

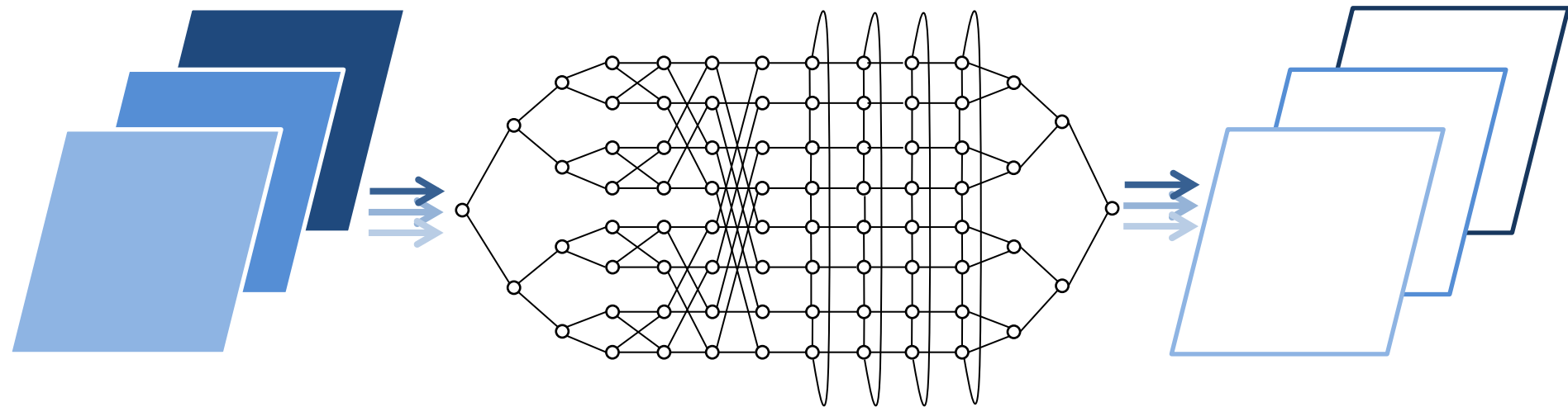
# Programmation Parallèle

**MPI:** Message Passing Interface

---

---

---



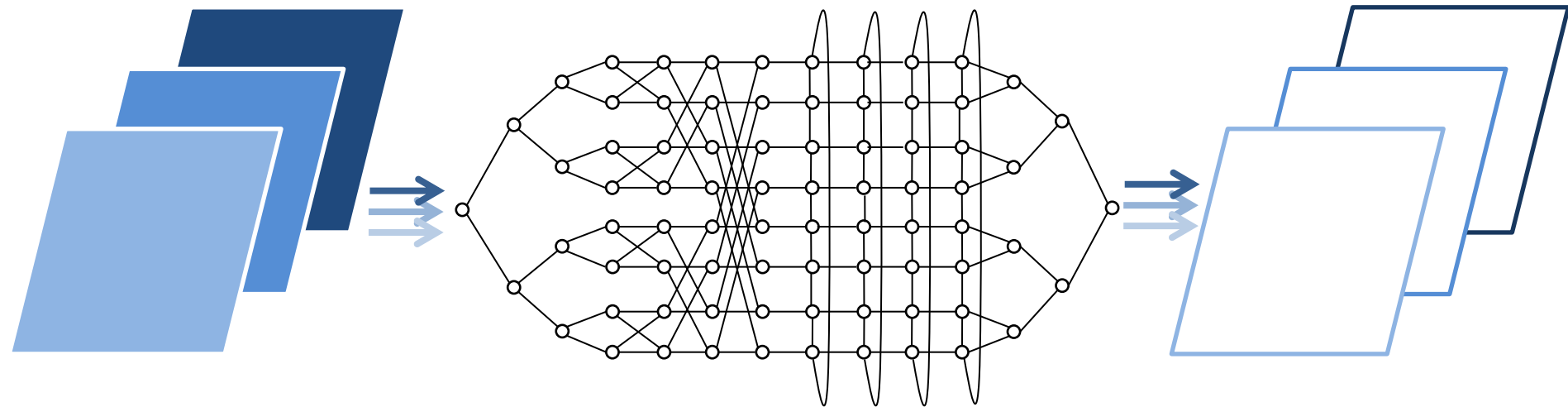
# Réseaux Rapides et Futur du standard MPI

ENSIIE-HPC/BigData-PP-IPAR-Lecture 7

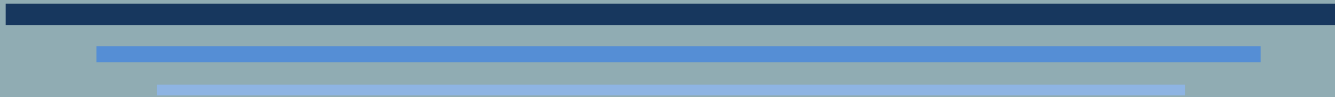
---

---

---



# SOME NETWORK TOPOLOGIES



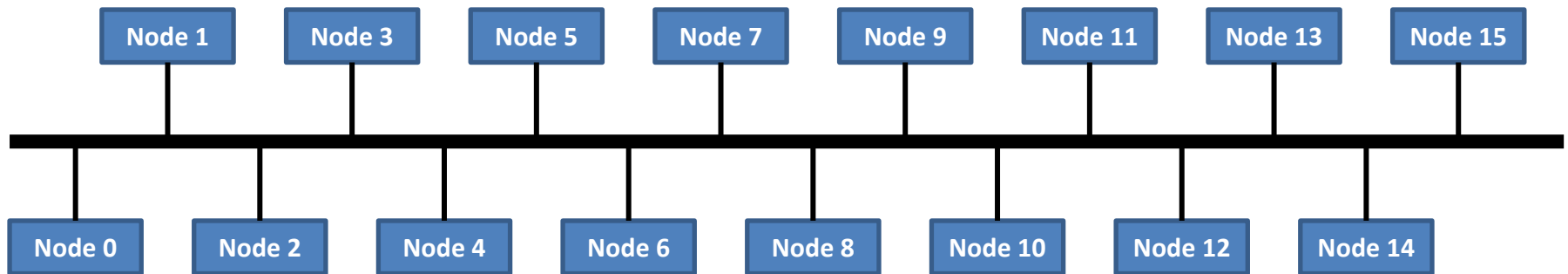
# Network Topology



- Performance of message passing also depends on the network topology
- Numerous network topology are possible
  - Bus
  - Ring
  - Mesh
  - Torus
  - Tree
  - Hypercube
  - Butterfly
  - Dragonfly
  - ...

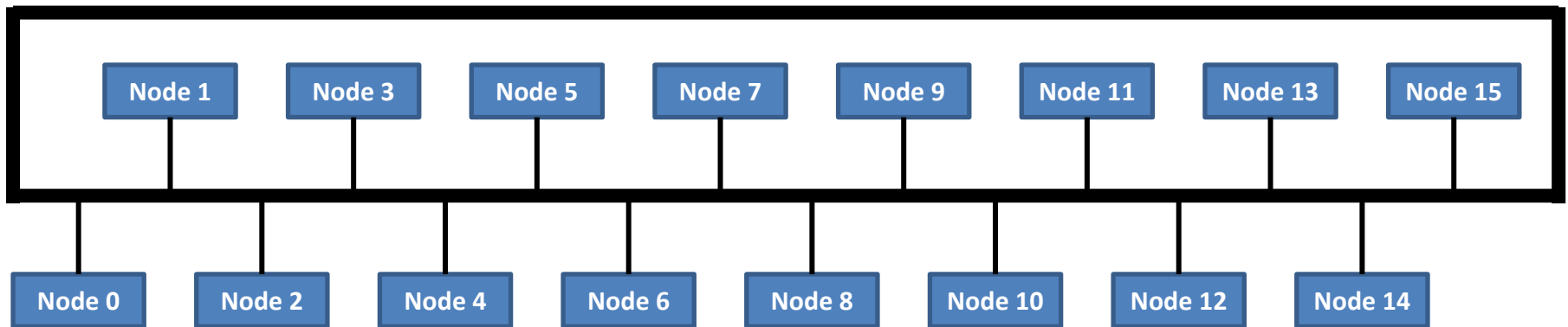
# Bus Topology

- All nodes are connected through a bus
- Simple
- Cost effective for a small number of nodes
- High contention
  - Not scalable for high number of nodes



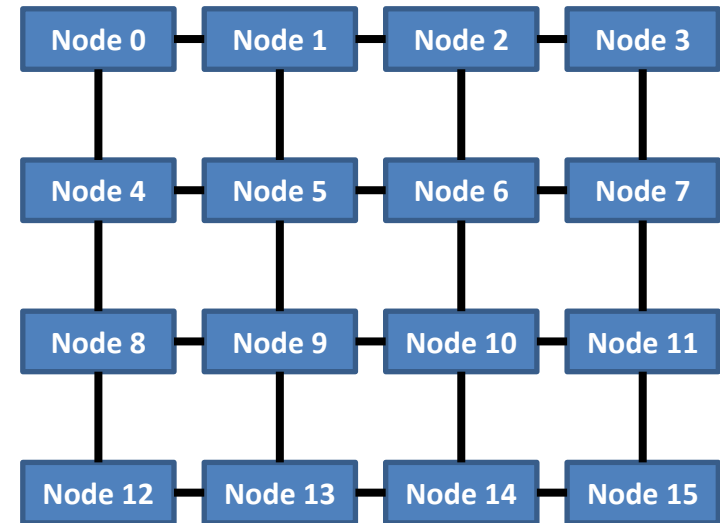
# Ring Topology

- Generalization of Bus topology
- All nodes are connected through a bus
- Both ends of the bus are connected
- Better contention
  - Still not scalable for high number of nodes



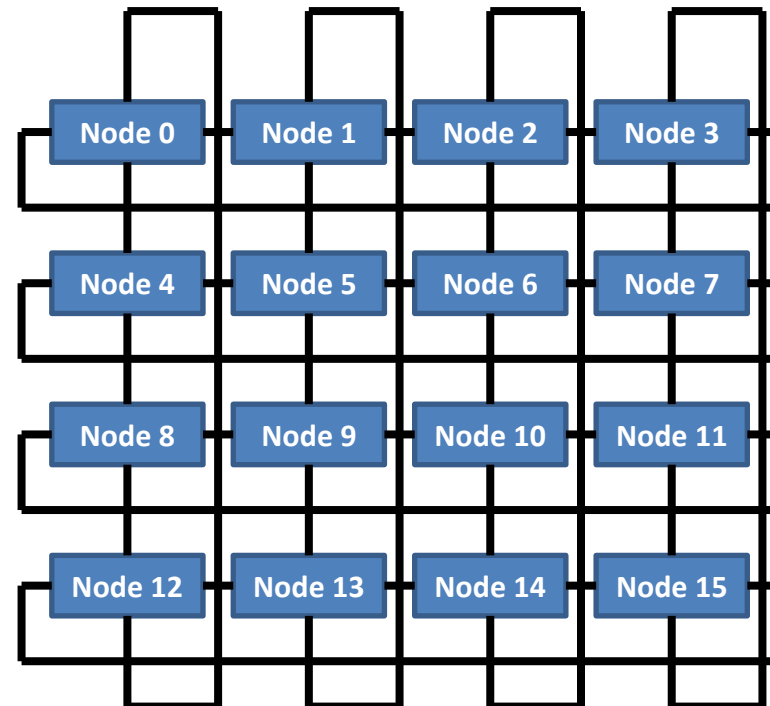
# Mesh Topology

- Average latency:  $O(\sqrt{N})$
- Easy to layout on chip: regular & equal length links
- Path diversity: many ways to get from one node to another
- Same problem as Bus: not regular on edges



# Torus Topology

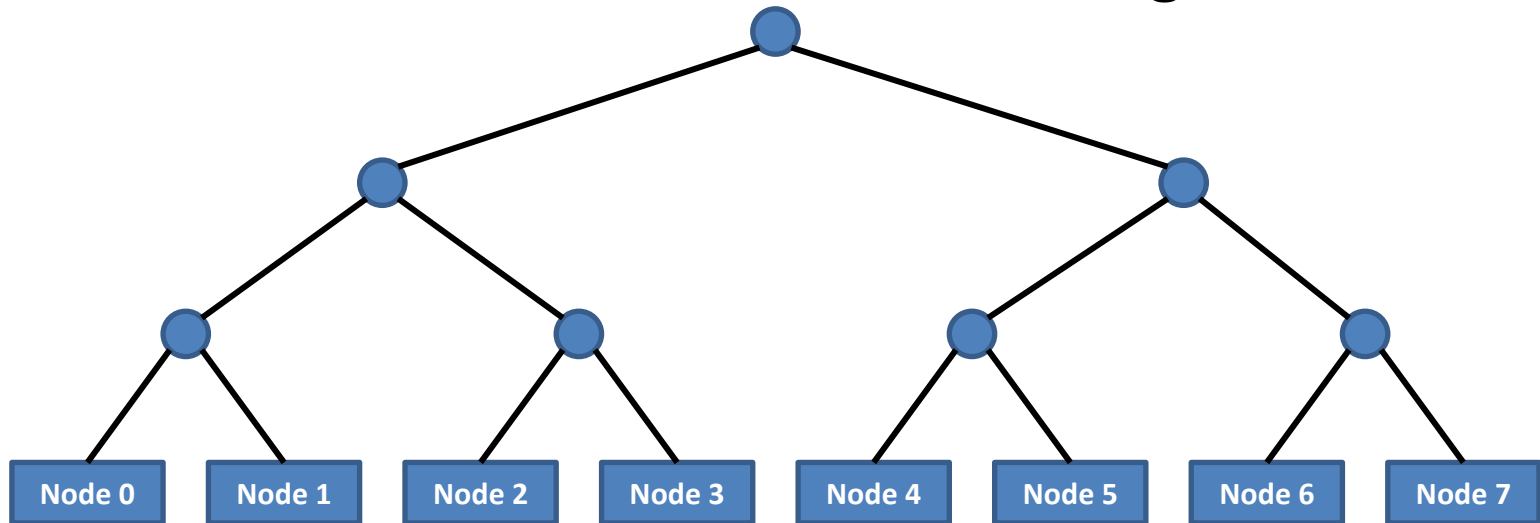
- Mesh is not symmetric on edges
  - performance very sensitive to placement of task on edge vs. middle
- Torus avoids this problem
- Higher path diversity (& bisection bandwidth) than mesh
- Higher cost
- Harder to lay out on chip
- Unequal link lengths





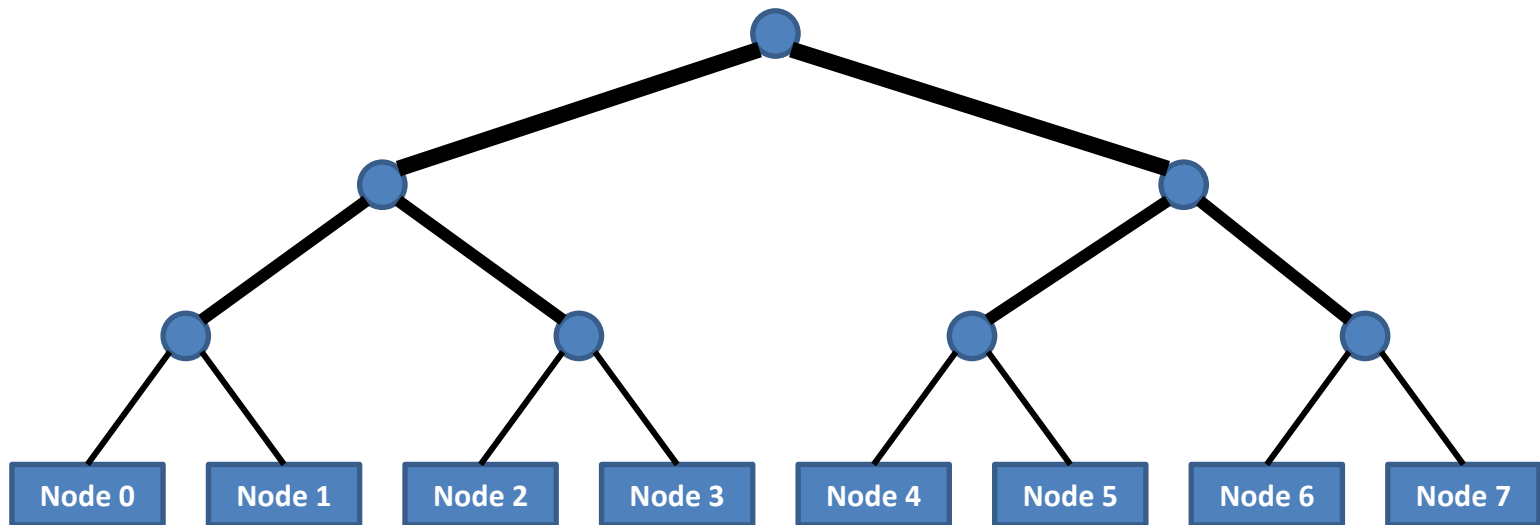
# Tree Topology

- Planar, hierarchical topology
- Latency:  $O(\log N)$
- Easy to Layout
- Good for local traffic, but Root can become a bottleneck if numerous distant messages



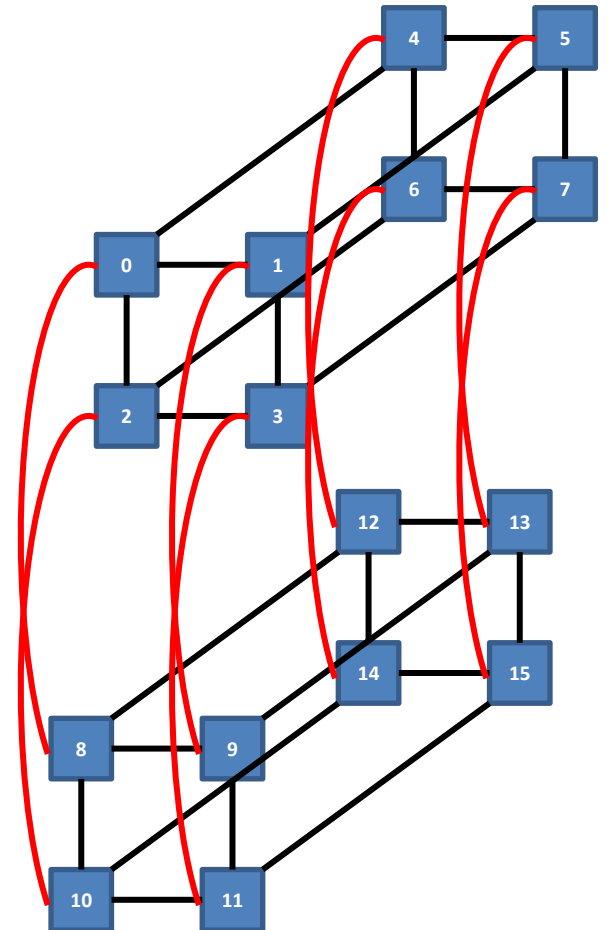
# Fat-Tree Topology

- Fat-tree tries to solve the “root can become a bottleneck” issue
- Each level in the tree is thicker than the previous one
- More bandwidth available at root level than at node level



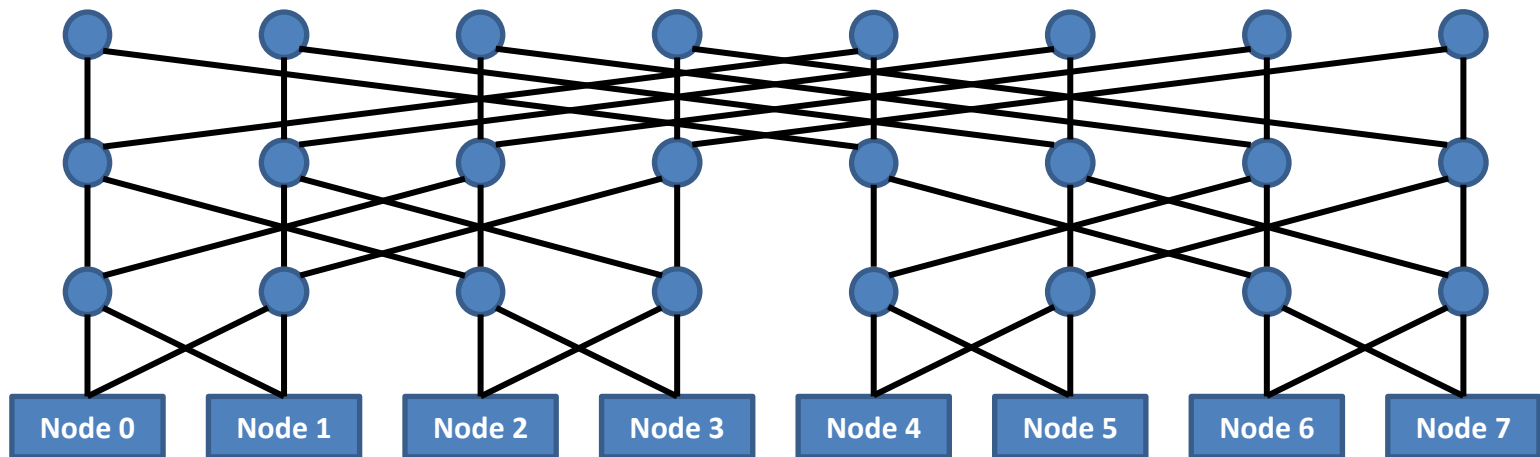
# Hypercube

- Latency:  $O(\log N)$
- Radix:  $O(\log N)$
- number of links:  $O(N \log N)$
- Low latency
- Hard to lay out in 2D/3D



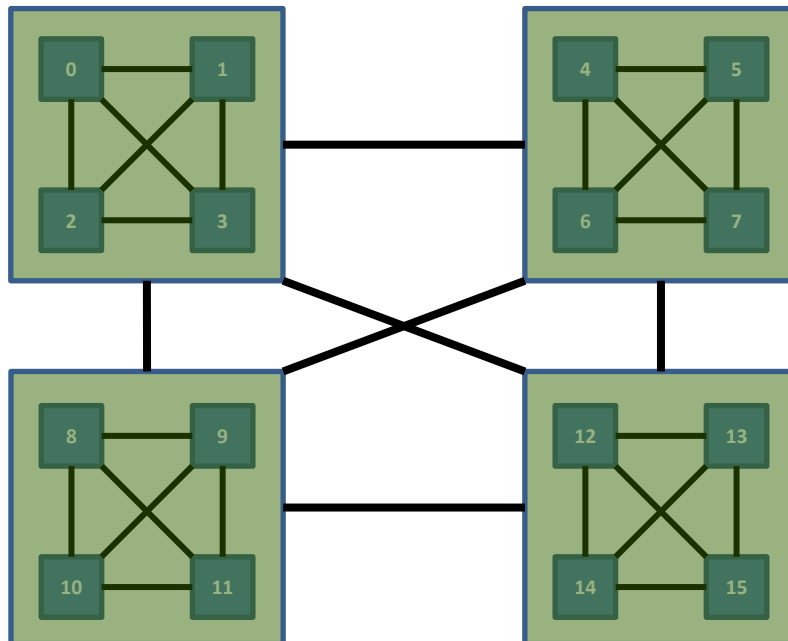
# butterfly

- Lower latency than ring and 2D-mesh
- High number of switches
  - Better resilience to switch failure
- Very similar in complexity to hypercube

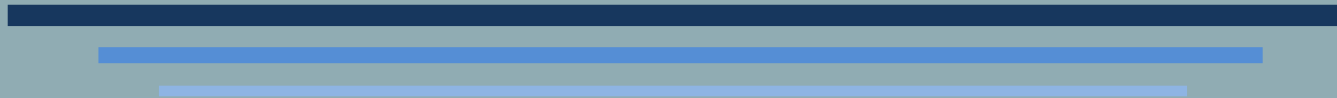


# dragonfly

- Full-connected graphs are nodes of a full-connected super graph





# SOME HIGH SPEED INTERCONNECT



Used in top500 supercomputers

# Mellanox Infiniband

- 
- 
- High Bandwidth link : EDR: 100 Gb/s , HDR: 200 Gb/s
  - Full CPU Offload
    - Kernel bypass: user level application get direct access to hardware
  - Memory exposed to remote node access
    - RDMA-read and RDMA-write
    - Atomic operations
  - Cluster scalability
    - Parallel routes between end nodes
    - Multiple topologies possible
    - Congestion control

# Cray Aries



- High Bandwidth link : 32Gb/s
- Fast Memory Access
  - Minimize overhead for 8-64 byte operations
  - Fast path for single word put and get
- Collective support
  - Latency optimization for their most important cases
    - Integer and floating point add
    - Max/min, compare and swap, bit operations
- Aries dragonfly topology
  - Two dimensions of all-all connected nodes comprise group
  - Average hop count flat regardless of system size
    - Up to 2 hops in source group, 1 hop to dest., up to 2 hops in dest. Group
  - Electrical in groups, optical between groups



# Intel Omnipath



- High Bandwidth link : 100 Gb/s
- Designed for the needs of HPC and RDMA IO
- Data integrity
  - CRC 14bits on data path at link level
  - Several CRCs for Packet integrity
- Congestion management
  - Distributed switch based adaptive routing
  - Dispersal routing
  - Explicit congestion notification protocol

# Fujitsu Tofu2



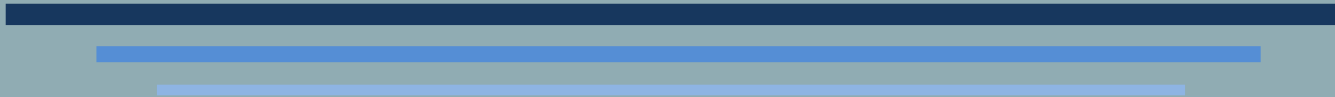
- High Bandwidth link : 25Gb/s
- Cache injection
  - Tofu can inject received data into L2 cache directly
    - bypassing main memory
- Session mode control queue
  - Offloading a collective communication of long messages
  - Command execution may be delayed until reception of a Put
- RDMA Atomic Read-Modify-Write
  - Atomically read, modify and write back remote data
  - Typical operations: compare and swap, fetch and add
  - Mutual atomicity with processor

# Atos/BULL BXI

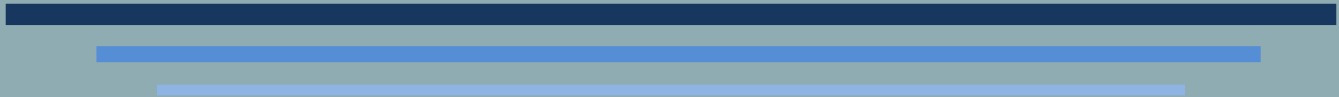


- High Bandwidth link : 100Gb/s
- HW implementation of Portals4 comm. primitives
  - Async progress by offloading to NIC
  - MPI two-sided: acceleration of matching on NIC
  - MPI one-sided: use fast path inside the NIC
- OS and application bypass
  - Applications issue commands to the NIC, avoiding kernel calls
- Data reliability
  - CRC 32bits added to each message
  - Protocol to retransmit lost or corrupted messages



# SOME INTERCONNECT APIS



# VERBS



# What is verbs?

- 
- 
- Verbs is an abstract description of the functionality that is provided for applications for using RDMA.
    - Verbs is not an API
    - There are several implementations for it
  - Verbs can be divided into two major groups
    - Control path – manage the resources and usually requires context switch
      - Create
        - Destroy
        - Modify
        - Query
        - Work with events
    - Data path – Use the resources to send/receive data and doesn't require context switch
      - Post Send
        - Post Receive
        - Poll CQ
        - Request for completion event

# What is verbs?



---

- Verbs is a low level description for RDMA programming
  - Verbs are close to the “bear-metal” and provide best performance
    - Latency
    - BW
    - Message rate
  - Verbs can be used as building blocks for many applications
    - Sockets
    - Storage
    - Parallel computing
- Any other level of abstraction over verbs may harm the performance
- Once you know it, verbs are not so mysterious ...

# libibverbs

- libibverbs is the de-facto verbs API standard in \*nix
  - Developed as an Open source
  - The kernel part integrated in the Linux kernel since 2005 – Kernel 2.6.11
  - Inbox in several \*nix distributions
  - There are level low-level libraries from several HW vendors
- Same API for all RDMA-enabled transport protocols
  - InfiniBand – Networking architecture which supports RDMA
    - requires both NICs and switches that supports it.
  - RDMA Over Converged Ethernet (RoCE)
    - encapsulation of RDMA packets over Ethernet/IP frames
    - requires NICs which supports it and standard Ethernet switches
  - Internet Wide Area RDMA Protocol (iWARP)
    - provides RDMA over Stream Control Transmission Protocol (SCTP) and TCP
    - requires NICs which supports it and standard Ethernet switches



# RDMA device handling

- ***struct ibv\_device \*\*ibv\_get\_device\_list(int num\_devices);***
  - Return a NULL-terminated list of RDMA device that exist in the local host
  - num\_devices is optional. NULL can be provided instead of a valid pointer.
- ***void ibv\_free\_device\_list(struct ibv\_device \*\*list);***
  - Free the list of RDMA devices that was returned from ibv\_get\_device\_list()
- ***const char \*ibv\_get\_device\_name(struct ibv\_device \*device);***
  - Return a string that describe the name of the RDMA device

# RDMA device handling



- ***struct ibv\_context \*ibv\_open\_device(struct ibv\_device \*device);***
  - Return a context from an RDMA device
    - This context will be used to create resources in this device
  - Notice the following fields in struct ibv\_context:
    - num\_comp\_vectors – Number of completions vectors that this device supports
    - async\_fd – File descriptor that will be used to report about asynchronous events from kernel
- ***int ibv\_close\_device(struct ibv\_context \*context);***
  - Close the context of the RDMA device
    - This verb should be called after destroying all the resources that were created using this context Not doing so will cause a memory leak

# Memory region



- Memory Region is a virtually contiguous memory block that was registered, i.e. prepared for work with RDMA.
  - Any memory buffer in the process' virtual space can be registered
  - Available permissions. One or more of the following permissions:
    - Local operations (Local Read is always supported)
      - IBV\_ACCESS\_LOCAL\_WRITE
      - IBV\_ACCESS\_MW\_BIND
    - Remote operations
      - IBV\_ACCESS\_REMOTE\_WRITE
      - IBV\_ACCESS\_REMOTE\_READ
      - IBV\_ACCESS\_REMOTE\_ATOMIC
- If Remote Write or Remote Atomic is enabled, local Write should be enabled too
- The same memory buffer can be registered multiple times
  - even with different permissions

# Memory region



- ***struct ibv\_mr \*ibv\_reg\_mr(struct ibv\_pd \*pd, void \*addr, size\_t length, enum ibv\_access\_flags access);***

- Register a memory buffer with specific permissions
- Notice the following fields in struct ibv\_mr:
  - lkey - The local key of this MR
  - rkey - The remote key of this MR
  - addr – The start address of the memory buffer that this MR registered
  - length – The size of the memory buffer that was registered

- ***int ibv\_dereg\_mr(struct ibv\_mr \*mr);***

- Deregister a Memory Region
- This verb should be called if there is no outstanding Send Request or Receive Request that points to it

# Queue Pair

- Queue Pair is the actual object that transfers data
  - It encapsulates both Send and Receive Queue
  - Each of them is completely independent
- A QP represent a real HW resource
- ***struct ibv\_qp \*ibv\_create\_qp(struct ibv\_pd \*pd, struct ibv\_qp\_init\_attr \*qp\_init\_attr);***
  - Create a new Queue Pair
- ***int ibv\_destroy\_qp(struct ibv\_qp \*qp);***
  - Destroy a Queue Pair
- ***int ibv\_modify\_qp(struct ibv\_qp \*qp, struct ibv\_qp\_attr \*attr, enum ibv\_qp\_attr\_mask attr\_mask);***
  - Modify the QP attributes

# Post send request



- Add a Send Request to the Send Queue
  - No context switch will occur
  - The HW will process it according to its scheduling algorithm
- Specify the attributes of the data transfer
  - How data will be sent (opcode, attributes)
  - How much data will be sent
  - Which local memory buffer(s) to read/write to
    - Depends on the opcode
- Every Send Request is considered outstanding until a work Completion was generated for it or a following Send Request
  - While a Send Request is outstanding, the resources that this Send Request use must not be destroyed/(re)used
    - The content of memory buffers that their content will be filled is undefined
    - The memory buffers that their content is sent must be available

# Post receive request



- Add a Receive Request to the Receive Queue
  - No context switch will occur
  - The HW will process it according to its scheduling algorithm
- Specify where incoming message that needs Receive Request will be saved
  - The local memory buffer(s) to write to
  - Each incoming message will consume one Receive Request
  - The S/G list must be able to hold the incoming message
- Every Receive Request is considered outstanding until a work Completion was generated to it
  - While a Receive Request is outstanding, the resources that this Receive Request use mustn't be destroyed/(re)used
  - The content of memory buffers that their content will be filled is undefined

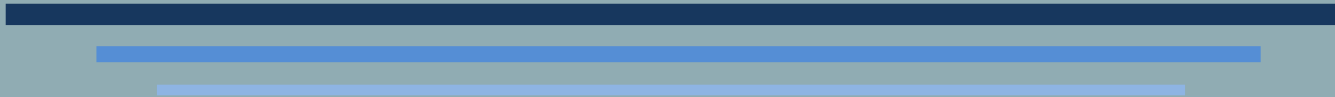
# Post, receive en completion API




- ***int ibv\_post\_send(struct ibv\_qp \*qp, struct ibv\_send\_wr \*wr, struct ibv\_send\_wr \*\*bad\_wr);***
  - Add a linked list of Send Requests to the Send Queue
- ***int ibv\_post\_recv(struct ibv\_qp \*qp, struct ibv\_recv\_wr \*wr, struct ibv\_recv\_wr \*\*bad\_wr);***
  - Add a linked list of Receive Requests to the Receive Queue
- ***int ibv\_poll\_cq(struct ibv\_cq \*cq, int num\_entries, struct ibv\_wc \*wc);***
  - Read one or more Work Completions from a CQ and remove them from the CQ
  - If the return value is non-negative – this is the number of polled Work Completions
  - If the return value is negative – error occurred





# PORTALS 4



# Portals ?

- 
- First implementation in 2007 (4th edition in April 2017).  
Initially raised by Sandia National Laboratories
  - Low-level RDMA-based network programming Interface.
  - Aims to target zero-copy messages and facilitate any software bypass (OS, application)
  - Build basic blocks, easy to reuse for upper-layer semantics (MPI, PGAS, SHM...)
  - Designed to be as scalable as the target application

# Communication



- 
- 
- Targets one-sided & two-sided communications
  - Rely on Put/Get/Atomics requests (RMDA)
  - And remotely matched by the target (filtering)
  - One process send a request (initiator) and one process resolves it (target) without interacting with the software stack
  - No explicit addressing, Portals uses the matching process to associate a given request to a local resource without involving neither the OS or the application
  - Application can infer with resource usage through published events (asynchronous polling)
  - Forward flow-control to the application (don't impact performance)

# Message matching

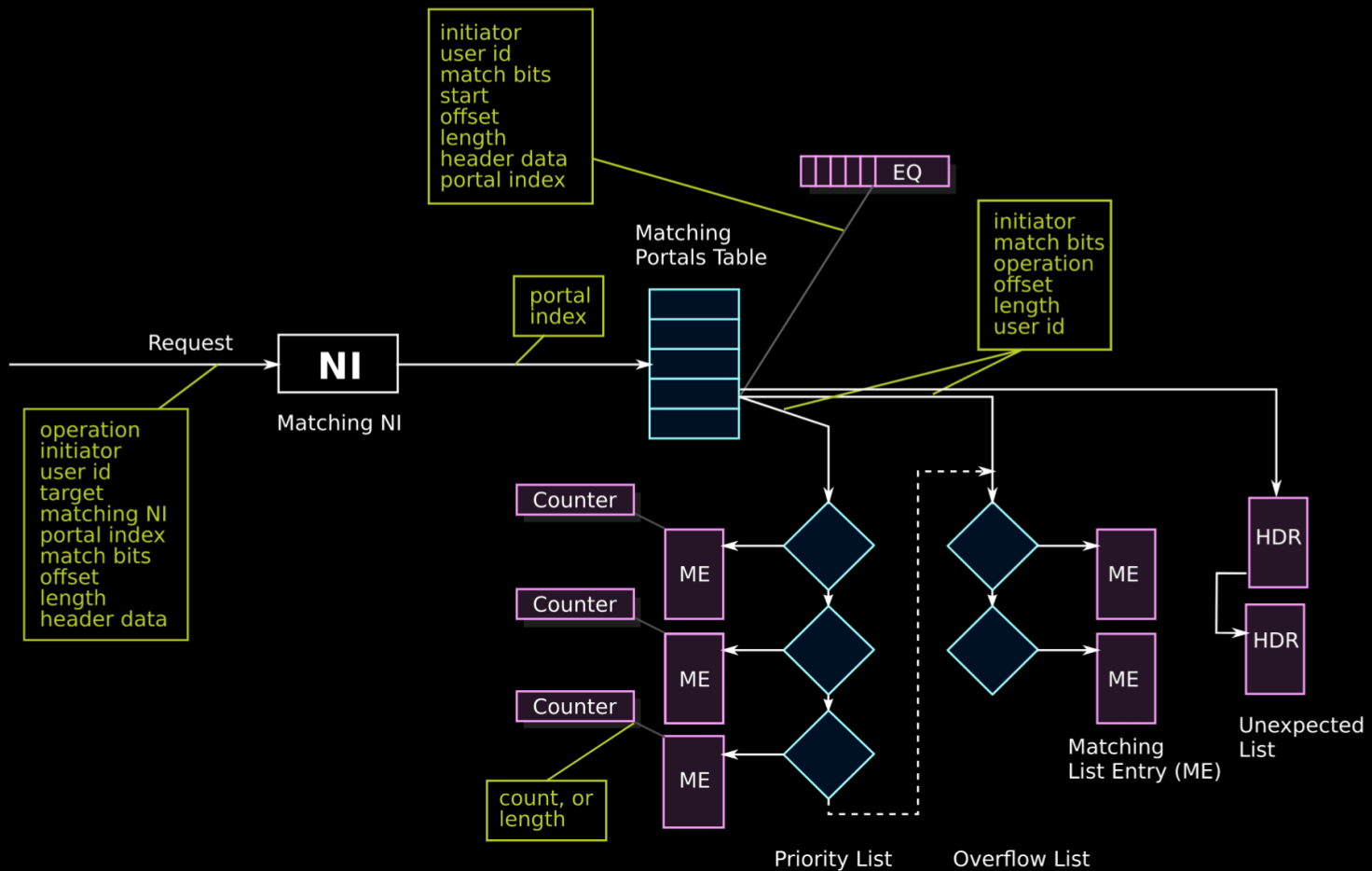


- User register memory segment (ME) to be accessed from the network (=pinning). Memory is identified by a remote process ID and a matching sequence
- Remote requests have to match with process ID (authentication), operation type (Put/Get) and matching bits to be allowed to alter data.
- To avoid high contention on MEs, they are packed in multiple lists (Portals entries). This adds another scattering criteria
- The message ordering is guaranteed for any request reaching the same Portals entry (from target side)

# Events

- 
- 
- Each time a new operation is submitted (and data might be altered), an event is appended to the appropriated queue
  - 2 kinds of events:
    - Full-events: Gathers informations about the ME and the request impact on the memory segment. This events are gathered in a dedicated Event Queue (EQ) for ME's full-events.
    - Counting events (CT): increment the success/failure counter associated with an operation. This reduces event generation overhead
  - A same EQ (or CT) can be associated to multiple operations.

# Model

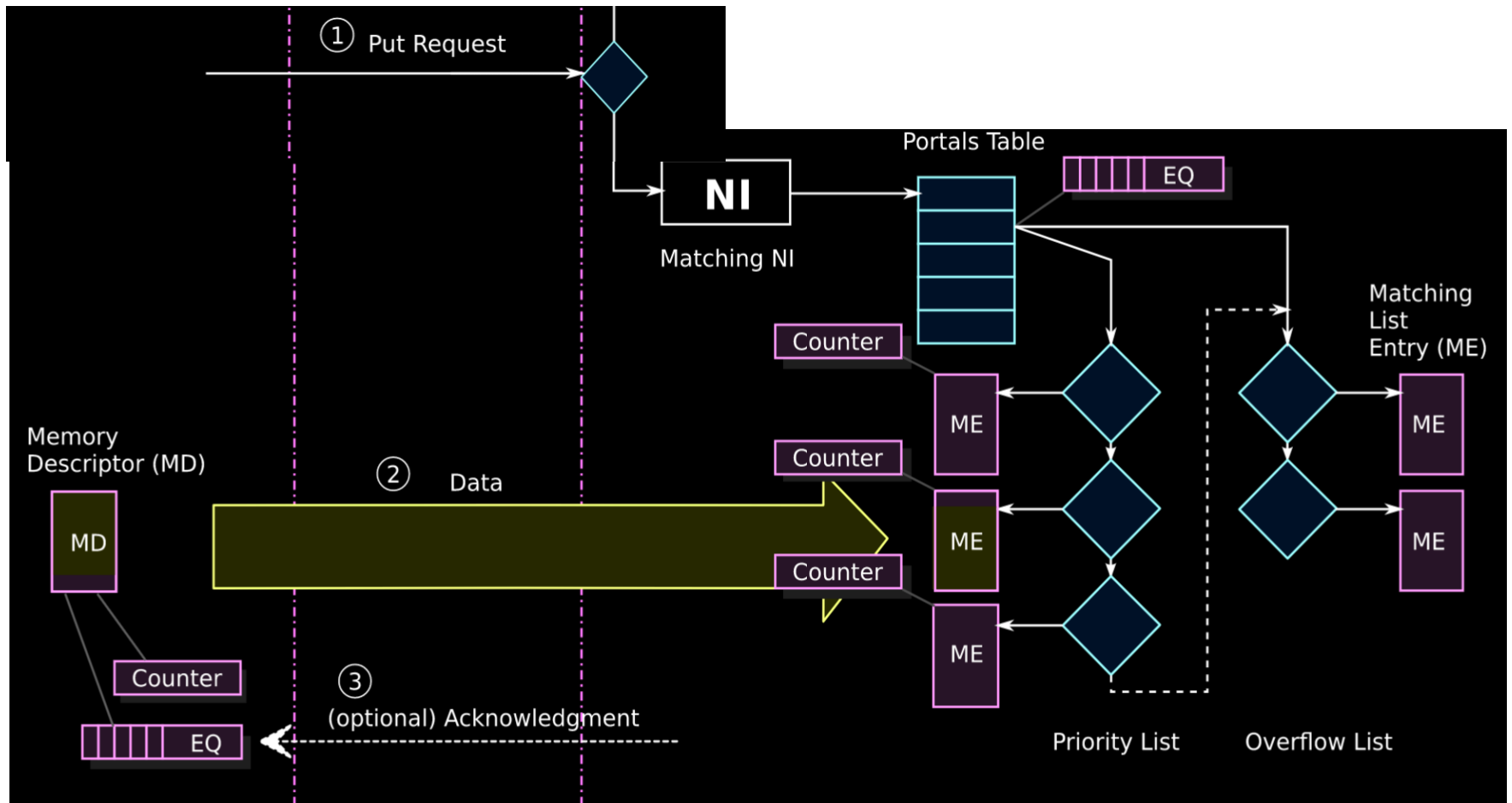


# Portals Objects



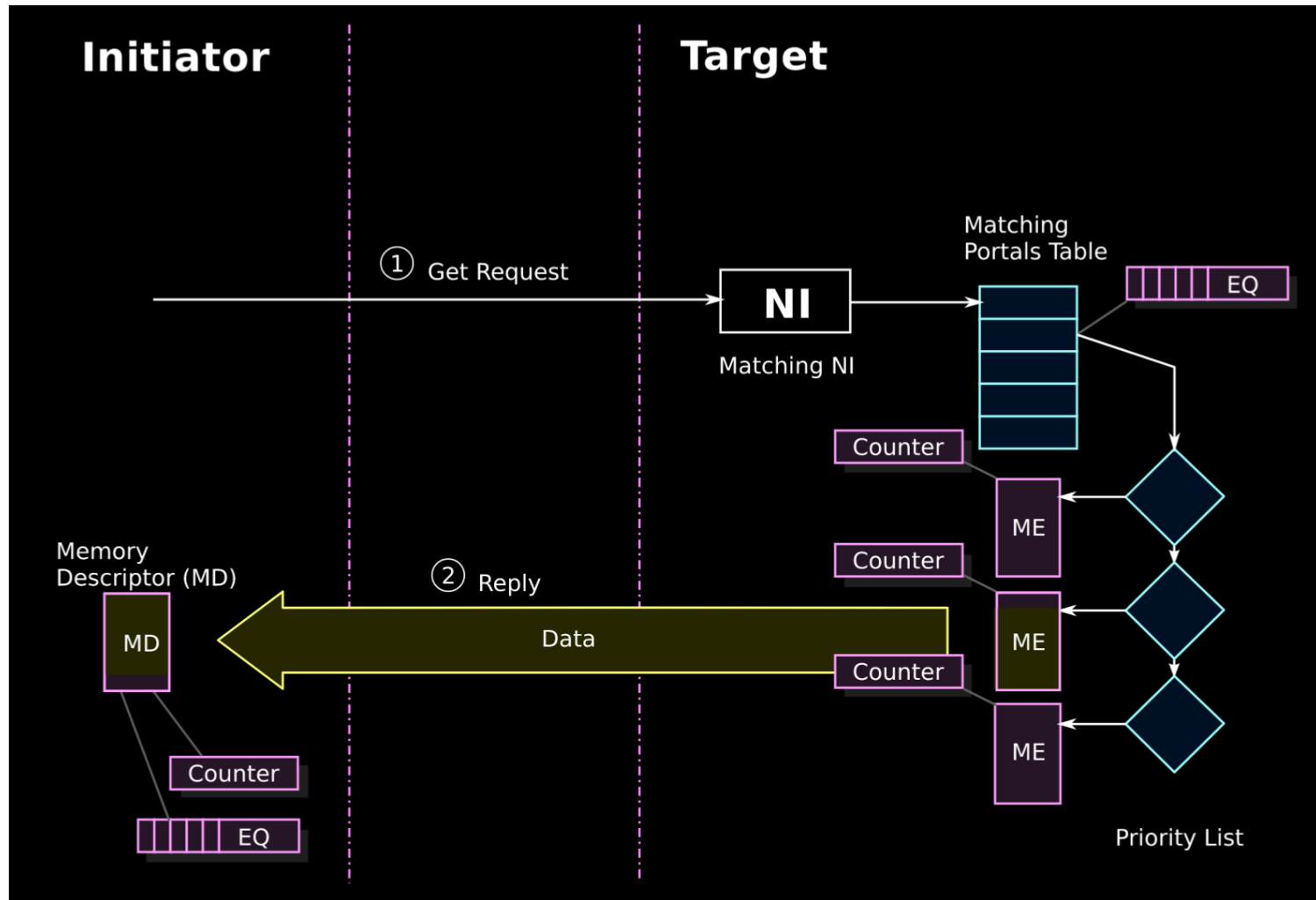
- NI: Physical representation of a network interface from Portals.
  - An NI can be also be logical.
- ME: Memory entry.
  - Segment mapped by the target to be remotely accessible.
- PTE: Portals entry.
  - The remote request entry point, gathering a ME list
- MD: Memory Descriptor.
  - Memory prepared at the initiator side, ready to send/recv data.
- PRIORITY\_LIST:
  - Each Portals entry hosts a priority list. Each posted ME is registered whether in this list or in the OVERFLOW\_LIST.
- OVERFLOW\_LIST:
  - Designed to handle unexpected messages from two-sided operations. If the remote request is received before the local process posted the associated ME, this list will allow to flag the incoming request as pending.

# PtIPut





# PtlGet

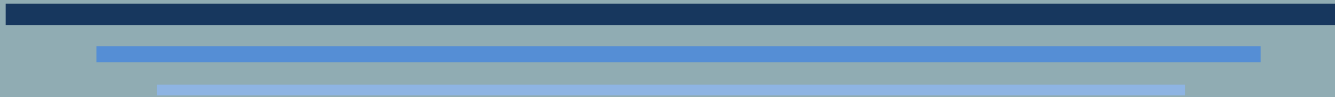


# Portals: What's different?



- Connection-less (no route establishment)
- Low-level message matching (much faster)
- Flexible basic blocks (ME, MD...) facilitate the implementation from higher-level protocols.
- Memory affinity (user-space, kernel-space, NIC memory...)
- Asynchronous message progression
- And with Portals 4:
  - Collective Offloading (TriggeredOps)
  - MPI nightmare handling: « unexpected messages »
  - Integrated flow-control

# MPI FORUM



# MPI Forum

- The MPI Forum is the standardisation forum for Message Passing Interface (MPI)
- The MPI Forum has several roles:
  - Conception of the MPI standard
  - Correction of text errors in the standard
  - Discussion of new directions for the standard
  - Suggestion of modifications to the standard
    - To add new features
    - To deprecate features
    - To improve/maintain the coherency of the document
    - To improve/maintain quality and readability of the document
- The MPI Forum represents MPI in the scientific community
  - ...and should be the link between the needs of scientists and the redaction (and implementation) of the MPI standard.



# MPI Forum organization



---

- The MPI Forum is composed of several scientists of numerous organizations
  - National labs, Microsoft, Intel, Nvidia, IBM, CISCO, Mellanox, HLRS, JSC, CEA, Paratools, Universities (Tennessee, Urbana-Champaign, Bordeaux, EPCC, ...) ...
- Separated in two overlapping sub-committee:
  - Chapter committee, focusing on the quality and coherency of the document
    - Each chapter in the MPI Standard has its own president.
  - Work Groups, discussing the evolution of the standard (new ideas, corrections, deprecations, ...)
    - Each Work Group focus on its own subject (IO, RMA, collectives, hybrid, tools,...)
    - Bi-weekly or monthly phone conf to discuss subjects on the Work Groups

# MPI Forum meetings



## ■ MPI Forum Meetings

- Face-to-face meetings every three months
- Three in the US, and one with EuroMPI
- Work Group discussions, presentations of new idea, modifications or results
- Vote to include new features in the next version of the standard

## ■ Voting rules

- Voting right if the organization has been present 1 time in the two previous meeting
  - No membership fee
  - Registration fee for the attended meetings
- Quorum must be reached: 2/3 of organizations with voting right
- API modification must be submitted in formatted version 2 weeks before the meeting
  - If text has been touched, no-no votes is required
- For errata correction, only one vote is necessary
- For big modification, one formal reading and two votes are required
  - Each step at a different Meeting

# ULFM (1)



## ■ GOAL

- Allow wide range of fault tolerance techniques by adding a minimal set of FT function to the MPI Standard
- Not necessarily designed to be simplest to use, but feature complete
- Is designed to encourage libraries to sit on top of ULFM and provide more user-friendly semantic

## ■ What is ULFM trying to solve?

- Failure Notification
  - Error codes
  - New API for getting group of failed processes
- Failure Propagation
  - Local notification
  - New API for notifying other processes
- Failure Recovery
  - Point-to-point
    - Nothing required
  - Communicator
    - New API to create communicator without failed processes

# ULFM (2)



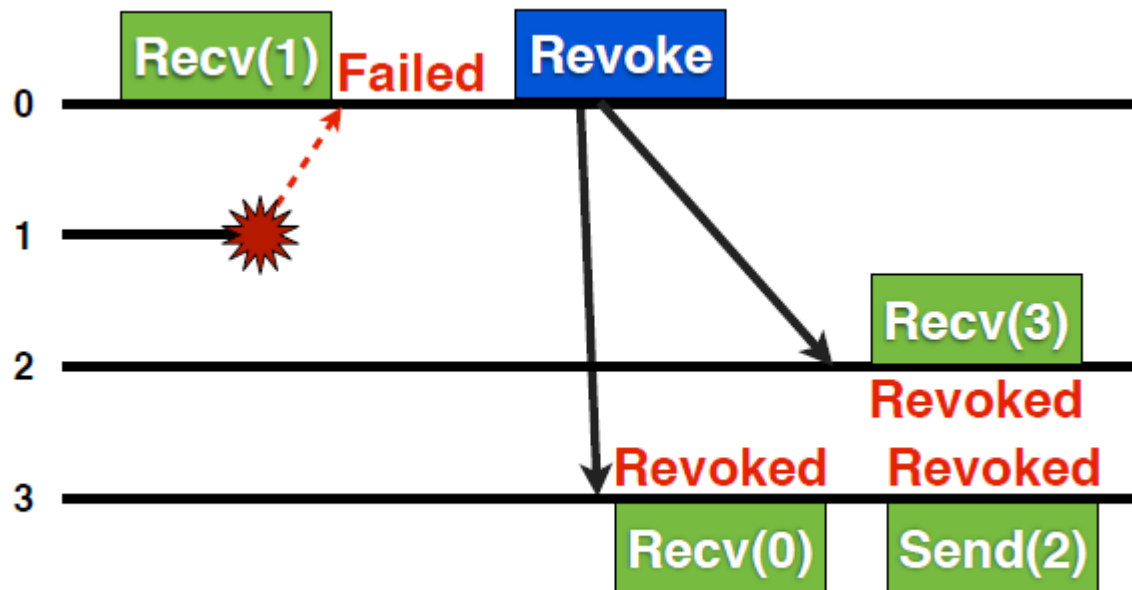
- The following error classes are added
  - MPI\_ERR\_PROC\_FAILED
    - The operation could not complete because of a process failure (a fail-stop failure).
  - MPI\_ERR\_PROC\_FAILED\_PENDING
    - The operation was interrupted by a process failure (a fail-stop failure). The request is still pending and the operation may be completed later.
  - MPI\_ERR\_REVOKED
    - The communication object used in the operation has been revoked.



# ULFM (3)

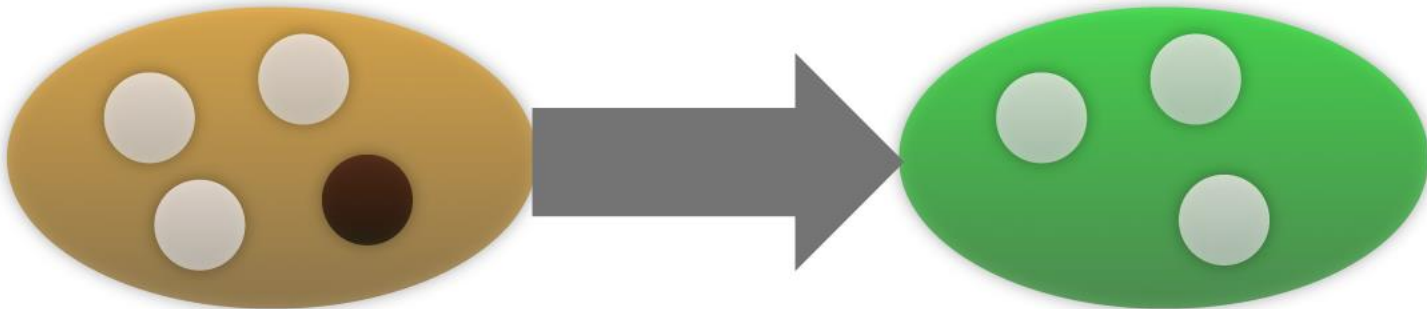
## ■ MPI\_Comm\_revoked(MPI\_Comm comm)

- Disables further (non-local) use of comm
- Local function
- Global effect (over *comm*)
- Other



# ULFM (4)

- `MPI_Comm_shrink (MPI_Comm comm, MPI_Comm* newcomm)`
  - Creates a new communicator with all of the processes from the original communicator
    - Excluding all failed processes
  - Old communicator is still around and usable (if not revoked)



# MPI Fsend – Arecv / MPI Persistent Coll.



## ■ Fsend – Arecv

- ULTIMATE GOAL: allow buffer ownership passing to avoid the communication
- Fsend: Freeing Send
  - The input buffer in the send communication function is freed upon return
- Arecv: Allocating Recv
  - The output buffer to receive the message is not given by the user, but allocated directly by the MPI runtime
- In shared memory, this means that you can “give” the input buffer to the receiving rank. No communication/copy is required.

## ■ MPI Persistence Collectives

- GOAL: initialize the collective once with a specific set of parameters, and use multiple times with the same configuration
- Allows to set aggressive communication optimization for this collective
  - Setting these optimizations may be time consuming and not efficient for a unique call to the collective
  - Using persistent collectives should ensure that the communication will be used several times

# Endpoints: Why?



## ■ Performance concerns

- MPI-3.1 provides a single view of the MPI stack to all threads
  - Requires all MPI objects (requests, communicators) to be shared between all threads
  - Not scalable to large number of threads
  - Inefficient when sharing of objects is not required by the user

## ■ Interoperability concerns

- MPI-3.1 does not allow a high-level language to interchangeably use OS processes or threads
  - No notion of addressing a single or a collection of threads
  - Some high-level languages do not expose whether their processing entities are processes or threads (E.g., PGAS languages)
- In MPI-3.1, there is no notion of sending a message to a thread
  - Communication is with MPI processes
  - May emulate such matching with tags or comm., but hard and inefficient

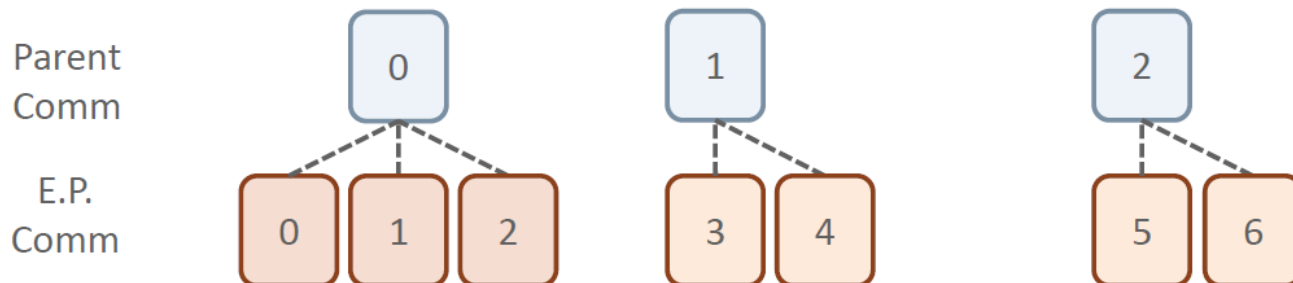
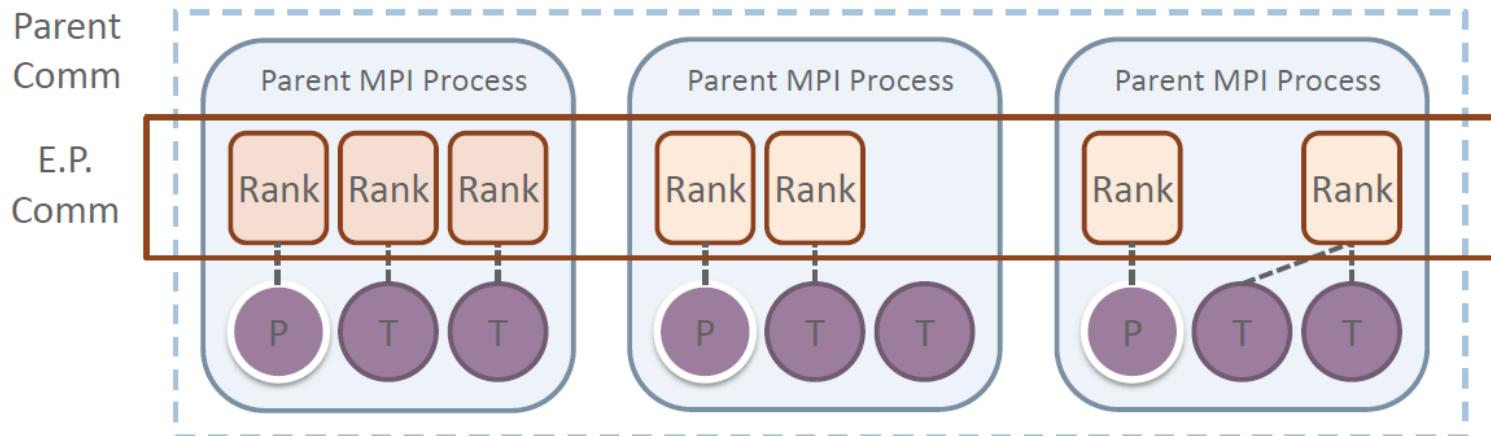
# Endpoints proposal

- MPI Endpoints: Proposal for MPI-4
  - Idea is to have multiple addressable communication entities within a single process
    - Instantiated in the form of multiple ranks per MPI process
  - Each rank can be associated with one or more threads
  - Lesser contention for communication on each “rank”
  - In the extreme case, we could have one rank per thread (or some ranks might be used by a single thread)
- `MPI_Comm_create_endpoints(MPI_Comm parent_comm, int my_num_ep, MPI_Info info, MPI_Comm out_comm_handles[])`
  - Creates new MPI ranks from existing ranks in parent communicator
    - Each process in parent comm. requests a number of endpoints
    - Array of output handles, one per local rank (i.e. endpoint) in endpoints communicator
    - Endpoints have MPI process semantics (e.g. progress, matching, ...)
  - Threads using endpoints behave like MPI processes
    - Provides per-thread communication state/resources
    - Allows implementation to provide process-like performance for threads

# Endpoints proposal

## MPI Endpoints

Relax the 1-to-1 mapping of ranks to threads/processes



# MPI Sessions: Why?



- In a perfect world:

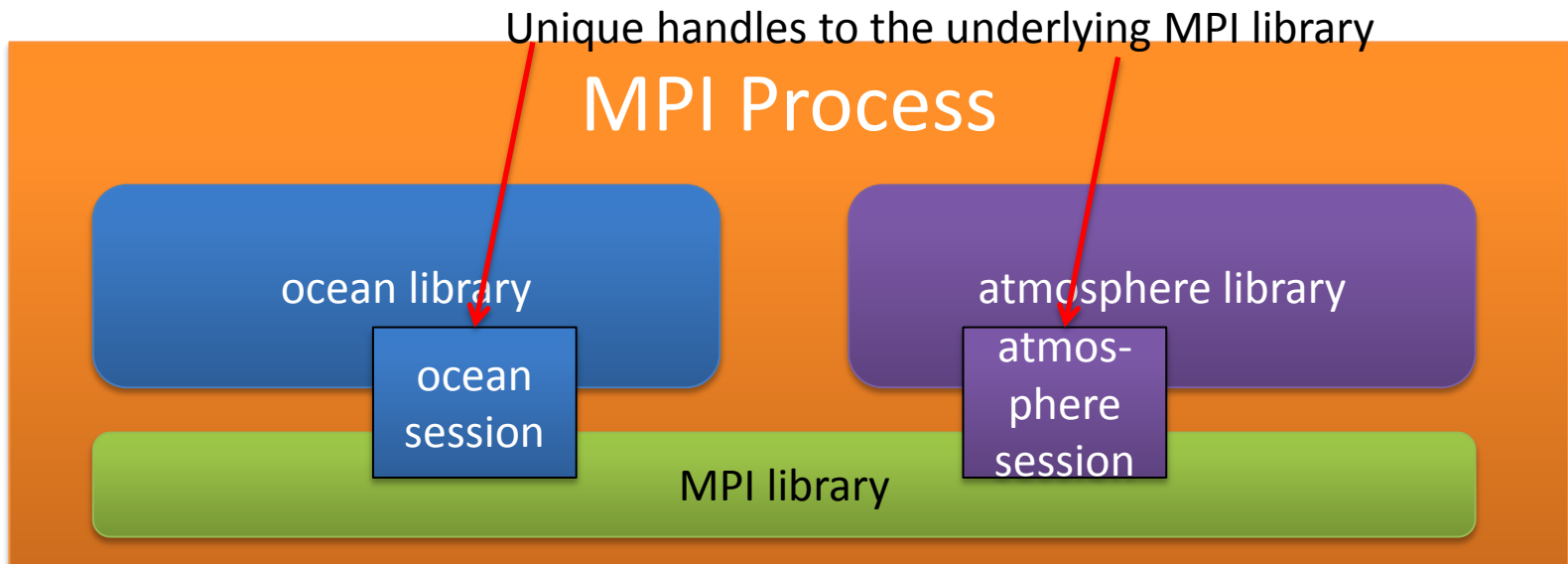
- Any thread (e.g., library) can use MPI any time it wants
- But still be able to totally clean up MPI if/when desired
- New parameters to initialize the MPI API

- Fix MPI-3.1 limitations:

- Cannot init MPI from different entities within a process without a priori knowledge / coordination
- Cannot initialize MPI more than once
- Cannot set error behavior of MPI initialization
- Cannot re-initialize MPI after it has been finalized

# New concept of sessions

- A local handle to the MPI library
  - Implementation intent: lightweight / uses very few resources
  - Can also cache some local state
- Can have multiple sessions in an MPI process
  - `MPI_Session_init(..., &session);`
  - `MPI_Session_finalize(..., &session);`



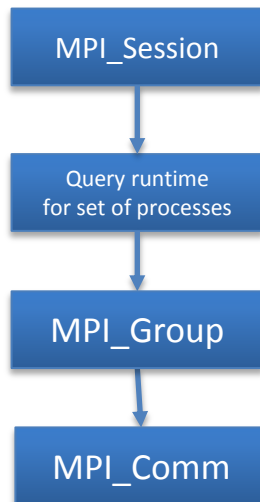


# With runtime cooperation



## ■ General scheme:

- Query the underlying run-time system
  - Get a “set” of processes
- Determine the processes you want
  - Create an MPI\_Group
- Create a communicator with just those processes
  - Create an MPI\_Comm



## ■ Runtime concept

- Expose 2 concepts to MPI from the runtime:
  - Static sets of processes
  - Each set caches (key,value) string tuples

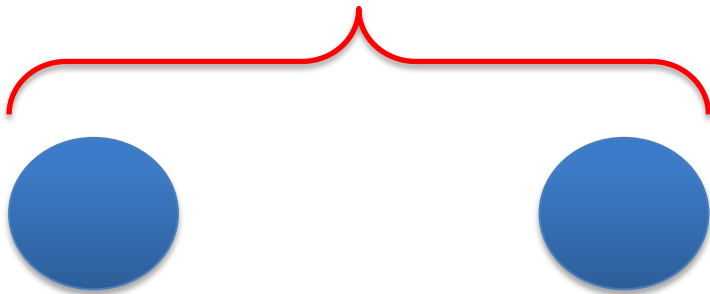
## ■ Getting a group of processes

- Sets are identified by string name
- Two sets are mandated
  - “mpi://WORLD” and “mpi://SELF”
- Other sets can be defined by the system:
  - “location://rack/19”
  - “network://leaf-switch/37”
  - “arch://x86\_64”
  - “job://12942”
  - ... etc.
- These names are implementation-dependent
- Processes can be in more than one set

# Example of sets

```
■ mpiexec \  
    --np 2 --set user://ocean      ocean.exe : \  
    --np 2 --set user://atmosphere atmosphere.exe
```

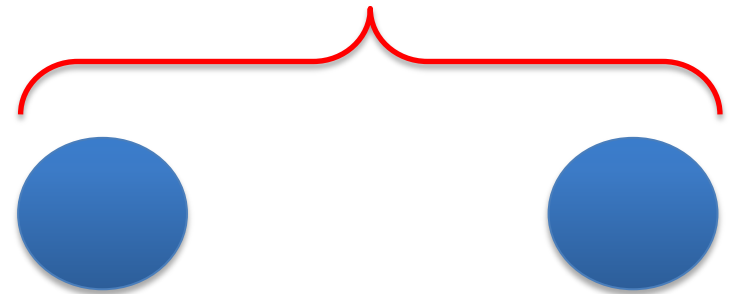
user://ocean



MPI process 0

MPI process 1

user://atmosphere



MPI process 2

MPI process 3

■ Each set has an associated MPI\_Info object

- One mandated key in each info:
  - “size”: number of processes in this set
- Runtime may also provide other keys
  - Implementation-dependent

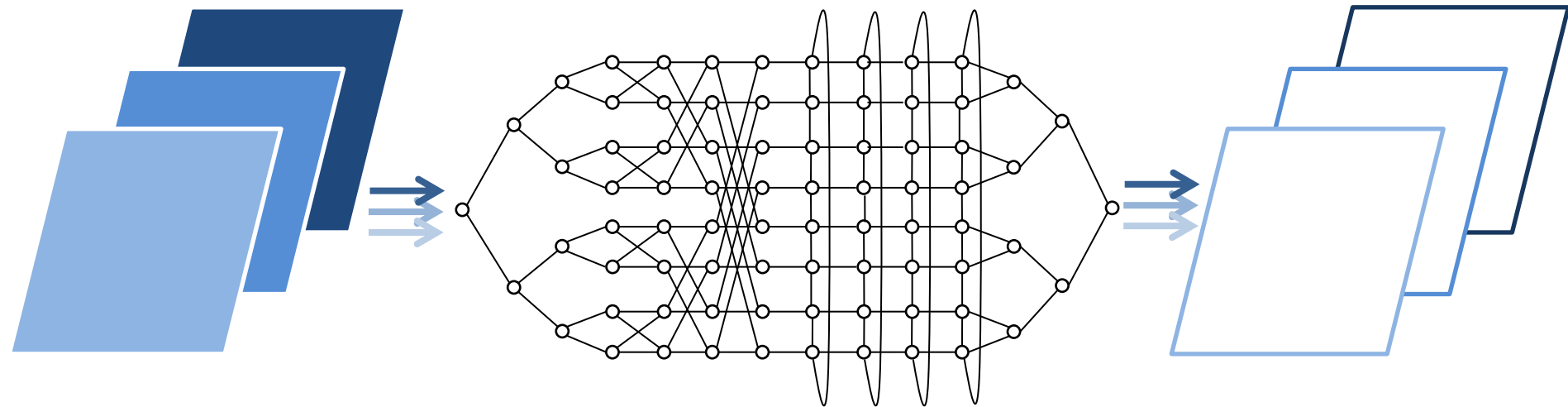
# Programmation Parallèle

**MPI:** Message Passing Interface

---

---

---



**FIN DU COURS**