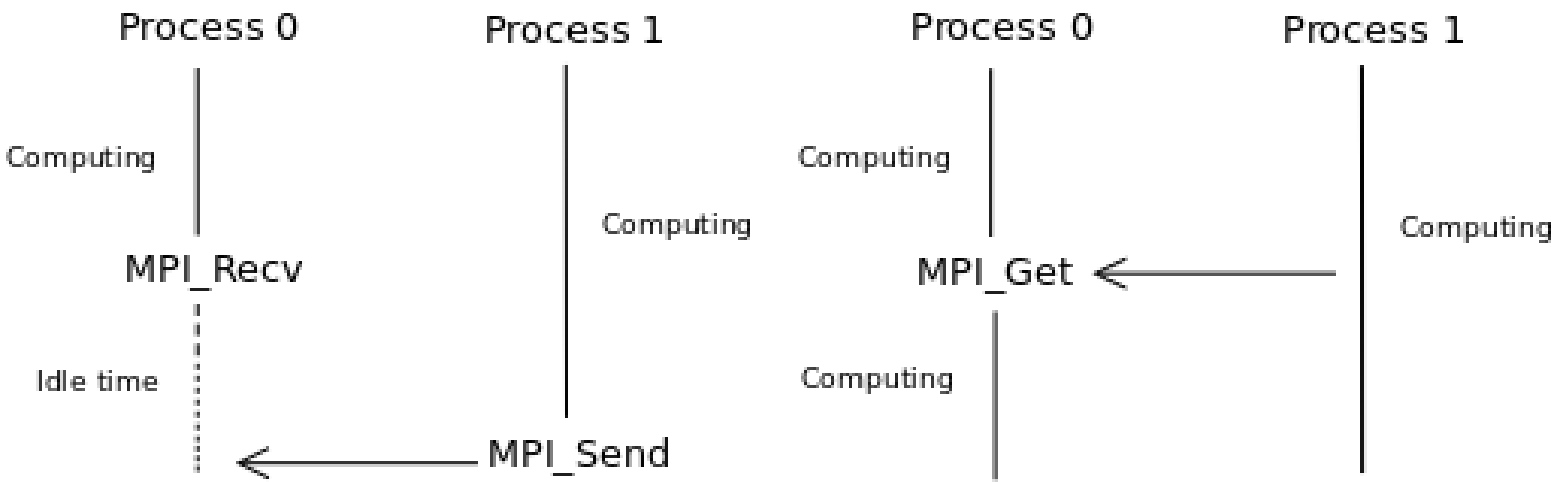
mpirun -np n ./your_benchmark_openmpi

To turn off the RDMA support use btl_openib_flags 1:

mpirun -np n –mca btl_openib_flags 1 ./your_benchmark_openmpi

# Demo 1
# Non-Blocking Two-Sided Progress

# Two-sided vs. One-sided Communication




Two-sided One-sided

In theory, one-sided communication allows to overlap communication with computation and save idle times.

# Demo 2
# One-Sided Get Progress

# MPI One-sided and RDMA

MPI one-sided routines show great potential for applications that benefit from communication/computation overlap.

MPI one-sided maps naturally on RDMA.

In case of not asynchronous progress, MPI calls needed on the target process (poking MPI engine).

Some MPI implementations have implemented the one-sided routines on top of the two-sided...

# Demo 3
# One-Sided Put Progress and Memory Allocation

# Conclusions

Message progression is an intricate topic.

Do not assume, always verify.

Asynchronous progress may penalize the performance.

Writing programs exploiting overlap is critical.

Consider to switch from non-blocking two-sided to one-sided.

# Acknowledgments


- Davide Del Vento
- Patrick Nichols
- Brian Vanderwende
- Rory Kelly

# Thanks