Practical Parallel Programming IV

The Message-Passing Paradigm

Jan Lemeire
Practical Parallel Programming
October - December 2021

Vrije Universiteit Brussel

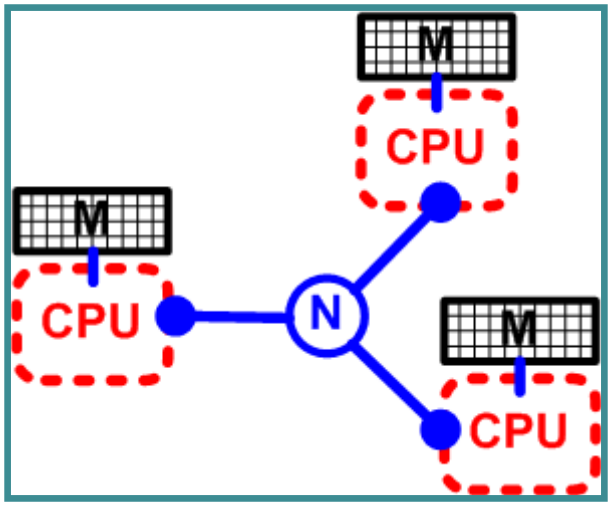
Overview

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 in some cases
  - *loosely synchronous* paradigm: between interactions (through messages), tasks execute completely asynchronously



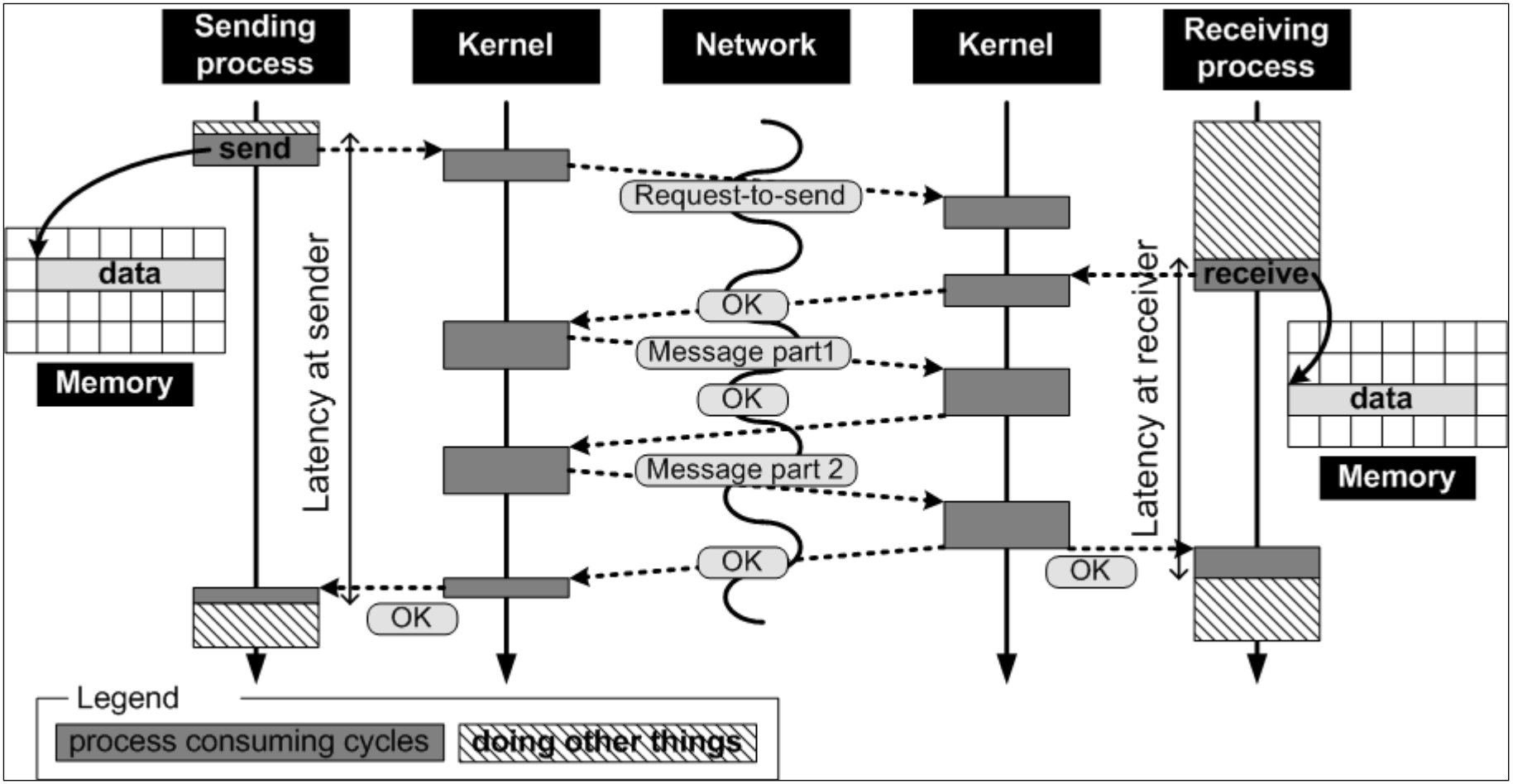
Messages…

- The ability to send and receive messages is all we need
  - void Send(message, destination)
  - char* Receive(source)
  - boolean IsThereAMessageForMe(source)

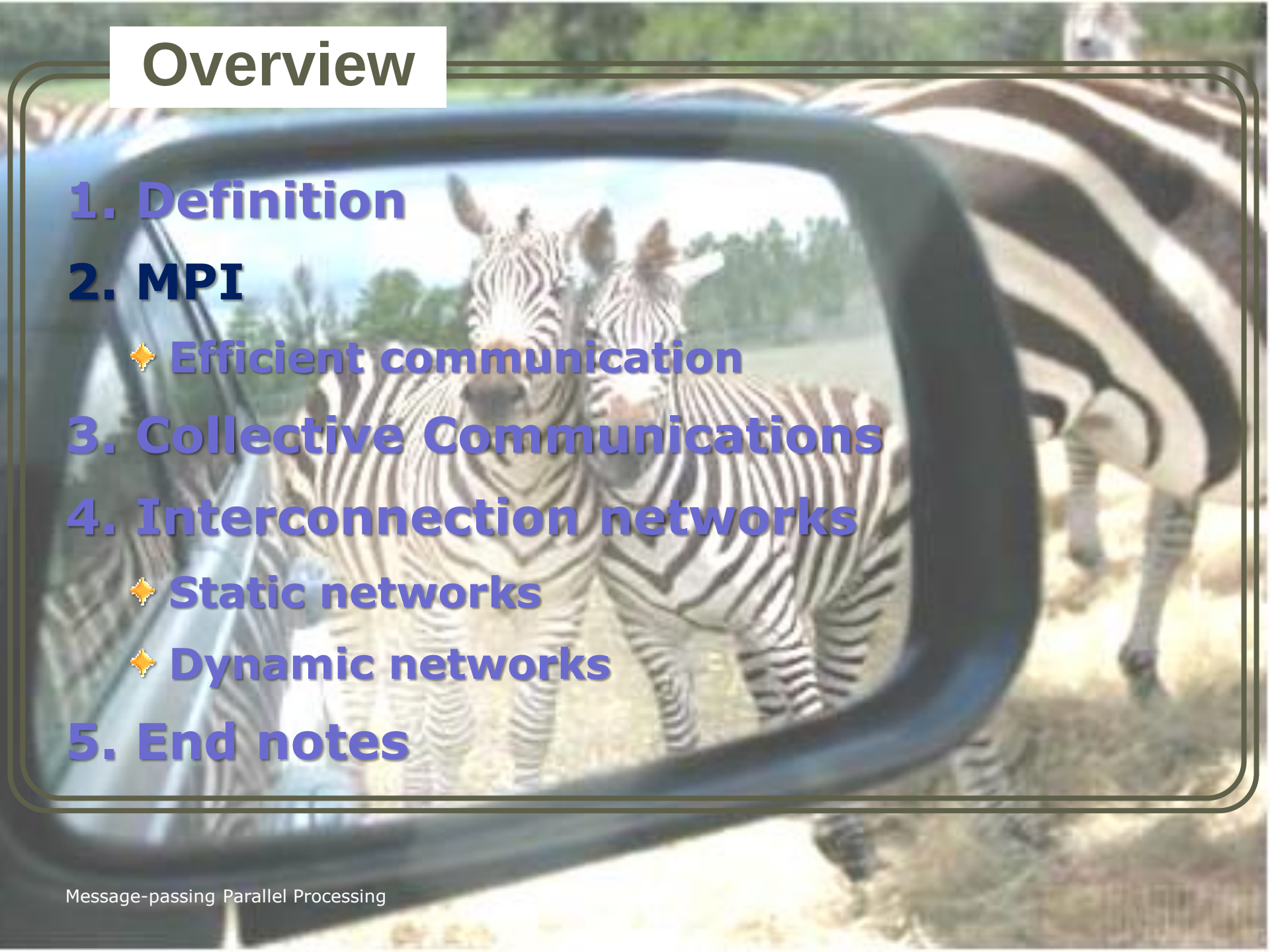
MPI_Probe() function in MPI

- But… we also want performance!
  - ➡ More functions will be provided

Message-passing



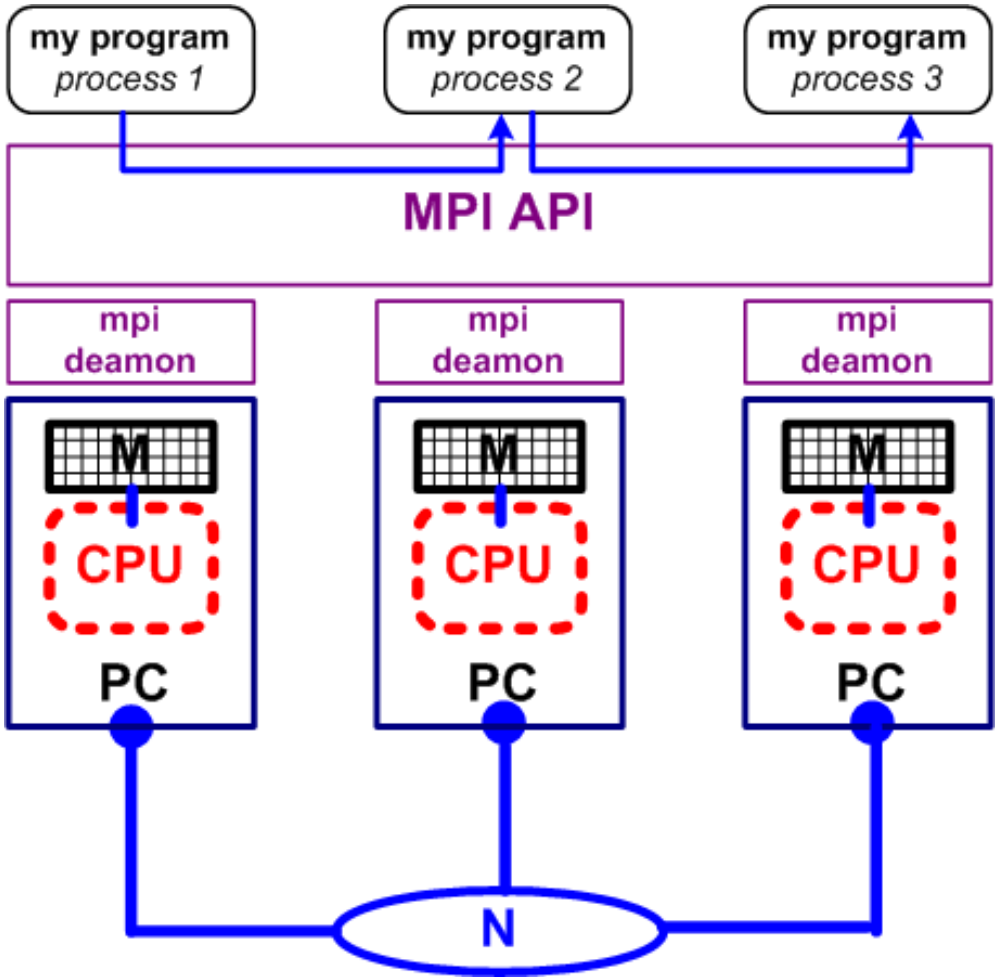
Overview



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 non-blocking communication calls


my program
process 1
my program
process 2
my program
process 3
MPI API
mpi
deamon
mpi
deamon
mpi
deamon
M
CPU
PC
M
CPU
PC
M
CPU
PC
N

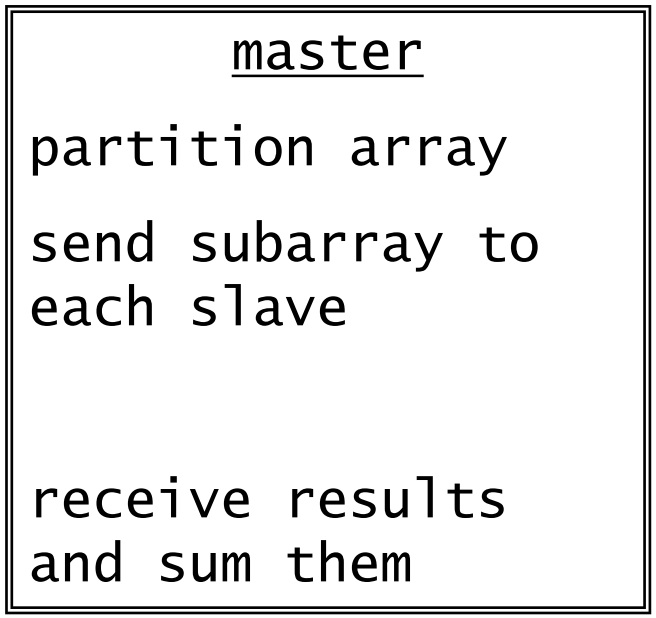
Fundamentals of MPI

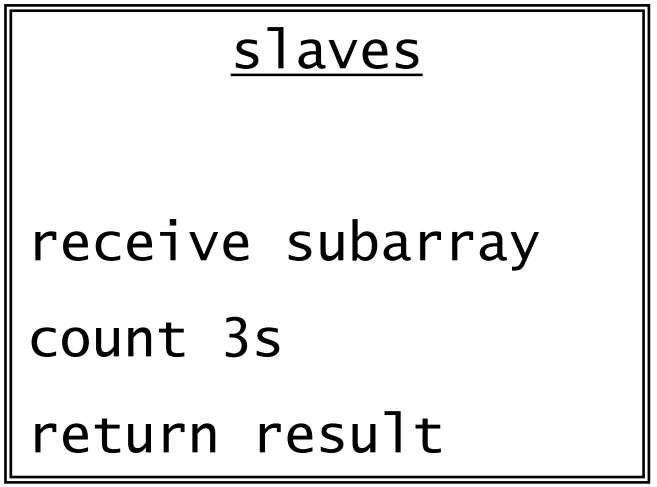
- Each process is identified by its **rank**, a counter starting from 0.
- **Tags** let you distinguish different types of messages
- **Communicators** let you specify groups of processes that can intercommunicate
  - Default is `MPI_COMM_WORLD`
- All MPI routines in C, data-types, and constants are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

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





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

```
MPI.Init();
int myRank, worldSize;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
if (myRank == 0) { // we choose rank 0 for master program
      int[] arr = createAndFillArray(ARRAY_SIZE ); // initialise data
      int count_par = 0;   int nbrSlaves = worldSize - 1;
      const int ELEMENTS_PER_SLAVE = ARRAY_SIZE / nbrSlaves;
      int* data_start = arr;
      for (int slaveID =1; slaveID < worldSize; slaveID++) {
         MPI_Send(data_start, ELEMENTS_PER_SLAVE, MPI_INT, slaveID, INPUT_TAG,
MPI_COMM_WORLD);
         data_start += ELEMENTS_PER_SLAVE;
      }
      // slaves are working...
      for (int slaveID = 1; slaveID < worldSize; slaveID++) {
         int count_slave;
         MPI_Recv(&count_slave, 1, MPI_INT, slaveID, RESULT_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
         count_par += count_slave;
     }
 } else {  // *** Slave Program ***
      MPI_Status status;   int data_size;
      MPI_Probe(0, INPUT_TAG, MPI_COMM_WORLD, &status); // slave waits here for message
      MPI_Get_count(&status, MPI_INT, &data_size);
      int* array = new int[data_size]; // check status to know data size and allocate buffer
      MPI_Recv(array, data_size, MPI_INT, 0, INPUT_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      int count = count3s(array, data_size); // the sequential program!
      MPI_Send(&count, 1, MPI_INT, 0, RESULT_TAG, MPI_COMM_WORLD);
}    MPI.Finalize(); // Don't forget!!
```

MPI Send and receive primitives

```
int MPI_Send(void  *buffer, int count, MPI_Datatype datatype,
             int destination, int tag, MPI_Comm communicator);
```

- buffer contains the data-to-be-sent
- tag is user-defined, it indicates the type of message

```
int MPI_Recv(void  *buffer, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm communicator,
             MPI_Status  *status );
```

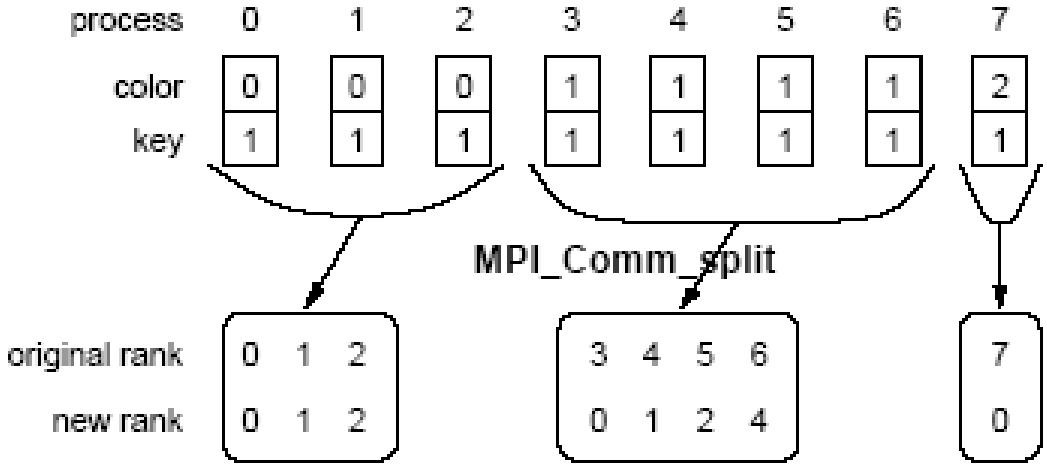
- buffer should be large enough to store the data (memory should be allocated)
- source can be **MPI_ANY_SOURCE** constant to specify that any source is acceptable
- tag can be the **MPI_ANY_TAG** constant to indicate that any tag is acceptable.

Communicators

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
  - Default is MPI_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

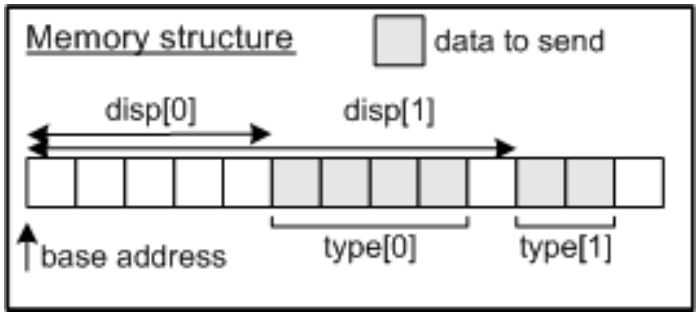


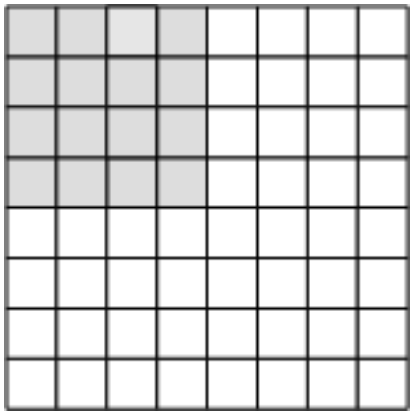
- A process has a specific rank in each communicator it belongs to.
- ***Other example***: use a different communicator in a library than application so that messages don't get mixed

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

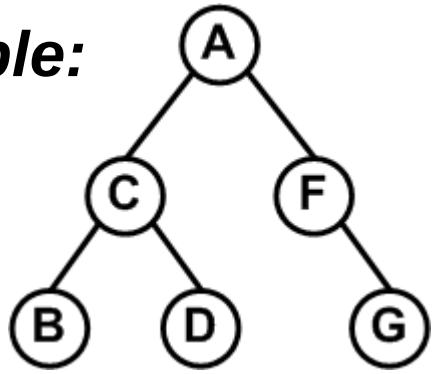


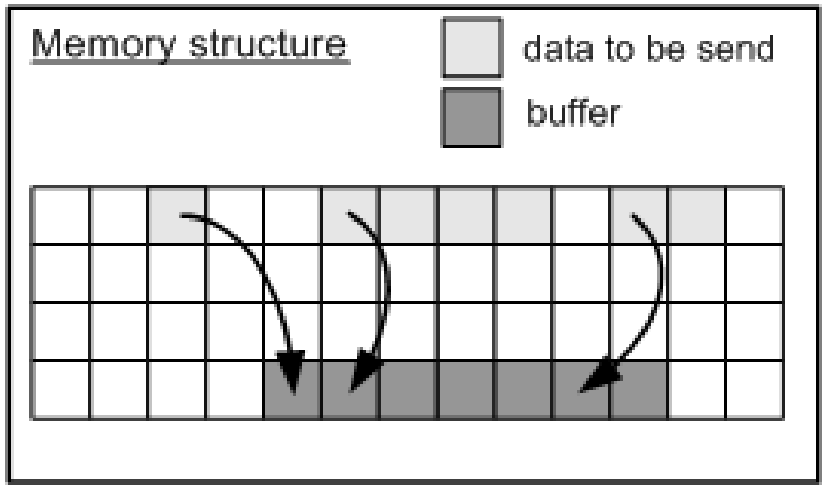


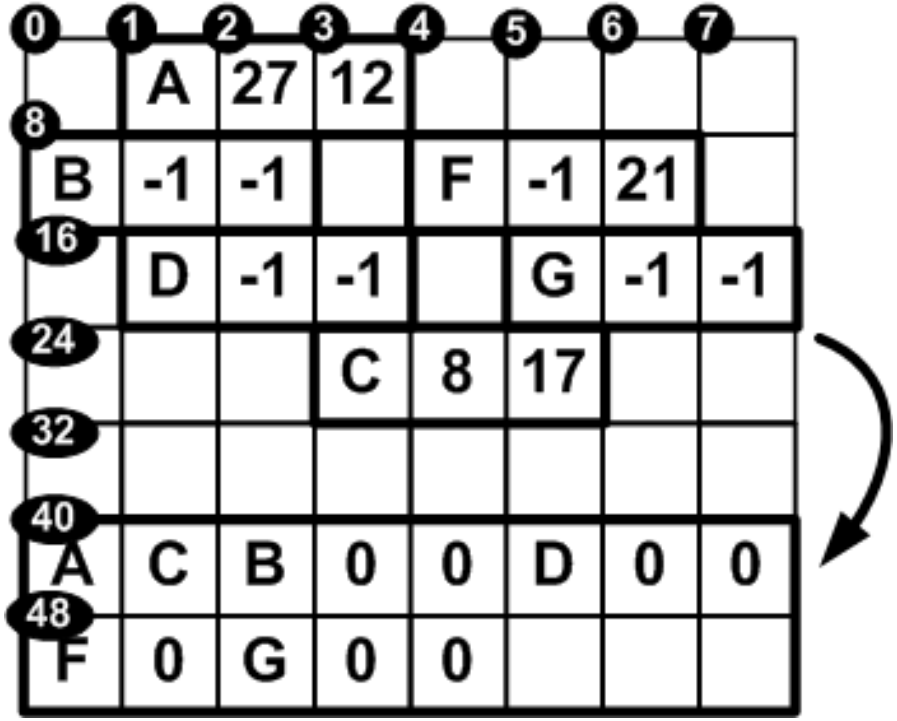
- Specify displacements and types => commit
- Irregular structure: use DataType.Struct
- Regular structure: Indexed, Vector, …
  - E.g. submatrix
- Alternative: packing & unpacking via buffer

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();
}
```

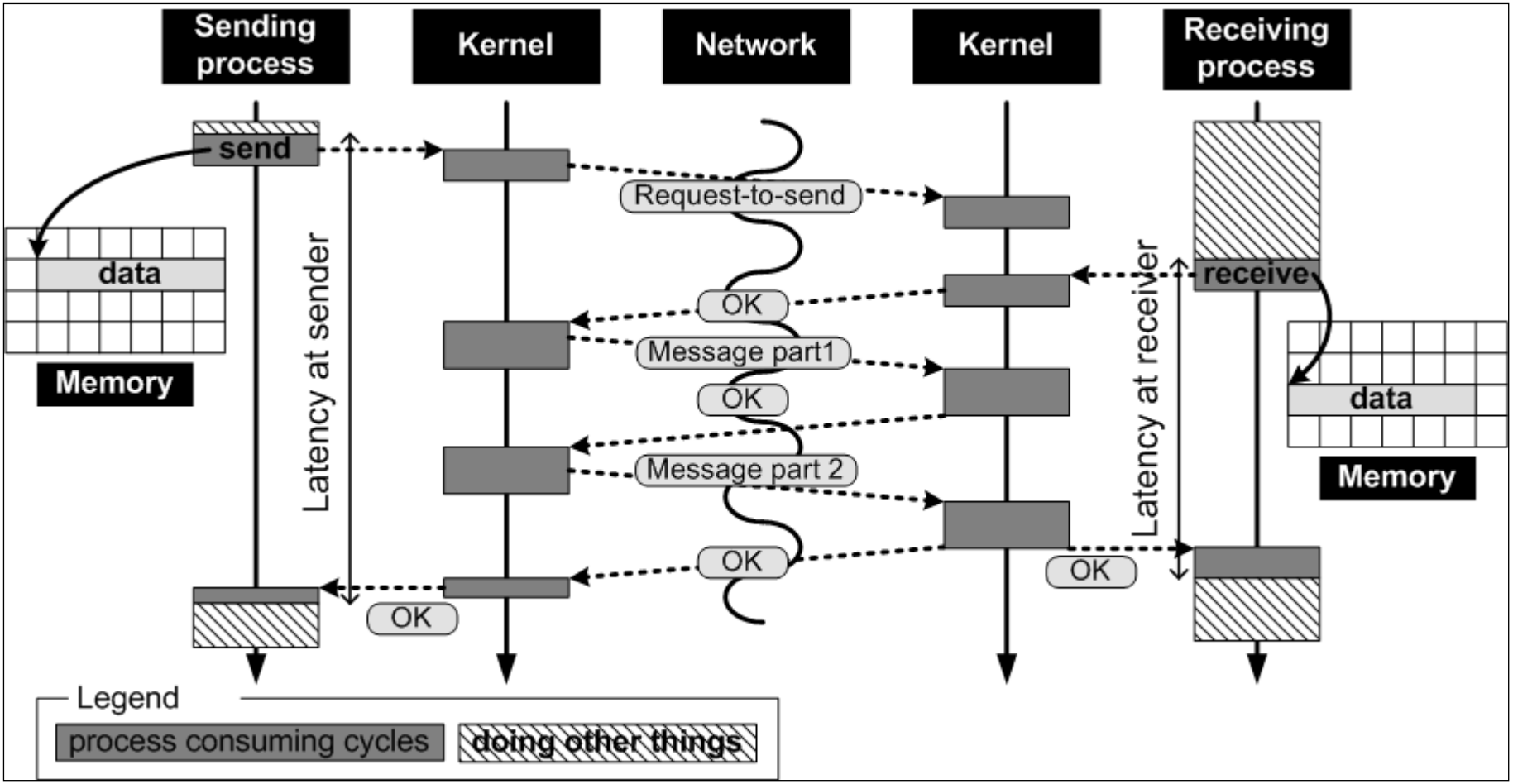
- Add serialVersionUID to denote class compatibility
  - private static final long serialVersionUID = 1;
- Attributes denoted as transient are not serialized

in C/C++: a standard library helping you with serialization

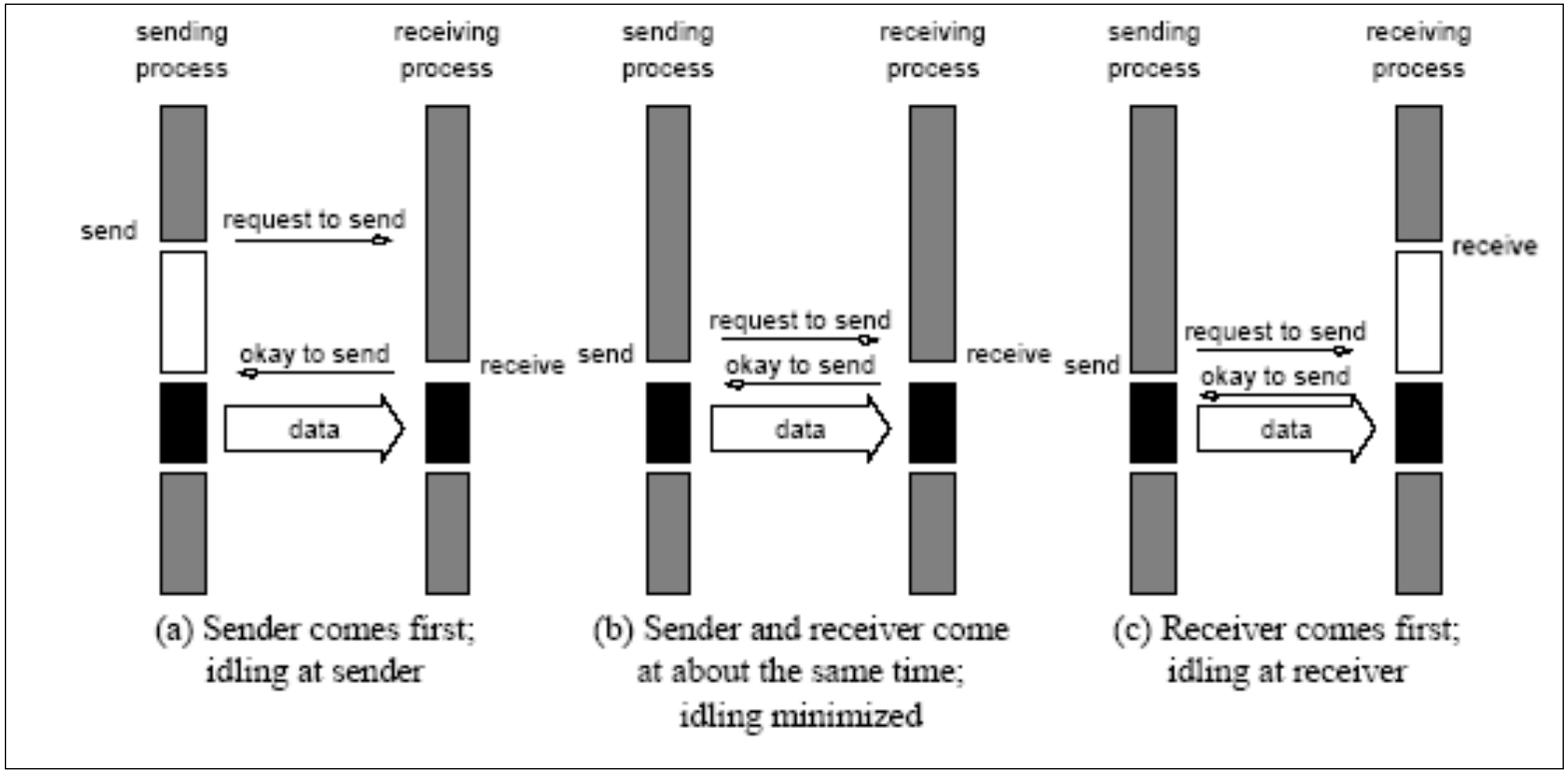
Overview

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

Message-passing

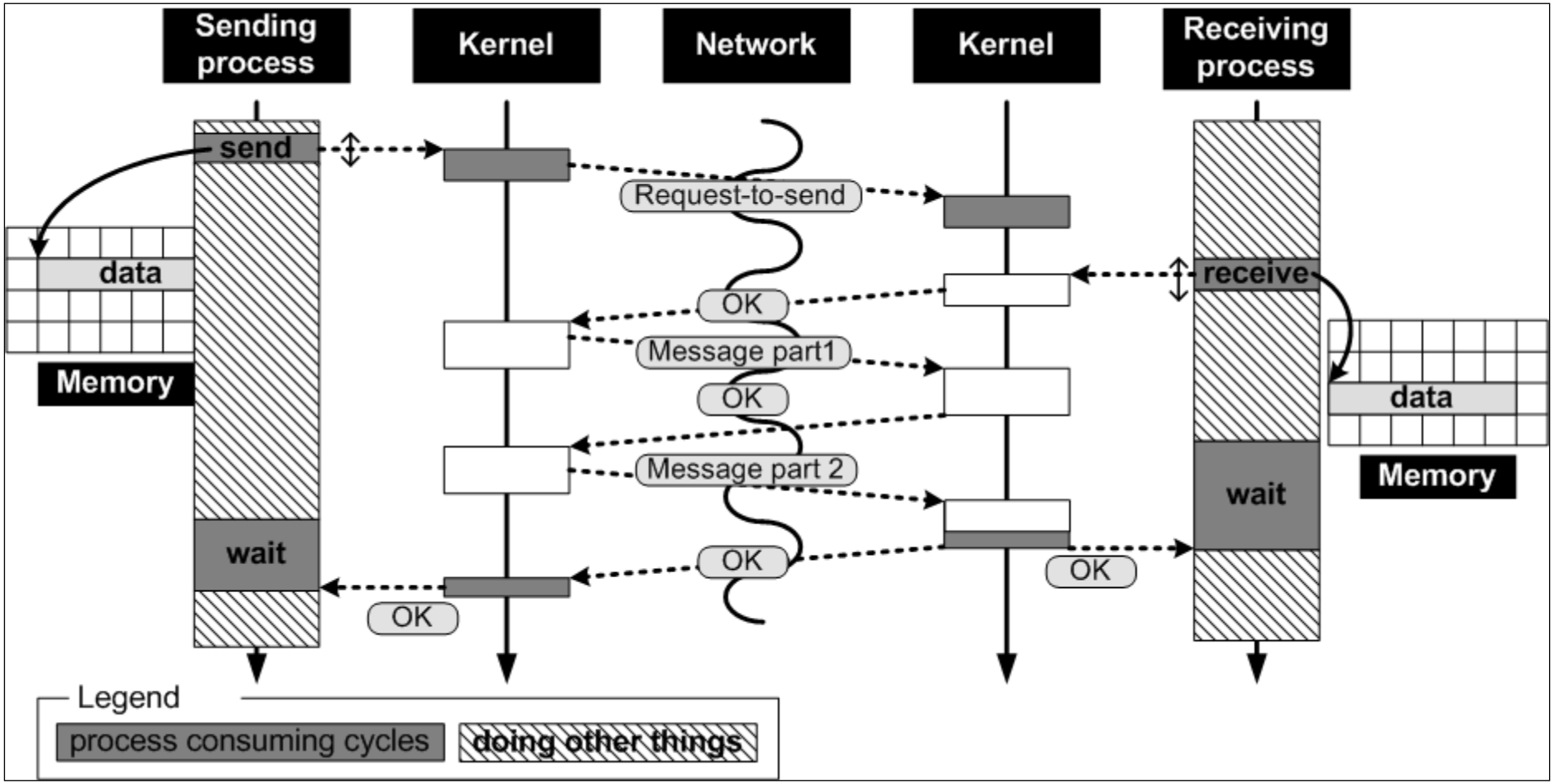


Non-Buffered Blocking Message Passing Operations



- *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&OS 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;
```

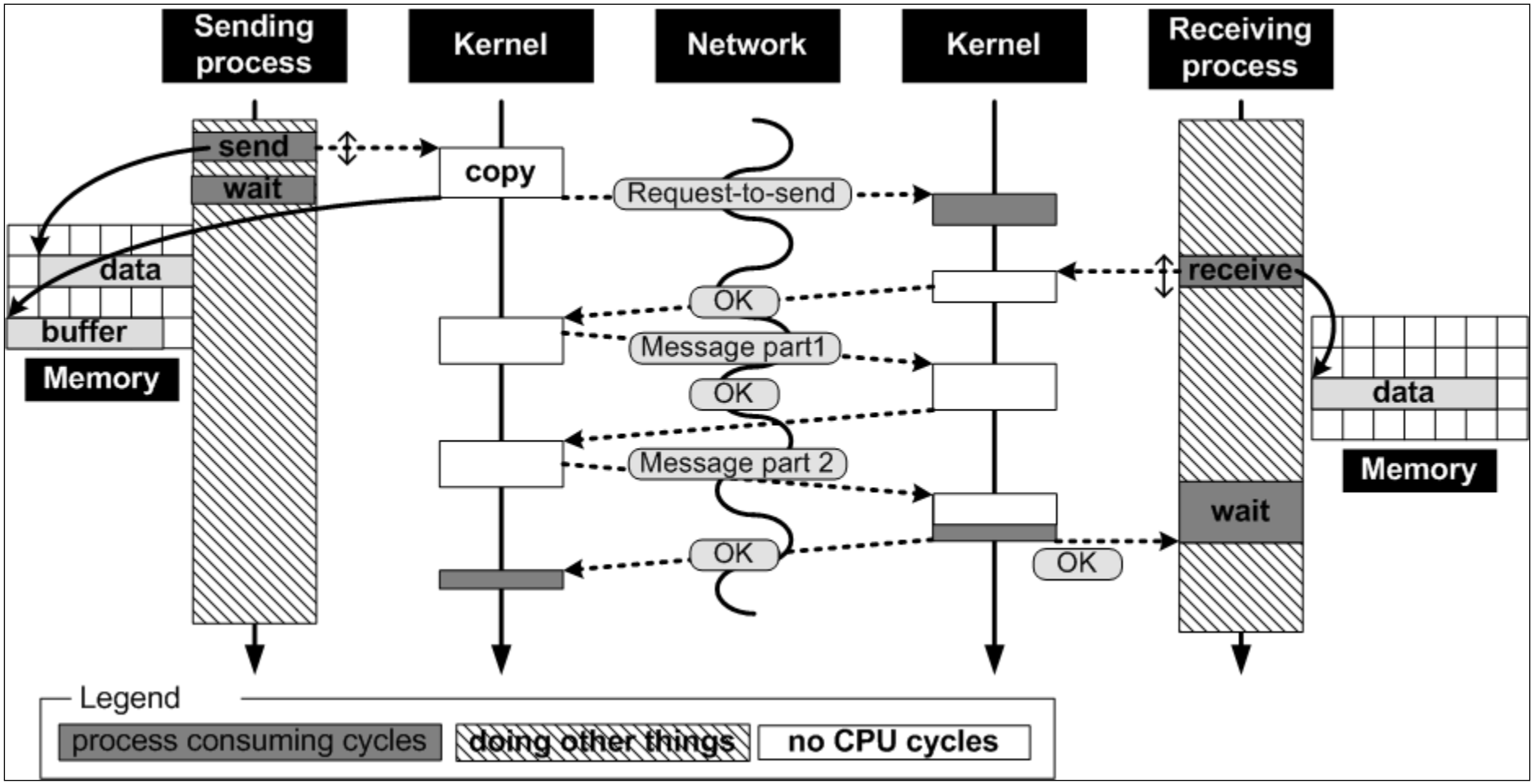
P1

```
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



Deadlock with blocking calls

All processes

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

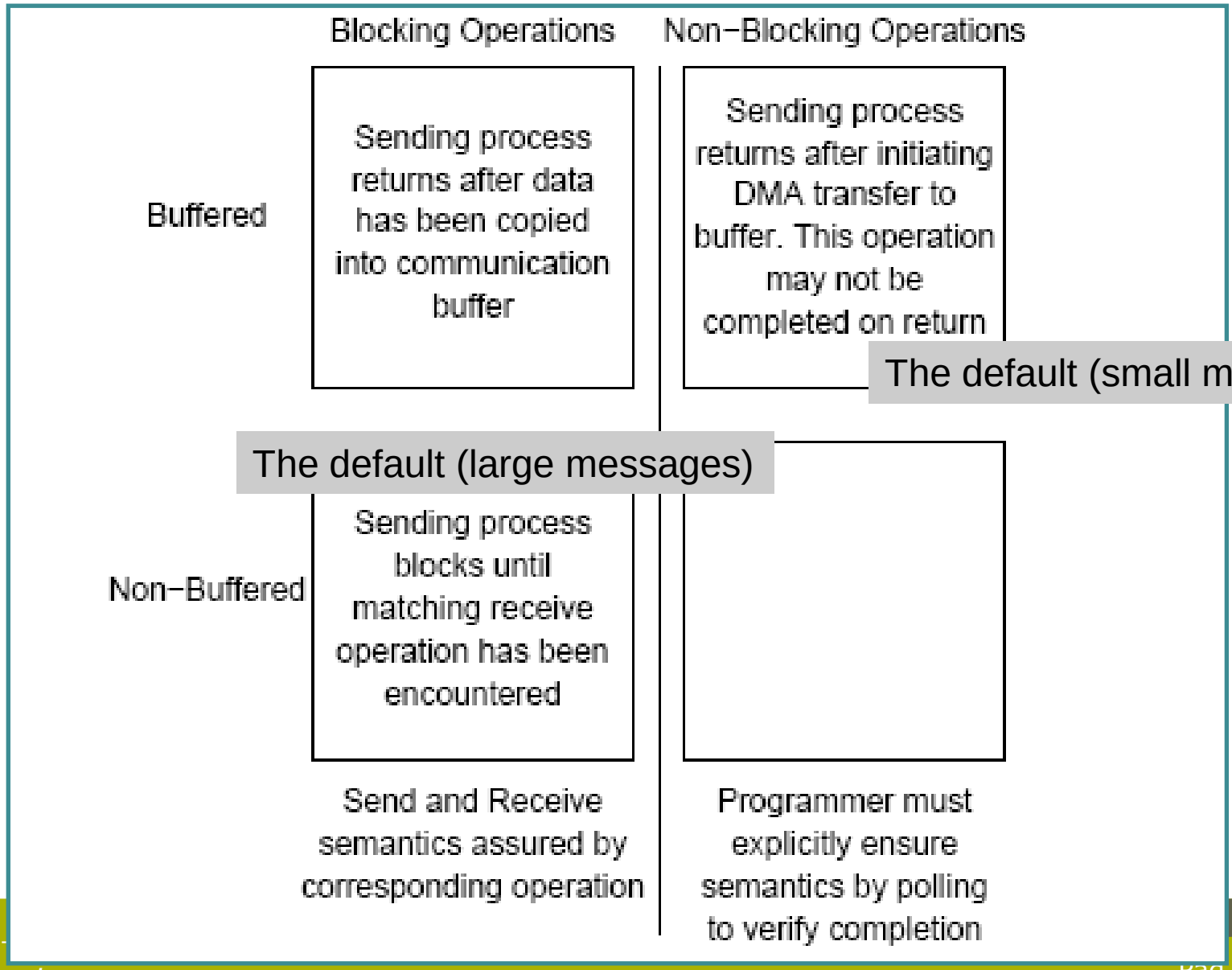
All processes

```
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 odd processor
- Buffered send
- Use non-blocking calls
  - Receive should use a different buffer!
- MPI built-in function: **MPI_sendrecv_replace**

Send and Receive Protocols



MPI Point-to-point communication

- Blocking
  - Returns if locally complete (<> globally complete)
- Non-blocking
  - Use MPI_Wait() & MPI_Test() for checking for completion of operation. Wait = blocking, Send = non-blocking
- Modes
  - *Buffered*
  - *Synchronous*: wait for a rendez-vous
  - *Ready*: no hand-shaking or buffering
    - Assumes corresponding receive is posted
- MPI_Sendrecv & MPI_Sendrecv_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*

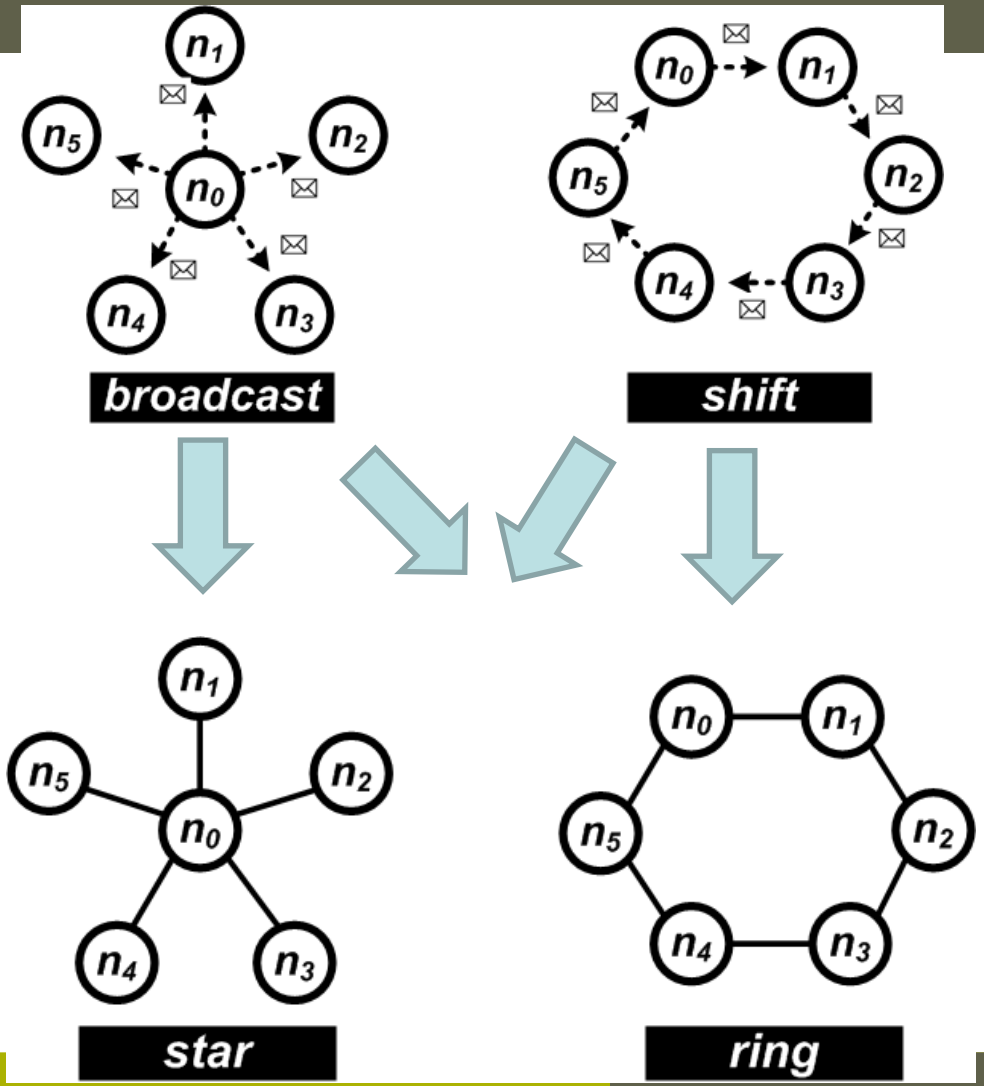
- The same program on master and slave

All processes

```
allocate subarray
scatter array from master to subarrays
count 3s
reduce subresults to master
```

```
int myRank, worldSize;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &worldSize);

if (myRank == 0)   // master contains the data
      int* arr = createAndFillArray(ARRAY_SIZE ); // initialise data

const int ELEMENTS_PER_SLAVE = ARRAY_SIZE / worldSize;

int* recv_array = new int[ELEMENTS_PER_SLAVE]; // allocate buffer
MPI_Scatter(input_array, ELEMENTS_PER_SLAVE, MPI_INT, recv_array, ELEMENTS_PER_SLAVE,
MPI_INT, 0, MPI_COMM_WORLD);

int count_process = 0;
for (int i = 0; i < ELEMENTS_PER_SLAVE; ++i)
    if (recv_array[i] == 3)
          count++;

int count_par; // global result
MPI_Reduce(&count_process, &count_par, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Optimization of Collective operations



MPI Collective Operations

- *Barrier synchronization* in MPI:

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

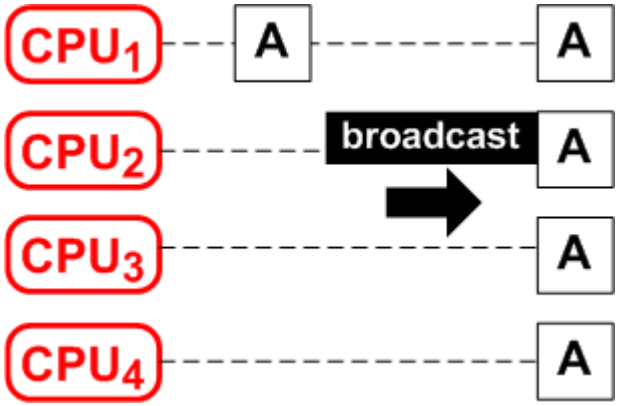
- The *one-to-all broadcast* operation is:

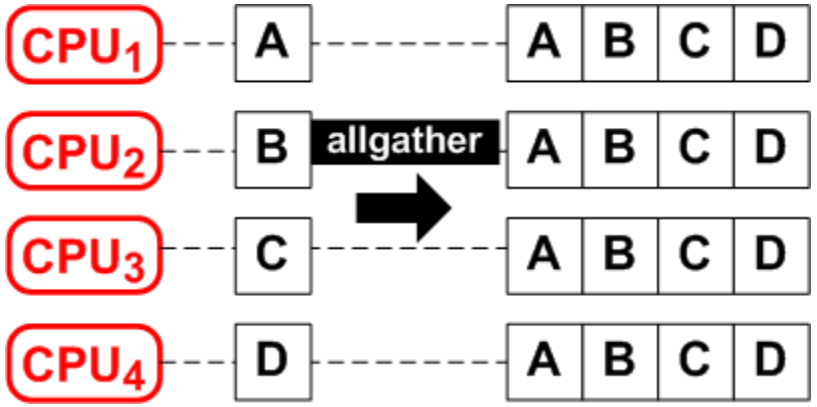
```
int MPI_Bcast(void *buf, int count, MPI_Datatype
datatype, int source, MPI_Comm comm)
```

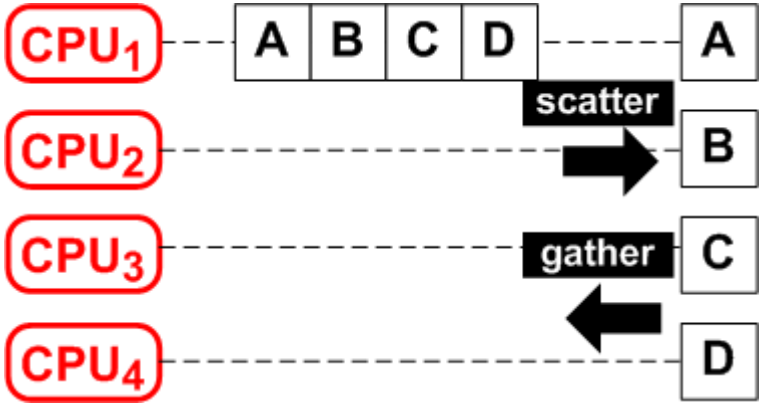
- The *all-to-one reduction* operation is:

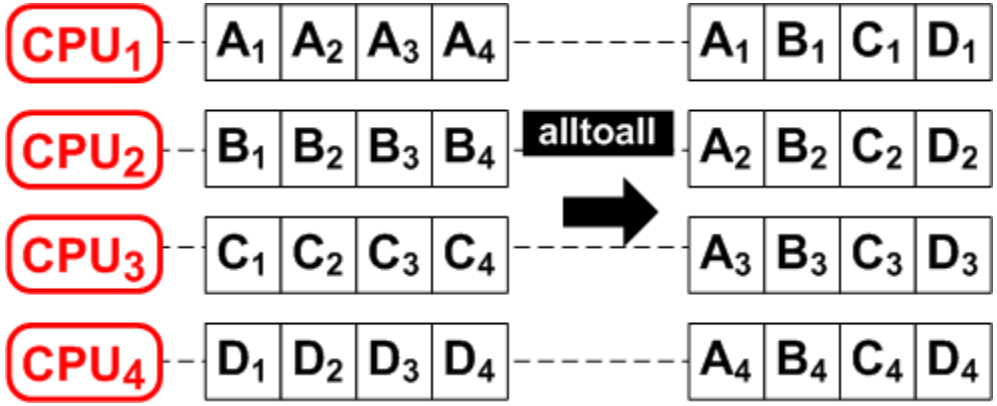
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int
target, MPI_Comm comm)
```

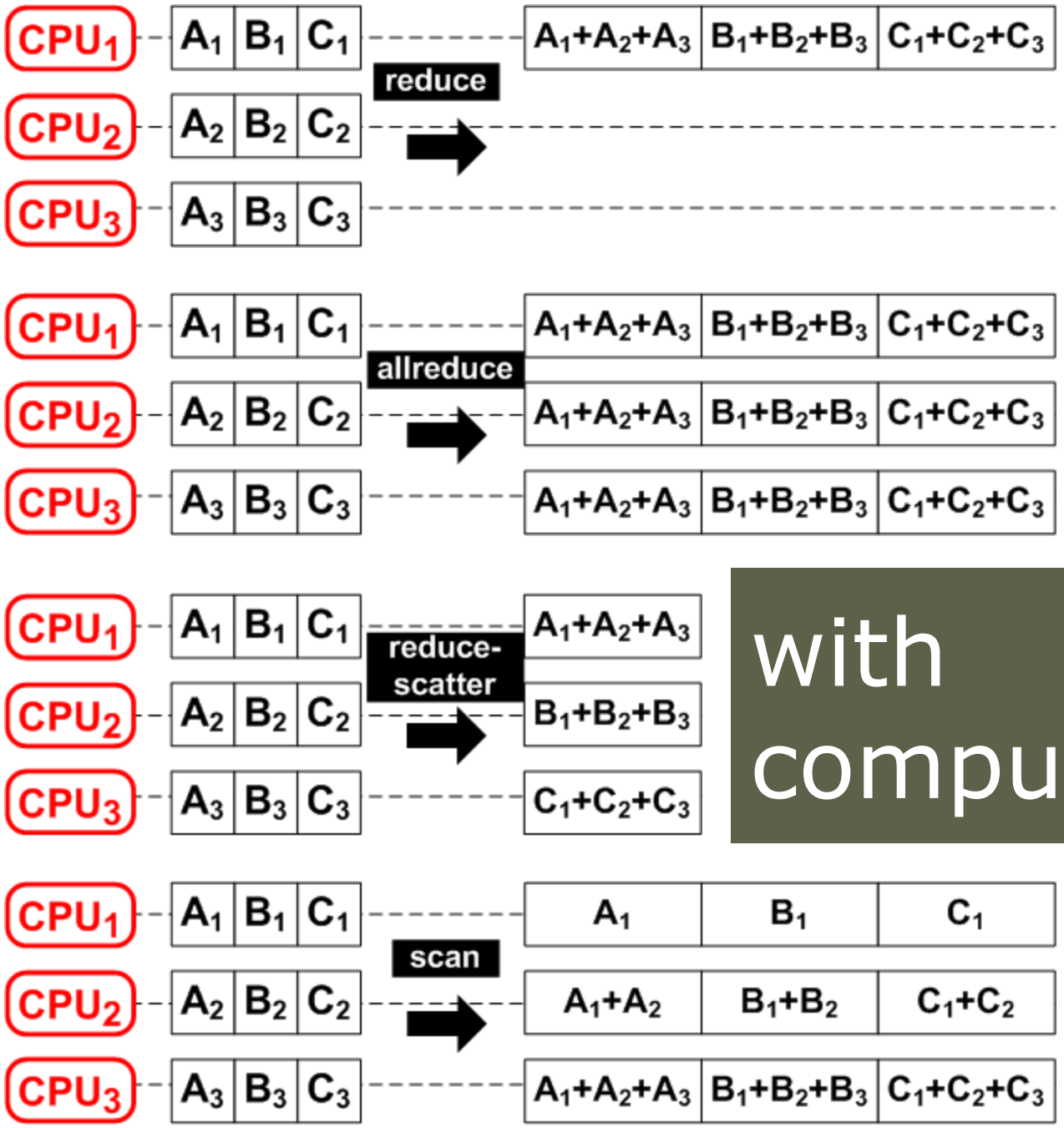
MPI Collective Operations









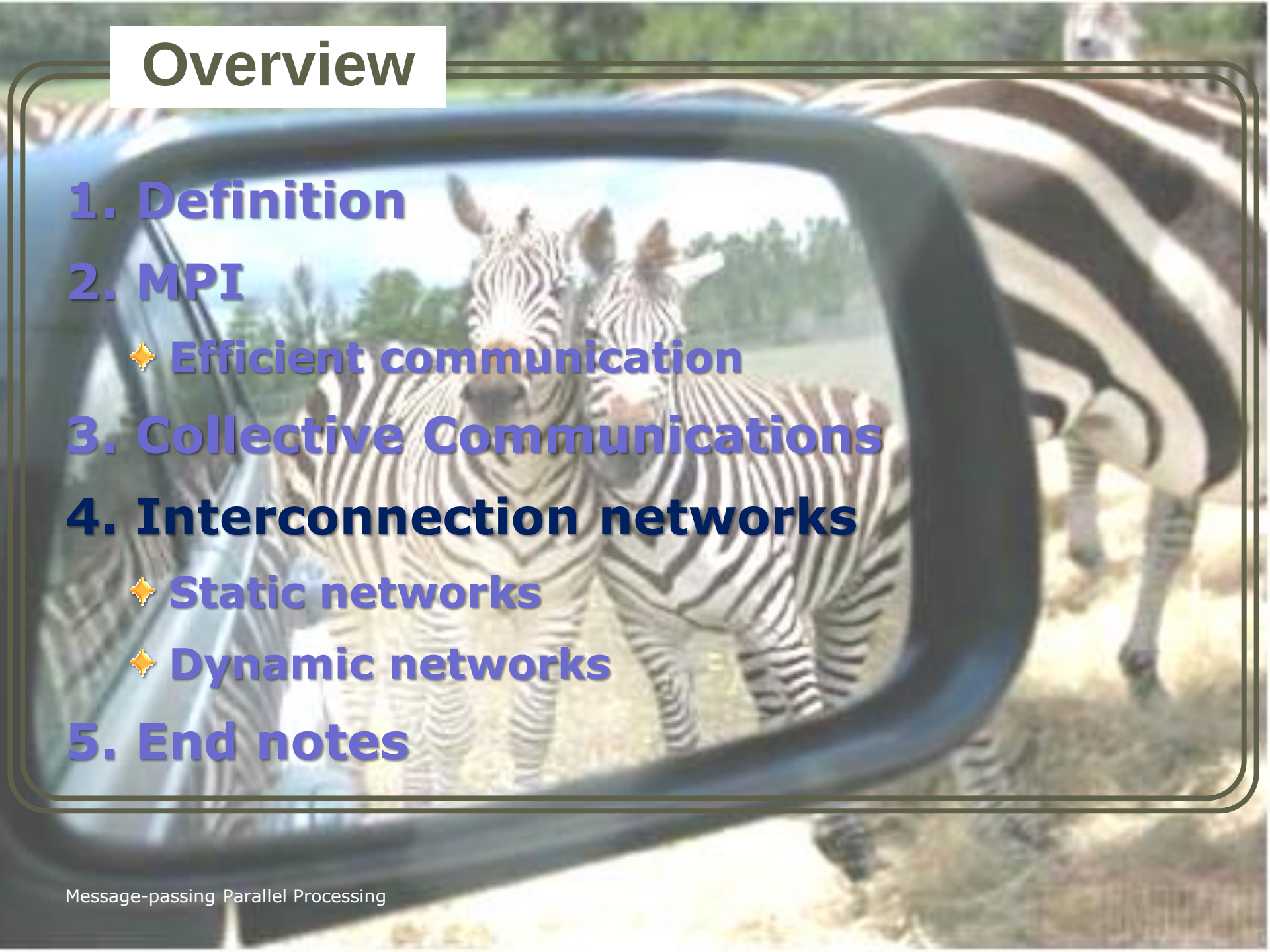


with computations

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

Overview

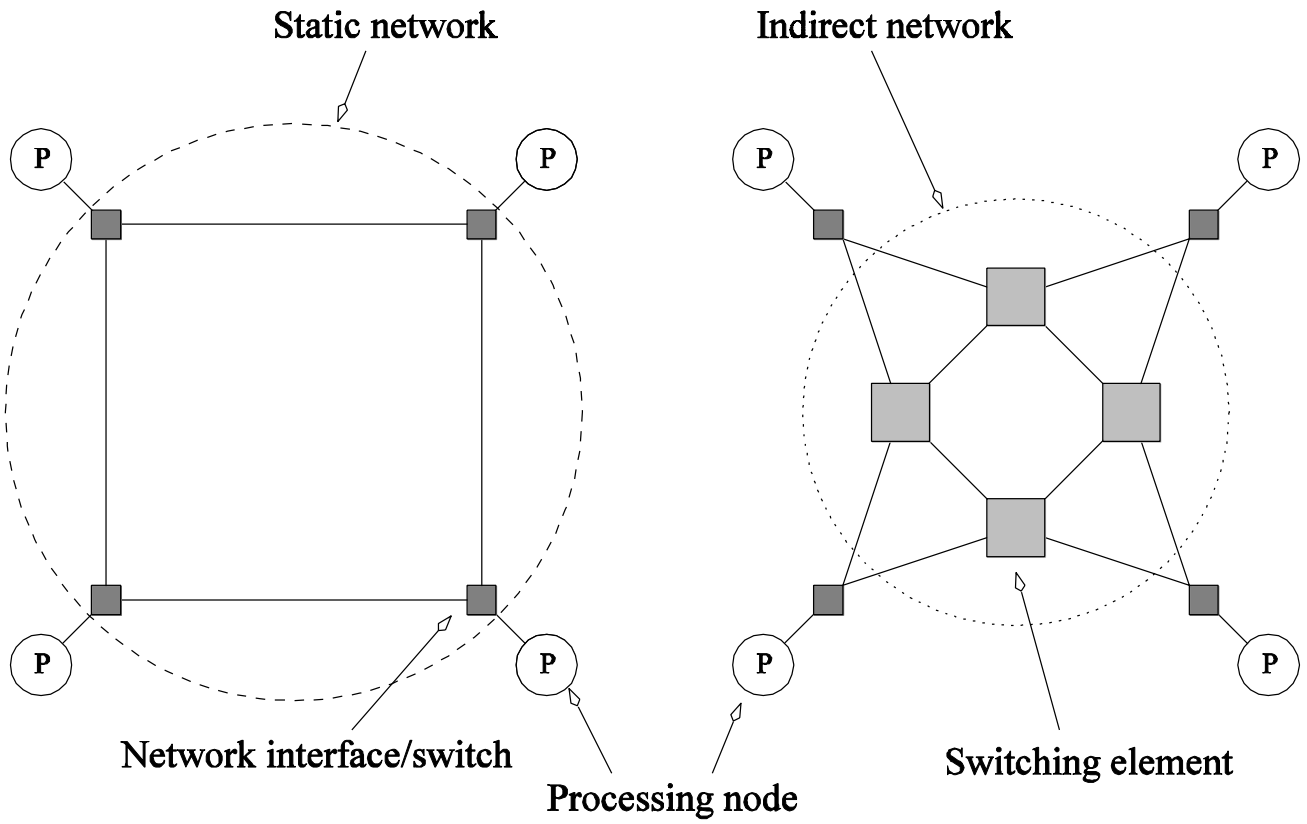


1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes

Interconnection Networks

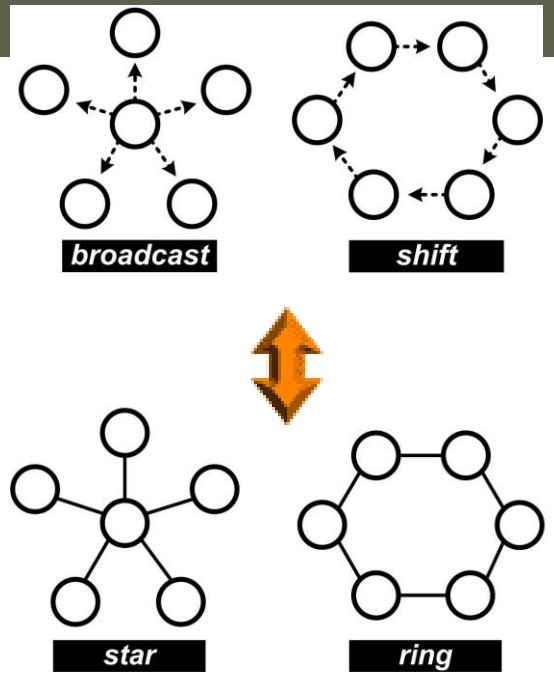
- Interconnection networks carry data between processors and memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as *static* or *dynamic*.
  - Static networks consist of point-to-point communication links among processing nodes and are also referred to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

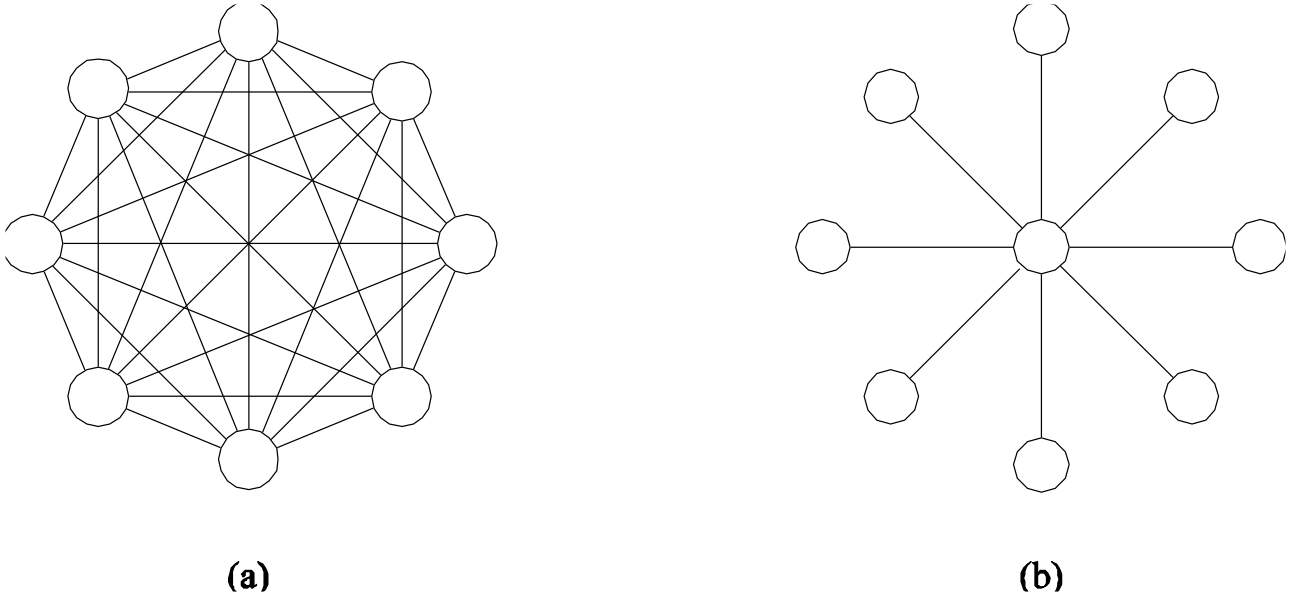


- Cost
- Difficulty to implement
- Scalability
  - Can processors be added with the same cost

Overview

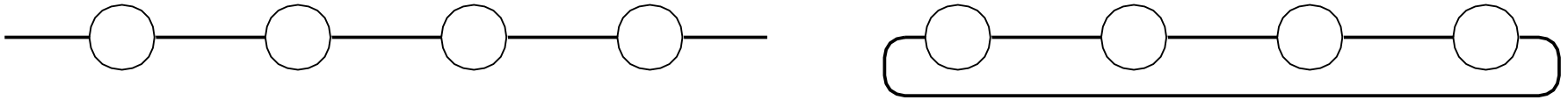
1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes

Network Topologies: Completely Connected and Star Connected Networks



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

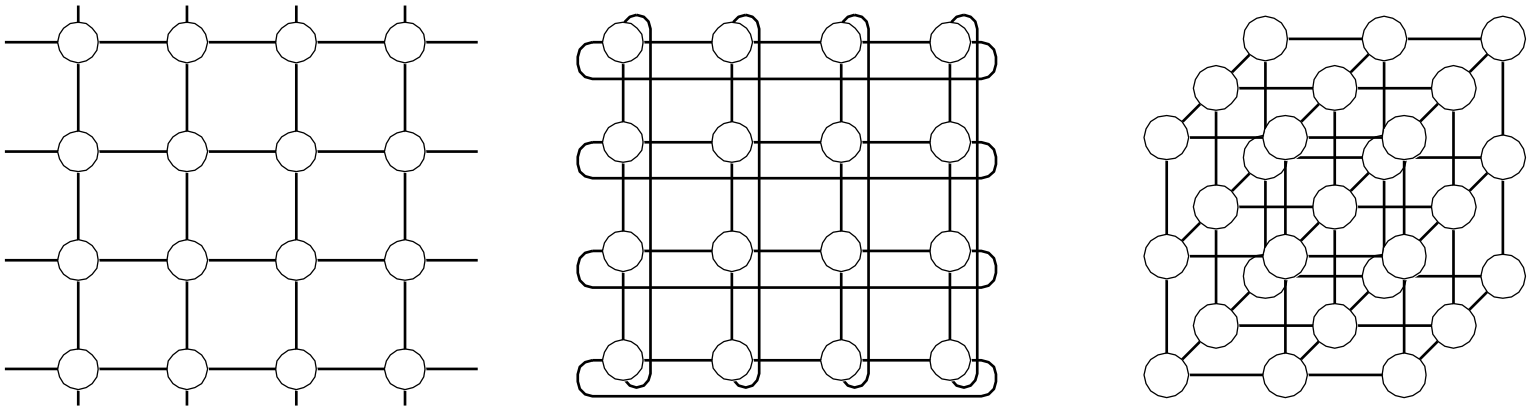
Linear Arrays



(a) (b)

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

Network Topologies: Two- and Three Dimensional Meshes

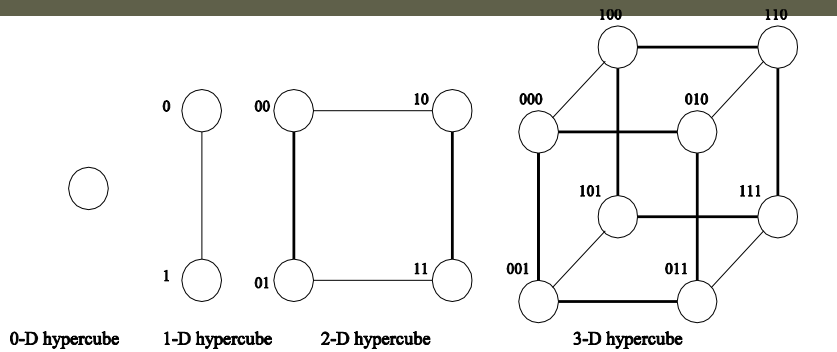


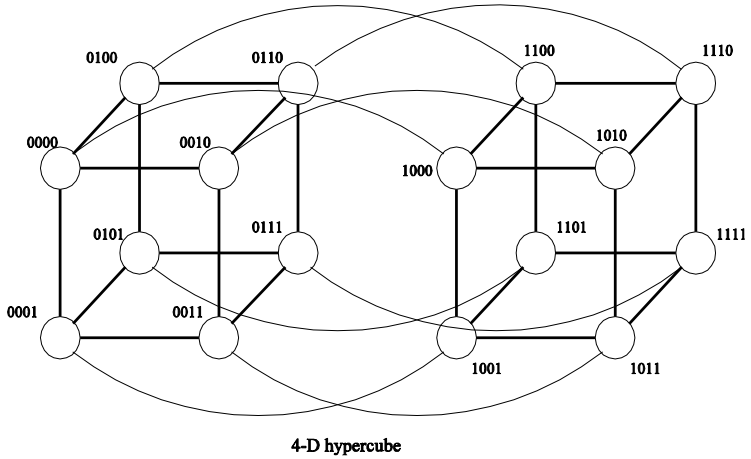
Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

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

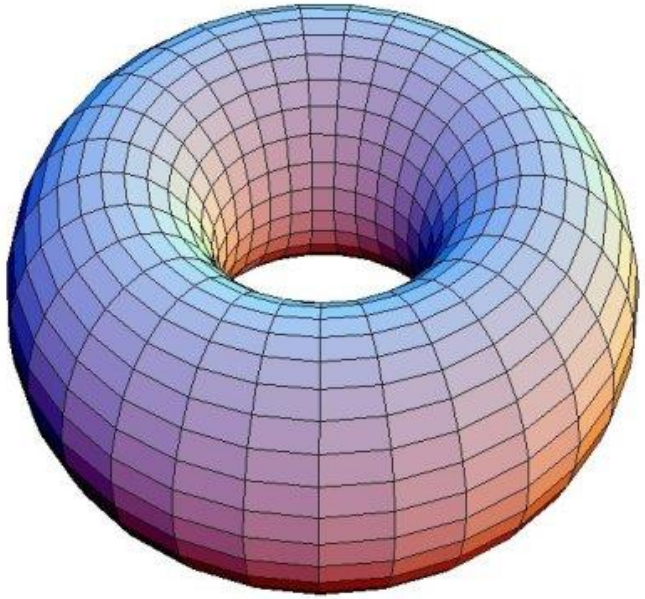
- In a ***linear array***, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a ***1D torus or a ring***.
- ***Mesh***: generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.
- A further 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





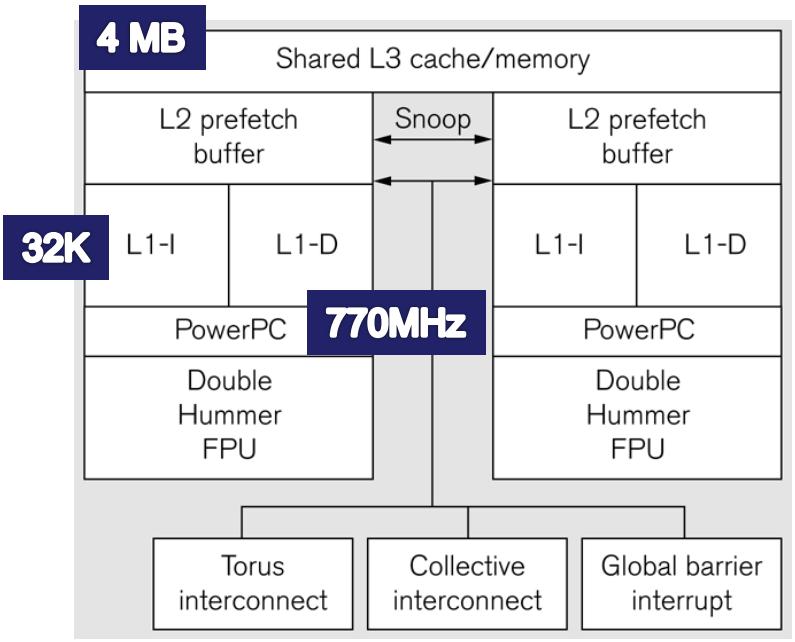
Construction of hypercubes from hypercubes of lower dimension.



Torus (2D wraparound mesh).

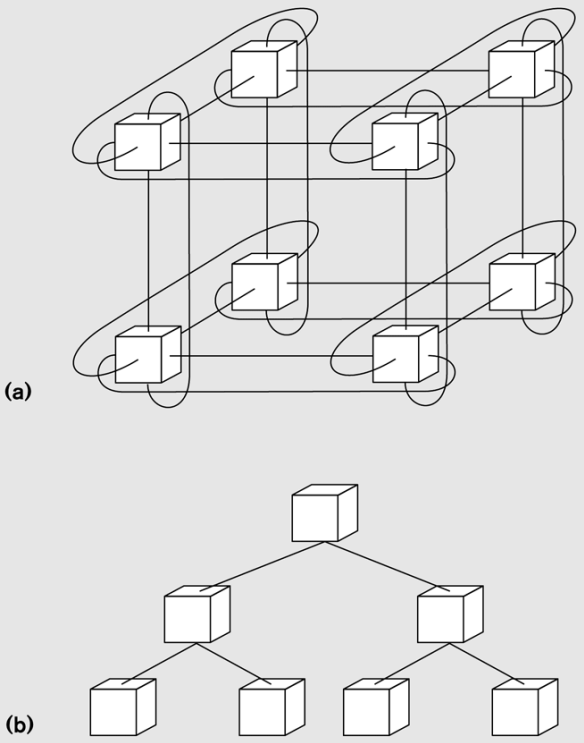
Super computer: BlueGene/L

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

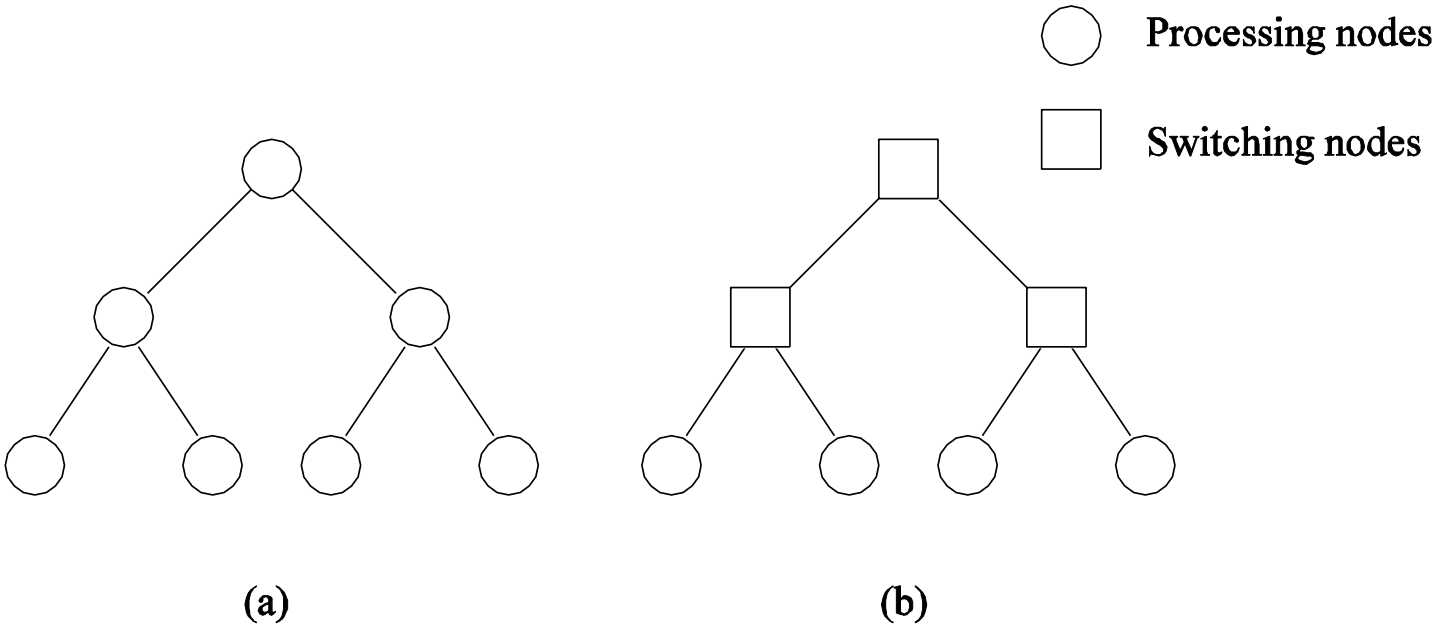


(a) 3D torus (64x32x32) for standard interprocessor data transfer
- Cut-through routing (see later)

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

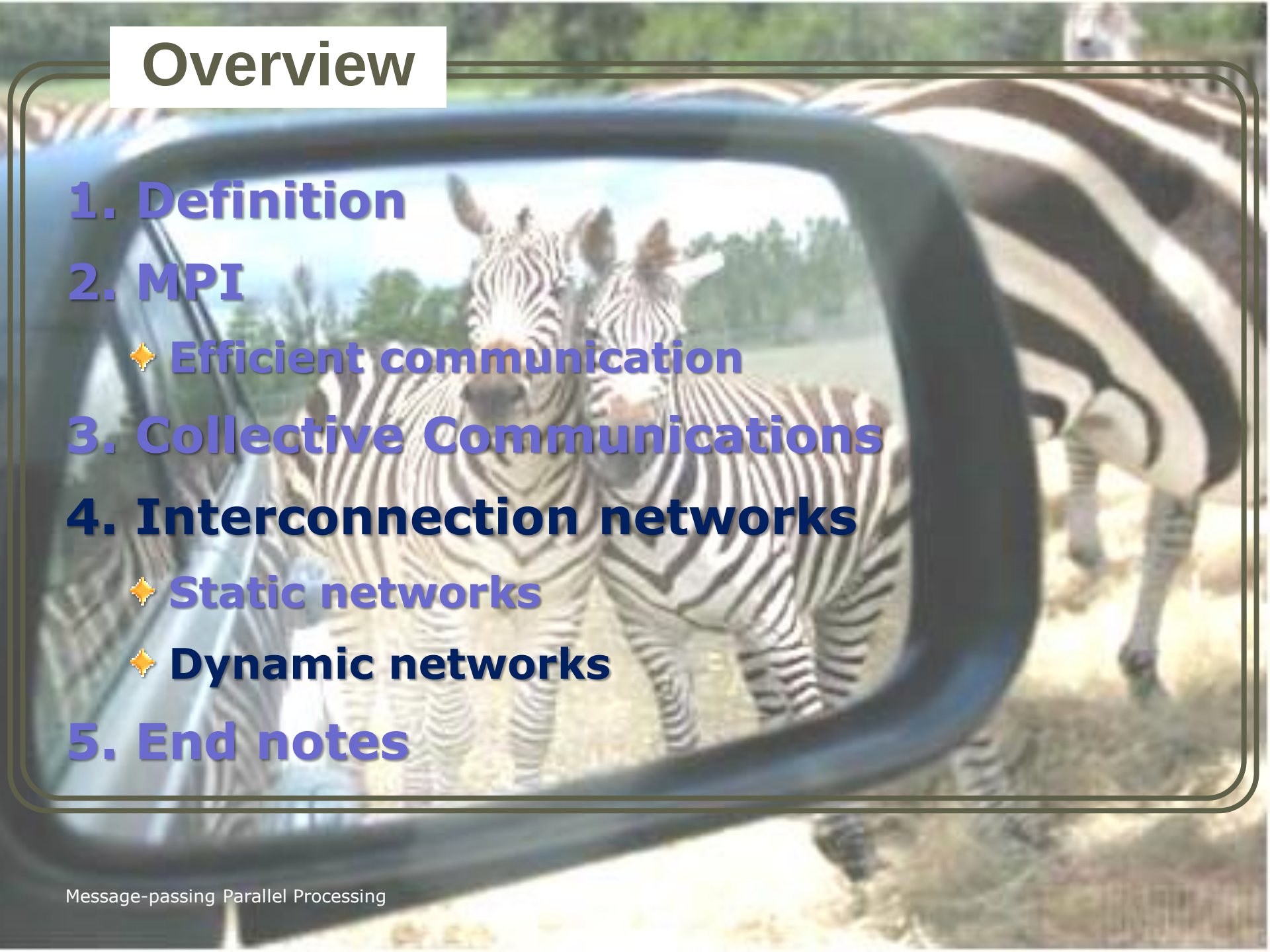
(c) Barrier network by a common wire

Network Topologies: Tree-Based Networks



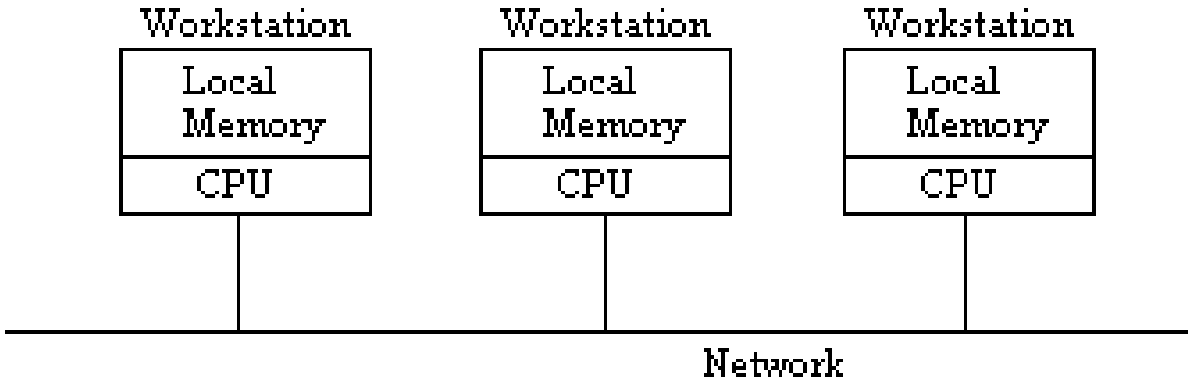
Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

Overview



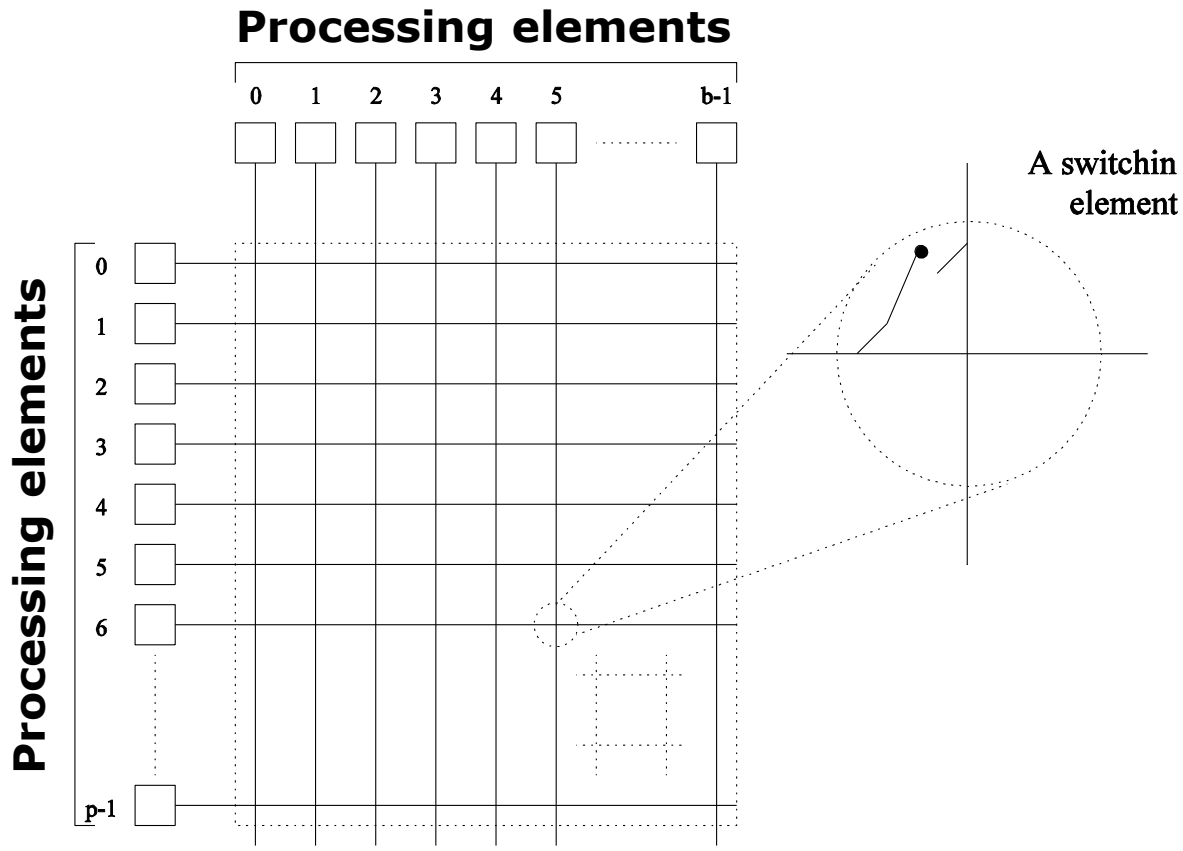
1. Definition
2. MPI
   - Efficient communication
3. Collective Communications
4. Interconnection networks
   - Static networks
   - Dynamic networks
5. End notes

Dynamic networks: Buses



Bus-based interconnect

Dynamic Networks: Crossbars

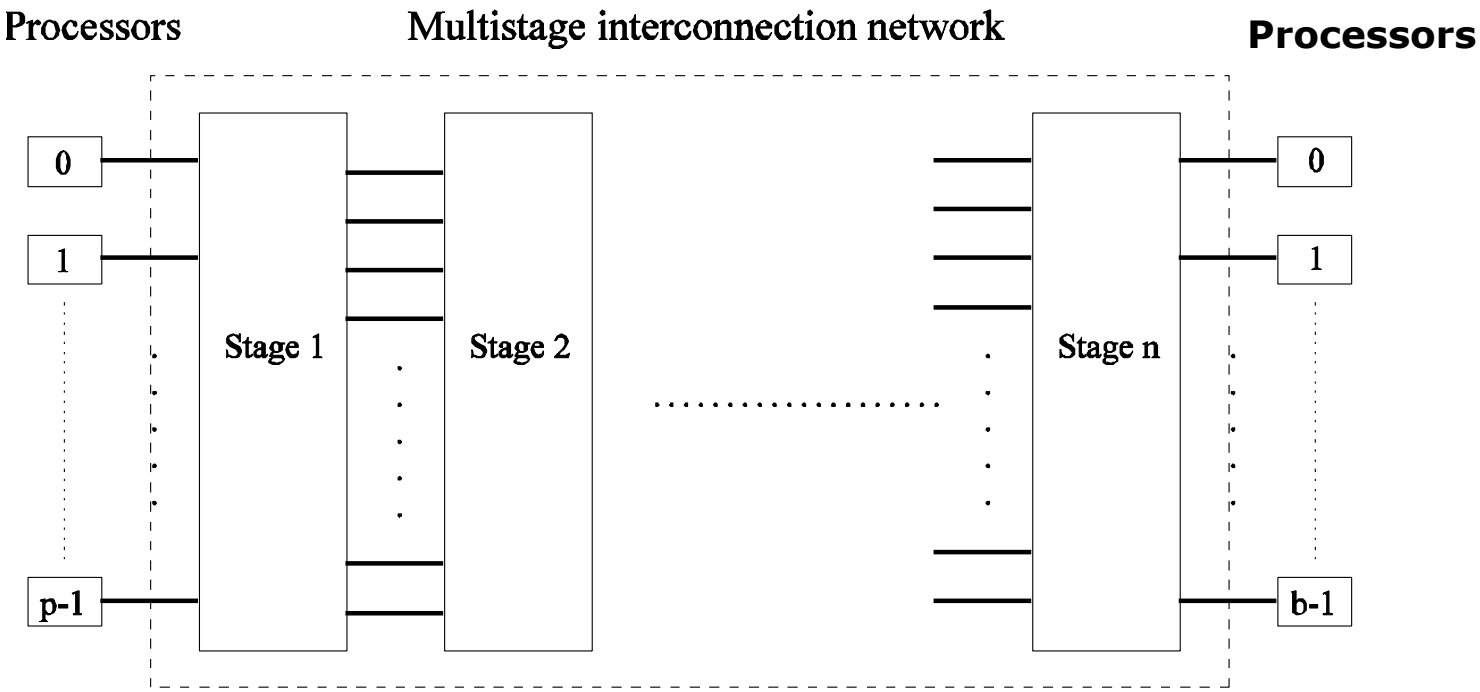


A crossbar network uses an $p \times m$ grid of switches to connect p inputs to m outputs in a non-blocking manner.

Multistage Dynamic Networks

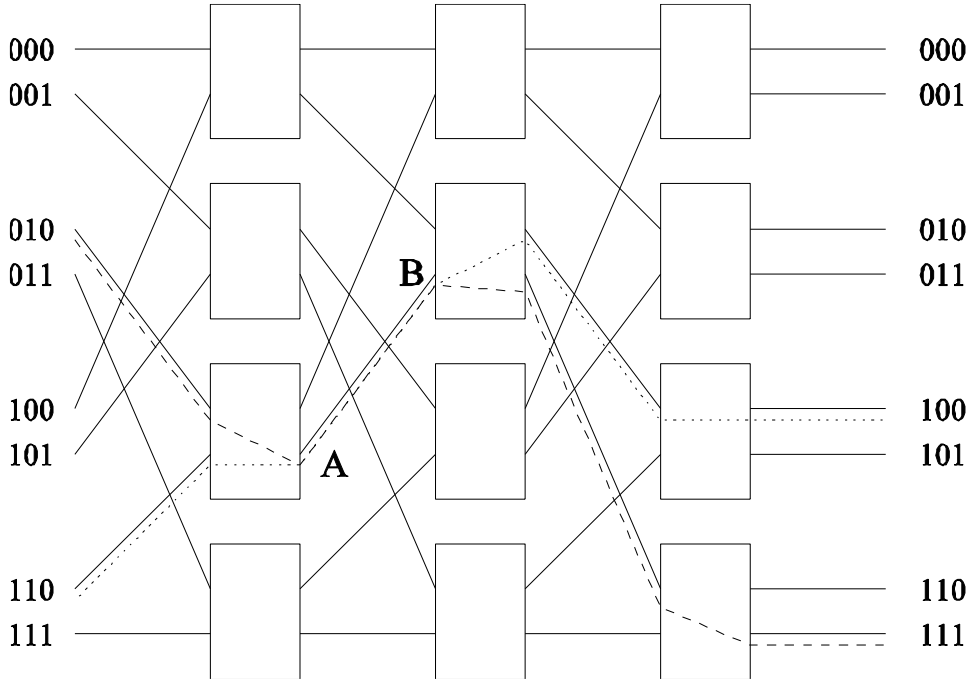
- Crossbars have excellent performance scalability but poor cost scalability.
  - The cost of a crossbar of p processors grows as $O(p^2)$.
  - This is generally difficult to scale for large values of p.
- Buses have excellent cost scalability, but poor performance scalability.
- Multistage interconnects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

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

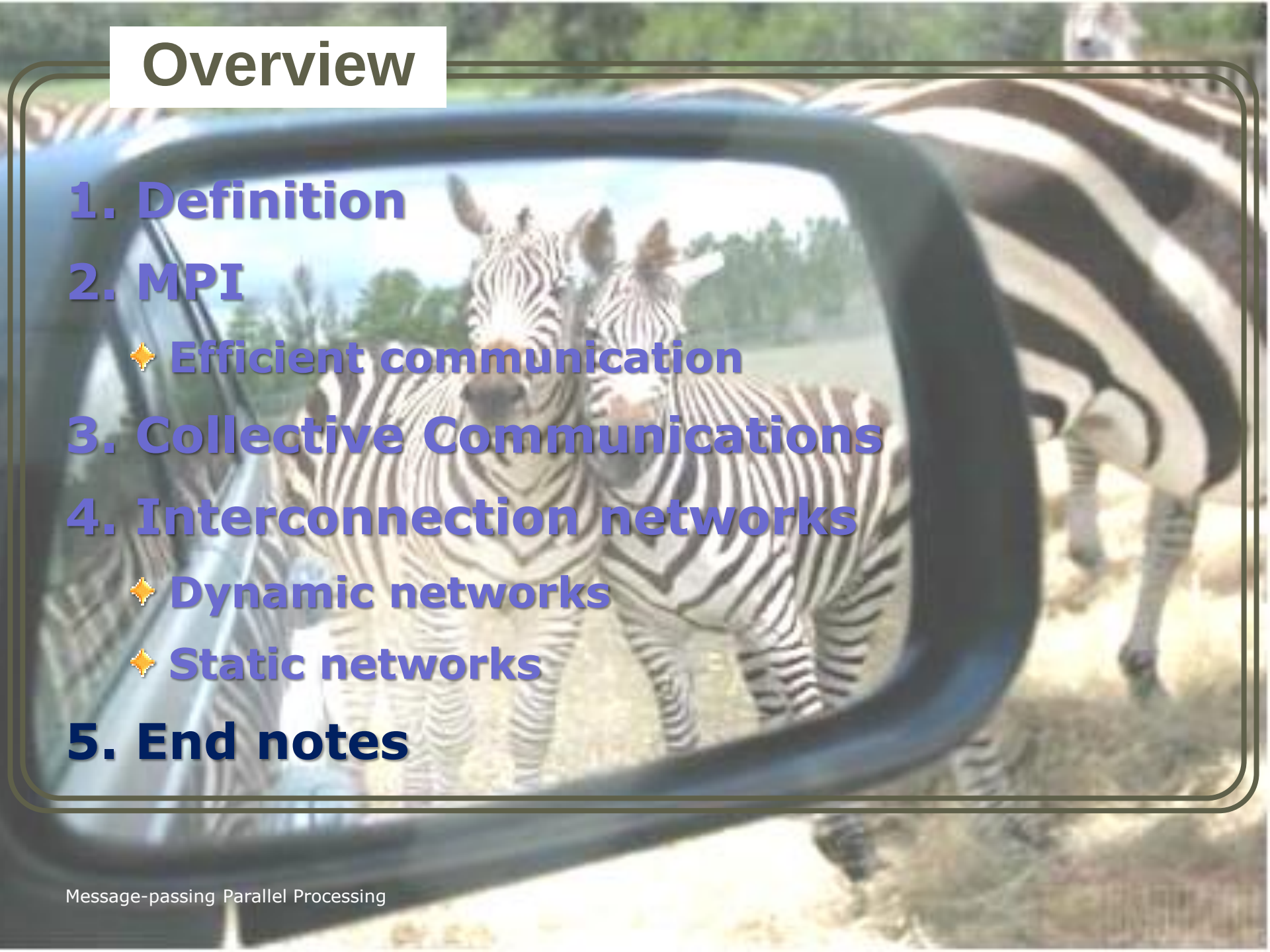
Memory Latency λ

- 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-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

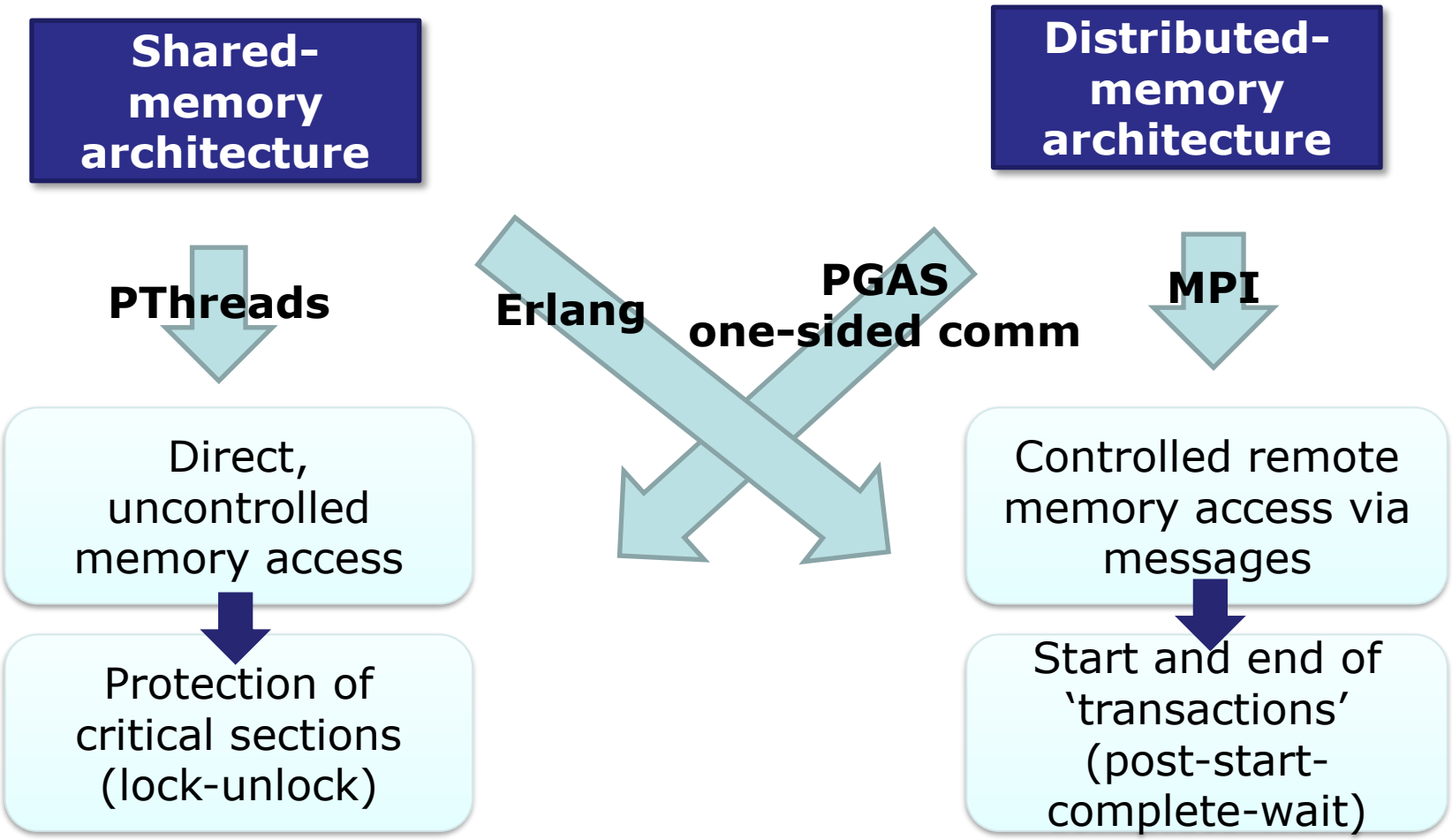
One-sided primitives

- 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)

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