Parallel Systems Course: Chapter III

The Message-Passing Paradigm

Jan Lemeire
Dept. ETRO
October - November 2016

Vrije Universiteit Brussel

Overview

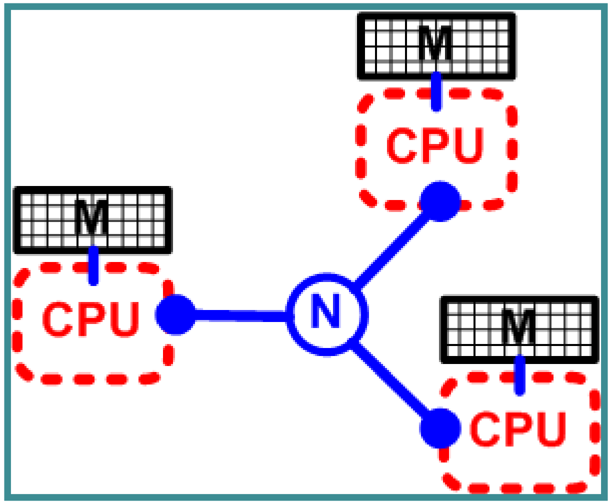
1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes

Overview

1. **Definition**
2. **MPI**
   - Efficient communication
3. **Collective Communications**
4. **Interconnection networks**
   - Static networks
   - Dynamic networks
5. **End notes**

KUMAR p233

Message-passing paradigm



- Partitioned address space
  - Each process has its own exclusive address space
  - Typical 1 process per processor
- Only supports explicit parallelization
  - Adds complexity to programming
  - Encourages locality of data access
- Often Single Program Multiple Data (SPMD) approach
  - The same code is executed by every process.
  - Identical, except for the master
  - *loosely synchronous* paradigm: between interactions (through messages), tasks execute completely asynchronously

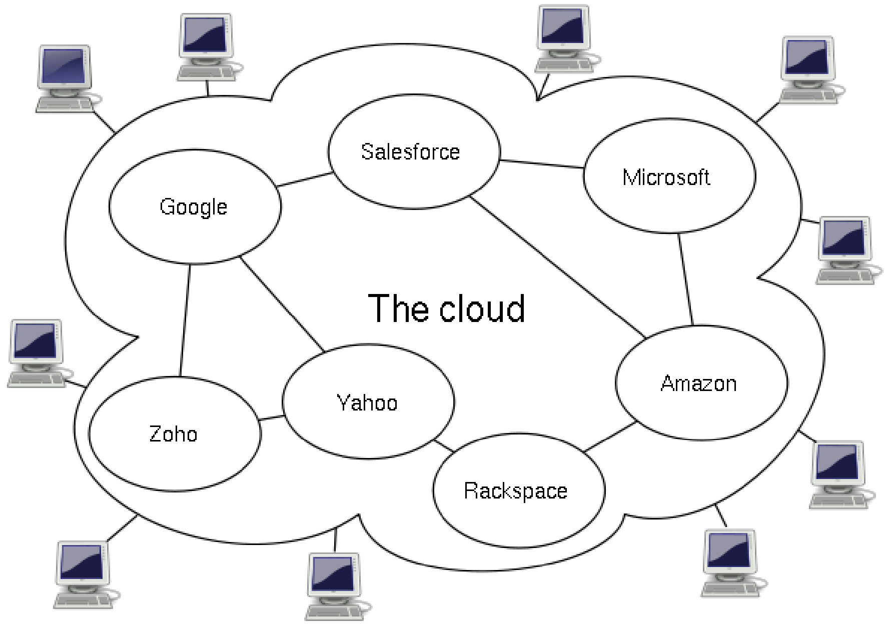
Clusters

- Message-passing
- Made from commodity parts
  - or blade servers
- Open-source software available

Computing Grids

- Provide computing resources as a service
  - Hiding details for the users (transparency)
  - Users: enterprises such as financial services, manufacturing, gaming, …
  - Hire computing resources, besides data storage, web servers, etc.
- Issues:
  - Resource management, availability, transparency, heterogeneity, scalability, fault tolerance, security, privacy.

Cloud Computing, the new hype

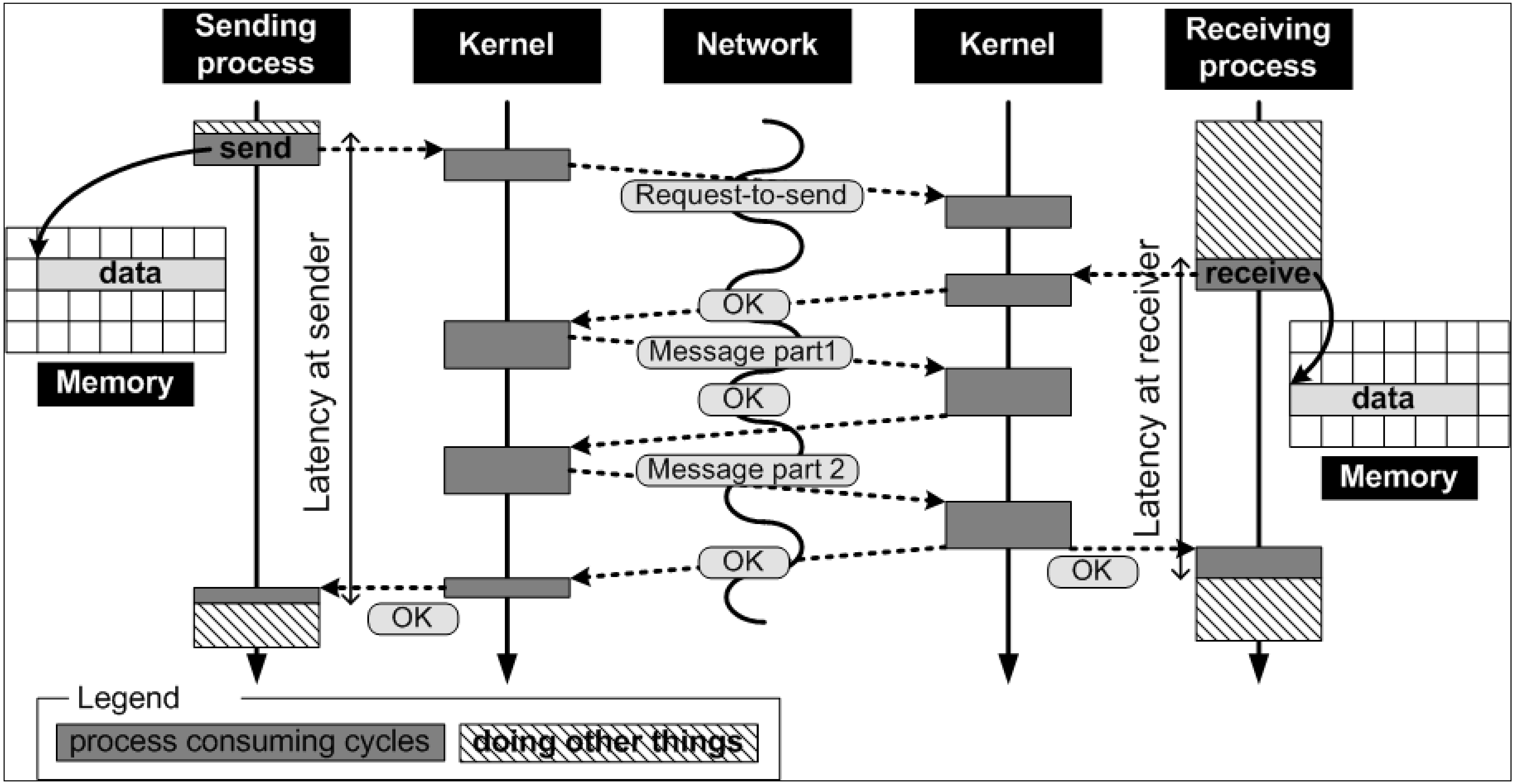


- Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand
- Like the electricity grid.

Messages…

- The ability to send and receive messages is all we need
  - void Send(message, destination)
  - char[] Receive(source)
  - boolean IsMessage(source)
- But… we also want performance!
  - More functions will be provided

Message-passing

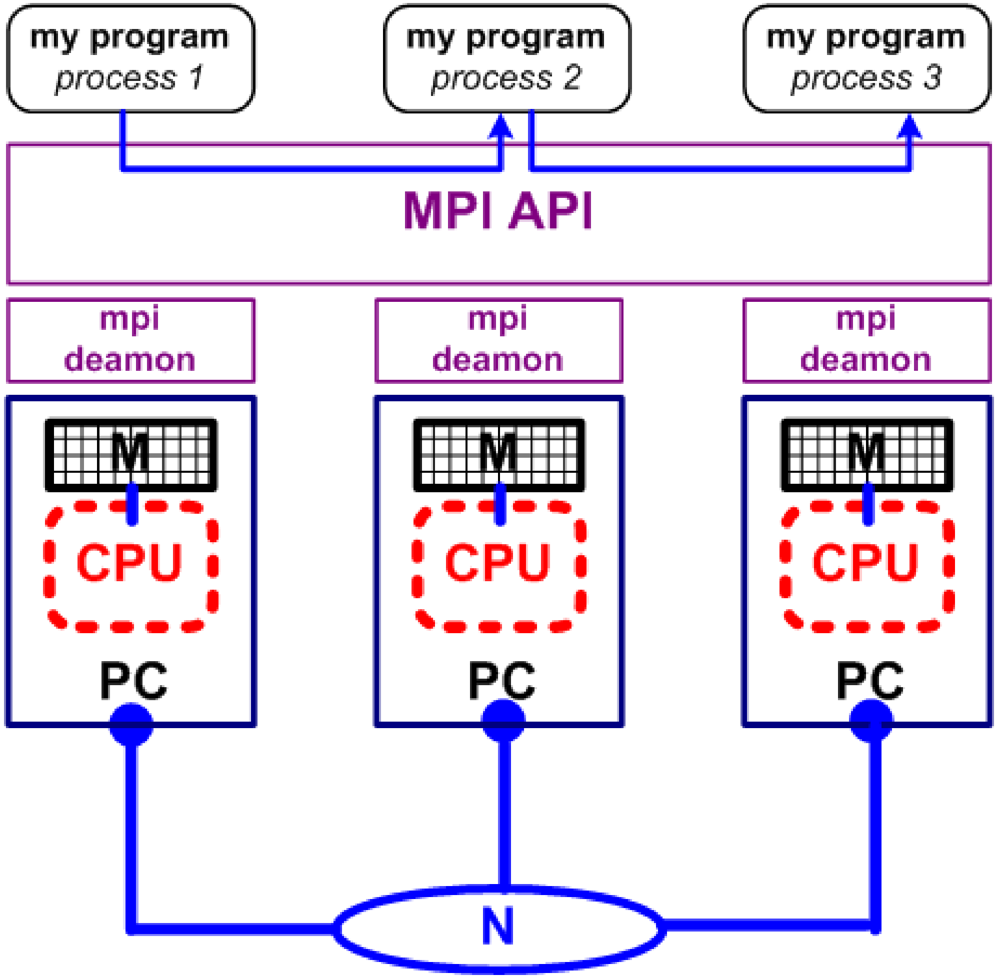


Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes

MPI: the Message Passing Interface

- A standardized message-passing API.
- There exist nowadays more than a dozen implementations, like LAM/MPI, MPICH, etc.
- For writing portable parallel programs.
- Runs transparently on heterogeneous systems (platform independence).
- Aims at not sacrificing efficiency for genericity:
  - encourages overlap of communication and computation by nonblocking communication calls



- Replaces the good old PVM (Parallel Virtual Machine)

Fundamentals of MPI

- Each process is identified by its **rank**, a counter starting from 0.
- **Tags** let you distinguish different types of messages
- **Communicators** let you specify groups of processes that can intercommunicate
  - Default is `MPI_COMM_WORLD`
- All MPI routines in C, data-types, and constants are prefixed by "`MPI_`"
- We use the MPJ API, an O-O version of MPI for java

LINK 2

The minimal set of MPI routines

MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of calling process.
MPI_Send	Sends a message.
MPI_Recv	Receives a message.
MPI_Probe	Test for message (returns `Status` object).

Counting 3s with MPI

master	slaves
partition array	
send subarray to each slave	receive subarray
	count 3s
	return result
receive results and sum them	

- Different program on master and slave
  - We'll see an alternative later

```
int rank = MPI.COMM_WORLD.Rank();  int size = MPI.COMM_WORLD.Size();  int nbrSlaves = size - 1;
if (rank == 0) { // we choose rank 0 for master program
   // initialise data
    int[] data = createAndFillArray(arraySize);
    // divide data over slaves
    int slavedata = arraySize / nbrSlaves; // # data for one slave
    int index = 0;

    for (int slaveID=1; slaveID < size; slaveID++) {
        MPI.COMM_WORLD.Send(data, index,  slavedata + rest, MPI.INT, slaveID, INPUT_TAG);
       index += slavedata;
    }
    // slaves are working...
      int nbrPrimes = 0;
    for (int slaveID=1; slaveID < size; slaveID++){
          int buff[] = new int[1]; // allocate buffer size of 1
          MPI.COMM_WORLD.Recv(buff, 0, 1, MPI.INT, slaveID, RESULT_TAG);
          nbrPrimes += buff[0];
     }
} else {  // *** Slave Program ***
    Status status = MPI.COMM_WORLD.Probe(0, INPUT_TAG);
     int[] array = new int[status.count]; // check status to know data size
     MPI.COMM_WORLD.Recv(array, 0, status.count, MPI.INT, 0, INPUT_TAG);

      int result = count3s(array); // sequential program

     int[] buff = new int[] {result};
     MPI.COMM_WORLD.Send(buff, 0, 1, MPI.INT, 0, RESULT_TAG)
}
MPI.Finalize(); // Don't forget!!
```

MPJ Express primitives

- void Comm.Send(java.lang.Object buf, int offset, int count, Datatype datatype, int dest, int tag)
- Status Comm.Recv(java.lang.Object buf, int offset, int count, Datatype datatype, int source, int tag)

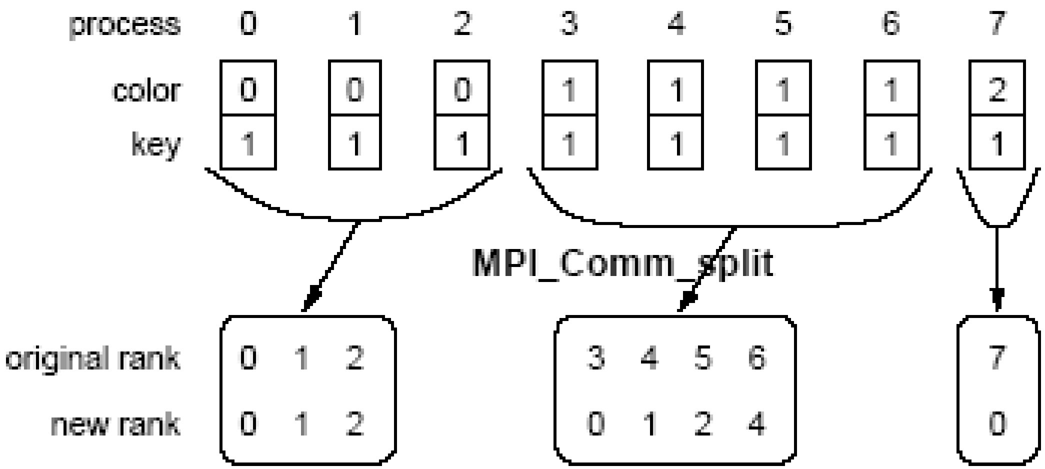
Java array

Communicators

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
  - Default is COMM_WORLD, includes all the processes
  - Define others when communication is restricted to certain subsets of processes
- Information about communication domains is stored in variables of type `Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.

KUMAR p237

Example

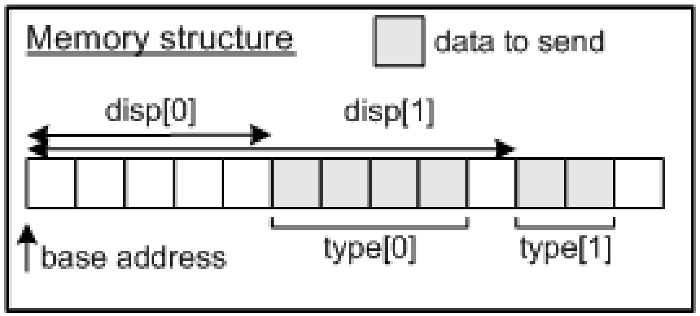


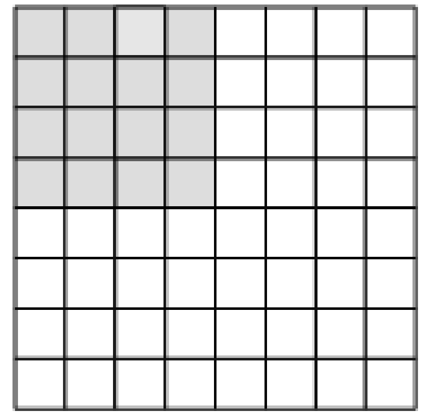
- A process has a specific rank in each communicator it belongs to.
- ***Other example***: use a different communicator in a library than application so that messages don't get mixed

MPI Datatypes

MPI++ Datatype	C Datatype	Java
MPI.CHAR	signed char	char
MPI.SHORT	signed short int	
MPI.INT	signed int	int
MPI.LONG	signed long int	long
MPI.UNSIGNED_CHAR	unsigned char	
MPI.UNSIGNED_SHORT	unsigned short int	
MPI.UNSIGNED	unsigned int	
MPI.UNSIGNED_LONG	unsigned long int	
MPI.FLOAT	float	float
MPI.DOUBLE	double	double
MPI.LONG_DOUBLE	long double	
MPI.BYTE		byte
MPI.PACKED		

User-defined datatypes

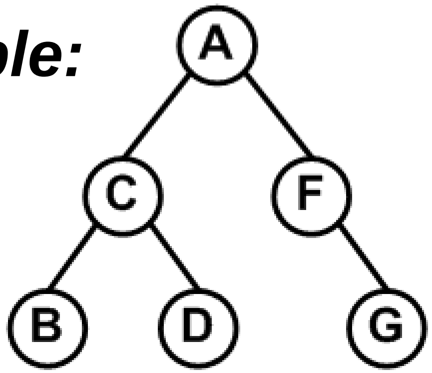


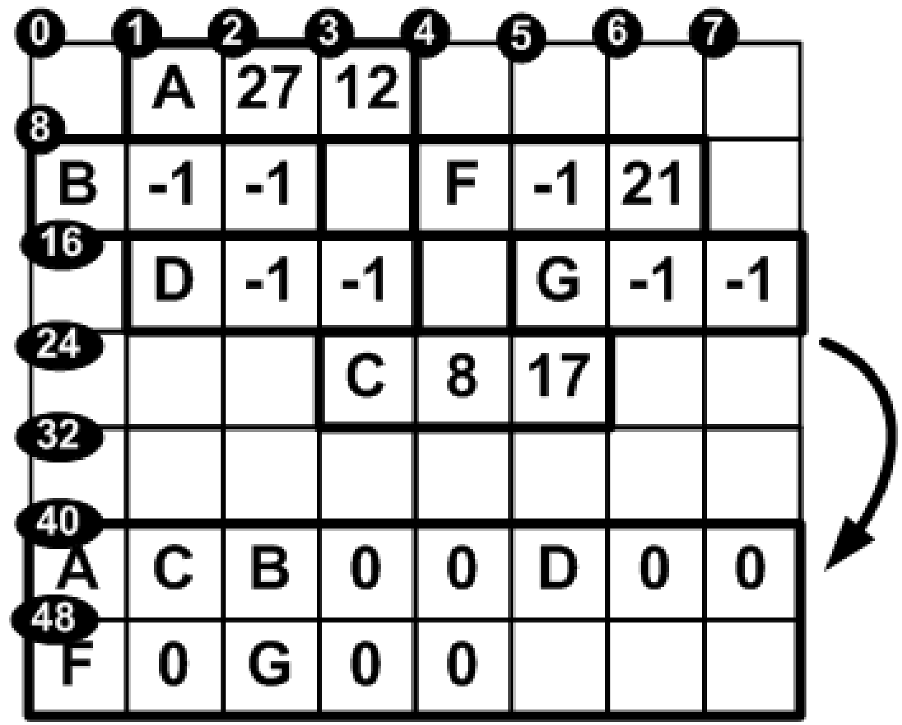


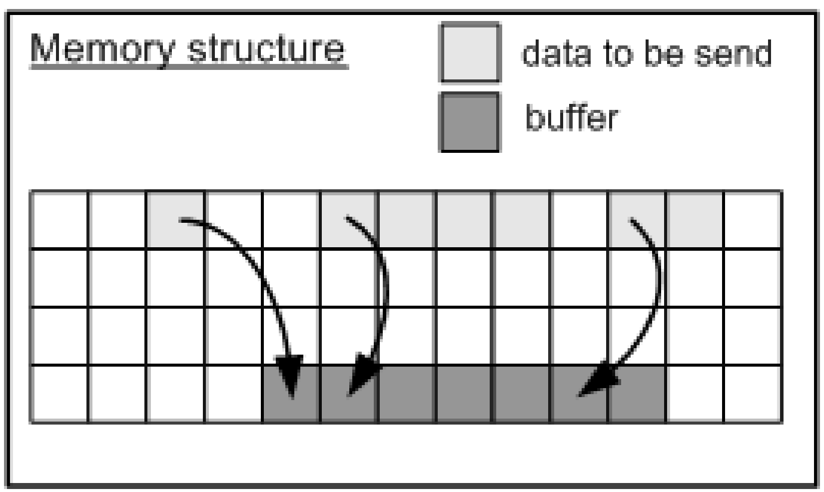
- Specify displacements and types => commit
- Irregular structure: use DataType.Struct
- Regular structure: Indexed, Vector, …
  - E.g. submatrix
- Alternative: packing & unpacking via buffer

Packing & unpacking

Example: tree







From objects and pointers to a linear structure… and back.

Inherent serialization in java

- For your class: implement interface *Serializable*
  - No methods have to be implemented, this turns on automatic serialization
- Example code of writing object to file:

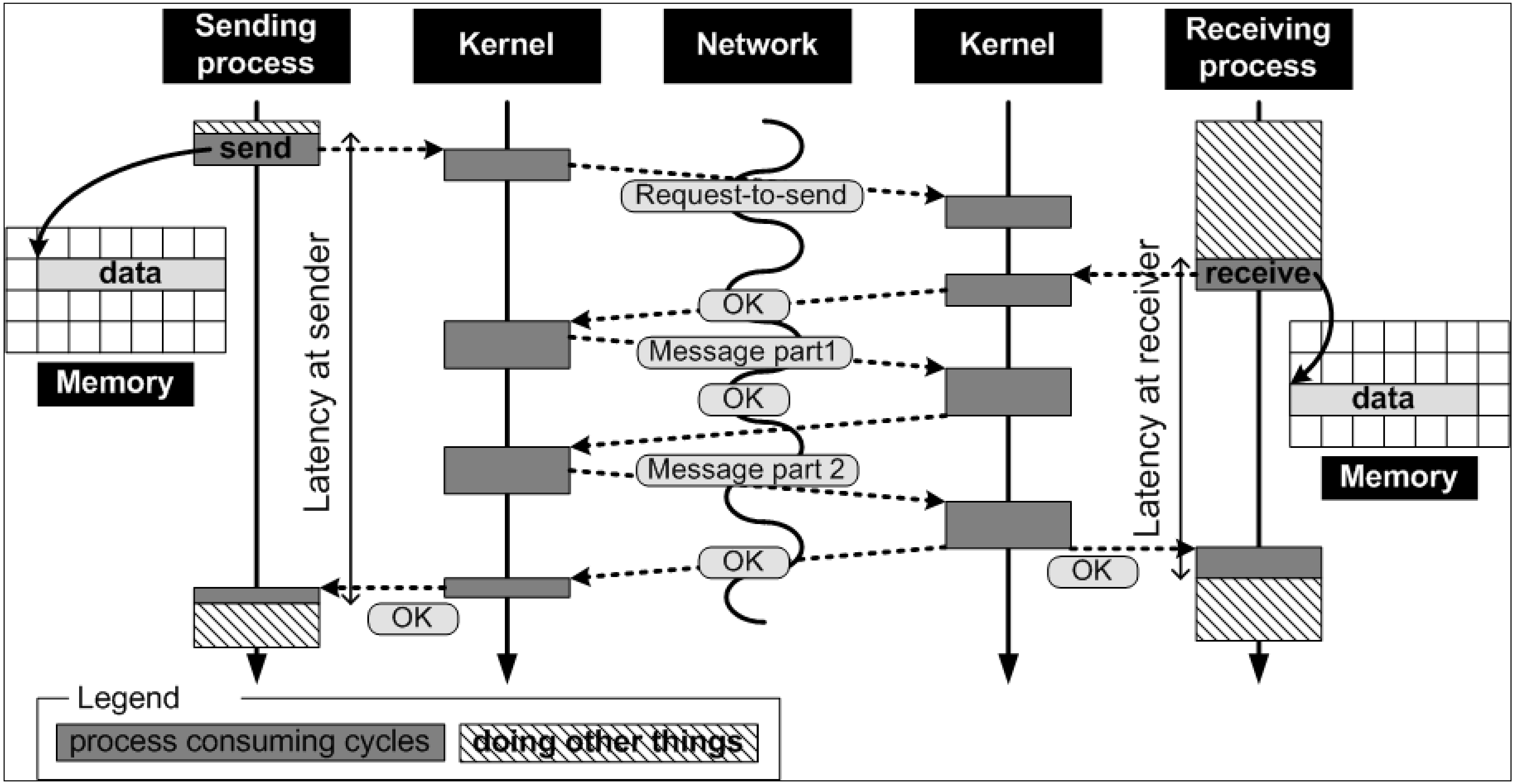
```
public static void writeObject2File(File file, Serializable o)
throws FileNotFoundException, IOException{
   FileOutputStream out = new FileOutputStream(file);
   ObjectOutputStream s = new ObjectOutputStream(out);
   s.writeObject(o);
   s.close();
}
```

- Add serialVersionUID to denote class compatibility
  - private static final long serialVersionUID = 2;
- Attributes denoted as transient are not serialized

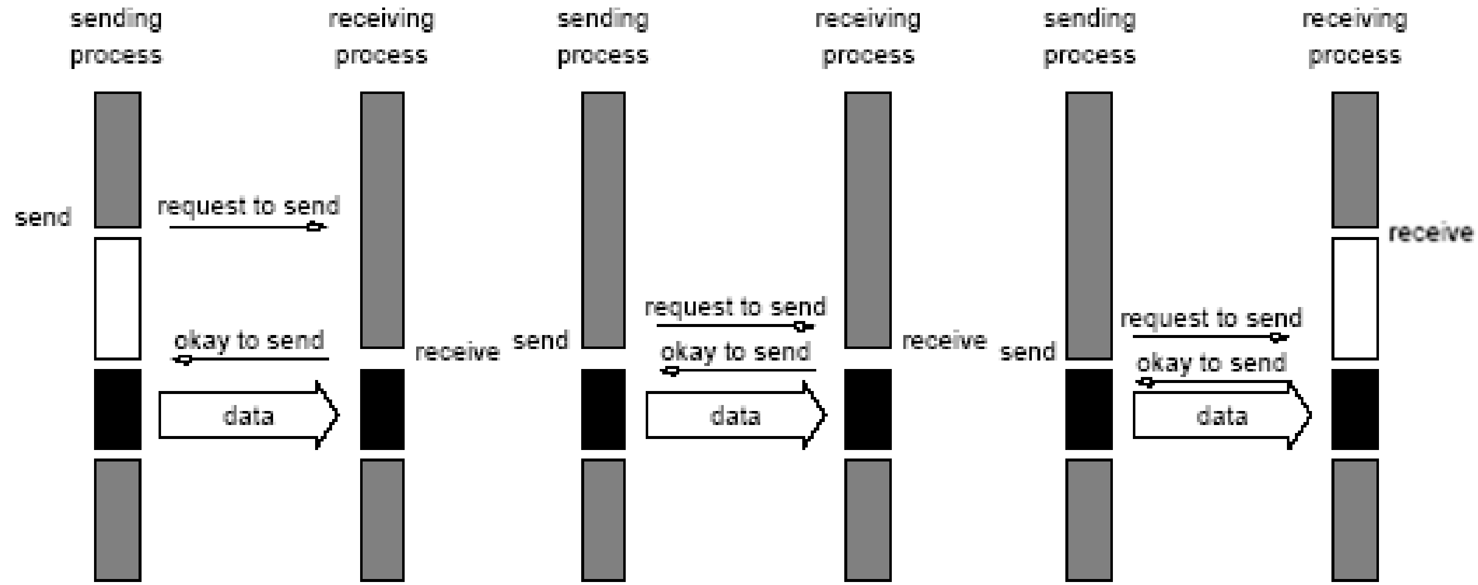
Overview

1. Definition
2. **MPI**
   - **Efficient communication**
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes

Message-passing



Non-Buffered Blocking Message Passing Operations



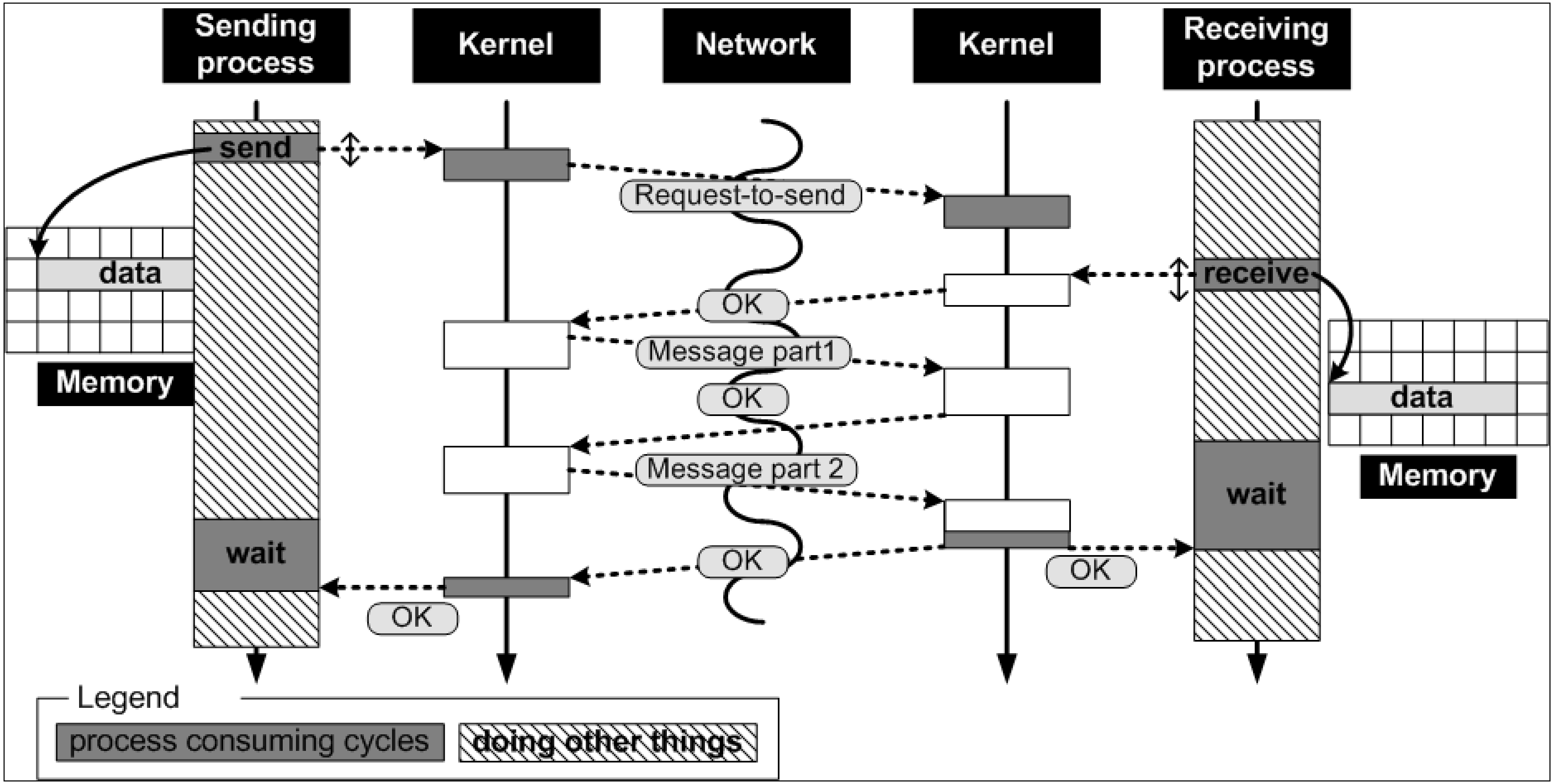
(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

- *Handshake* for a blocking non-buffered send/receive operation.
- There can be considerable idling overheads.

Non-Blocking communication



- With support for overlapping communication with computation

Non-Blocking Message Passing Operations

- With HW support: communication overhead is completely masked (*Latency Hiding 1*)
  - Network Interface Hardware allow the transfer of messages without CPU intervention
- Message can also be buffered
  - Reduces the time during which the data is unsafe
  - Initiates a DMA operation and returns immediately
    - DMA (Direct Memory Access) allows copying data from one memory location into another without CPU support *(Latency Hiding 2)*
- Generally accompanied by a check-status operation (whether operation has finished)

Be careful!

- Consider the following code segments:

P0

```
a = 100;
send(&a, 1, 1);
a=0;
```

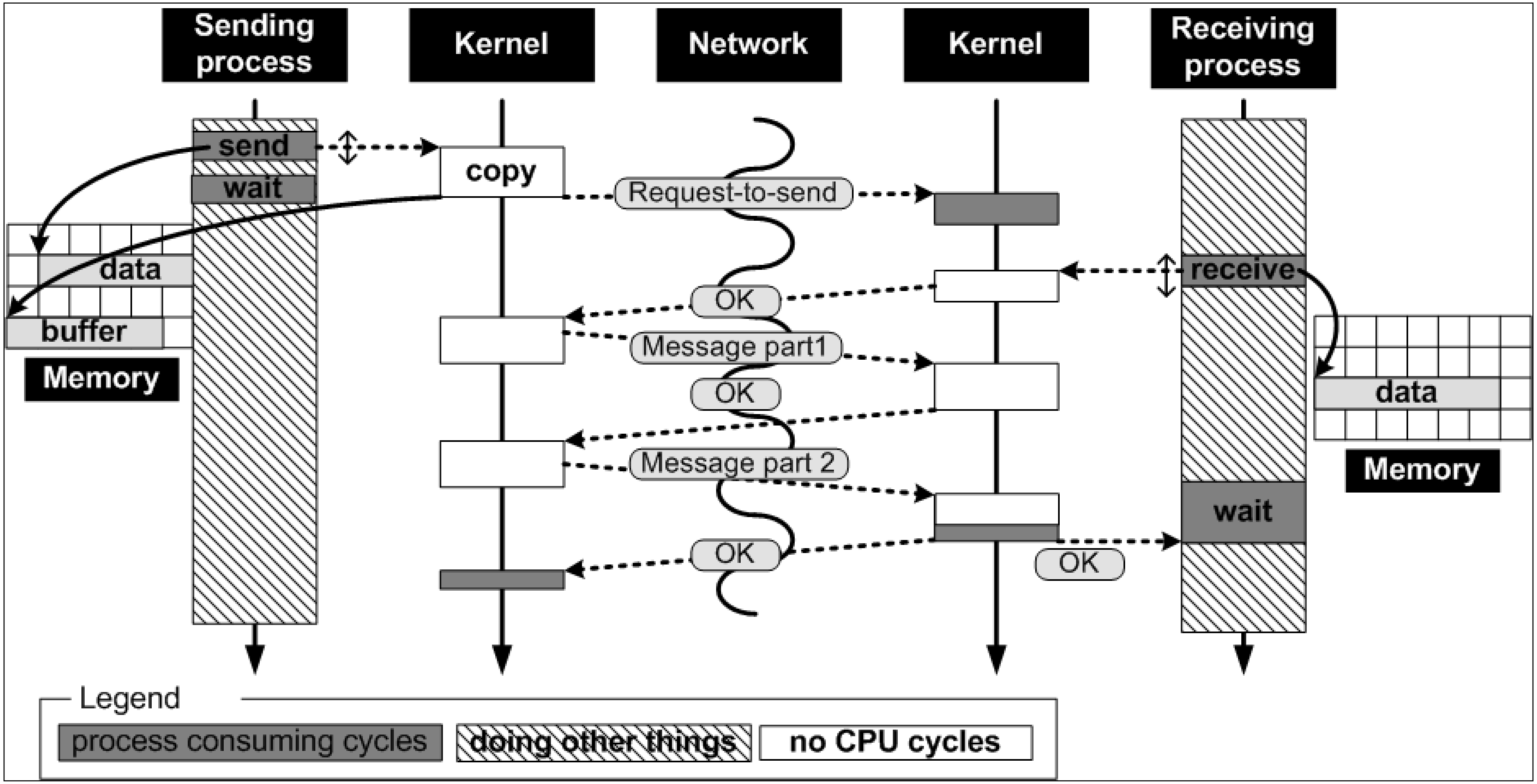
P1

```
receive(&a, 1, 0);
cout << a << endl;
```

Which protocol to use?

- Blocking protocol
  - Idling…
- Non-blocking buffered protocol
  - Buffering alleviates idling at the expense of copying overheads

Non-blocking buffered communication



KUMAR p246

Deadlock with blocking calls

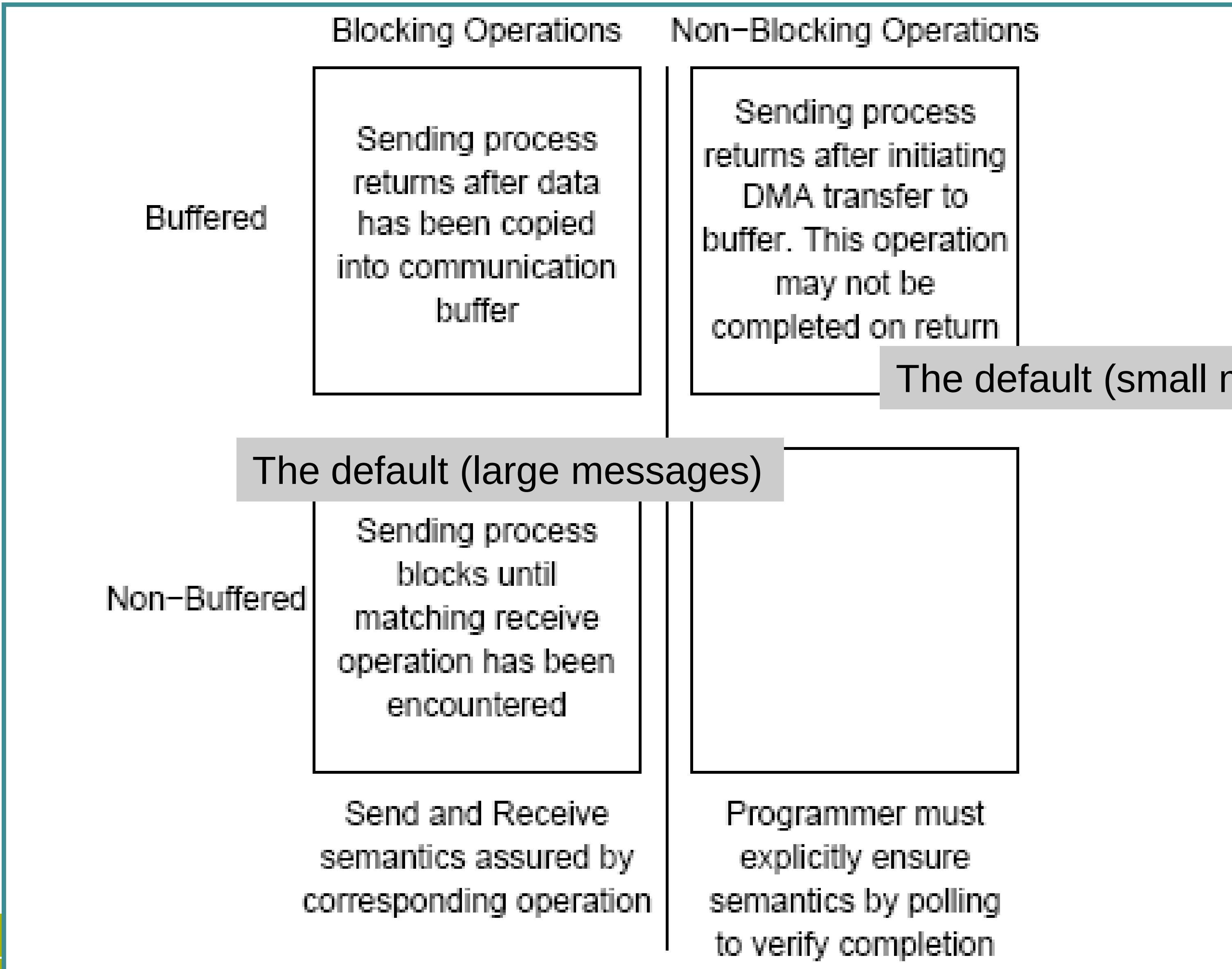
```
All processes

send(&a, 1, rank+1);
receive(&a, 1, rank-1);
```

```
All processes
If (rank % 2 == 0){
   send(&a, 1, rank+1);
   receive(&a, 1, rank-1);
} else {
   receive(&b, 1, rank-1);
   send(&a, 1, rank+1);
   a=b;
}
```

Solutions

- Switch send and receive at uneven processor
- Buffered send
- Use non-blocking calls
  - Receive should use a different buffer!
- MPI built-in function: Send_recv_replace

Send and Receive Protocols



The default (small messages

The default (large messages)

MPI Point-to-point communication

- Blocking
  - Returns if locally complete (<> globally complete)
- Non-blocking
  - Wait & test for completion functions
- Modes
  - Buffered
  - Synchronous: wait for a rendez-vous
  - Ready: no hand-shaking or buffering
    - Assumes corresponding receive is posted
- Send_recv & send_recv_replace
  - Simultaneous send & receive. Solves slide 31 problem!

Overview

1. Definition
2. MPI
   - Efficient communication
3. **Collective Communications**
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes

Collective Communication Operations

KUMAR 260

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.
- For convenience & performance
  - Collective operations can be optimized by the library by taking the underlying network into consideration!

Counting 3s with MPI *bis*

- The same program on master and slave

<u>All processes</u>

allocate subarray

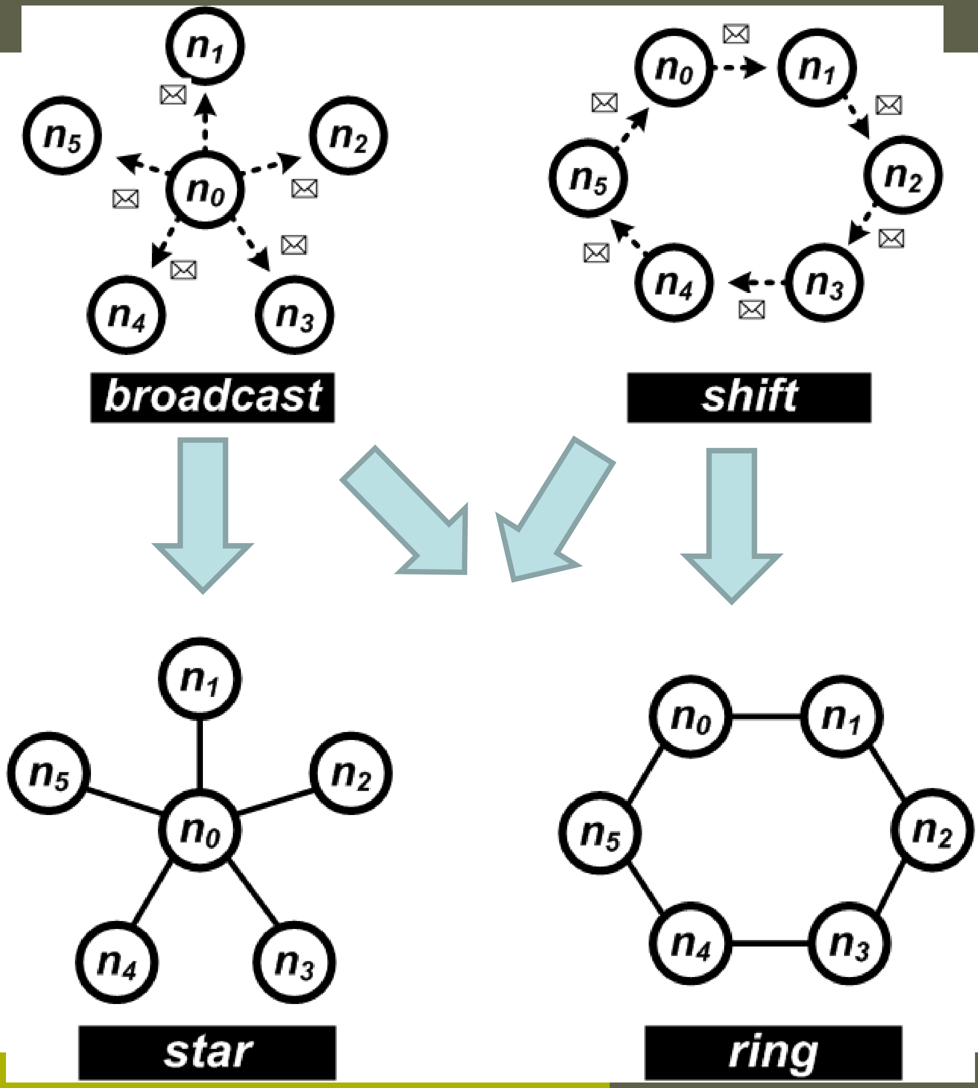
scatter array from master to subarrays

count 3s

reduce subresults to master

```
public static int countPrimesPar(int[] data, String[] args) {
    final int myRank = MPI.COMM_WORLD.Rank();
    final int NBR_PROCESSES = MPI.COMM_WORLD.Size();
    final int NBR_ELEMENTS_PER_PROCESS = data.length/NBR_PROCESSES;
    final int NBR_REST_ELEMENTS = data.length%NBR_PROCESSES; // modulo.

    int[] process_data = new int[NBR_ELEMENTS_PER_PROCESS]; // send buffer cannot be reused
in this MPI implementation...

    // scatter
    MPI.COMM_WORLD.Scatter(data, NBR_REST_ELEMENTS, process_data.length, MPI.INT , process_data, 0,
process_data.length, MPI.INT, 0);

    // count 3s
    int n = 0;

    for (int value: process_data)
       if (value == 3)
           n++;

     int[] send_buffer  = new int []{n};
     int[] recv_buffer = new int [1];

     // reduce
     MPI.COMM_WORLD.Reduce(send_buffer, 0, recv_buffer, 0, 1, MPI.INT, MPI.SUM, 0);

     return recv_buffer[0];
}
```

Optimization of Collective operations



MPI Collective Operations

- *Barrier synchronization* in MPI:

```
int MPI_Barrier(MPI_Comm comm)
```

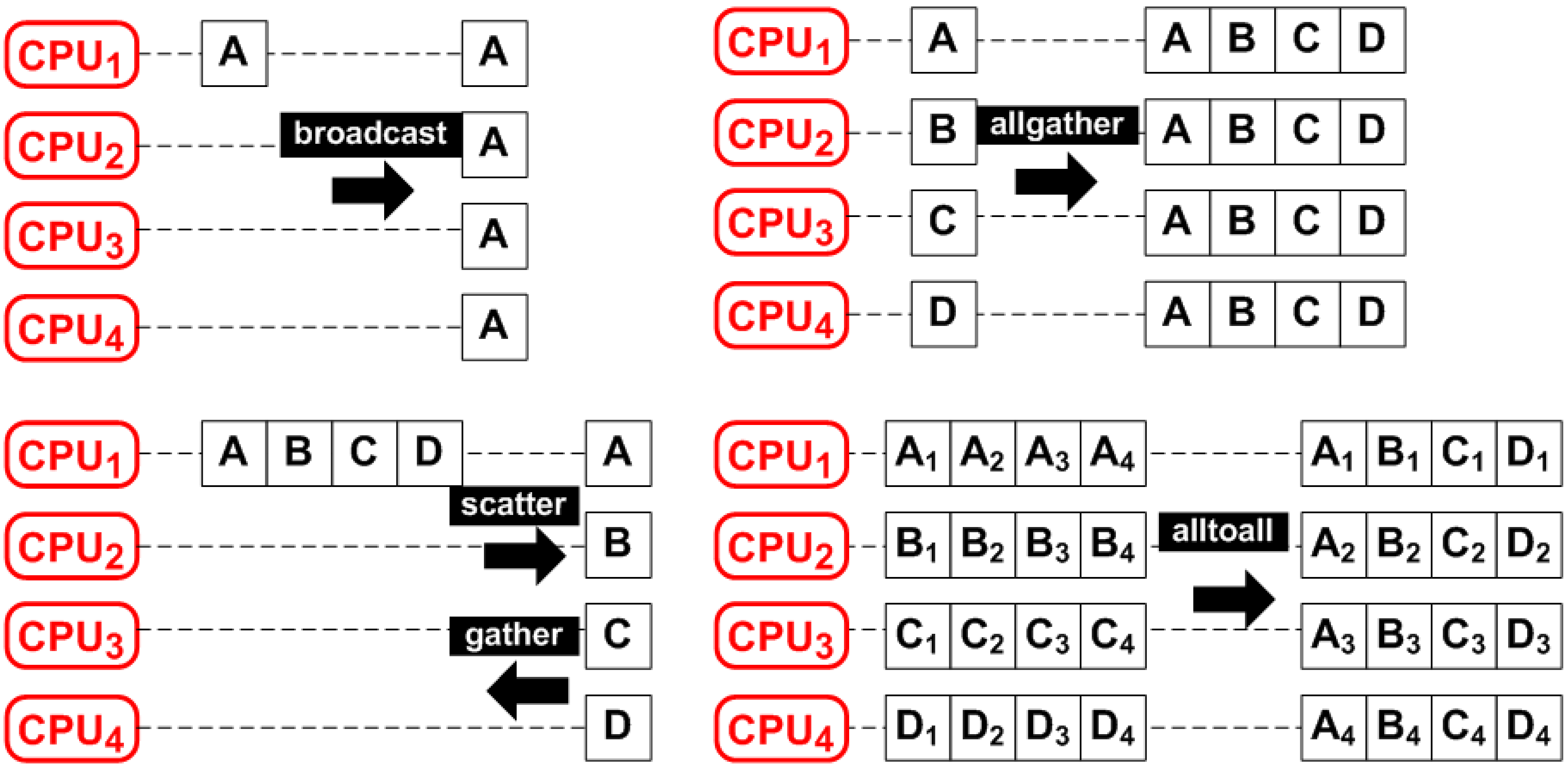
- The *one-to-all broadcast* operation is:

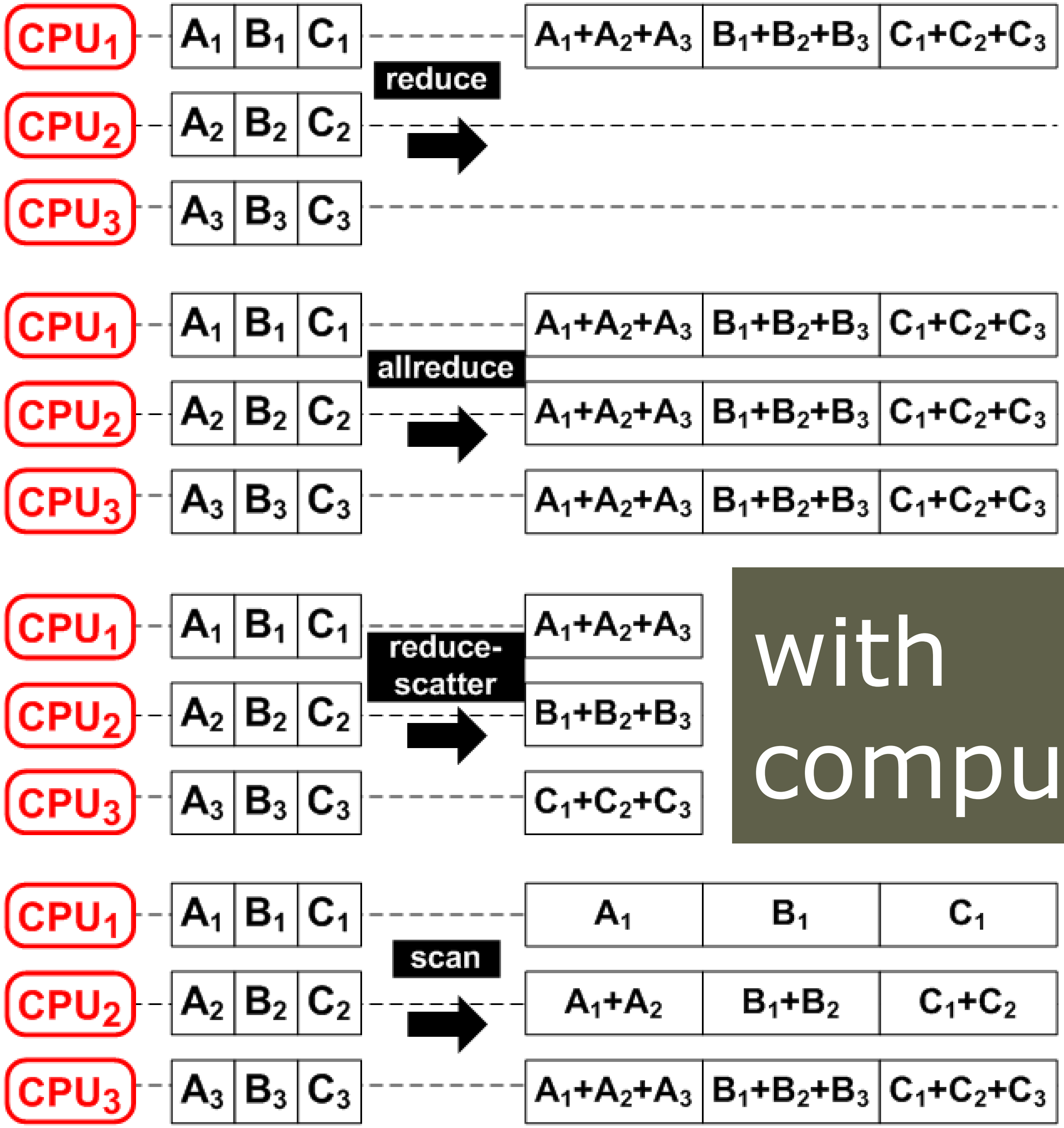
```
int MPI_Bcast(void *buf, int count, MPI_Datatype
datatype, int source, MPI_Comm comm)
```

- The *all-to-one reduction* operation is:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int
target, MPI_Comm comm)
```

MPI Collective Operations





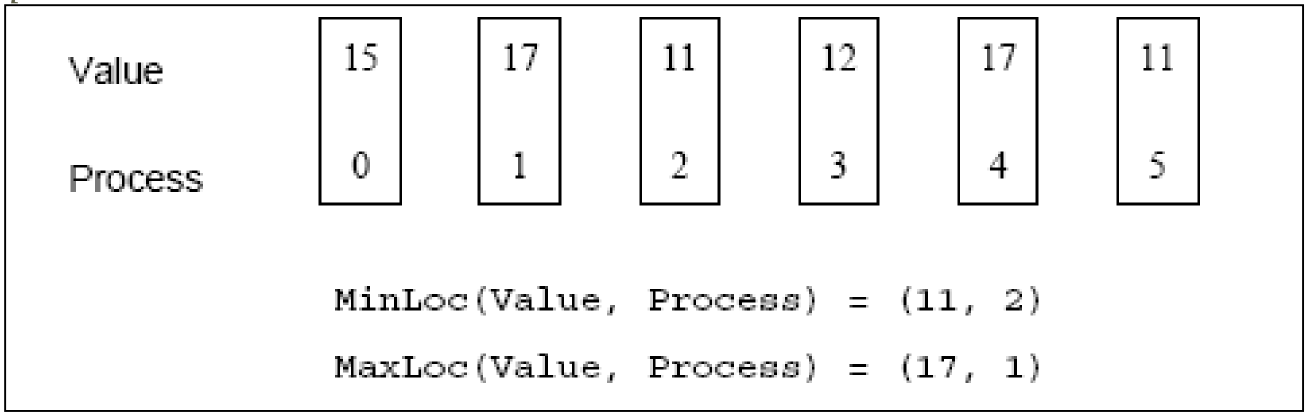
with computations

Predefined Reduction Operations

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte
`MPI_MAXLOC`	max-min value-location	Data-pairs
`MPI_MINLOC`	min-min value-location	Data-pairs

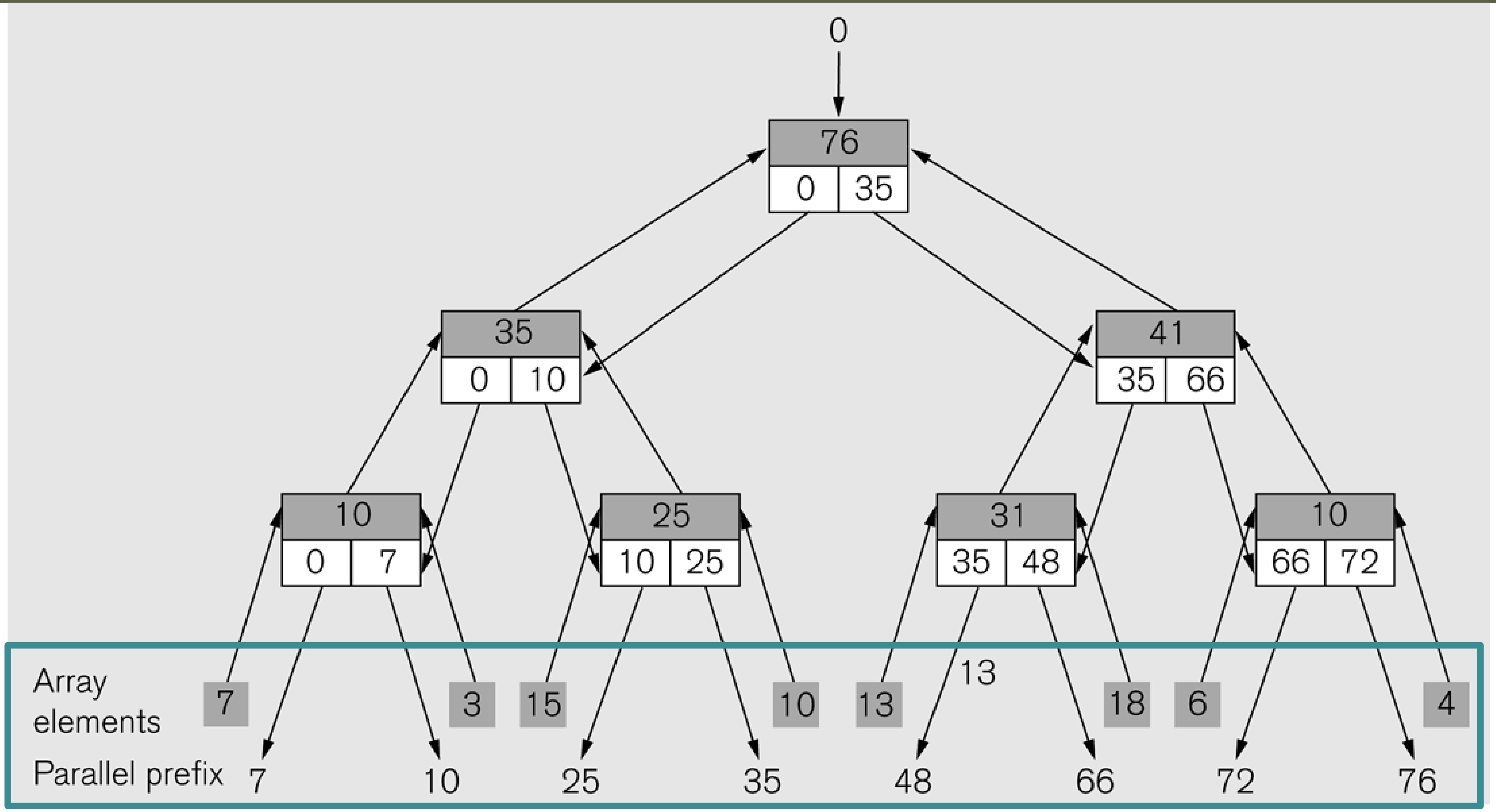
Maximum + location

- `MPI_MAXLOC` returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.



An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

Scan operation



- *Parallel prefix sum*: every node got sum of previous nodes + itself

Overview

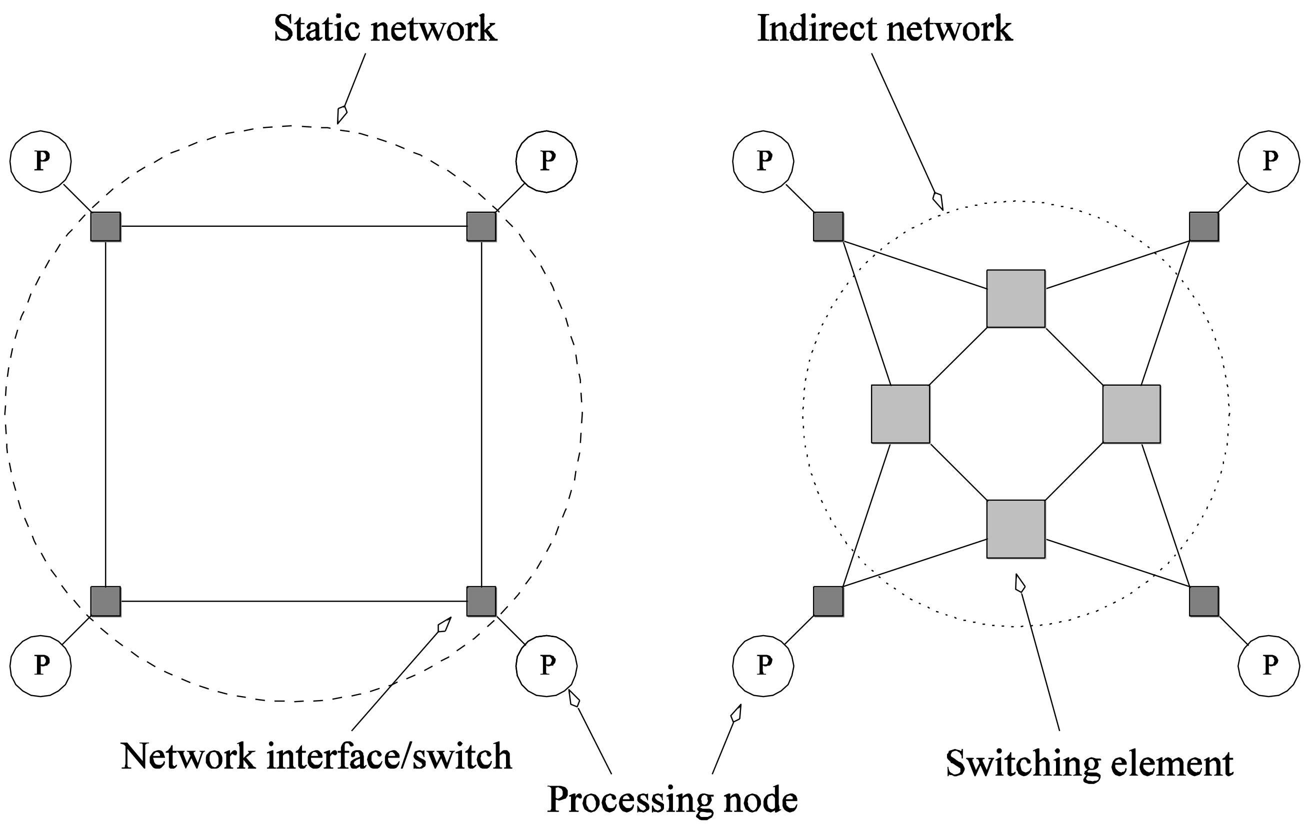
1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. **Interconnection networks**
   - Static networks
   - Dynamic networks
5. End notes

KUMAR 33-45

Interconnection Networks

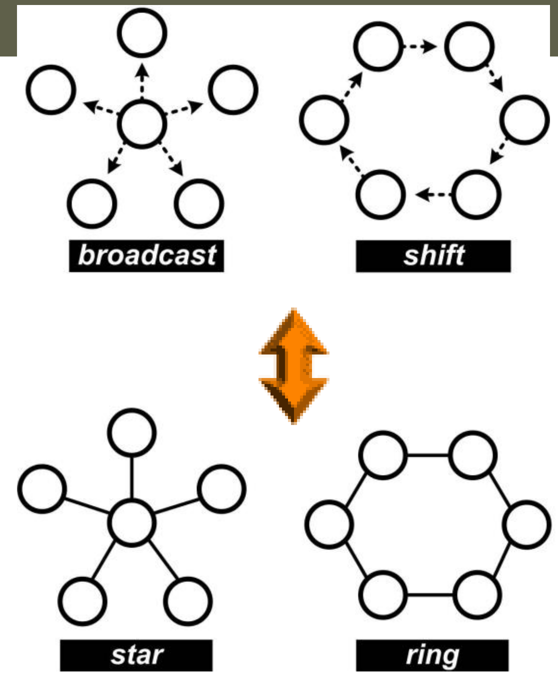
- Interconnection networks carry data between processors and memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as *static* or *dynamic*.
  - Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.
  - Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.

Static and Dynamic Interconnection Networks



Important characteristics

- Performance
  - Depends on application:

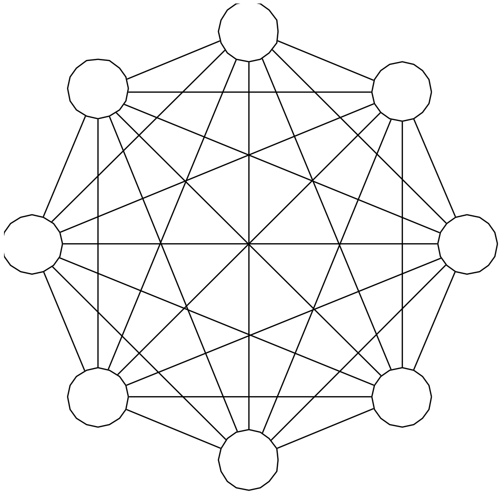


- Cost
- Difficulty to implement
- Scalability
  - Can processors be added with the same cost

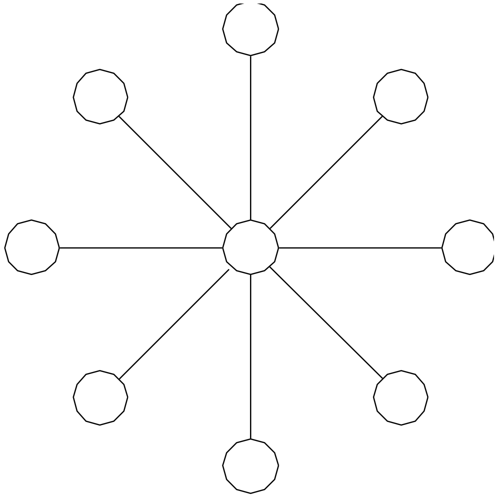
Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. **Interconnection networks**
   - **Static networks**
   - Dynamic networks
5. End notes

Network Topologies: Completely Connected and Star Connected Networks



(a)



(b)

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

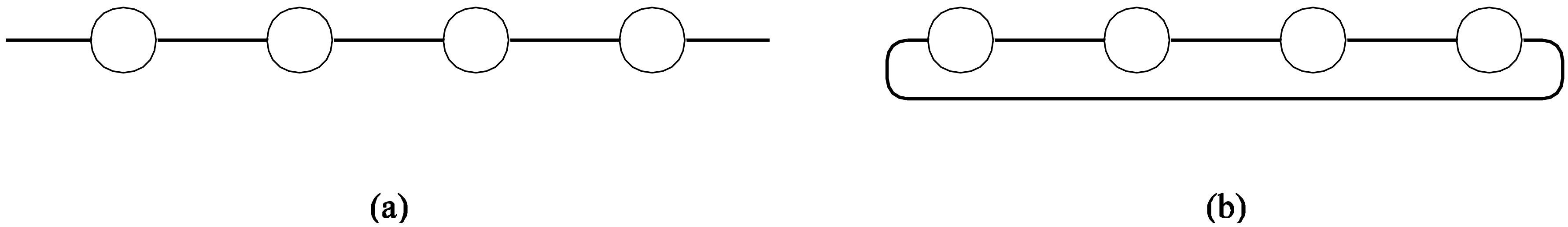
Completely Connected Network

- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of p.
- In this sense, these networks are static counterparts of crossbars (see later).

Star Connected Network

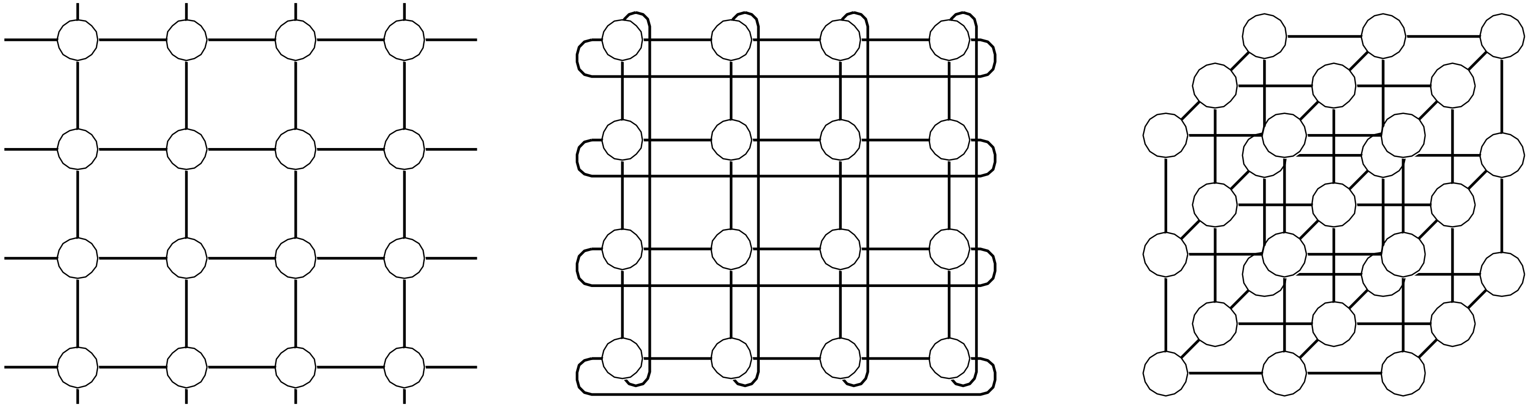
- Every node is connected only to a common node at the center.
- Distance between any pair of nodes is *O(1).* However, the central node becomes a bottleneck.
- In this sense, star connected networks are static counterparts of buses.

Linear Arrays



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

Network Topologies: Two- and Three Dimensional Meshes



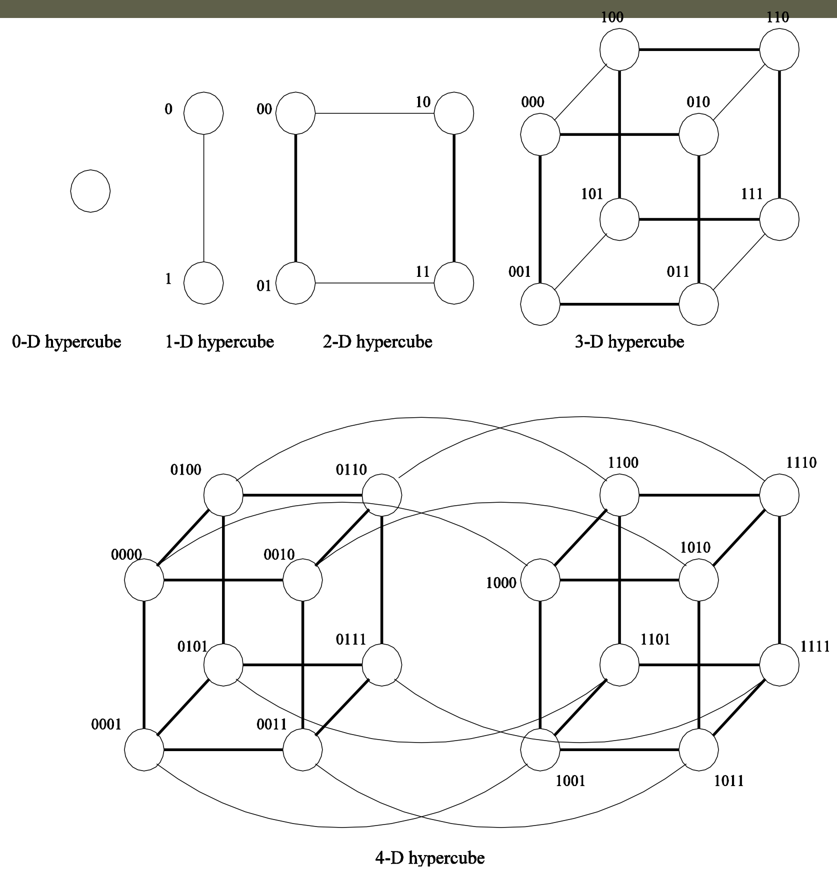
(a) (b) (c)

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.

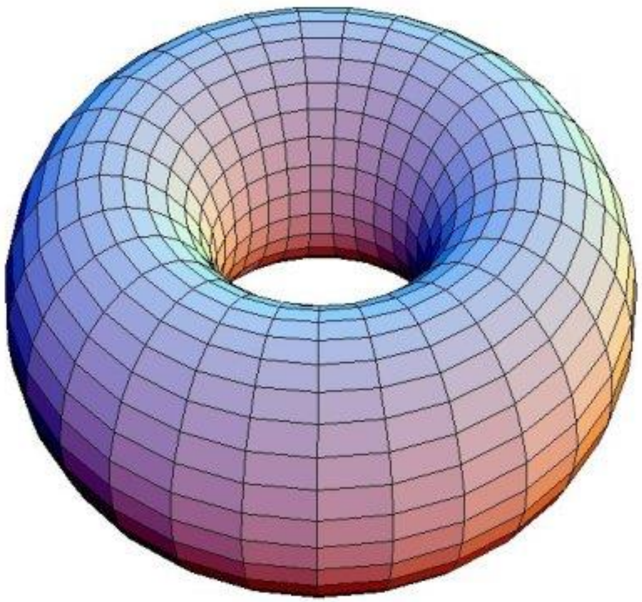
Network Topologies: Linear Arrays, Meshes, and *k-d* Meshes

- In a ***linear array***, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a ***1D torus or a ring***.
- ***Mesh***: generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.
- A further generalization to *d* dimensions has nodes with *2d* neighbors.
- A special case of a *d*-dimensional mesh is a ***hypercube***. Here, $d = log\ p$, where *p* is the total number of nodes.

Hypercubes and torus



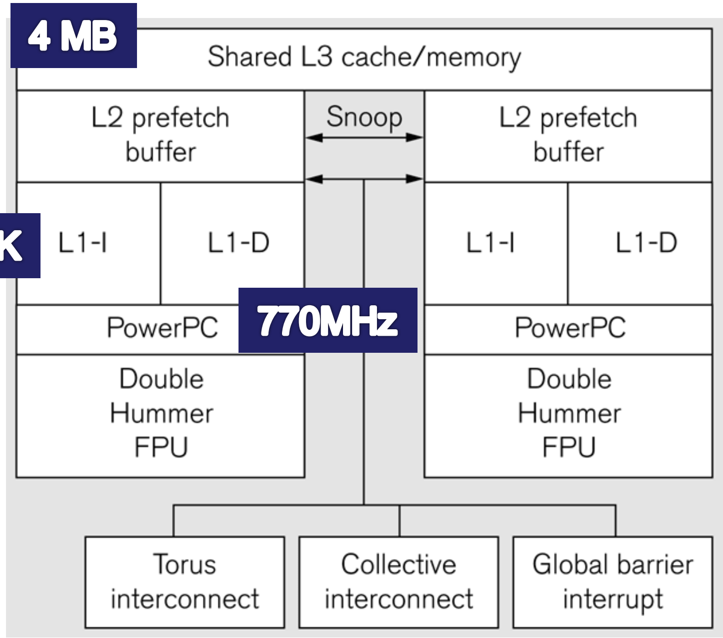
Construction of hypercubes from hypercubes of lower dimension.



Torus (2D wraparound mesh).

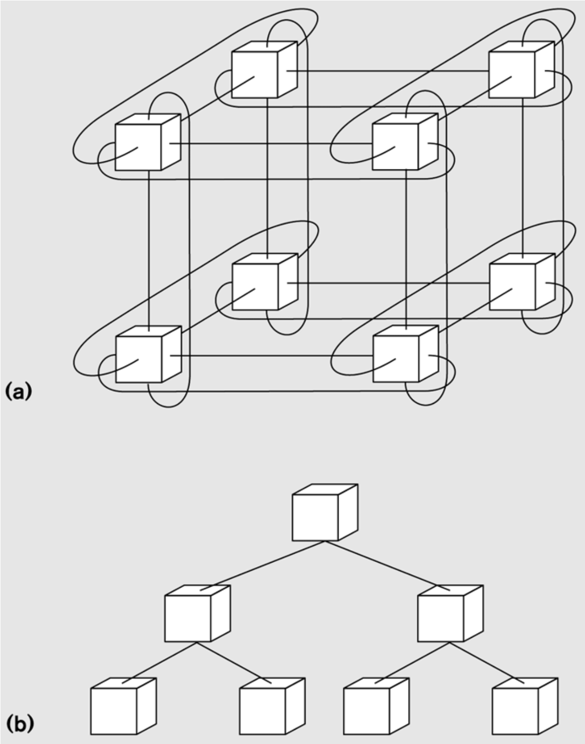
Super computer: BlueGene/L

a BlueGene/L node.



- IBM, No 1 in 2007
  - www.top500.org
- 65.536 dual core nodes
  - E.g. one processor dedicated to communication, other to computation
  - Each 512 MB RAM
- US$100 miljoen
- Now replaced by BlueGene/P and BlueGene/Q

BlueGene/L communication networks

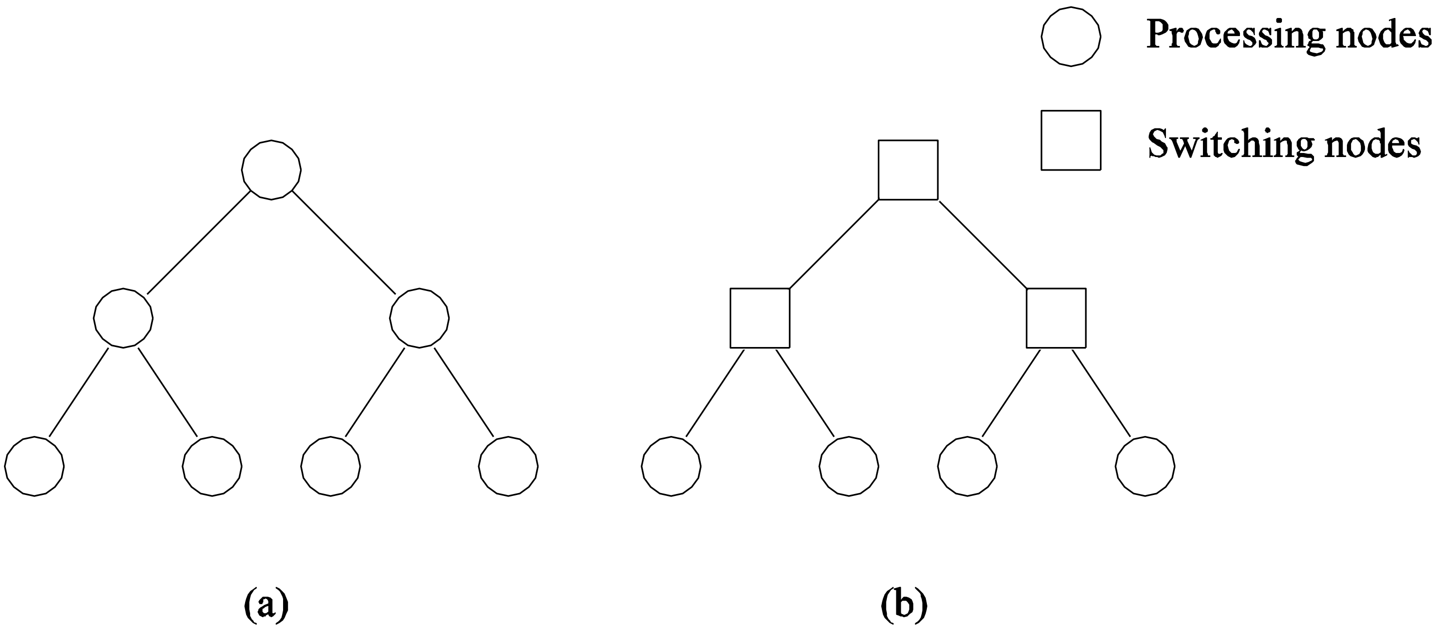


(a) 3D torus (64x32x32) for standard interprocessor data transfer
- Cut-through routing (see later)

(b) collective network for fast evaluation of *reductions*.

(c) Barrier network by a common wire

Network Topologies: Tree-Based Networks

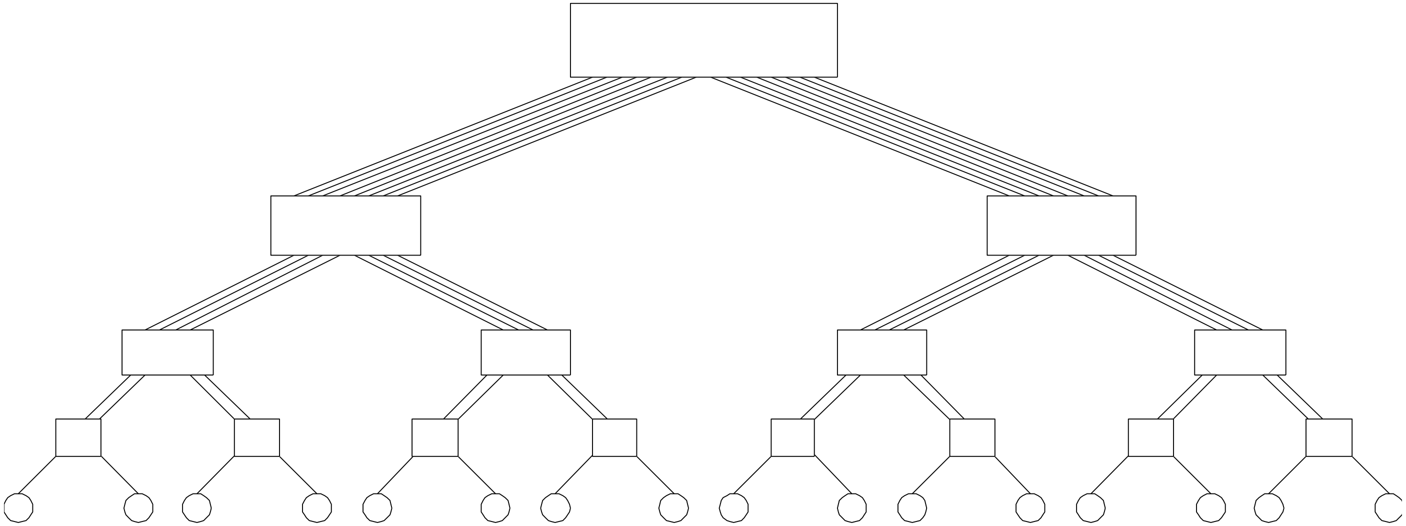


Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

Tree Properties

- $p = 2^d - 1$ with d depth of tree
- The *distance* between any two nodes is no more than ***2 log p***.
- Links higher up the tree potentially carry more traffic than those at the lower levels.
- For this reason, a variant called a *fat-tree*, fattens the links as we go up the tree.
- Trees can be laid out in 2D with no wire crossings. This is an attractive property of trees.

Network Topologies: Fat Trees



A fat tree network of 16 processing nodes.

Network Properties

- *Diameter:* The distance between the farthest two nodes in the network.
- *Bisection Width:* The minimum number of links you must cut to divide the network into two equal parts.
- *Arc connectivity*: minimal number of links you must cut to isolate two nodes from each other. A measure of the multiplicity of paths between any two nodes.
- *Cost:* The number of links. Is a meaningful measure of the cost.
  - However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor into the cost.

Static Network Properties

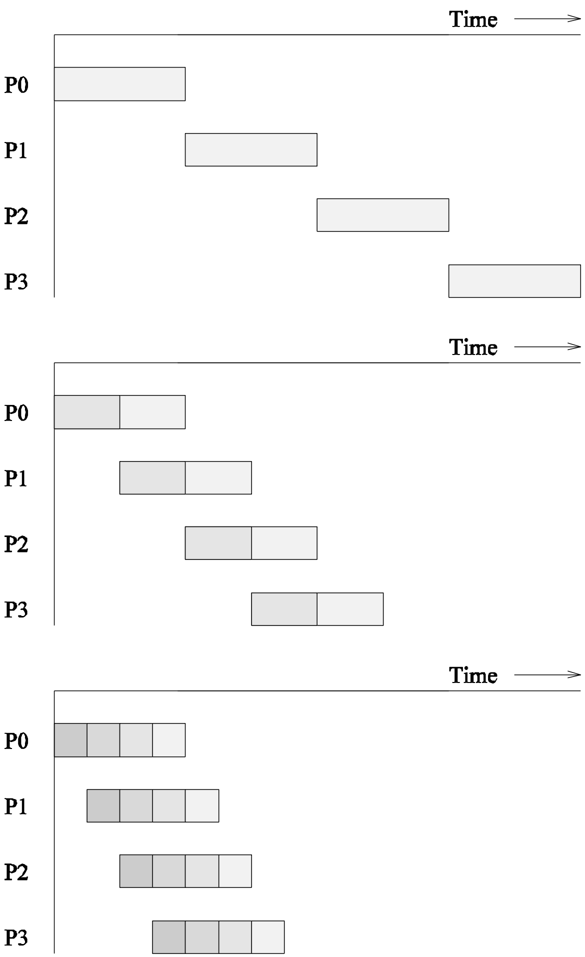
Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	/	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh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

KUMAR 53-60

Message Passing Costs

- The total time to transfer a message over a network comprises of the following:
  - *Startup time* (t_s): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
  - *Per-hop time* (t_h): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
  - *Per-word transfer time* (t_w): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

Routing Techniques



(a) A single message sent over a store-and-forward network

(b) The same message broken into two parts and sent over the network.

(c) The same message broken into four parts and sent over the network.

Passing a message from node P_0 to P_3:

(a) a ***store-and-forward*** communication network;

(b) and (c) extending the concept to *cut-through routing*. The shaded regions: message is in transit. The startup time of message transfer is assumed to be zero.

Store-and-Forward Routing

- A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.
- The total communication cost for a message of size *m* words to traverse *l* communication links is

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, t_h is small and the above expression can be approximated by

$$t_{comm} = t_s + mlt_w.$$

Cut-Through Routing

- The total communication time for cut-through routing is approximated by:

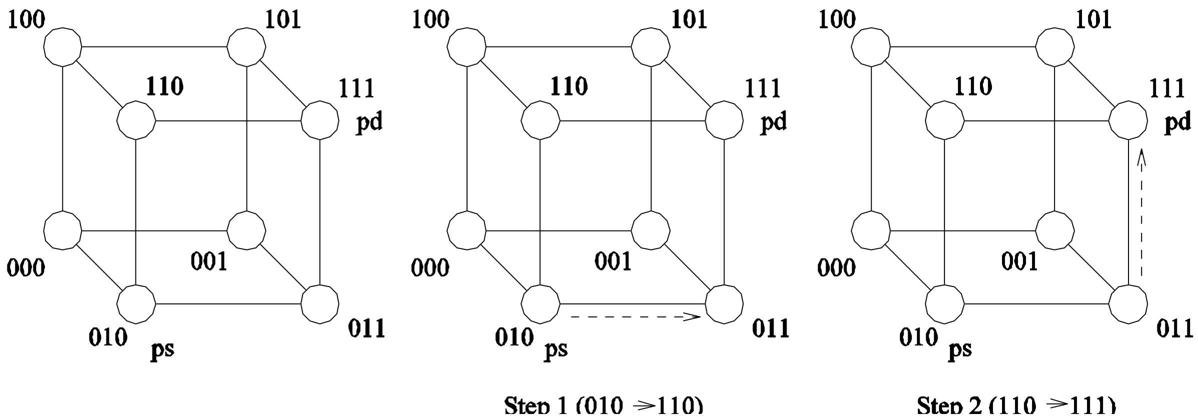
$$t_{comm} = t_s + t_h l + t_w m.$$

- Identical to packet routing, however, t_w is typically much smaller.
- t_h is typically smaller than t_s and t_w. Thus, particularly, when m is large:

$$t_{comm} = t_s + t_w m.$$

Routing Mechanisms for Interconnection Networks

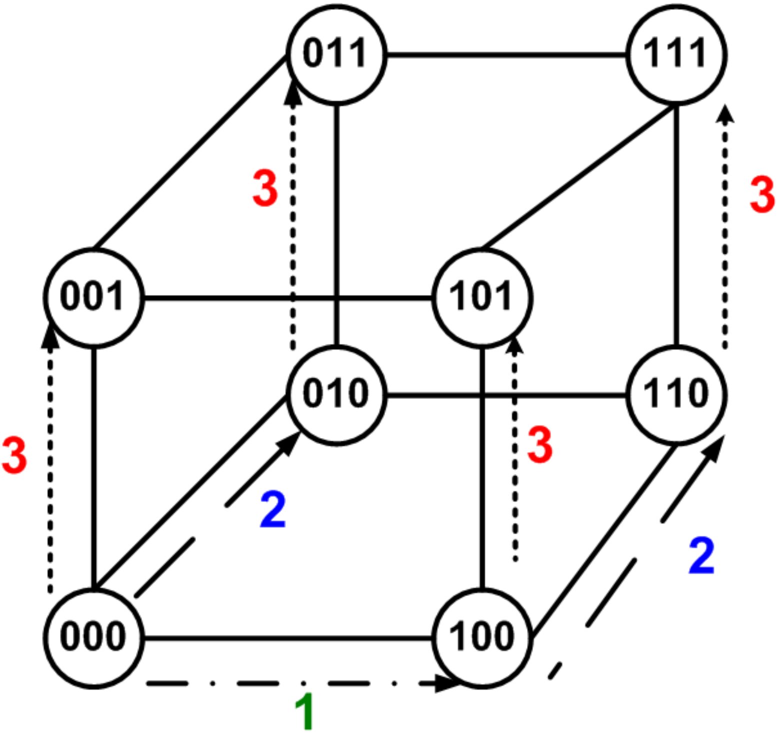
KUMAR 64



Routing a message from node P_s (010) to node P_d (111) in a three-dimensional hypercube using E-cube routing.

A broadcast in a Hypercube

Message from node 0 to all others: *d* steps



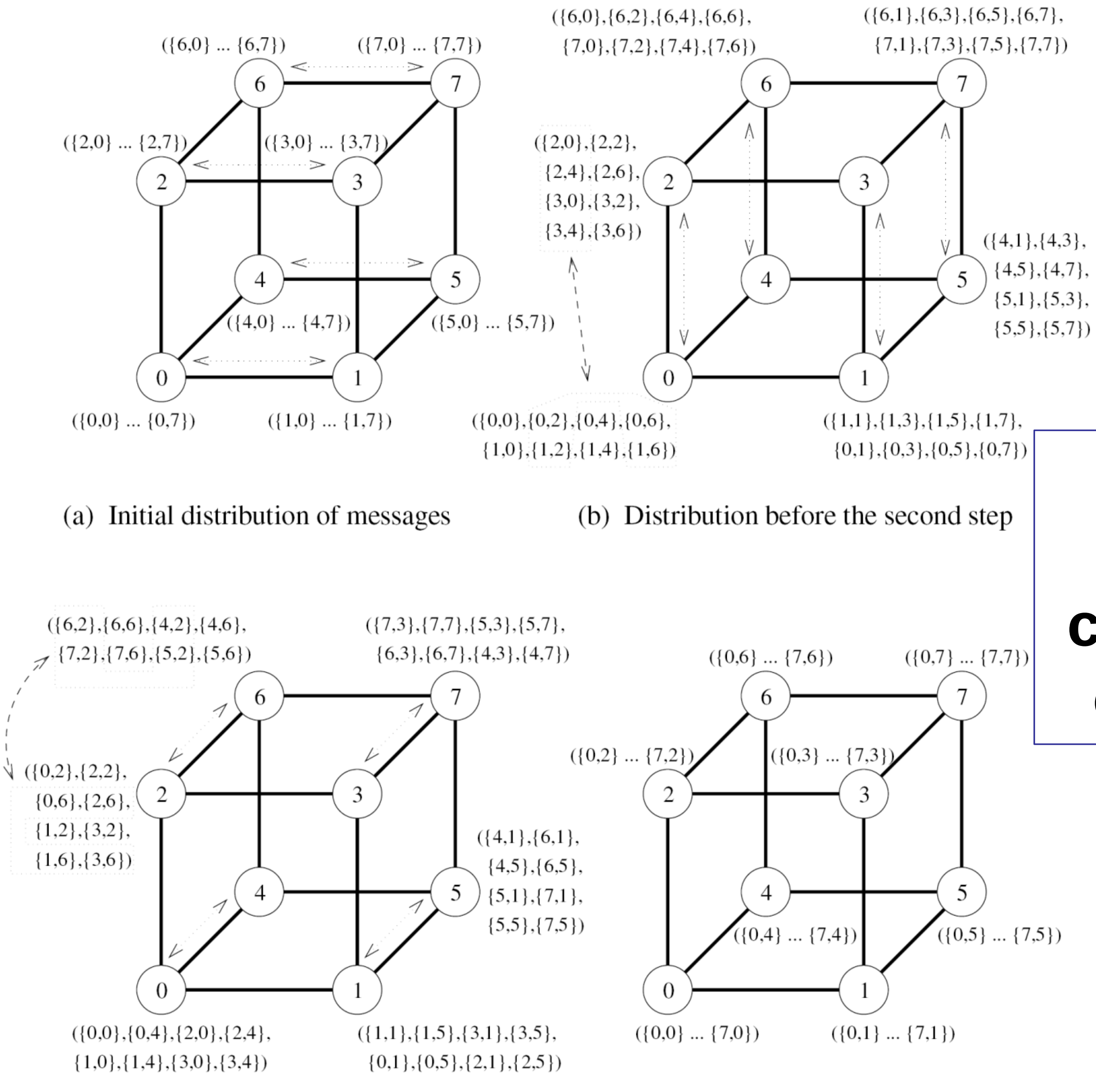

```
for(int d: dimensions)
 if (all bits with index > d are 0)
   if (dth bit == 0)
     send message to (flip dth bit)
   else
     receive message from (flip dth
bit)
```

Reduce operation is the opposite…

Cost of Communication Operations

- Broadcast on hypercube: *log p* steps
  - With cut-through routing: $T_{comm}=(t_s+t_w m).\log p$
- All-to-all broadcast (full duplex links)
  - Hypercube: *log p* steps
  - Linear array: *p-1* steps
  - ring: p/2 steps
  - 2D-Mesh: $2\sqrt{p}$ steps
- Scatter and gather: similar to broadcast
- Circular q-shift: send msg to *(i+q)mod p*
  - Mesh: maximal $\sqrt{p}/2$ steps
  - In a hypercube: embedding a linear array

All-to-all personalized communication on hypercube



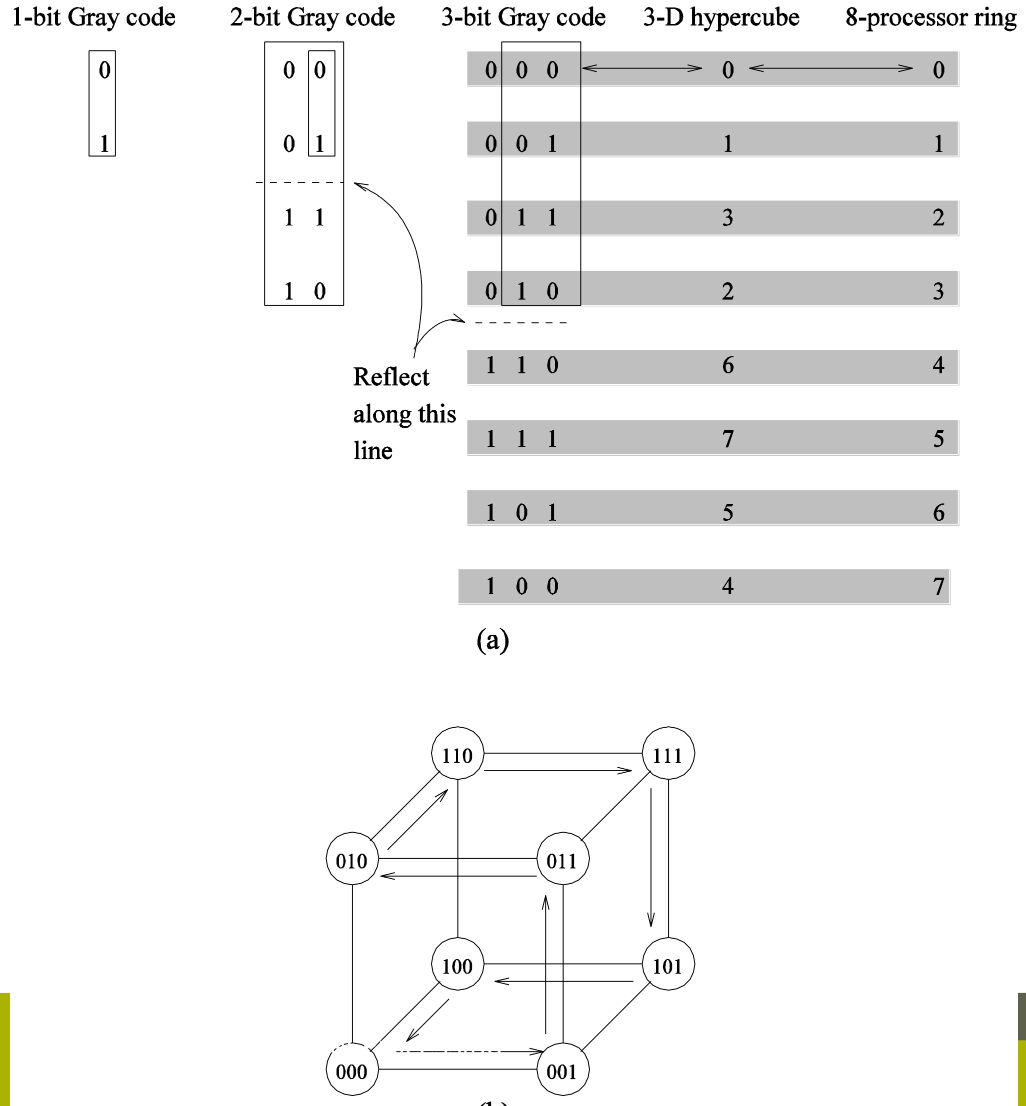
(a) Initial distribution of messages

(b) Distribution before the second step

(c) Distribution before the third step

(d) Final distribution of messages

Embedding a Linear Array into a Hypercube



Gray code problem: *arrange nodes in a ring so that neighbors only differ by 1 bit*

(a) A three-bit reflected Gray code ring
(b) its embedding into a three-dimensional hypercube.

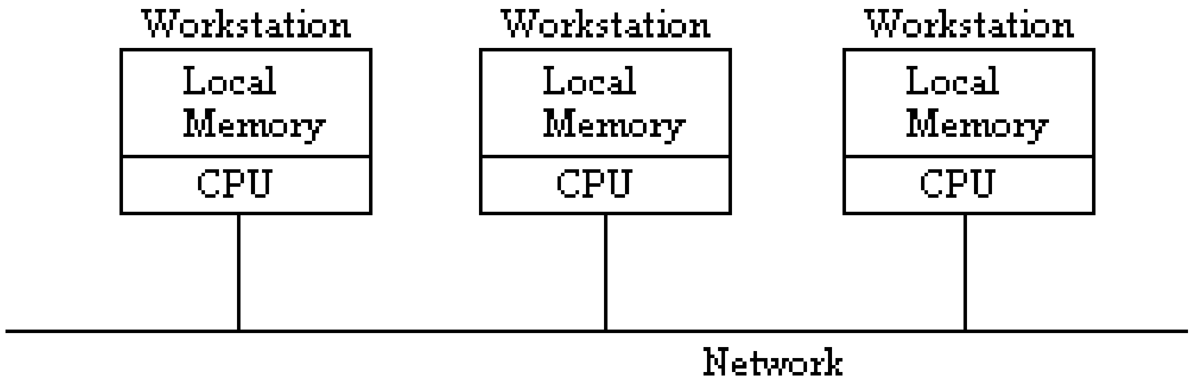
Application of Gray code

- To facilitate error correction in digital communications
- The problem with natural binary codes is that, with real switches, it is very unlikely that switches will change states exactly in synchrony
- transition from 011 (*3*) to 100 (*4*) might look like 011 - 001 — 101 — 100
  - For receiver it is unclear whether 101 is send or not…
  - Solution: use Gray code

Overview

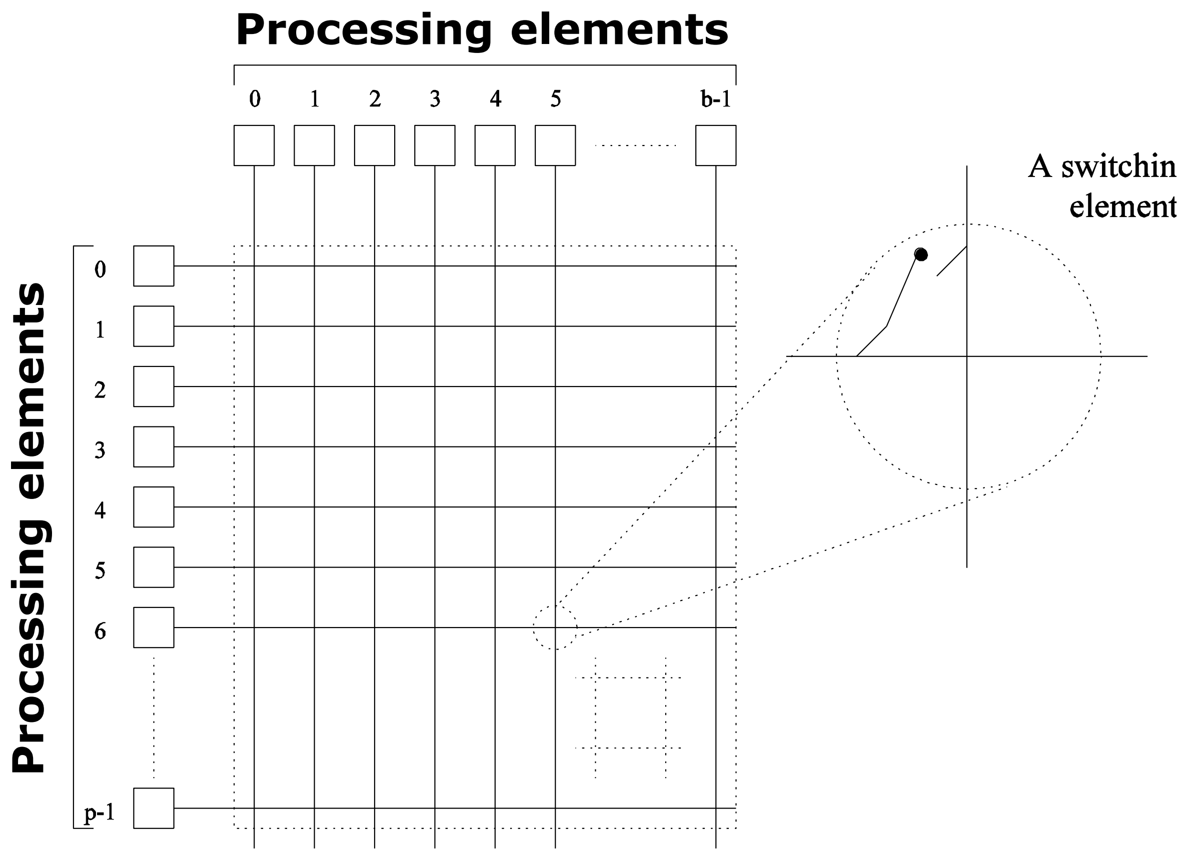
1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. **Interconnection networks**
   - Static networks
   - **Dynamic networks**
5. End notes

Dynamic networks: Buses



Bus-based interconnect

Dynamic Networks: Crossbars

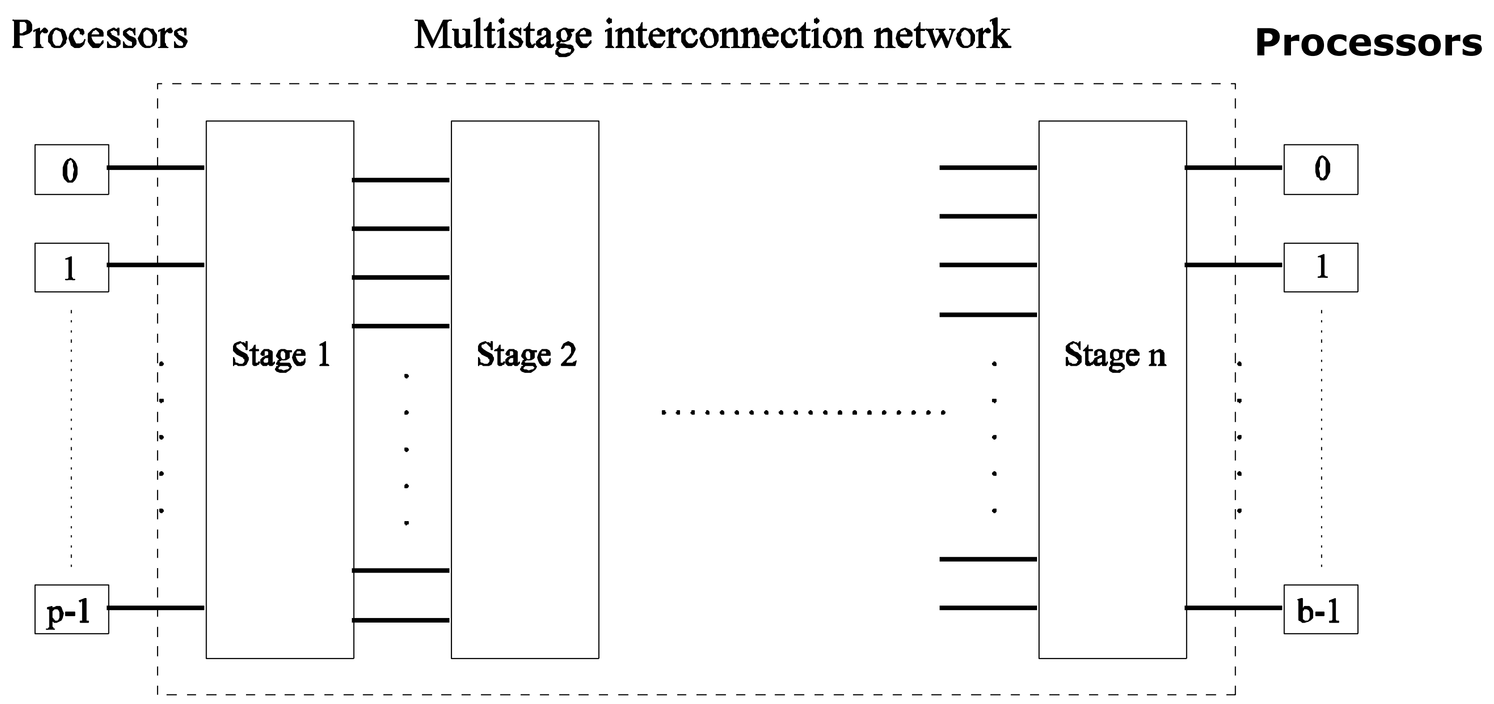


A crossbar network uses an *p×m* grid of switches to connect *p* inputs to m outputs in a non-blocking manner.

Multistage Dynamic Networks

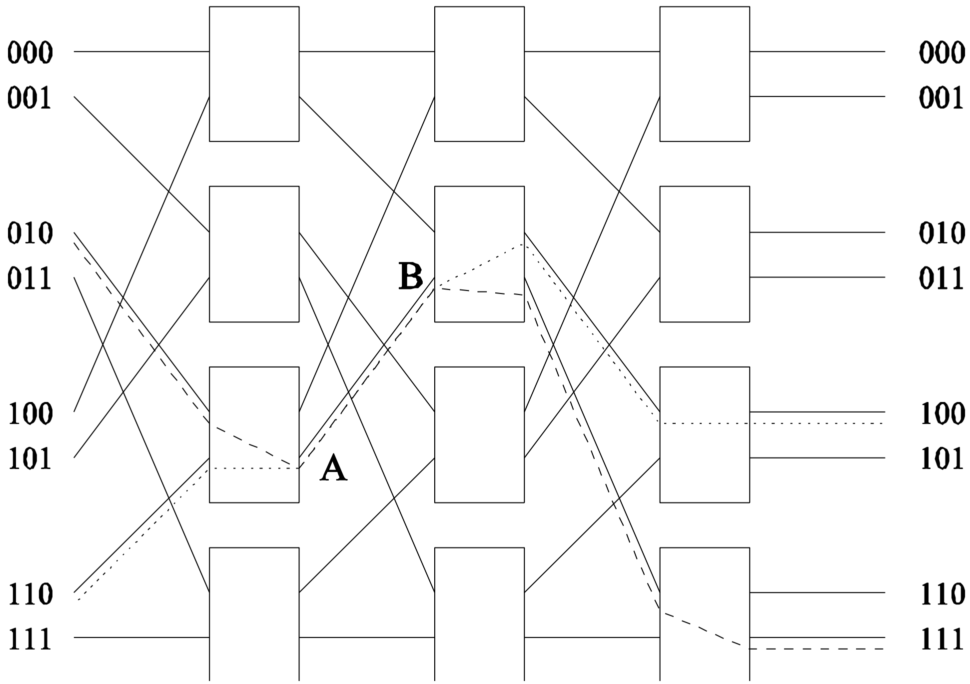
- Crossbars have excellent performance scalability but poor cost scalability.
  - The cost of a crossbar of p processors grows as $O(p^2)$.
  - This is generally difficult to scale for large values of p.
- Buses have excellent cost scalability, but poor performance scalability.
- ⇨ Multistage interconnects strike a compromise between these extremes.

Multistage Dynamic Networks



The schematic of a typical multistage interconnection network.

Multistage Dynamic Networks



An **Omega network** is based on 2×2 switches.

An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

Evaluating Dynamic Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	1	p log p
Dynamic Tree	$2 \log p$	1	2	$p - 1$

Recent trend: networks-on-chip

- Many-cores (such as cell processor)

- Increasing number of cores
  - ➡ bus or crossbar switch become infeasible
  - ➡ specific network has to be chosen

- When even more cores
  - ➡ scalable network required

Memory Latency λ

- Memory Latency = *delay required to make a memory reference,* relative to processor's local memory latency, ≈ unit time ≈ one word per instruction

Architecture Family	Computer	Lambda
Chip Multiprocessor*	AMD Opteron	100
Shared-memory Multiprocessor	Sun Fire E25K	400–660
Co-processor	Cell	N/A
Cluster	HP BL6000 w/GbE	4,160–5,120
Supercomputer	BlueGene/L	8960

*CMP's λ value measures a transfer between L1 data caches on chip.

Overview

1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Dynamic networks
   - Static networks
5. End notes

Choose MPI

- Makes the fewest assumptions about the underlying hardware, is the least common denominator. It can execute on any platform.
- Currently the best choice for writing large, long-lived applications.

MPI Issues

- MPI messages incur large overheads for each message
  - Minimize cross-process dependences
  - Combine multiple message into one
- Safety
  - Deadlock & livelock still possible…
    - But easier to deal with since synchronization is explicit
  - Sends and receives should be properly matched
  - Non-blocking and non-buffered messages are more efficient but make additional assumptions that should be enforced by the programmer.

MPI-3: non-blocking collective communication operations

- Start a collective operation
- Proceed with some other stuff
- Check whether collective has been finished

➡ Hide communication behind useful computations

MPI-2: also supports one-sided communication

- process accesses remote memory without interference of the remote ‘owner’ process
- Process specifies all communication parameters, for the sending side and the receiving side
  - exploits an interconnect with RDMA (Remote DMA) facilities
- Additional synchronization calls are needed to assure that communication has completed before the transferred data are locally accessed.
  - User imposes right ordering of memory accesses

One-sided primitives

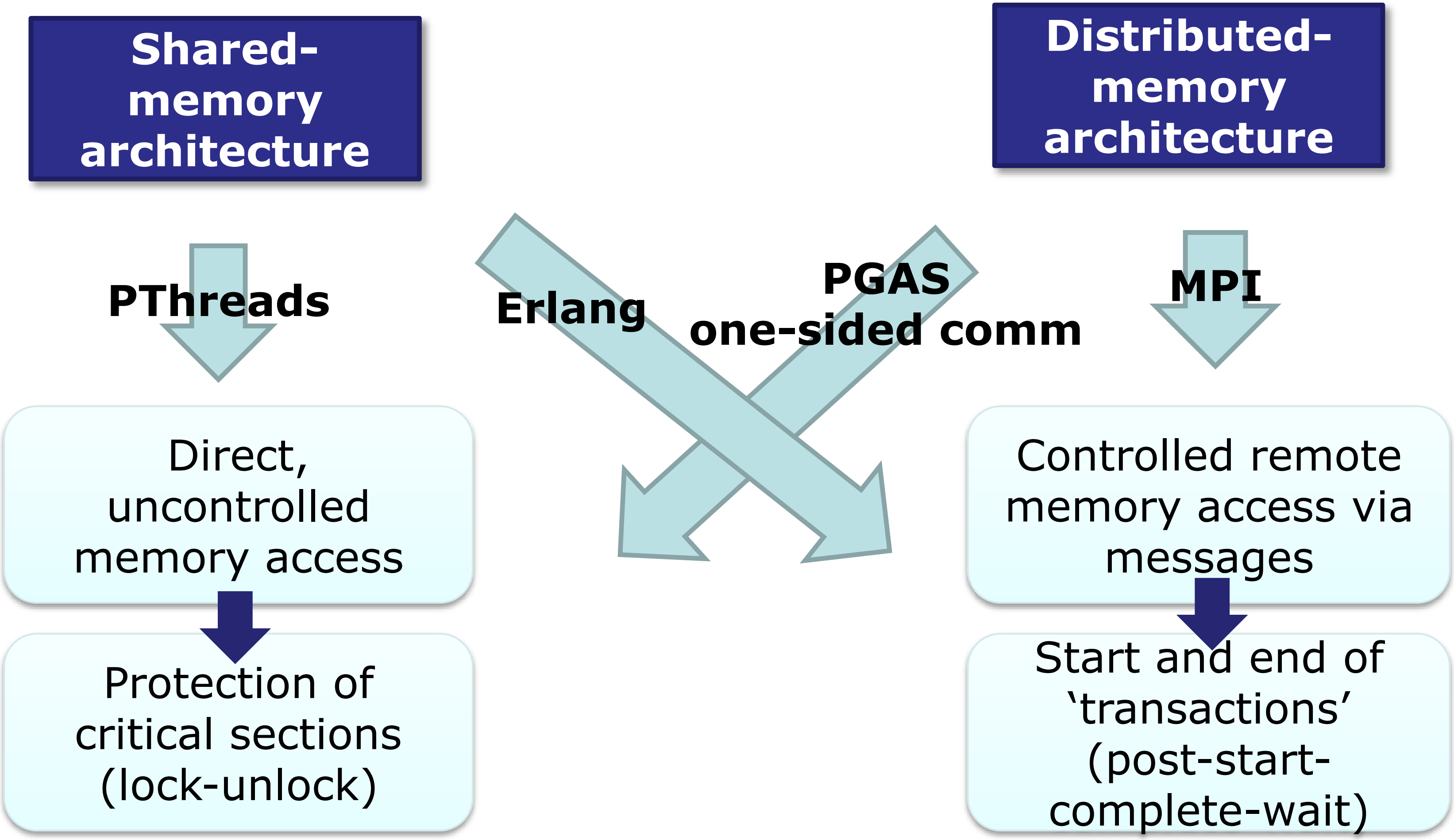
- Communication calls
  - **MPI_Get**: Remote read.
  - **MPI_Put**: Remote write.
  - **MPI_Accumulate**: accumulate content based on predefined operation
- Initialization: first, process must create window to give access to remote processes
  - **MPI_Win_create**
- Synchronization to prevent concflicting accesses
  - **MPI_Win_fence**: like a barrier
  - **MPI_Win_post**, **MPI_Win_start**, **MPI_Win_complete**, **MPI_Win_wait** : like message-passing
  - **MPI_Win_lock**, **MPI_Win_unlock**: like multi-threading

Partitioned Global Address Space Languages (PGAS)

PPP 243

- Higher-level abstraction: overlay a single address space on the virtual memories of the distributed machines.
- Programmers can define global data structures
  - Language eliminates details of message passing, all communication calls are generated.
  - Programmer must still distinguish between local and non-local data.

Parallel Paradigms



Supercomputers are like Formula 1

- Do we need ever bigger supercomputers?

1. Always more expensive (> 10^8 euro)
2. Enormous power consumption (price = equals to cost!)
3. Efficiency decreases (<5 %)
4. Which applications need this power?