Technical University of Munich

Lecture 10: Distributed computing - Part 1

Informatik elective: GPU Computing

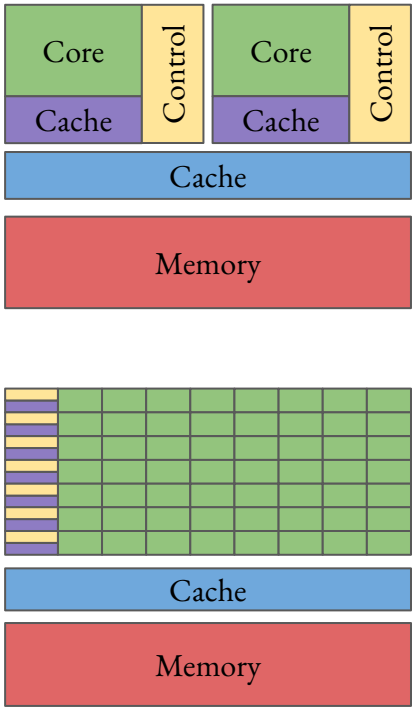
Pratik Nayak

In this session

- Distributed computing basics
- The MPI programming model
  - Groups and communicators
  - Point to point communication
  - Collective communication

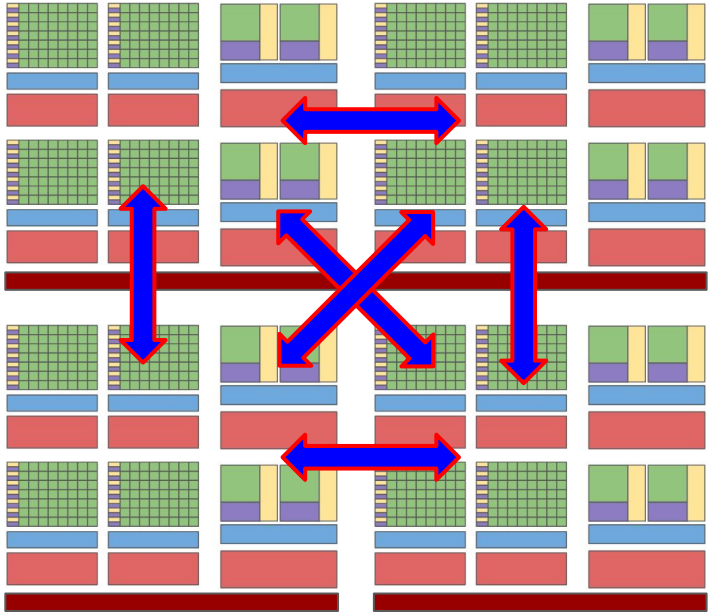
Shared memory architectures

- Shared Memory: Many compute units sharing the same contiguous memory: Modern CPUs and GPUs.
- Limited, but fast main memory accesses.
- Programmable using shared memory programming models.
  - OpenMP, CUDA, SYCL etc.



Distributed memory architectures

- Large scale computing requires collection of shared memory architectures.
- Large but slow (wrt BW and latency) memory accesses
- Programmed through distributed memory programming models
  - MPI, PGAS etc
- Multiple nodes interconnected through a network.



Distributed system: 4 nodes, each with 4 GPUs and 2 CPUs

Parallel programming models

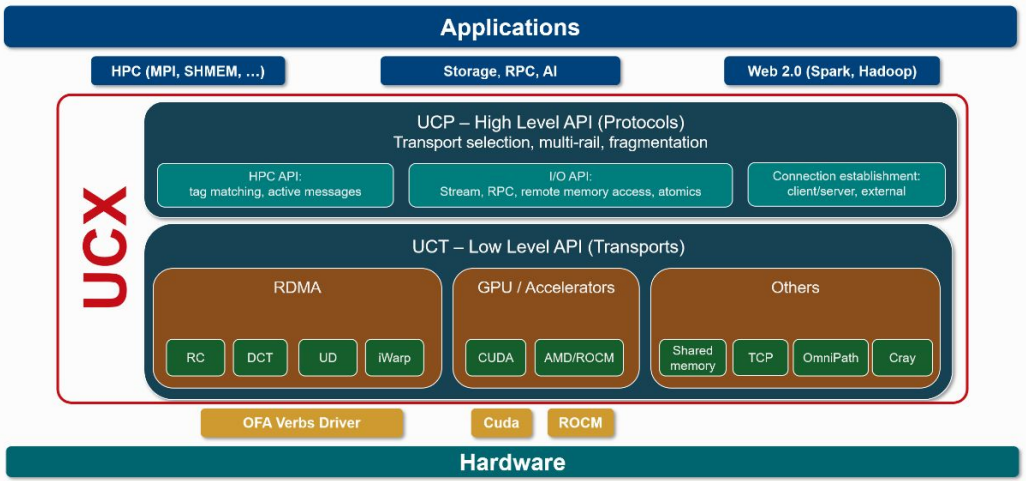
	Parallel programming models		
	Shared memory	Distributed memory	Distributed shared memory
Memory unit	Physically shared memory units	Memory units connected through a network	Virtually shared memory units
Examples	CUDA, SYCL, OpenMP	Message passing interface (MPI)	PGAS
Latency	Low latency	Higher latency	Higher latency
Communication	Implicit	Explicit	Implicit

Programming distributed memory - Languages / frameworks

- Need a way to communicate with neighbour nodes connected with a network.
- Many frameworks exist, which provide semantics for distributed communication and computation:
  - Chapel
  - Unified Parallel C (UPC)
  - Apache Hadoop
  - Apache Spark
  - Message Passing Interface
- ... and others
- Choice depends on application, required flexibility, ease of use and performance.
- In general, all of them are layers that help communicate between the node-level memories and the network drivers.

Programming distributed memory - Network

- Multiple networking options available:
  - Ethernet/iWARP: Networking over internet protocols (TCP/IP)
  - Ethernet/RoCE: Networking over ethernet.
  - High performance networks: Infiniband, Intel Omni-path (better latency and bandwidths).
- UCX is one such communication layer that tries to encapsulate the above with the OpenUCX standard

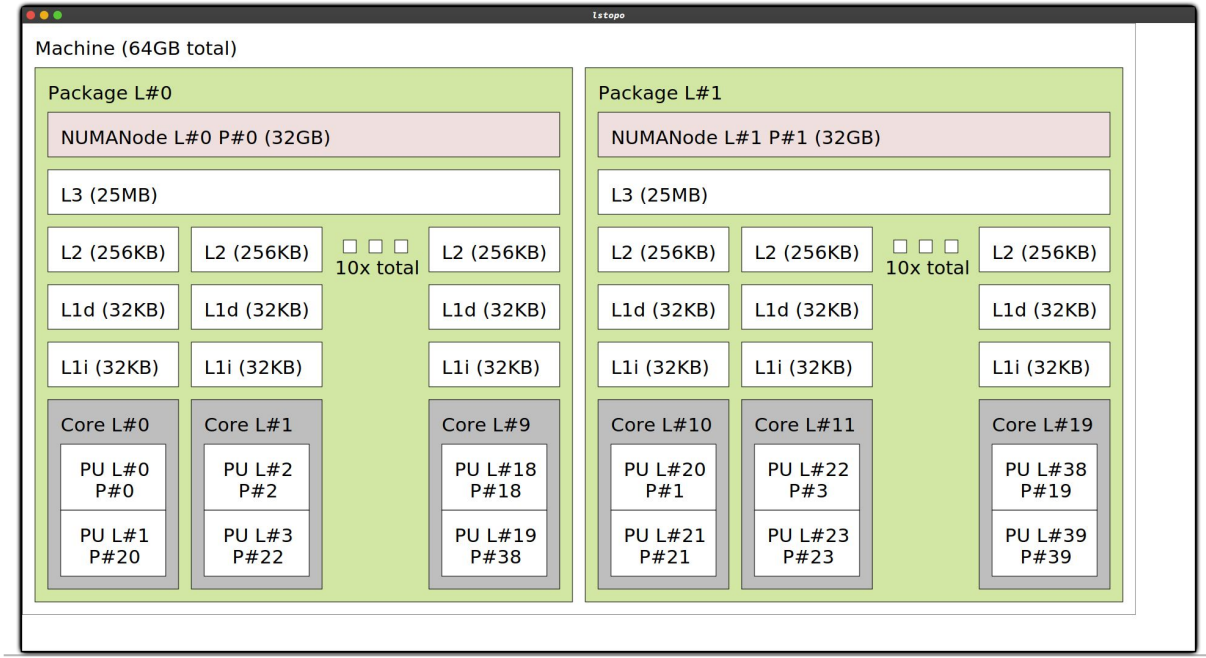


https://openucx.readthedocs.io/en/master/

Interlude - Getting to know your system

- With so many components, complexity increases.
- For robustness and efficiency, intimate architecture knowledge is essential.

```
> lstopo
```



`hwloc` library: Hardware locality
https://www.open-mpi.org/projects/hwloc

Features:

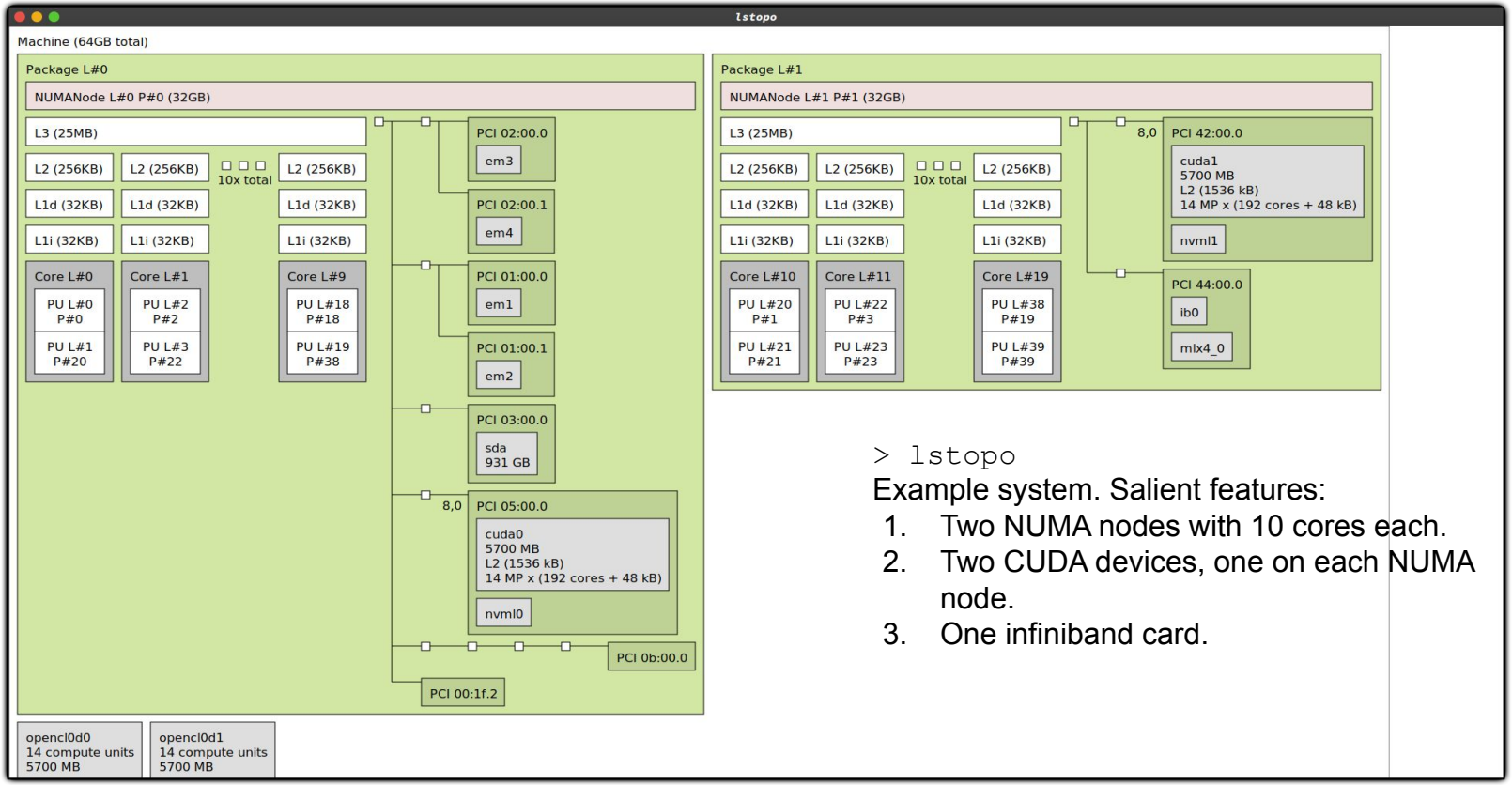
1. Detect hardware on the machine.
2. Bind to different objects.
3. Powerful and relatively simple API.
4. Bundled with most open-source MPI implementations.
5. Installed/ easy to install on many clusters.

```
> module load hwloc
```

```
> sudo apt-get install hwloc # Debian

> brew install hwloc # MacOS
```

Interlude - Getting to know your system



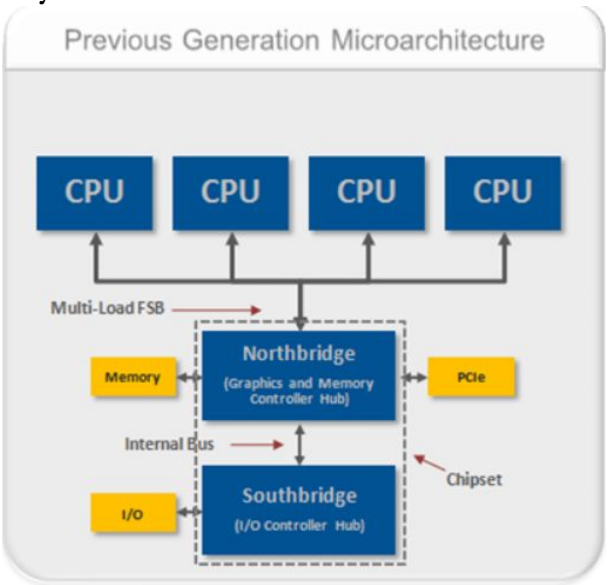

```
> lstopo
```

Example system. Salient features:

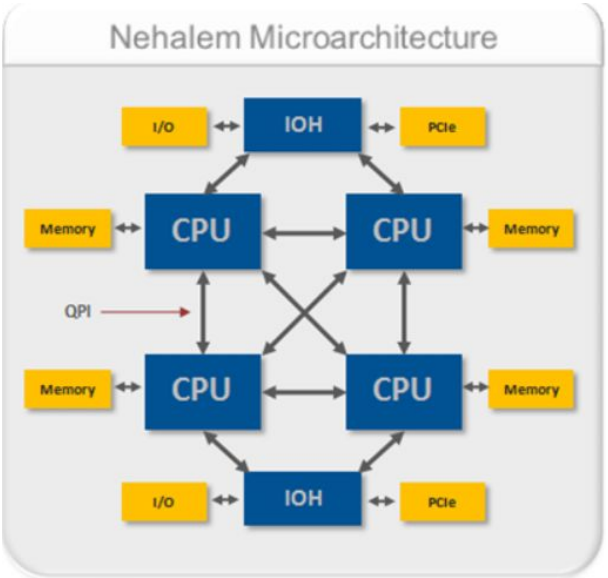
1. Two NUMA nodes with 10 cores each.
2. Two CUDA devices, one on each NUMA node.
3. One infiniband card.

Interlude - Getting to know your system

- Most of the newer processors are NUMA architectures.
- Memory distance is lesser, but access can be non-uniform.



UMA (Uniform Memory Access)
FSB(Front side Bus)



NUMA (Non-Uniform Memory Access)
with Intel QPI (Quick path interconnect)

Interlude - Getting to know your system

shell (Linux)

```
> numactl --hardware                                  # Check numa properties of your
>                                                     # machine
>
> numactl --show                                      # Show the current process
>                                                     # numa affinity settings
>
> man numactl                                         # Show the doc for numactl
>
> numactl --physcpubind=+0-4,8-12 exec arguments      # Run exec on cpus 0-4 and 8-12
>                                                     # of the current cpuset
>
> numactl --cpubind=0 --membind=0,1 exec              # Run exec on node 0 with memory
>                                                     # allocated on node 0 and 1
```

Message Passing Interface

- Message passing interface (MPI): A library specification for message specification.
- A standard API to create parallel applications.
- http://mpi-forum.org/, http://www.open-mpi.de/doc/v2.0/
- Bindings for C and Fortran.
- Set of functions to pass messages between processes.
- MPI is the standard that defines the behaviour of these functions.

History

1980's: Many comm libraries: LAM P4
1992: Agreement to develop a standard, MPI; financed by many: Cray, IBM ...
1994: 1st version: MPI-1
1998: 2nd version: MPI-2
2008: 3rd version: MPI-3
2020: 4th version: MPI-4

Current implementations include: OpenMPI, IntelMPI, MPICH2, MVAPICH2 ...

MPI Standard

MPI-1 standard

Basic communication functions: point to point, collective, datatypes, topology.

MPI-2 standard

Parallel I/O, RDMA, dynamic processes.

MPI-3 standard

Improved RDMA, Neighboring collectives, Fortran 2008 bindings, probing.

MPI-4 standard

Streaming messages, fault tolerance, Better support for Hybrid MPI + X models, Cancellation.

Message passing Interface

- Operate on local resources
- Each process is isolated. Interaction requires exchange of messages
- Exchanging messages takes more time than accessing local memory
- Messages can be *data/synchronization/instructions*
- Also works in a shared memory system, performance depends on implementation.
- Mainly based on three things: Send, Receive and Synchronize.

Message passing Interface - on GPUs

- MPI standard does not specify where the buffers reside → Can be on GPUs/CPUs.
- Therefore, the usual distributed codes using MPI to do distributed computing are still valid if the buffers are on GPUs ***AND*** the MPI implementation supports GPUs
- GPU-Aware MPI: A MPI implementation that can call MPI functions on buffers resident on the GPU without needing to transfer the data to the CPU.
- Examples: OpenMPI, MVAPICH2, CrayMPICH.
- Most new MPI versions support GPU-Awareness → Critical for performance on supercomputers.

Me

- M
- T
- G
- E
- M

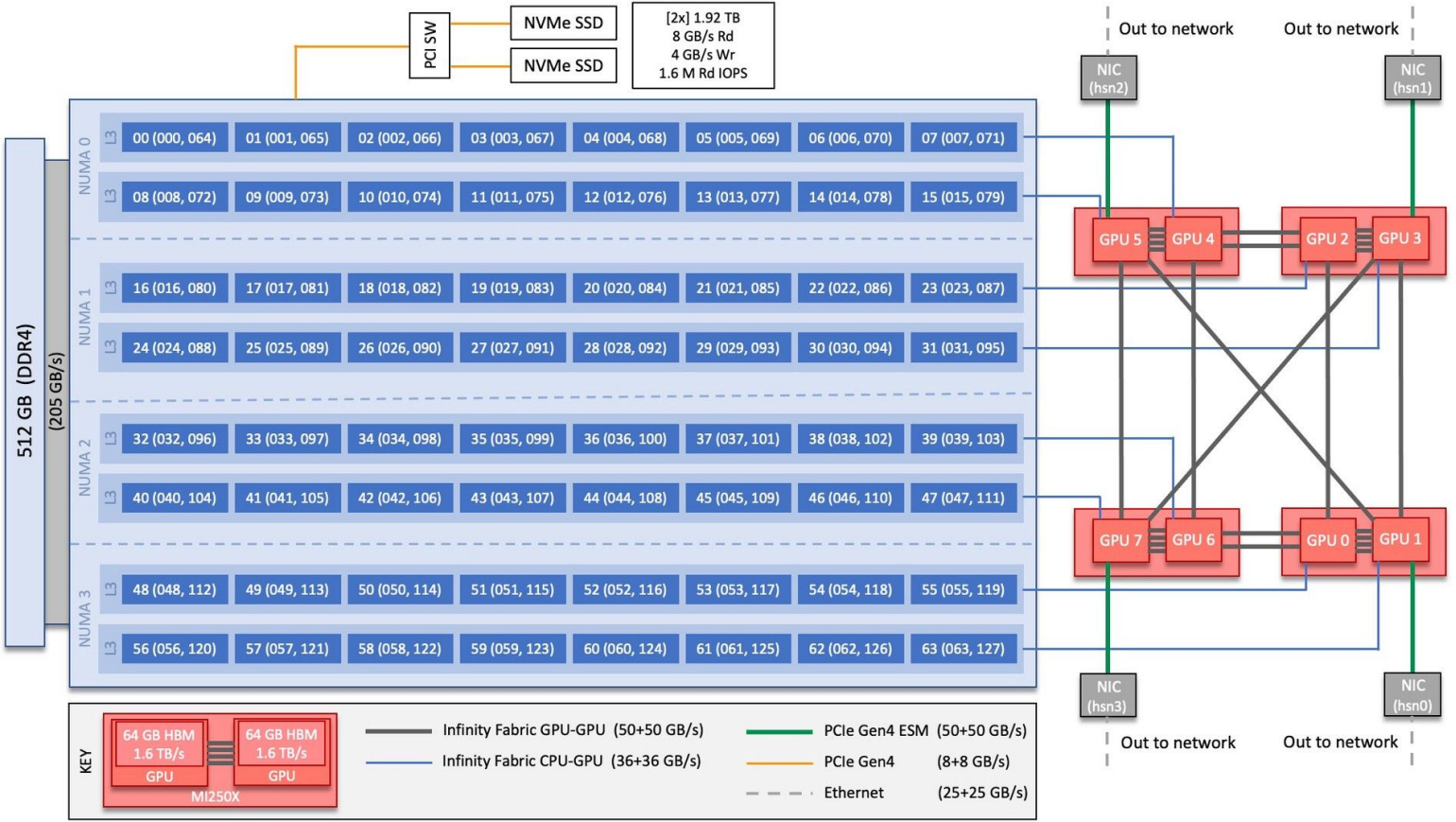
if

or

su

alid

nt



Me

- M
- T
- G
- E
- M

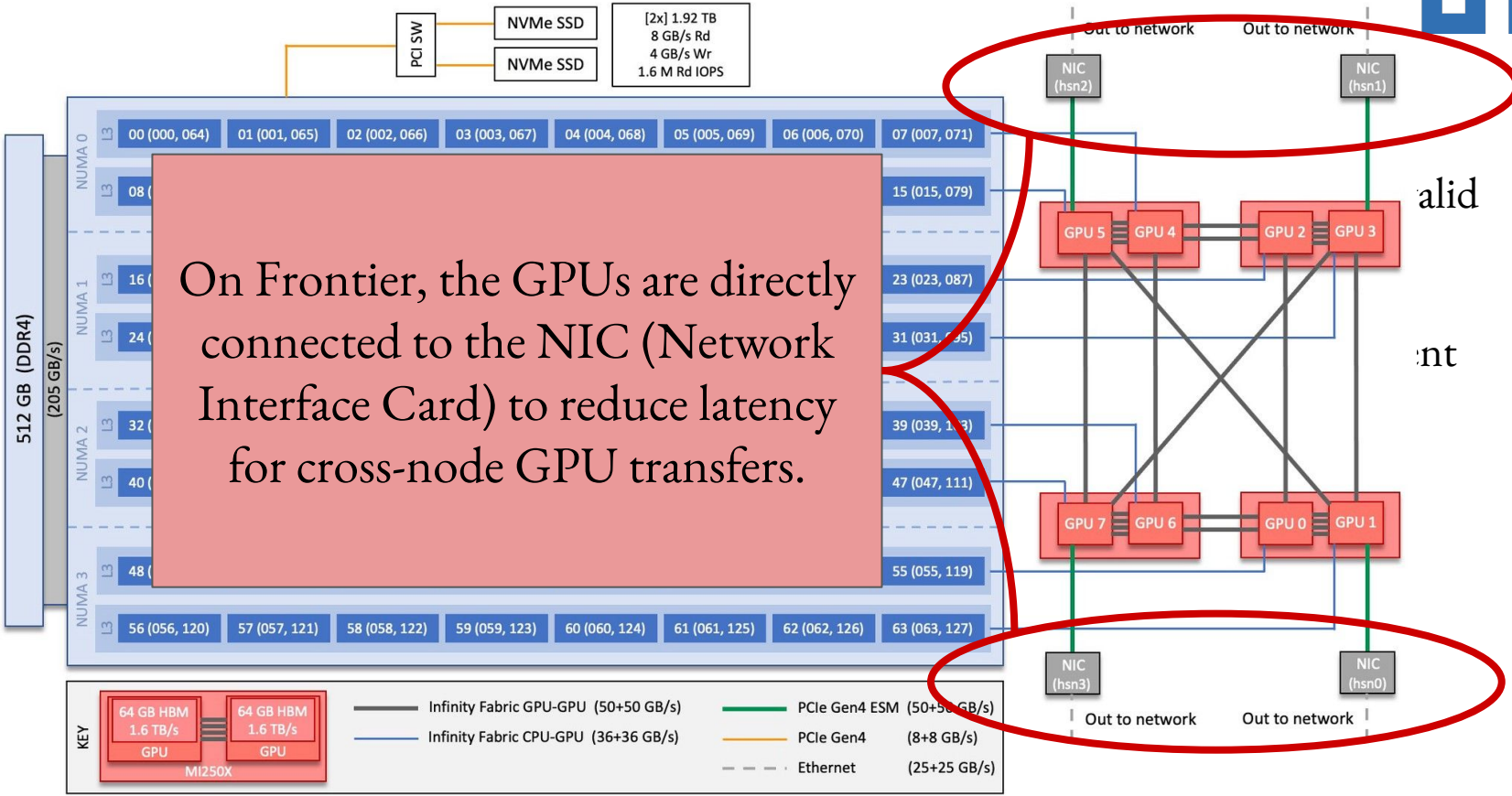
if

or

su

alid

nt



Using MPI

- Add the appropriate header file

C/C++

```
#include <mpi.h>
```

Using MPI

- Add the appropriate header file

C/C++

```
#include <mpi.h>
```

- Initialize and Finalize MPI: Always necessary! , Usually at the beginning and the end of the program!

C/C++

```
int MPI_Init(int* argc, char***argv)

int MPI_Finalize()
```

Using MPI

- Add the appropriate header file

C/C++

```
#include <mpi.h>
```

- Initialize and Finalize MPI: Always necessary! , Usually at the beginning and the end of the program!

C/C++

```
int MPI_Init(int* argc, char***argv)

int MPI_Finalize()
```

- Identify and group processes
- Communicate, when required!

Using MPI

- Compile the program with mpiX (Usually a convenience wrapper to the language compiler, gcc, icc, gfortran

C++

```
> mpicxx program.cpp -o program
```

C

```
> mpicc program.c -o program
```

Fortran

```
> mpif90 program.f90 -o program
```

Using MPI

- Compile the program with mpiX (Usually a convenience wrapper to the language compiler, gcc, icc, gfortran

C++

```
> mpicxx program.cpp -o program
```

- Run the program with mpiexec (See doc: https://www.open-mpi.org/doc/v3.1/man1)

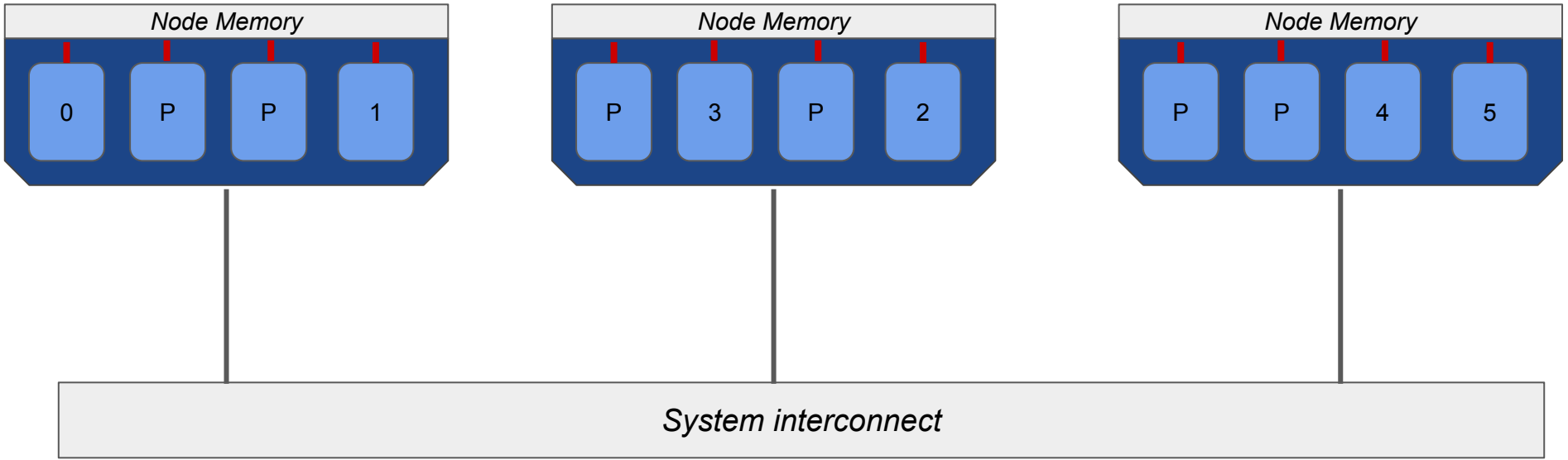
```
> mpiexec -n <number of processes> program1 [ : -n <np2> program2 . . .]
```

MPI ranks and communicators

- Consider a MPI program with 6 processes and 2 processes per node.

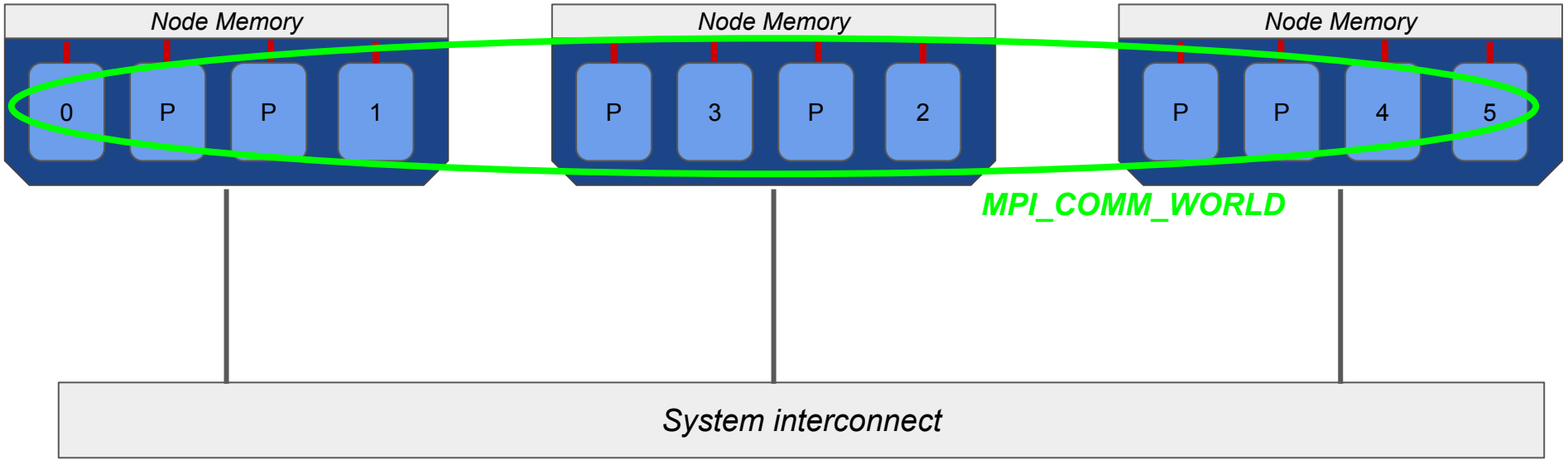
OpenMPI

```
> mpiexec -n 6 --map-by ppr:2:node program
```



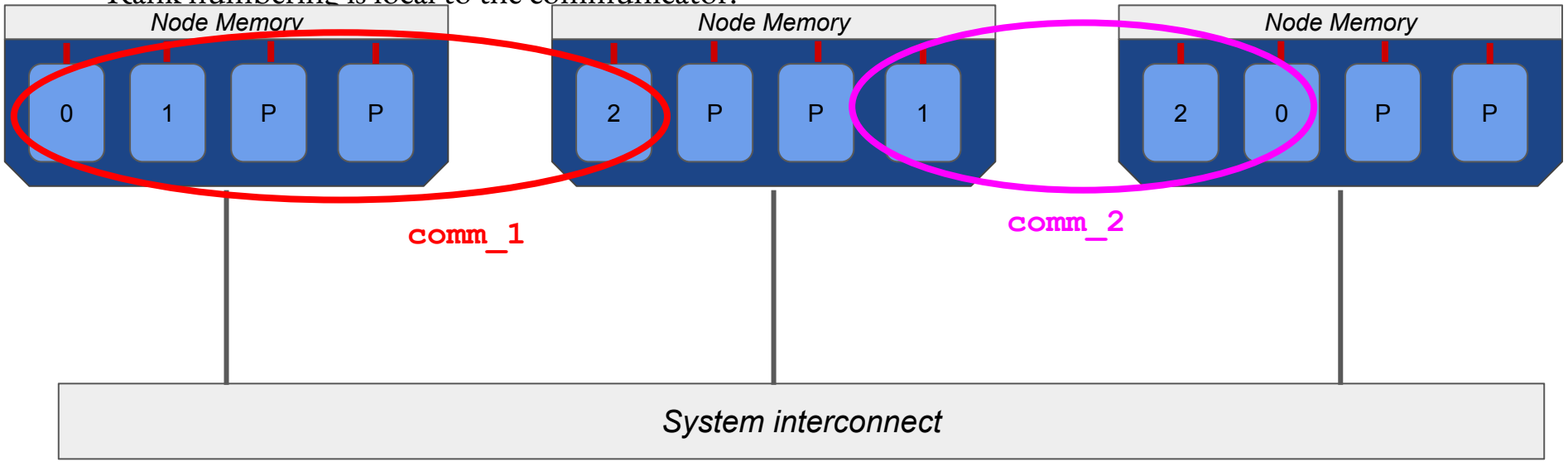
MPI ranks and communicators

- Each process running a program has a MPI rank and it belongs to a MPI communicator.
- MPI rank acts as an identification for the process in question.
- A communicator is a collection of the ranks.



MPI ranks and communicators

- Each process running a program has a MPI rank and it belongs to a MPI communicator
- MPI rank acts as an identification for the process in question
- A communicator is a collection of the ranks.
- Rank numbering is local to the communicator.



Groups and communicators

- Group is an ordered set of processors
  - Represented as an object within system memory
  - Accessible only through handle
  - Always associated with a communicator object

Groups and communicators

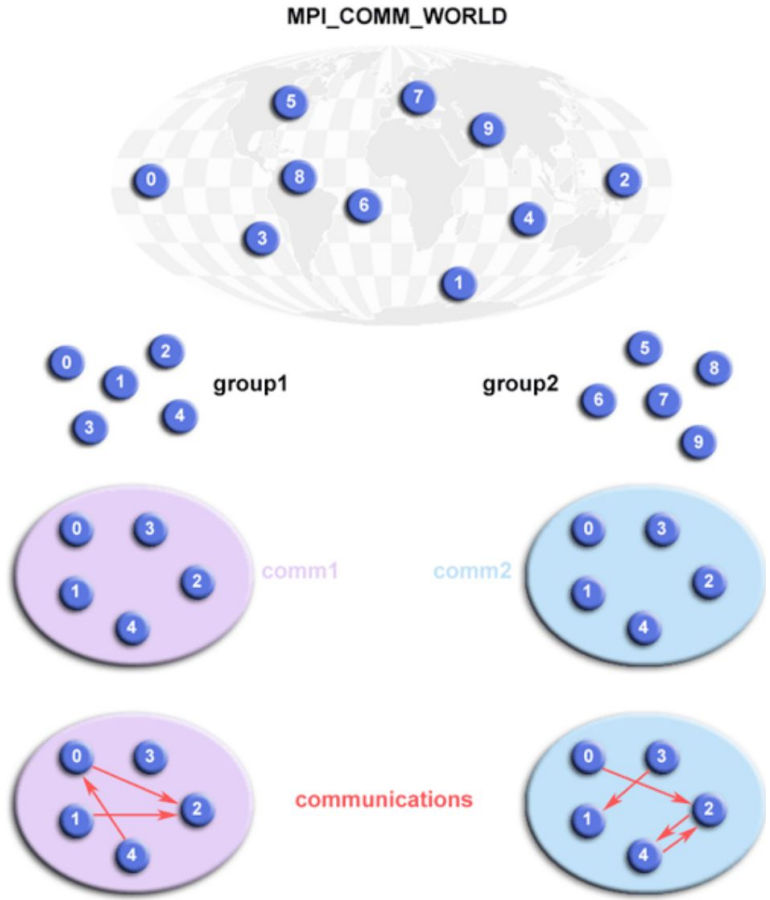
- Group is an ordered set of processors
  - Represented as an object within system memory
  - Accessible only through handle
  - Always associated with a communicator object
- Communicator is a group of processes that communicate with each other
  - All messaging routines require the specification of a communicator
  - Accessible only through handle
  - Communicator encompassing all processes is called MPI_COMM_WORLD

Groups and communicators

- Group is an ordered set of processors
  - Represented as an object within system memory
  - Accessible only through handle
  - Always associated with a communicator object
- Communicator is a group of processes that communicate with each other
  - All messaging routines require the specification of a communicator
  - Accessible only through handle
  - Communicator encompassing all processes is called MPI_COMM_WORLD
- Group routines are used to specify which processes should be used to construct a communicator

Groups and communicators

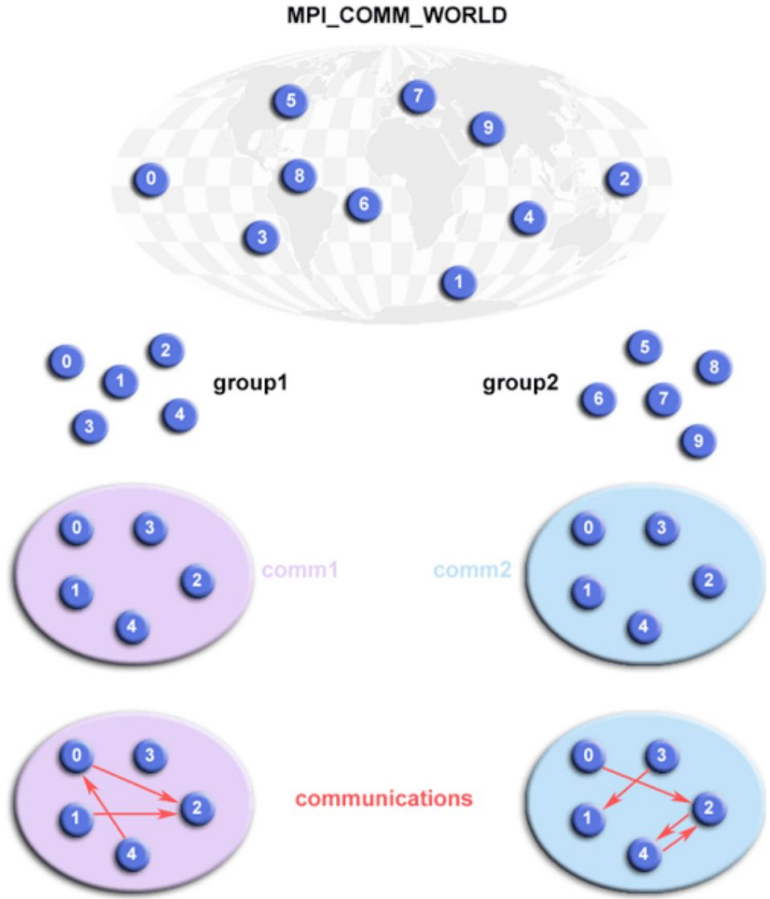
- Allows programmer to organize tasks into task groups
- Enable communication across tasks
- Implementing user defined virtual topology



Groups and communicators

- Allows programmer to organize tasks into task groups
- Enable communication across tasks
- Implementing user defined virtual topology

- Groups/comm can be created and destroyed during program execution
- Processes are not unique to a group/communicator and may have a unique rank within each group/comm



Defining a new communicator

- The hard way:
  - Get the Group associated the communicator with MPI_Comm_group(...)
  - Reorder the processes associated with the group with MPI_Group_incl(...)
  - Create the new communicator using the reordered group using MPI_Comm_create(...)

Defining a new communicator

- The hard way:
  - Get the Group associated the communicator with MPI_Comm_group(...)
  - Reorder the processes associated with the group with MPI_Group_incl(...)
  - Create the new communicator using the reordered group using MPI_Comm_create(...)
- The smart and easier way:
  - Use MPI_Comm_split(...)

C/C++

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
    MPI_Comm *newcomm)
```

color (INT) identifier for the group
key (INT) specifies the rank in a group

Defining a new communicator

- The smart and easier way:
  - Use MPI_Comm_split()

C/C++

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
    MPI_Comm *newcomm)
```

P0	P1
P2	P3
P4	P5

P0	P1
P2	P3
P4	P5

P0	P1
P2	P3
P4	P5

myRank	0	1	2	3	4	5
iRow	0	0	1	1	2	2
jCol	0	1	0	1	0	1

C/C++

```
irow = rank/mcol
jcol = mod(rank,mcol)

int MPI_Comm_split(MPI_COMM_WORLD, irow, rank, row_comm)
int MPI_Comm_split(MPI_COMM_WORLD, jcol, rank, col_comm)
```

Identifying your processes

- Getting the size of the communicator

C/C++

```
MPI_Comm_size(MPI_comm comm, int*size)
```

- Getting the rank of the process

C/C++

```
MPI_Comm_rank(MPI_Comm comm, int*rank)
```

- Getting the name of the node of the calling process

C/C++

```
MPI_Get_processor_name(char* name, int* len)
```

MPI intrinsic datatypes

- MPI has a "handle" to allow the user to refer to datatypes for a language.
- This allows for portability when used on different environments.

- As a general rule, the MPI datatype must match between corresponding send and recv.

MPI Datatype	*C datatype*
`MPI_CHAR, MPI_UNSIGNED_CHAR`	`signed char, unsigned char`
`MPI_SHORT, MPI_UNSIGNED_SHORT`	`short int, unsigned short int`
`MPI_INT, MPI_UNSIGNED_INT`	`signed int, unsigned int`
`MPI_LONG, MPI_UNSIGNED_LONG`	`signed long int, unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE, MPI_LONG_DOUBLE`	`double, long double`
`MPI_BYTE`	-
`MPI_PACKED`	-

Exchanging messages

Data is exchanged through buffers, with arrays with a certain known number of elements and of known data type.

- The type must be known and given to the MPI routine. This allows for portability between different environments.

Messages are identified by envelopes which contain its identifying information and needs to be specified correctly by both the receiver and sender.

BODY			*ENVELOPE*			
`buffer`	`count`	`datatype`	`source`	`destination`	`communicator`	`tag`

- Each process has a first in first out queue. If a process sends multiple messages to another process, they arrive in the same order.

MPI point to point

Sending and Receiving messages

C/C++

```
Rank i: int MPI_Send(<*buf, data buffer>,int count, <type,MPI_Type>,
<dest,j, receiving rank>, int tag, Comm )

Rank j: int MPI_Recv(<*buf, data buffer>,int count, <type,
MPI_Type>, <source,i, sending rank>, int tag, Comm )
```

- The buffers should be compatible:
  - Recv buffer should be large enough
  - Data type should be the same

buf	array of type to be sent or received.
count	(INT) number of elements of **buf** to be sent
type	(INT) MPI type of **buf** array
dest	(INT) rank to which the message is sent
source	(INT) rank of the sending process
tag	(INT) a number serving as message ID
comm	(MPI_Comm) communicator of sender and receiver
status	(MPI_Status) array containing the communication status

Sending and Receiving messages

C/C++

```
Rank i: int MPI_Send(<*buf, data buffer>,int count, <type,MPI_Type>,
<dest,j, receiving rank>, int tag, Comm )

Rank j: int MPI_Recv(<*buf, data buffer>,int count, <type,
MPI_Type>, <source,i, sending rank>, int tag, Comm )
```

- The buffers should be compatible:
  - Recv buffer should be large enough
  - Data type should be the same

buf	array of type to be sent or received.	body
count	(INT) number of elements of **buf** to be sent	
type	(INT) MPI type of **buf** array	
dest	(INT) rank to which the message is sent	envelope
source	(INT) rank of the sending process	
tag	(INT) a number serving as message ID	
comm	(MPI_Comm) communicator of sender and receiver	
status	(MPI_Status) array containing the communication status	

Deadlocks

- Deadlocks happen when one function is waiting another while the other function is waiting for the first function.

C/C++

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if( myrank == 0 ) {
        MPI_Recv( b, 10, MPI_DOUBLE, 1, 2, MPI_COMM_WORLD, &status );
        MPI_Send( a, 10, MPI_DOUBLE, 1, 3, MPI_COMM_WORLD );
    }
    else if( myrank == 1 ) {
        MPI_Recv( b, 10, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD, &status );
        MPI_Send( a, 10, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD );
    }
```

Deadlocks

- Is there a deadlock here ?

C/C++

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if( myrank == 0 ) {
        MPI_Send( a, 10, MPI_DOUBLE, 1, 3, MPI_COMM_WORLD );
        MPI_Recv( b, 10, MPI_DOUBLE, 1, 2, MPI_COMM_WORLD, &status );
    }
    else if( myrank == 1 ) {
        MPI_Send( a, 10, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD ) ;
        MPI_Recv( b, 10, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD, &status );
    }
```

Deadlocks

- Yes. And it is resolved by

C/C++

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if( myrank == 0 ) {
        MPI_Send( a, 10, MPI_DOUBLE, 1, 3, MPI_COMM_WORLD );
        MPI_Recv( b, 10, MPI_DOUBLE, 1, 2, MPI_COMM_WORLD, &status );
    }
    else if( myrank == 1 ) {
        MPI_Recv( b, 10, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD, &status );
        MPI_Send( a, 10, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD ) ;
    }
```

Deadlocks

- Question: Try this code out and tell me if you actually observe a deadlock. If not, can you try to figure out why it does not ? *Hint: Try different sized messages.*

C/C++

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if( myrank == 0 ) {
        MPI_Send( a, 10, MPI_DOUBLE, 1, 3, MPI_COMM_WORLD );
        MPI_Recv( b, 10, MPI_DOUBLE, 1, 2, MPI_COMM_WORLD, &status );
    }
    else if( myrank == 1 ) {
        MPI_Send( a, 10, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD ) ;
        MPI_Recv( b, 10, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD, &status );
    }
```

Deadlocks

- Can be eliminated by:

  -- Careful attention to the buffers being used and the matching of the send and receive calls.

  -- Use *non-blocking send and receive.*

  -- Use MPI_Sendrecv().

Sending and Receiving messages: non-blocking

Blocking communication

The buffer can be reused after the function has returned.

Sending and Receiving messages: non-blocking

Blocking communication

The buffer can be reused after the function has returned.

Non-Blocking communication

The buffer can be reused only after the wait function has returned. The buffer cannot be read or written into until then.

Sending and Receiving messages: non-blocking

Blocking communication

The buffer can be reused after the function has returned.

Non-Blocking communication

The buffer can be reused only after the wait function has returned. The buffer cannot be read or written into until then.

Both above do not mean that ...

Communication has been completed when the function has returned.

Blocking and non-blocking

C/C++

```
int buf[N];
MPI_Send(buf,
MPI_INT,N,...);
//buffer, buf can be re-used
buf [k] = 42;
```

C/C++

```
int buf[N];
MPI_ISend(buf, MPI_INT,N,...);
//buffer, buf cannot be reused,
//undefined behaviour

MPI_Wait(...);
buf [k] = 42;
```

Blocking and non-blocking

C/C++

```
int buf[N];
MPI_Send(buf,
MPI_INT,N,...);
//buffer, buf can be re-used
buf [k] = 42;
```

Blocking

C/C++

```
int buf[N];
MPI_ISend(buf, MPI_INT,N,...);
//buffer, buf cannot be re-used,
//undefined behaviour

MPI_Wait(...);
buf [k] = 42;
```

Non-Blocking

Blocking and non-blocking

C/C++

```
int buf[N];
MPI_Send(buf,
MPI_INT,N,...);
//buffer, buf can be re-used
buf [k] = 42;
```

Blocking

C/C++

```
int buf[N];
MPI_ISend(buf, MPI_INT,N,...);
//buffer, buf cannot be re-used,
//undefined behaviour

MPI_Wait(...);
buf [k] = 42;
```

Non-Blocking

Ranks involved in the communication do not have an idea on whether the communication has been completed when the MPI_Send/MPI_Isend (standard-mode) function returns

Non-blocking Sending and Receiving messages

C/C++

```
Rank i: int MPI_Isend(<*buf, data buffer>,int count, <type,MPI_Type>,
<dest,j, receiving rank>, int tag, Comm, *req )

Rank j: int MPI_Irecv(<*buf, data buffer>,int count, <type,
MPI_Type>, <source,i, sending rank>, int tag, Comm, *req )
```

- Request is the opaque object that matches the rank that calls it to the comm operation that ends it.

buf	array of type to be sent or received.	body
count	(INT) number of elements of **buf** to be sent	
type	(INT) MPI type of **buf** array	
dest	(INT) rank to which the message is sent	envelope
source	(INT) rank of the sending process	
tag	(INT) a number serving as message ID	
req	(MPI_Request) identifier for the communications	
comm	(MPI_Comm) communicator of sender and receiver	
status	(MPI_Status) array containing the communication status	

Waiting and testing for completion

C/C++

```
int MPI_Wait(*req, *status )
int MPI_Waitall(int count, *array_req[], *array_status[] )
```

C/C++

```
int MPI_Test(*req, *flag, *status )
int MPI_Testall(int count, *array_req[], *flag, *array_status[] )
```

count	(INT) number of elements in the arrays
req	(MPI_Request) identifier for the communications
flag	(INT) true if req has completed, false otherwise
status	(MPI_Status) array containing the communication status

The same modes as in blocking can be used: standard, synchronous, buffered and ready

Combined Send and Receive

C/C++

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype
sendtype,
    int dest, int sendtag, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int source, int recvtag,
    MPI_Comm comm, MPI_Status *status)
```

- Useful for executing shift operations across a chain of processes.
- Will block until the send buffer is free for reuse.

...buf	array of type to be sent or received.
...count	(INT) number of elements of **buf** to be sent
type	(INT) MPI type of **buf** array
dest	(INT) rank to which the message is sent
source	(INT) rank of the sending process
...tag	(INT) a number serving as message ID
comm	(MPI_Comm) communicator of sender and receiver
status	(MPI_Status) array containing the communication status

MPI Collective operations

- One to many, many to one or many to many constitute collective operations
- All processes call the routine, one is root
- No tags
- Examples:
  - Barrier Synchronization
  - Broadcast
  - Gather/Scatter
  - All to all
  - Reduction

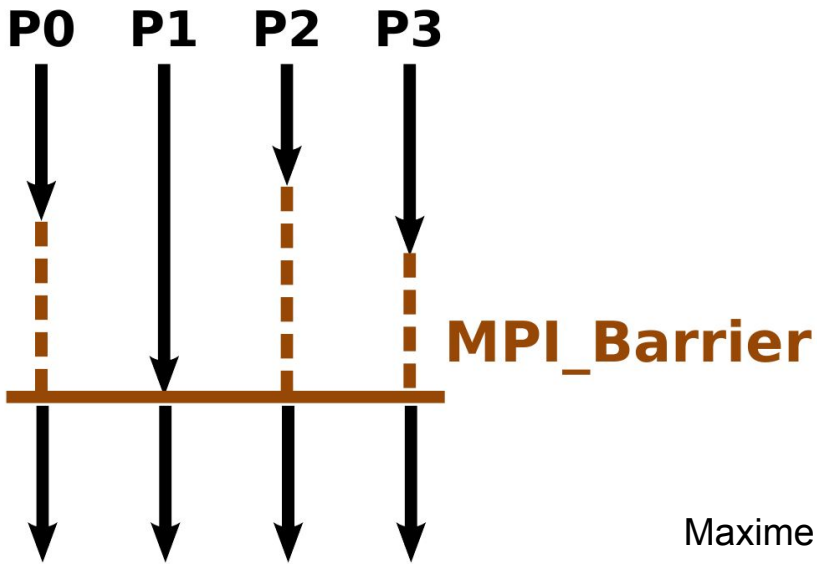
The MPI library implementation decides the most efficient algorithm for that platform (mainly based on network topology) and application (message size) .

Barrier

C/C++

```
int MPI_Barrier(comm)
```

- Wait until all processes belonging to the communicator comm have reached the barrier.



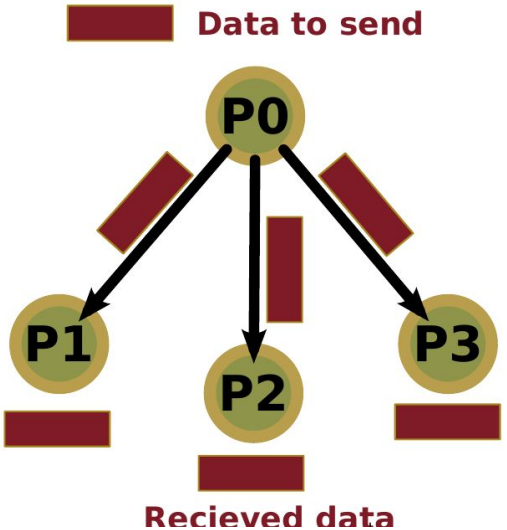
Maxime Martinasso

Broadcast

C/C++

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

- One to all communication: Same data sent to all processes in comm from root process.



Maxime Martinasso

Scatter

C/C++

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

- One to all communication: Different data sent to all processes in comm from root process.

sndcount	(INT) number of elements to be sent to each separate process and not the size of the sndbuf
rcvcount	(INT) number of elements in the recvbuf

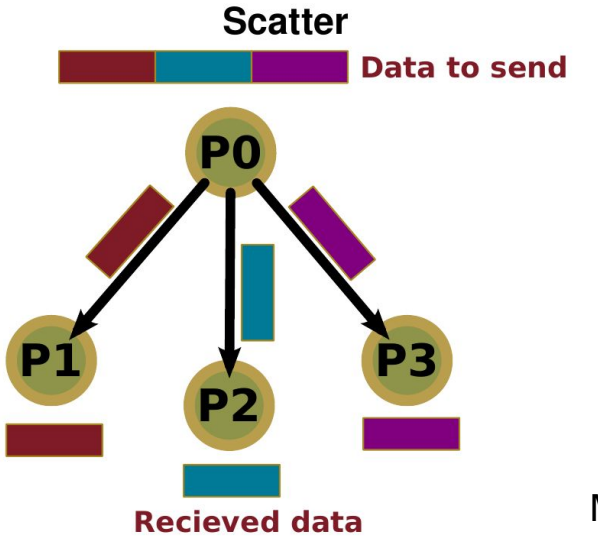
Scatter

C/C++

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

- One to all communication: Different data sent to all processes in comm from root process.

The number of elements sent to each process is the same. The `sendbuf` is divided equally among the processes



Maxime Martinasso

Gather

C/C++

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

- All to one communication: Different data sent from all processes in comm to root process.

sndcount	(INT) number of elements in the sndbuf
rcvcount	(INT) number of elements collected from each process, no the size of the recvbuf

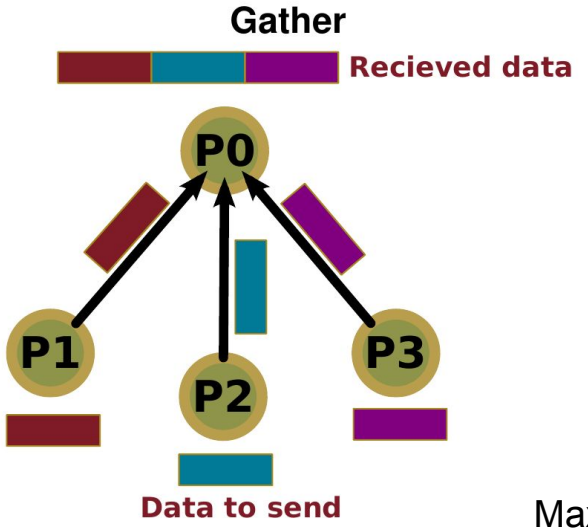
Gather

C/C++

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

- All to one communication: Different data sent from all processes in comm to root process.

The number of elements received from each process is the same. The `recvbuf` is divided equally among the processes



Maxime Martinasso

Global collective operations

C/C++

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

- All to all collective communication: Different data gathered from all processes in comm but stored in all.

C/C++

```
int MPI_Allgatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
    const int displs[], MPI_Datatype recvtype, MPI_Comm comm))
```

- All to all collective communication: Different sized data gathered from all processes in comm but stored in all.

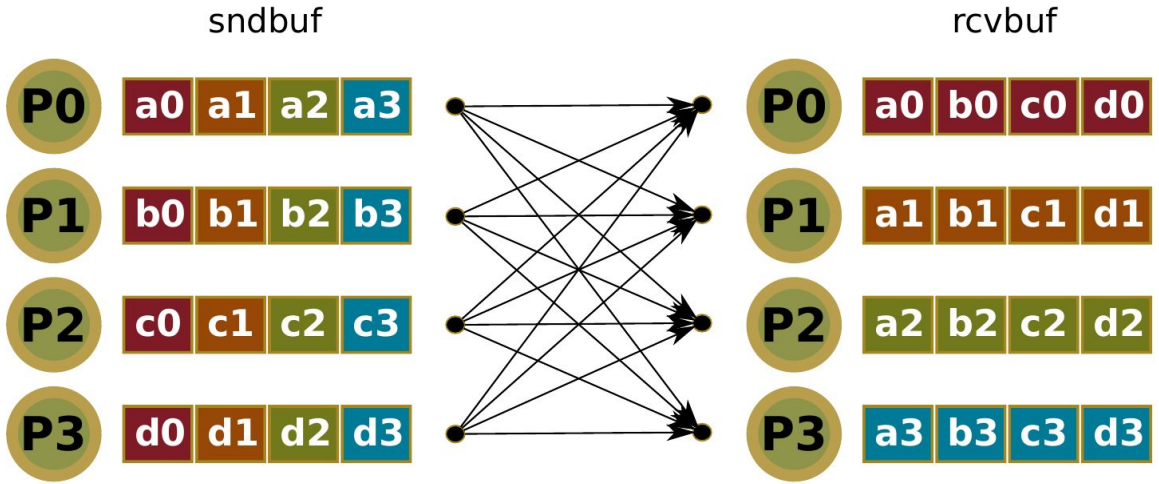
Note that the root process need not be specified.

Global exchange: All to all

C/C++

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

- All processes exchange the same amount of data.
- Useful for operations such as transpose



Maxime Martinasso

Global exchange: All to all

C/C++

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

- All processes exchange the same amount of data.
- Useful for operations such as transpose

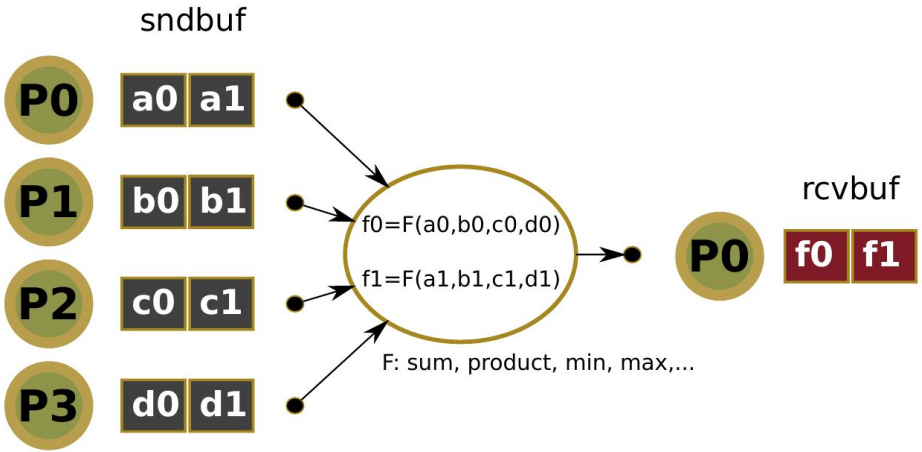
- MPI_Alltoallv for varied amount of data exchanges and non-contiguous buffers
- MPI_Alltoallw: Allows different datatypes as well.

Reduction

C/C++

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```

- Op (handle) parallel operation to perform



Maxime Martinasso

Reduction

C/C++

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```

- Op (handle) parallel operation to perform

MPI op	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Other collective routines

C/C++

```
MPI_Igather(), MPI_Igatherv(),MPI_Iscatter(),MPI_Iscatterv(), MPI_Ireduce(
),
 MPI_Ialltoall(), MPI_Ibarrier()
```

C/C++

```
MPI_Neighbor_gather, MPI_Neighbor_alltoall
```

- Neighbor operations based on topology

C/C++

```
MPI_Scan, MPI_Exscan
```

- Partial reduction

Summary

- Distributed computing basics and programming models
- Message Passing interface:
  - API, functionality and capabilities
  - Communicators and groups
  - Point to point communication
  - Collective communication

Next lecture

- Multi-GPU computing part 2: NVSHMEM