Xi'an Jiaotong-Liverpool University
西交利物浦大学

DTS205TC High Performance Computing

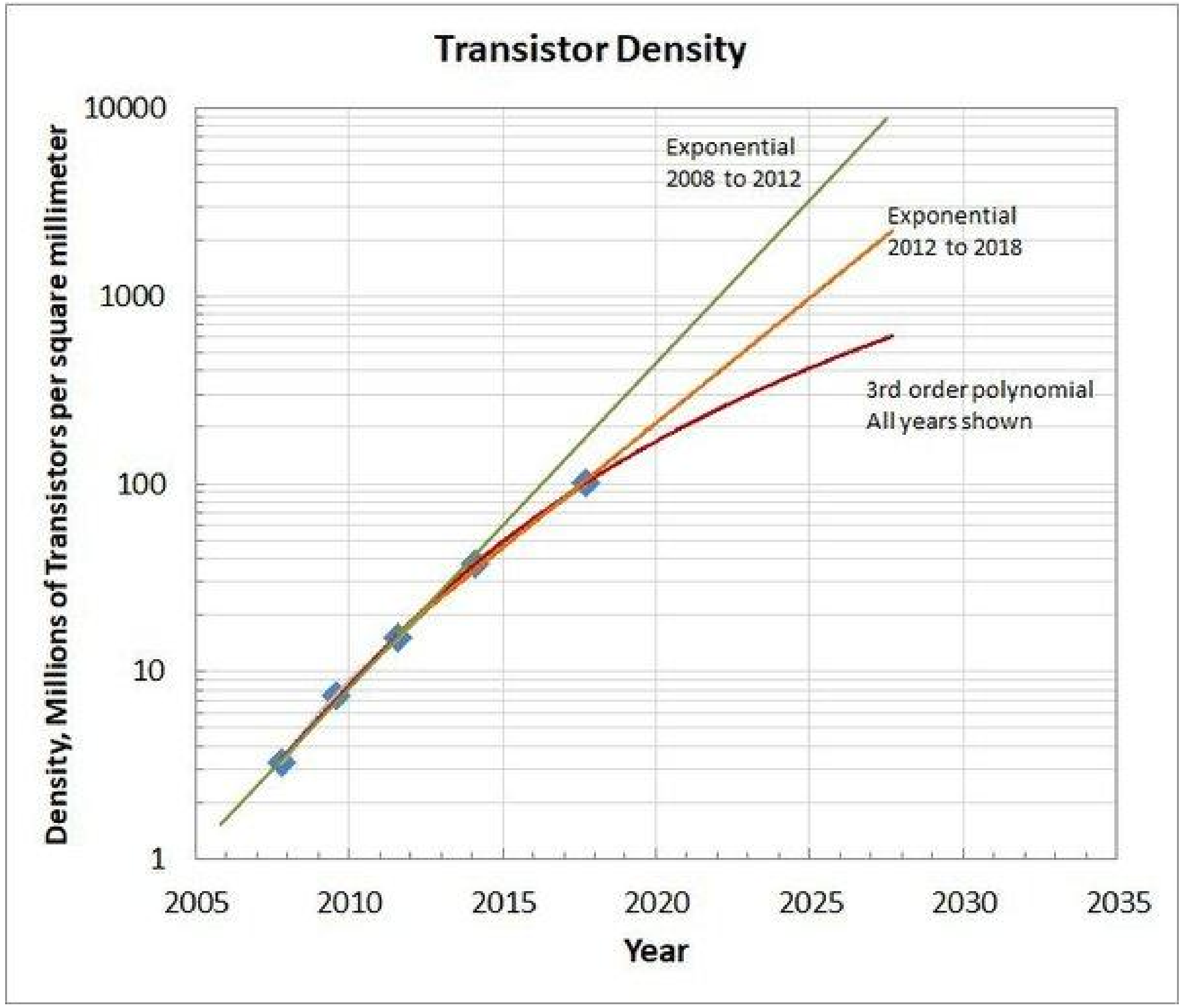
Lecture 5 Network Topology and MPI

Di Zhang, Spring 2024

Section

- Networks inside Data Center

Motivation of Parallel Computing

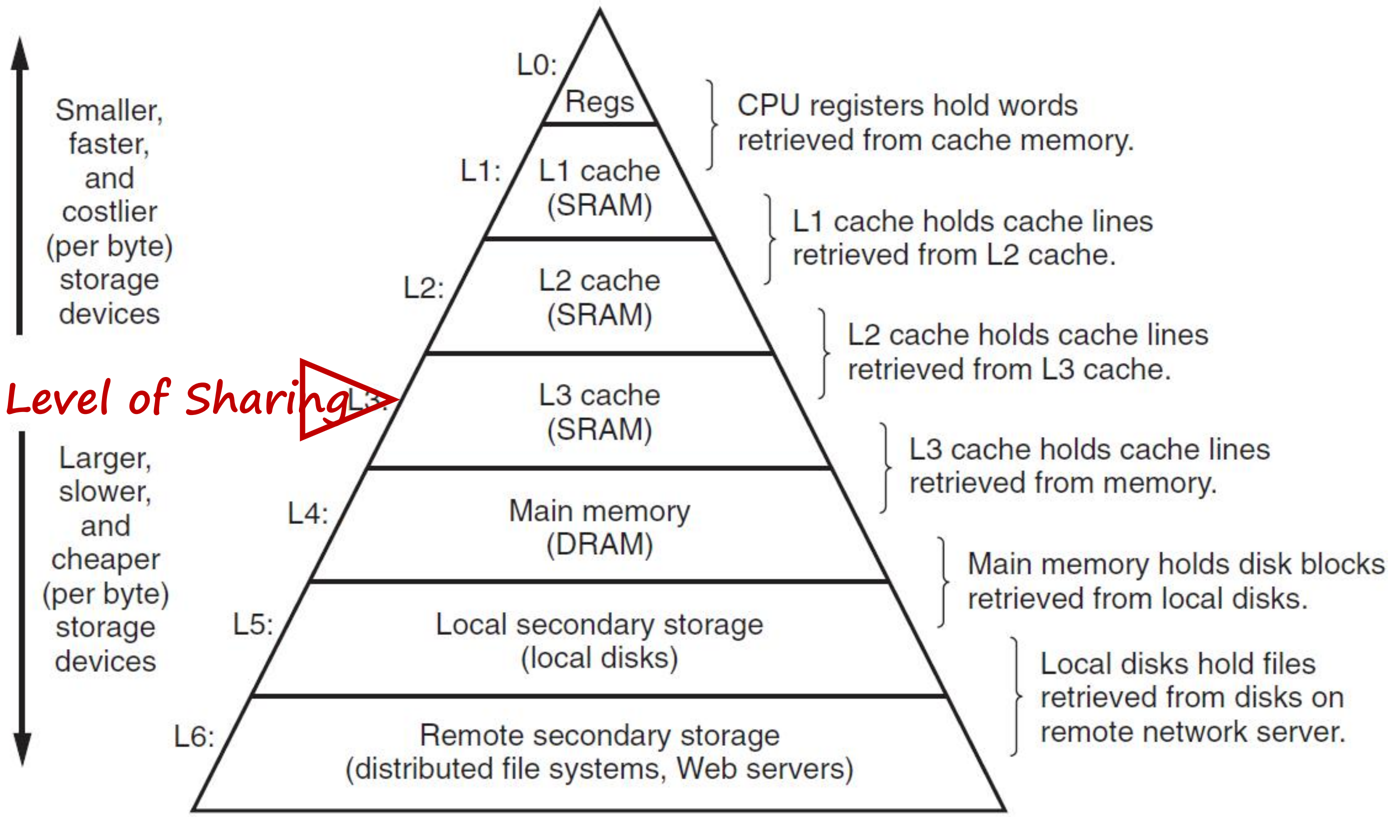


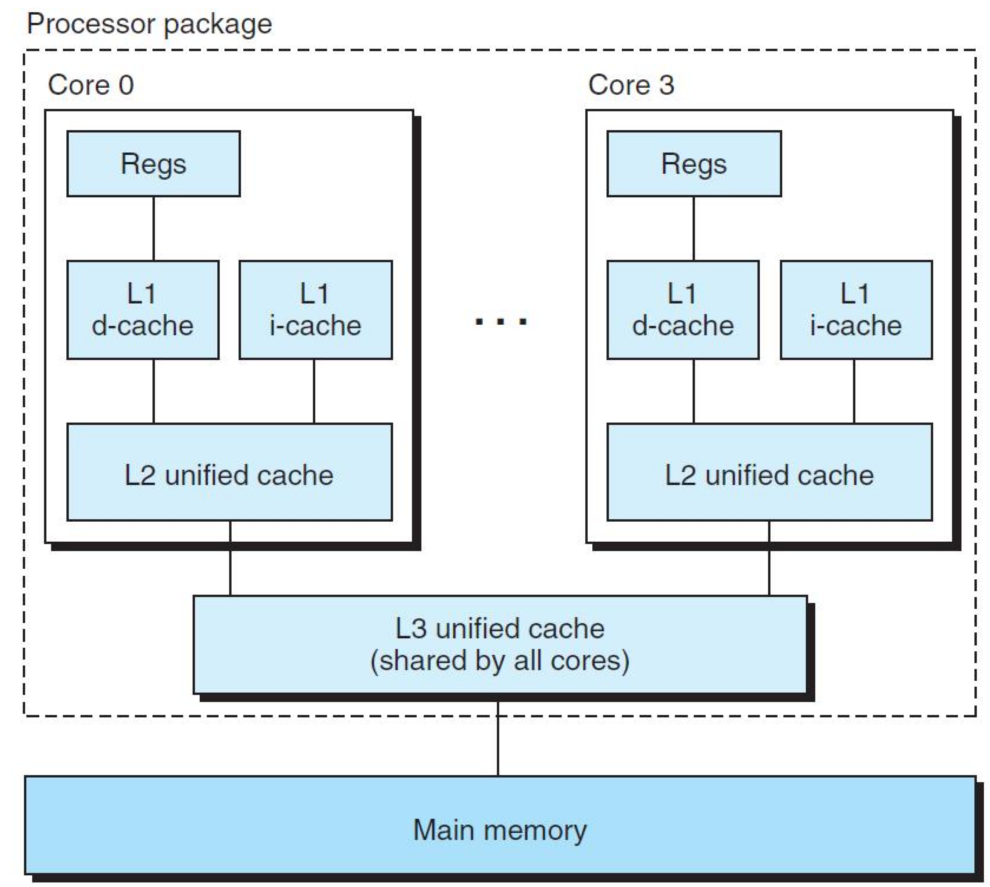
The failure of Moore's Law

- A single processor can no longer be faster. How to design the architecture so that multiple processors can cooperate to shorten the running time?

Multi-core solution

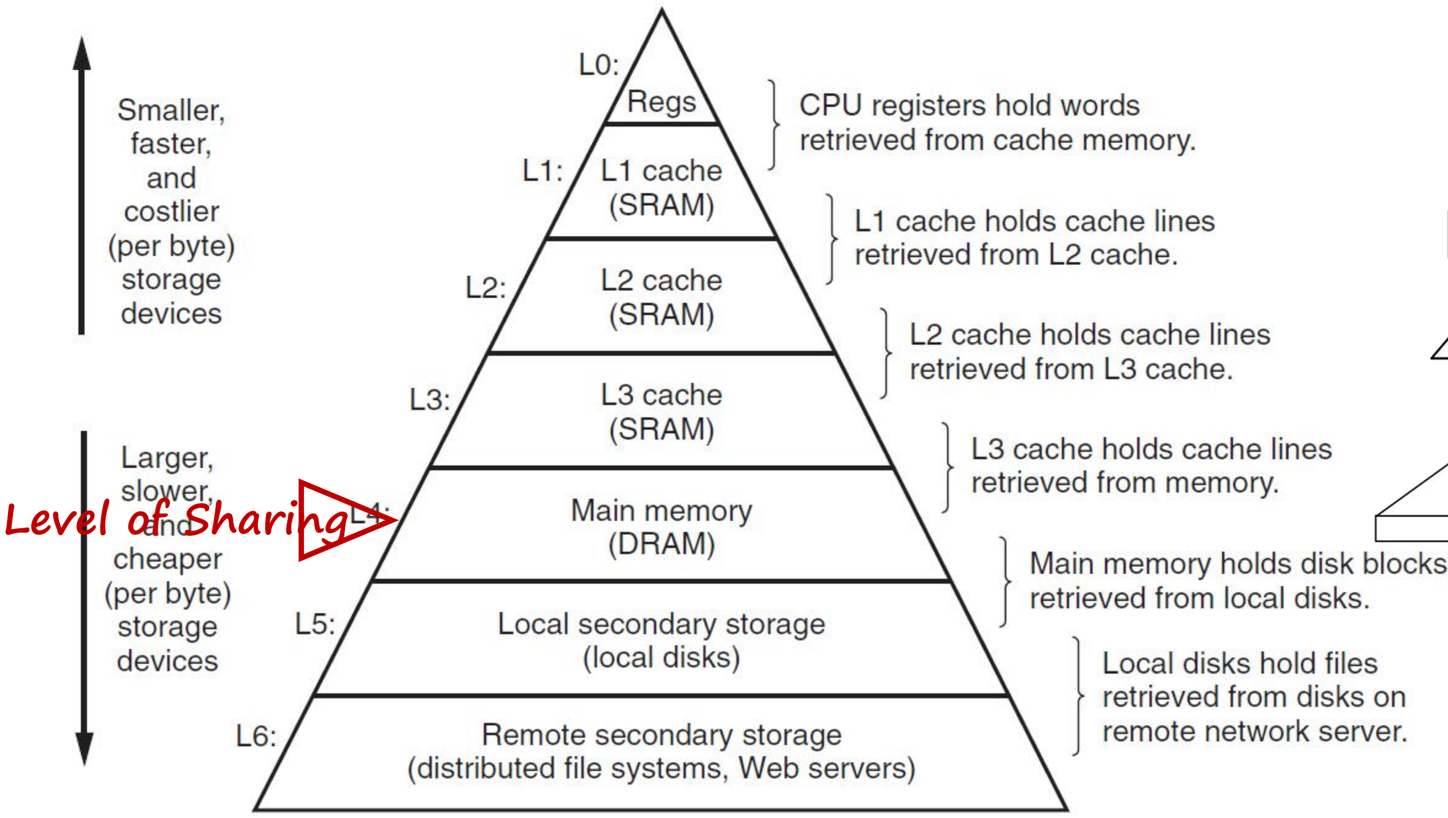
Xi'an Jiaotong-Liverpool University 西交利物浦大学

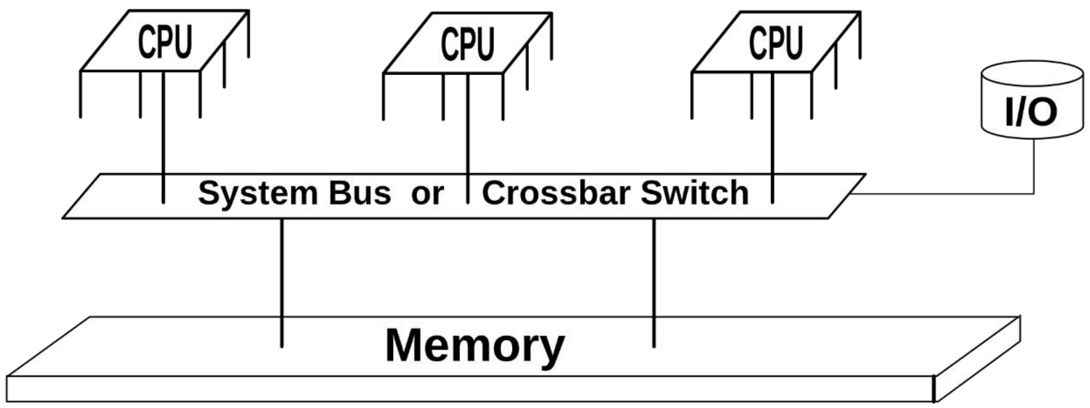




- Architectural choice: depends on what level you want multiple processors to share ‘storage’
- Influencing factors:
  - Performance: We want multiple processors to be as close together as possible to reduce synchronization latency
    Cost: Integrating too many cores in a limited space can be expensive, and even impossible

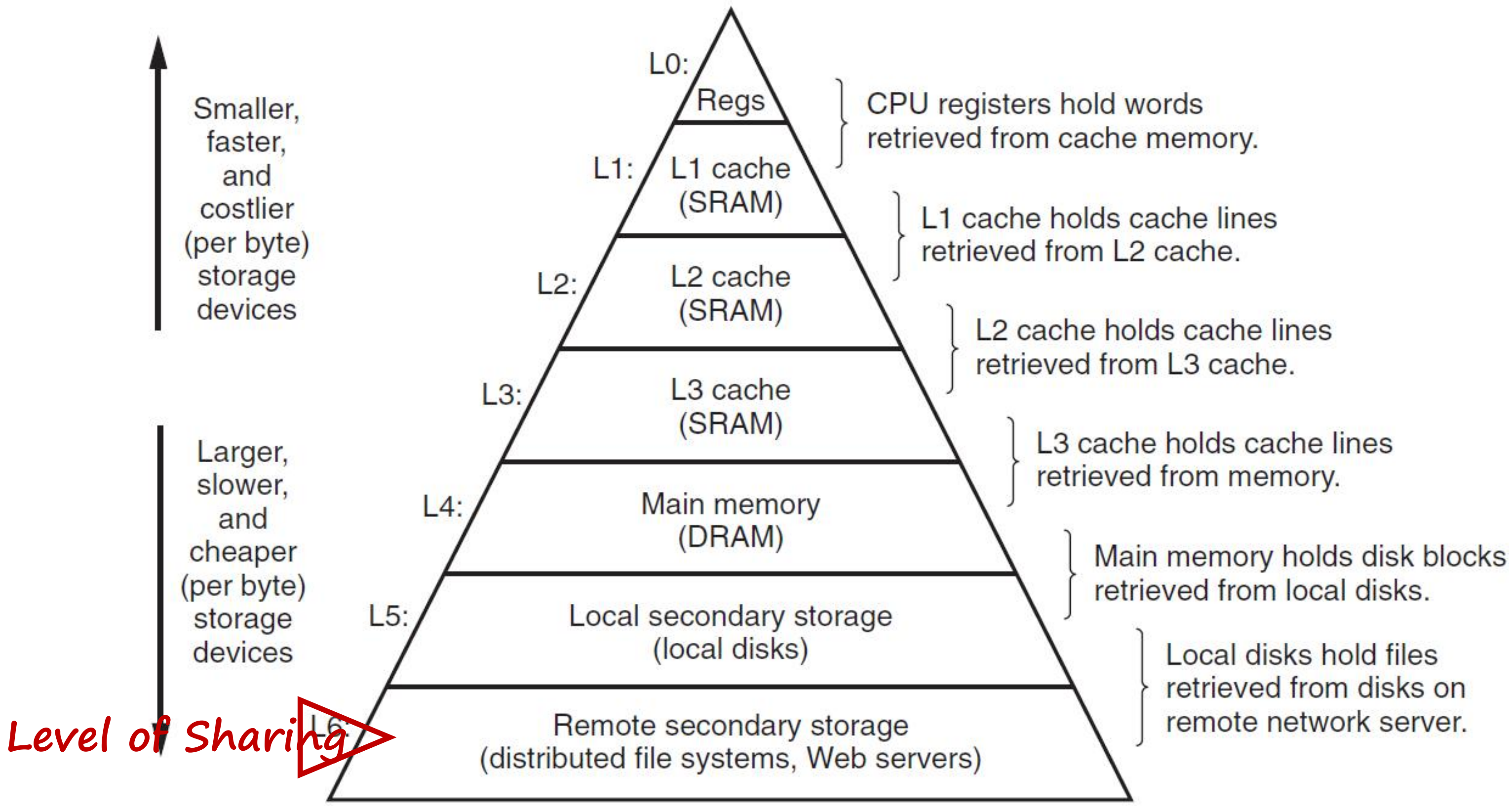
Shared Memory

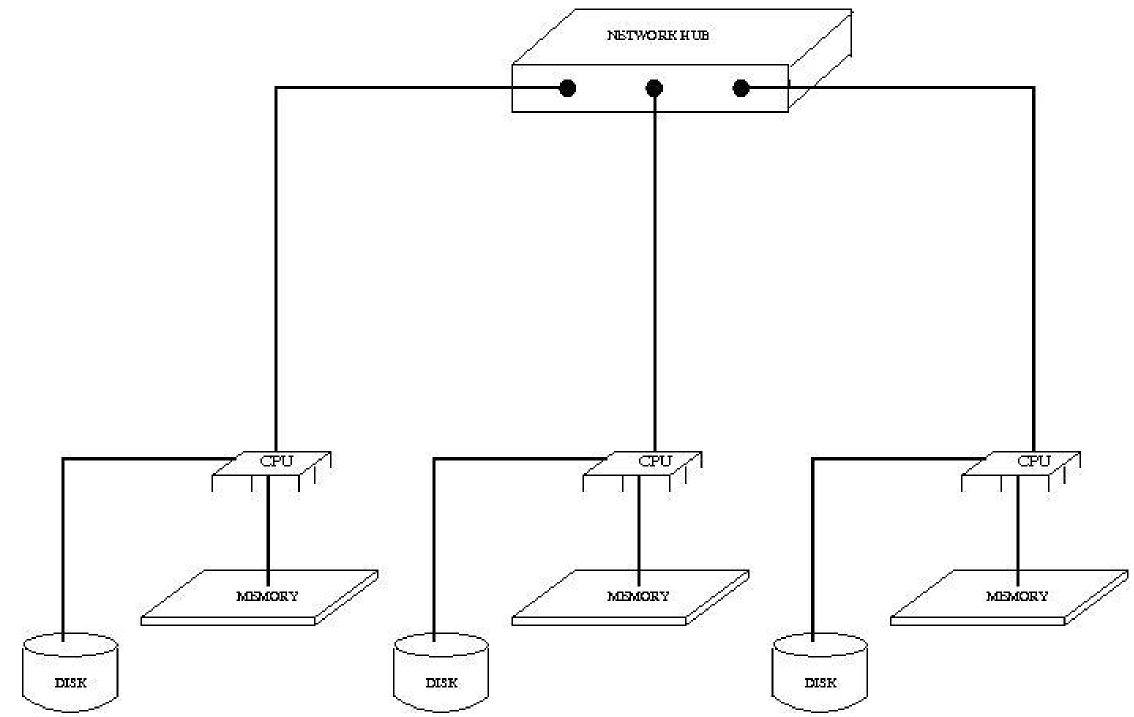




- Pros: Parallelism is easy to implement because the data synchronization/messaging process is implicit
  Programming Model: OpenMP
  Cons: There are still scalability problems, and the number of CPUs cannot be too large

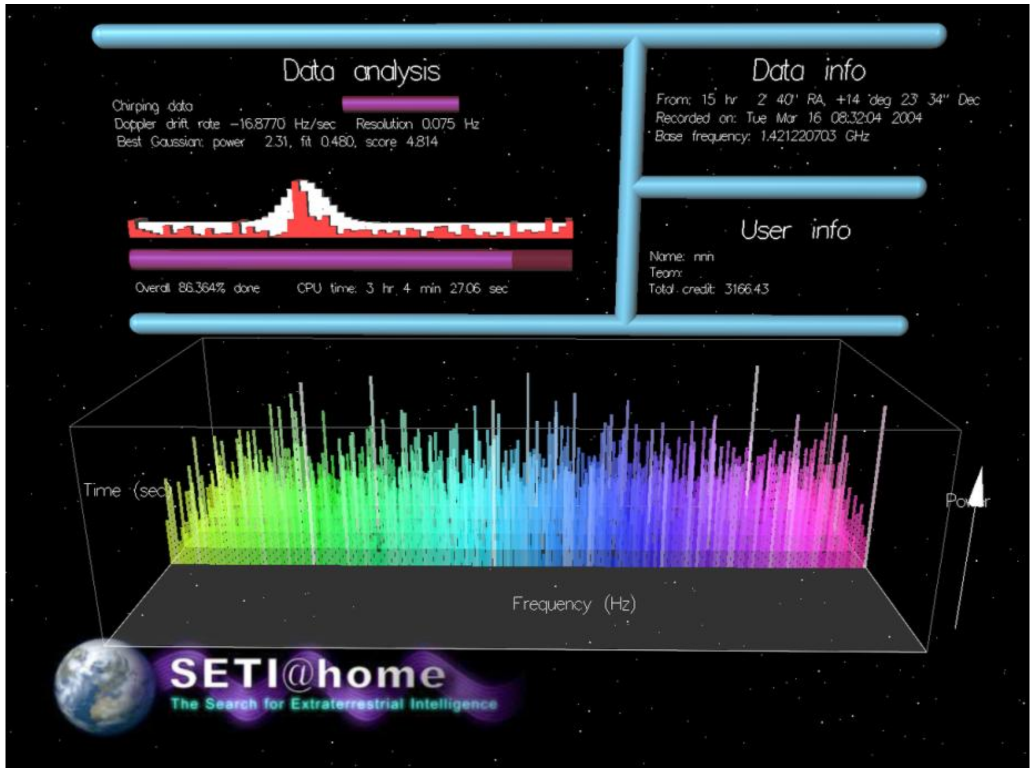
Distributed Memory





- Advantages: Can scale to very large scales, but the network topology has a large impact on performance
  Programming Model: MPI
- Disadvantages: Messaging must be handled explicitly, with the understanding of network topologies; complex to develop

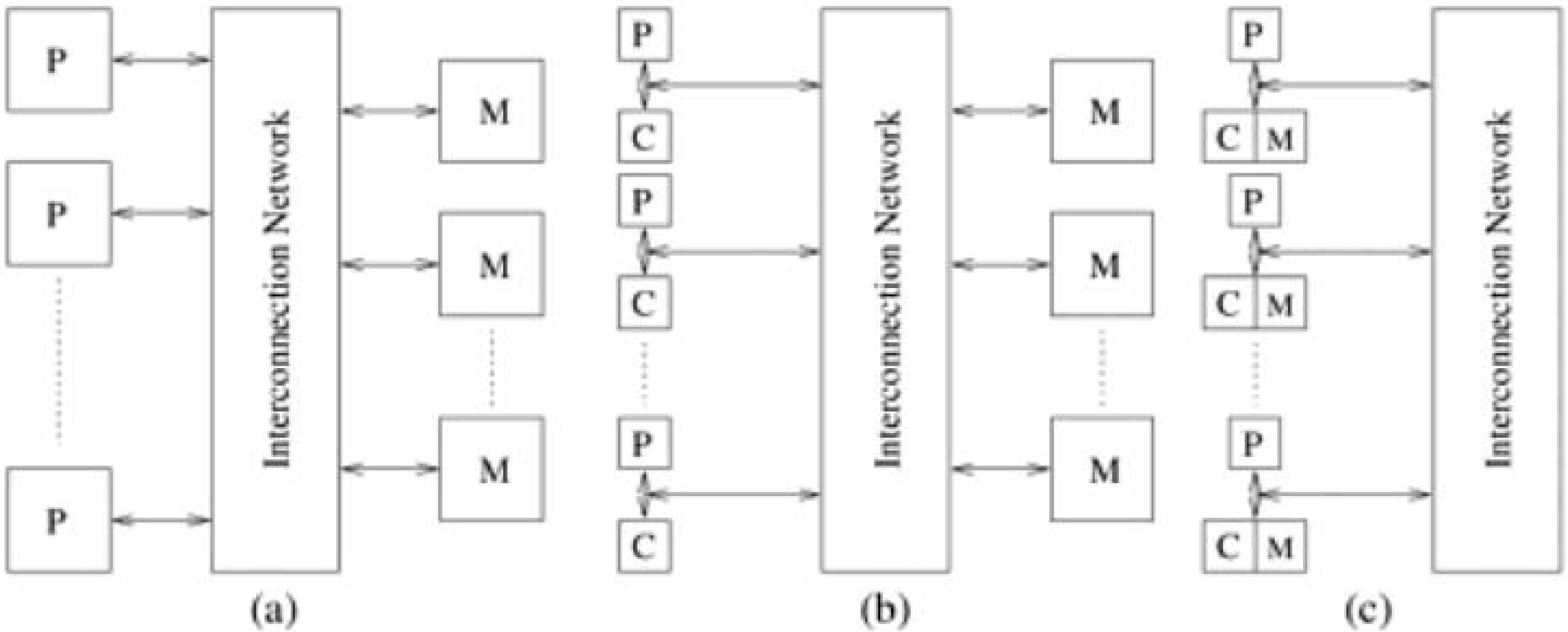
Example: the ‘Slowest’ HPC



Search Alien Civilization, using volunteer PCs on the internet

- Found nothing…
  - Fermi Paradox

Q: Can we have a global memory model on clusters?



It is possible, yet not very popular:

•A) UMA (Uniform memory access)

•B) COMA (Cache-only memory architecture)

•C) NUMA (Non-uniform memory access)

•--> Easy to program, but not easy to tune

Memory Array

Example of clusters

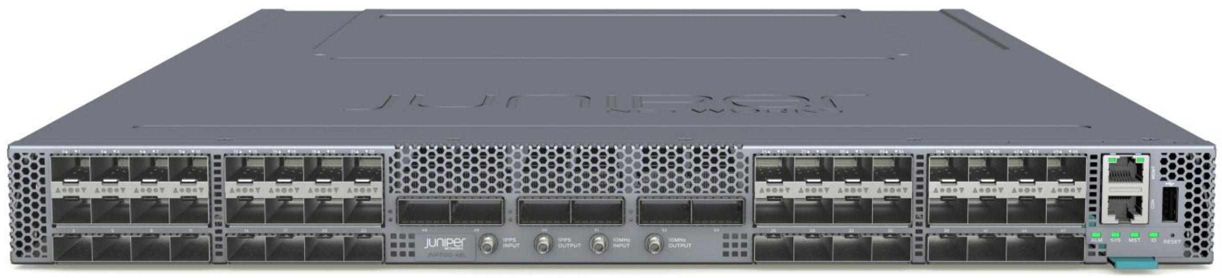
Cluster: tens of servers

Supercomputer:
hundreds of servers

Datacenter: thousands of servers

- To balance between cost and performance:
  - Generally, within a single node, a shared memory is used; Between nodes, networks are used

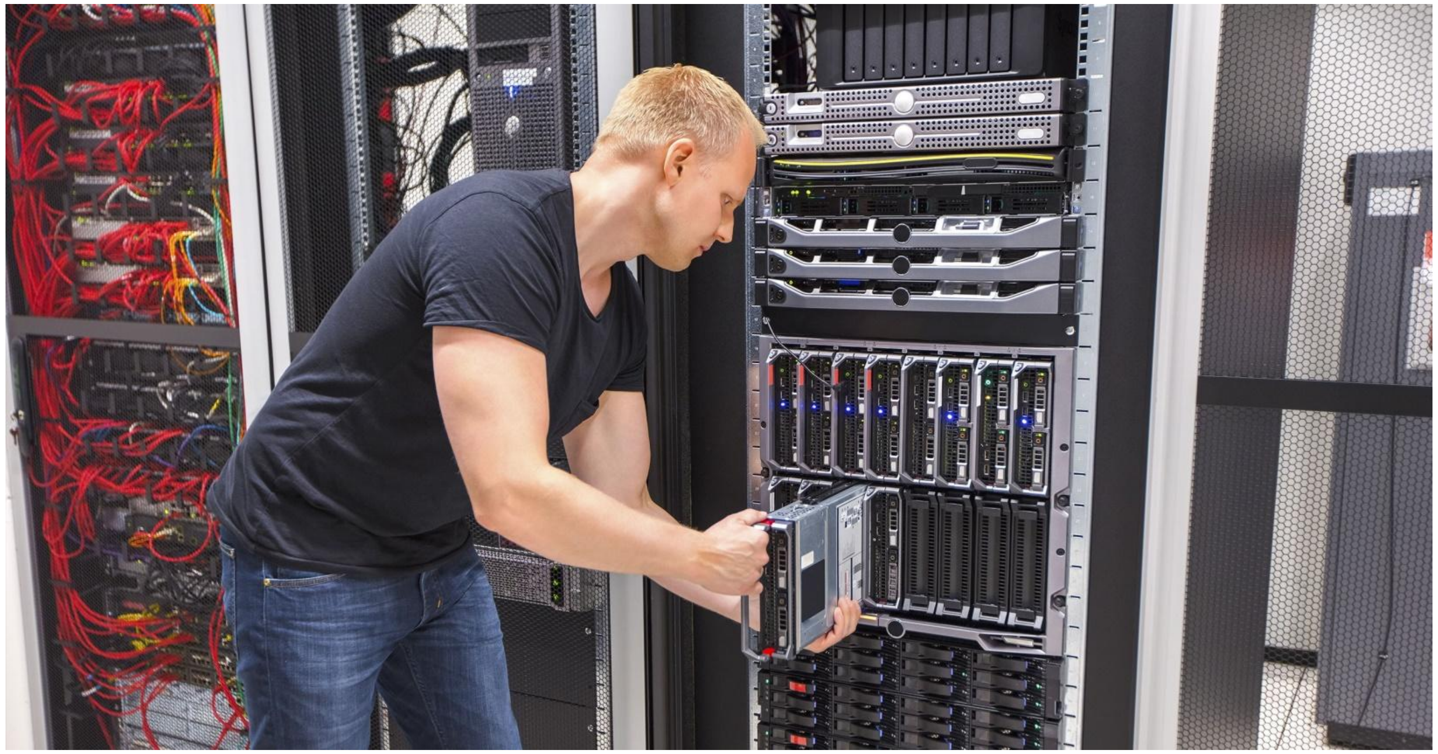
Interlude: appearance



Network Router

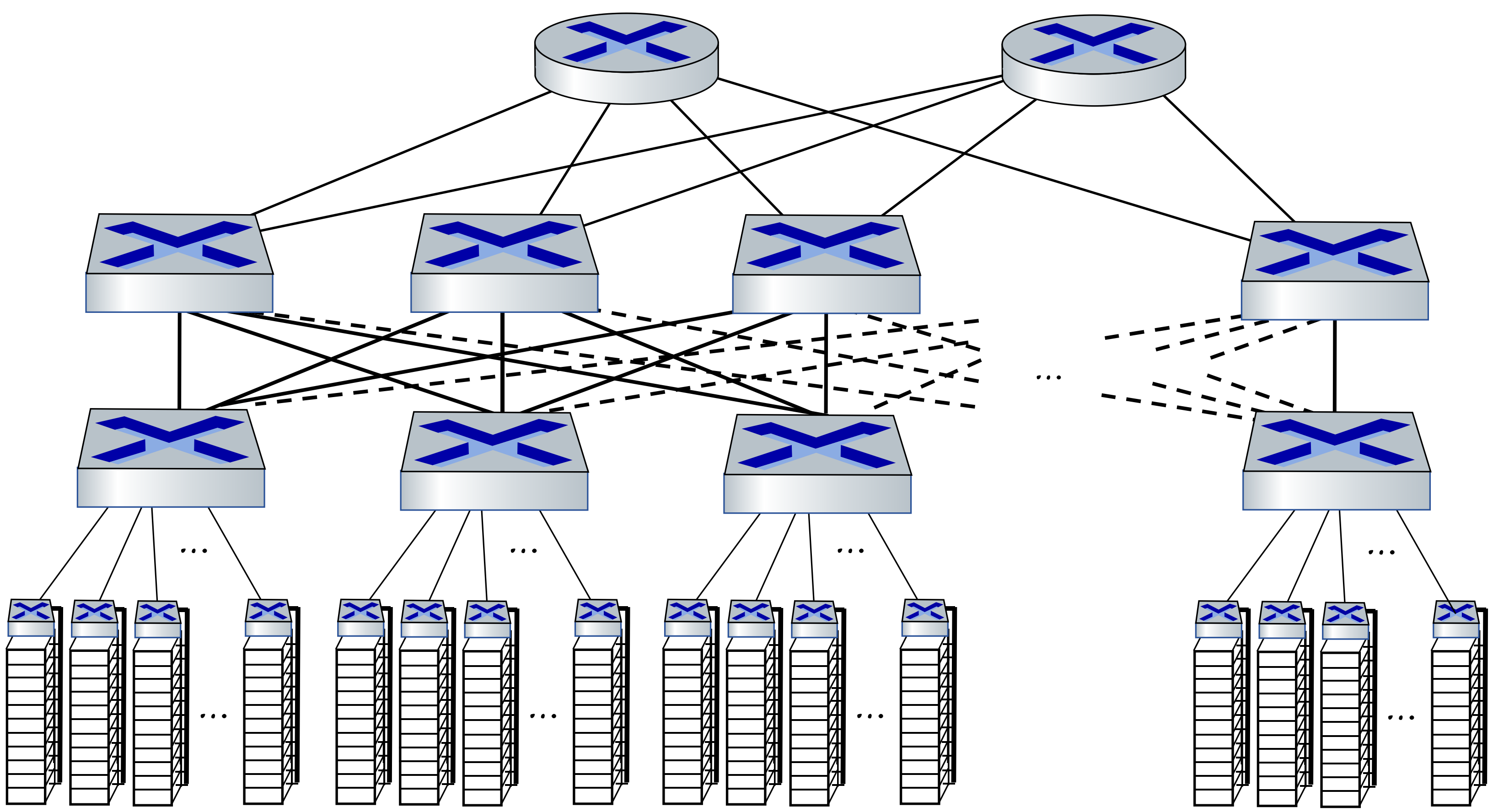
Storage

Server



- Why do all these devices look similar?
  - Easy to install uniformly on the rack!

Datacenter networks: elements



Border routers

- connections outside datacenter

Tier-1 switches

- connecting to ~16 T-2s below

Tier-2 switches

- connecting to ~16 TORs below

Top of Rack (TOR) switch

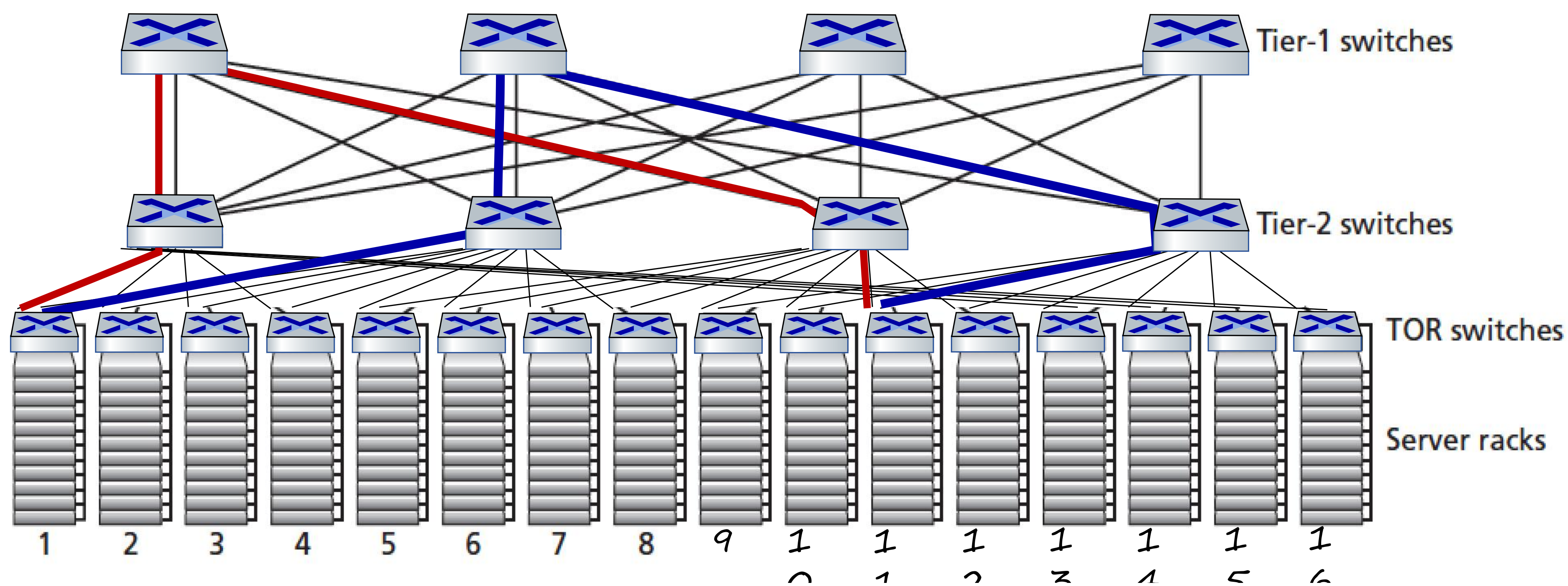
- one per rack
- 40-100Gbps Ethernet to blades

Server racks

- 20- 40 server blades: hosts

Datacenter networks: multipath

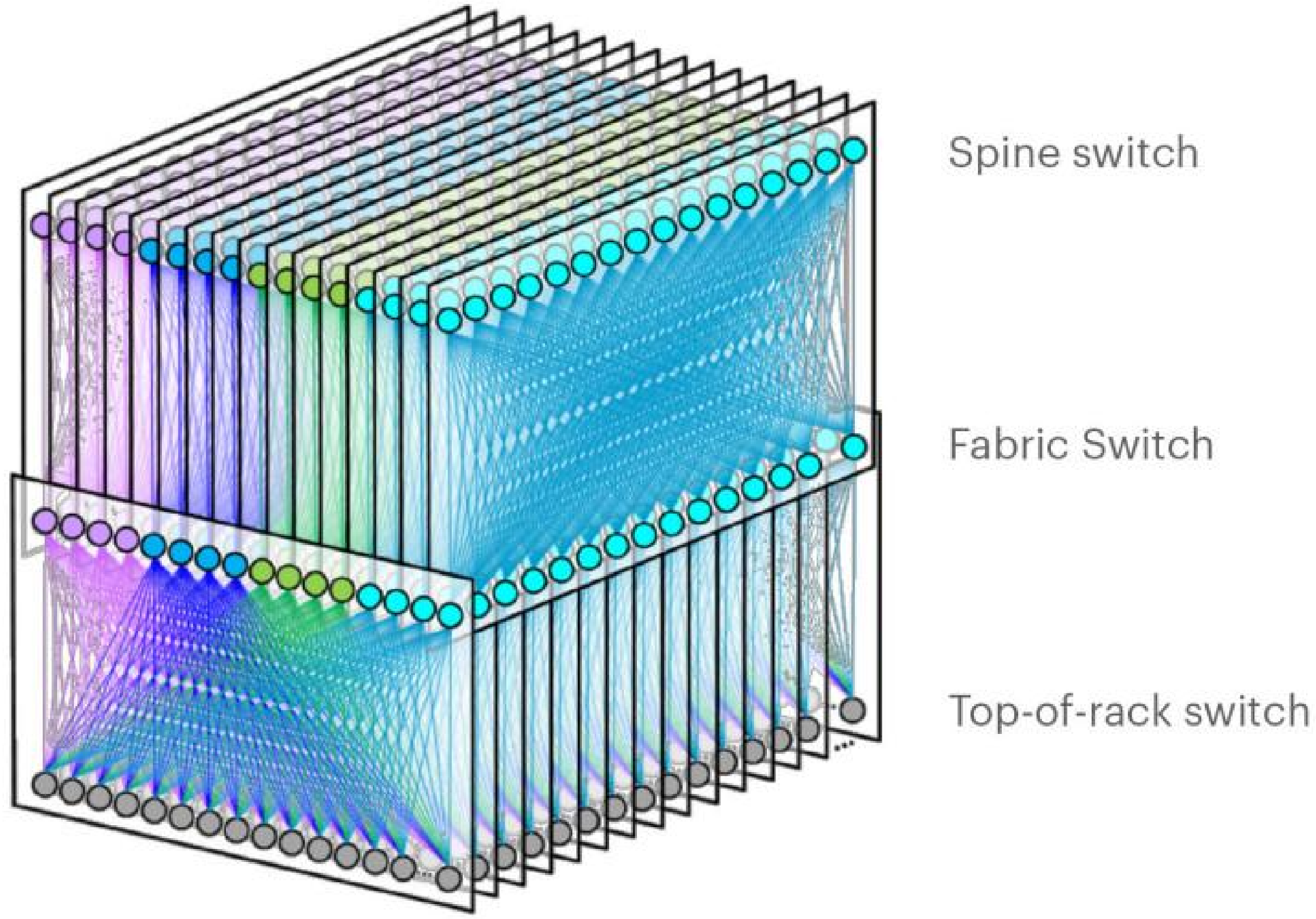
- rich interconnection among switches, racks:
  - increased throughput between racks (multiple routing paths possible)
  - increased reliability via redundancy



two *disjoint* paths highlighted between racks 1 and 11

Example

Facebook F16 data center network topology:



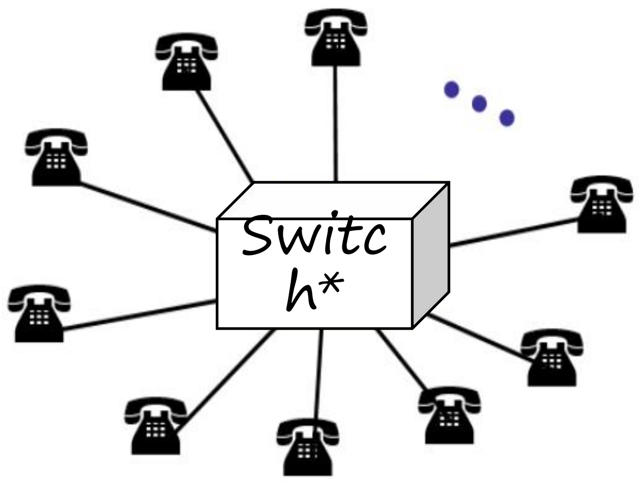
https://engineering.fb.com/data-center-engineering/f16-minipack/ (posted 3/2019)

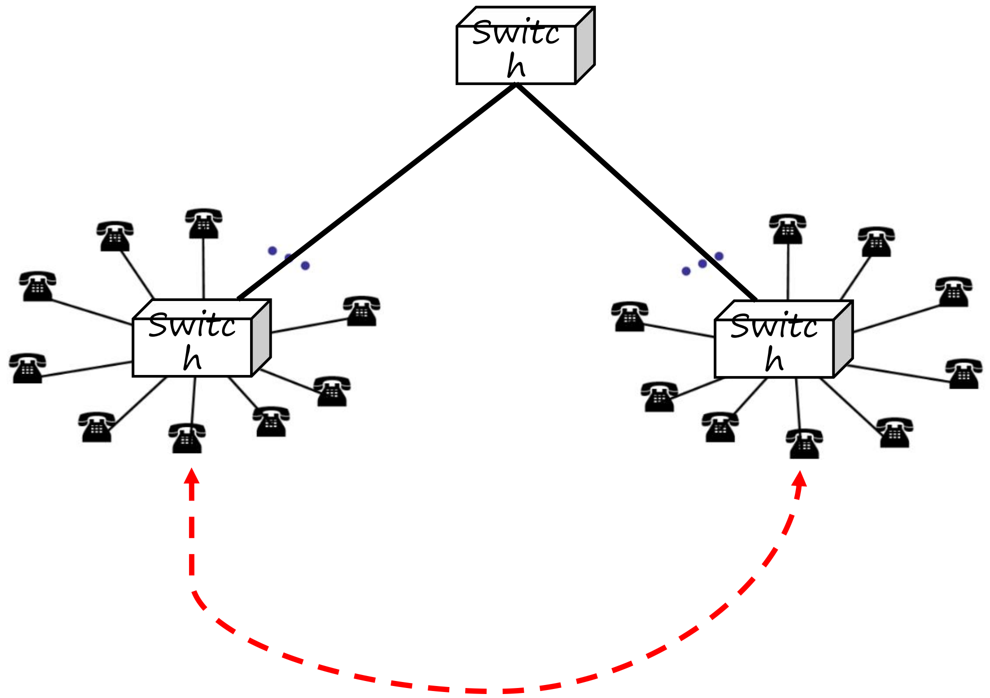
目录

Xi'an Jiaotong-Liverpool University 西交利物浦大学

- Network topology

Topology of usual networks

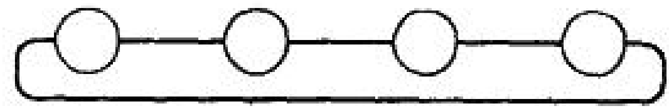




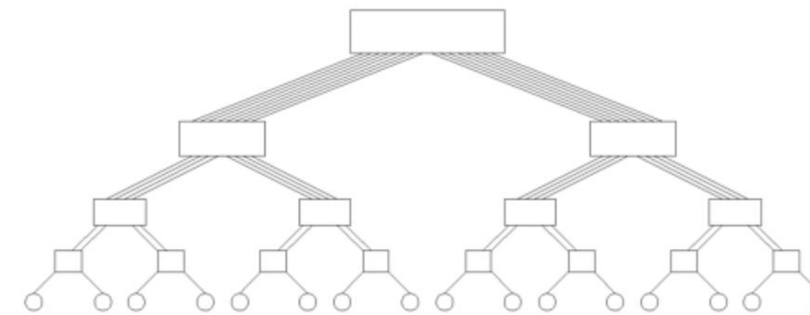
- Topology
  - describes how elements of a set relate spatially to each other (wiki)
- Star: The simplest structure (* -- linked by a switch with an internal bus)
- Tree: Direct contact between two leaves will be slow. For general needs, such as surfing Web, direct contact is rarely required, but it is not for HPC.

Other possible network structures

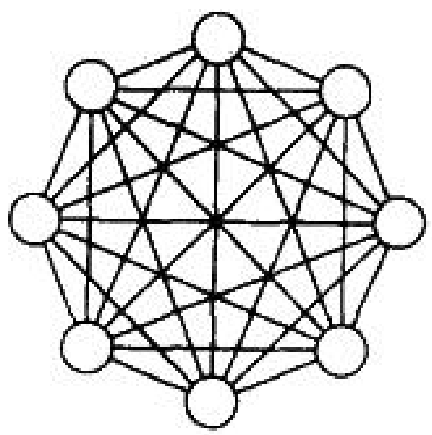
Linear



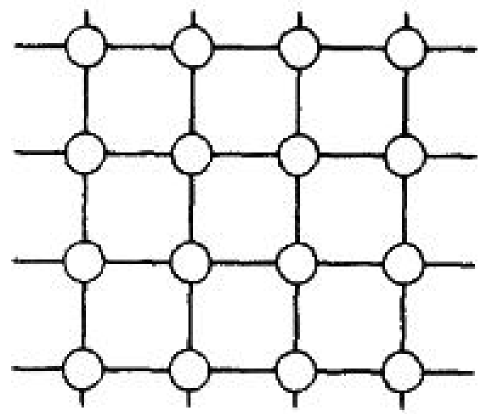
Loop



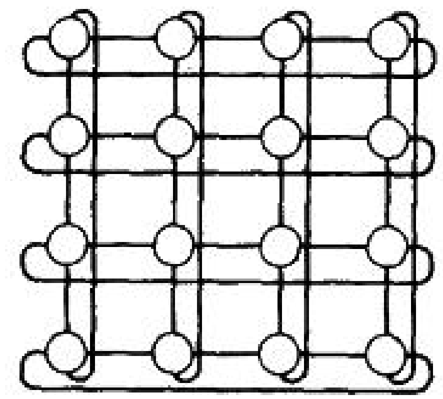
Fat Tree



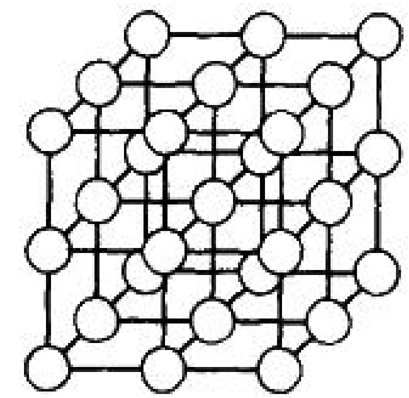
Fully Connected



2-D mesh



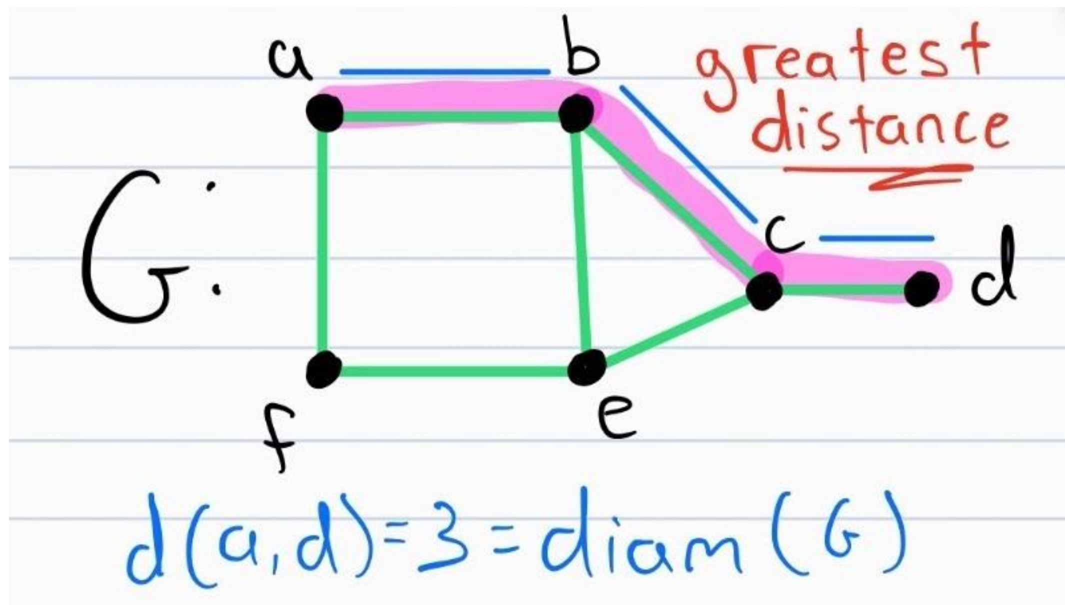
2-D wraparound mesh



3-D mesh

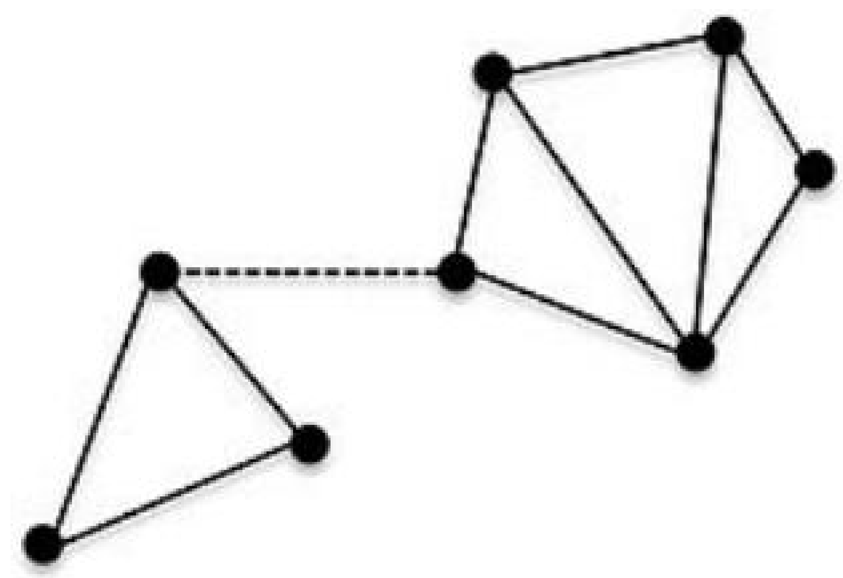
- Balance between cost and performance:
  - More connections always make communication faster
  - The higher the number of connections, the higher the cost
- Obviously, some structures look more “reasonable” and “connected” than others
  - How to quantify this concept?

Network performance metrics



- Network Diameter: The longest distance between any two nodes in a network.
  - The larger the diameter, the greater the hop count to complete the communication at the farthest two points, which can easily lead to high latency.

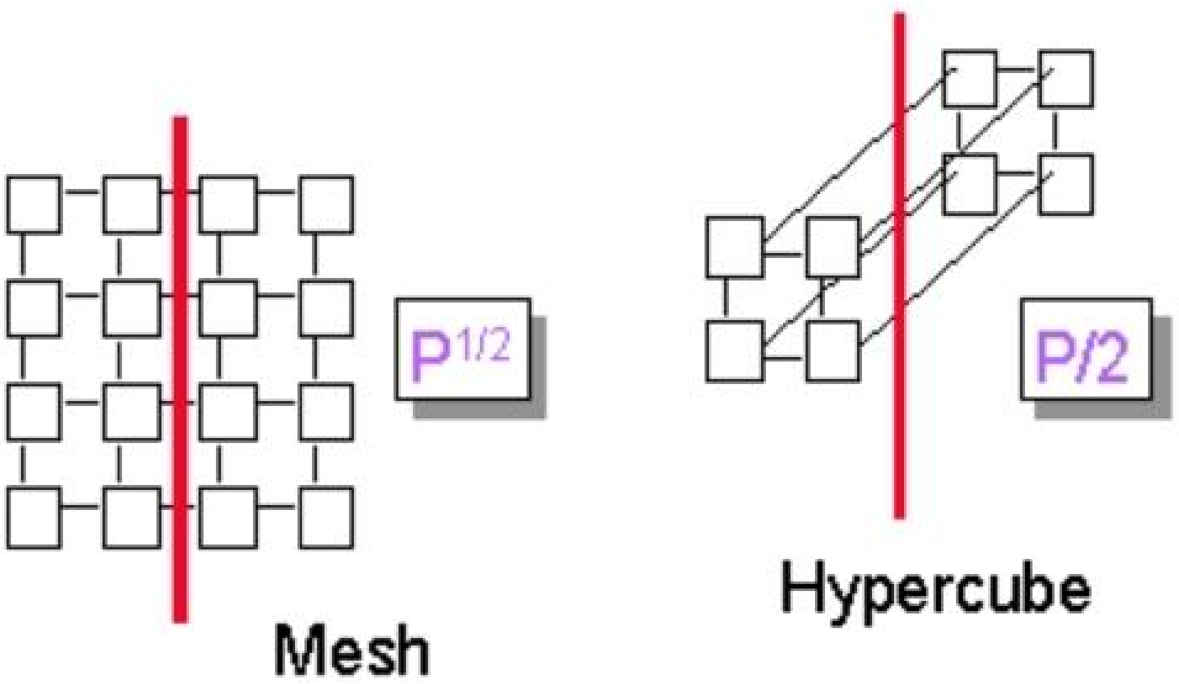
Network performance metrics



Conn=1

- Connectivity: The minimum number of arcs that need to be removed to divide a network into two non-connected networks.
  - Low connectivity, easy to form congestion in some paths, worsening the overall bandwidth and latency.

Network performance metrics



- Bisection Width: The minimum number of edges that must be removed for cutting network into two equal parts.
  - On average, it measures how easy it is to send data from one side to the other. This indicator relates to the effectiveness of global communications.

Features of some networks

	Minimize	*Maximize*	*Maximize*	*Minimize*
Network	**Diameter**	**Bisection Width**	**Arc Connectivity**	**Cost (No. of links)**
Completely-connected	1	$p^2/4$	$p - 1$	$p(p - 1)/2$
Star	2	1	1	$p - 1$
Complete binary tree	$2 \log((p + 1)/2)$	1	1	$p - 1$
Linear array	$p - 1$	1	1	$p - 1$
2-D mesh, no wraparound	$2(\sqrt{p} - 1)$	$\sqrt{p}$	2	$2(p - \sqrt{p})$
$\boxed{\text{2-D wraparound mesh}}$	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	2p

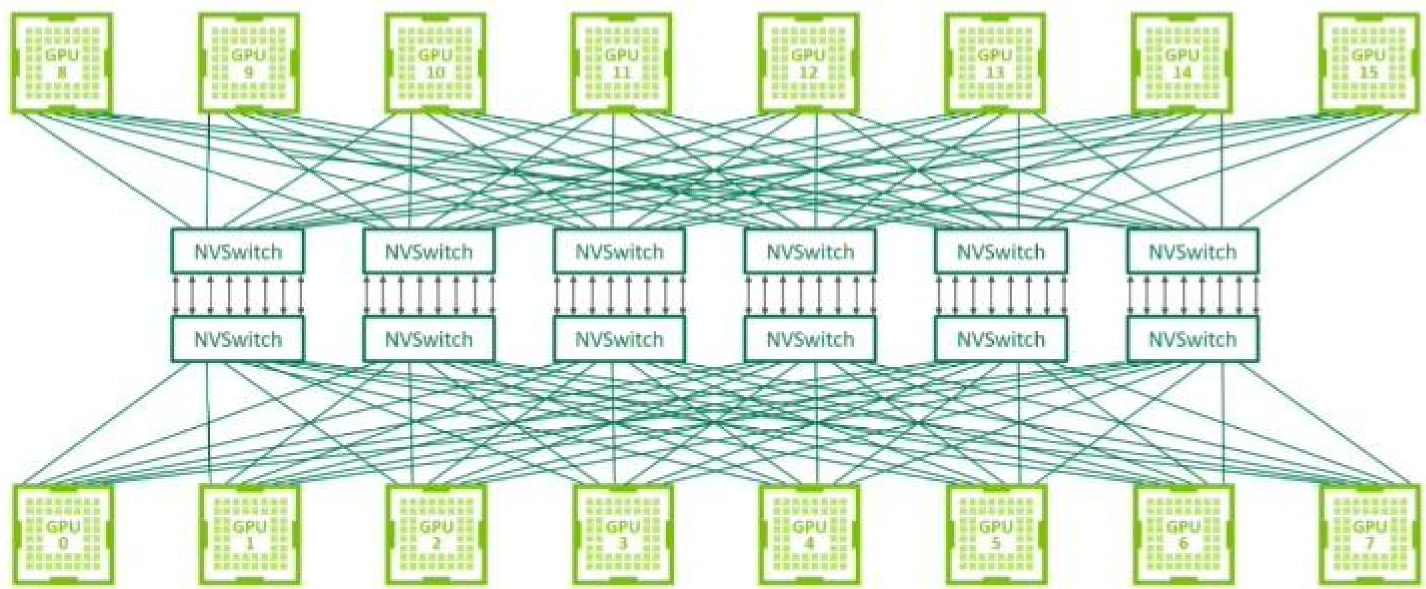
Looks better!

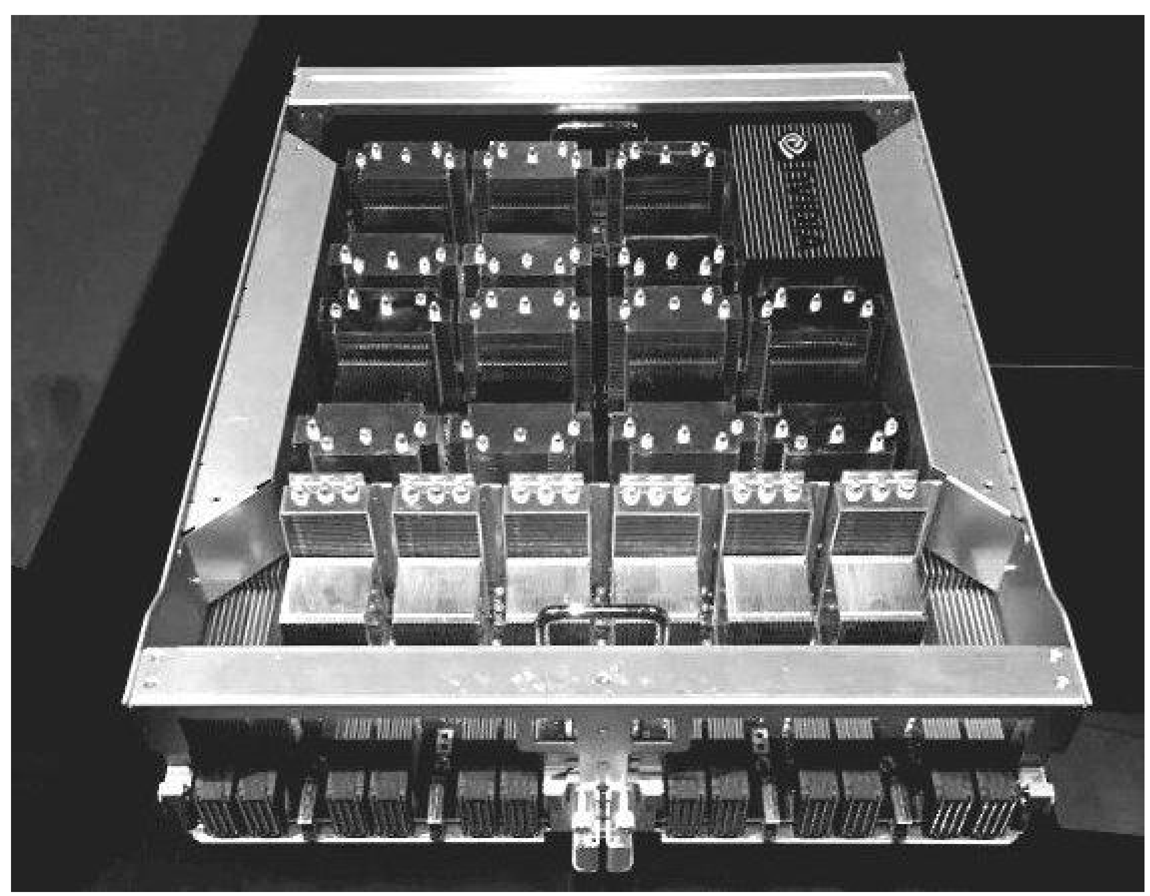
A summary of the characteristics of various static network topologies connecting p nodes.

- The key of design lies in finding the optimal connection under the constraint of cost.

Example: Fat-Tree

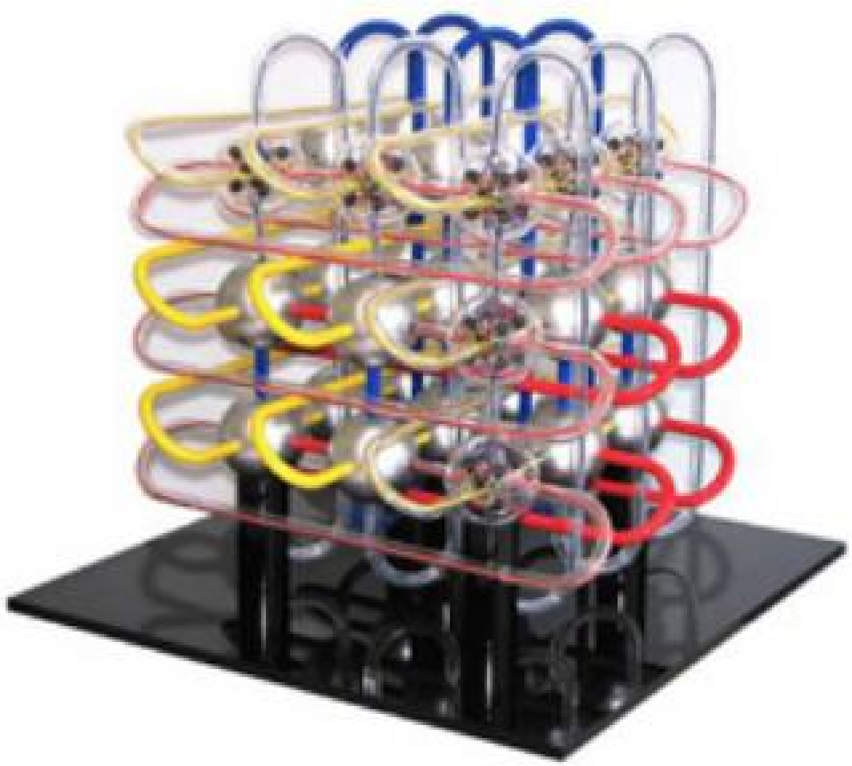
- GPU Server: NVIDIA'S DGX-2 SYSTEM PACKS





Example: 6D wraparound mesh

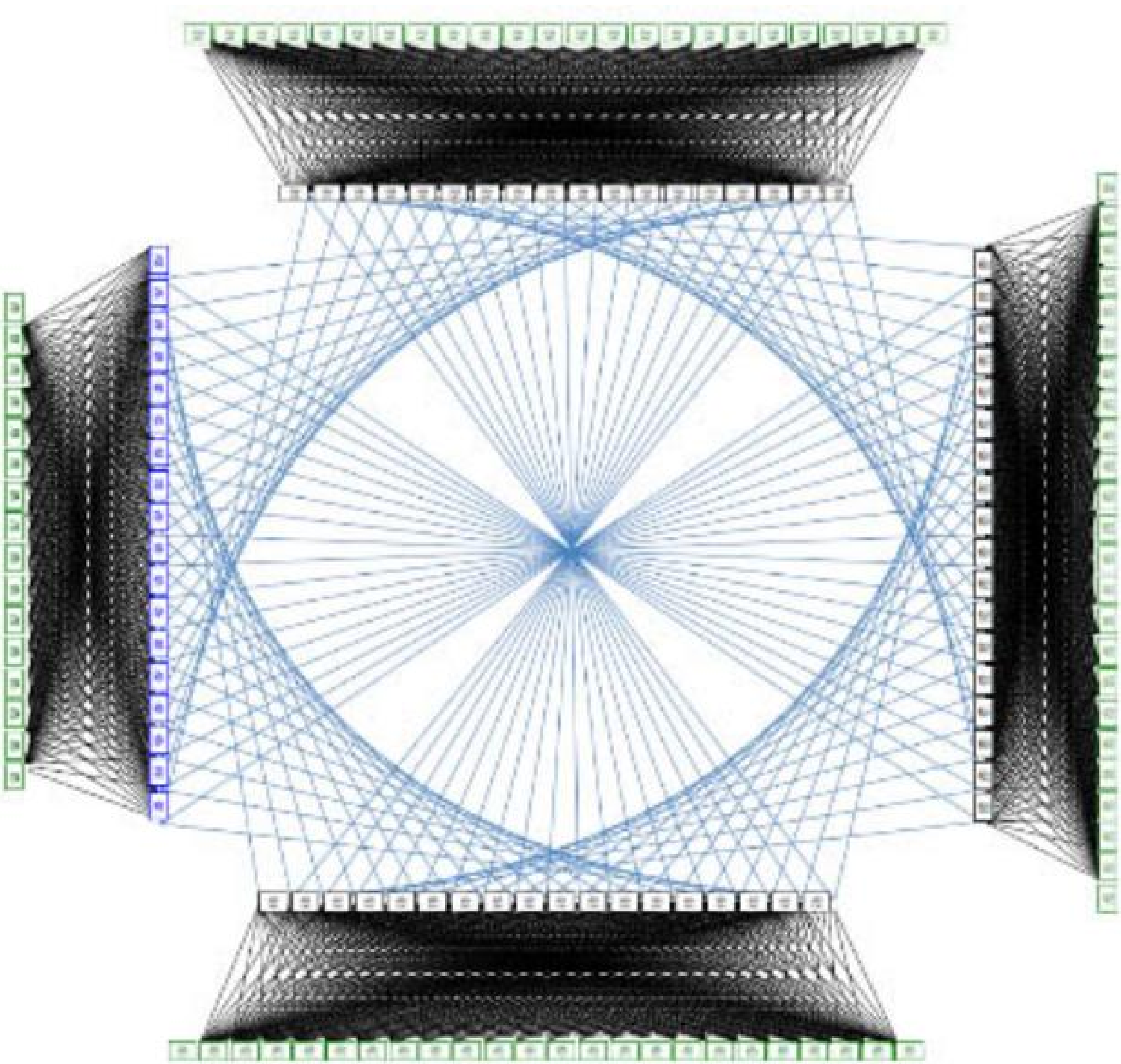
- K-computer
  - 80,000 compute nodes; 640,000 cores



"6-dimensional mesh/torus" topology (model)

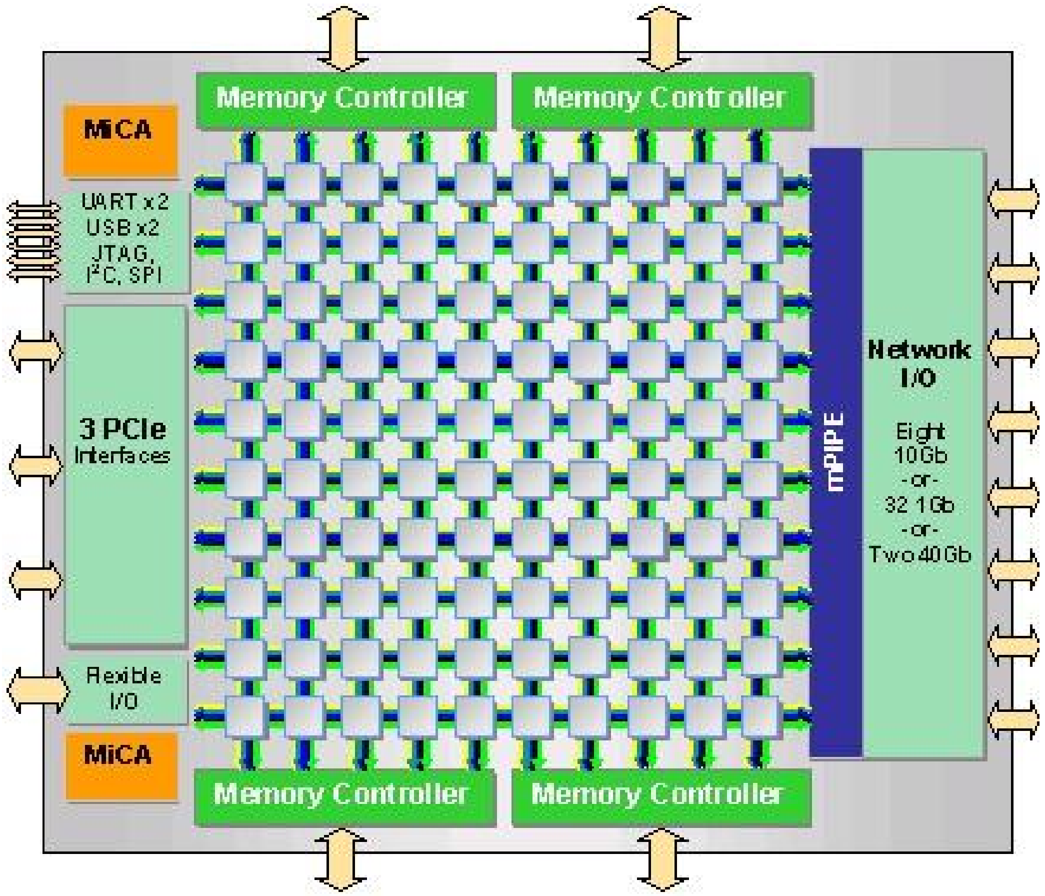
Example: Dragonfly: interconnected fat-trees

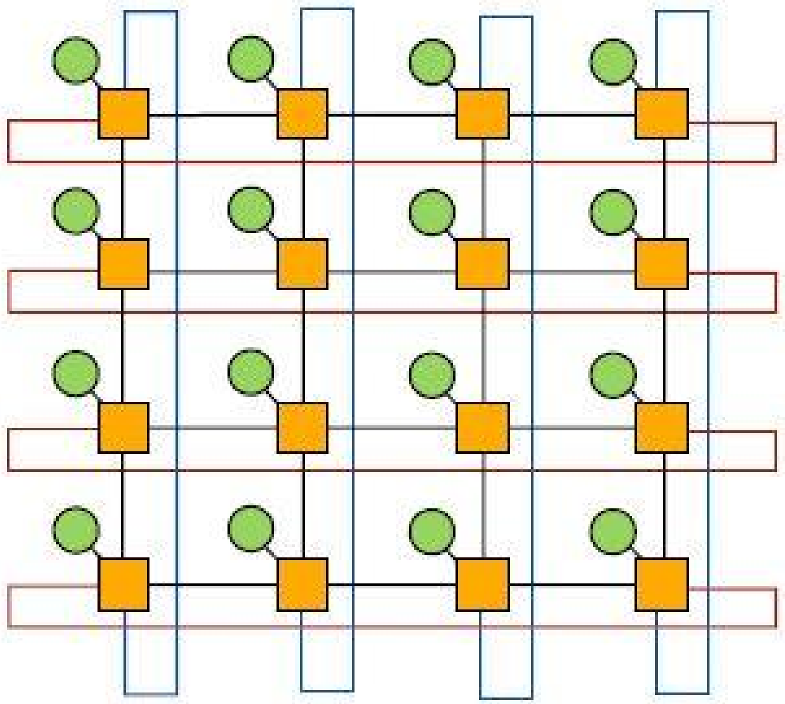
- Niagara supercomputer



Example: 2D wraparound mesh

- Many-core CPU: Tilera TILE-Gx, consists of a mesh network of up to 100 cores

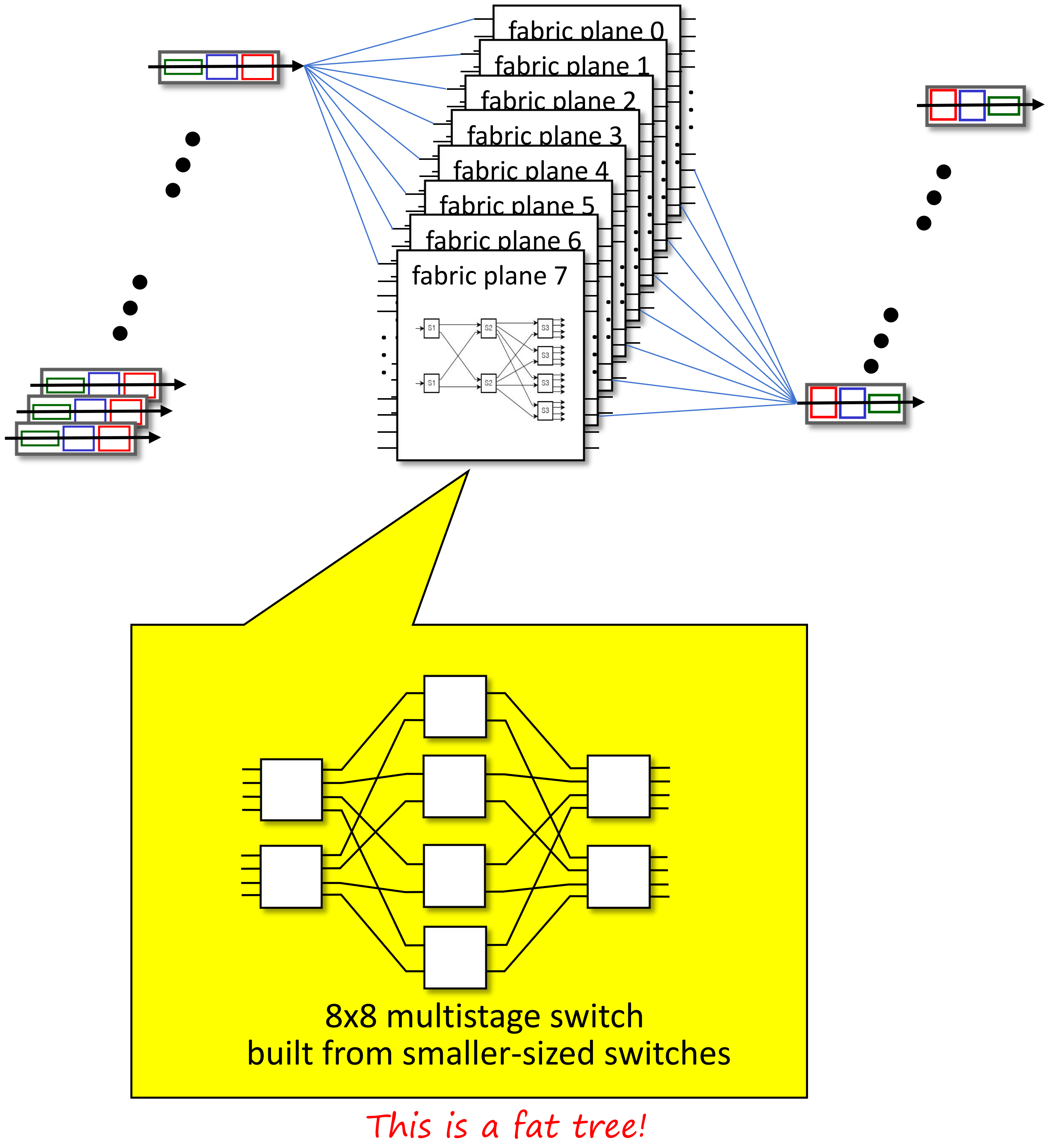




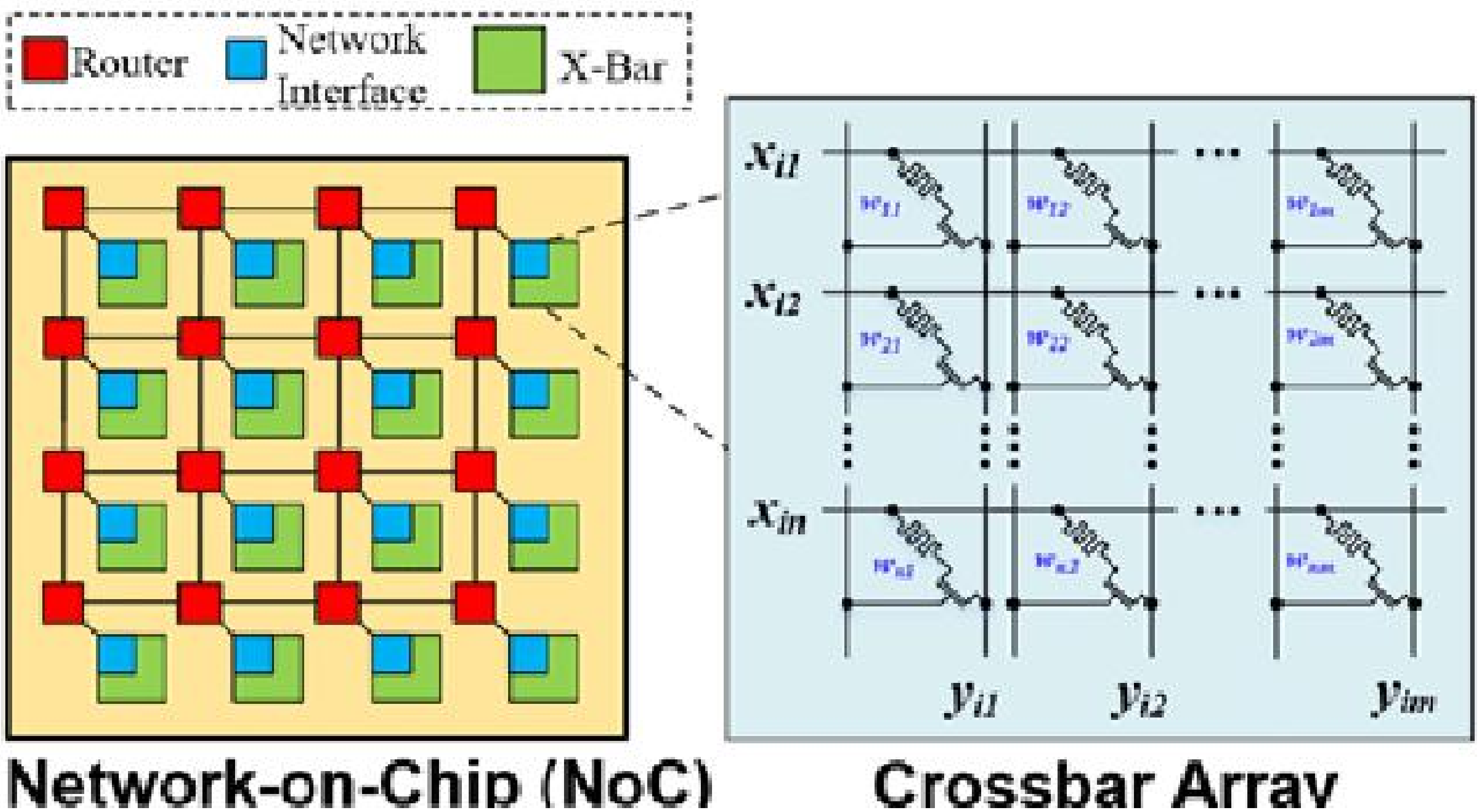
2D Torus

Example: Multiple Fat tree

- Cisco CRS router: up to 100's Tbps switching capacity



Further Reading：Dynamic networking



- Dynamic networking
  - Switching the connection structure takes time, but it can be fully adapted to the needs of algorithms
  - Usually more expensive, and less scalable

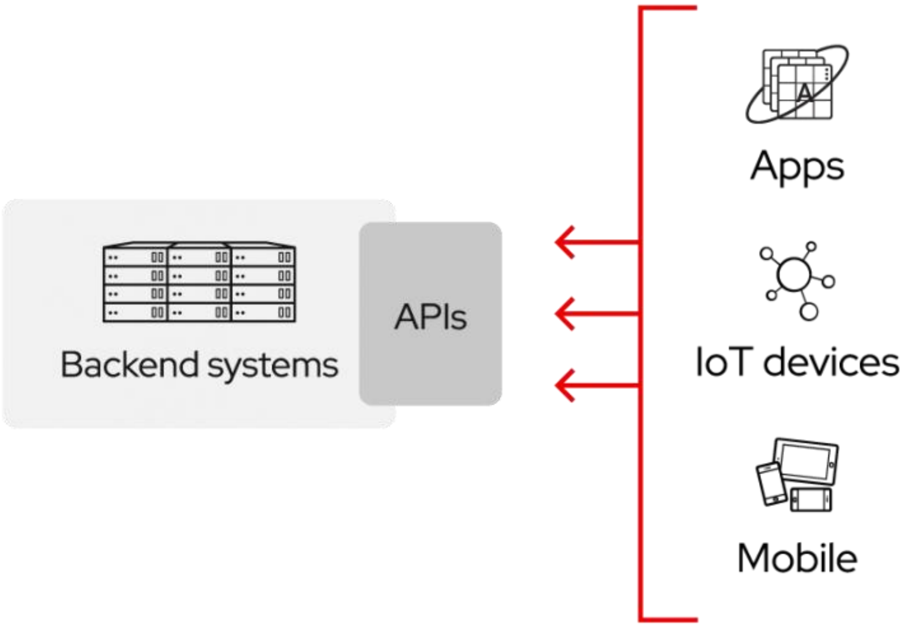
Section

Xi'an Jiaotong-Liverpool University 西交利物浦大学

- Review of MPI

The MPI is an API

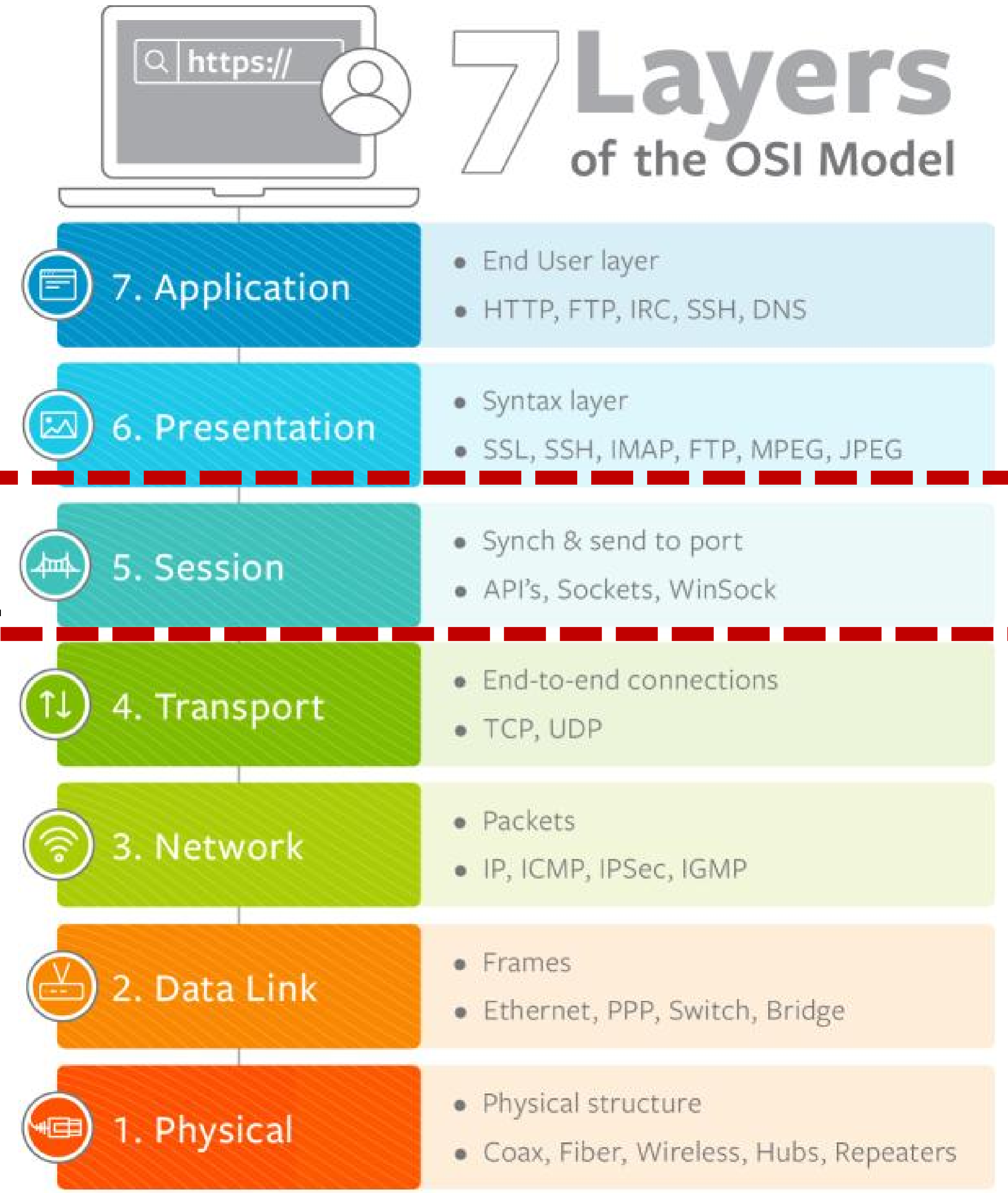
- API (**A**pplication **P**rogramming **I**nterface)
  - An API is a way for two or more computer programs to communicate with each other.
  - A document or standard that describes how to build or use such a connection or interface is called an API specification.
  - A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation.



MPI

Xi'an Jiaotong-Liverpool University 西交利物浦大学

- Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures.
- Although MPI belongs in layers 5 of the OSI Reference Model, implementations may cover most layers. If without support from hardware, the socket programming can be used to implement that.
- MPI is language-independent, portable and light-weighted.
- Currently we use MPI-3.1 (published in 2015)
  and its Python library: MPI4py

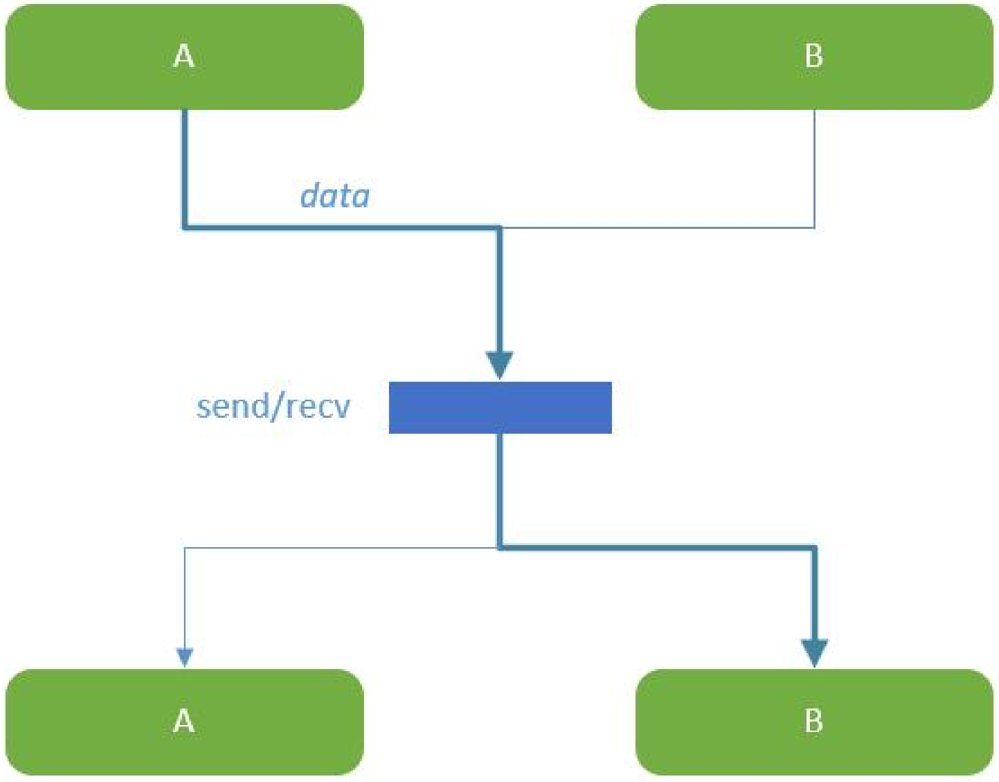


The core functions of MPI

- Communication
  - One-to-one (blocking/non-blocking) – This is the only part supported directly by the socket programming
    Collective
    - Global communication, including broadcast, gather, and scatter;
      Global reduction, including sum, max, min, etc.
      Synchronization, including Barriers, etc.;
  - One-Side
- Topology
  Process-hardware mapping relationship

The concerns of MPI: on the one hand, to meet the algorithms' requirements, on the other hand, to give hardware the chance of optimization.

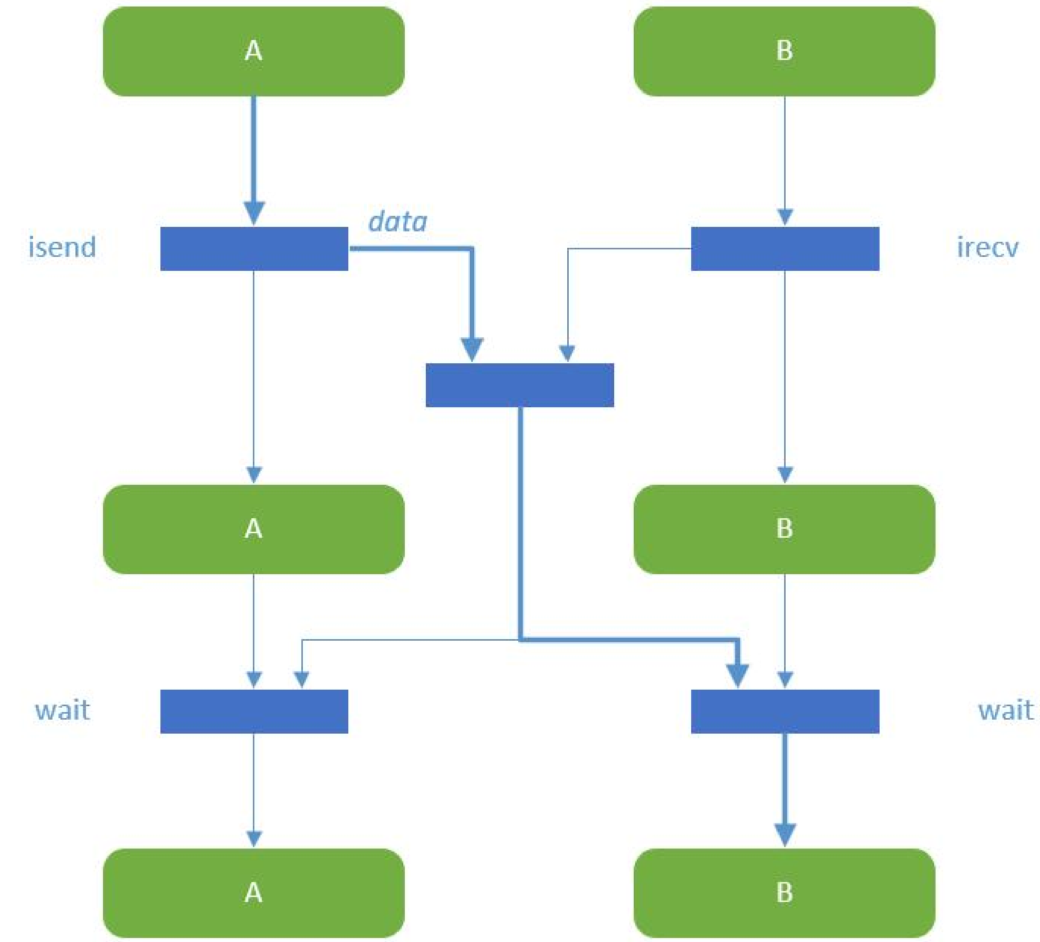
Point-to-point (blocking)




```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

- Note: *recv* can specify no source, that is, it can accept messages from any source

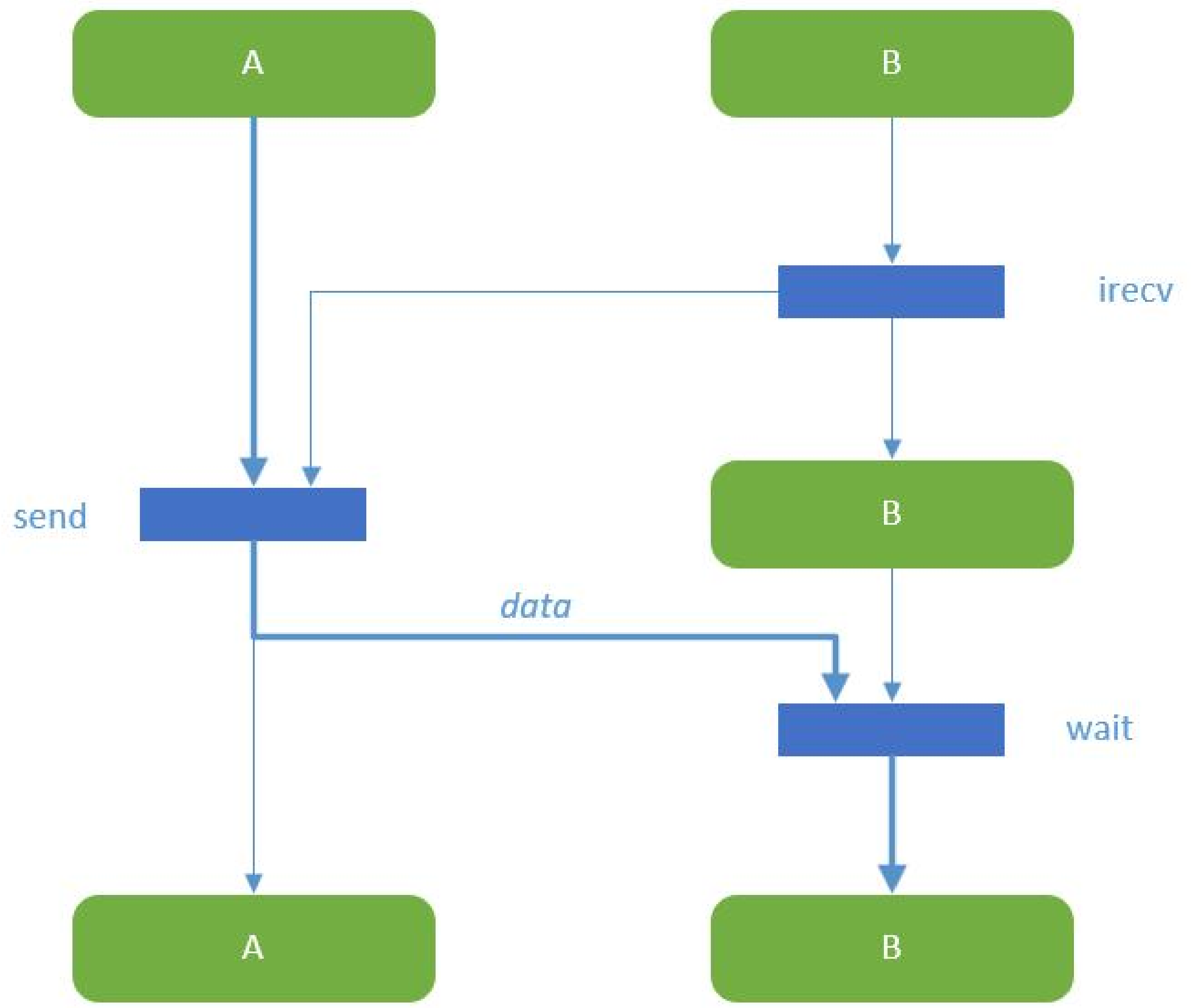
Point-to-point (non-blocking)



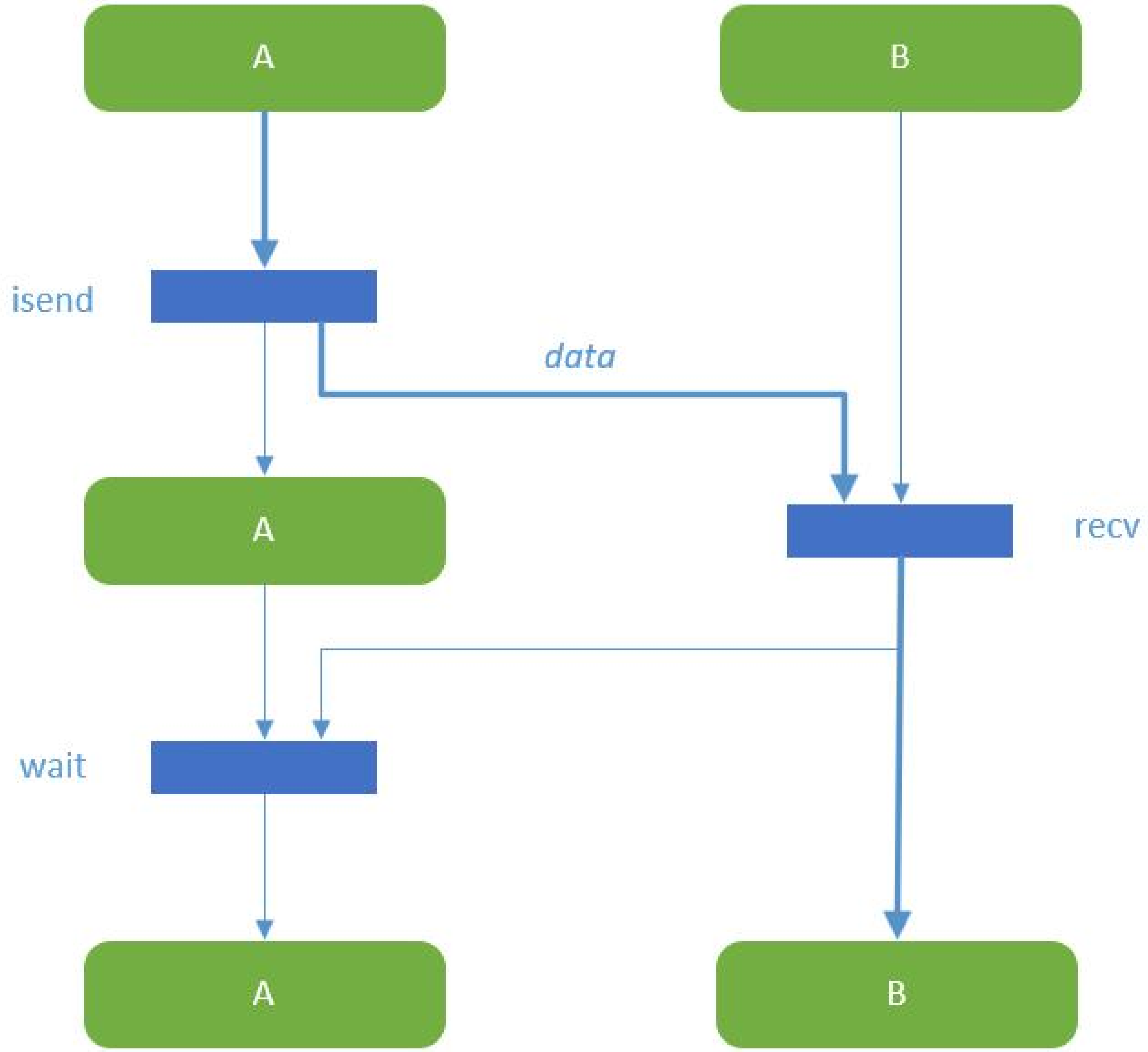

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

- Note: *test* can be used instead of *wait* to query the state of comm. without blocking

Blocking send + non-blocking receive



Non-blocking send + blocking receive



Broadcast

```python
# bcast.py

from mpi4py import MPI


comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ( 'abc', 'xyz')}
    print 'before broadcasting: process %d has %s' % (rank, data)
else:
    data = None
    print 'before broadcasting: process %d has %s' % (rank, data)

data = comm.bcast(data, root=0)
print 'after broadcasting: process %d has %s' % (rank, data)
```

```
$ mpiexec -n 2 python bcast.py
before broadcasting: process 0 has {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
after broadcasting: process 0 has {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
before broadcasting: process 1 has None
after broadcasting: process 1 has {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
```

- A broadcast operation copies data from the root process to all other processes in the same group.
  Example: configurations, small inputs, …

Scatter

Xi'an Jiaotong-Liverpool University 西交利物浦大学

```
# scatter.py

from mpi4py import MPI


comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [ (i + 1)**2 for i in range(size) ]
    print 'before scattering: process %d has %s' % (rank, data)
else:
    data = None
    print 'before scattering: process %d has %s' % (rank, data)

data = comm.scatter(data, root=0)
print 'after scattering: process %d has %s' % (rank, data)
```

```
$ mpiexec -n 3 python scatter.py
before scattering: process 0 has [1, 4, 9]
after scattering: process 0 has 1
before scattering: process 1 has None
after scattering: process 1 has 4
before scattering: process 2 has None
after scattering: process 2 has 9
```

- *Scatter disperses different messages from the root process to other processes in the group.*
- *Example: large inputs*

Gather

Xi'an Jiaotong-Liverpool University 西交利物浦大学

```python
# gather.py

from mpi4py import MPI


comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

data = (rank + 1)**2
print 'before gathering: process %d has %s' % (rank, data)

data = comm.gather(data, root=0)
print 'after scattering: process %d has %s' % (rank, data)
```

```
$ mpiexec -n 3 python gather.py
before gathering: process 0 has 1
after scattering: process 0 has [1, 4, 9]
before gathering: process 1 has 4
after scattering: process 1 has None
before gathering: process 2 has 9
after scattering: process 2 has None
```

- *Gather* is the reverse of *Scatter*, where the root process collects different messages from other processes and puts them into its own receive buffer.
- Example: Intermediate results

Allgather

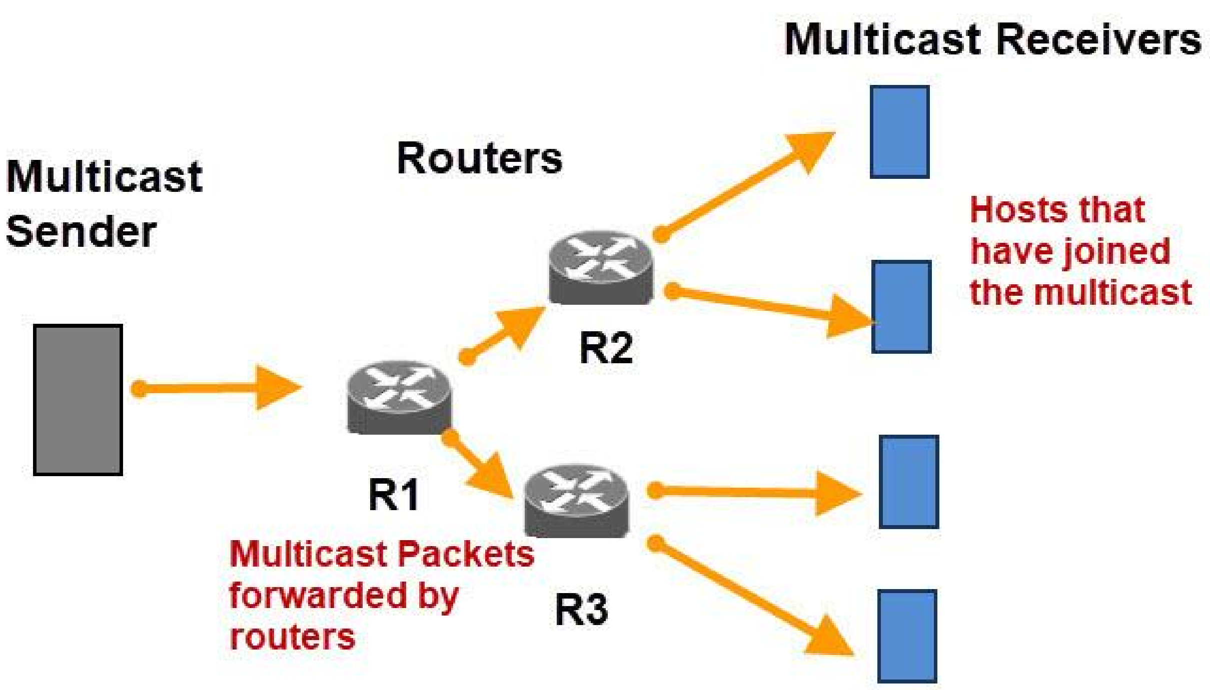

```
from mpi4py import MPI
import numpy

def matvec(comm, A, x):
    m = A.shape[0] # local rows
    p = comm.Get_size()
    xg = numpy.zeros(m*p, dtype='d')
    comm.Allgather([x,  MPI.DOUBLE],
                   [xg, MPI.DOUBLE])
    y = numpy.dot(A, xg)
    return y
```

Xg is a large vector residing on different nodes

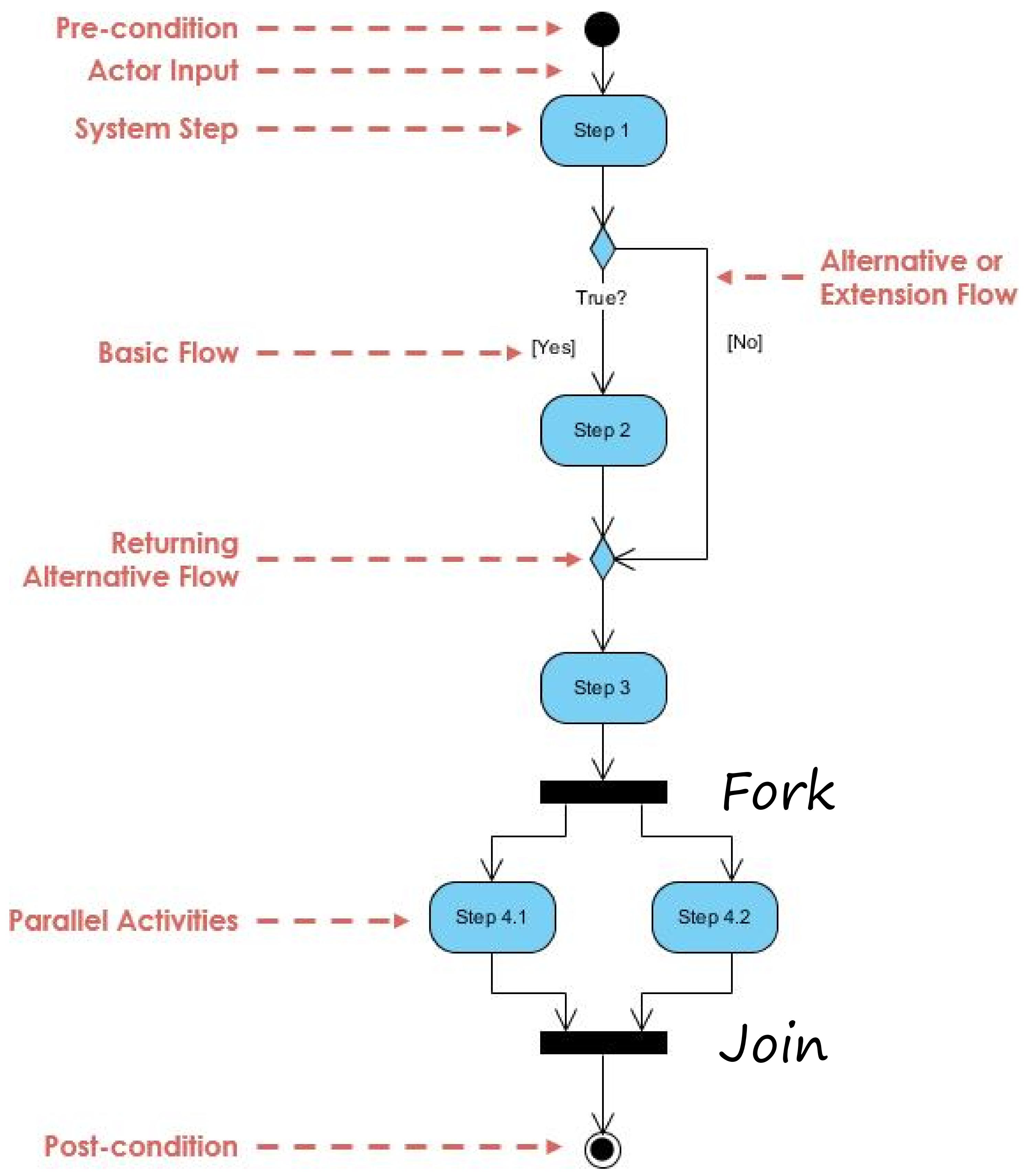
- It concatenates data from the send buffers of all processes in the group and sends them to the receive buffers of all processes.
- Example: concatenate local results

Hardware/protocol support

- Link Layer
  - If nodes happen to be connected to one branch of a tree, the broadcast message can be sent only once
- Network Layer
  - IP Multicast

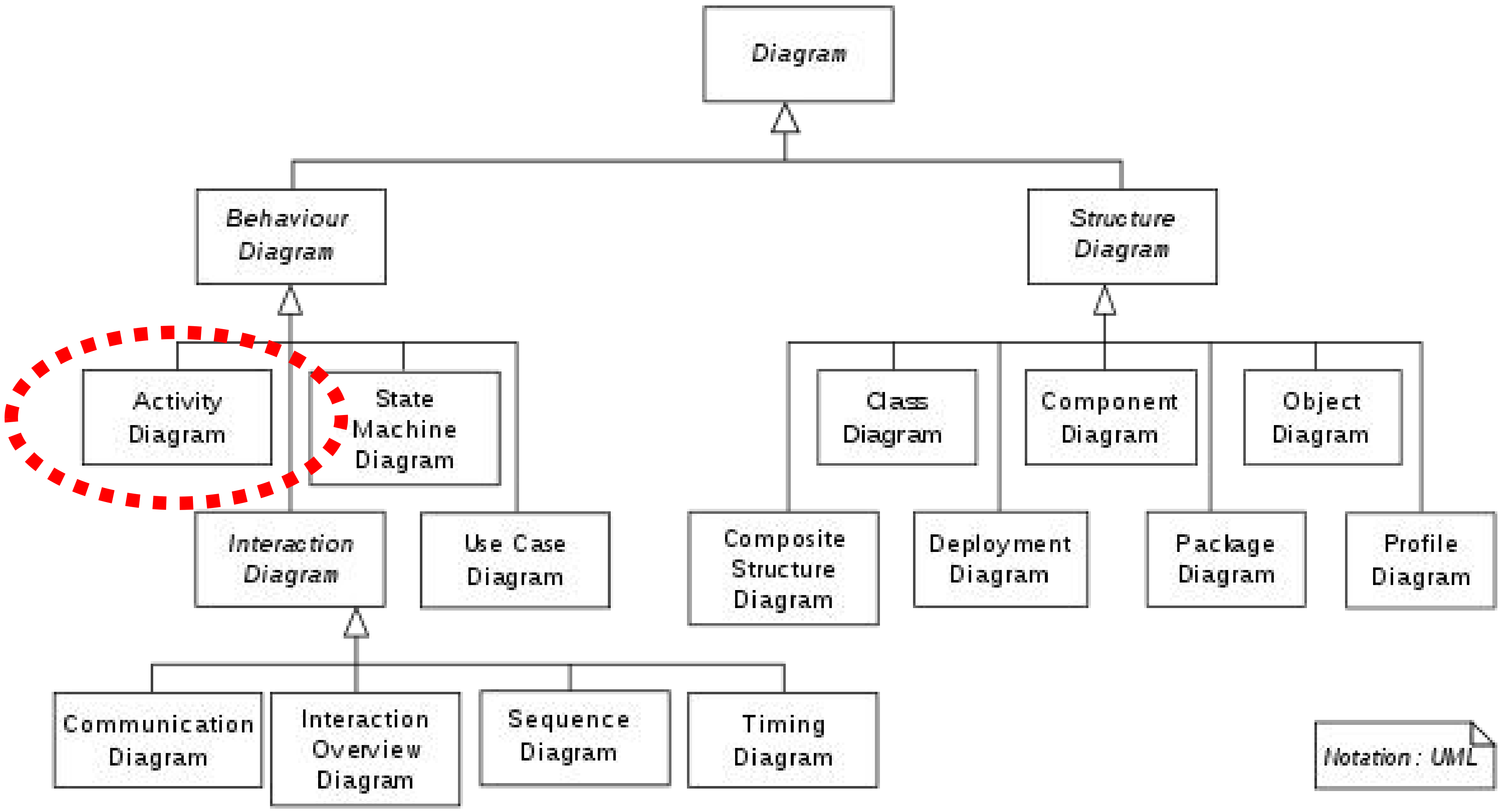


Further Reading: UML activity diagram



Further Reading: UML

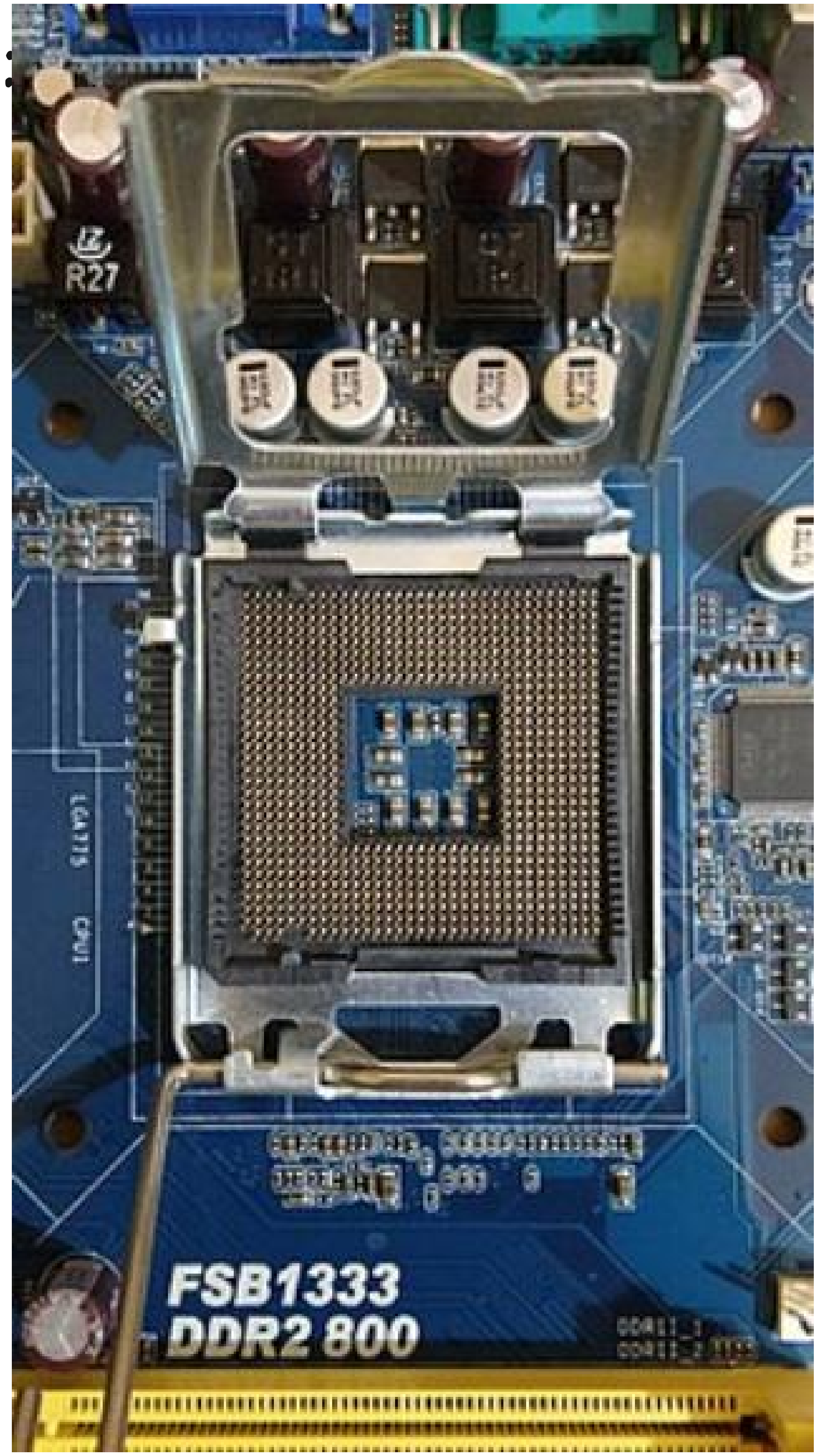
Xi'an Jiaotong-Liverpool University 西交利物浦大学



- Activity Diagram is a component of UML
- The Unified Modeling Language (UML) is a general-purpose, developmental modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.

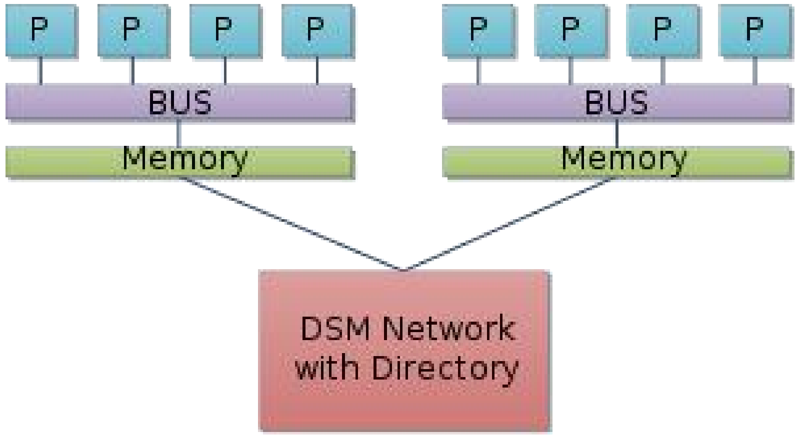
MPI map and bind *

- unit of hardware:
  - hwthread
  - core
  - L1cache
  - L2cache
  - L3cache
  - socket
  - numa
  - board
  - node



socket

HP Z820 Workstation



numa

map-by option

- -map-by unit is the most basic of the mapping policies, and makes process assignments by iterating over the specified unit until the process count reaches the number of available slots.
- Purpose: control the iteratively distributive pattern of processes

```
% mpirun -host hostA:4,hostB:2 -map-by core ...
R0  hostA  [BB/../../../../../../..][../../../../../../../..]
R1  hostA  [../BB/../../../../../..][../../../../../../../..]
R2  hostA  [../../BB/../../../../..][../../../../../../../..]
R3  hostA  [../../../BB/../../../..][../../../../../../../..]
R4  hostB  [BB/../../../../../../..][../../../../../../../..]
R5  hostB  [../BB/../../../../../..][../../../../../../../..]
```

```
% mpirun -host hostA:4,hostB:2 -map-by socket ...
R0  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R1  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R2  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R3  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R4  hostB  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R5  hostB  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
```

-rank-by

Xi'an Jiaotong-Liverpool University 西交利物浦大學

- A natural hardware ordering can be created by specifying a smaller unit over which to iterate for ranking
- Purpose: control neighbor relationship of processes

```
% mpirun -host hostA:4,hostB:2 -map-by socket -rank-by core ...
R0  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R1  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R2  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R3  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R4  hostB  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R5  hostB  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
```

- Compared to the previous:

```
% mpirun -host hostA:4,hostB:2 -map-by socket ...
R0  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R1  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R2  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R3  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R4  hostB  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R5  hostB  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
```

-bind-to

- A common binding pattern involves binding to cores, but spanning those core assignments over all of the available sockets.
- sses

```
% mpirun -host hostA:4,hostB:2 -map-by socket -rank-by core -bind-to core ...
R0  hostA  [BB/../../../../../../..][../../../../../../../..]
R1  hostA  [../BB/../../../../../..][../../../../../../../..]
R2  hostA  [../../../../../../../..][BB/../../../../../../..]
R3  hostA  [../../../../../../../..][../BB/../../../../../..]
R4  hostB  [BB/../../../../../../..][../../../../../../../..]
R5  hostB  [../../../../../../../..][BB/../../../../../../..]
```

-

```
% mpirun -host hostA:4,hostB:2 -map-by socket -rank-by core ...
R0  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R1  hostA  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R2  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R3  hostA  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
R4  hostB  [BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../..]
R5  hostB  [../../../../../../../..][BB/BB/BB/BB/BB/BB/BB/BB]
```

Reference

- Computer Networking A Top-Down Approach
  - Chapter 6.6
- Introduction to Parallel Computing
  - Chapter 2.4.1-2.4.5
- Tutorial — MPI for Python 3.1.4 documentation (mpi4py.readthedocs.io)
- --map-by unit option - IBM Documentation