LECTURE 11

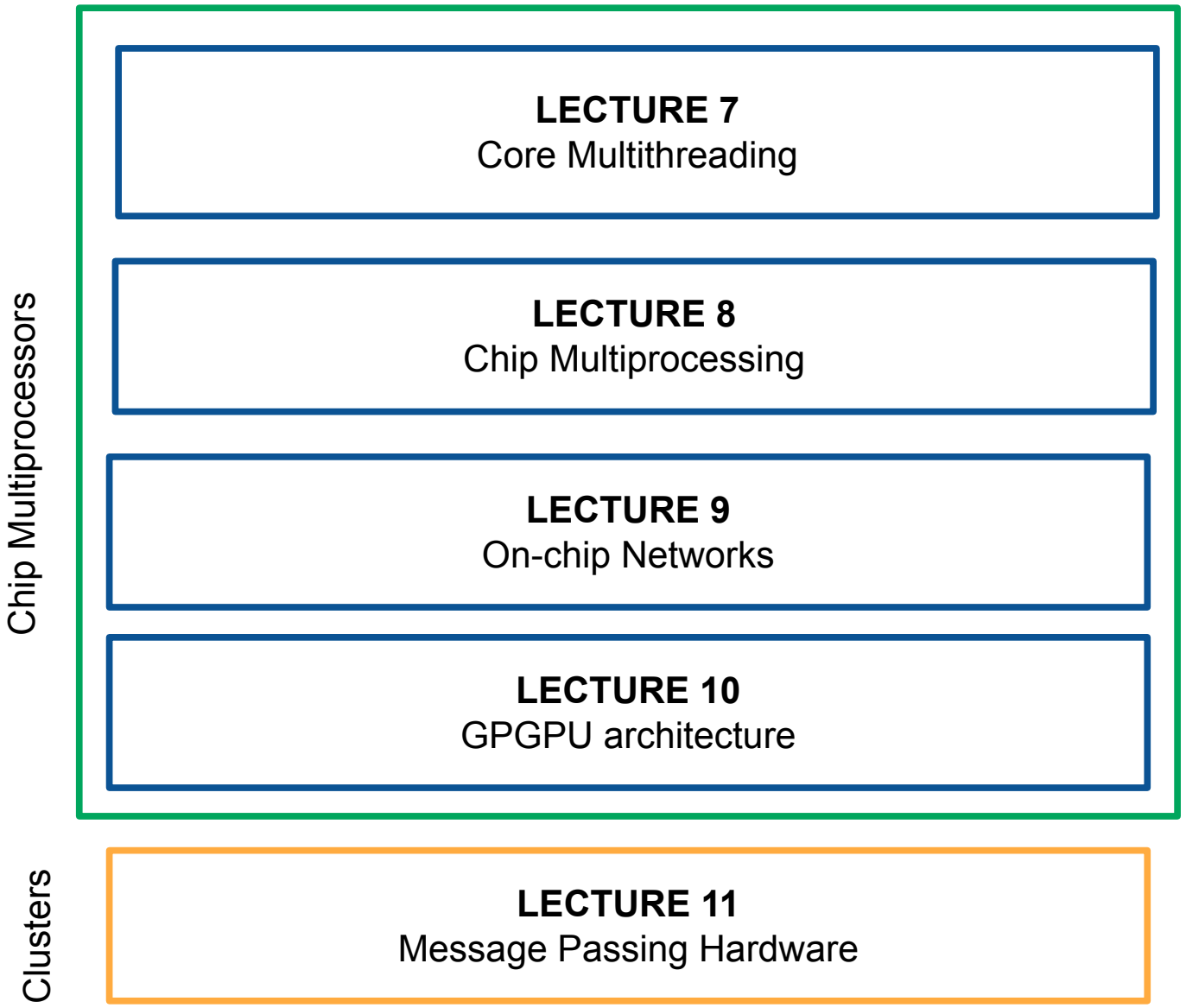
Message Passing Hardware

Miquel Pericàs
EDA284/DIT361 - VT 2020

What's cooking

1. **Lectures**
   - Today (9h-12h): **GPGPU (part 2)** + **Message Passing Hardware**
   - Tomorrow Feb 26th (8h-10h) **Synchronization**
2. **Lab session**
   - Friday (8h-12h @ ED3507), GEM5 + Vector
3. **Project Status**
   - Tomorrow: deadline for first round of peer feedback

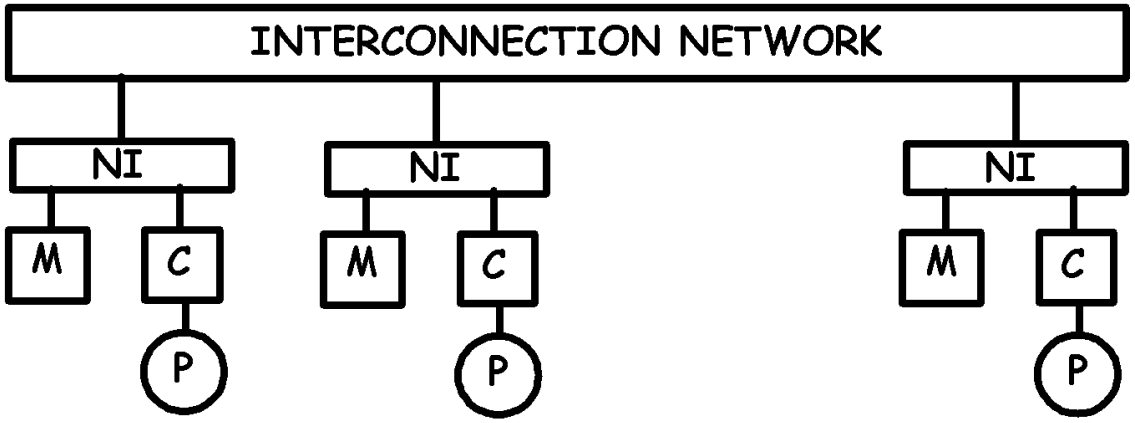
Lectures 7 - 11 Overview



Lecture 11: Outline

- **Message Passing Programming**
  - Synchronous / Asynchronous
- **Message Passing Hardware**
  - CPU based, DMA & Message Processors
- **Example: Infiniband**
  - Infiniband Architecture
  - Programming API (IB Verbs vs Sockets)

Message-Passing: Multicomputers



- Processing nodes interconnected by a network
- Communication carried out by message exchanges
- Scales well
- Hardware is inexpensive
- Software is complex, mostly based on MPI (Message Passing Interface)
  - This programming style is called “Message Passing”

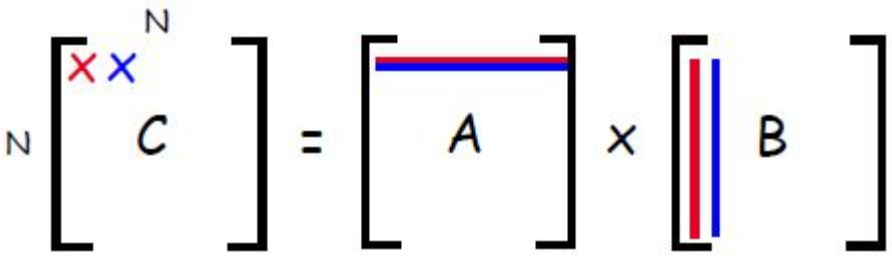
- Nowadays, processing nodes are often CMPs. Software becomes even more complex: “MPI + X”, where X is OpenMP, pthreads, TBB, etc..

EXAMPLE: MATRIX MULTIPLY + SUM

sequential program:

```
1          sum = 0;
2          for (i=0;i<N;i++)
3                for (j=0;j<N;j++){
4                     C[i,j] = 0;
5                     for (k=0;k<N;k++)
6                          C[i,j] = C[i,j] + A[i,k]*B[k,j];
7                     sum += C[i,j];
8                }
```

- multiply matrices A[N,N] by B[N,N] and store result in C[N,N]
- add all elements of C[,] and store in variable 'sum'



How to parallelize this on a message passing computer?

Example: Matrix Multiply + Sum

MESSAGE-PASSING PROGRAM

```
myN = N/nproc;
if(pid == 0){
  for(i=1; i<nproc; i++){
    k=i*N/nproc;
    SEND(&A[k][0], myN*N*sizeof(float),i, IN1);
    SEND(&B[0][0], N*N*sizeof(float),  i, IN2);
} else {
    RECV(&A[0][0], myN*N*sizeof(float),0,IN1);
    RECV(&B[0][0], N*N*sizeof(float),  0,IN2);
}
mysum = 0;
for (i=0, i<myN, i++)
  for (j=0, j<N, j++)
    C[i,j] = 0;
    for (k=0,k<N, k++)
      C[i,j] = C[i,j] + A[i,k]*B [k,j]:
  mysum += C[i,j]:
}
```

```
if (pid == 0){
  sum = mysum;
  for(i=1;i<nproc;i++){
    RECV(&mysum, sizeof(float),i,SUM);
    sum +=mysum;
  }
  for(i=1; i<nproc;i++){
    k=i*N/nproc;
    RECV(&C[k][0], myN*N*sizeof(float),i,RES);
  }
} else{
  SEND(&mysum, sizeof (float),0, SUM);
  SEND(&C[0][0], myN*N*sizeof(float),0,RES),
}
```

Assume matrices A and B initially in one computational node, the master node (PID=0)
Strategy: (1) Copy full matrix 'B' and rows of matrix 'A' to compute nodes, (2) accumulate sum, and (3) copy back rows of matrix 'C'

Synchronous Message-Passing

- **CODE FOR THREAD T1:**

```
A = 10;
SEND(&A,sizeof(A),T2,SEND_A);
A = A+1;
RECV(&C,sizeof(C),T2,SEND_B);
printf(C);
```

CODE FOR THREAD T2:

```
     B = 5;
RECV(&B,sizeof(B),T1,SEND_A);
     B=B+1;
SEND(&B,sizeof(B),T1,SEND_B);
```

- **EACH SEND/RECV HAS 4 OPERANDS:**
  - Starting address in memory
  - Size of message
  - Destination/Source thread id
  - Tag connecting sends and receives

Synchronous Message-Passing

- **CODE FOR THREAD T1:**

```
A = 10;
  SEND(&A,sizeof(A),T2,SEND_A);
  A = A+1;
  RECV(&C,sizeof(C),T2,SEND_B);
  printf(C);
```

10 →

← 11

CODE FOR THREAD T2:

```
B = 5;
RECV(&B,sizeof(B),T1,SEND_A);
B=B+1;
SEND(&B,sizeof(B),T1,SEND_B);
```

- **In synchronous M-P sender blocks until `recv` is started and receiver blocks until first message bytes are sent**
  - Communication implies Synchronization
  - Note: this is usually much more than waiting for message propagation

- **Question: what is the value printed under synchronous M-P?**
  - Value 10 is received in B by T2; B is incremented by 1
  - Then the new value of B (11) Is sent and received by T1 Into C
  - And Thread 1 Prints “11”

Synchronous Message-Passing

ADVANTAGE:

- communication enforces synchronization
- simpler to reason about the outcome of a program

DISADVANTAGES:

- prone to deadlock
- blocks threads (no overlap of communication with computation)

DEADLOCK Example

Code for thread T1:

```
A = 10;
SEND(&A,sizeof(A),T2,SEND_A);
RECV(&C,sizeof(C),T2,SEND_B);
```

Code for thread T2:

```
B = 5;
SEND(&B,sizeof(B),T1,SEND_B);
RECV(&D,sizeof(D),T1,SEND_A);
```

To eliminate the deadlock: swap the send/recv pair in T2 or employ asynchronous message-passing

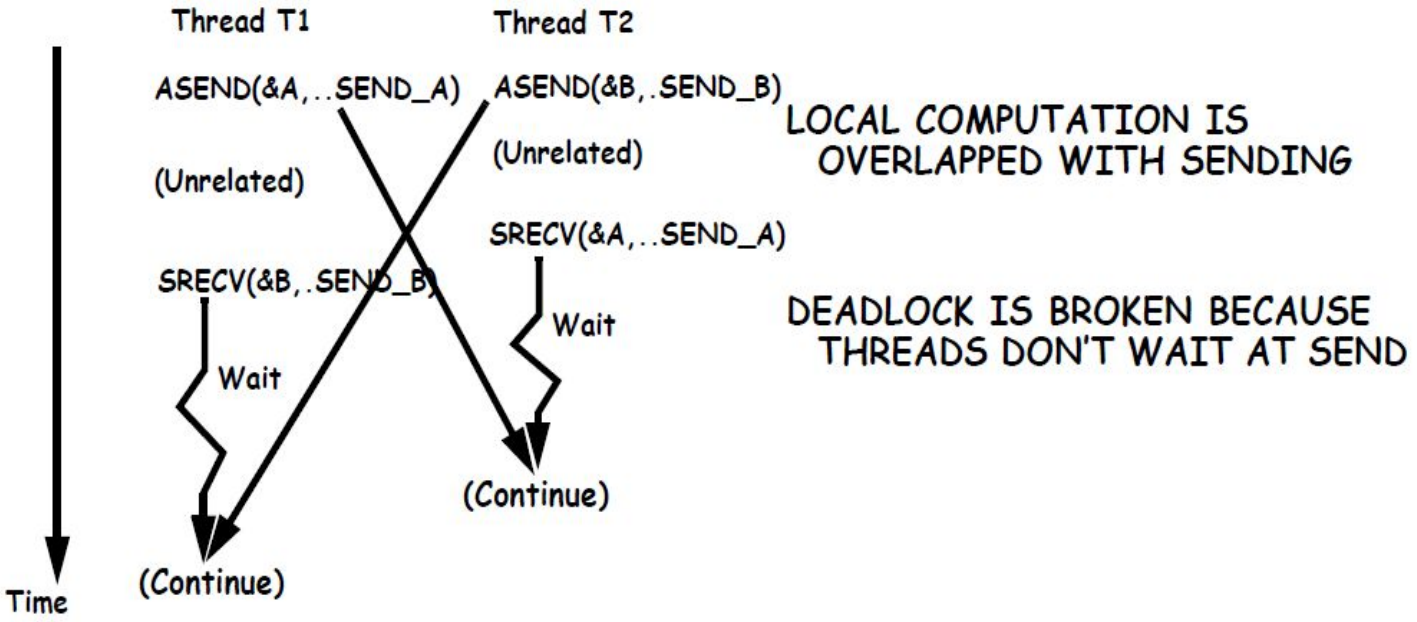
Asynchronous Message-Passing

CODE FOR THREAD T1:

```
A = 10;
ASEND(&A,sizeof(A),T2,SEND_A);
       <Unrelated computation;>
SRECV(&B,sizeof(B),T2,SEND_B);
```

CODE FOR THREAD T2:

```
B = 5;
ASEND(&B,sizeof(B),T1,SEND_B);
       <Unrelated computation;>
SRECV(&A,sizeof(B),T1,SEND_A);
```

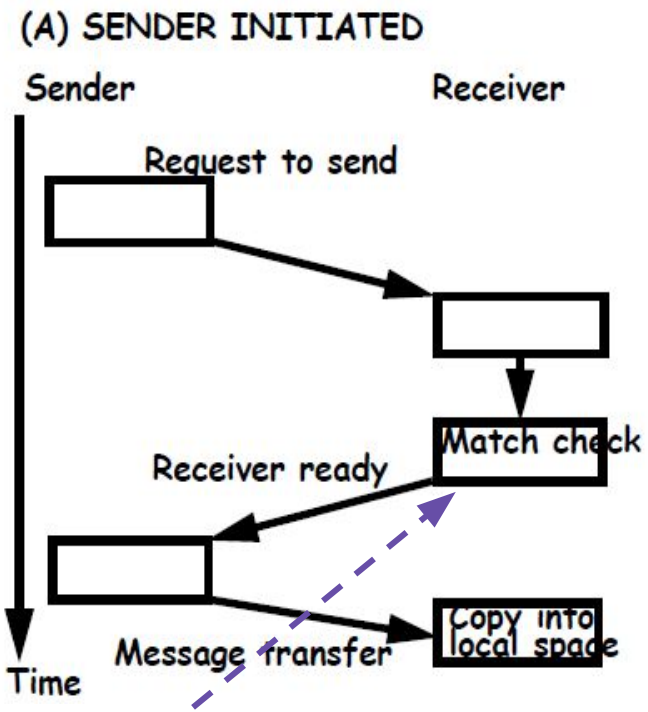


- **Blocking vs Non-blocking asynchronous message passing**
  - BLOCKING: resume only when area of memory has been buffered (`send`) and when message has been copied to user space (`recv`)
  - NON-BLOCKING: resume early, unsafe to reuse buffer space. Probe functions can be used to check whether data has been copied (e.g. `MPI_Wait()`)

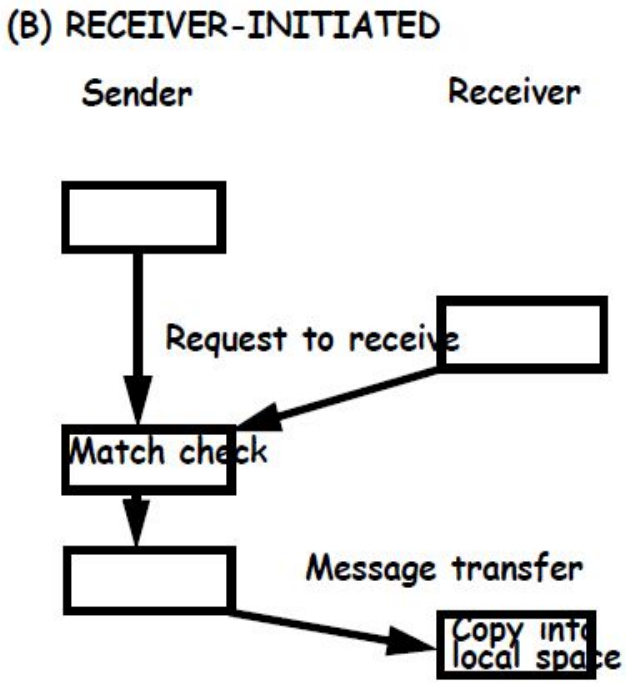
Message-Passing Protocols

Synchronous M-P

Three-phase protocol



Two-phase protocol



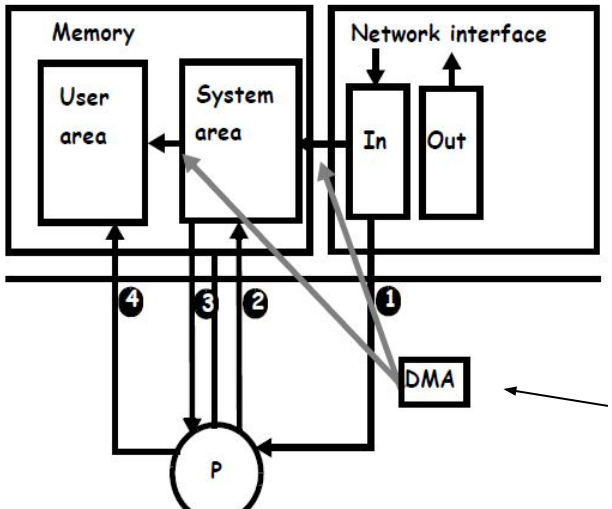
- **Case (A)**: Match table keeps track of the status of all `RECVs` previously executed with the receiver process identity and the tag.
- **Case (B)**: Match table at the sender. Keeps track of the status of all `SENDs`

HW support for message passing protocols

- **General interconnect networks provide primitive network transactions to implement M-P protocols. Additional HW targets:**
  - reduce message transfer latency, particularly important for synchronous M-P
    - (also benefits asynchronous M-P, but to a smaller degree)
  - offload computation processors from network tasks

- **Data must be copied from/to memory to/from network interface (NI)**
  - without HW support: sender copies data to system memory, and then from system memory to network interface + vice-versa. Processor may not keep up with injection rate!
  - DMA (Direct Memory Access) can speed up message transfers and offload the processor
    - DMA engine performs the copies in the background, offloading processor
    - DMA is programmed by the processor; specify start address and size

Processor-executed
RECV( )

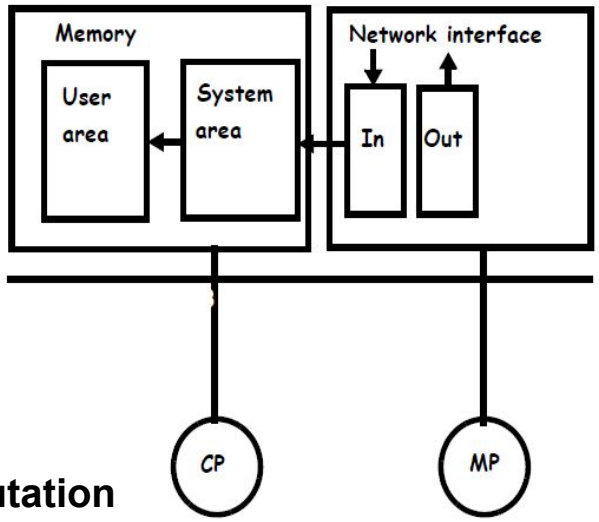


DMA to offload processor

HW support for message passing protocols

- **Support for "user-level" vs "system-level" messages**
  - Source/destination in user memory vs system memory
  - Basic message passing systems drive DMA engine from O/S
    - This is needed for protection between users
    - Message is first copied into system space and then into user space (receive)
    - Message is copied from user space to system space (send)
  - Optimization: Tag user-level messages so that they are picked up and delivered directly in user memory space (called "**zero-copy**")

- **Beyond DMA: Dedicated message processors**
  - Use a special processor to process messages on both ends + other higher-level functions (matching, packet forming, sending acks etc)
  - Relieves the compute O/S from doing it
  - Preferred approach for HPC interconnects
- Potentially, a core or thread in a CMP can also perform this function



What are the capabilities of HPC Networks?

- Intelligent Network Interface Cards
- Support entire protocol processing completely in hardware (hardware protocol offload engines)
- Provide a rich communication interface to applications
  - ***User-level communication capability***
  - Gets rid of intermediate data buffering requirements
- No software signaling between communication layers
  - All layers are implemented on a ***dedicated*** hardware unit, and not on a ***shared*** host CPU

Example: Infiniband

P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan and J. M. Dennis, "An Integrated Tutorial on InfiniBand, Verbs, and MPI," in IEEE Communications Surveys & Tutorials, vol. 19, no. 4, pp. 2894-2926, Fourth quarter 2017. doi: 10.1109/COMST.2017.2746083

Why InfiniBand (IB)?

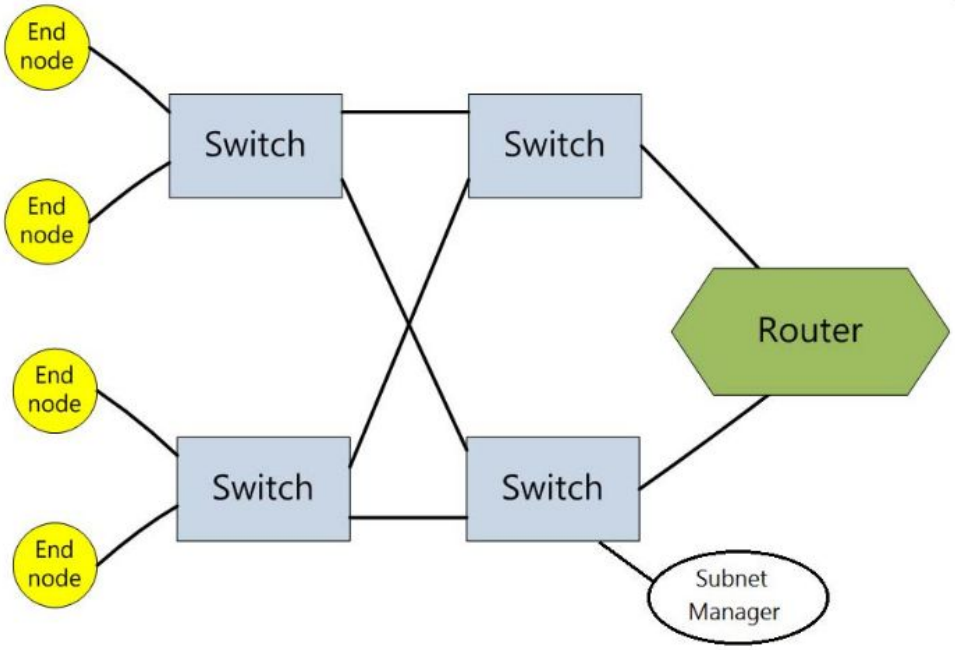
- Scientific HPC apps today run on supercomputing platforms with **1000s of communicating** cores
  - Need high performance interconnect to support high bandwidth and low latency requirements
- InfiniBand (IB) is a packet-switched networking technology defined by the InfiniBand Trade Association (IBTA) as a *high bandwidth*, *low latency* interconnect for **data center** and **HPC clusters**.
- Currently used in 28% of the top 500 supercomputers (as of November 2019)

TOP 10 Interconnect Technologies: in Top500!

		Count	System Share (%)	Rmax (TFlops)	Rpeak (TFlops)	Cores
1	Gigabit Ethernet	259	51.8	430,575	897,359	16,913,772
2	Infiniband	140	28	662,896	1,013,372	15,946,634
3	Omnipath	50	10	181,500	285,493	4,704,264
4	Custom Interconnect	45	9	356,933	524,811	24,991,220
5	Proprietary Network	5	1	13,009	16,406	508,496
6	Myrinet	1	0.2	1,975	6,595	89,600

Infiniband Architecture

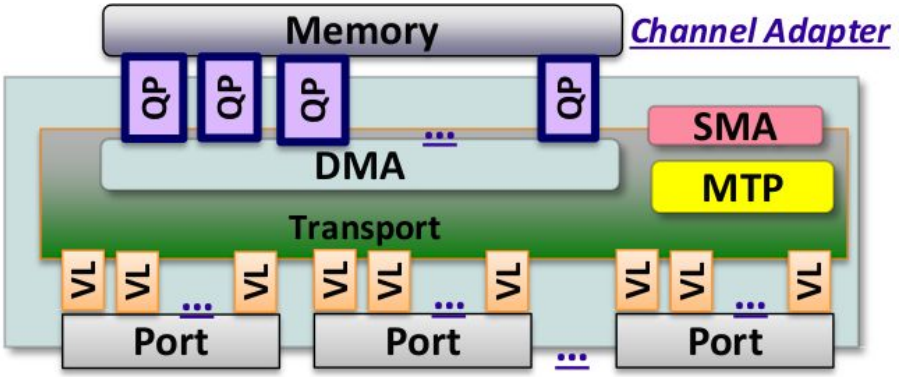
- InfiniBand Architecture of a single subnet with end nodes (typically computers), switches, a Subnet Manager (SM), and optionally a router connecting various subnets.
- End nodes attach to the subnet via Host Channel Adapters (HCAs), which are comparable to Ethernet Network Interface Cards (NICs). Each HCA supports one or more physical ports.



Host Channel Adapter

- In InfiniBand clusters, the hardware NIC is replaced by a hardware **Host Channel Adapter (HCA)**:
  - provides InfiniBand transport and network layers (comparable to TCP and IP SW)
  - provides link and physical layers (comparable to Ethernet hardware layers).
  - **Hardware protocol offload engine**

- Used by processing and I/O units to connect to fabric
- Consume & generate IB packets
- Programmable DMA engines with protection features
- May have multiple ports
  - Independent buffering channeled through Virtual Lanes



Queue Pairs

- Transport layer communication is done between a Queue Pair (QP) in each communicating HCA port.

- A Queue Pair consists of:
  - **Send Queue (SQ)**, used to send outgoing messages
  - **Receive Queue (RQ),** used to receive incoming messages

- User applications create, use, and tear down queue pairs via the **verbs API**.
  - A “queue” is analogous to a First-In First-Out (FIFO) waiting line – items must leave the queue in the same order that they enter it.

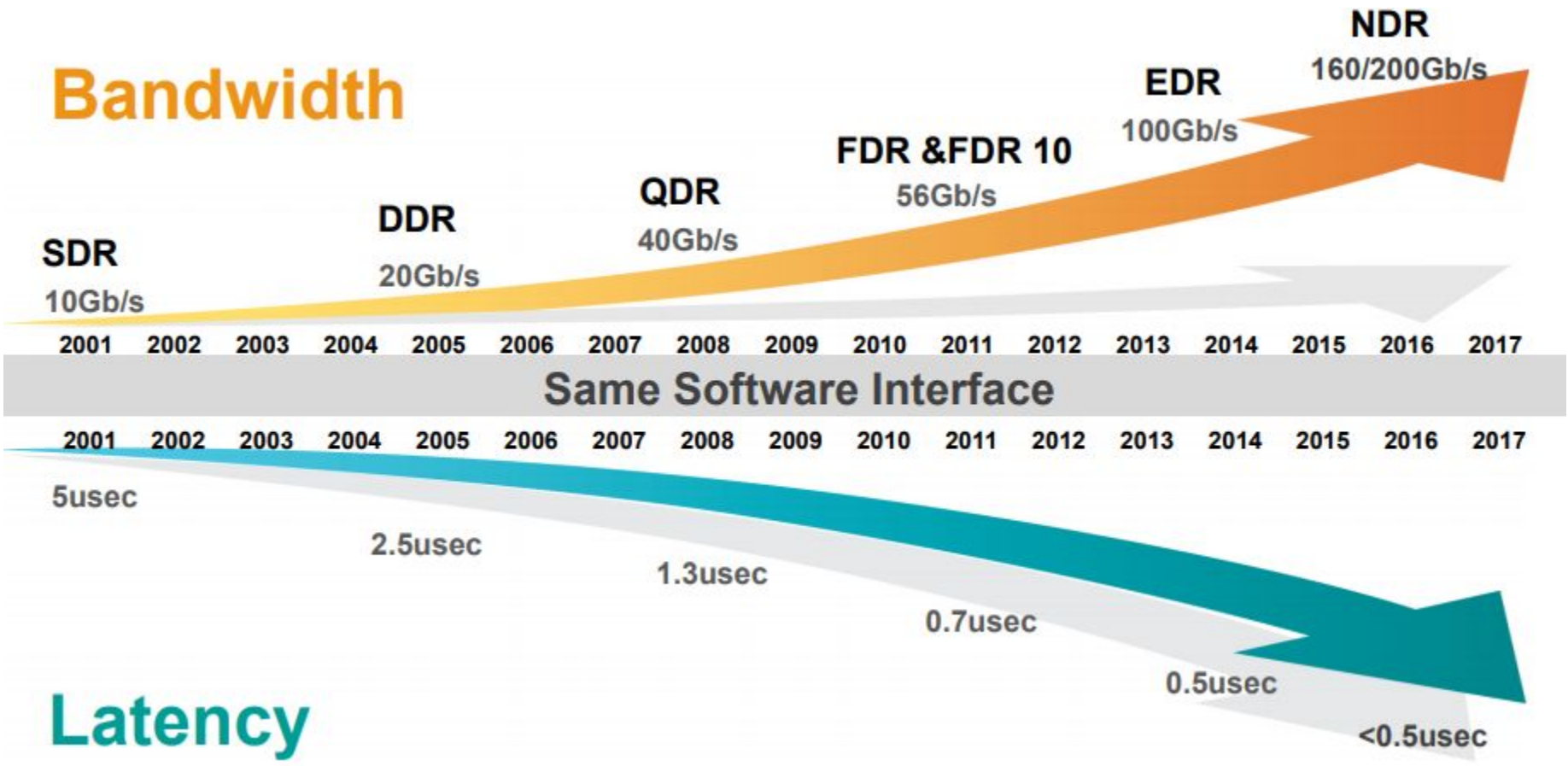
Infiniband Physical Layer

TABLE II
STANDARD INFINIBAND SPEEDS, AND THEIR PER-LANE SIGNALING AND THEORETICAL DATA RATES IN GIGABITS PER SECOND (GBPS). ALL PER-LANE SIGNALING RATES ARE MULTIPLES OF 0.15625 GBPS [35]

Name	Signaling Rate	Encoding	Data Rate
Single Data Rate (SDR)	2.5	8b/10b	2
Double Data Rate (DDR)	5	8b/10b	4
Quad Data Rate (QDR)	10	8b/10b	8
Fourteen Data Rate (FDR)	14.0625	64b/66b	13.64
Extended Data Rate (EDR)	25.78125	64b/66b	25

- *Note that achieving these speeds requires that both the Peripheral Component Interconnect express (PCIe) bus and the host memory also support them!*
- Each link contains 1, 4, 8, or 12 lanes, with **4 lanes being the most common**, leading to a data rate of 4 × 2 = 8 Gbps (1GB/s) across an SDR link.

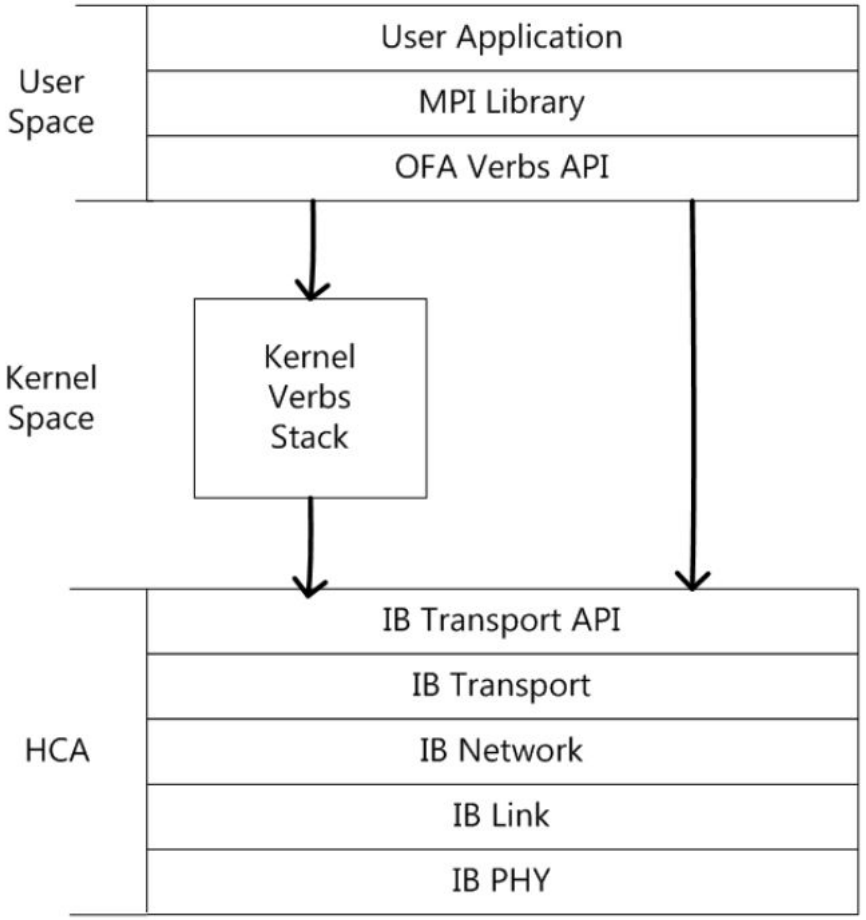
Infiniband Bandwidth and Latency



Infiniband Stack: MPI, Verbs and HW

Host Channel Adapter (HCA)

- provides Verbs interface (API) that directly accesses the HCA
- HPC applications usually access the Verbs interface via MPI



Example: List all HCAs (RDMA devices)

```
struct ibv_device **device_list;
int               num_devices;
int               i;

device_list = ibv_get_device_list(&num_devices);
if (!device_list) {
  fprintf(stderr, "Error, ibv_get_device_list() failed\n");
  exit(1);
}

for (i = 0; i < num_devices; ++ i)
  printf("RDMA device[%d]: name=%s\n", i, ibv_get_device_name(device_list[i]));

ibv_free_device_list(device_list);
```


https://www.csm.ornl.gov/workshops/openshmem2013/documents/presentations_and_tutorials/Tutorials/Verbs%20programming%20tutorial-final.pdf

POSIX Sockets and IB Verbs

- The **sockets API** has been the traditional high-level interface to networking
  - introduced in 1983 as part of the Berkeley Software Distribution (BSD) of the Unix operating system. It has evolved since then, and forms the basis for the **POSIX Sockets API** standard
- The **verbs API** is an interface introduced by the OpenIB Alliance in 2005 as the interface to InfiniBand technology.

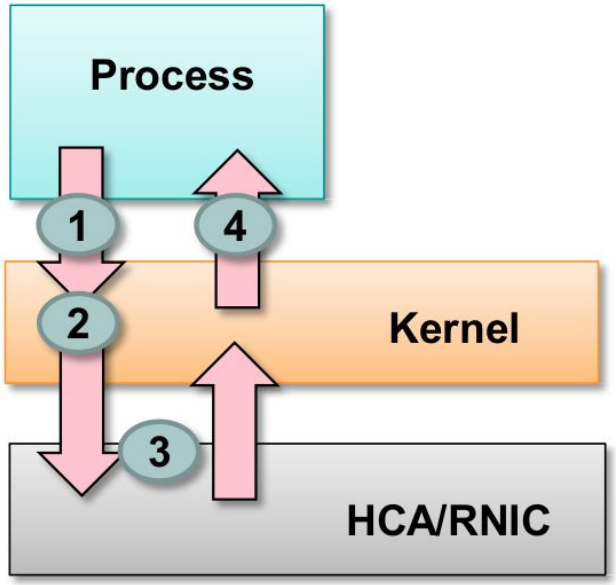
Main differences and main **IB performance features:**

1. Sockets provide a stream of bytes. Verbs provide **only messages** (better match to MPI programming style)
2. Sockets rely upon software buffering in the end nodes. Verbs do **not use buffering in the end nodes** (i.e. "zero-copy")
3. Sockets require kernel intervention during data transfers. Verbs do not, as the **user program deals directly with the HCA**. (i.e. "kernel-bypass")
4. Sockets can utilize any user-space memory for data transfers. Verbs can also utilize any user-space memory for data transfers, but they require that the application registers this memory prior to initiating the transfer.
5. Sockets operate synchronously, whereas **verbs operate asynchronously**

Memory Registration

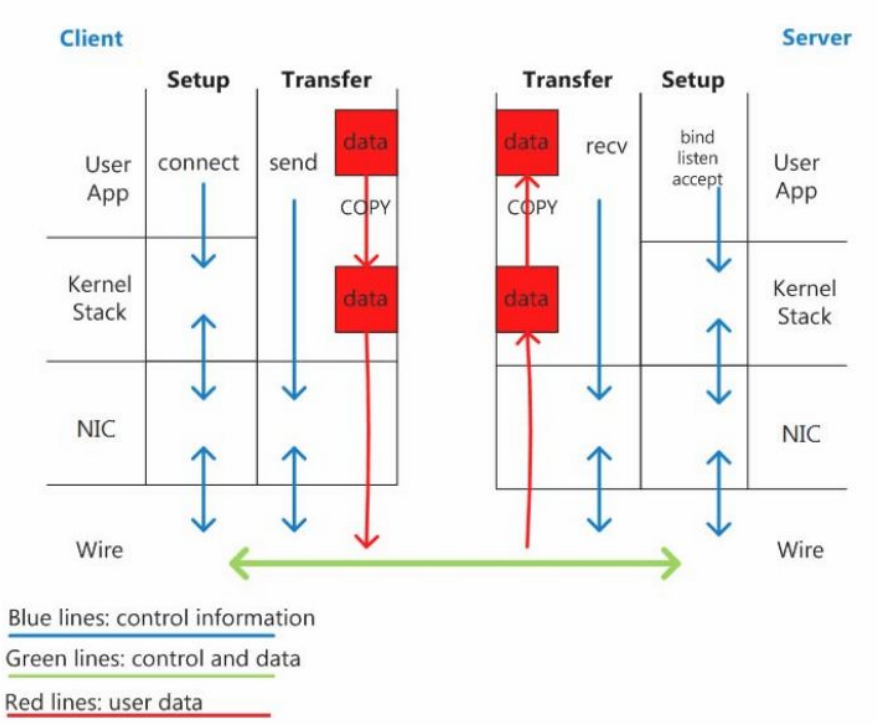
Memory Registration

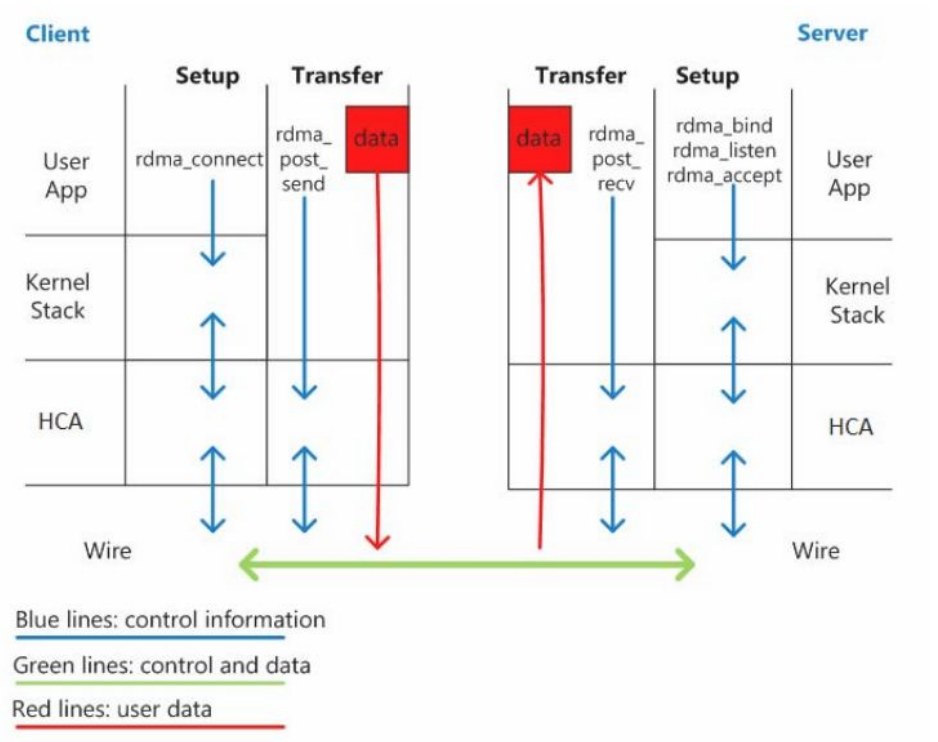
Before we do any communication:
All memory used for communication must be registered



1. Registration Request
   - Send virtual address and length
2. Kernel handles virtual->physical mapping and pins region into physical memory
   - Process cannot map memory that it does not own (security !)
3. HCA caches the virtual to physical mapping and issues a handle
   - Includes an *l_key* and *r_key*
4. Handle is returned to application

TCP/IP vs Infiniband data transfer





TCP/IP data transfer

InfiniBand Data Transfer

The RDMA Data Transfer Model

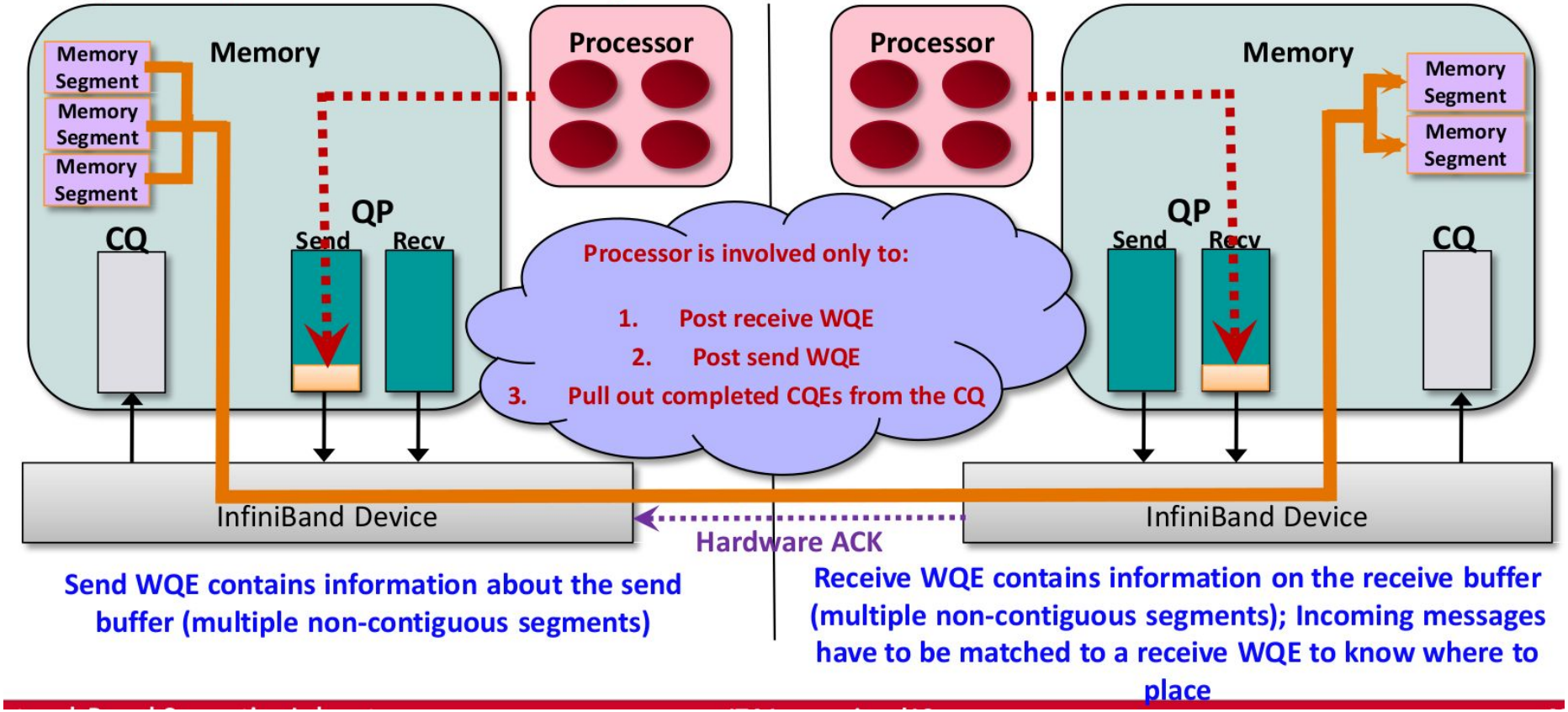
Send (Channel Semantics)

- Just like the classic model
- Data is read in local side
  - Can be gathered from multiple buffers
- Sent over the wire as a message
- Remote side specify where the message will be stored
  - Can be scattered to multiple buffers

RDMA (Memory Semantics)

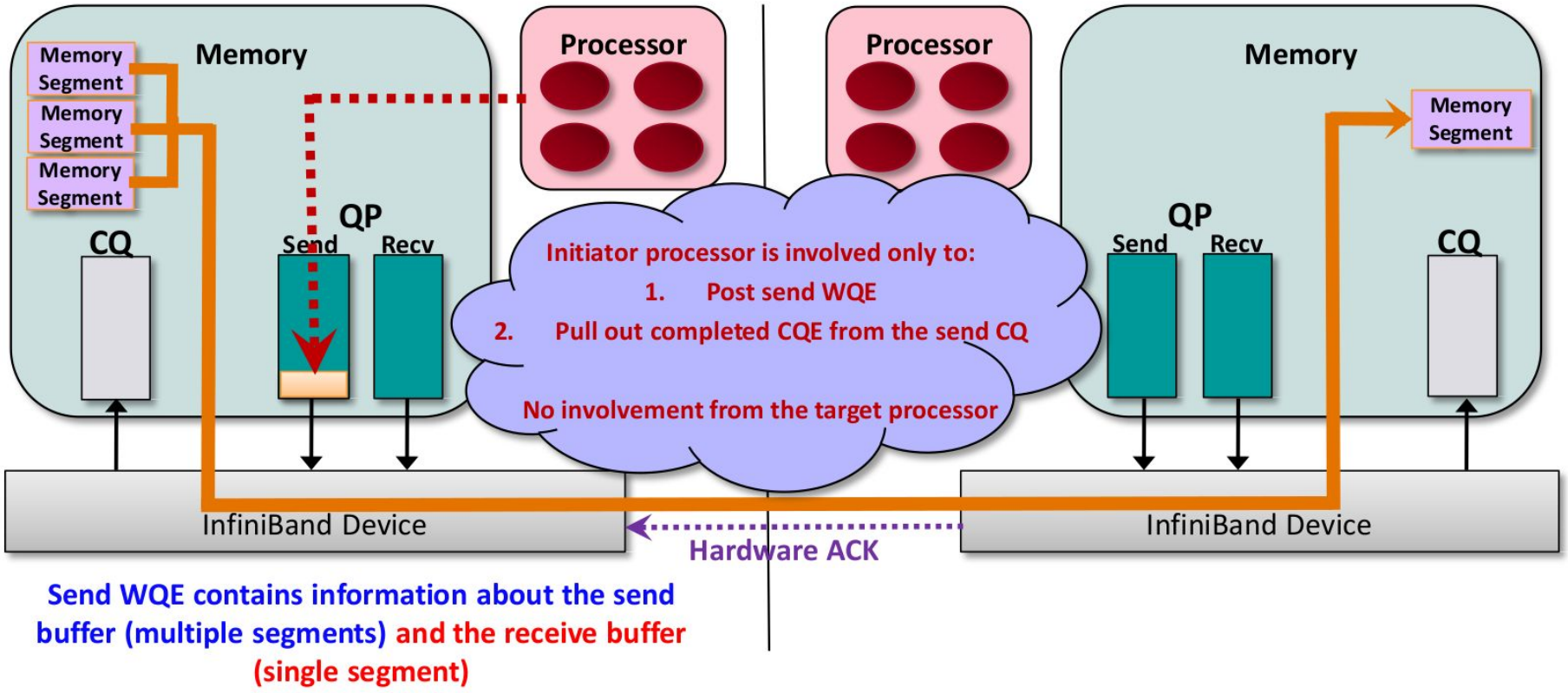
- Local side can write data directly to remote side memory
  - Can be gathered locally from multiple buffers
- Local side can read data directly from remote side memory
  - Can be scattered locally to multiple buffers
- Remote side isn't aware to any activity
  - No CPU involvement at remote side

Communication in the Channel Semantics (Send/Receive Model)



Channel semantics: similar to the well-known channel model of I/O
The `rdma_post_send` and `rdma_post_recv` verbs are analogous to similar `send` and `recv` functions in TCP because the user-level programs on both sides in a data transfer must actively participate in the transfer:

Communication in the Memory Semantics (RDMA Model)



Memory semantics: Effectively the memory of the passive user becomes an extension of the memory of the active user, since the active user can read and/or write to that memory without the passive user being aware of it.

Summary

- **Message Passing Programming**
  - Synchronous / Asynchronous
- **Message Passing Hardware**
  - CPU based, DMA & Message Processors
- **Example: Infiniband**
  - Infiniband Architecture
  - Programming API (IB Verbs vs Sockets)