

# Chapter 1: Introduction

# Definition

- Autonomous processors communicating over a communication network
- Some characteristics
  - ▶ No common physical clock
  - ▶ No shared memory
  - ▶ Geographical separation
  - ▶ Autonomy and heterogeneity

# Distributed System Model

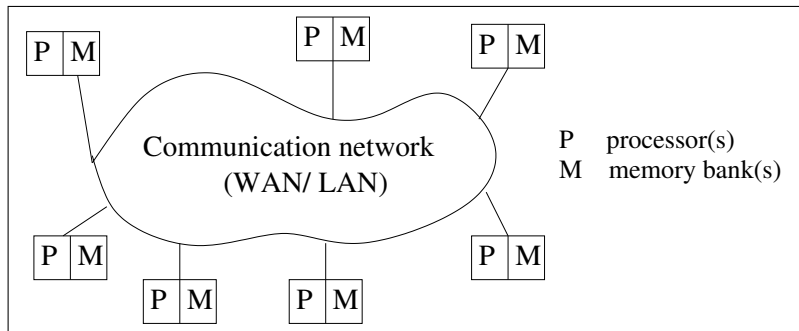


Figure 1.1: A distributed system connects processors by a communication network.

# Relation between Software Components

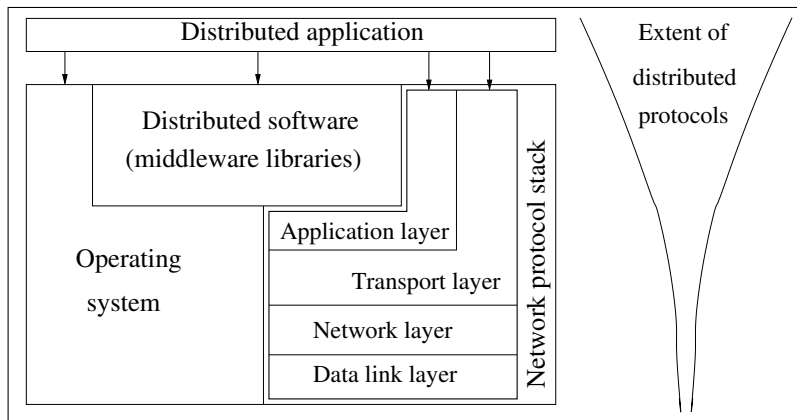











Figure 1.2: Interaction of the software components at each process.

# Motivation for Distributed System

- Inherently distributed computation 
- Resource sharing 
- Access to remote resources 
- Increased performance/cost ratio
- Reliability
  -  availability, integrity, fault-tolerance 
  -  scalability 
  -  Modularity and incremental expandability 

# Parallel Systems

- Multiprocessor systems (direct access to shared memory, UMA model)
  - ▶ Interconnection network - bus, multi-stage switch
  - ▶ E.g., Omega, Butterfly, Clos, Shuffle-exchange networks
  - ▶ Interconnection generation function, routing function
- Multicomputer parallel systems (no direct access to shared memory, NUMA model)
  - ▶ bus, ring, mesh (w w/o wraparound), hypercube topologies
  - ▶ E.g., NYU Ultracomputer, CM\* Connecticut Machine, IBM Blue gene
- Array processors (colocated, tightly coupled, common system clock)
  - ▶ Niche market, e.g., DSP applications

# UMA vs. NUMA Models

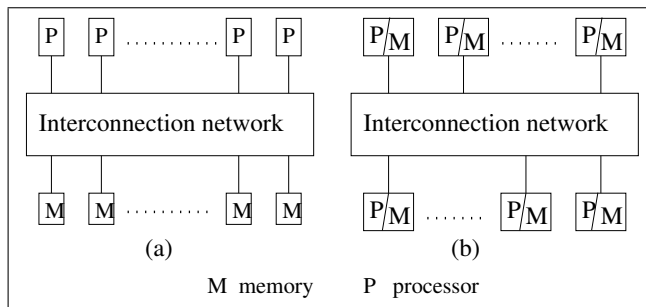


Figure 1.3: Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.

# Omega, Butterfly Interconnects

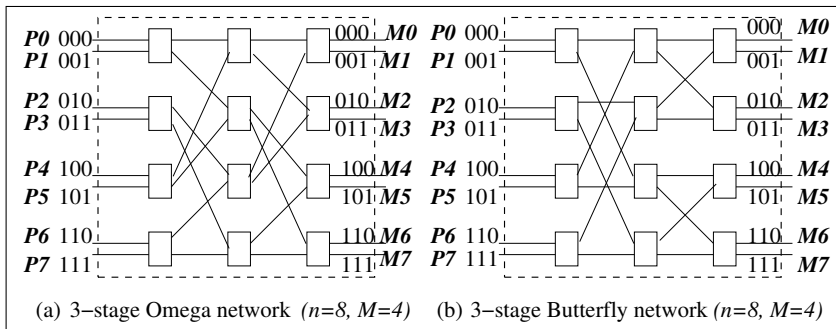


Figure 1.4: Interconnection networks for shared memory multiprocessor systems.  
 (a) Omega network (b) Butterfly network.



# Omega Network

- $n$  processors,  $n$  memory banks
- $\log n$  stages: with  $n/2$  switches of size  $2 \times 2$  in each stage
- Interconnection function: Output  $i$  of a stage connected to input  $j$  of next stage:

$$j = \begin{cases} 2i & \text{for } 0 \leq i \leq n/2 - 1 \\ 2i + 1 - n & \text{for } n/2 \leq i \leq n - 1 \end{cases}$$

- Routing function: in any stage  $s$  at any switch:  
to route to dest.  $j$ ,  
if  $s + 1$ th MSB of  $j = 0$  then route on upper wire  
else [ $s + 1$ th MSB of  $j = 1$ ] then route on lower wire

# Interconnection Topologies for Multiprocessors

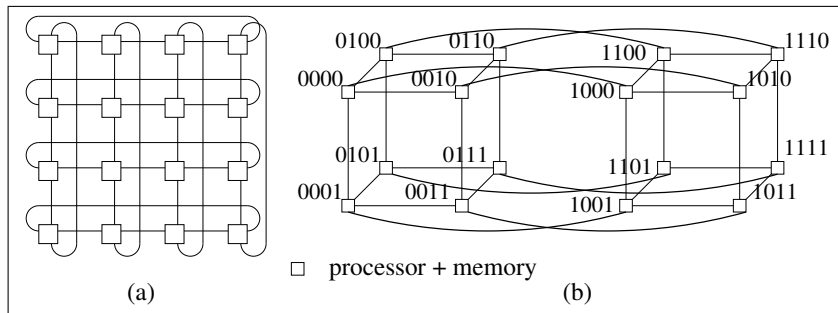


Figure 1.5: (a) 2-D Mesh with wraparound (a.k.a. torus) (b) 3-D hypercube

# Flynn's Taxonomy

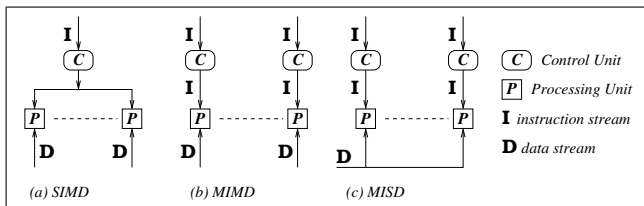


Figure 1.6: SIMD, MISD, and MIMD modes.

- **SISD: Single Instruction Stream Single Data Stream (traditional)**
- **SIMD: Single Instruction Stream Multiple Data Stream**
  - ▶ scientific applications, applications on large arrays
  - ▶ vector processors, systolic arrays, Pentium/SSE, DSP chips
- **MISD: Multiple Instruction Stream Single Data Stream**
  - ▶ E.g., visualization
- **MIMD: Multiple Instruction Stream Multiple Data Stream**
  - ▶ distributed systems, vast majority of parallel systems

# Terminology

- **Coupling**
  - ▶ Interdependency/binding among modules, whether hardware or software (e.g., OS, middleware)
- **Parallelism**:  $T(1)/T(n)$ .
  - ▶ Function of program and system
- **Concurrency of a program**
  - ▶ Measures productive CPU time vs. waiting for synchronization operations
- **Granularity of a program**
  - ▶ Amt. of computation vs. amt. of communication
  - ▶ Fine-grained program suited for tightly-coupled system

# Message-passing vs. Shared Memory

- **Emulating MP over SM:**




- ▶ Partition shared address space
- ▶ Send/Receive emulated by writing/reading from special mailbox per pair of processes

- **Emulating SM over MP:**



- ▶ Model each shared object as a process
- ▶ Write to shared object emulated by sending message to owner process for the object
- ▶ Read from shared object emulated by sending query to owner of shared object

# Classification of Primitives (1)

- **Synchronous (send/receive)** 
  - ▶ Handshake between sender and receiver
  - ▶ Send completes when Receive completes
  - ▶ Receive completes when data copied into buffer
- **Asynchronous (send)**
  - ▶ Control returns to process when data copied out of user-specified buffer



## Classification of Primitives (2)

- **Blocking (send/receive)**
  - ▶ Control returns to invoking process after processing of primitive (whether sync or async) completes
- **Nonblocking (send/receive)**
  - ▶ Control returns to process immediately after invocation
  - ▶ **Send:** even before data copied out of user buffer
  - ▶ **Receive:** even before data may have arrived from sender

# Non-blocking Primitive

```

Send(X, destination, handlek)           // handlek is a return parameter
...
...
Wait(handle1, handle2, ..., handlek, ..., handlem)    // Wait always blocks

```

Figure 1.7: A nonblocking *send* primitive. When the *Wait* call returns, at least one of its parameters is posted.

- Return parameter returns a system-generated handle
  - ▶ Use later to check for status of completion of call
  - ▶ Keep checking (loop or periodically) if handle has been posted
  - ▶ Issue Wait(*handle<sub>1</sub>*, *handle<sub>2</sub>*, ...) call with list of handles
  - ▶ Wait call blocks until one of the stipulated handles is posted



# Blocking/nonblocking; Synchronous/asynchronous; send/receive primitives

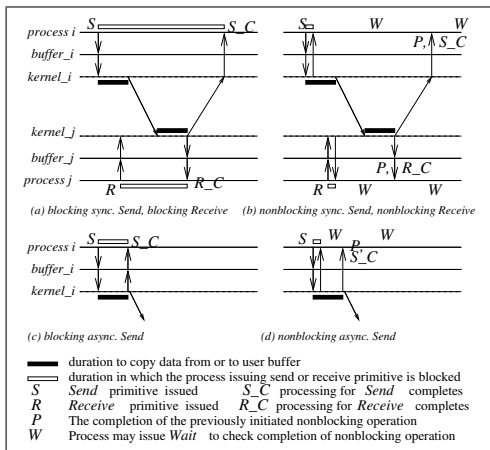


Figure 1.8: Illustration of 4 send and 2 receive primitives

# Asynchronous Executions; Message-passing System

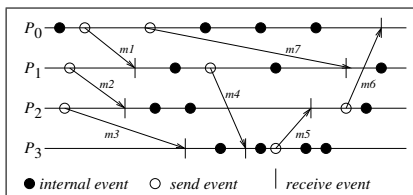


Figure 1.9: Asynchronous execution in a message-passing system

# Synchronous Executions: Message-passing System

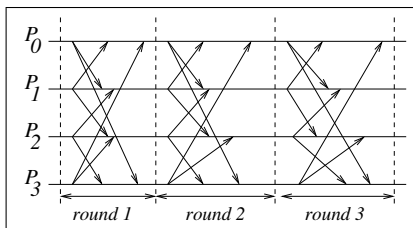


Figure 1.10: Synchronous execution in a message-passing system  
 In any round/step/phase:  $(send \mid internal)^*(receive \mid internal)^*$

- (1) *Sync\_Execution*(**int**  $k$ ,  $n$ ) //  $k$  rounds,  $n$  processes.
- (2) **for**  $r = 1$  **to**  $k$  **do**
- (3)     proc  $i$  sends msg to  $(i + 1) \bmod n$  and  $(i - 1) \bmod n$ ;
- (4)     each proc  $i$  receives msg from  $(i + 1) \bmod n$  and  $(i - 1) \bmod n$ ;
- (5)     compute app-specific function on received values.

# Synchronous vs. Asynchronous Executions (1)

- Sync vs async processors; Sync vs async primitives
- Sync vs async executions
- Async execution
  - ▶ No processor synchrony, no bound on drift rate of clocks
  - ▶ Message delays finite but unbounded
  - ▶ No bound on time for a step at a process
- Sync execution
  - ▶ Processors are synchronized; clock drift rate bounded
  - ▶ Message delivery occurs in one logical step/round
  - ▶ Known upper bound on time to execute a step at a process

## Synchronous vs. Asynchronous Executions (2)

- **Difficult to build a truly synchronous system;** can simulate this abstraction
- **Virtual synchrony:**
  - ▶ **async execution,** processes synchronize as per application requirement;
  - ▶ **execute in rounds/steps**
- **Emulations:**
  - ▶ **Async program** on sync system: trivial (A is special case of S)
  - ▶ **Sync program** on async system: tool called *synchronizer*

# System Emulations

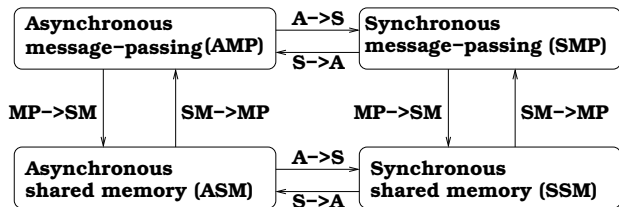


Figure 1.11: Sync  $\leftrightarrow$  async, and shared memory  $\leftrightarrow$  msg-passing emulations

- Assumption: failure-free system
- System A emulated by system B:
  - ▶ If not solvable in B, not solvable in A
  - ▶ If solvable in A, solvable in B