COMP 790-033 - Parallel Computing

Lecture 12
November 2, 2022

Interconnection Networks
and
MPI: Message Passing Interface

- **Skim through**
  - Message Passing Interface

Topics

- **Short overview of basic issues in message passing**
- **MPI: A message-passing interface for distributed-memory parallel programming**
- **Collective communication operations**

Topics

- **Interconnection networks for parallel processors**
  - components
  - characteristics
  - network models

- **Analysis of networks**
  - diameter
  - bisection bandwidth
  - degree

- **MPI message-passing interface**
  - portable distributed-memory parallel programming
  - collective communication operations

Kinds of networks

- **Wide-area networks (WAN)**
  - internet
- **Local-area networks (LAN)**
  - ethernet, wireless 802.11x
- **System-level networks**
  - processor to processor
  - (processor to memory)

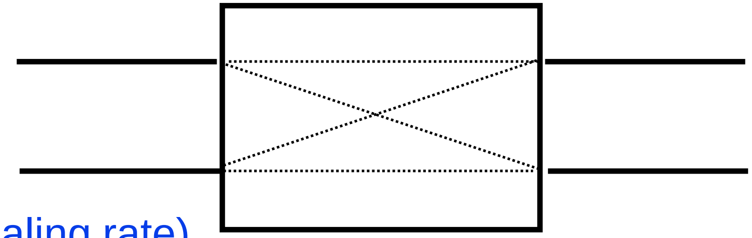
These networks differ in scalability, assumptions, cost

- Primary focus in this course is system-level networks

Components of a network

- **clusters**
  - each processor has a dedicated network interface
- **switches**
  - k inputs, m outputs, $m \geq k$
    - » simplest: $k = m = 2$
- **links**
  - characteristic bandwidth

    (# parallel bits per link) • (signaling rate)



Four characteristics of networks

- **Network topology**
  - physical interconnection structure of network
    - » analogy: Roadmap showing interstates
- **Routing algorithm**
  - rules that specify which routes a message may follow
    - » analogy: To go from Durham to DC, take I-85N to I-95N to I-495
- **Switching Strategy**
  - determines how a message traverses a route
    - » analogy: Presidential convoy reserves entire route in advance, while a group of travelers in separate cars make individual switching decisions
- **Flow control**
  - determines when a message makes progress
    - » analogy: Traffic signals and rules: two cars cannot occupy the same location at the same time

Network topology

- **Connected undirected graph G = (N, C)**
  - N = set of nodes
  - C = set of channels (bidirectional links)

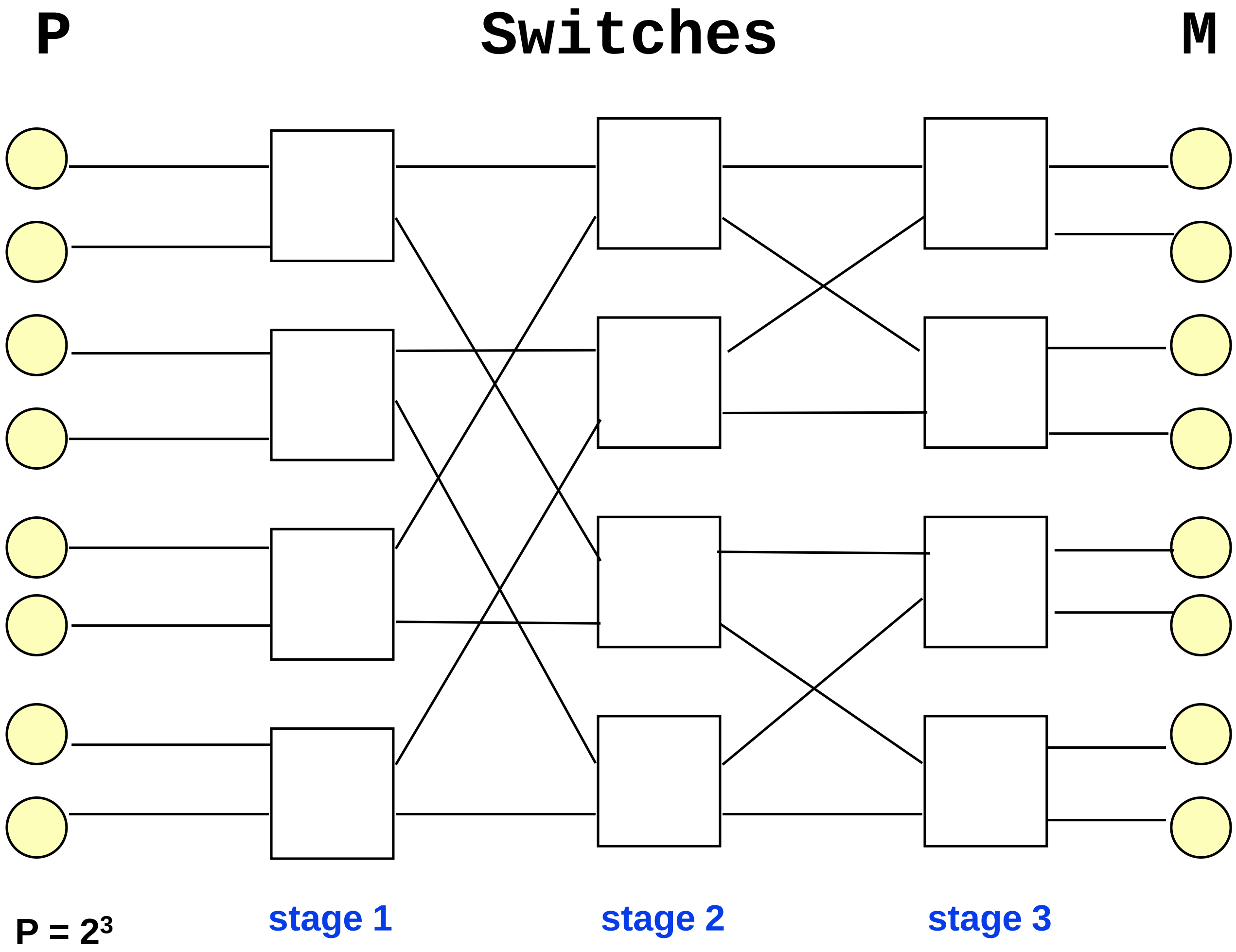
- **Indirect network (switching fabric)**
  - employs switching nodes without an attached processor or memory
  - switching nodes do not generate traffic
  - typical case in modern networks

- **Direct network**
  - every node can be a producer and/or consumer of messages
  - no pure switching nodes

Indirect networks

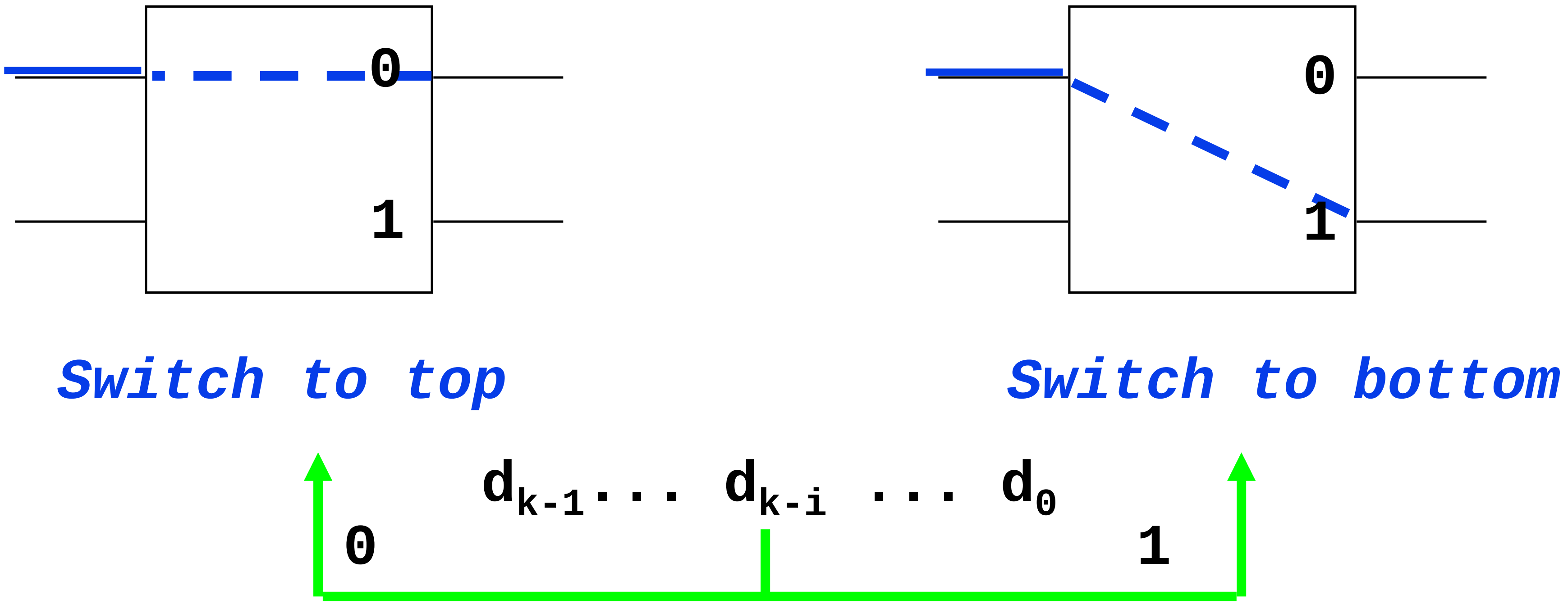
- **Processor to memory interconnect in shared-memory machines**
- **Connect p processors to p memory banks**
  - Example: bus
    - » $\Theta(p)$ switches
    - » simultaneous references always serialize
  - Example: crossbar
    - » $\Theta(p^2)$ switches
    - » simultaneous references in disjoint banks serviced in parallel
  - Example: multistage network
    - » $\Theta(p \lg p)$ switches and links
      - $\Theta(\lg p)$ stages of $\Theta(p)$ switches each
    - » simultaneous reference of disjoint memories may be serialized
      - due to contention within the network

Multistage Butterfly indirect network (p = 8)



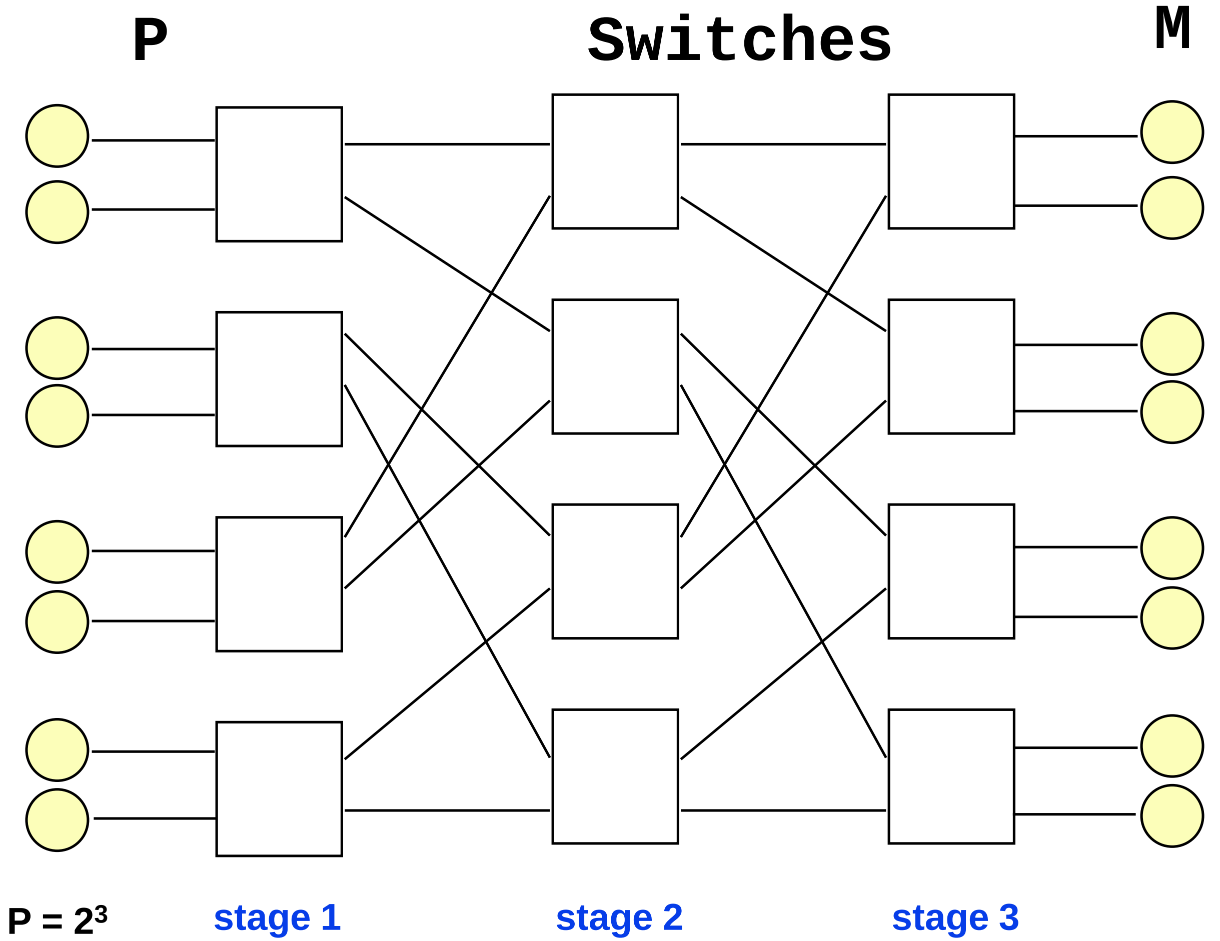
Routing in butterfly networks

- **based on destination address**
  - destination address d_{k-1} ….. d_0
  - in stage i, switch setting is determined by d_{k-i}
    - » switch to top or bottom



Multistage Omega network (p = 8)

- **Isomorphic to butterfly network**
  - same “perfect shuffle” connection pattern between successive stages



Network Topology: Graph-theoretic measures

- **Diameter:** **Maximum length of shortest path between any pair of nodes**

$$\max_{u,v \in N} \left(\min_{u \to v \in C^*} |u \to v| \right)$$

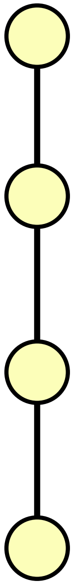
  - i.e. distance between maximally separated nodes - related to latency

- **Bisection width:** **Minimum number of edges crossing approximately equal bipartition of nodes**
  - related to bandwidth with full applied load
  - a *scalable* network has bisection width Ω(p)

- **Degree:** **number of edges (links) per node (switch)**
  - related to cost and switch complexity
  - fixed degree is simpler and more scalable

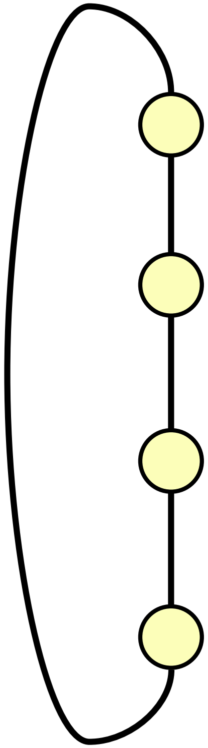
- **Cost:** **number of wires**
  - length of wires and wiring regularity is also an issue

Linear array



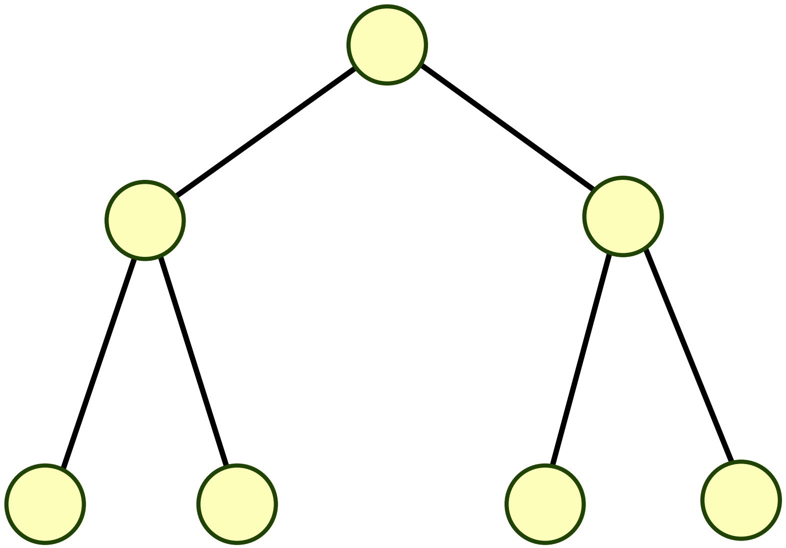
- $|C| = p-1$
- Diameter = p-1
- Degree ≤ 2
- Bisection width = 1

Ring



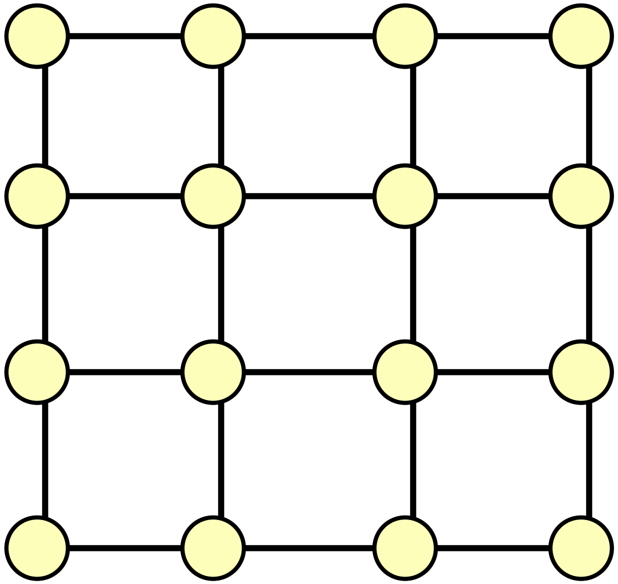
- |C| = p
- Diameter = p/2
- Degree = 2
- Bisection width = 2

Binary Tree



- $|C| = p - 1$
- Diameter = 2 lg p
- Degree ≤ 3
- Bisection width = 1

d-dimensional mesh

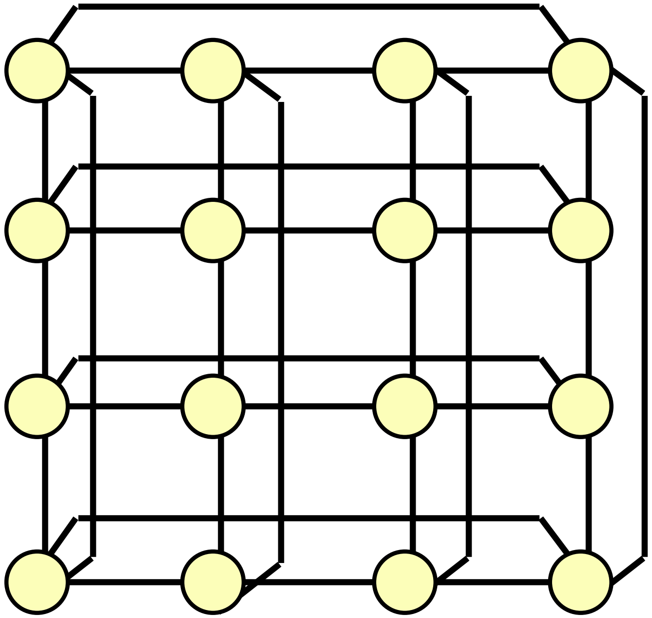


- $p = k^d$
  - Cartesian product of *d* linear arrays with $k = p^{1/d}$ nodes each
- $|C| < 2dp$
  - short wires when $d \leq 3$
- Diameter = $dp^{1/d}$
- $d \leq$ Degree $\leq 2d$
- Bisection width = $p^{(1-1/d)}$

$$\sqrt{p} \times \sqrt{p}$$

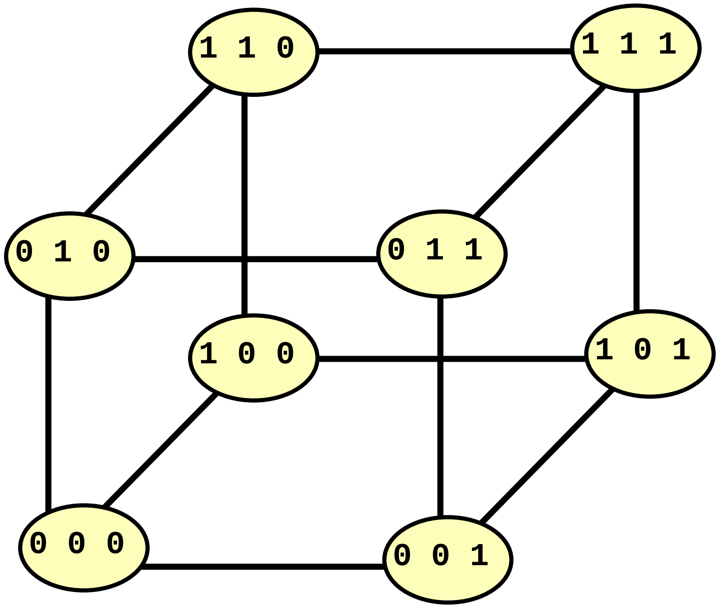
  - 2-D mesh, $d = 2$

k-ary d-cubes



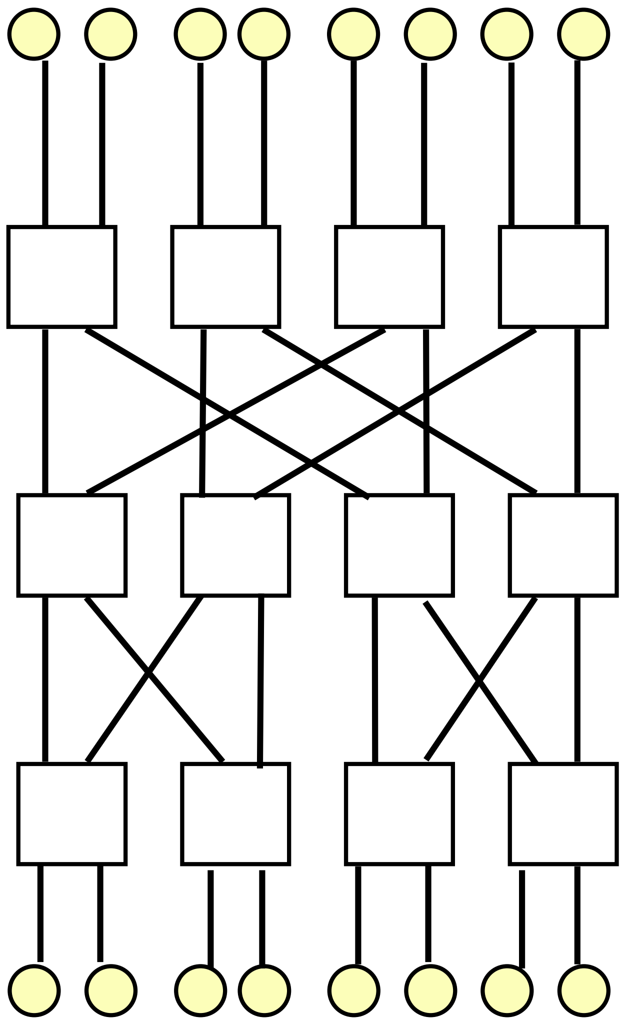
- $p = k^d$
  - Cartesian product of d rings with $k = p^{1/d}$ nodes each
- $|C| = 2dp = 2dk^d$
- Diameter = $dp^{1/d} / 2$
- Degree = $2d$
- Bisection width = $2\ p^{(1-1/d)} = 2k^{d-1}$
  - Ring: p-ary 1-cube
  - 2-D Torus: $\sqrt{p}$ – ary 2 – cube
  - 3-D Torus: $\sqrt[3]{p}$ – ary 3 – cube
  - Hypercube: 2-ary (lg p)-cube

(Boolean) Hypercube



- **$|C| = p \lg p$**
- **Diameter = lg *p***
- **Degree = lg *p***
- **Bisection width = $\Theta(p)$**

Butterfly (Indirect)

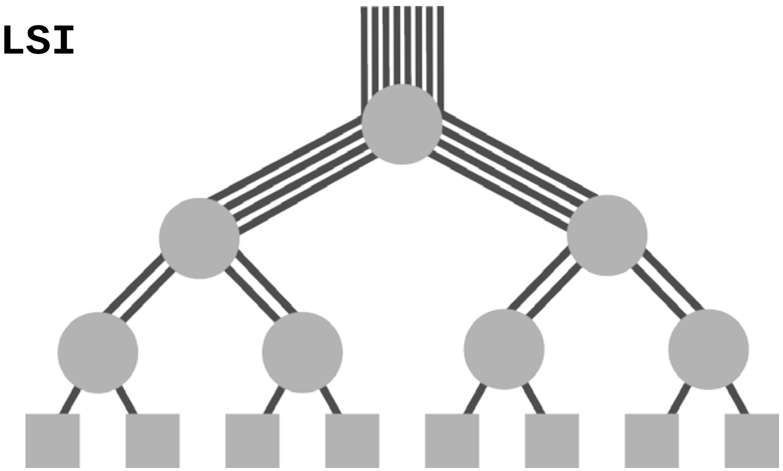


- |C| = p lg p
- Diameter = lg p
- Degree = 2
- "Bisection" width (congestion)
  - There are some bad permutations $\Theta(p^{1/2})$
  - Overwhelming majority have bisection of $\Theta(p)$

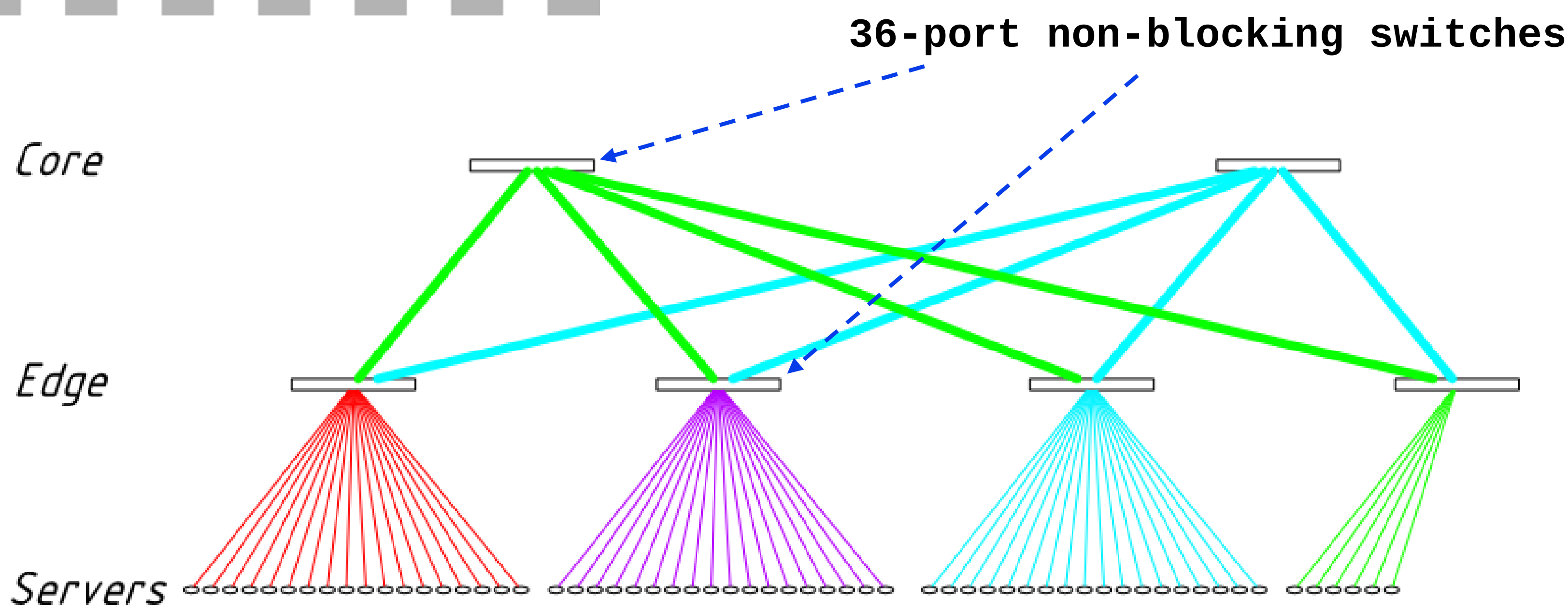
Fat-tree (Indirect)

- $|C| = p \lg p$
- Diameter = $2 \lg p$
- Degree = varying ($2^i \;\; i \in 0..\lg p$)
- Bisection width = $\Theta(p)$

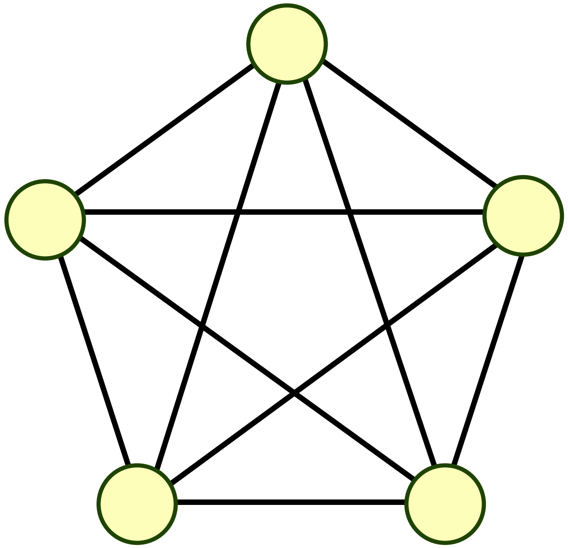
VLSI

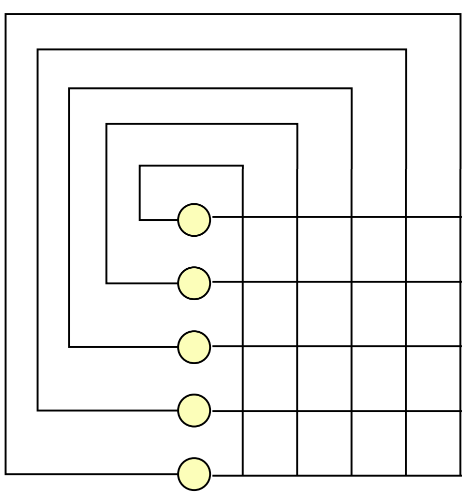


Cluster



Crossbar





- **Complete graph on p nodes**
- **|C| = p(p-1)/2**
- **Diameter = 1**
- **Degree = p-1**
- **Bisection width = $p^2/4$**

Networks in current parallel computers

- **Modern interconnects are indirect**
  - Hardware routing between source and destination

- **Indirect networks**
  - Cluster of commodity nodes
    - » Fat-tree (assembled using 36 port non-blocking switches)
  - IBM Summit (ORNL)
    - » Fat-tree Infiniband [4,608 nodes] (24,000 GPU, 202,752 cores)
  - Fujitsu Fugaku
    - » 6D torus [160,000 nodes k-ary d-cube, ? k~7 d=6] (3M+ cores)

- **Processor – memory interconnects (p procs, m memories)**
  - Tera MTA
    - » 3D torus (p = 256, m = 4,096)
  - NEC SX-9
    - » crossbar (p = 16 procs * 16 channels/proc = 256, m = 8,192)

Routing and flow control

- **System-level networks**
  - Tradeoffs are very different than WAN (TCP)
    - » use flow control instead of dropping packets
    - » mostly static routing instead of dynamic routing
  - Routing algorithm
    - » prescribes a unique path from source to destination
      - e.g. dimension ordered routing on hypercube and lower dimensional d-cubes
      - some networks dynamically “misroute” if a needed link is unavailable
    - » routing can be store-and-forward or cut-through
  - Flow control
    - » contention for output links in a switch can block progress
    - » generally low-latency per-link flow control is used
      - delay in access to a link rapidly propagates back to sender

Communication cost model

- **Message size m bits**

- **Number of hops (links) to travel h**

- **Channel width W and link cycle time t_c**
  - Per-bit transfer time $t_w = t_c/W$
    - » assuming m is sufficiently large

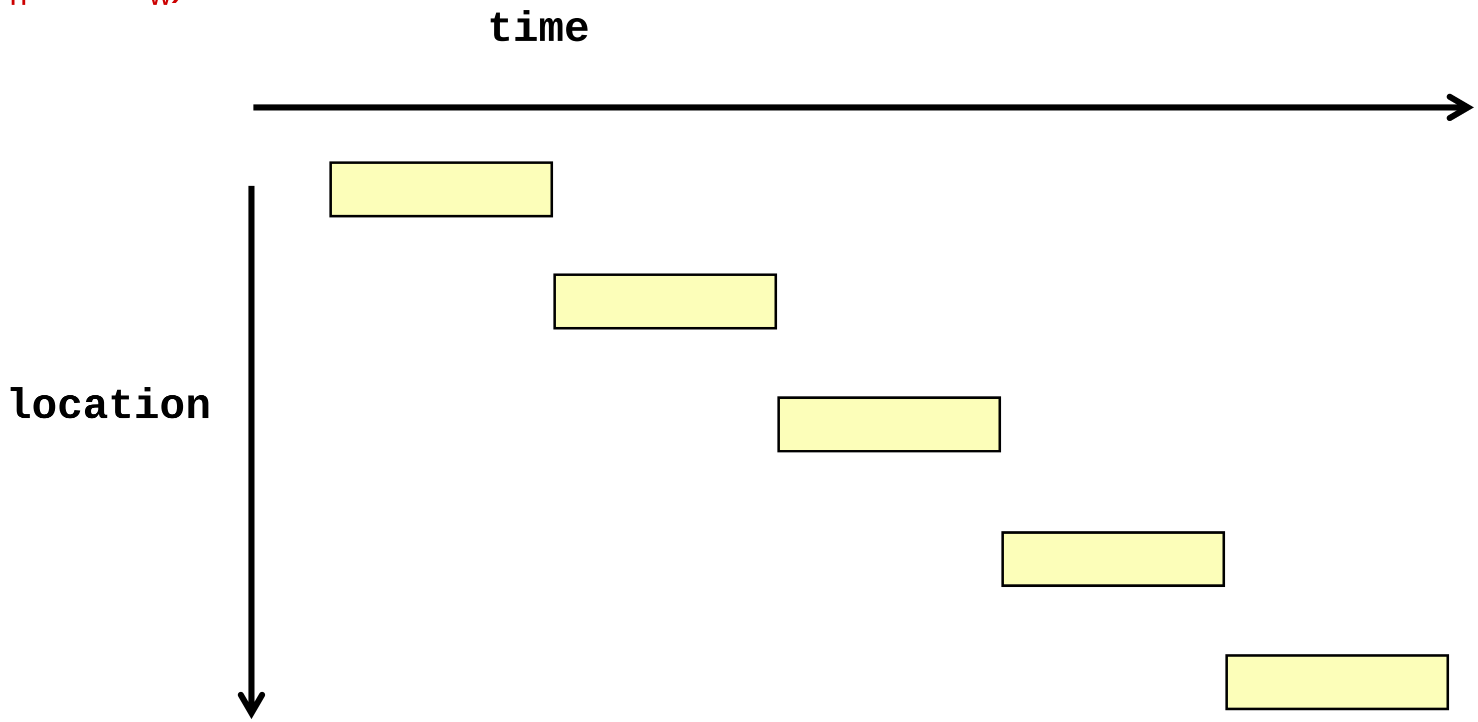
- **Startup time t_s**
  - overhead to insert message into network

- **Node latency or per-hop time t_h**
  - time taken by message header cross channel and be interpreted at destination

Store-and-forward routing

- **flow-control mechanism at message or packet level**
- **packet s are transferred one link at a time**
- **large buffers, high latency**
- **cost**

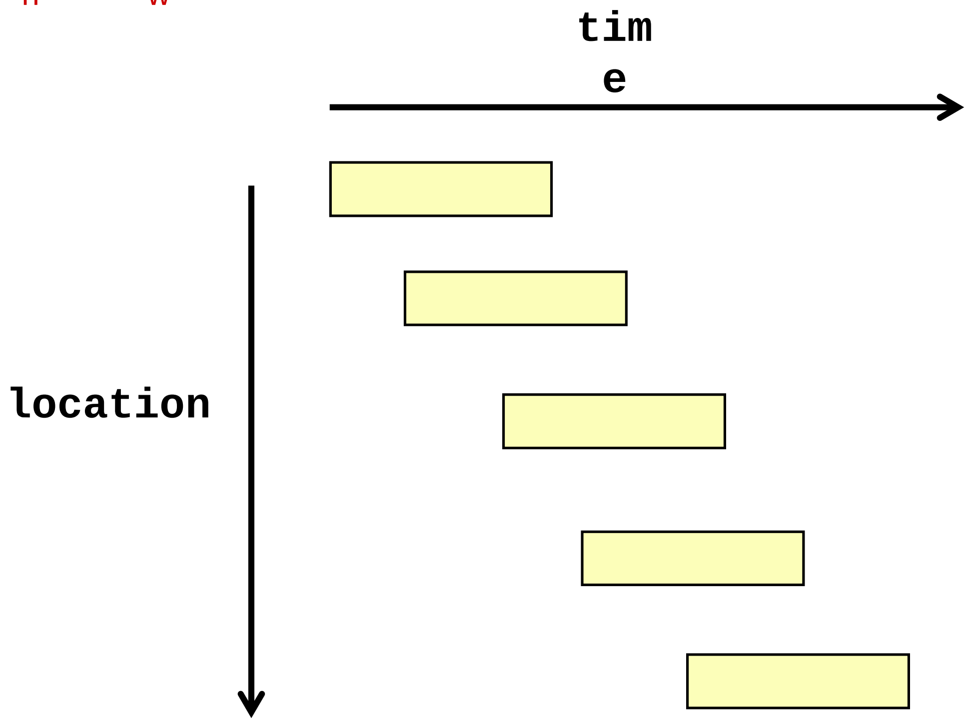
  $t_{SF} = t_s + (t_h + m\ t_w)\ h$



Cut-through routing

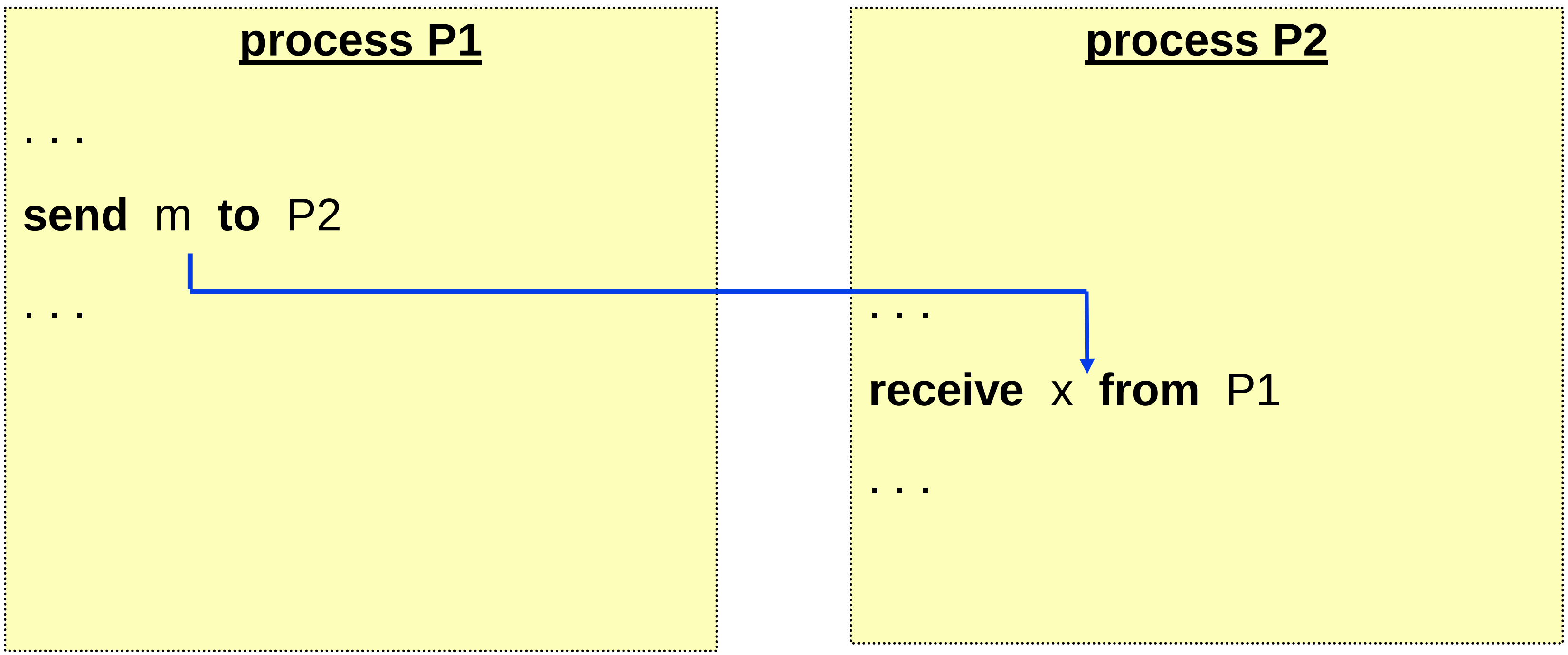
- **flow control is per-link and payload transmission is pipelined**
- **message spread out across multiple links in the network**
- **small buffers, low latency**
- **cost**

$t_{CT} = t_s + ht_h + mt_w$



Basic Interprocess Communication

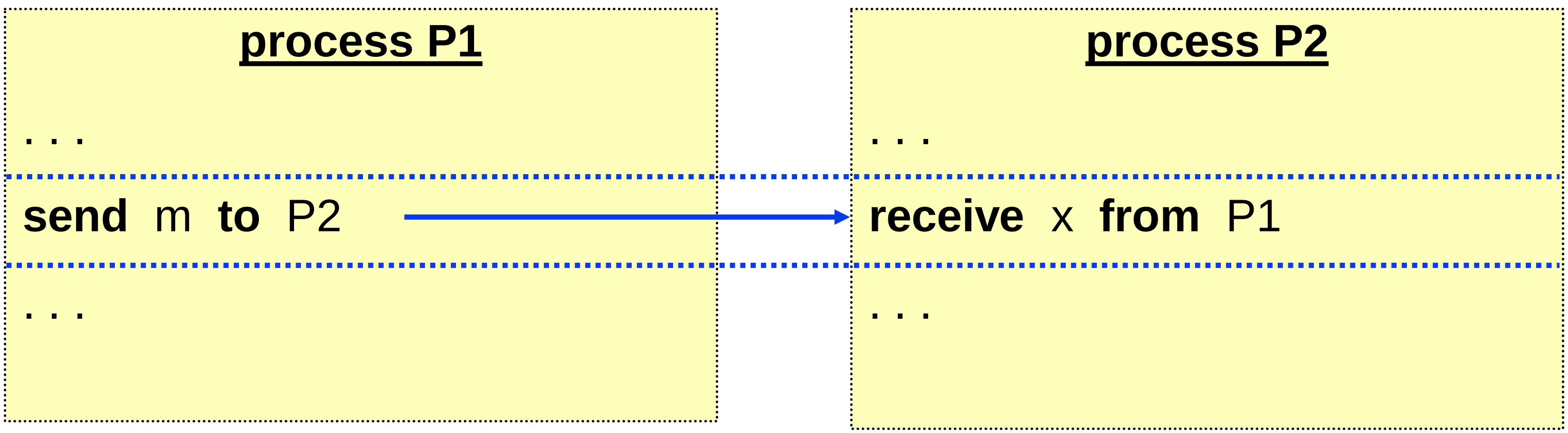
- **Basic building block**
  - message passing: send and receive operations between in different address spaces



How will this really be performed?

Synchronous Message Passing

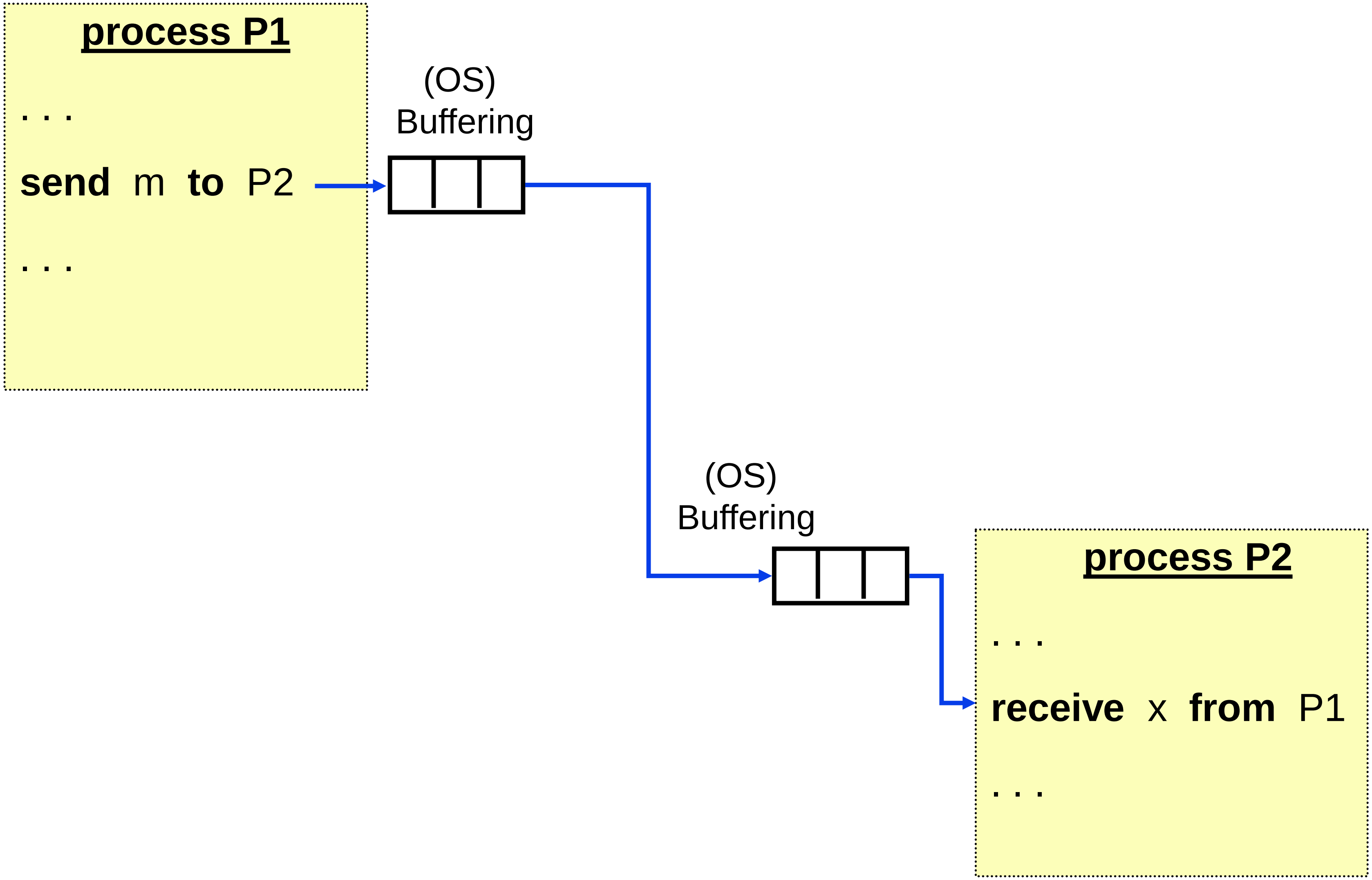
- **Communication upon synchronization**
  - Hoare’s Communicating Sequential Processes (1978)
- **BLOCKING send and receive operations**
  - unbuffered communication
  - several steps in protocol
    - » synchronization, data movement, completion
  - delays participating processes



Asynchronous Message Passing

- **Buffered communication**
  - send/receive via OS-maintained buffers
    - » e.g. pipes or TCP connections
    - » may increase concurrency (e.g. producer/consumer)
    - » may increase transit time
  - send operation
    - » send operation completes when message is completely copied to buffer
    - » generally non-blocking but will block if buffer is full
  - receive operation – two flavors
    - » BLOCKING
      - receive operation completes when message has been delivered
    - » NON-BLOCKING
      - receive operation provides location for message
      - notified when receive complete (via flag or interrupt)

Asynchronous Message Passing



Deadlock in message passing

- **Can concurrent execution of P1 and P2 lead to deadlock?**
  - assuming synchronous message passing?
  - assuming asynchronous message passing?

process P1

. . .

send m1 **to** P2

receive y **from** P2

. . .

process P2

. . .

send m2 **to** P1

receive x **from** P1

. . .

Non-determinism in Message Passing

- In what order should the receive operations be performed?

Two producers

One consumer

process P1

```
...
send m1 to P3
...
```

process P2

```
...
send m2 to P3
...
```

process P3

```
...
receive x from ?
...
receive y from ?
...
```

Here we want

```
receive x from any_process
receive y from any_process
```

Safe communication

- **MPI has four pairwise message passing modes**
  - Synchronous
    - » unbuffered, but all send-receive pairs must synchronize
  - Buffered (asynchronous)
    - » Programmer supplies (sufficient) buffer space
  - Ready
    - » Receiver guaranteed to be ready to receive at the time of the send
  - "Standard"
    - » OS Buffered for small messages, synchronous for large messages

- **Most programs rely on a certain amount of buffering in communication**
  - SPMD programming models: send, then receive
  - Nondeterminacy: receive from left, receive from right

- **Most programs use standard model**
  - Dangerous, as buffer size is system-dependent

Destination naming

- **How are messages addressed to their receiver?**
  - Static process to processor mapping
    - » Fixed set of processes at compile time
    - » *mapper* statically assigns processes to processors at run time.
    - » Ex: Communicating Sequential Processes (CSP)
  - Semi-dynamic process to processor mapping (SPMD)
    - » Unknown set of processes at compile time
    - » Fixed set of processes at run time
    - » fixed mapping over execution lifetime
    - » Ex: MPI communicators
  - Dynamic process to processor mapping
    - » Unknown set of processes at compile time
    - » Processes may be created or moved dynamically at run time
    - » Communication requires lookup
    - » MPI-2

Data Representation

- **In general, prefer to send an abstract data type (ADT) rather than single elements**
  - ADTs represent abstractions suited to application
  - higher performance can be obtained for large messages
    - » e.g. aggregate data types
- **How are components of an ADT combined together?**
  - data marshalling
    - » packing components into a send buffer
- **How is a message represented as a sequence of bits?**
  - encoding must be suitable for source and destination
    - » XDR (eXternal Data Representation)
- **How is a message disassembled into an ADT?**
  - data unmarshalling
    - » extracting components from a receive buffer

Message Selection

- **Receiving process may need to receive message from multiple potential senders**
  - How to specify/distinguish message to be received?
    - » sender selection (socket, MPI, CSP)
    - » message data type selection (MPI, CSP)
    - » condition selection (CSP)
    - » message “tag” (MPI)
  - specification of message to be received can decrease nondeterminacy
    - » Non-deterministic reception order requires care with blocking sends/receives

Message Passing Interface (MPI)

- **A library of communication operations for distributed-memory parallel programming**
  - history
    - » TCP/IP, …., PVM (1990), MPI (1994), MPI-2 (1997), MPI-3 (2012), Open MPI v5 (2021)
  - programming model
    - » SPMD - single program with library calls
  - MPI functionality
    - » send/receive, synchronization, collective communication
    - » MPI specifies 129 procedures
      - widely implemented and generally efficient
    - » MPI-2 adds one-sided communication, dynamic processes, parallel I/O and more
      - One-sided communication: remote direct memory access – good for BSP.
      - Over 15 years from full specification to correct and (generally) efficient implementations
    - » MPI-3
      - Tweaks and shared memory segments between MPI processes
  - portability
    - » MPI is the most portable parallel programming paradigm – it runs on
      - shared and distributed memory machines
      - homogeneous and heterogeneous systems
      - variety of interconnection networks
    - » BUT functional portability ≠ performance portability !

MPI Example (C + MPI)

```
#include <mpi.h>
main(int argc, char **argv) {
  int nproc, myid;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank (MPI_COMM_WORLD, &myid);

  printf("Hello World! Here is process %d of %d.\n",
             myid, nproc);

  MPI_Finalize ();
}
```

At UNC, the dogwood cluster implements MPI

MPI return codes

```
#include <mpi.h>
#include <stdio.h>
#include <err.h>
main(int argc, char **argv) {
  int nproc, myid, ierr;

  ierr = MPI_Init(&argc, &argv);
  if (ierr != MPI_SUCCESS) err(4, "Error %d in MPI_Init\n", ierr);

  ierr =  MPI_Comm_size (MPI_COMM_WORLD, &nproc);
  if (ierr != MPI_SUCCESS) err(4, "Error %d in MPI_Comm_size\n", ierr);

  ierr = MPI_Comm_rank (MPI_COMM_WORLD, &myid);
  if (ierr != MPI_SUCCESS) err(4, "Error %d in MPI_Comm_rank\n", ierr);

  printf("Hello World! Here is process %d of %d.\n", myid, nproc);

  ierr = MPI_Finalize();
  if (ierr != MPI_SUCCESS) err(4, "Error %d in mpi_finalize\n", ierr);
}
```

Point-to-point communication

- **Specification of message to receive**
      - » communicator – identifies logical set of processors
          - intracommunicator vs. intercommunicator
      - » sending process rank (= proc id)
      - » tag
  - – details of received message via status parameter
      - » wildcard specifications may result in non-deterministic programs

- **Type Specification**
  - – must provide types of transmitted values
      - » predefined types & user-defined types
      - » implicit conversions in heterogeneous* systems

- **Protocol specification**
  - – send
      - » blocking / non-blocking / repeated / …
          - standard / buffered / synchronous / “ready”

Simple message exchange

- no deadlock
- two sequential transfers

Addr of data to send

Number of elements

Element type

Destination rank

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
```

Process 0:

```
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
```

Process 1:

```
MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
```

Non-blocking message exchange

- no deadlock
- possibility of concurrent transfer

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD

MPI_Request request;
MPI_Status status;
```

Process 0:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request);
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
MPI_Wait(&request, &status);
```

Process 1:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request);
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
MPI_Wait(&request, &status);
```

Overlapping communication and computation

Process 0 and 1:

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD

MPI_Request requests[2];
MPI_Status statuses[2];

// p is process id of the partner in a pairwise exchange

MPI_Irecv(B, 100, MPI_DOUBLE, p, 0, WORLD, &request[1]);
MPI_Isend(A, 100, MPI_DOUBLE, p, 0, WORLD, &request[0]);
```

.... do some useful work here ....

```
MPI_Waitall(2, requests, statuses);
```

- no deadlock
- concurrent transfer
- communication and computation may be overlapped on some machines
  - requires hardware communication support

Communicators

- **MPI_COMM_WORLD is a communicator**
  – group of processes numbered 0 ... p-1
  – set of logical communication channels between them
- **Message sent with one communicator cannot be received in another communicator**
  – all communication is intra-communicator
  – enables development of safe libraries
  – restricting communication to subgroups is useful
- **Creating new communicators**
  – duplication
  – splitting
- **Intercommunicators**
  – orchestrate communication between two different communicators

Collective Communication

- **Operations involve all processes in an (intra)communicator**
  - encapsulate important communication patterns (cf. BSP)
    - » broadcast
    - » total exchange (transpose)
    - » reduction + scan
    - » barrier
  - operations do not necessarily imply a barrier synchronization
    - » however, all processes must issue the same collective communication operations in the same order

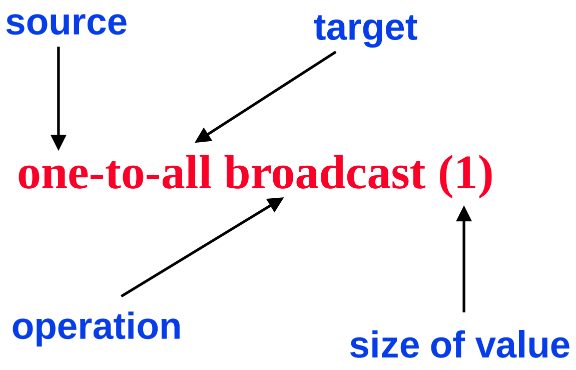
- **Type specification**
  - predefined or user-defined types
  - predefined or user-defined associative operation for reduction & scan

- **Distinguished process**
  - for broadcast or reduction operations

Collective communication operations

- **classified by**
  - source of values
    - » one/all processor(s)
  - target of result
    - » one/all processors(s)
  - operation
    - » broadcast
    - » exchange
    - » accumulate (reduce)
  - size of values
    - » 1 or n

Ex:

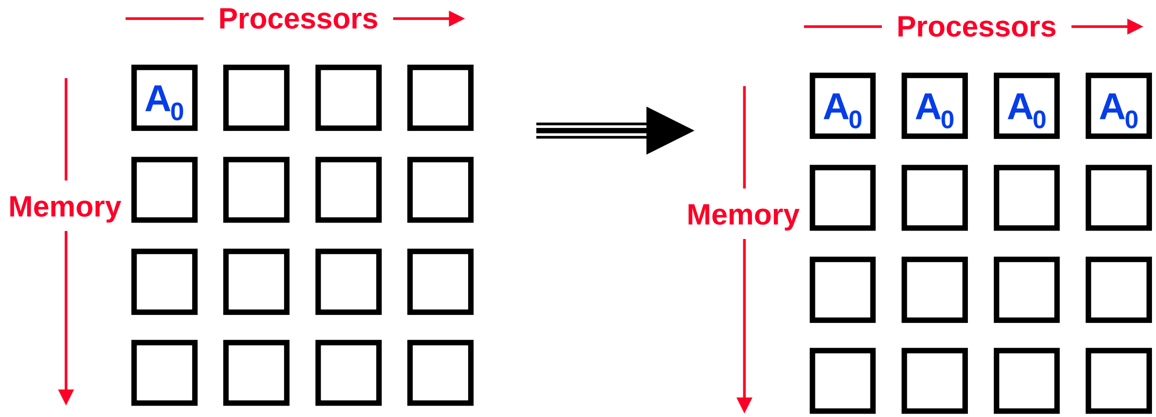


- **duality of communication operations**
  - communication patterns are related
  - broadcast & reduction are duals
  - exchange is its own dual

Broadcast: single source, single value

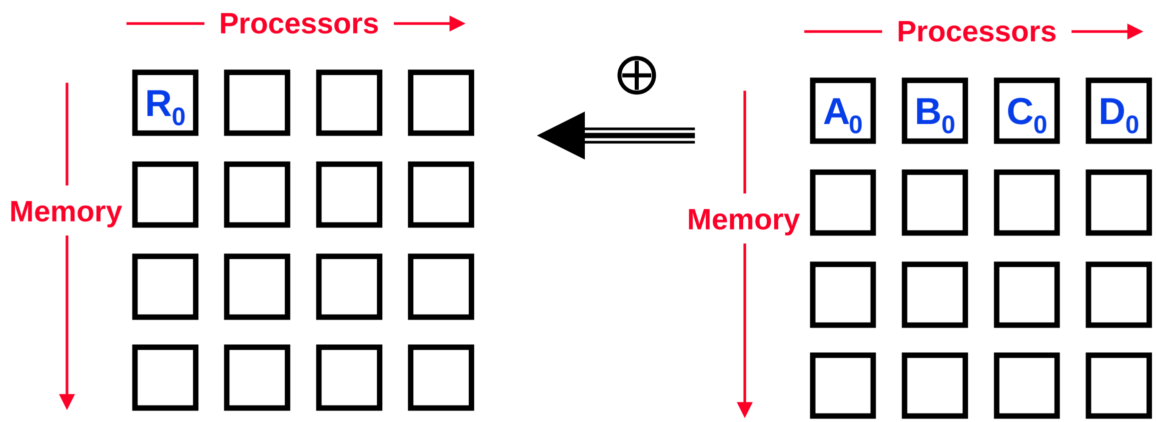
one-to-all broadcast (1)

`MPI_Bcast(…1…)`



all-to-one sum (1)

`MPI_Reduce(…1…)`

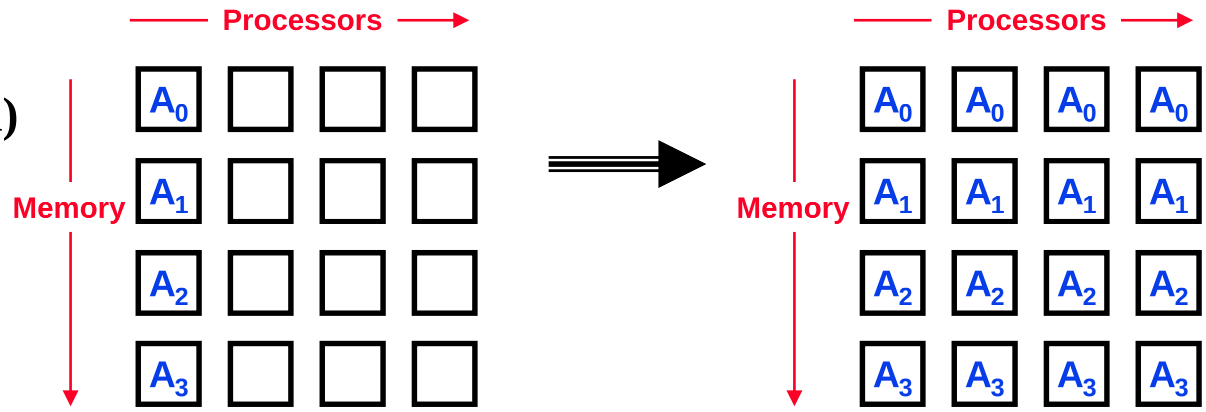


$R_0 = A_0 \oplus B_0 \oplus C_0 \oplus D_0$

Broadcast: single source, multiple values

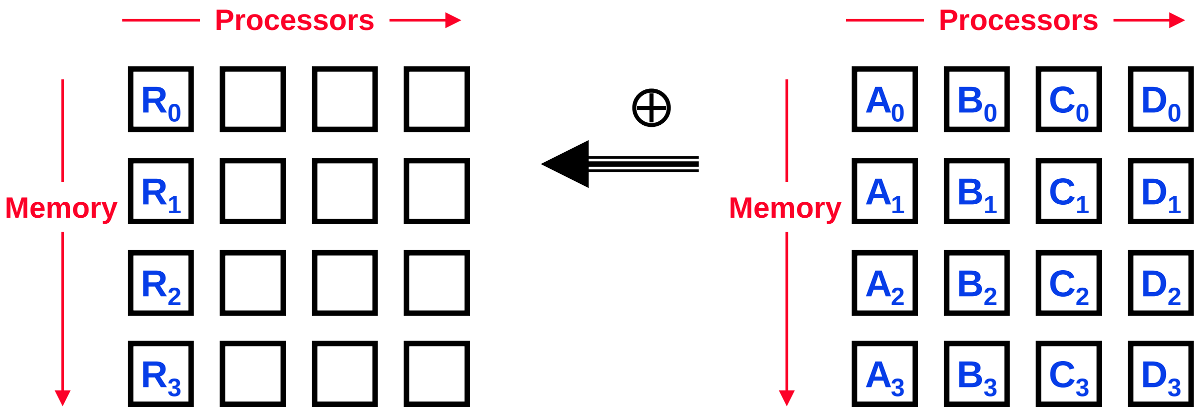
one-to-all broadcast (n)

`MPI_Bcast(…n…)`



all-to-one sum (n)

`MPI_Reduce(…n…)`

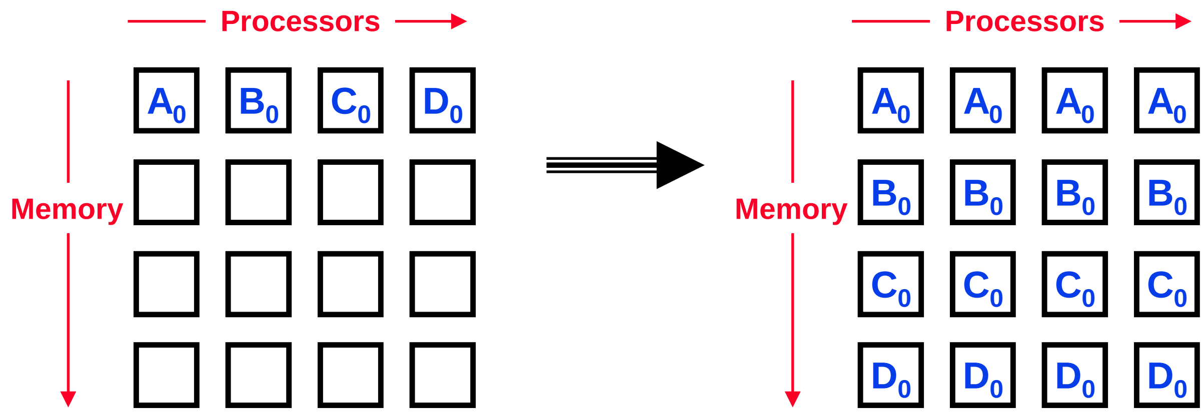


$R_i = A_i \oplus B_i \oplus C_i \oplus D_i$

Broadcast: multiple source, single value

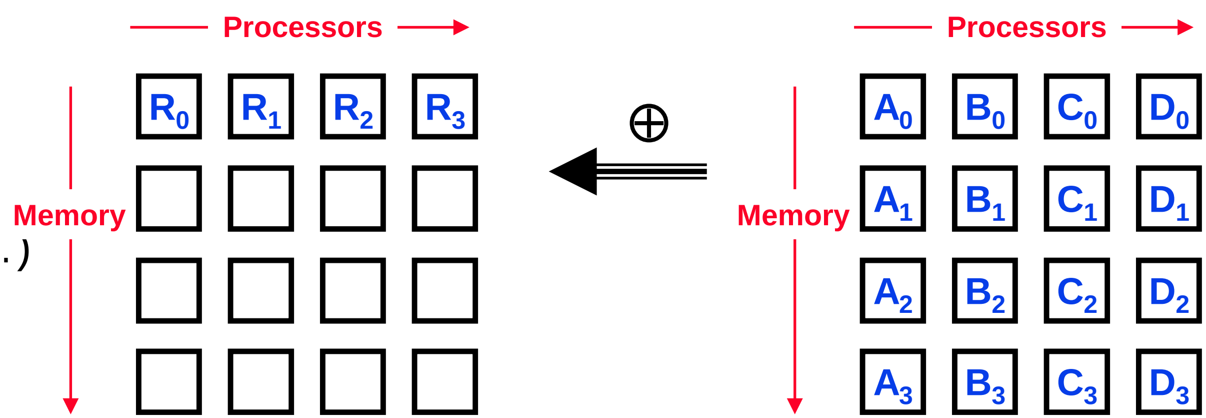
all-to-all broadcast (1)

`MPI_Allgather(…n…)`



all-to-all sum (1)

`MPI_Reduce_scatter(…n…)`



$R_i = A_i \oplus B_i \oplus C_i \oplus D_i$

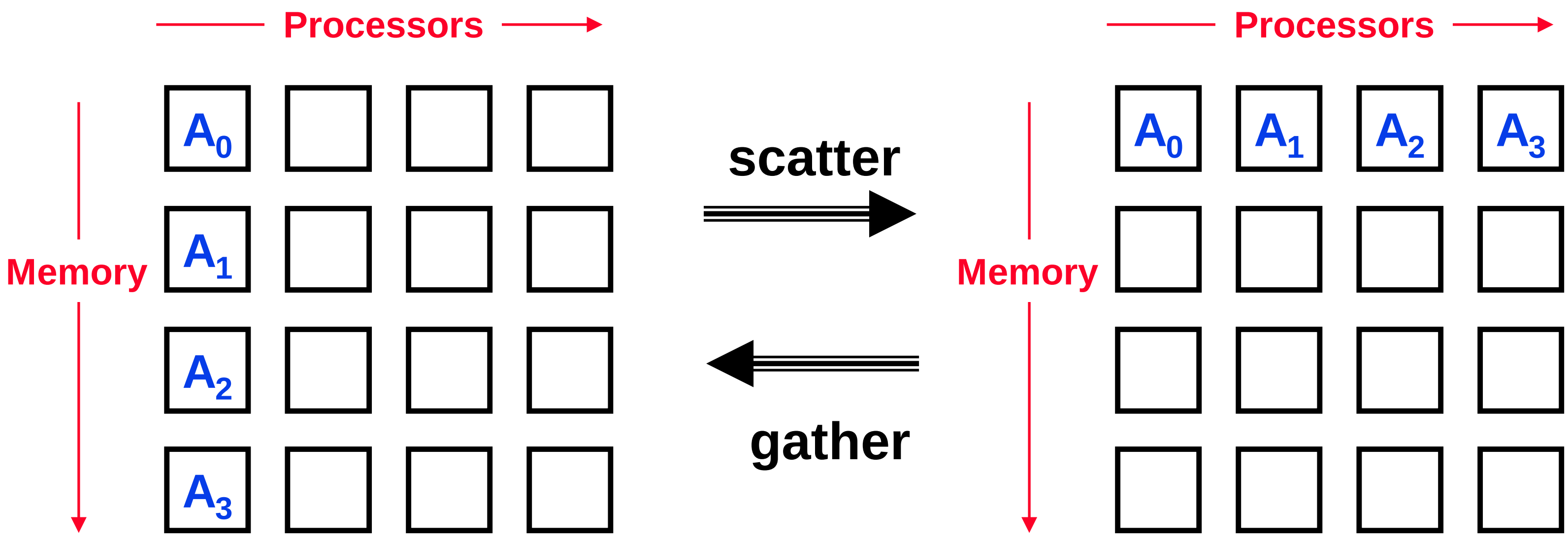
Exchange: single source or single target

- **One-to-all exchange (n)**

  `MPI_Scatter( … )`

- **All-to-one exchange (1)**

  `MPI_Gather( … )`

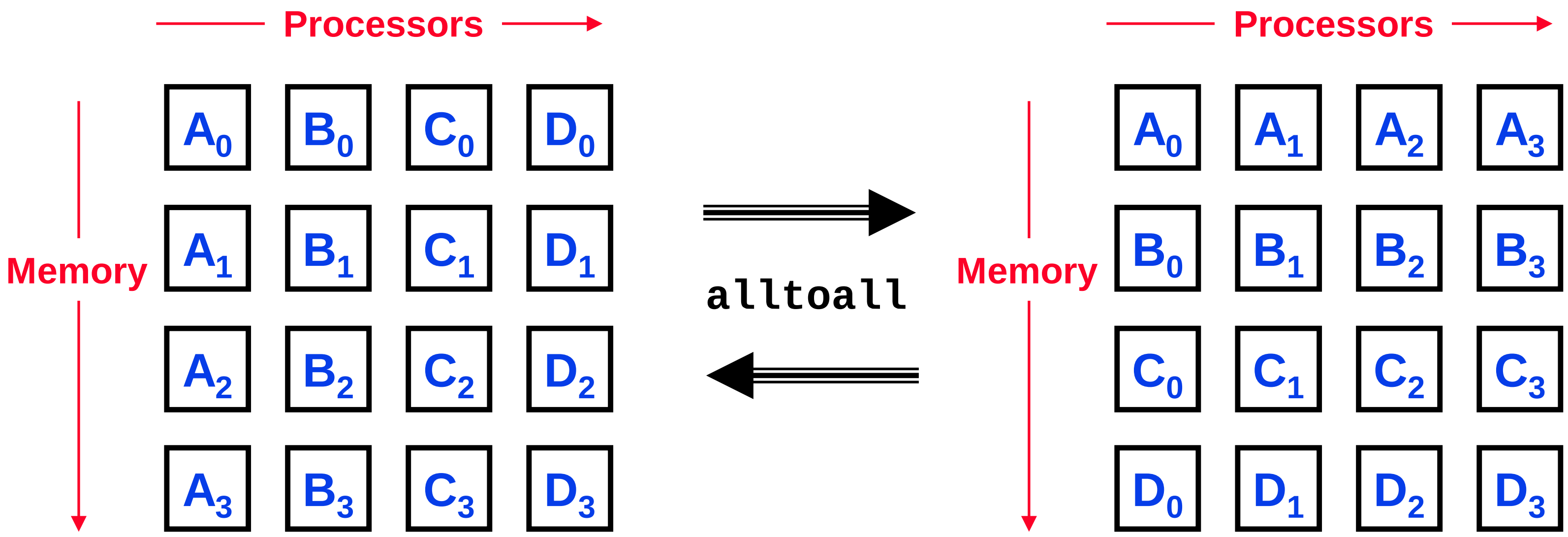


Exchange: multiple source, multiple values

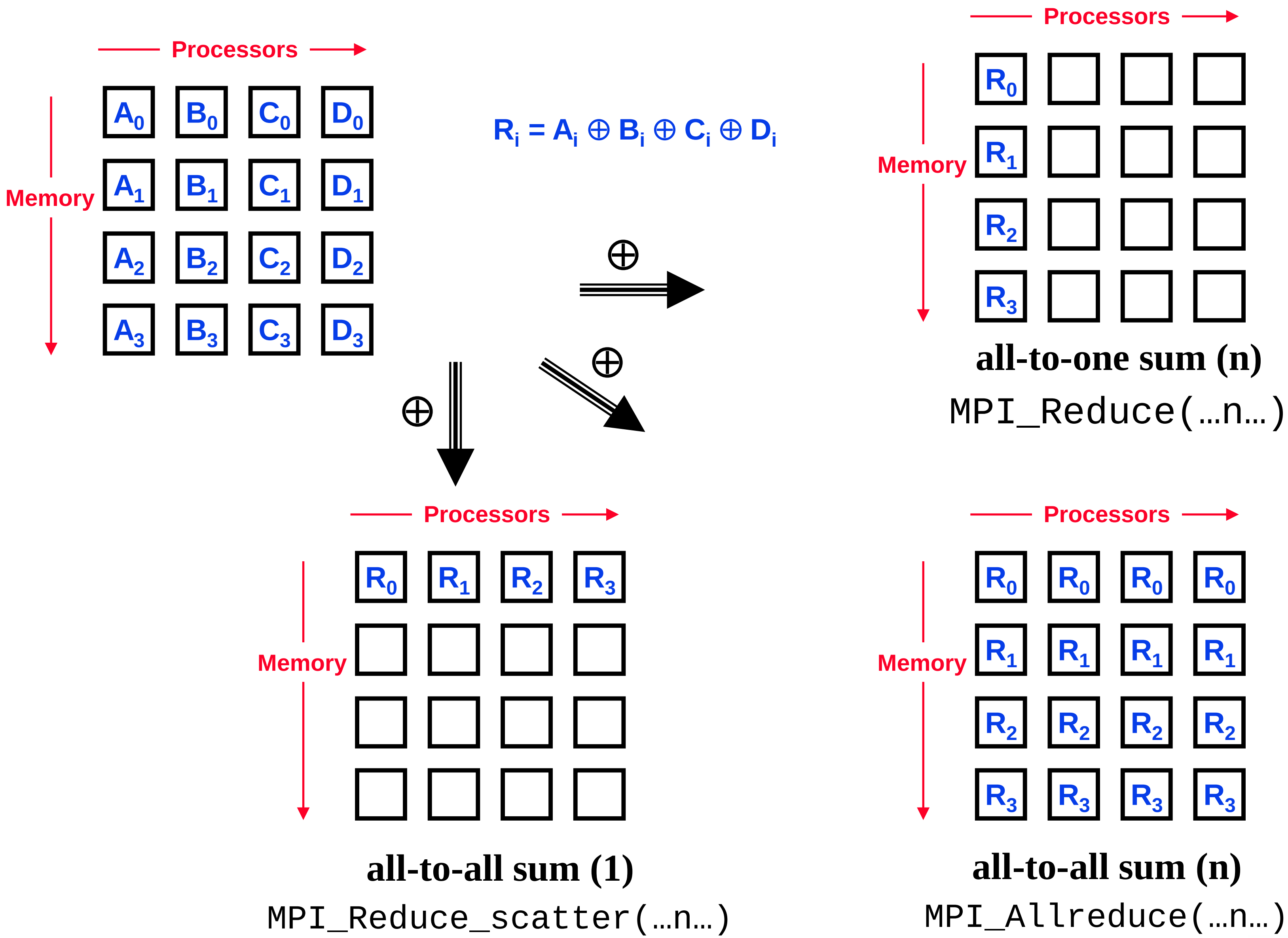
- **all-to-all exchange (n)**

  `MPI_Alltoall(…)`

  – BSP “total exchange” or transpose



Reductions: multiple source, multiple values



MPI: All-pairs N-body problem

- **Problem**
  - n bodies
    - each body position occupies d words
  - for each body i
    - » accumulate total force f_i
      - each pairwise interaction requires c_1 FLOPS
    - » update velocities and positions
      - each body update requires c_2 FLOPS
  - half-pairs optimization: $f_{ij} = -f_{ji}$

$$f_i = \sum_{\substack{j \in 1:n, \\ j \neq i}} \frac{G m_i m_j r_{ij}}{\left| r_{ij} \right|^3}$$

- **MPI solution strategies**
  - ring communication pattern
    - » all-pairs
    - » half-pairs
  - collective communication
    - » all-pairs
    - » half-pairs

Running your projects

- **Shared memory**
  - use phaedra.cs.unc.edu
    - » p = 20 primary cores, 20 secondary cores
- **Distributed memory**
  - use dogwood.unc.edu (requires a cluster account)
- **GPUs**
  - use departmental GPUs
  - use SNP nodes on longleaf.unc.edu