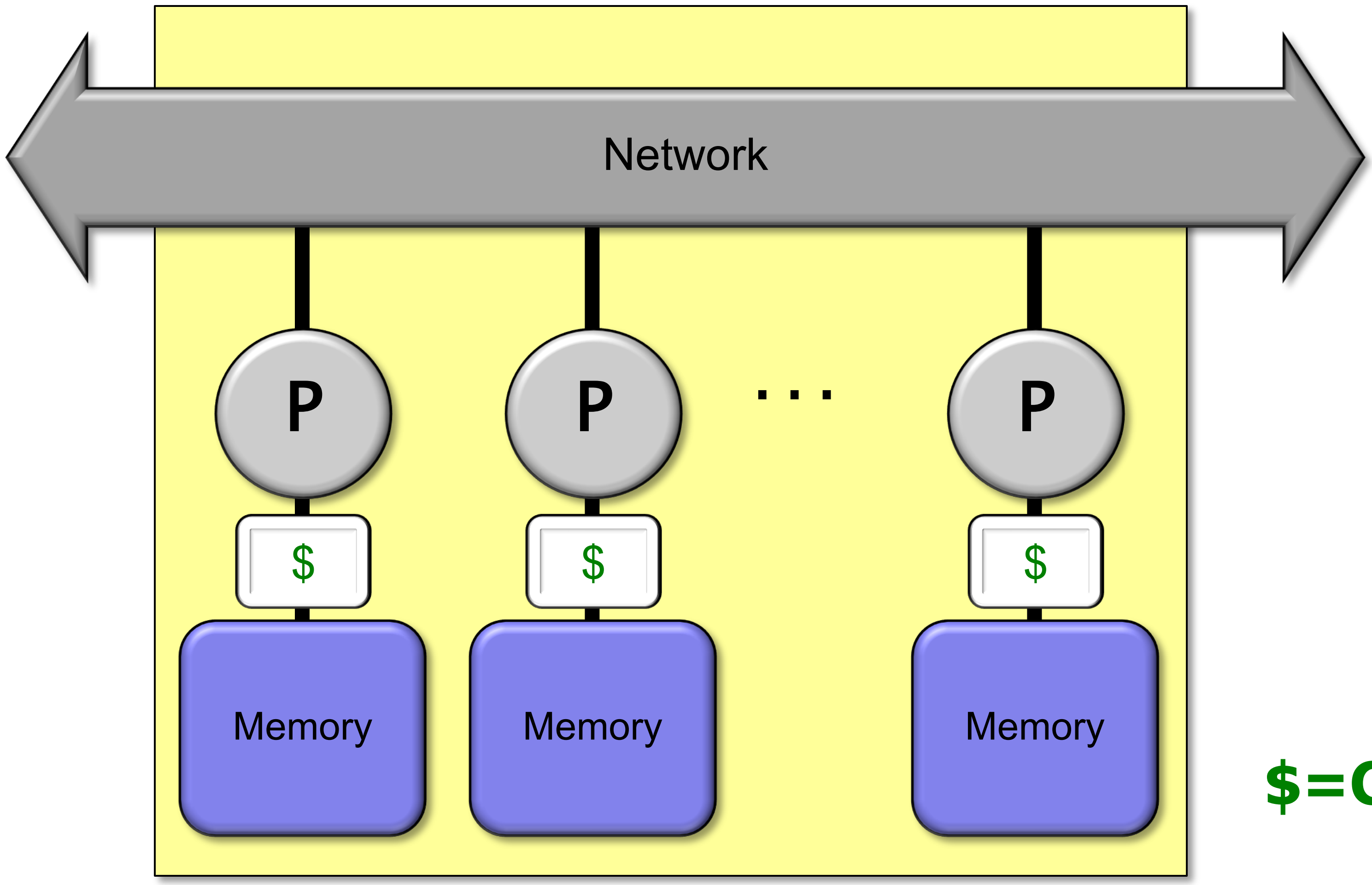
CSC367 Parallel computing

Lecture 12: Distributed Memory Architectures and their Parallel Programming Model

Thanks to Scinet, NSF XSEDE, James Demmel, and Kathy Yelick

Distributed Memory Architecture



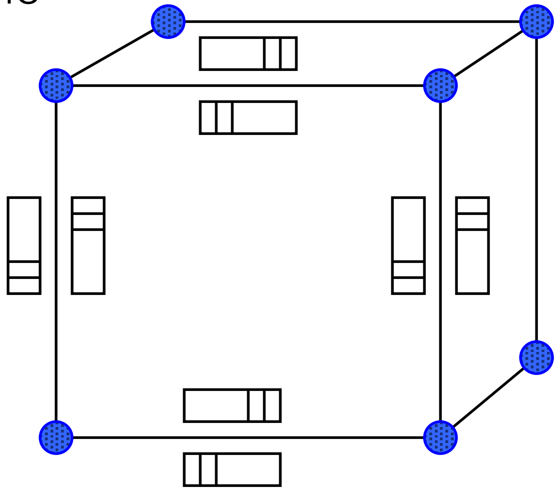
$=Cache

Historical Perspective

- Early distributed memory machines were:

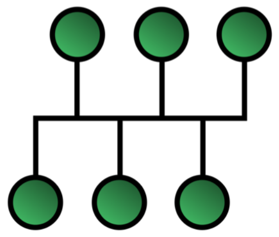
  API that is hardware dependent, different machines hard to communicate

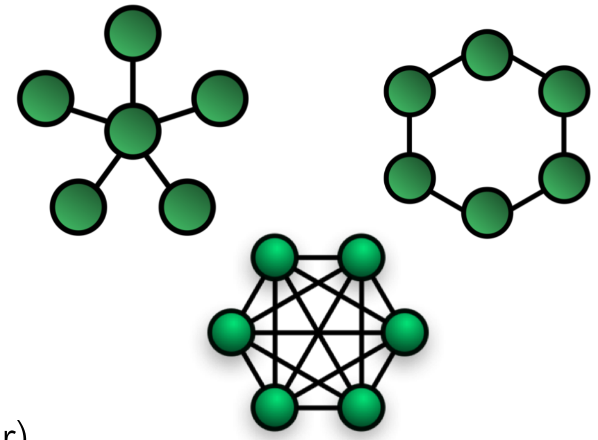
  - Collection of microprocessors.
  - Communication was performed using bi-directional queues between nearest neighbors.
- Messages were forwarded by processors on path.
  - “Store and forward” networking
- There was a strong emphasis on topology in algorithms, in order to minimize the number of hops = minimize time



Network Analogy

- To have a large number of different transfers occurring at once, you need a large number of distinct wires
  - Not just a bus, as in shared memory
- Networks are like streets:
  - Link = street.
  - Switch = intersection.
  - Distances (hops) = number of blocks traveled.
  - Routing algorithm = travel plan.
- Properties:
  - Latency: how long to get between nodes in the network.
    - Street: time for one car = dist (miles) / speed (miles/hr)
  - Bandwidth: how much data can be moved per unit time.
    - Street: cars/hour = density (cars/mile) * speed (miles/hr) * #lanes
    - Network bandwidth is limited by the bit rate per wire and #wires


Bus-shared memory


Network topologies

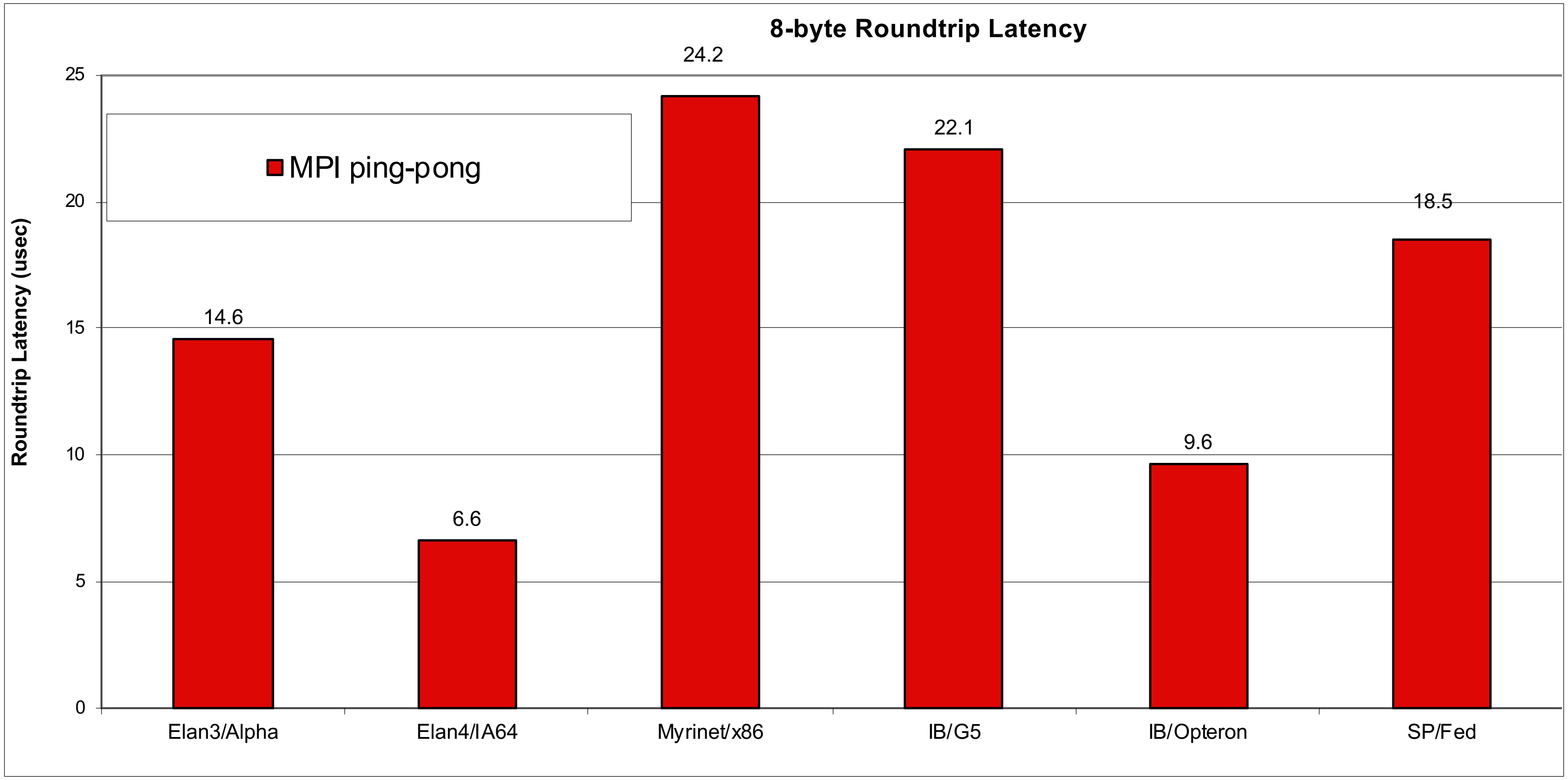
Design Characteristics of a Network

- Topology (how things are connected)
  - Crossbar; ring; 2-D, 3-D, higher-D mesh or torus; hypercube; tree; butterfly, …
- Routing algorithm:
  - Example in 2D torus: all east-west then all north-south (avoids deadlock).
- Switching strategy:
  - Circuit switching: full path reserved for entire message, like the telephone.
  - Packet switching: message broken into separately-routed packets, like the post office, or internet
- Flow control (what if there is congestion):
  - Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.

Performance Properties of a Network: Latency

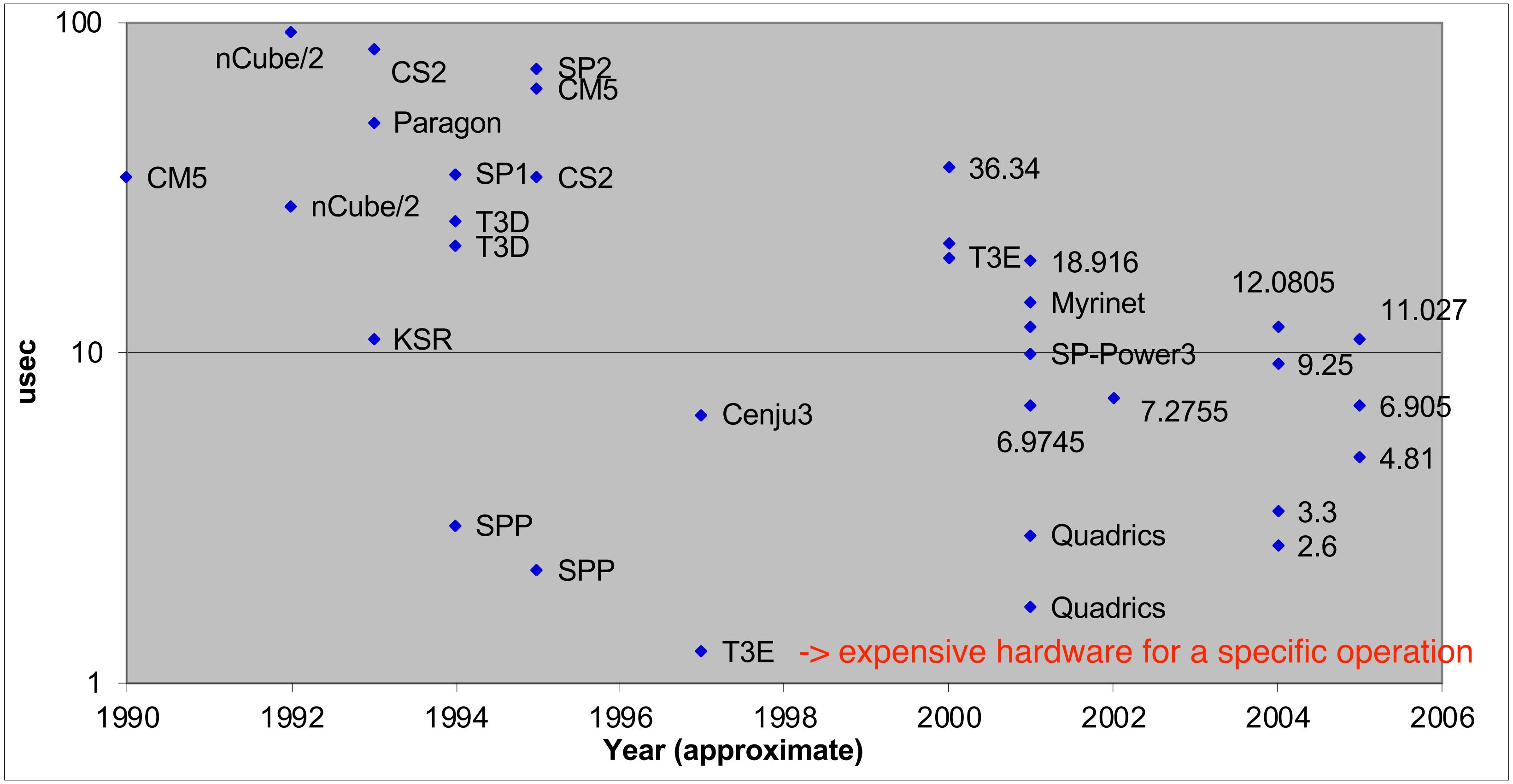
- Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software latencies (user program to user program)
- Latency is key for programs with many small messages

Latency on Some Machines/Networks



- Latencies shown are from a ping-pong test using MPI
- These are roundtrip numbers: many people use ½ of roundtrip time to approximate 1-way latency (which can't easily be measured)

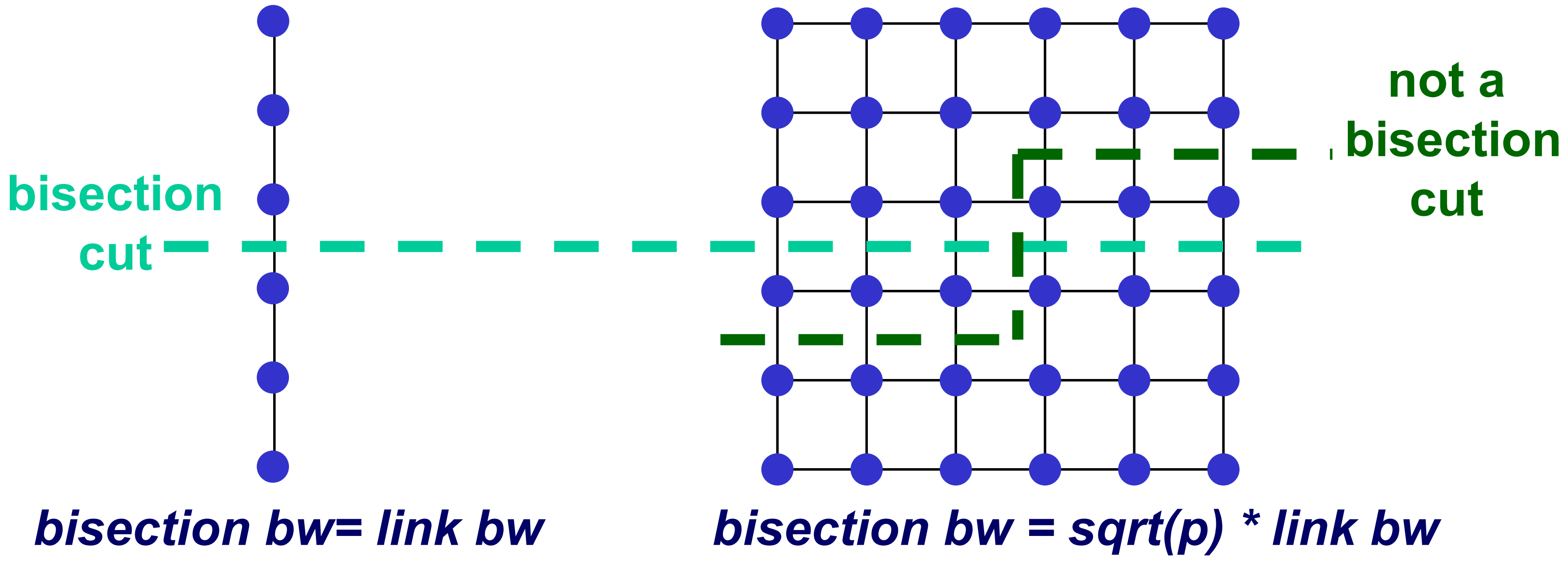
End to End Latency (1/2 roundtrip) Over Time



- Latency has not improved significantly, unlike Moore’s Law
  - T3E (shmem) was lowest point – in 1997

Performance Properties of a Network: Bisection Bandwidth

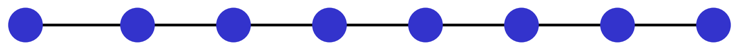
- Bisection bandwidth: bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across "narrowest" part of the network



- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

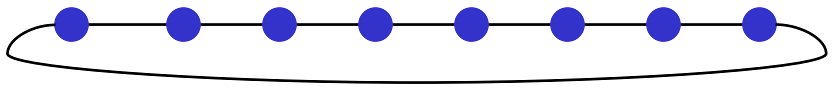
Linear and Ring Topologies

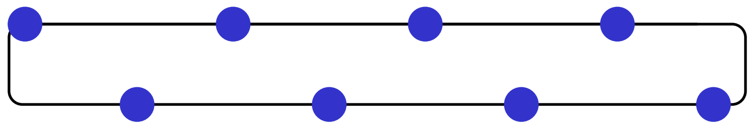
- Linear array (1D mesh)



  - Diameter = n-1;
  - Bisection bandwidth = 1 (in units of link bandwidth).

- Torus or Ring



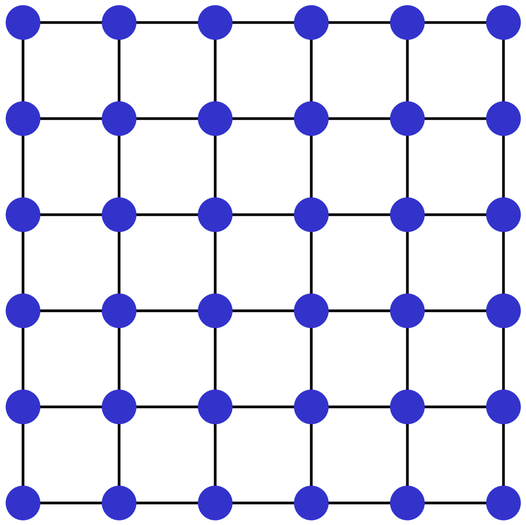


  - Diameter = n/2;
  - Bisection bandwidth = 2.
  - Natural for algorithms that work with 1D arrays.

Meshes and Tori

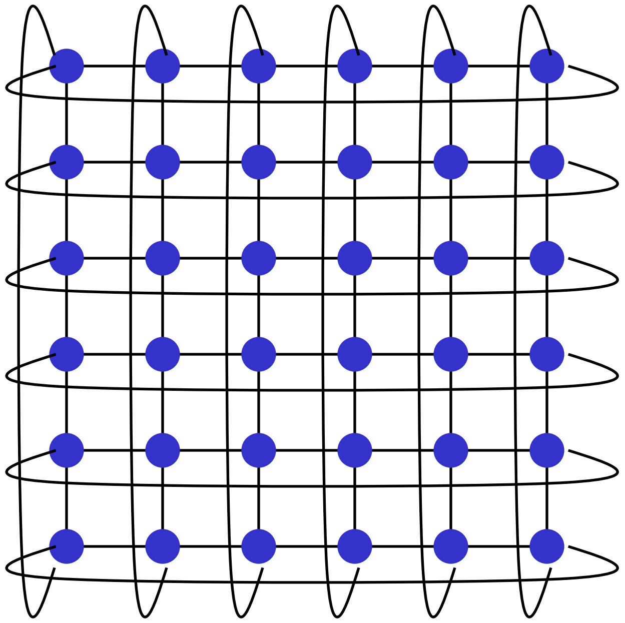
Two dimensional mesh

- Diameter = 2 * (sqrt(n) – 1)
- Bisection bandwidth = sqrt(n)



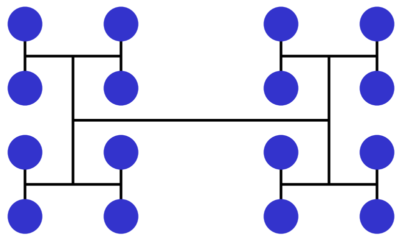
Two dimensional torus

- Diameter = sqrt(n)
- Bisection bandwidth = 2 * sqrt(n)

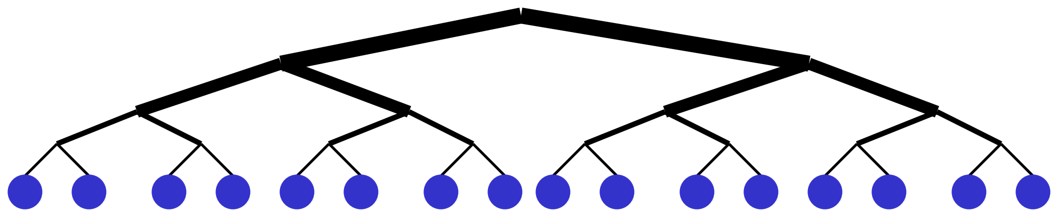


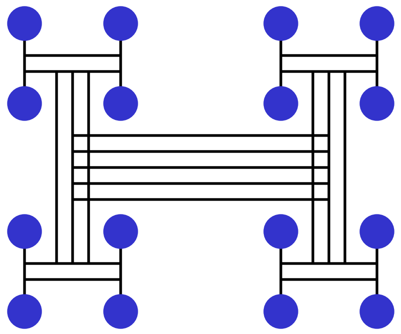
- Generalizes to higher dimensions
    - Cray XT (eg Hopper@NERSC) uses 3D Torus
- Natural for algorithms that work with 2D and/or 3D arrays (matmul)

Trees very popular …



- Bisection bandwidth = 1.
- Easy layout as planar graph.
- Many tree algorithms benefit from tree topologies (e.g., summation).
- Fat trees avoid bisection bandwidth problem:
  - More (or wider) links near top. improves bisection bandwidth !
  - Example: Thinking Machines CM-5.





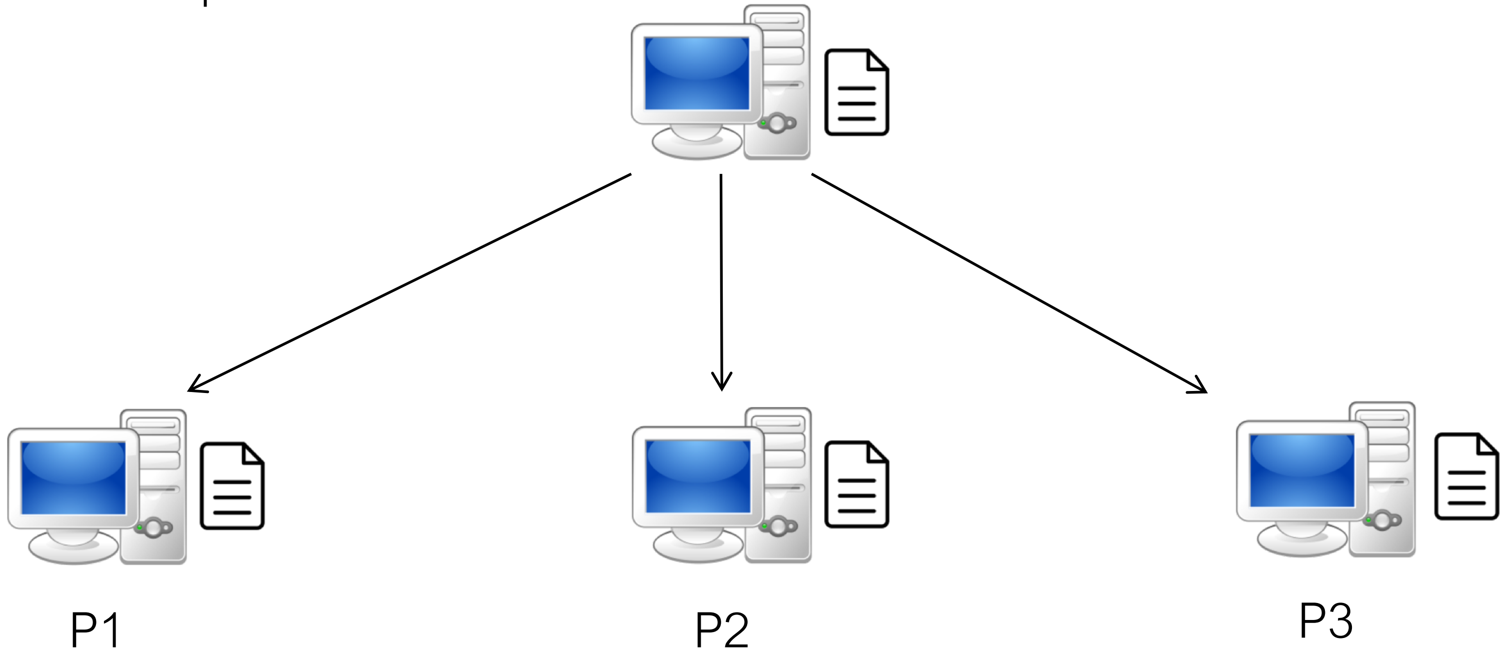
Message passing model

write 1 program, and ran on multiple machines

- Single Program Multiple Data (SPMD)

  instruction are sent over network

  - All processes execute same code, communicate via messages
  - Technically, it does support executing different programs on each host/process



Why message passing?

- Build a parallel multi-computer from lots and lots of nodes
  - Nodes are connected with a network
  - Partition the data across the nodes, computation in parallel
  - If local data is needed on remote node, send it over the interconnect
  - Computation is done collectively
  - Can always add more and more nodes => bottleneck is not the number of cores or memory on a single node
    - Scale out instead of scale up: Can increase the data size while increasing processors

scale up: 1 machine
scale out: more machine

good for physical simulation

- Downside: harder to program
  - Every communication has to be hand-coded by the developer

Message passing – logical & physical view

- Logical view: Each process is a separate entity with its own memory
  - Each process runs a separate instance of the same (parallel) program
  - A process cannot access another process's memory except via explicit messages
  - Message passing implies a distributed address space
- Physical view: the underlying architecture could be either distributed memory or shared memory
  - Message passing on a physical shared memory architecture: messages can be simulated by copying (or mapping) data between different processes' memory space
- Take away:
  - You can use message passing on a multicore/shared memory machine, however, the message passing interface simulates a "distributed environment" on the shared memory machine.
  - Consider a core is a processor and just see your shared memory machine as a distributed memory machine. idea: should optimize code on 1 node -> then scale out

Message passing compared to OpenMP/Pthreads

- Idea: let the processes compute independently and coordinate only rarely to exchange information
    - Communication adds overheads, so it should be done rarely
- Need to structure the program differently, to incorporate the message passing operations for exchanging data
- Need to partition data such that we maximize locality and minimize transfers
    - Recall the concepts discussed a few lectures ago

Building Blocks

- Interactions are carried out via passing messages between processes
- Building blocks: send and receive primitives – general form:
  - `send(void *sendbuf, int nelems, int dest)`
  - `receive(void *recvbuf, int nelems, int source)`
- Complexity lies in how the operations are carried out internally
- Example (pseudocode):

```
P0                              P1
---                             ---
msg = 5;                        recv(&msg, 1, 0);
send(&msg, 1, 1);               printf("%d", msg);
msg = 200;
```

- Key question: What will P1 receive?
  - Send operation may be implemented to return before the receipt is confirmed
  - Supporting this kind of send is not a bad idea

Blocking operations

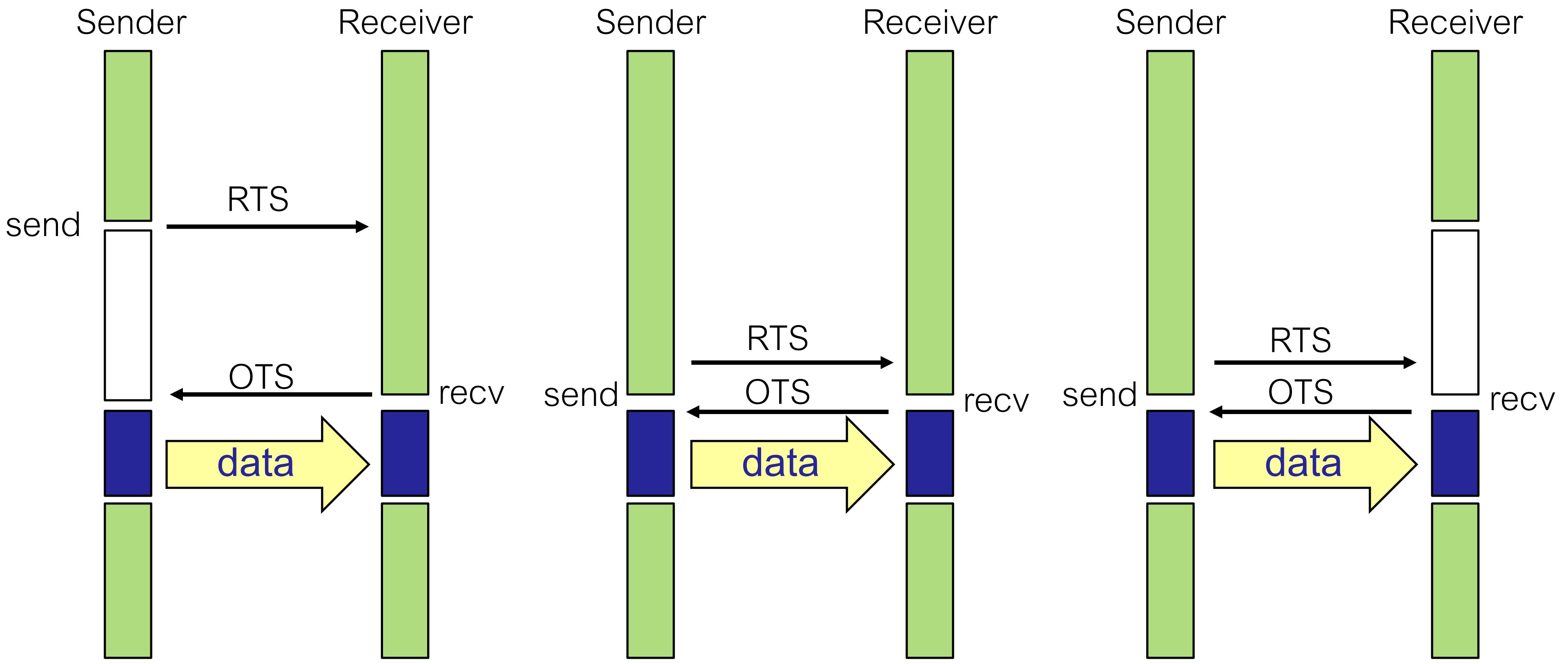
- Only return from an operation once it's safe to do so
  - Not necessarily when the msg has been received, just guarantee semantics
- Two possibilities:
  - Blocking non-buffered send/receive
  - Blocking buffered send/receive

Blocking non-buffered send/recv

- Send operation does not return until matching receive is encountered at the receiver and communication operation is completed
- Non-buffered handshake protocol – idling overheads:

RTS = request to send

OTS = ok to send



1. Sender is first; idling at sender	2. Same time; idling minimized	3.Receiver is first; idling at recv

Deadlocks in blocking non-buffered comm

- Deadlocks can occur with certain orderings of operations, due to blocking
- Example – this deadlocks:

```
P0                          P1
---                         ---
send(&m1, 1, 1);            send(&m1, 1, 0);
recv(&m2, 1, 1);            recv(&m2, 1, 0);
```

- Solution: switch order in one of the processes
    - But, more difficult to write code this way, and could create bugs

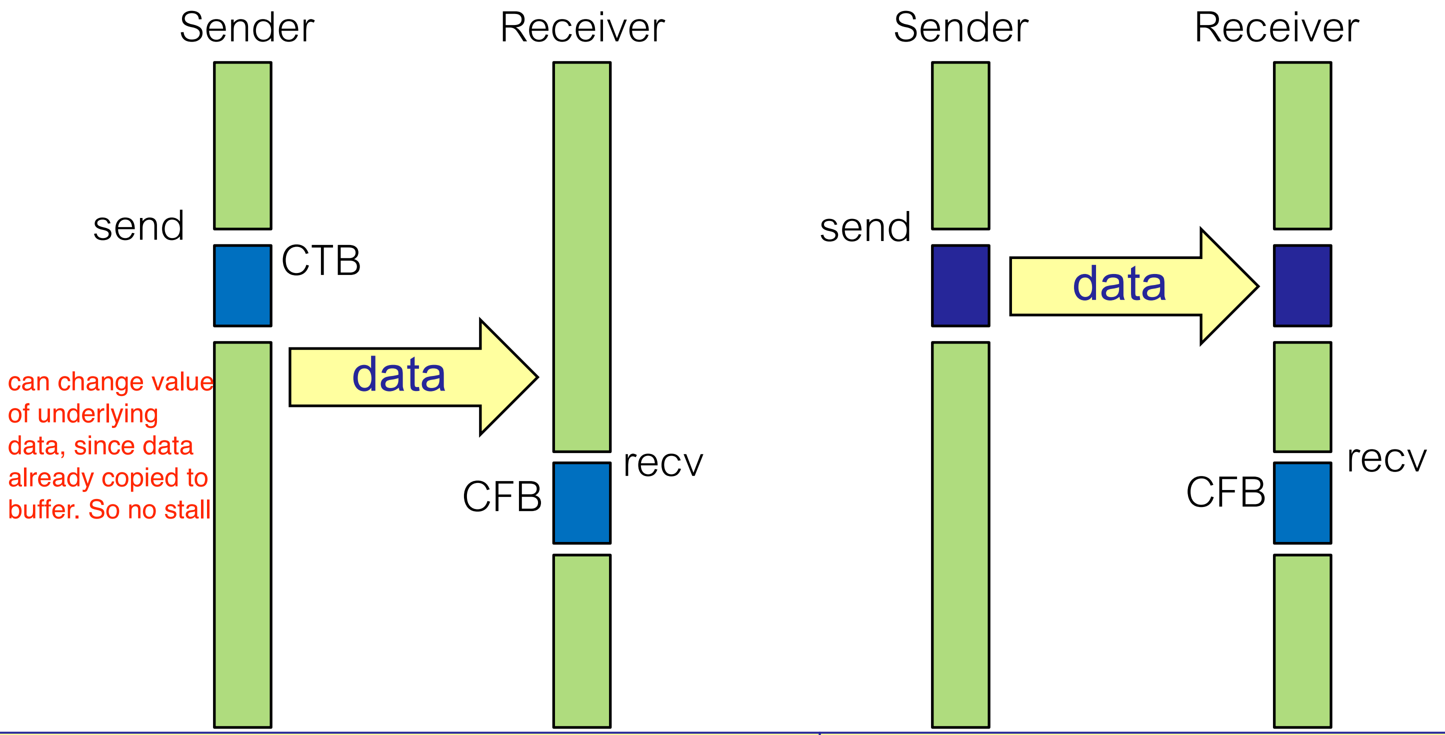
Hardware support for send/receives

- Most message passing platforms have additional hardware support for sending and receiving messages.

- They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware.
  - **Network interfaces** allow the transfer of messages from buffer memory to desired location without CPU intervention.
  - Similarly, **DMA** allows copying of data from one memory location to another (e.g., communication buffers) without CPU support

Blocking buffered send/recv

- Sender copies data into buffer and returns once the copy to buffer is completed
- Receiver must also store the data into a buffer until it reaches the matching recv
- Buffered transfer protocol – with or without hardware support:


Sender
Receiver
send
CTB
data
can change value of underlying data, since data already copied to buffer. So no stall
recv
CFB
Sender
Receiver
send
data
recv
CFB

CTB = Copy to buffer

CFB = Copy from buf

In both cases, no idling overheads! But, now buffer management overheads!

1. Use buffer at both sender and receiver Communication handled by H/W (network interface)

2. Buffer only on one side. E.g., sender interrupts receiver and deposits the data in a buffer (or vice-versa)

Problems with blocking buffered comm

- 1. Potential problems with finite buffers
  - Example:

producer block when buffer is full

```
P0 (producer)                      P1 (consumer)
---                                ---
for(i = 0; i < 1000000; i++){      for(i = 0; i < 1000000; i++){
   create_message(&m);                recv(&m, 1, 0);
   send(&m, 1, 1);                    digest_message(&m);
}                                  }
```

- 2. Deadlocks still possible
  - Example:

```
P0                                 P1
---                                ---
recv(&m1, 1, 1);                   recv(&m1, 1, 0);
send(&m2, 1, 1);                   send(&m2, 1, 0);
```

- Solution is similar: break circular waits
- Unlike previously, in this protocol, deadlocks can only be caused by waits on recv

if send first … since with buffer -> nonblocking can recv, -> no deadlock

Non-blocking operations

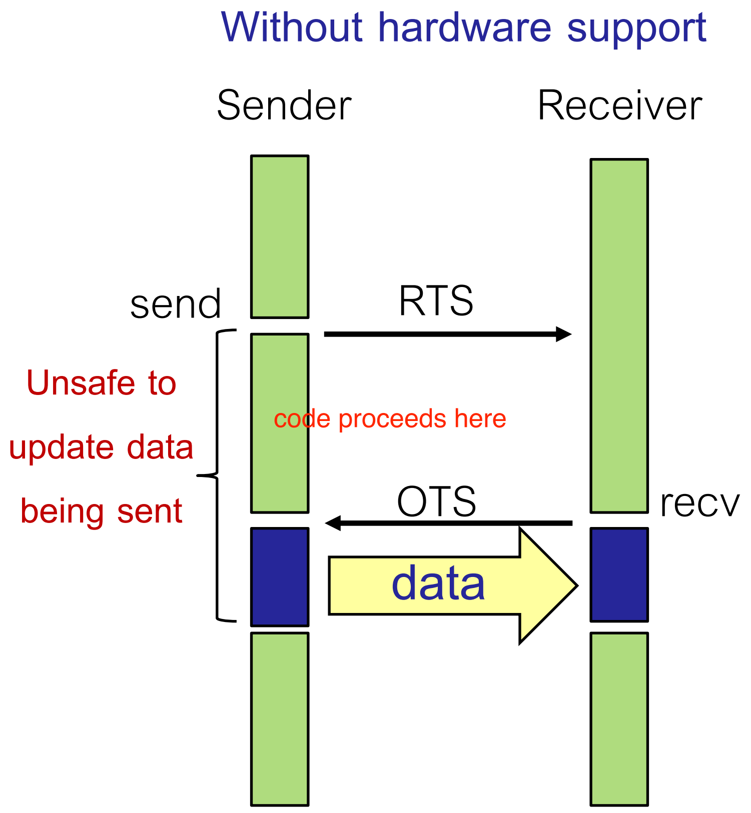
- Why non-blocking? Performance!
- User is responsible to ensure that data is not changed until it's safe
  - Typically a check-status operation indicates if correctness could be violated by a previous transfer which is still in flight
- Can also be buffered or non-buffered
  - Can be implemented with or without hardware support

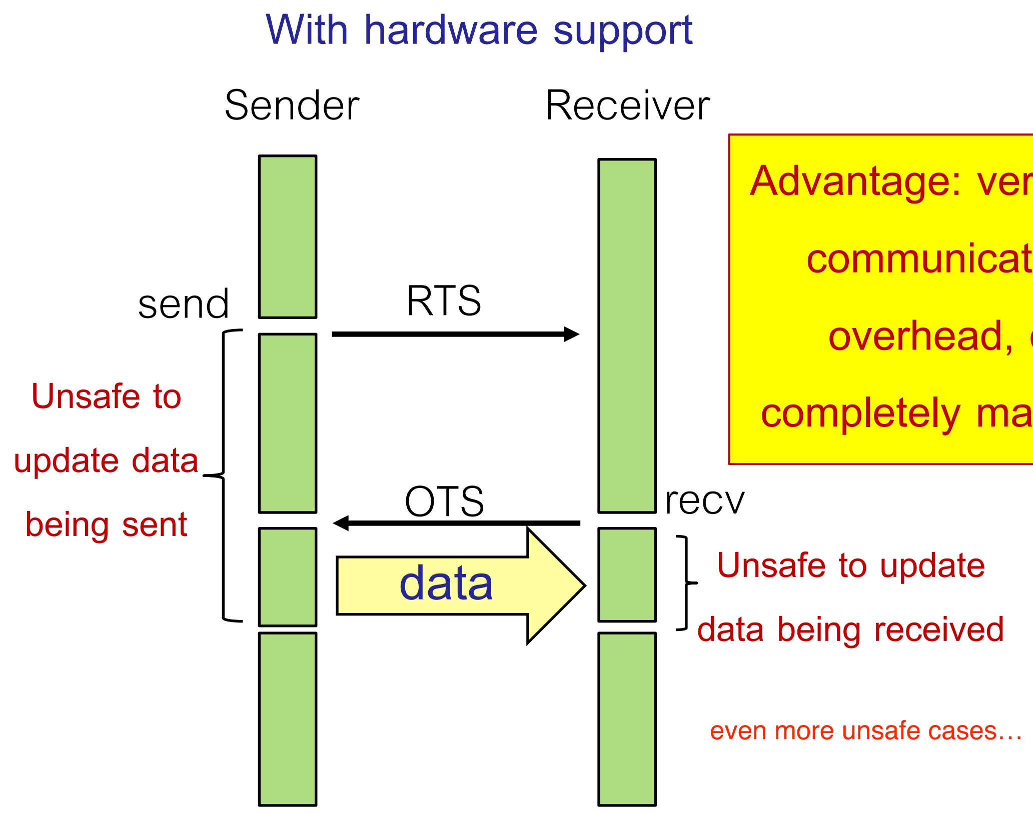
Example: non-blocking non-buffered

- Sender issues a request to send and returns immediately
- When the receive is encountered, communication is initiated

RTS = request to send

OTS = ok to send





Advantage: very little communication overhead, or completely masked!

1. When recv is encountered, transfer is handled by interrupting the sender

2. When recv is found, comm. hardware handles the transfer and receiver can continue doing other work

Summery

	Buffered	Non-Buffered	
Blocking Operations	Sending process returns after data has been copied into communication buffer.	Sending process blocks until matching receive operation has been encountered.	send and recv semantics ensured by corresponding operation
Non-blocking Operations	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return.		Programmer must explicitly ensure semantics by polling to verify completion

Take-aways

- Carefully consider the implementation guarantees
  - Communication protocol and hardware support
  - Blocking vs. non-blocking, buffered vs. non-buffered
- Tradeoffs in terms of correctness and performance
  - Need automatic correctness guarantees => might not hide communication overhead that well
  - Need performance => user is responsible for correctness via polling