

# Architecture and Parallel Computers

Chris Kauffman

*Last Updated:  
Thu Jan 29 01:58:06 PM EST 2026*

# Logistics

## Reading: Grama Ch 2

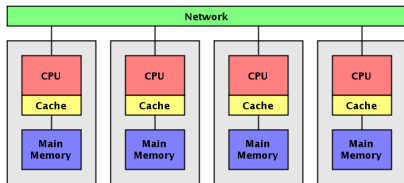
- ▶ **Focus on 2.3-5**, material pertaining to **distributed memory**
  - ▶ We will return to shared memory arch later in the course
  - ▶ Cache Coherence, PRAM models, False Sharing, Memory Bus are all shared memory topics we'll discuss later
- ▶ Sections 2.1 and 2.2 optional, deeper architectures
- ▶ Sections 2.6 and 2.7 encouraged, deeper on networks

## SISD, SIMD, MIMD, SPAM, and other 4-letter words

- ▶ Traditional CPU, Single Instruction Single Data (SISD)  
`ADD r1, r2            # add int in r2 to r1`
- ▶ Most computers now have cpu instructions to add multiple  
`PHADD mm1, mm2    # add two ints in mm2 to ints in mm1`
- ▶ Smart compilers will select **SIMD / Vector instructions** when appropriate architecture support is available
- ▶ Explicit hardware parallelism is good for multimedia stuff (graphics, games, images, sound, videos)
- ▶ Flynn's taxonomy of Parallel Architecture includes  
    I: Instruction    SISD    SIMD    SPMD    P: Program  
    D: Data           MISD    MIMD    MPMD
- ▶ Some parallel programs exist as Multiple Program Multiple Data (MPMD) like client server models (`client.c` and `server.c` are separate programs)
- ▶ Our focus and the most common type of parallel program: **Single Program Multiple Data (SPMD)**: *Write one program which processes different hunks of data in parallel*

# Distributed vs Shared Memory Architectures

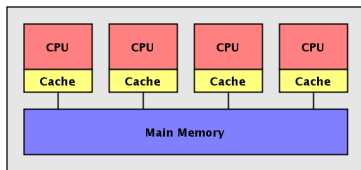
## Distributed Memory



Source: Kaminsky/Parallel Java

- ▶ Far more scalable/cost effective
- ▶ Sharing information requires explicit send/receive commands between processors
- ▶ Communication requires more care/more expensive

## Shared Memory



Source: Kaminsky/Parallel Java

- ▶ Convenience: no explicit send/receive, write shared memory address
- ▶ Requires coordination to prevent corrupting memory
- ▶ Communication cost is low but requires discipline

# Modeling Distributed Memory Parallel Computers

Will spend a some time discussing networks used in parallel computing. These have consequences for algorithms, but unless you're building your own machine (min cost \$1M) you're stuck with what you get. Examples:

1. You can test programs on your laptop which likely has 4 cores and no interconnect; MPI models distributed memory as individual processes but scale is limited...
2. We will use the Zaratán cluster for MPI programming which uses an [Infiniband Interconnect](#) comprised of switches / processors / storage in a [Switched Fabric Topology](#), balances cost and scale reasonably
3. If you have a chance to work on the [#7 Super Computer in the World, Fugaku, in Japan](#), it is reported to have a [Multi-dimensional Torus Topology](#) dubbed "Tofu"

Communication protocols on individual machines vary in details BUT coding via a library like MPI will utilize the best available underlying tech possible

# Static Networks for Distributed Machines

- ▶ String up a bunch of **Processing Elements (PEs)**
- ▶ Decide which PE is connected to which other PE
- ▶ Live with the effects on cost of communication

## Communication Cost Factors

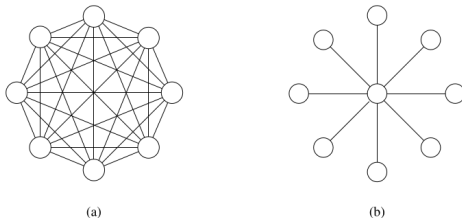
When sending a message of size  $m$  words of memory

- ▶  $t_s$ : Startup time, incurred once
- ▶  $t_h$ : Per-hop time, overhead incurred for each link between source and destination
- ▶  $t_w$ : Per-word transfer time between two nodes, takes  $t_w \times M$  time for each link between source and destination
- ▶  $L$ : number of links to traverse
- ▶  $M$ : number of words being sent
- ▶ Typical model for communication time w/ packet routing

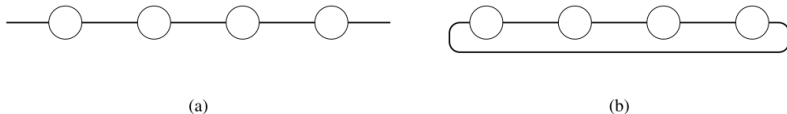
$$t_{comm} = t_s + Lt_h + t_w M$$

## Basics of Network Design : Cost vs Communication

- ▶ Balance number of links / connection pattern complexity
- ▶ VS “Distance” between PEs + Contention

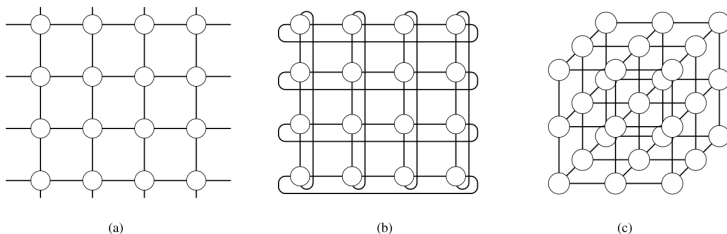


**Figure 2.14** (a) A completely-connected network of eight nodes; (b) a star connected network of nine nodes.



**Figure 2.15** Linear arrays: (a) with no wraparound links; (b) with wraparound link.

# Grid and Torus



**Figure 2.16** Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

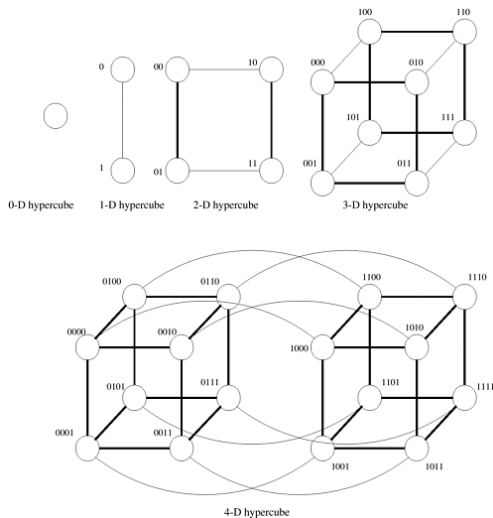
Source: Grama, Sec 2.4.3

- ▶ Common arrangement of links between PEs
- ▶ Each PE node connected to neighbors
- ▶ When wrapping around, grid becomes a torus
- ▶ For a 2D torus with  $p$  nodes, **how many links are required?**
- ▶ *Hint: surprisingly simple, think of each processor “owning” down and right links*
- ▶ **How many links in a 3D torus?**



## Exercise: HyperCube

- ▶ D-dimension hypercube: connect two  $(D - 1)$  dimension hypercubes, link corresponding nodes
- ▶ How many nodes and links in a D-dimension hypercube?
- ▶ *Hint: Nodes are easy, links are tricky, try Grama textbook...*



## Answers: HyperCube

D-dimensional Hypercube has

- ▶  $2^D$  Processors
- ▶  $2^D \times D/2$  links

Can show this via Proof by Induction but that's not our focus

### That's a lot of Links

- ▶ Many communication patterns have excellent performance on a hypercube
- ▶ Building one requires wiring processors together in a highly complex manner<sup>1</sup>
- ▶ Ex: 10-dimensional hypercube with 1024 Processors each with 10 links to a unique set of other processors
- ▶ Hypercubes are a favorite theoretical topology and useful in some cases for algorithm analyses but ...
- ▶ Too expensive/complex for large-scale machines

---

<sup>1</sup>Academic papers that describe new network architectures sometimes include *wiring algorithms* to show their complex network is actually practical to construct in reality: [example](#)

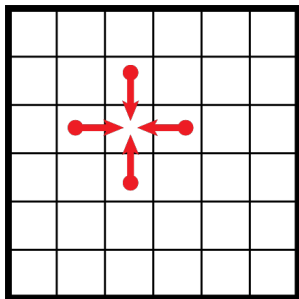
## Exercise: Compare Networks on Parallel Stencil

- ▶  $P$  processors
- ▶ Network 1: 2D-Mesh: around  $2P$  links
- ▶ Network 2:  $\log_2(P)$  dim. Hypercube w/  $(P \log_2(P)/2)$  links
- ▶ **Discuss** advantages/disadvantages of Mesh vs Hypercube arrangement for this application
- ▶ Outline an algorithm, estimate cost-effectiveness of code+hardware

### Image “blurring”

- ▶ A large image is distributed across the  $P$  processors
- ▶ Each proc holds a 2D hunk of the image
- ▶ To blur the entire image, must assign RGB values which are average of “neighborhood”

### Stencil



## Answers: Compare Networks on Parallel Stencil

- ▶ Divide image into 2D hunks
- ▶ PEs must communicate with other PEs that have neighboring hunks of the image

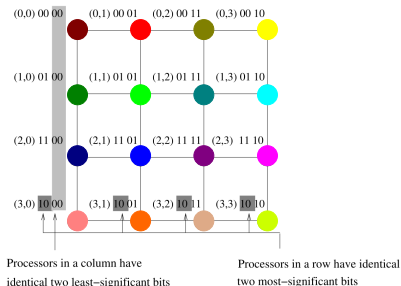
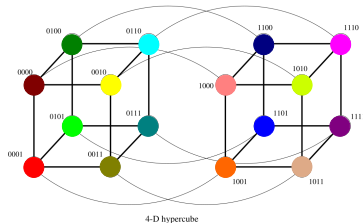
### 2D Mesh

- ▶ Maps VERY easily onto a 2D Mesh / Grid / Torus
- ▶ PEs locally blur own portion of image
- ▶ Exchange boundary pixels with 4 neighbors except for outer edge PEs

# Answers: Compare Networks on Parallel Stencil

## Hypercube

- ▶ Intuition: have many more links than in the 2D Torus, should be possible to place neighboring pixel hunks on neighboring procs
- ▶ *Embed 2D-Mesh into a Hypercube*: discussed in Grama 2.7.1, uses Gray Codes for Proc Numbering and is beyond in-class / exam questions (perhaps an assignment problem)
- ▶ After embedding Mesh in Hypercube, use Mesh algorithm



# Network Embedding

- ▶ Some algorithms work well in a particular network
- ▶ When running them on another network, look for an **embedding** that replicates (as much as possible) features of the original network
- ▶ Embedding (informally):
  - ▶ Assignment of PEs in network A to PEs in network B
  - ▶ Assignment of links network A to links in network B
- ▶ Assignments lead to consequences
  - ▶ Ex: PE4 and PE5 are connected by a single link in Network A but are 2 links apart in Network B
  - ▶ Ex: In network A, parts of links X,Y,Z are all mapped onto Link W in network B
- ▶ Metrics like **Dilation** and **Congestion** evaluate the quality of different embedding choices
- ▶ Previous example found it is possible embed 2D Mesh in a Hypercube with Dilation / Congestion of 1

## Exercise: Compare Networks on Parallel Sum

- ▶  $P$  processors (assume  $P$  is a power of 2)
- ▶ Network 1: 2D-torus:  $2P$  links
- ▶ Network 2:  $\log_2(P)$  dim. Hypercube w/  $(P \log_2(P)/2)$  links
- ▶ **Discuss** advantages/disadvantages of torus vs hypercube arrangement for this application
- ▶ Outline an algorithm, estimate cost-effectiveness of code+hardware

### Sum Array of Numbers

- ▶ Each proc holds a hunk of the data array
- ▶ Want total sum on root PE0 at end of algorithm
- ▶ **State your algorithm:** Try to minimize communication at each step, exploit as much parallelism as possible

### Networks

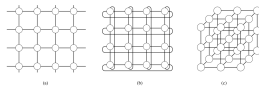
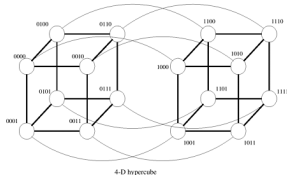


Figure 2.16 Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.



## Answers: Compare Networks: Parallel Sum

*Goal:* Get sum on Proc 0

First, each Proc sums its own chunk of numbers then...

### 2D Torus: N by N Square

- ▶ Send values UP rows then LEFT across columns
  - ▶  $2*N$  Communication steps, always neighbors
  - ▶ Many Procs **Idle** during communication
- ▶ Other Communication steps will result in multi-hop communication with non-neighbor procs - will revisit this later

### N-dimensional HyperCube

- ▶ Each Proc has a binary address: ex: 100110
- ▶ Starting with bit  $i = (N - 1)$  while  $i > 0$ 
  - ▶ Each Proc with bit  $i == 1$  sends to  $i == 0$
  - ▶ Decrement  $i$ , repeat
- ▶ Takes N communication steps



# Communication Patterns Later

- ▶ We will talk more about Parallel Sum later
- ▶ Parallel Sum is an example of a **reduction** - general communication pattern that recurs often in Parallel Computing
- ▶ Covered in more detail in Section 6.6
- ▶ Parallel Sum is discussed in [Lecture notes by Susan Hayes](#)

# Characteristics of Various Networks

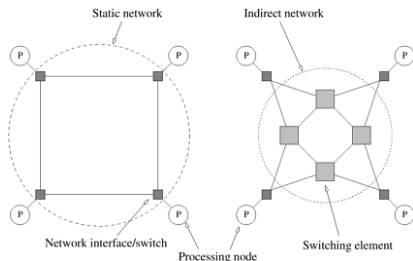
**Table 2.1** A summary of the characteristics of various static network topologies connecting  $p$  nodes.

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	$2(p - \sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p \log p)/2$
Wraparound $k$ -ary $d$ -cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	$dp$

Several metrics described in textbook

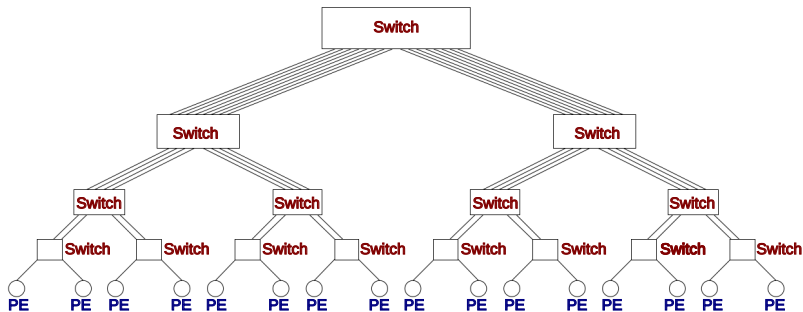
- ▶ *Diameter*: max hops away any two procs can be
- ▶ *Bisection width*: remove  $N$  links to get 2 networks, equal size
- ▶ *Arc Connectivity*: remove  $N$  links to get 2 networks, any size
- ▶ *Cost*: can correspond to number of links

# Dynamic Networks



- ▶ In a static network, connections are fixed
- ▶ Dynamic networks use switches: send data into network with destination, may alter a connection to point in a different direction
- ▶ Akin to the internet: packet switching network
- ▶ **Most modern HPC networks are Dynamic:** balances cost and scale better than the older static networks

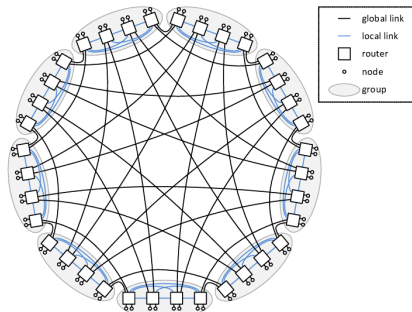
# Fat Trees



**Figure 2.19** A fat tree network of 16 processing nodes.

Often used as network switches are inexpensive and widely available while still providing good communication speeds

# Dragonfly Networks



Source: [Garcia et al. International Conference on Parallel Processing 2012](#)

- ▶ Some years ago a new HPC network topology coined “Dragonfly Networks” were introduced
- ▶ Proved influential due to...
- ▶ Low diameter: limits number of hops between any nodes (processors)
- ▶ “Groups” are fully interconnected
- ▶ Low Cost due to use of available commodity switches

# Topology vs Interconnect

- ▶ The “shapes” we’ve seen so far are examples of network **Topology** which dictates how networks are interconnected, how many links required, diameter of the network, etc.
- ▶ The specific hardware used to create the network is not part of the description
- ▶ Most HPC facilities will describe the **Interconnect** they use: specific technology for communication
- ▶ Two most common Interconnects in Top-500 HPC facilities
  - ▶ **Infiniband**: Various owners/vendors (recently NVidia)
  - ▶ **Slingshot-11**: Cray (Hewlett Packard Enterprises)
- ▶ Interconnects usually support configuration into various Topologies: Slingshot-11 can be in Dragonfly, Fat-Tree, Butterfly, etc. according to docs

## Routing: Store/Forward Packet and Packet Switching

When sending messages, intermediate nodes must decide what to do with a message: **Routing protocol/scheme**

### Store and Forward

- ▶ Accumulate the whole message (all  $M$  words), store it until it can be forwarded to next hop
- ▶ Easy to build but requires large-ish internal buffers and generally has bad performance

### Standard Packet Switching

- ▶ Break message into chunks (packets)
- ▶ Use packet header to carry error-correction info, routing info
- ▶ Optimized for the unreliable internet: go around overloaded / dead nodes, adjust to faster paths if found
- ▶ Better but incurs robustness overhead isn't necessary present in most reliable HPC machine networks

## Routing: Cut-Through Communications

- ▶ Standard in HPC network design to optimize communication protocol: sacrifice some robustness to improve speed
- ▶ Cut-through Routing is an abstract version of this
- ▶ Similar to packet switching: break message into chunks
- ▶ Send a *tracer* from source to destination to determine route - all packets then follow that route
- ▶ Send message in *flits* (packets) along tracer route - reduces latency over Store/Forward
- ▶ Minimize data in packet for error correction, re-routing, etc. - reduced overhead vs Standard Packet Switching
- ▶ Comm time dominated by initial path determination  $t_h L$  and total message size  $t_w M$



# Our Approach

## Algorithm + Specific Network

Assume Cut-Through Routing,  
account for hops between PEs

$$t_{comm} = t_s + Lt_h + t_w M$$

- ▶ Simplified model for Comm but reasonable enough to guide algorithm decisions on how to utilize specific network
- ▶ Minimize  $L$  between communicating PEs in algorithms
- ▶  $L$  changes with topology: e.g. Hypercube needs fewer communication steps than a Torus due to more abundant links

## Algorithm + Arbitrary Network

Will ignore network topology,  
congestion, number of hops

$$t_{comm} = t_s + t_w M$$

- ▶ Abstracted away from specific network features which will vary
- ▶ Ignores path lengths, unrealistic but understandable when network structure is unknown
- ▶ Still accounts for number and size of communications in algorithm