MPI Internals

CSC5001 – High Performance systems

TELECOM SudParis
INSTITUT POLYTECHNIQUE DE PARIS
Institut Mines-Télécom

Objectives

- Understand how an MPI implementation works internally
- Understand the impact of these internals on applications

Introduction

- MPI: Message Passing Interface
  - Defines an API (C, Fortran)
  - Several libraries implement this API
  - MPI programs are portable from one implementation to another
- API defined by the MPI forum
  - academia (Univ. Tennessee, ORNL, ANL, Riken, INRIA, ...)
  - industrials (IBM, Intel, Fujitsu, NEC, Mellanox, ...)

Institut Mines-Télécom

History

- 1994: MPI 1
  - Inspired by PVM
- 1997 : MPI 2
  - One-sided communications
  - Dynamic creation of MPI processes
  - MPI-IO
- 2012: MPI 3
  - Non-blocking collective communication
  - Fault tolerance
  - Improved one-sided communications
- 2020: MPI 4
  - Persistent collective communications
  - Improved error handling

MPI implementations

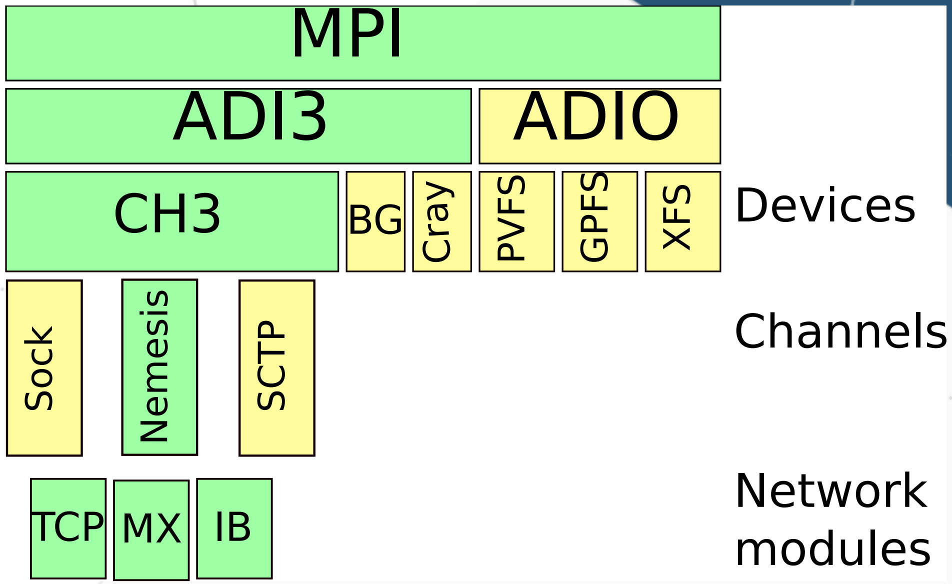
- Generic implementations
  - MPICH
  - OpenMPI
- Platform-specific implementations
  - Usually derived from a generic implementation
  - MVAPICH, Intel MPI, HP MPI, Bull MPI, IBM MPI

MPICH

- Developped at Argonne National Laboratory
- Base of several other implementations
  - Intel MPI
  - MVAPICH
  - IBM MPI

MPICH3 architecture

- ADIO: high level interface for disk IO (MPI-IO)
- ADI3: high level interface for communications
- Devices: implement the ADI3 interface,
  - Drivers for networks with a similar interface to MPI (Blue Gene, Cray, Myrinet)
  - Modules that implement some ADI3 features (collective communication, etc.)
- Channels: implement point-to-point communications
  - Drivers for inter-node communication(ex : Sock)
  - Some modules implement intra-node communication (ex : Nemesis)

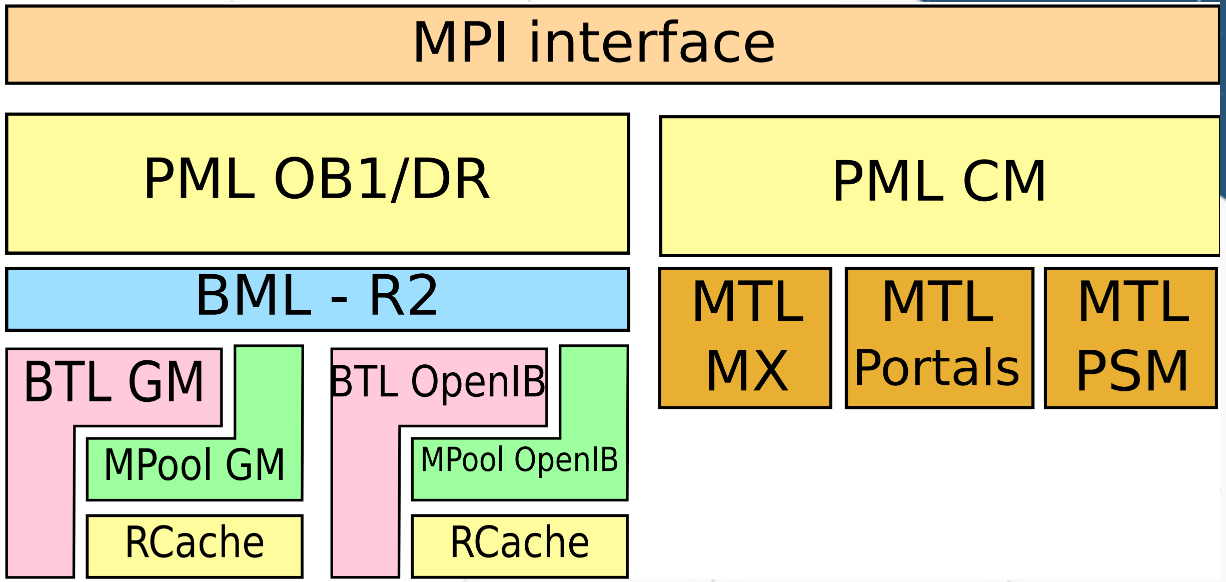


OpenMPI

- Developped by several academia (Indiana Univ., Univ. Tennessee, …)
- Rely on several older implementations
  - FT-MPI (University of Tennessee)
  - LA-MPI (Los Alamos National Lab)
  - LAM-MPI (Indiana University)
  - MVAPICH (Ohio State University)
- Architecture based on software components

OpenMPI architecture

- PML : Point-to-point Messaging Layer
- BML : BTL Management Layer
- BTL : Byte Transfer Layer
- MPool : Memory Pool
- RCache : Registration Cache
- MTL : Message Transfer Layer



TL;DR

- For networks whose API is similar to MPI
  - Myrinet MX, Blue Gene, Quadrics, ...
  - Direct translation of MPI calls
- For other networks
  - TCP, Infiniband, ...
  - A message may be processed before being transmitted
    - Collective communication –> P2P communication
    - Message matching
    - Other processing ? (aggregation, multi-rail, ...)

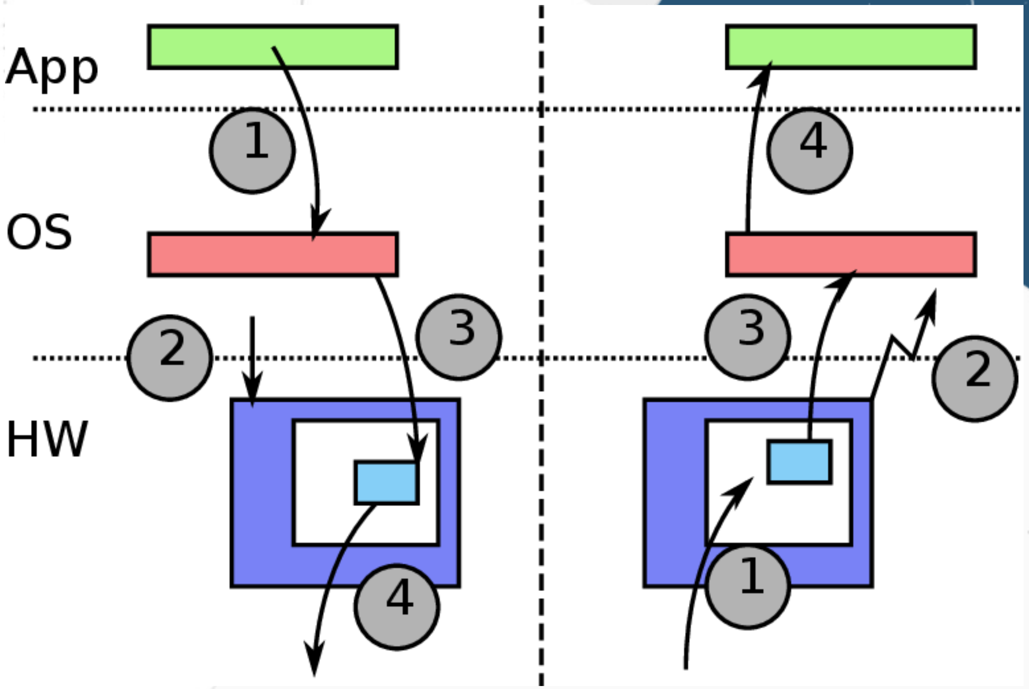
MPI transfer methods

- Exercise:
  - Analyze and run bibw.c
  - The program only works for small messages
  - Is it always the same message size that fails ?
    - Is it the same for intra-node and inter-node ?

Sending/Receiving a message

- On a classic network (eg. TCP/Ethernet)
  - Sending
    - 1 copy the message to a buffer (+adding headers)
    - 2 ask the NIC to send
    - 3-4 copie the message to the NIC memory, and the NIC sends the data to the network
  - Receiving
    - 2 NIC raises an interrupt
    - 3 OS copies the message to an internal buffer
    - 4 application polls the OS
    - 5 OS copies the message to the application buffer

  –> 2 copies per transfer



Institut Mines-Télécom

Sending/Receiving a message: zero copy

On a high performance network (eg. InfiniBand)

- Sending

    1 ask to NIC to send data

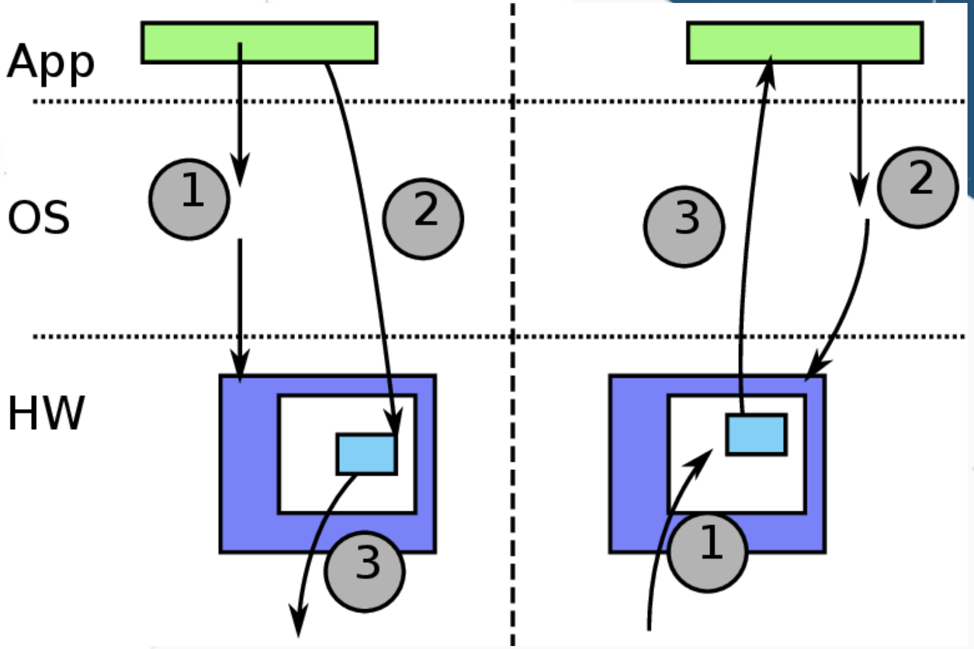
    2-3 NIC copies the data to the NIC memory, and send it to the network

- Receiving

    2 application polls the NIC

    3 upon reception, the NIC copies the message to the application buffer

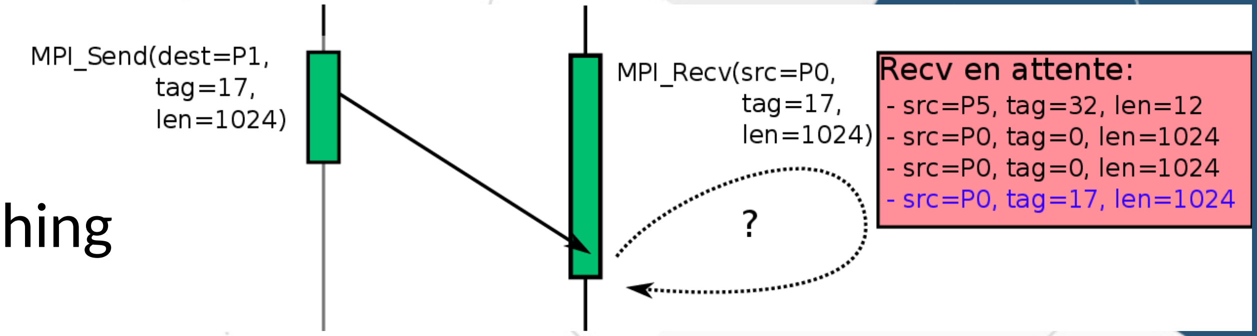
–> only one message copy



Eager mode

If MPI_Recv happens before sending

- List of pending MPI_*recv
- When a message arrives, search for a matching receive in the reception list
  - Takes into account the src, and tag
- If found, copy the message in the application buffer



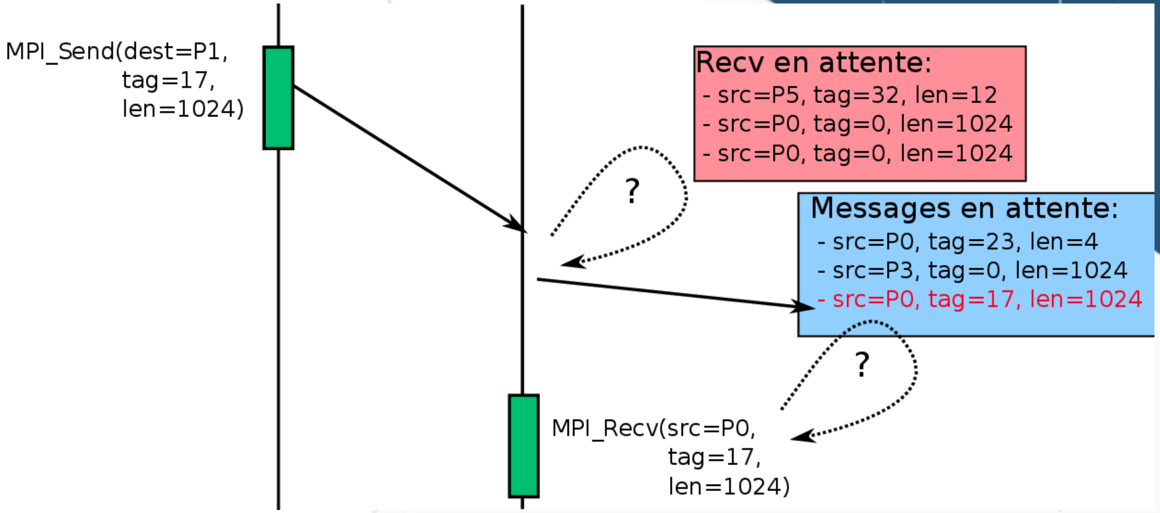
Unexpected messages

In case there's no matching MPI_Recv

- A message arrive
- Search for a matching recv
  - Not found
- Add the message to a list of "unexpected" messages
- When MPI_Recv is called:
  - Search for a matching unexpected message
  - Copy the message in the application buffer

→ spurious message copy

→ need to store unexpected messages



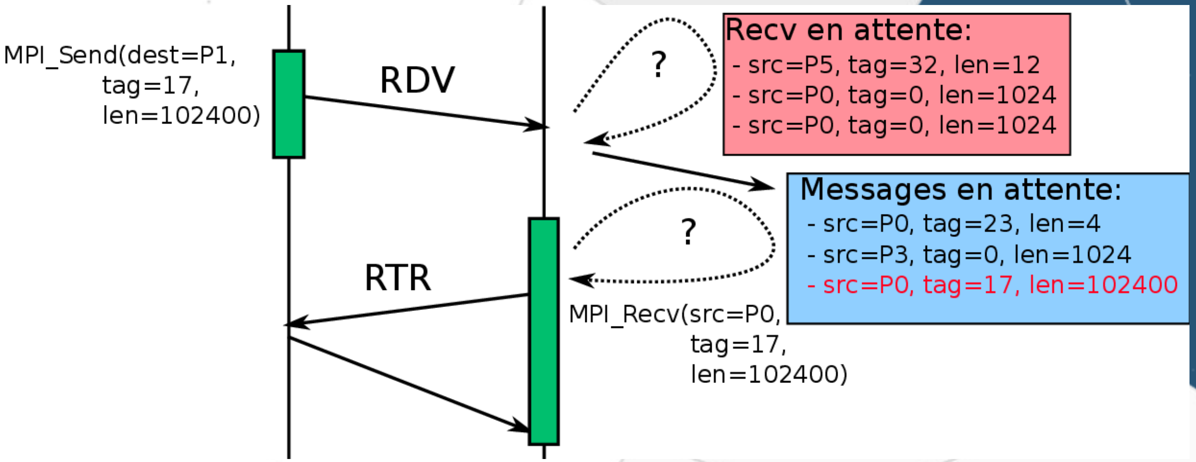
Rendez-vous mode

Send a Rendez-Vous request

- When the matching MPI_Recv happens, reply "Ready To Receive"
- Send the message
- Receive the message in the application buffer

→ No spurious message copy

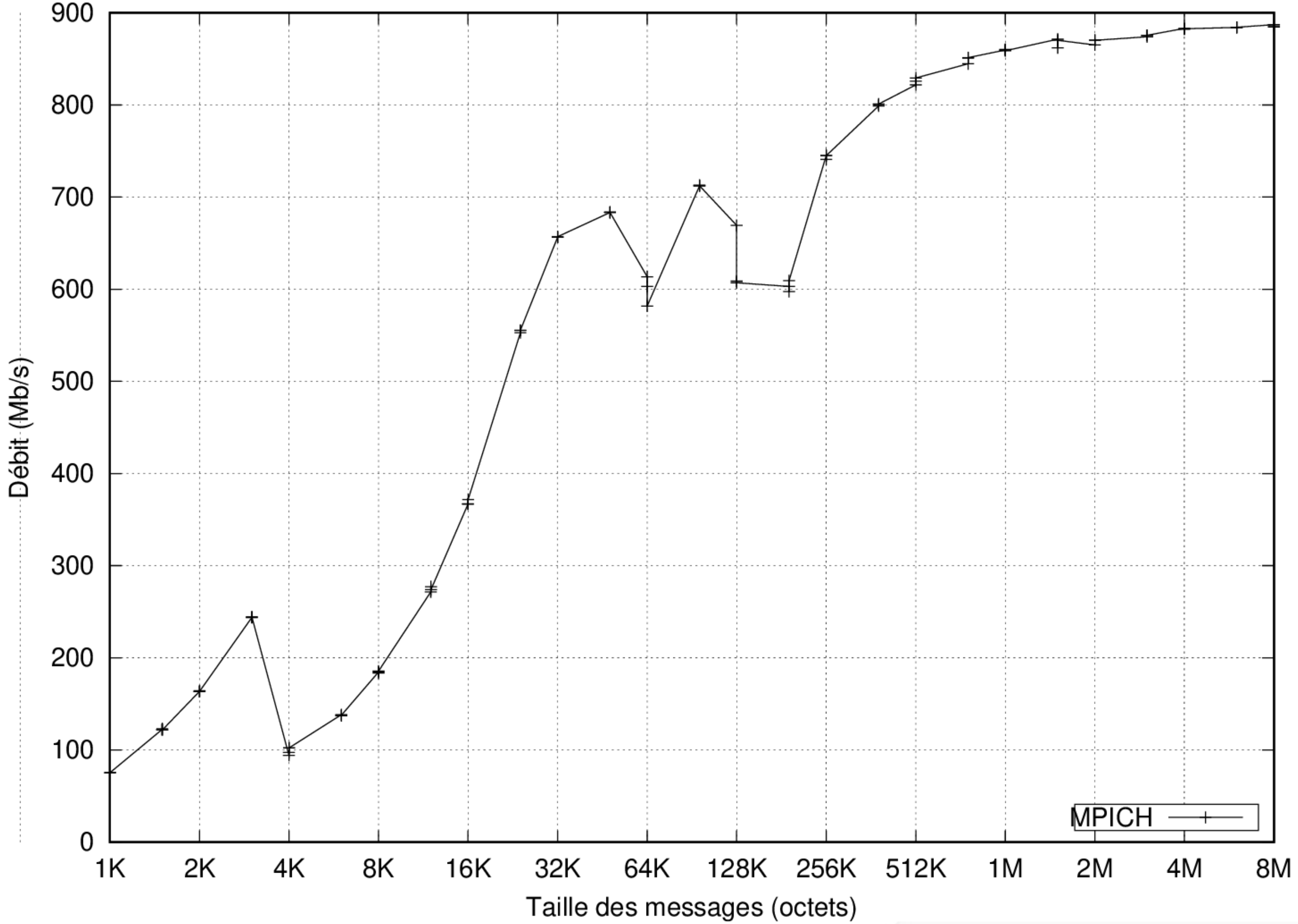
→ Additional messages, synchronization between nodes



Data transfer in MPI

- For small messages: use the eager mode
  - Directly send the date along with a header (src=X, tag=Y, len=Z)
  - If no MPI_Recv matches, store the message in the unexpected messages list
- For large messages: Rendez-Vous mode
  - Send a Rendez-Vous request that contains the header (src=X, tag=Y, len=Z)
  - When the receiver is ready, it replies Ready To Receive
  - Send the message
  - Recevie the message in the application buffer
- The Rendez-Vous prevents from
  - Storing large messages in the unexpected list
  - Copying a large message from one buffer to another

Eager vs Rendez-vous

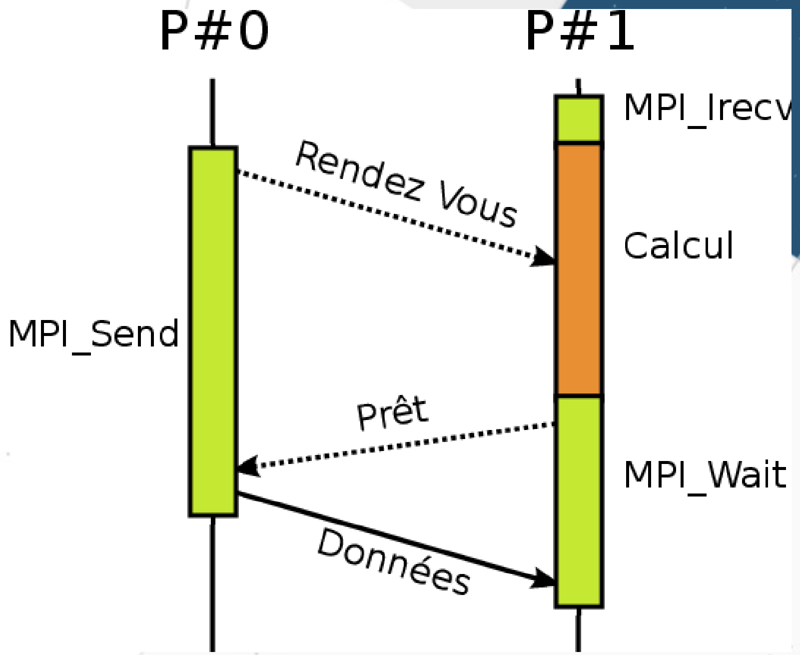


Exercise

- Analyze the `pingpong` program
- Run the program with 2 MPI ranks
- Explain the behavior of the program for large messages

Progression of communications

- Problem: MPI needs to poll the network to answer Rendez-Vous requests
- Possible solutions:
  - Call MPI_Test while computing
  - Add a thread dedicated to communication
  - Use another protocol that does not rely on rendez-vous



Progression of communications: exercise

- Analyze and run stencil_mpi.c with 2 MPI ranks.
  - Vary the problem size N
- With some values of N, the program stalls
- Fix the problem !

Thread-safety

- The MPI standard defines several thread-safety levels :
  - `MPI_THREAD_SINGLE`: only one thread runs
  - `MPI_THREAD_FUNNELED`: the program may have threads, but only the main thread calls MPI
  - `MPI_THREAD_SERIALIZED`: the program may have multiple threads that call MPI, but once at a time
  - `MPI_THREAD_MULTIPLE`: the program may call MPI concurrently from multiple threads
- Instead of initializing MPI with MPI_Init, we use:

```
int MPI_Init_thread( int *argc, char ***argv, int required, int *provided)
```

Warning : `provided` contains the thread-safety level chosen by MPI (may be different than required)

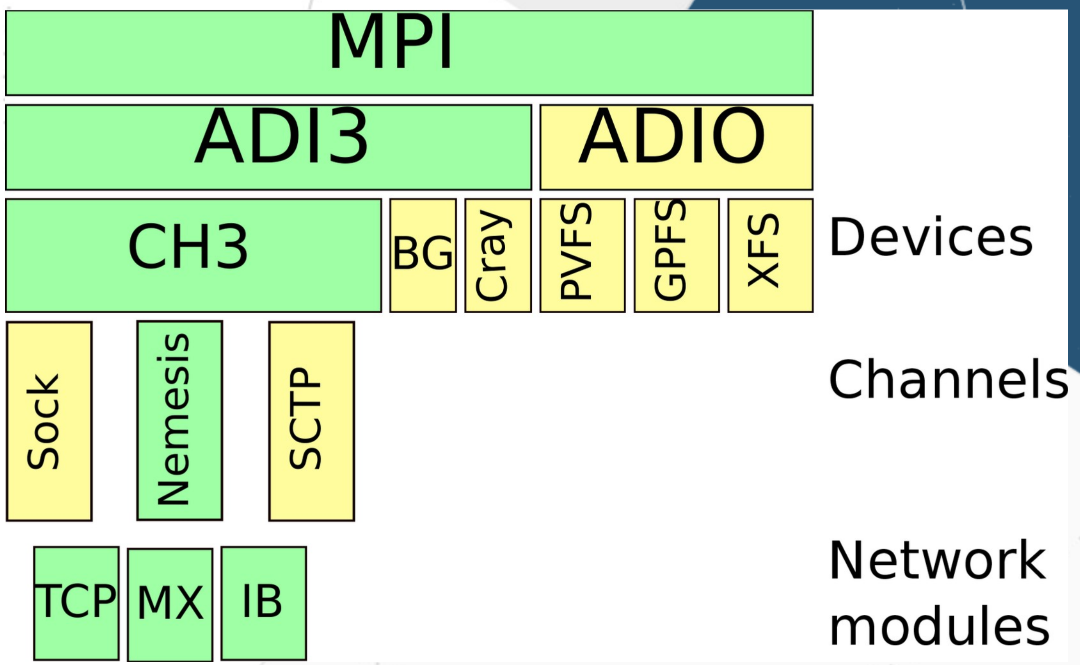
Consequences for an MPI implementation

- For an MPI implentation, the thread safety levels mean:
  - `MPI_THREAD_FUNNELED`: MPI cannot use a non thread-safe library
  - `MPI_THREAD_SERIALIZED`: code must be reentrant, no thread-specific variable
  - `MPI_THREAD_MULTIPLE`: data structured must be protected from concurrent access

Support for MPI_THREAD_MULTIPLE

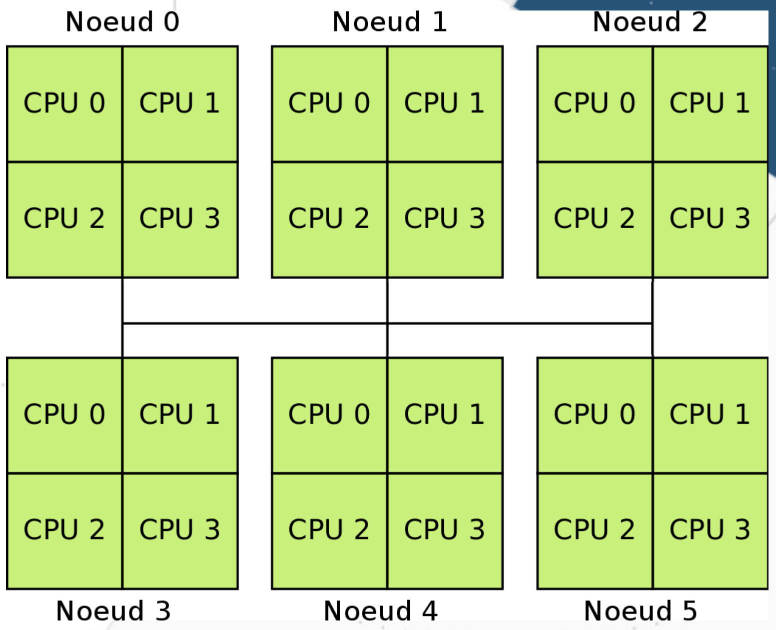
- Need to protect data structure from concurrent access
- Without degrading performance
  → Problems
  - Lots of modules to protect
  - Interactions between modules

  → Quite often, only a part of MPI is thread-safe



Hybrid programming models

- Typical cluster of compute nodes
  - N machines connected to a network
  - Each machine has M cores
- How to exploit this cluster ?



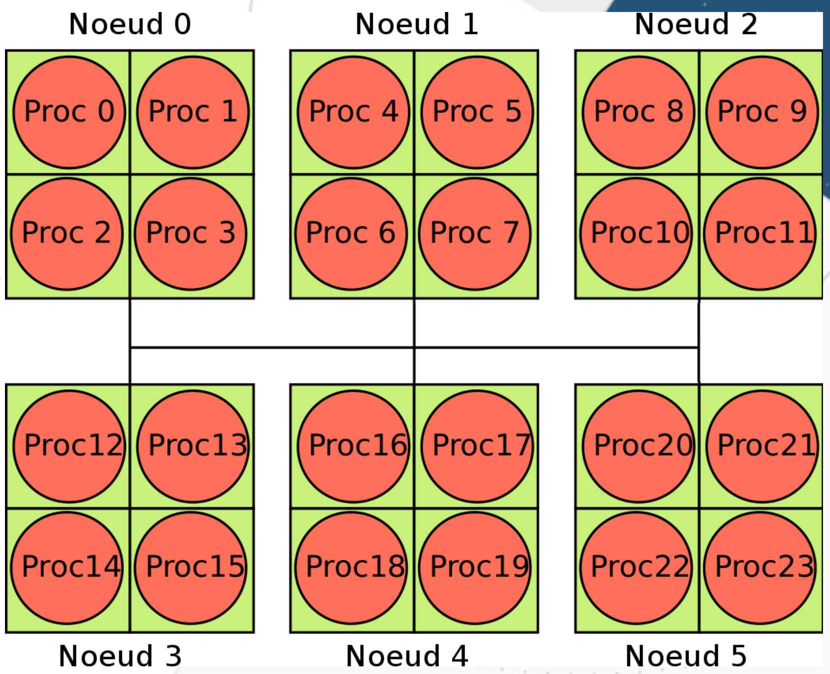
Using MPI only

- How to exploit this cluster ?
- 1 MPI rank per core

  + MPI handles the inter/intra node communication

  – no shared memory between processes on a node

  – locality of MPI ranks is hard to exploit



Mixing MPI with threads

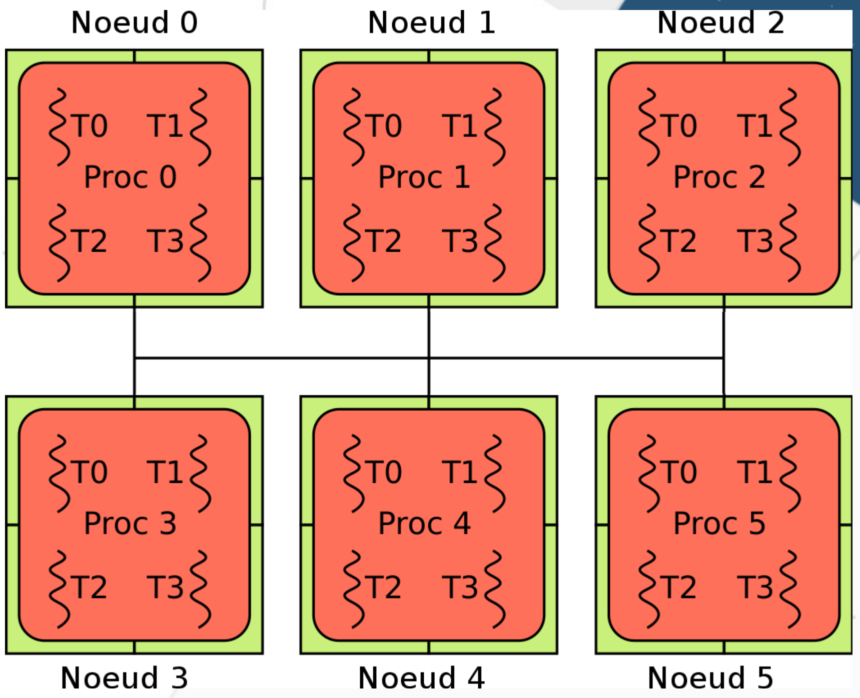
- How to exploit this cluster ?
- 1 MPI rank per node
- 1 thread per core

  + shared memory within a node

  + load balancing is easier

  + fewer MPI ranks (→ better scalability for collective communication)

  – hard to debug

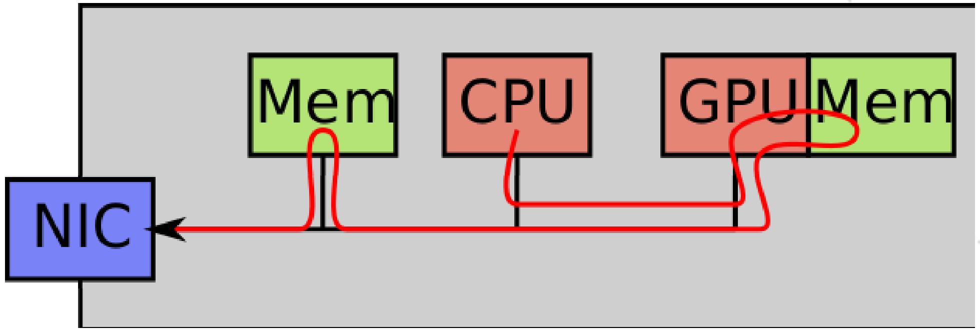


MPI + OpenMP

- 1 MPI rank per node
- Within a node : parallelization with OpenMP
- What level of thread-safety ?
  - MPI_THREAD_FUNNELED: MPI calls outside of parallel regions
  - MPI_THREAD_SERIALIZED: MPI call in critical sections
  - MPI_THREAD_MULTIPLE: no restriction

MPI + CUDA

- 1 MPI rank per GPU
- Some computations are offloaded to the GPU
- How to transfer data on the network from one GPU to another ?

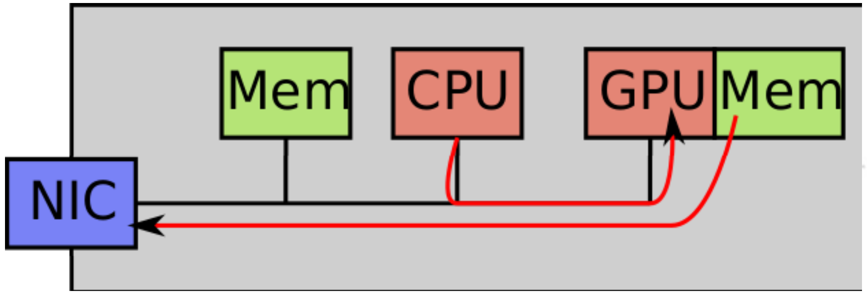


→ use a lot of CPU (cudaMemCopy, then MPI_Send)

→ multiple memory copies

MPI + CUDA: GPUDirect

- Available with some network technologies (eg. InfiniBand)
- DMA-based copies



→ low usage of the CPU

→ no spurious memory copy

MPI + OpenMP: exercise

- Parallelize the program stencil_mpi.c with OpenMP
- Initialize MPI properly
- A program that runs fine may still be bugous
  - Watchout for race condition

MPI + CUDA: exercise

- Parallelize the program stencil_cuda.c with MPI