Interconnects and MPI

Sep 26th, 2020

Recap

- What is Parallel Computing? Different forms or types of parallelism.
- Why Parallel everywhere? Why HPC/Supercomputing?
- A generic parallel architecture
- Shared memory programming model
- OpenMP common core
  - With Pi as an example program
- OpenMP programming assignment

Discussion today

- Interconnection networks
  - Discussion aim: to provide a high-level view and feel (bear with the technical jargon)
- Distributed-memory programming
  - MPI

U.S. DOE System Architecture Targets

System attributes	2010	2018-2019		2021-2022	
System peak	2 Peta	150-200 Petaflop/sec		1 Exaflop/sec	
System memory	0.3 PB	5 PB		32-64 PB	
Node performance	125 GF	3 TF	30 TF	10 TF	100 TF
Node memory BW	25 GB/s	0.1TB/sec	1 TB/sec	0.4TB/sec	4 TB/sec
Node concurrency	12	O(100)	O(1,000)	O(1,000)	O(10,000)
System size (nodes)	18,700	50,000	5,000	100,000	10,000
Total Node Interconnect BW	1.5 GB/s	20 GB/sec		200GB/sec	
MTTI	days	O(1day)		O(1 day)	

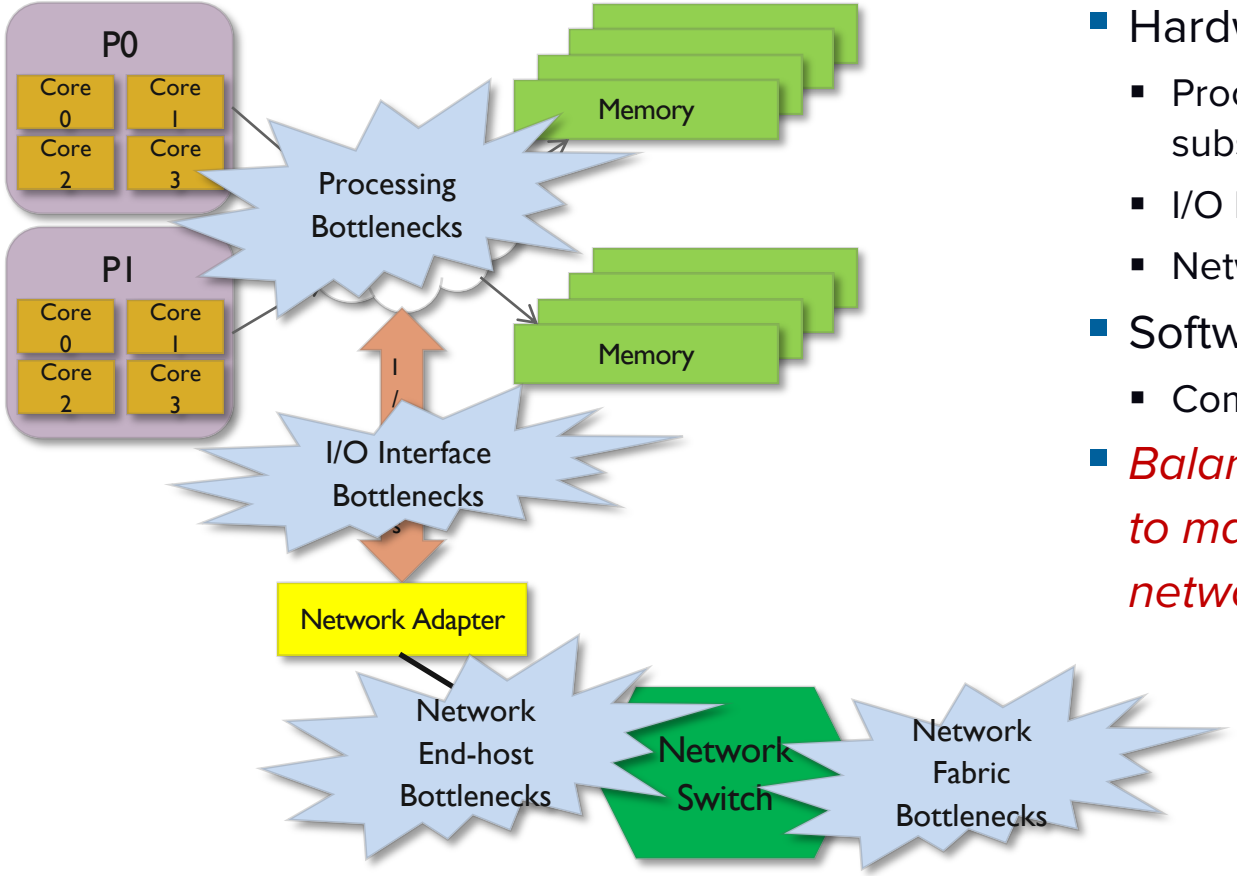
Past production ***Current generation (e.g., CORAL)*** ***Exascale Goals***

[Includes modifications to the DOE Exascale report]

General Trends in System Architecture

- Number of nodes is increasing, but at a moderate pace
- Number of cores/threads on a node is increasing rapidly
- Each core is not increasing in speed (clock frequency)
- What does this mean for networks?
  - More sharing of the network infrastructure
  - The aggregate amount of communication from each node will increase moderately, but will be divided into many smaller messages
  - A single CPU core may not be able fully saturate the NIC

A Simplified Network Architecture

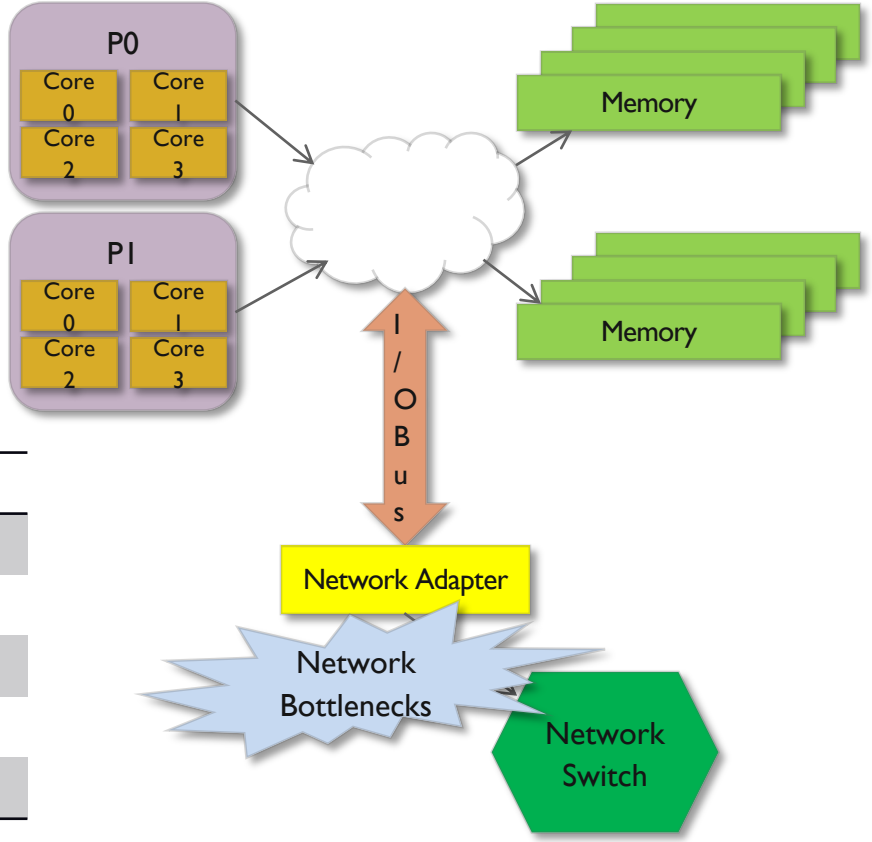


- Hardware components
  - Processing cores and memory subsystem
  - I/O bus or links
  - Network adapters/switches
- Software components
  - Communication stack
- *Balanced approach required to maximize user-perceived network performance*

Bottlenecks on Traditional Network Adapters

- Network speeds plateaued at around 1 Gbps
  - Features provided were limited
  - Commodity networks were not considered scalable enough for large-scale systems

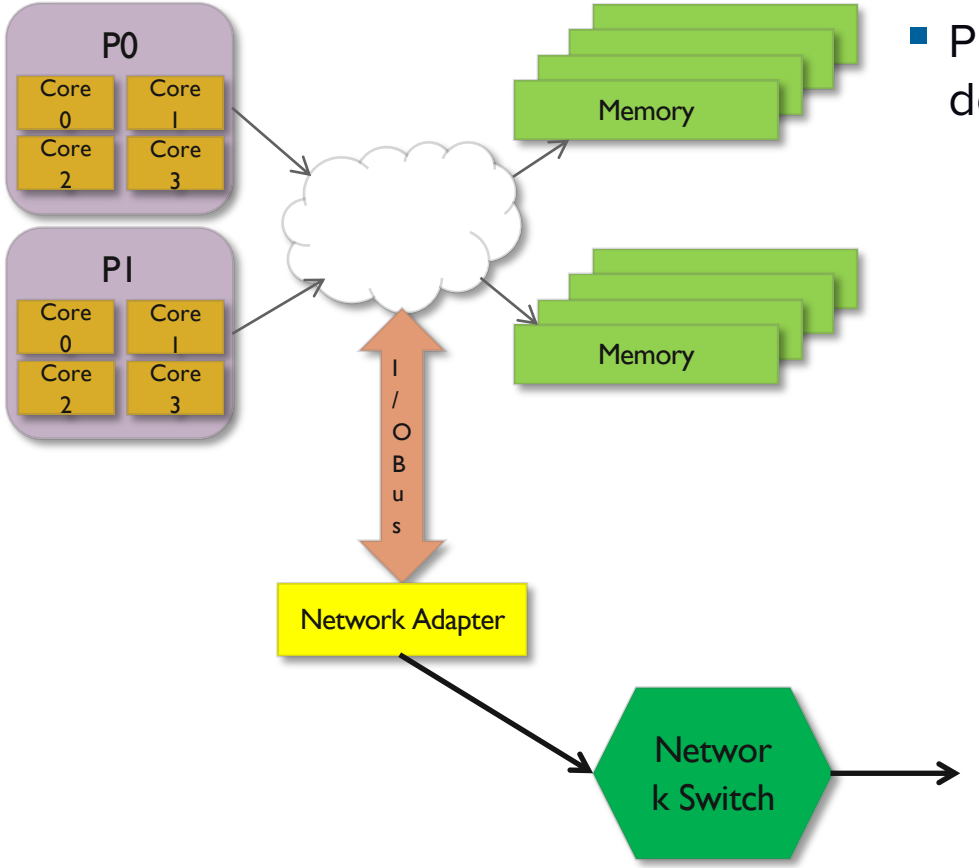
Ethernet (1979 -)	10 Mbit/sec
Fast Ethernet (1993 -)	100 Mbit/sec
Gigabit Ethernet (1995 -)	1000 Mbit /sec
ATM (1995 -)	155/622/1024 Mbit/sec
Myrinet (1993 -)	1 Gbit/sec
Fibre Channel (1994 -)	1 Gbit/sec



End-host Network Interface Speeds

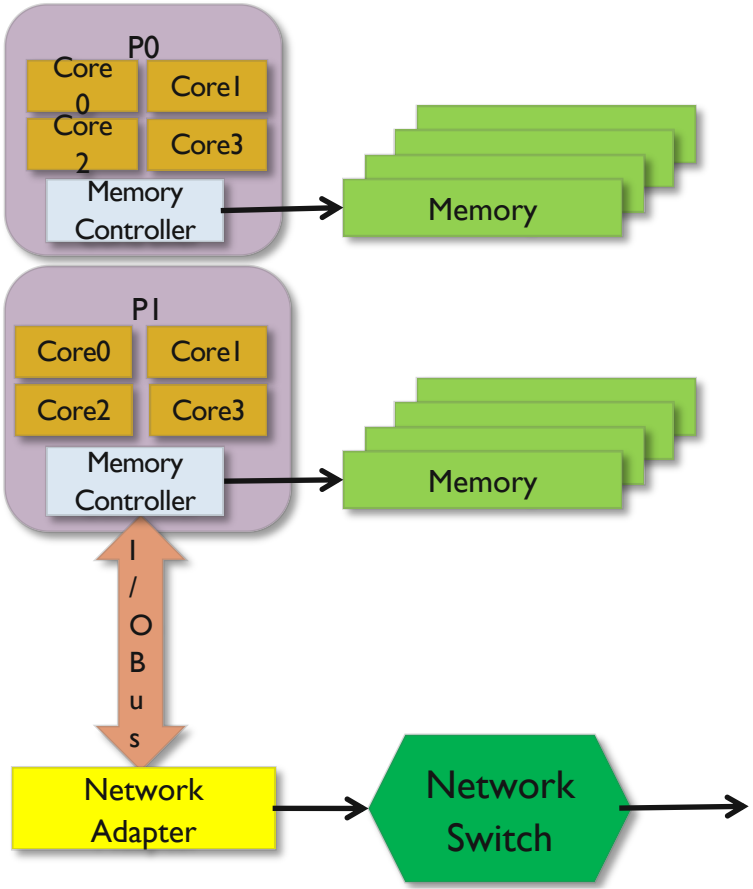
- HPC network technologies provide high bandwidth links
  - InfiniBand EDR gives 100 Gbps per network link
    - Will continue to increase (HDR 200 Gbps, etc.)
  - Multiple network links becoming a common place
    - ORNL Summit and LLNL Sierra machines, Japanese Post T2K machine
    - Torus style or other multi-dimensional networks
- End-host peak network bandwidth is “mostly” no longer considered a major limitation
- Network latency still an issue
  - That’s a harder problem to solve – limited by physics, not technology
    - There is some room to improve it in current technology (trimming the fat)
    - Significant effort in making systems denser so as to reduce network latency
- Other important metrics: message rate, congestion, …

Simple Network Architecture (past systems)



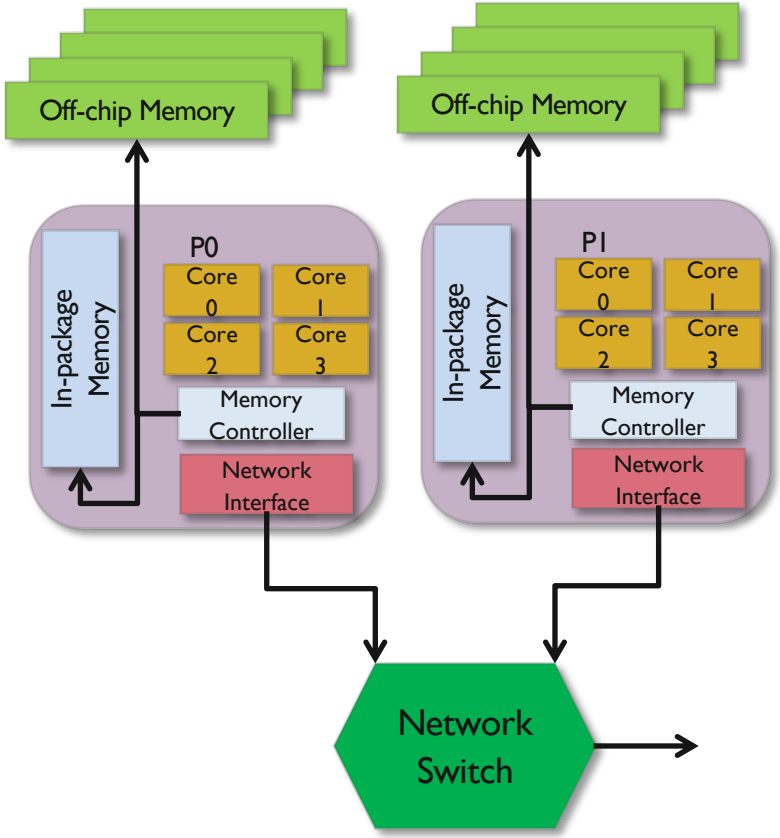
- Processor, memory and network are all decoupled

Integrated Memory Controllers (current systems)



- Memory controllers have been integrated on to the processor
- Primary purpose was scalable memory bandwidth (NUMA)
- Also helps network communication
  - Data transfer to/from network requires coordination with caches
- Several network I/O technologies exist
  - PCIe, HTX, NVLink
  - Expected to provide higher bandwidth than what network links will have

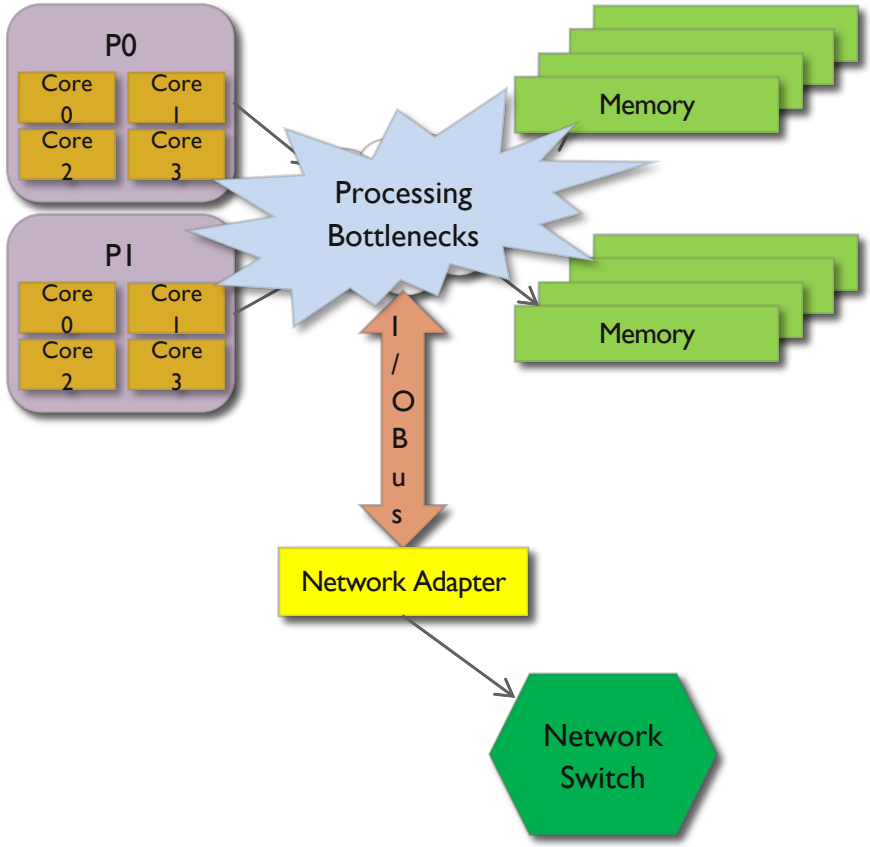
Integrated Network?



- May improve network bandwidth
  - Unclear if the I/O bus would be a bottleneck
- Improves network latencies
  - Control messages between the processor, network, and memory are now on-chip
- Improved network functionality
  - Communication is a first-class citizen and better integrated with processor features
  - E.g., network atomic operations can be atomic with respect to processor atomics
- Seems unlikely in the near-term

Processing Bottlenecks in Traditional Protocols

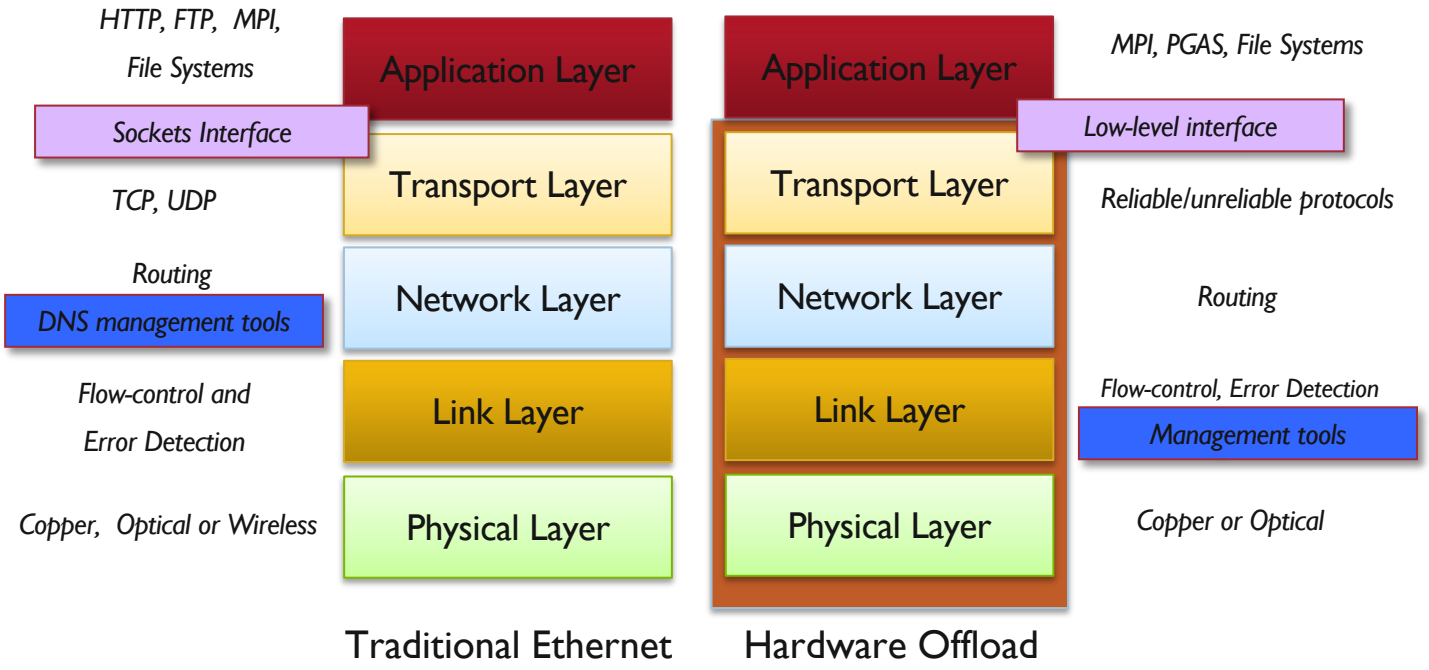
- Ex: TCP/IP, UDP/IP
- Generic architecture for all networks
- Host processor handles almost all aspects of communication
  - Data buffering (copies on sender and receiver)
  - Data integrity (checksum)
  - Routing aspects (IP routing)
- Signaling between different layers
  - Hardware interrupt on packet arrival or transmission
  - Software signals between different layers to handle protocol processing in different priority levels



Network Protocol Stacks: The Offload Era

- Modern networks are spending more and more network real-estate on offloading various communication features on hardware
- Network and transport layers are hardware offloaded for most modern HPC networks
  - Reliability (retransmissions, CRC checks), packetization
  - OS-based memory registration, and user-level data transmission

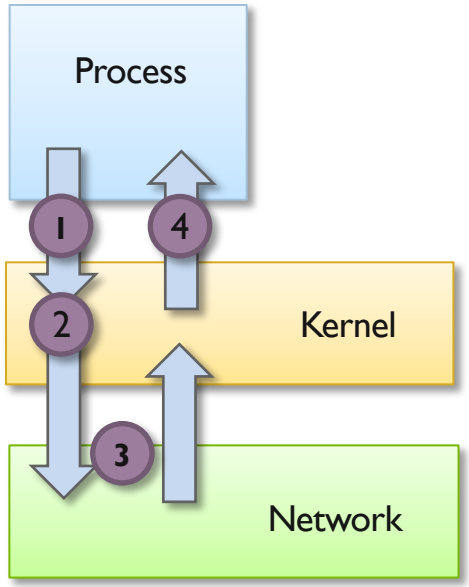
Offloaded Network Stacks vs. Traditional Network Stacks



User-level Communication: Memory Registration

Before we do any communication:

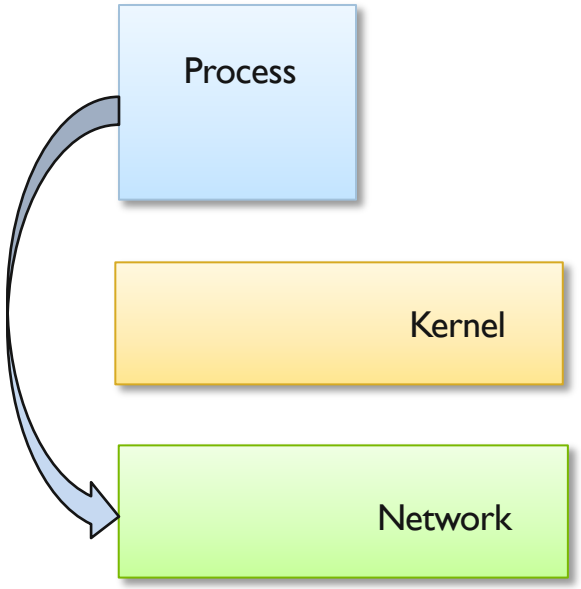
All memory used for communication must be registered



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. Network adapter caches the virtual to physical mapping and issues a handle
4. Handle is returned to application

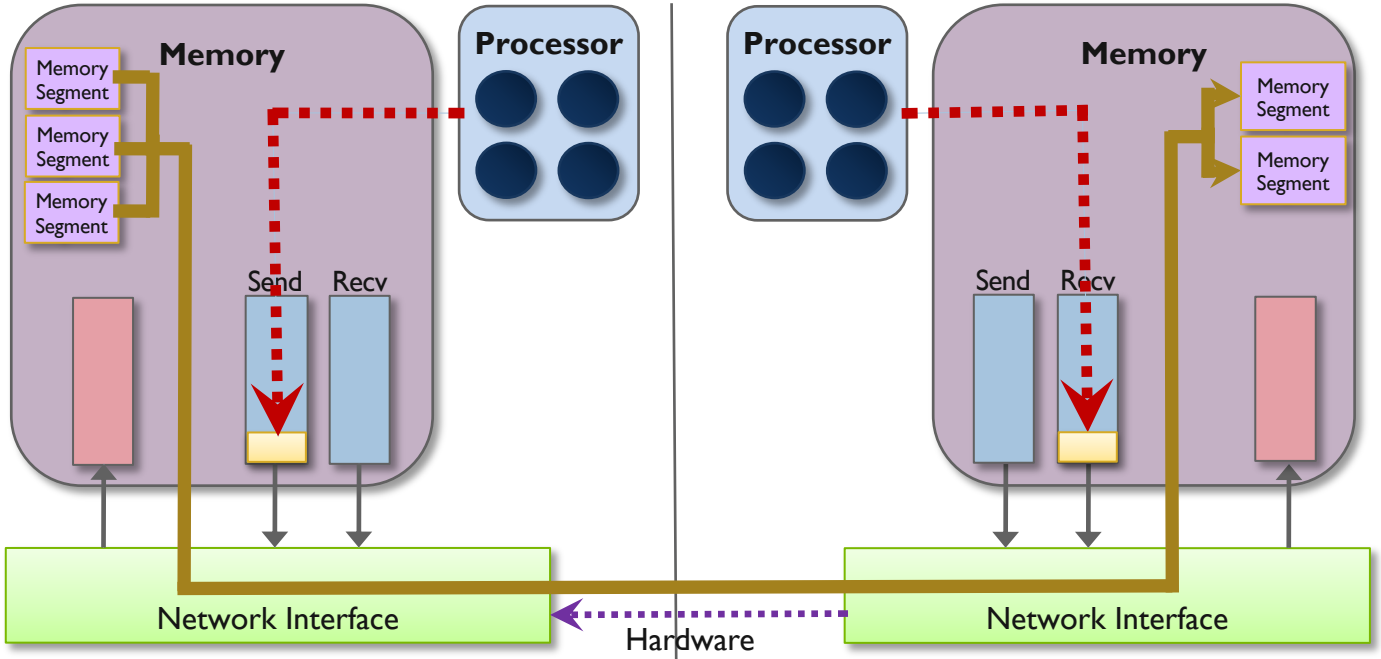
User-level Communication: OS Bypass

User-level APIs allow direct interaction with network adapters



- Contrast with traditional network APIs that trap down to the kernel
- Eliminates heavyweight context switch
- Memory registration caches allow for fast buffer re-use, further reducing dependence on the kernel

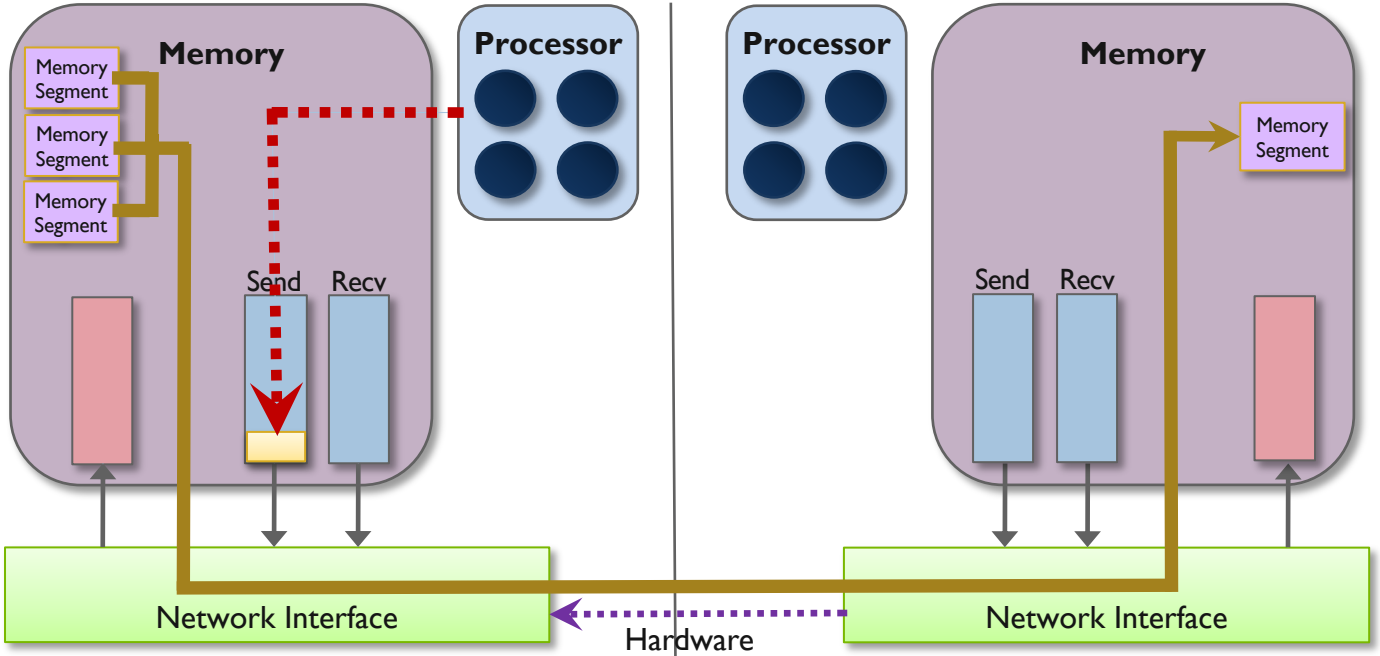
Send/Receive Communication



Send entry contains information about the send buffer (multiple non-contiguous segments)

Receive entry contains information on the receive buffer (multiple non-contiguous segments); Incoming messages have to be matched to a receive entry to know where to place

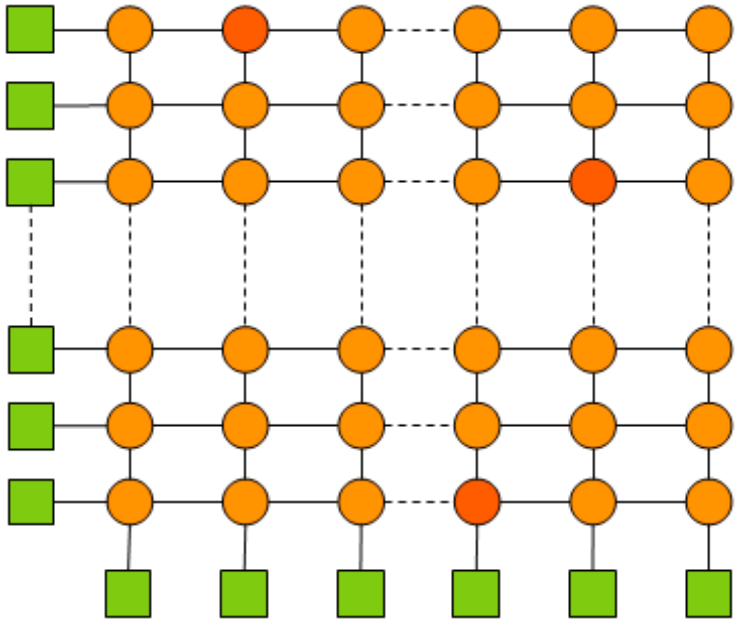
PUT/GET Communication



Send entry contains information about the send buffer (multiple segments) and the receive buffer (single segment)

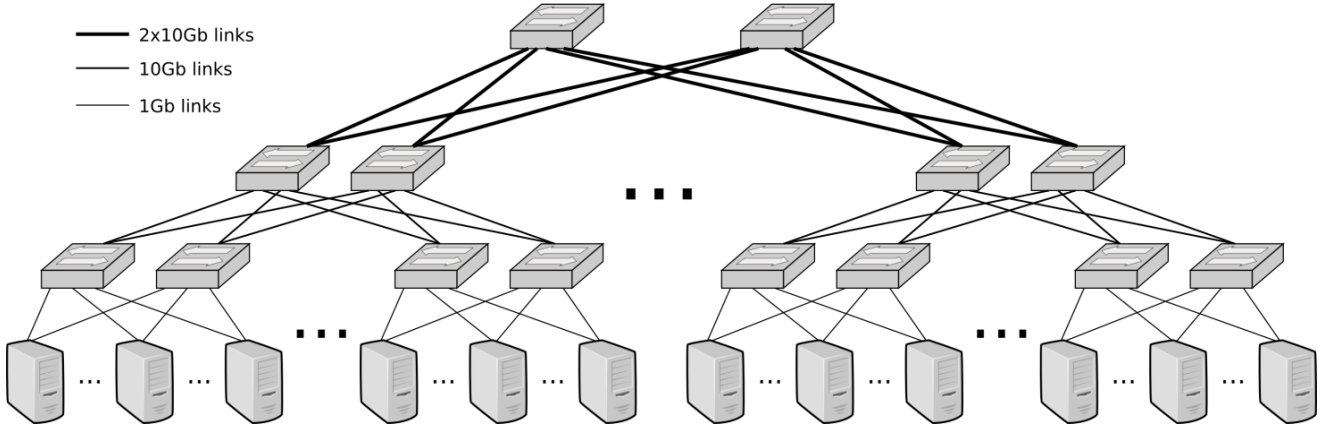
Traditional Network Topologies: Crossbar

- A network topology describes how different network adapters and switches are interconnected with each other
- The ideal network topology (for performance) is a crossbar
  - Alltoall connection
  - Typically done on a single network ASIC
  - Current network crossbar ASICs go up to ~64 ports
  - All communication is nonblocking



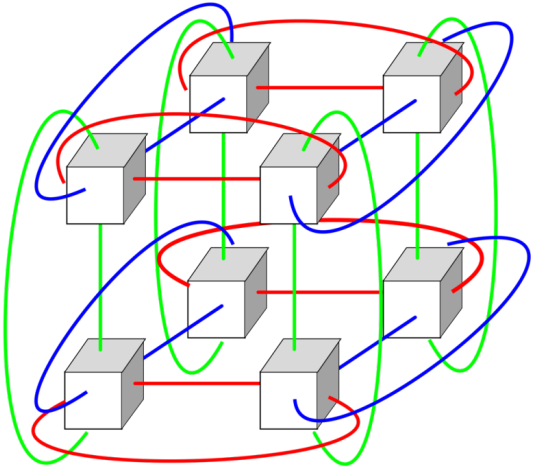
Traditional Network Topologies: Fat-tree

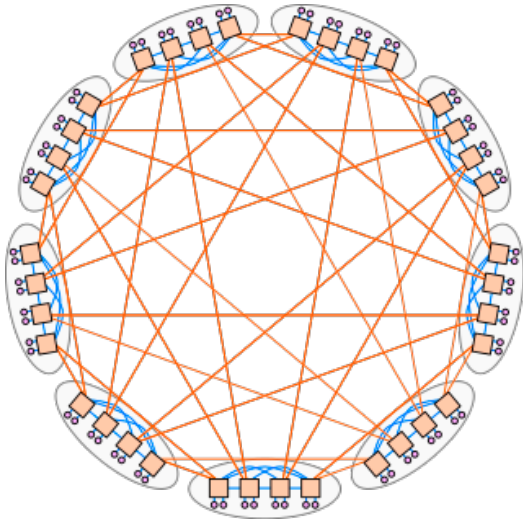
- Common topology for small and medium scale systems
  - Nonblocking fat-tree switches available in abundance
    - Allows for pseudo nonblocking communication
    - Between all pairs of processes, there exists a completely nonblocking path, but not all paths are nonblocking
  - More scalable than crossbars, but the number of network links still increases super-linearly with node count
    - Can get expensive at scale



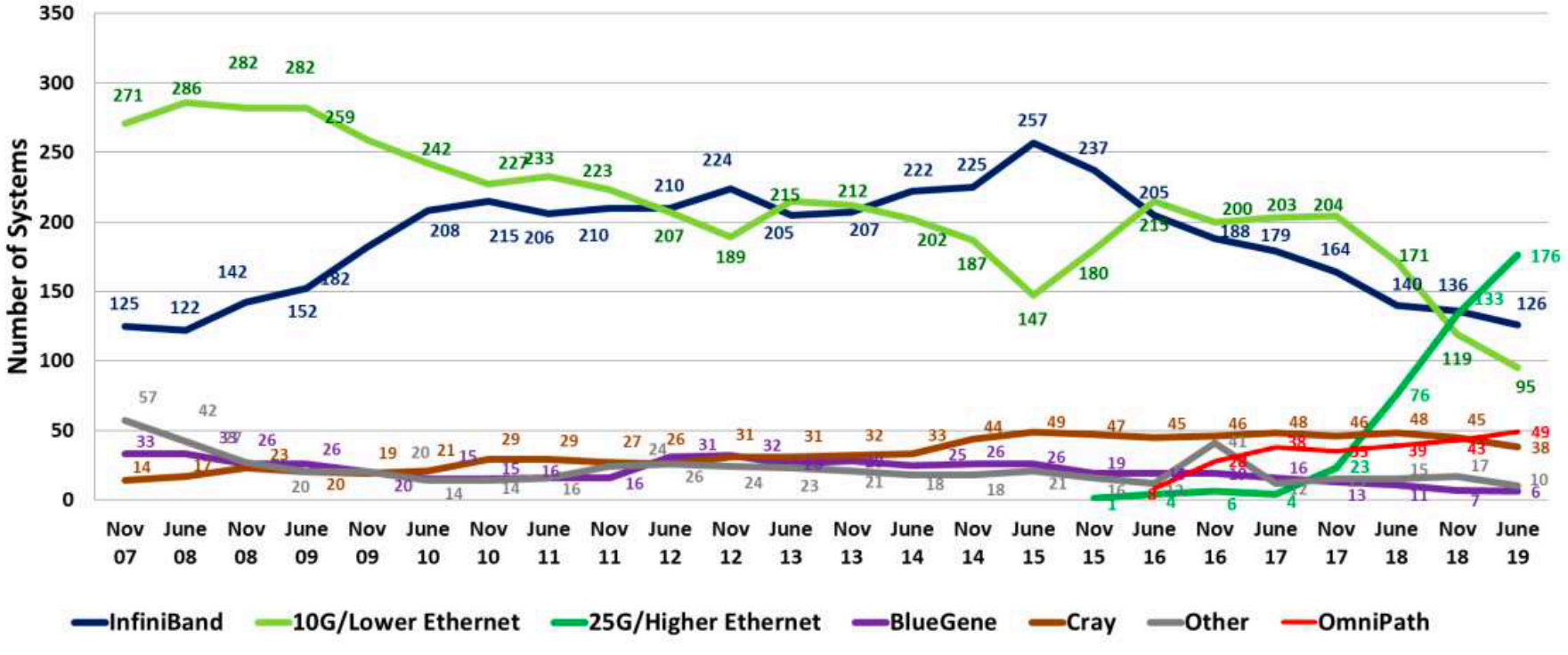
Scalable Network Topologies

- Large-scale topologies must account for hardware cost
- BlueGene, K, and Fugaku supercomputers use a torus network; Cray systems use dragonfly
  - Linear increase in the number of links/routers with system size (cost savings)
  - Increased network diameter causing increased latency
  - Any communication that is more than one hop away has a possibility of interference – congestion is not just possible, but common
  - Adaptive routing and traffic classes aim to help minimize congestion
- Take-away: data locality is critical





Top500 Interconnect Trends



The TOP500 List has Evolved to Include Both HPC and Cloud / Web2.0 Hyperscale Platforms

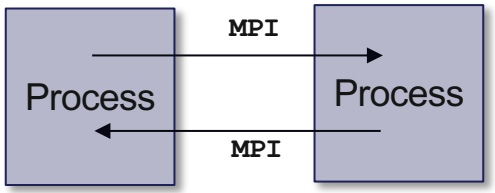
Introduction to MPI

Sample Parallel Programming Models

- Shared Memory Programming
  - Processes share memory address space (threads model)
  - Application ensures no data corruption (Lock/Unlock)
- Transparent Parallelization
  - Compiler works magic on sequential programs
- Directive-based Parallelization
  - Compiler needs help (e.g., OpenMP)
- Message Passing
  - Explicit communication between processes (like sending and receiving emails)

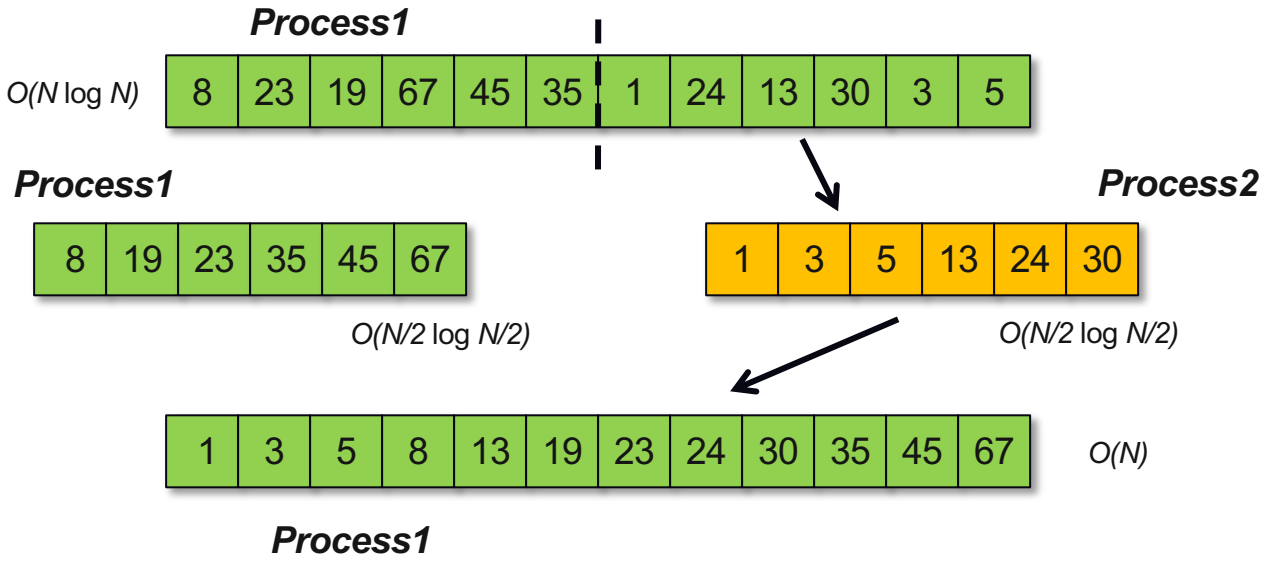
The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Inter-process communication consists of
  - synchronization
  - movement of data from one process's address space to another's.



The Message-Passing Model (an example)

- Each process has to send/receive data to/from other processes
- Example: Sorting Integers



Standardizing Message-Passing Models with MPI

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum was a collection of vendors, portability writers and users that wanted to standardize all these efforts

What is MPI?

- MPI: Message Passing Interface
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers
    - MPI-1 finished in 18 months
  - Incorporates the best ideas in a “standard” way
    - Each function takes fixed arguments
    - Each function has fixed semantics
      - Standardizes what the MPI implementation provides and what the application can and cannot expect
      - Each system can implement it differently as long as the semantics match
- MPI is not…
  - a language or compiler specification
  - a specific implementation or product

MPI-1

- MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, etc
- MPI-1 was defined (1994) by a broadly based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)

Following MPI Standards

- MPI-2 was released in 1997
  - Several additional features including MPI + threads, MPI-I/O, remote memory access functionality and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were released with some corrections to the standard and small features
- MPI-3 (2012) added several new features to MPI (see next slide)
- MPI-3.1 (2015) is the latest version of the standard with minor corrections and features
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of material including tutorials, a FAQ, other MPI pages

Status of MPI-3.1 Implementations

	MPICH	MVAPICH	Open MPI	Cray	Tianhe	Intel		IBM			HPE	Fujitsu	MS	MPC	NEC	Sunway	RIKEN	AMPI
						IMPI	MPICH-OFI	BG/Q (legacy)[1]	PE (legacy)[2]	Spectrum								
NBC																		
Nbr. Coll.													✗					
RMA													(*)					Q2 '18
Shr. mem																		Q1 '18
MPI_T													*					Q2 '18
Comm-create group													*					
F08 Bindings									✗			✗	✗					Q2 '18
New Dtypes																		
Large Counts																		
MProbe																		Q1 '18
NBC I/O					✗			✗	✗			✗	✗	*		✗		Q3 '18

[1] Open Source but unsupported [2] No MPI_T variables exposed * Under development (*) Partly done

Release dates are estimates; subject to change at any time "✗" indicates no publicly announced plan to implement/support that feature Platform-specific restrictions might apply to the supported features

Web Pointers

- MPI Standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: www.microsoft.com/en-us/download/details.aspx?id=39961
  - Open MPI : http://www.open-mpi.org/
  - IBM Spectrum MPI, Cray MPI, TH MPI, …
- Several MPI tutorials can be found on the web

MPI Example Programs

- The lecture notes is based on slides from https://www.mcs.anl.gov/~raffenet/permalinks/argonne19_mpi.php
- Examples can be accessed from https://anl.app.box.com/v/2019-ANL-MPI/folder/79663311293

Compiling and Running MPI applications (more details later)

- MPI is a library
  - Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required
- Compilation:
  - Regular applications:

```
% gcc test.c -o test
```

  - MPI applications

```
% mpicc test.c -o test
```

- Execution:
  - Regular applications

```
% ./test
```

  - MPI applications (running with 16 processes)

```
% mpiexec –n 16 ./test
```

What is MPICH?

- MPICH is a high-performance and widely portable open-source implementation of MPI
- It provides all features of MPI that have been defined so far (including MPI-1, MPI-2.0, MPI-2.1, MPI-2.2, MPI-3.0 and MPI-3.1)
- Active development led by Argonne National Laboratory
  - Several close collaborators who contribute many features, bug fixes, testing for quality assurance, etc.
    - Intel, Cray, Mellanox, The Ohio State University, Microsoft, and many others

Getting Started with MPICH

- From UNIX distributions
  - Most UNIX/Linux distributions package MPICH for easy installation
  - apt-get (Ubuntu/Debian), yum (Fedora, Centos), brew/port (Mac OS)
- Open-source for source-based installation
  - Download MPICH
    - Go to http://www.mpich.org and follow the downloads link
    - The download will be a zipped tarball
  - Build MPICH
    - Unzip/untar the tarball

```
% tar xvfz mpich-<latest_version>.tar.gz
% cd mpich-<latest_version>
% ./configure --prefix=/where/to/install/mpich |& tee
      c.log
% make |& tee m.log
% make install |& tee mi.log
% Add /where/to/install/mpich/bin to your PATH
```

Compiling MPI programs with MPICH

- Compilation Wrappers
  - For C programs:
  - For C++ programs:
  - For Fortran programs:

```
% mpicc test.c -o test
% mpicxx test.cpp -o test
% mpifort test.f90 -o test
```

- You can link other libraries are required too
  - To link to a math library:

```
% mpicc test.c -o test -lm
```

- You can just assume that “mpicc” and friends have replaced your regular compilers (gcc, gfortran, etc.)

Running MPI programs with MPICH

- Launch 16 processes on the local node:

```
% mpiexec -n 16 ./test
```

Launch 16 processes on 4 nodes (each has 4 cores)

```
% mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test
```

Runs the first four processes on h1, the next four on h2, etc.

```
% mpiexec -hosts h1,h2,h3,h4 -n 16 ./test
```

Runs the first process on h1, the second on h2, etc., and wraps around

So, h1 will have the 1st, 5th, 9th and 13th processes

If there are many nodes, it might be easier to create a host file

```
% cat hf
        h1:4
        h2:2
% mpiexec -hostfile hf -n 16 ./test
```

Trying some example programs

- MPICH comes packaged with several example programs using almost ALL of MPICH's functionality
- A simple program to try out is the PI example written in C (cpi.c) – calculates the value of PI in parallel (available in the examples directory when you build MPICH)

```
% mpiexec -n 16 ./examples/cpi
```

  - The output will show how many processes are running, and the error in calculating PI
  - Next, try it with multiple hosts

```
% mpiexec -hosts h1:2,h2:4 -n 16 ./examples/cpi
```

Interaction with Resource Managers

- Resource managers such as SGE, PBS, SLURM or Cray ALPS are common in many managed clusters
  - MPICH automatically detects them and interoperates with them
- For example with PBS, you can create a script such as:

```
#! /bin/bash
% cd $PBS_O_WORKDIR
# No need to provide -np or -hostfile options
% mpiexec ./test
```

```
% qsub -l nodes=2:ppn=2 test.sub
```

- Job can be submitted as:
  - "mpiexec" will automatically know that the system has PBS, and ask PBS for the number of cores allocated (4 in this case), and which nodes have been allocated
- The usage is similar for other resource managers

MPI Communicators

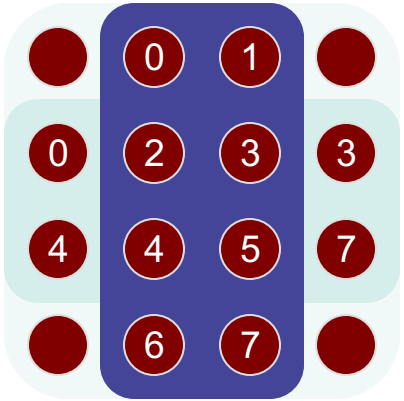
- MPI processes can be collected into groups
  - Each group can have multiple colors (some times called context)
  - *Group + color == communicator (it is like a name for the group)*
  - When an MPI application starts, the group of all processes is initially given a predefined name called **`MPI_COMM_WORLD`**
    - The same group can have many names, but simple programs do not have to worry about multiple names
- A process is identified by a unique number within each communicator, called *rank*
  - For two different communicators, the same process can have two different ranks: so the meaning of a “rank” is only defined when you specify the communicator

Communicators

```
% mpiexec -n 16 ./test
```

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called a “rank”



When you start an MPI program, there is one predefined communicator **MPI_COMM_WORLD**

Can make copies of this communicator (same group of processes, but different “aliases”)

The same process might have different ranks in different communicators

Communicators can be created “by hand” or using tools provided by MPI (not discussed in this tutorial)

Simple programs typically only use the predefined communicator **MPI_COMM_WORLD**

Simple MPI Program Identifying Processes

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

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank + 1, size);

    MPI_Finalize();
    return 0;
}
```

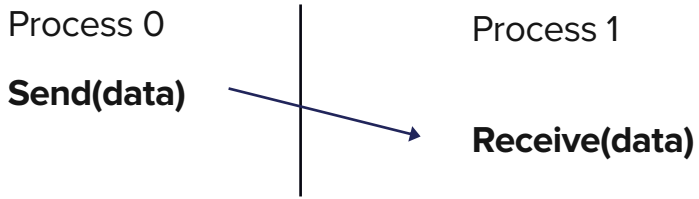
Basic requirements for an MPI program

Example: Hello World!

- *communicators/hello.c*
- Basic program where each process prints its rank

MPI Basic Send/Receive

- Simple communication model



- Application needs to specify to the MPI implementation:
  1. How do you compile and run an MPI application?
  2. How will processes be identified?
  3. How will “data” be described?

Data Communication

- Data communication in MPI is like email exchange
  - One process sends a copy of the data to another process (or a group of processes), and the other process receives it
- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to understand what type of data is being received)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be **`MPI_ANY_SOURCE`**, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be **`MPI_ANY_TAG`**)

More Details on Describing Data for Communication

- MPI Datatype is very similar to a C or Fortran datatype
  - `int` → `MPI_INT`
  - `double` → `MPI_DOUBLE`
  - `char` → `MPI_CHAR`
- More complex datatypes are also possible:
  - E.g., you can create a structure datatype that comprises of other datatypes → a char, an int and a double.
  - Or, a vector datatype for the columns of a matrix
- The "count" in `MPI_SEND` and `MPI_RECV` refers to how many datatype elements should be communicated

MPI Basic (Blocking) Send

```
MPI_Send(const void *buf, int count,
         MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- The message buffer is described by (**buf, count, datatype**).
- The target process is specified by **dest** and **comm**.
  - **dest** is the rank of the target process in the communicator specified by **comm**.
- **tag** is a user-defined "type" for the message
- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Basic (Blocking) Receive

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype,
         int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- Waits until a matching (on **source**, **tag**, **comm**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator **comm**, or **MPI_ANY_SOURCE**.
- Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.
- **status** contains further information:
  - Who sent the message (can be used if you used **MPI_ANY_SOURCE**)
  - How much data was actually received
  - What tag was used with the message (can be used if you used **MPI_ANY_TAG**)
  - **MPI_STATUS_IGNORE** can be used if we don't need any additional information

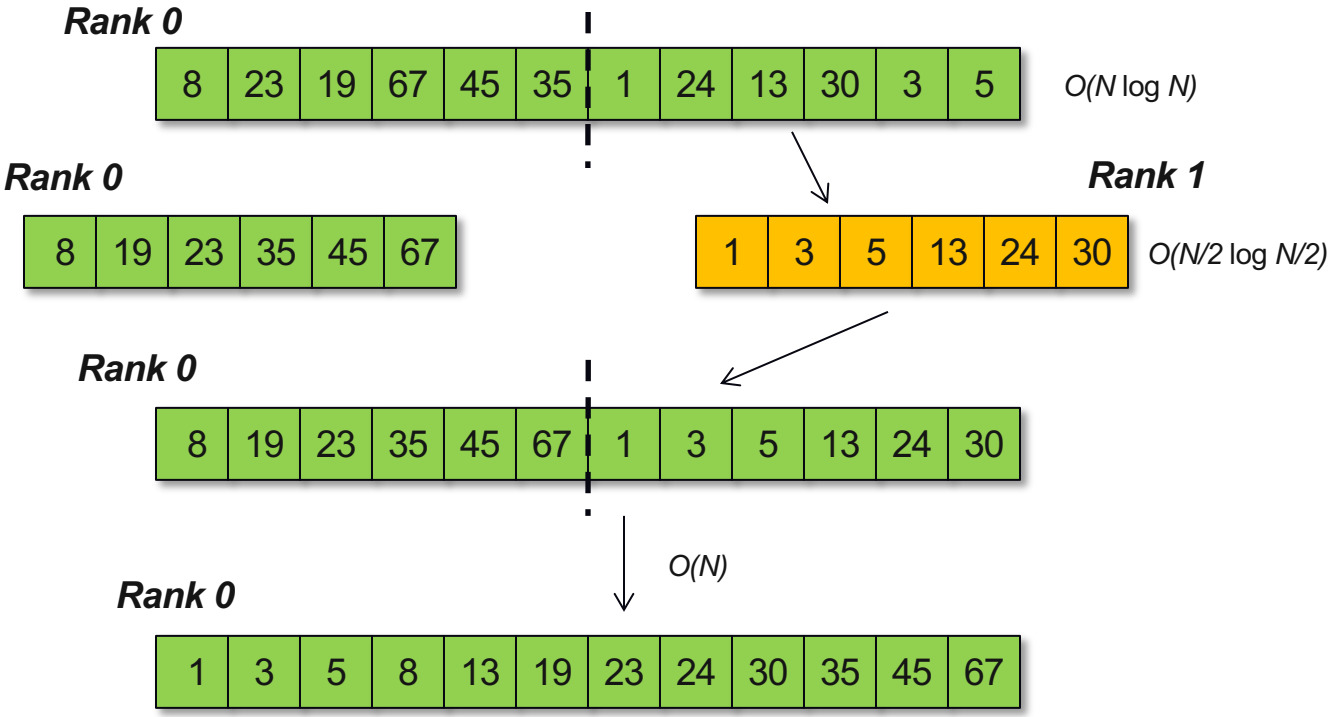
Simple Communication in MPI

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

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

Example: Basic Send/Receive

- *blocking_p2p/sendrecv.c*
- Simple send/receive program to show basic data transfer

Parallel Sort using MPI Send/Recv



Parallel Sort using MPI Send/Recv (contd.)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
    int rank, a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        /* Serial: Merge array b and sorted part of array a */
    }
    else if (rank == 1) {
        MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
```

Example: Sorting with Two Processes

- *blocking_p2p/sort_2_procs.c*
- Sorting using two processes

Status Object

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (**`status.MPI_SOURCE`**)
  - The message tag (**`status.MPI_TAG`**)
  - Error status (**`status.MPI_ERROR`**)
- The number of elements received is given by:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

`status`	return status of receive operation (status)
`datatype`	datatype of each receive buffer element (handle)
`count`	number of received elements (integer)(OUT)

Using the “status” field (contd.)

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

int main(int argc, char ** argv)
{
    [..snip..]
    if (rank < world_size - 1) {  /* worker process */
        MPI_Send(data, rand() % 100, MPI_INT, world_size - 1, rank / 2,
                 MPI_COMM_WORLD);
    }
    else {  /* master process */
        for (i = 0; i < world_size - 1; i++) {
            MPI_Recv(data, 100, MPI_INT, MPI_ANY_SOURCE,
                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT, &count);
            printf("worker ID: %d; task ID: %d; count: %d\n",
                   status.MPI_SOURCE, status.MPI_TAG, count);
        }
    }
    [..snip..]
}
```

Example: Master-worker Computation

- *blocking_p2p/master_worker_simple.c*
- Master-worker example
  - Each task is computed by three workers
  - Every worker knows statically what task it needs to compute
  - Results accumulated by the master
  - The amount of data sent by each worker is arbitrary, but less than 100 integers

Summary

- MPI is simple
- Many (if not all) parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`** – initialize the MPI library (must be the first routine called)
  - **`MPI_COMM_SIZE`** – get the size of a communicator
  - **`MPI_COMM_RANK`** – get the rank of the calling process in the communicator
  - **`MPI_SEND`** – send a message to another process
  - **`MPI_RECV`** – send a message to another process
  - **`MPI_FINALIZE`** – clean up all MPI state (must be the last MPI function called by a process)
- For performance, however, you need to use other MPI features