

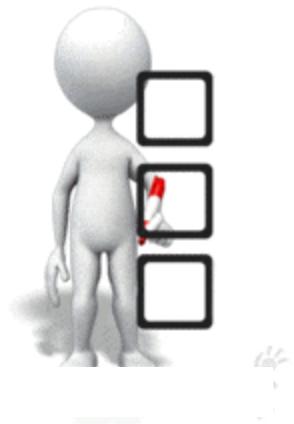
# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



# Outline

- Syllabus overview
- Unit -1 Overview
- Introduction to CUDA



# Syllabus overview

[Syllabus Link](#)

# Parallel Architecture

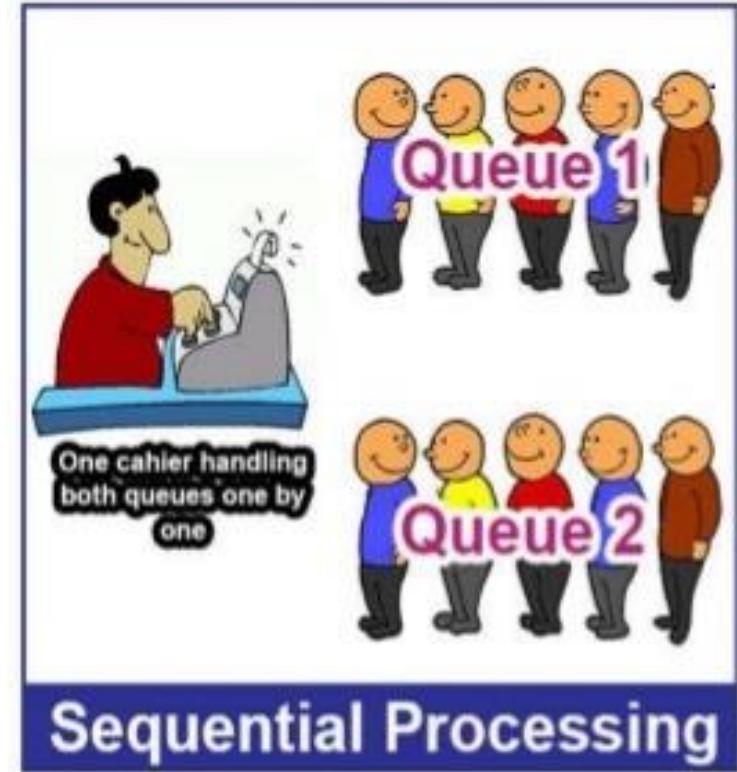


# Laptops/Desktops



# Sequential Processing

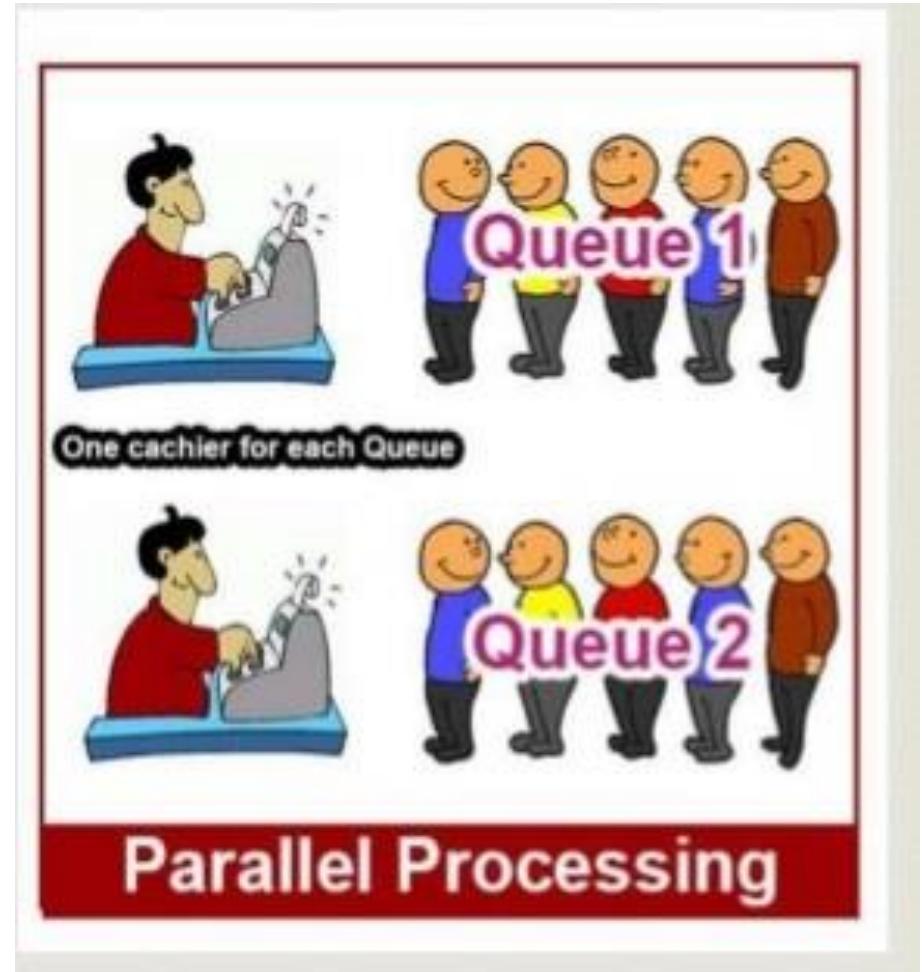
- **Serial computation**
- Problem is broken into a discrete series of instructions
- Instructions are executed **sequentially** one after another
- Executed on a **single processor**
- Only one instruction may execute at any moment in time



Source : <https://www.slideshare.net/BhavikVashi1/parallel-processing-simd-and-mimd>

# Parallel Processing

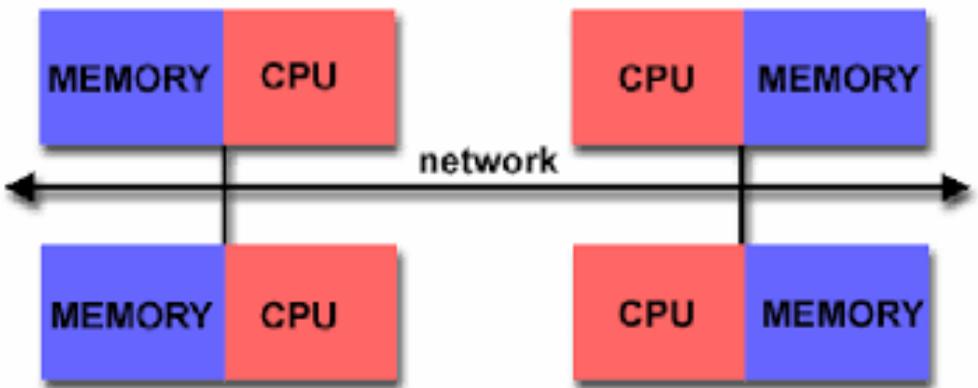
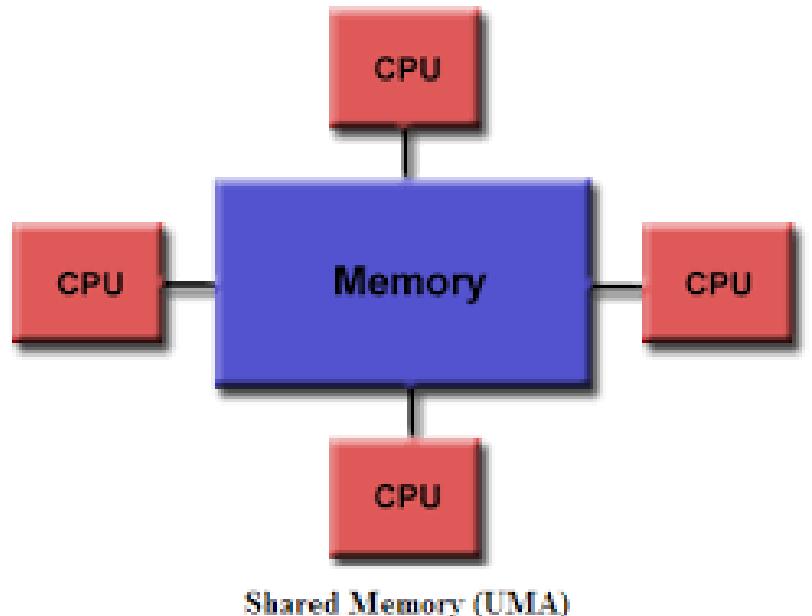
- **Parallel** computation
- Problem is broken into discrete parts that can be solved **concurrently**
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processor
- Overall control/coordination mechanism is employed



Source : <https://www.slideshare.net/BhavikVashi1/parallel-processing-simd-and-mimd>

# Parallel Processing

## Shared Memory

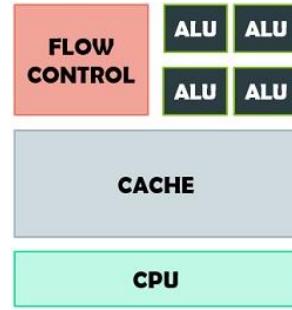


Source :<http://oskmadesimple.blogspot.com/2012/12/parallel-process-memory-architecture.html>

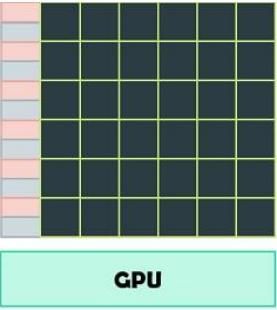
Sourceimg:[https://www.researchgate.net/figure/Distributed-memory-architecture\\_fig2\\_272377248](https://www.researchgate.net/figure/Distributed-memory-architecture_fig2_272377248)

# CPU verus GPU

**CPU**



**VS**



**GPU**

- Central Processing Unit
- Several cores
- Low latency
- Good for serial processing
- Can do a handful of operations at once
- Graphics Processing Unit
- Many cores
- High throughput
- Good for parallel processing
- Can do thousands of operations at once

Source img : <https://techdifferences.com/difference-between-cpu-and-gpu.html>

- Intel
- Nvidia
- AMD / ATI
- S3 Graphics
- Matrox
- Qualcomm
- Imagination Technologies
- Mali GPUs from ARM

- Designs graphics processing units (GPUs) for the gaming and professional markets, as well as system on a chip units
- Provides parallel processing capabilities to researchers and scientists that allow them to efficiently run high-performance applications
- Nvidia GPU models
  - GeForce
  - Tesla
  - Quadro
  - Jetson
  - Titan

# Unit -1 Overview

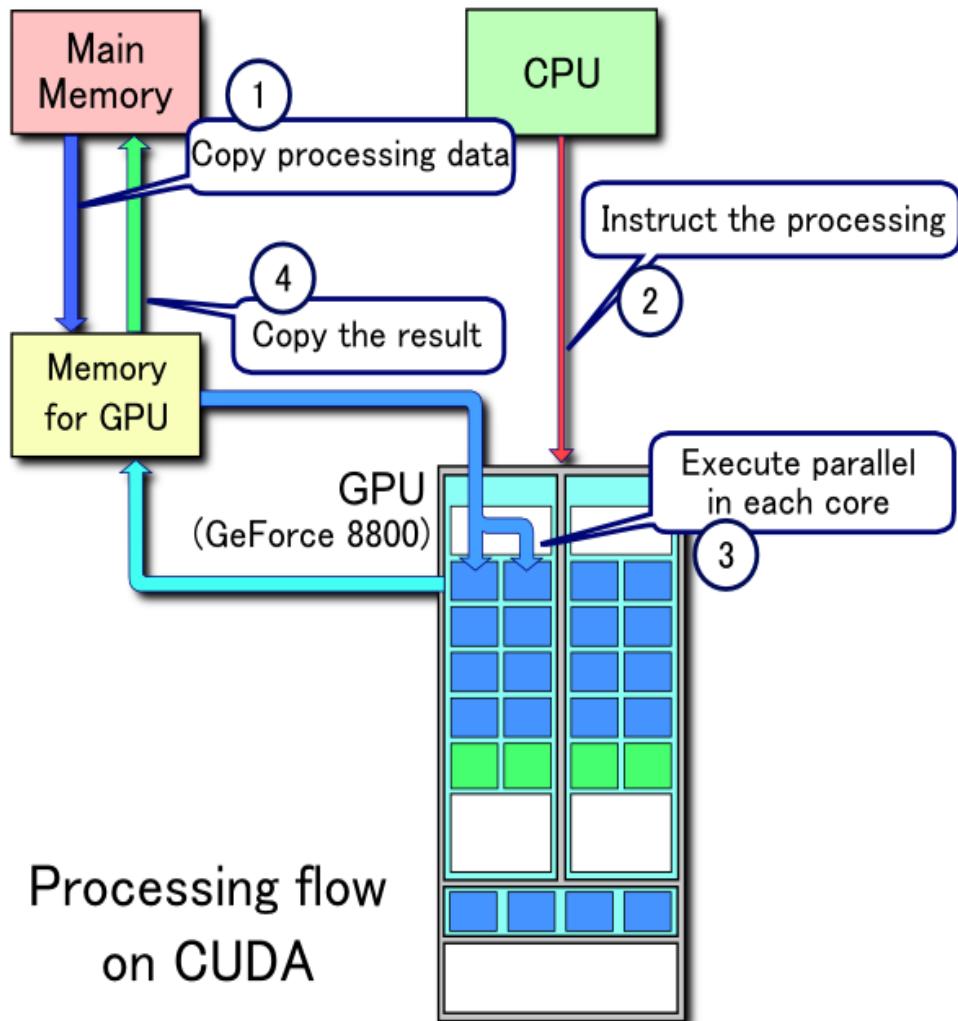
**UNIT - I****15 Periods**

**Heterogeneous Parallel Computing with CUDA:** Parallel Computing - Heterogeneous Computing - **CUDA Programming Model:** Timing Your Kernel - Organizing Parallel Threads - **Global Memory** - CUDA memory model - Memory Management - **Shared Memory and Constant Memory:** Shared Memory Allocation - Banks and Access Mode - Configuring the Amount of Shared Memory - Synchronization - Constant Memory - **Streams and Concurrency:** Introducing Streams and Events - **Tuning Instruction-level primitives:** CUDA Instructions

- Compute Unified Device Architecture
- Parallel computing platform and programming model developed by Nvidia for general computing on its own GPUs (graphics processing units)
- Enables developers to speed up compute-intensive

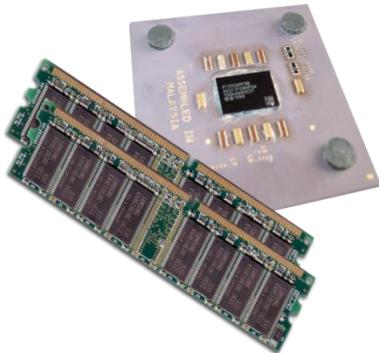
# Basic CUDA program structure

- Allocating memory space in device (GPU) for data
- Allocating memory space in host (CPU) for data
- Transferring data from host (CPU) to device (GPU)
- Declaring “kernel” routine to execute on device (GPU)
- Transferring data from device (GPU) to host (CPU)
- Free memory space in device (GPU)
- Free memory space in host (CPU)



Source : [https://www.researchgate.net/figure/CUDA-Process-workflow-diagram-1\\_fig6\\_300080119](https://www.researchgate.net/figure/CUDA-Process-workflow-diagram-1_fig6_300080119)

- **Terminology:**
  - **Host** The CPU and its memory (host memory)



- **Device** The GPU and its memory (device memory)



# CUDA

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int index = index + RADIUS;
    int tempIndex = index;

    // Read input elements into shared memory
    temp[index] = in[index];
    if (threadIdx.x == RADIUS) {
        temp[index - RADIUS] = in[index - RADIUS];
        temp[index + BLOCK_SIZE] = in[index + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[index + offset];

    // Store the result
    out[index] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<(NBLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

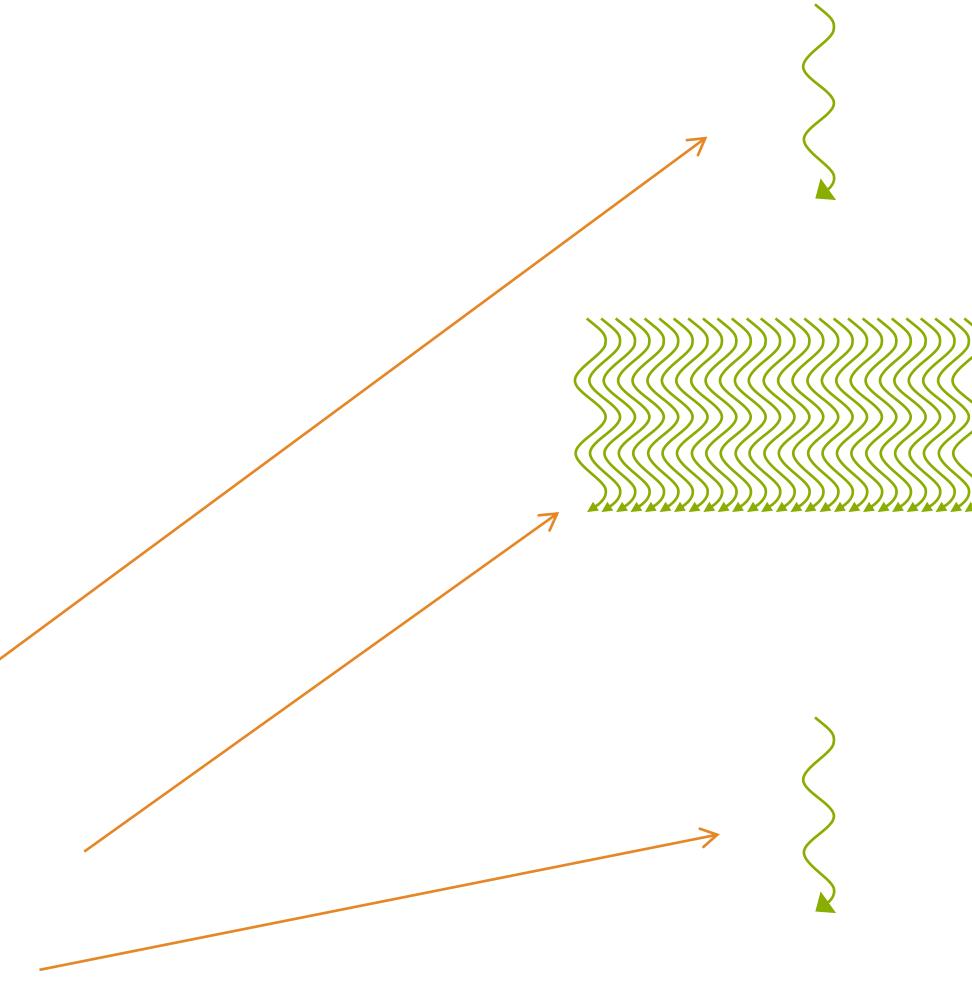
    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

serial code

parallel code  
serial code



```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc hello_world.cu  
$ a.out  
Hello World!  
$
```

```
__global__ void mykernel(void)
{
}

int main(void) {
    mykernel<<<1,1>>>0;
    printf("Hello World!\n");
    return 0;
}
```

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

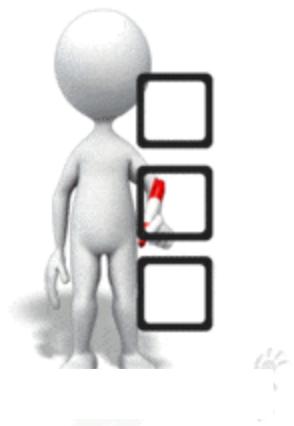
## Heterogeneous Parallel Computing with CUDA

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



# Outline

- Understanding heterogeneous computing architectures
- Recognizing the paradigm shift of parallel programming
- Grasping the basic elements of GPU programming
- Knowing the differences between CPU and GPU programming



# Introduction



- *High-Performance Computing* (HPC) - use of multiple processors or computers
  - to accomplish a complex task **concurrently** with high throughput and efficiency
- not only a computing architecture but also as a set of elements, including hardware systems, software tools, programming platforms, and parallel programming paradigms
- evolved significantly- emergence of **GPU-CPU heterogeneous architectures**, which have led to a fundamental paradigm shift in parallel programming.

# Parallel Computing



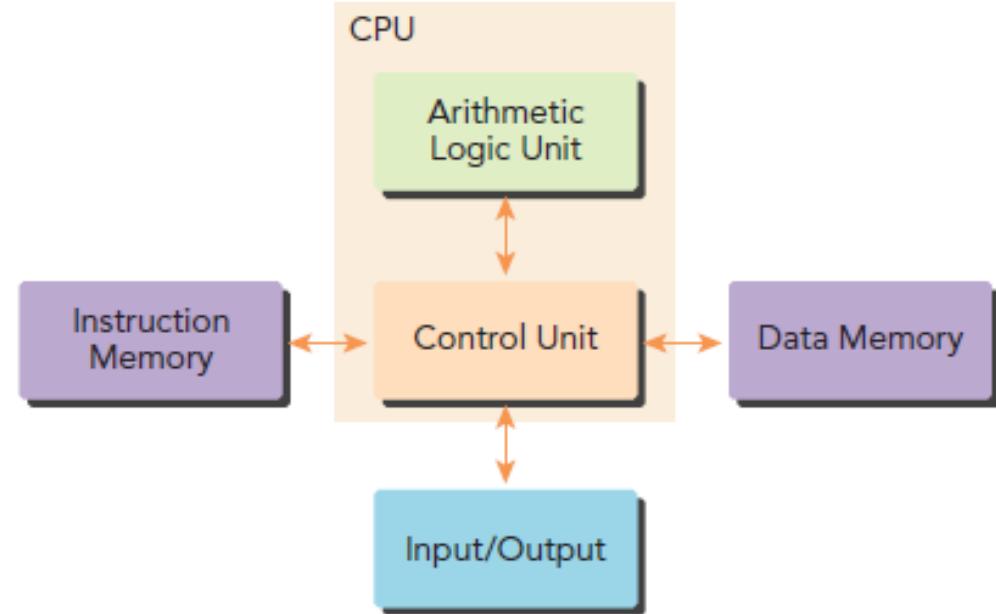
- Primary goal of parallel computing is to improve the **speed of computation**
- *Parallel computing* can be defined as a form of computation in which many calculations are carried out simultaneously
  - operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently
- How to map the concurrent calculations onto computers
- Multiple computing resources
- Parallel computing can then be defined as the simultaneous use of multiple computing resources (cores or computers) to perform the concurrent calculations
- A large problem is broken down into smaller ones, and each smaller one is then solved concurrently on different computing resources



- Parallel computing - two distinct areas of computing technologies:
  - Computer architecture (hardware aspect)
    - focuses on supporting parallelism at an architectural level
  - Parallel programming (software aspect)
    - focuses on solving a problem concurrently by fully using the computational power of the computer architecture
  - In order to achieve parallel execution in software, the hardware must provide a platform that supports concurrent execution of multiple processes or multiple threads

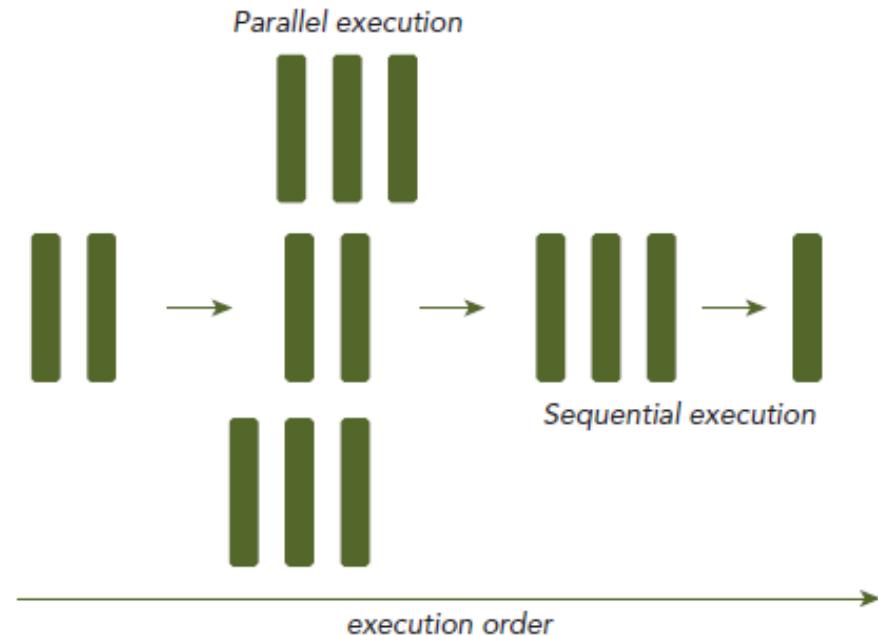
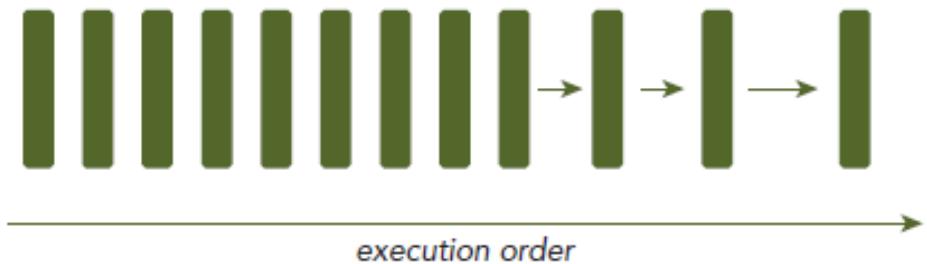
# Software and Hardware Aspects

- *Central Processing Unit (CPU)*, usually called the *core*.
  - ✓ only one core on a chip- *uniprocessor architecture*
  - ✓ integrate multiple cores onto a single processor- *multicore architecture*
  - support parallelism at the architecture level
- Therefore, programming can be viewed as the process of mapping the computation of a problem to available cores such that parallel execution is obtained.
- When implementing a sequential algorithm, you may not need to understand the details of the computer architecture to write a correct program.
- However, when implementing algorithms for multicore machines, it is much more important for programmers to be aware of the characteristics of the underlying computer architecture.
- Writing both correct and efficient parallel programs requires a fundamental knowledge of multicore architectures.



# Sequential and Parallel Programming

*The problem is divided into small pieces of calculations.*



# Sequential and Parallel Programming



- Program consists of two basic ingredients:
  - Instruction
  - Data
- When a computational problem is broken down into many small pieces of computation, each piece is called a task
- In a task, individual instructions
  - Consume inputs,
  - Apply a function
  - Produce outputs
- Data dependency occurs when an instruction consumes data produced by a preceding instruction-can classify the relationship between any two tasks as either dependent
  - One consumes the output of another, or independent Analyzing data dependencies is a fundamental skill in implementing parallel algorithms because dependencies are one of the primary inhibitors to parallelism
  - Understand to obtain application speedup in the modern programming world
  - Multiple independent chains of dependent tasks offer the best opportunity for parallelization

# Parallelism



- Fundamental types of parallelism in applications:
  - Task parallelism
  - ✓ Arises when there are many tasks or functions that can be operated independently and largely in parallel
  - ✓ Focuses on distributing functions across multiple cores
    - Data parallelism
    - ✓ Arises when there are many data items that can be operated on at the same time
    - ✓ Focuses on distributing the data across multiple cores
- CUDA programming is especially well-suited to address problems that can be expressed as data parallel computations
- Many applications that process large data sets can use a data-parallel model to speed up the computations
- Data-parallel processing maps data elements to parallel threads

# Partition

- First step in designing a data parallel program is to partition data across threads, with each thread working on a portion of the data
- Two approaches to partitioning data:
  - **block partitioning**
    - ✓ many consecutive elements of data are chunked together
    - ✓ Each chunk is assigned to a single thread in any order, and threads generally process only one chunk at a time
  - **cyclic partitioning**
    - ✓ fewer data elements are chunked together
    - ✓ Neighboring threads receive neighboring chunks, and each thread can handle more than one chunk.
    - ✓ Selecting a new chunk for a thread to process implies jumping ahead as many chunks as there are threads

# 1D and 2D Data Partition



*Block partition: each thread takes one data block*



*Cyclic partition: each thread takes two data blocks*



*Block partition on one dimension*



*Block partition on both dimensions*



*Cyclic partition on one dimension*

## DATA PARTITIONS

There are two basic approaches to partitioning data:

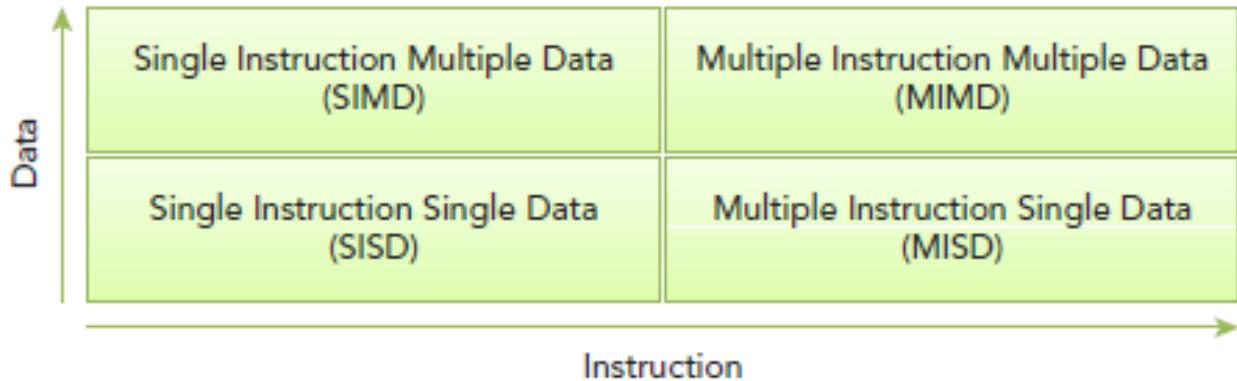
- Block: Each thread takes one portion of the data, usually an equal portion of the data.
- Cyclic: Each thread takes more than one portion of the data.

# Partition

- First step in designing a data parallel program is to partition data across threads, with each thread working on a portion of the data
- Two approaches to partitioning data:
  - **block partitioning**
    - ✓ many consecutive elements of data are chunked together
    - ✓ Each chunk is assigned to a single thread in any order, and threads generally process only one chunk at a time
  - **cyclic partitioning**
    - ✓ fewer data elements are chunked together
    - ✓ Neighboring threads receive neighboring chunks, and each thread can handle more than one chunk.
    - ✓ Selecting a new chunk for a thread to process implies jumping ahead as many chunks as there are threads

# Computer Architecture

- Flynn's Taxonomy—which classifies architectures into four different types according to how instructions and data flow through cores



# Computer Architecture



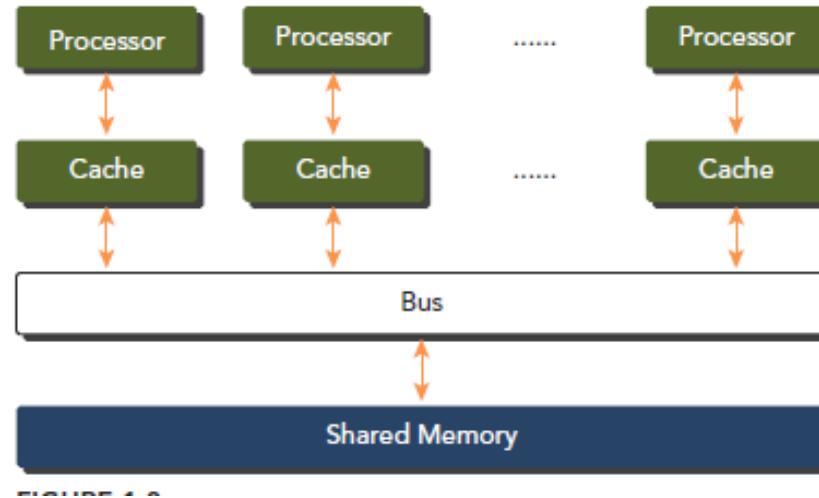
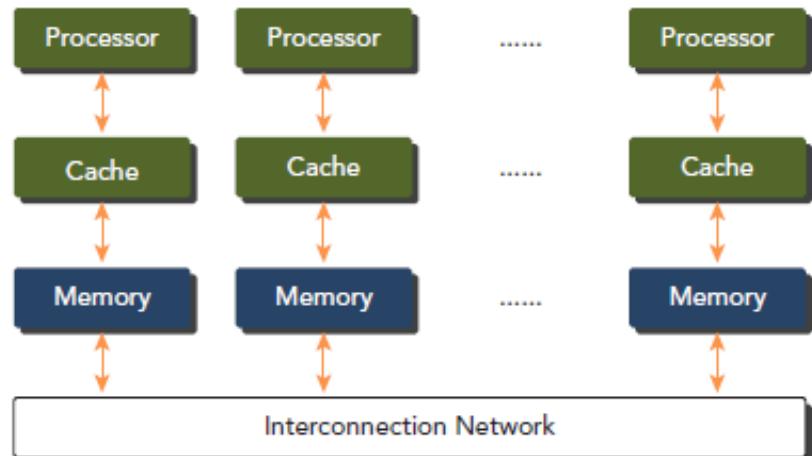
S.No	Architecture	Features
1	Single Instruction Single Data	<ul style="list-style-type: none"><li>✓ Traditional computer: a serial architecture</li><li>✓ Only one core in the computer</li><li>✓ At any time only one instruction stream is executed, and operations are performed on one data stream</li></ul>
2	Single Instruction Multiple Data	<ul style="list-style-type: none"><li>✓ Parallel architecture</li><li>✓ Multiple cores in the computer</li><li>✓ All cores execute the same instruction stream at any time, each operating on different data streams</li></ul>
3	Multiple Instruction Single Data	<ul style="list-style-type: none"><li>✓ Each core operates on the same data stream via separate instruction streams</li></ul>
4	Multiple Instruction Multiple Data	<ul style="list-style-type: none"><li>✓ Parallel architecture</li><li>✓ Multiple cores operate on multiple data streams, each executing independent instructions</li><li>✓ Many MIMD architectures also include SIMD execution sub-components</li></ul>

# Architectural Level Objectives

- At the architectural level, many advances have been made to achieve the following objectives:
  - ▶ Decrease latency
    - ✓ time it takes for an operation to start and complete, and is commonly expressed in microseconds
    - ▶ Increase bandwidth
      - ✓ amount of data that can be processed per unit of time, commonly expressed as megabytes/sec or gigabytes/sec
      - ▶ Increase throughput
        - ✓ amount of operations that can be processed per unit of time, commonly expressed as gflops (which stands for billion floating-point operations per second)
  - Latency measures the time to complete an operation, while throughput measures the number of operations processed in a given time unit

# Computer Architecture-Memory Organization

- Computer architectures can also be subdivided by their memory organization, which is generally classified into the following two types:
  - ▶ Multi-node with distributed memory
  - ▶ Multiprocessor with shared memory



## GPU CORE VERSUS CPU CORE

Even though many-core and multicore are used to label GPU and CPU architectures, a GPU core is quite different than a CPU core.

A CPU core, relatively heavy-weight, is designed for very complex control logic, seeking to optimize the execution of sequential programs.

A GPU core, relatively light-weight, is optimized for data-parallel tasks with simpler control logic, focusing on the throughput of parallel programs.

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## Heterogeneous Computing

**Dr. P. Padmakumari**

**CSE/SoC/SASTRA**



# Outline

- CPU designed to run general programming tasks
- GPU, originally designed to perform specialized graphics computations in parallel
  - More powerful and more generalized
  - Applied to general purpose parallel computing tasks with excellent performance
  - High power efficiency
- Typically, CPUs and GPUs are discrete processing components connected by the PCI-Express bus within a single compute node
- GPUs are referred to as **discrete devices**.
- **Homogeneous computing** uses one or more processor of the same architecture to execute an application.
- **Heterogeneous computing** instead uses a suite of processor architectures to execute an application, applying tasks to architectures to which they are well-suited, yielding performance improvement as a result

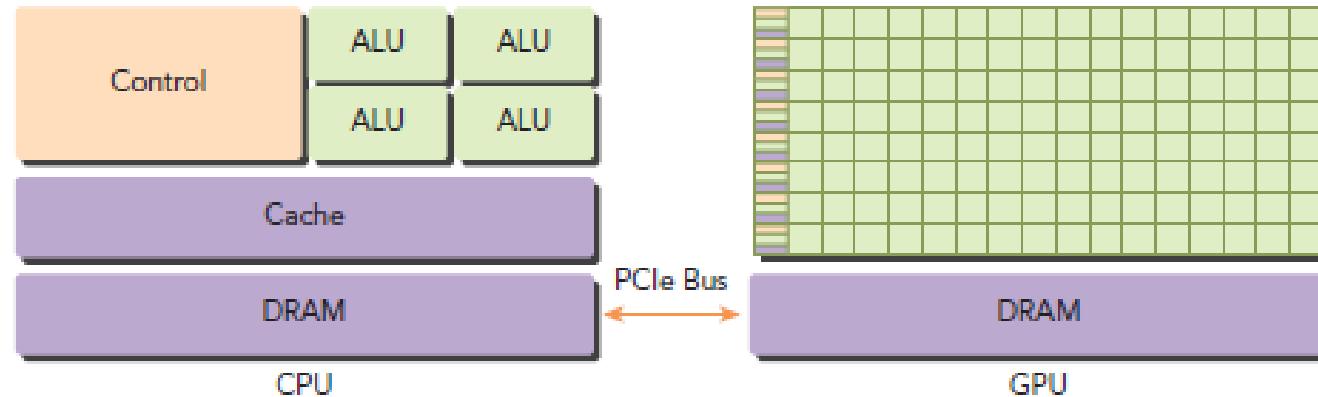
# Heterogeneous Systems



- Heterogeneous systems : advantages compared to traditional high performance computing systems
  - Effective use of systems is currently limited by the increased application design complexity
- While parallel programming has received much recent attention, the inclusion of heterogeneous resources adds complexity

# Heterogeneous Architecture

- Heterogeneous compute node nowadays consists of **two multicore CPU sockets and two or more many-core GPUs**
- A GPU is currently not a standalone platform but a **co-processor** to a CPU
- Therefore, GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus
- That is why, in GPU computing terms, the CPU is called the **host** and the GPU is called the **device**



# Heterogeneous application



- Heterogeneous application consists of two parts:

➤ Host code

- ✓ Runs on CPUs
- ✓ Application executing on a heterogeneous platform is typically initialized by the CPU
- ✓ CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks on the device

➤ Device code

- ✓ Runs on GPUs
- ✓ Program sections often exhibit a rich amount of data parallelism, GPUs are used to accelerate the execution of this portion of data parallelism

- Hardware component physically separate from the CPU is used to accelerate computationally intensive sections of an application, it is referred to as a **hardware accelerator**
- **GPUs** are arguably the most common example of a hardware accelerator
- NVIDIA's GPU computing platform is enabled on the following product families:
  - ▶ **Tegra** - designed for mobile and embedded devices such as tablets and phones
  - ▶ **GeForce** - for consumer graphics
  - ▶ **Quadro**- for professional visualization
  - ▶ **Tesla**- for datacenter parallel computing
    - **Fermi**- the GPU accelerator in the Tesla product family, has recently gained wide spread use as a computing accelerator for high-performance computing applications
    - **Fermi** - released by NVIDIA in 2010, is the world's first complete GPU computing architecture
    - **Areas** - seismic processing, biochemistry simulations, weather and climate modeling, signal processing, computational finance, computer-aided engineering, computational fluid dynamics, and data analysis

- Kepler, the current generation of GPU computing architecture after Fermi, released in the fall of 2012, offers much higher processing power than the prior GPU generation and provides new methods to optimize and increase parallel workload execution on the GPU, expecting to further revolutionize high-performance computing
- Tegra K1 contains a Kepler GPU and provides everything you need to unlock the power of the GPU for embedded applications

# GPU capability



- There are two important features that describe GPU capability:
  - ▶ Number of CUDA cores
  - ▶ Memory size
- Accordingly, there are two different metrics for describing GPU performance:
  - ▶ **Peak computational performance** - measure of computational capability, usually defined as how many single-precision or double-precision floating point calculations can be processed per second
    - ✓ expressed in gflops (billion floating-point operations per second) or tflops (trillion floating-point calculations per second)
  - ▶ **Memory bandwidth**- measure of the ratio at which data can be read from or stored to memory.
    - ✓ expressed in gigabytes per second, GB/s

# Fermi and Kepler



	<b>FERMI</b> (TESLA C2050)	<b>KEPLER</b> (TESLA K10)
CUDA Cores	448	2 x 1536
Memory	6 GB	8 GB
Peak Performance*	1.03 Tflops	4.58 Tflops
Memory Bandwidth	144 GB/s	320 GB/s

\* Peak single-precision floating point performance

# Compute Capabilities



## COMPUTE CAPABILITIES

NVIDIA uses a special term, *compute capability*, to describe hardware versions of GPU accelerators that belong to the entire Tesla product family. The version of Tesla products is given in Table 1-2.

Devices with the same major revision number are of the same core architecture.

- Kepler class architecture is major version number 3.
- Fermi class architecture is major version number 2.
- Tesla class architecture is major version number 1.

The first class of GPUs delivered by NVIDIA contains the same Tesla name as the entire family of Tesla GPU accelerators.

Compute Capabilities of Tesla GPU Computing Products

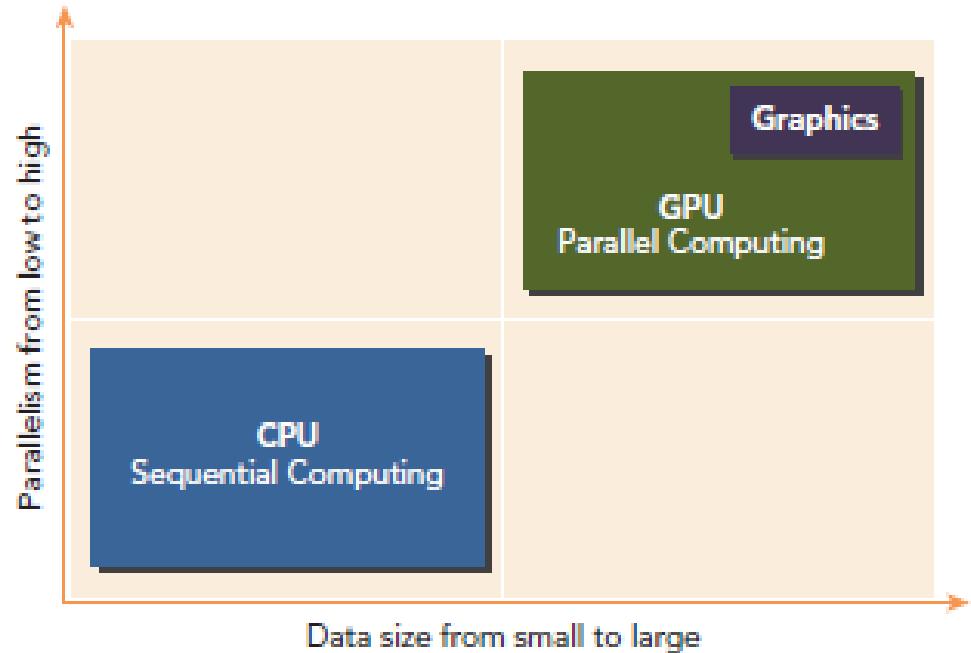
GPU	COMPUTE CAPABILITY
Tesla K40	3.5
Tesla K20	3.5
Tesla K10	3.0
Tesla C2070	2.0
Tesla C1060	1.3

# Paradigm of Heterogeneous Computing

- GPU computing is not meant to replace CPU computing
- Each approach has advantages for certain kinds of programs
- CPU computing is good for control-intensive tasks
- GPU computing is good for data-parallel computation-intensive tasks
- When CPUs are complemented by GPUs, it makes for a powerful combination
- The CPU is optimized for dynamic workloads marked by short sequences of computational operations and unpredictable control flow
- GPUs aim at the other end of the spectrum: workloads that are dominated by computational tasks with simple control flow

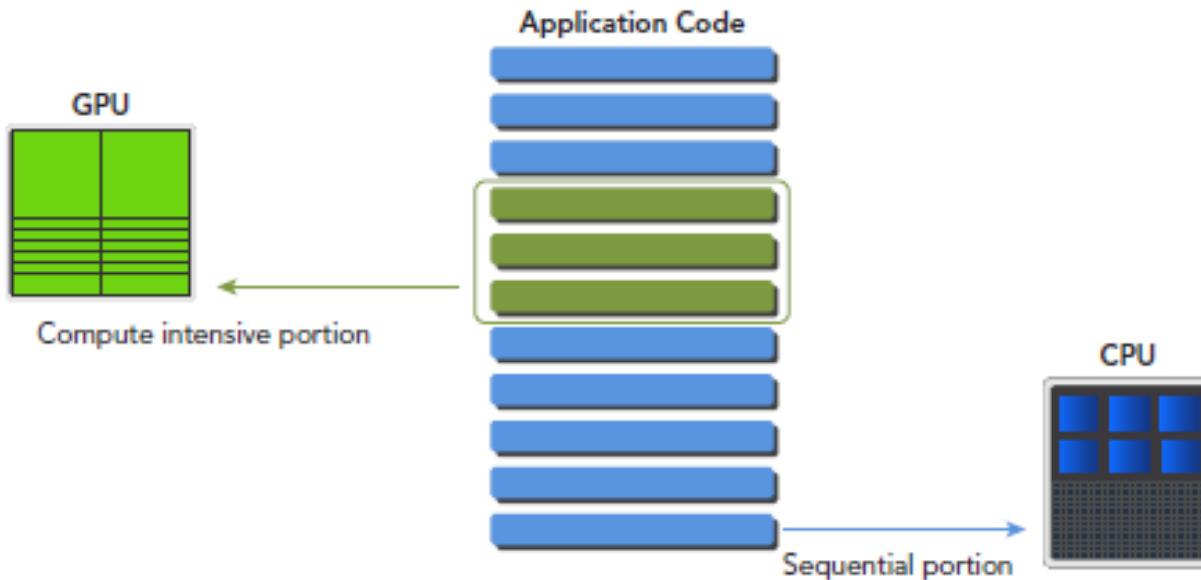
# Scope of Applications for CPU and GPU

- Two dimensions that differentiate the scope of applications for CPU and GPU:
  - Parallelism level
  - Data size
- If a problem has a small data size, sophisticated control logic, and/or low-level parallelism, the CPU is a good choice because of its ability to handle complex logic and instruction-level parallelism
- If the problem at hand instead processes a huge amount of data and exhibits massive data parallelism, the GPU is the right choice because it has a large number of programmable cores, can support massive multi-threading, and has a larger peak bandwidth compared to the CPU



# CPU + GPU heterogeneous parallel computing

- CPU + GPU heterogeneous parallel computing architectures evolved because the CPU and GPU have complementary attributes that enable applications to perform best using both types of processors
- Optimal performance you may need to use both CPU and GPU for your application, executing the sequential parts or task parallel parts on the CPU and intensive data parallel parts on the GPU



## CPU THREAD VERSUS GPU THREAD

Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.

Threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.

CPU cores are designed to minimize latency for one or two threads at a time, whereas GPU cores are designed to handle a large number of concurrent, lightweight threads in order to maximize throughput.

Today, a CPU with four quad core processors can run only 16 threads concurrently, or 32 if the CPUs support hyper-threading.

Modern NVIDIA GPUs can support up to 1,536 active threads concurrently per multiprocessor. On GPUs with 16 multiprocessors, this leads to more than 24,000 concurrently active threads.

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **CUDA: A Platform for Heterogeneous Computing**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



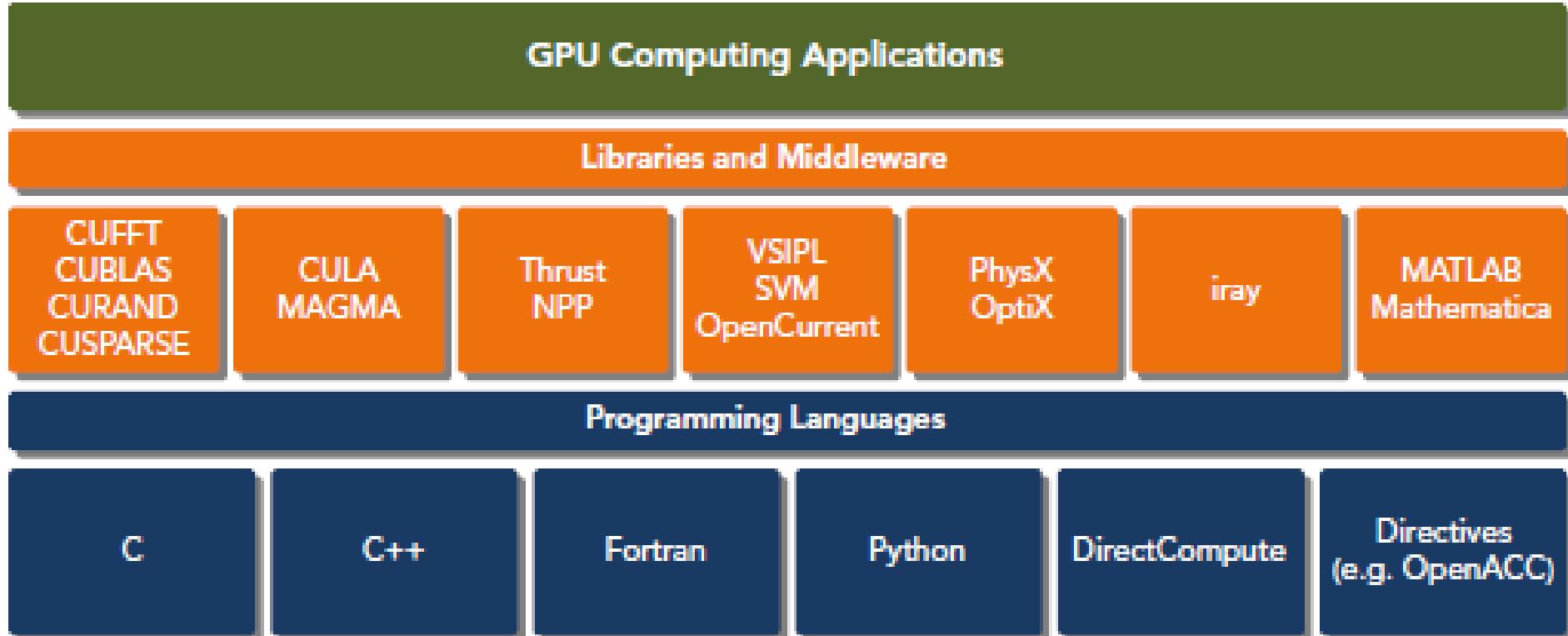
PresenterMedia

# Outline



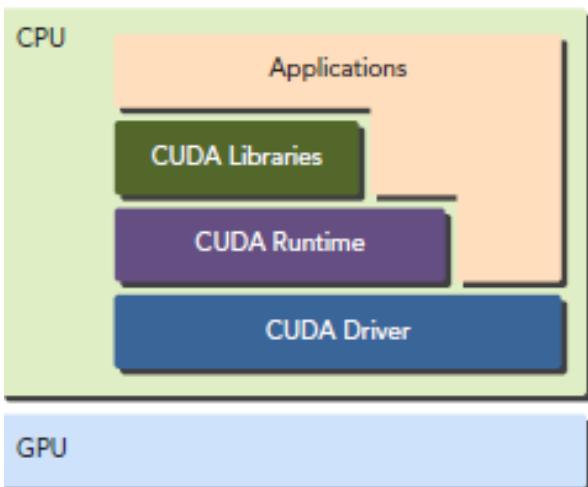
- CUDA: A Platform for Heterogeneous Computing CUDA
- General-purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs
- Solve many complex computational problems in a more efficient way
- CUDA platform is accessible through
  - CUDA-accelerated libraries
  - Compiler directives
  - Application programming interface
  - Extensions to industry-standard programming languages
  - C, C++, Fortran, and Python

# CUDA platform



- CUDA C is an extension of standard ANSI C - straightforward APIs to manage devices, memory, and other tasks
- CUDA is also a scalable programming model that enables programs to transparently scale their parallelism to GPUs with varying numbers of cores

- CUDA provides two API levels for managing the GPU device and organizing threads
  - CUDA Driver API
  - CUDA Runtime API
- Driver API is a low-level API and is relatively hard to program, but it provides more control over how the GPU device is used
- Runtime API is a higher-level API implemented on top of the driver API
- Each function of the runtime API is broken down into more basic operations issued to the driver API



## RUNTIME API VERSUS DRIVER API

There is no noticeable performance difference between the runtime and driver APIs. How your kernels use memory and how you organize your threads on the device have a much more pronounced effect.

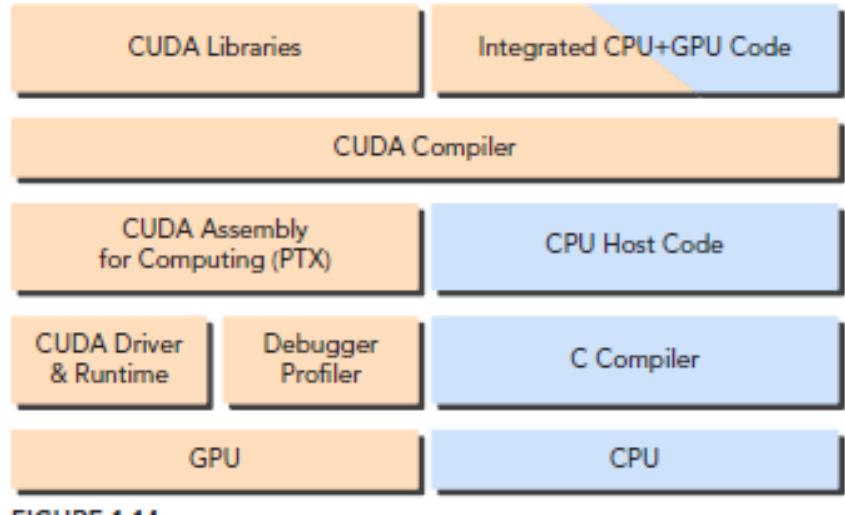
These two APIs are mutually exclusive. You must use one or the other, but it is not possible to mix function calls from both. All examples throughout this book use the runtime API.

A CUDA program consists of a mixture of the following two parts:

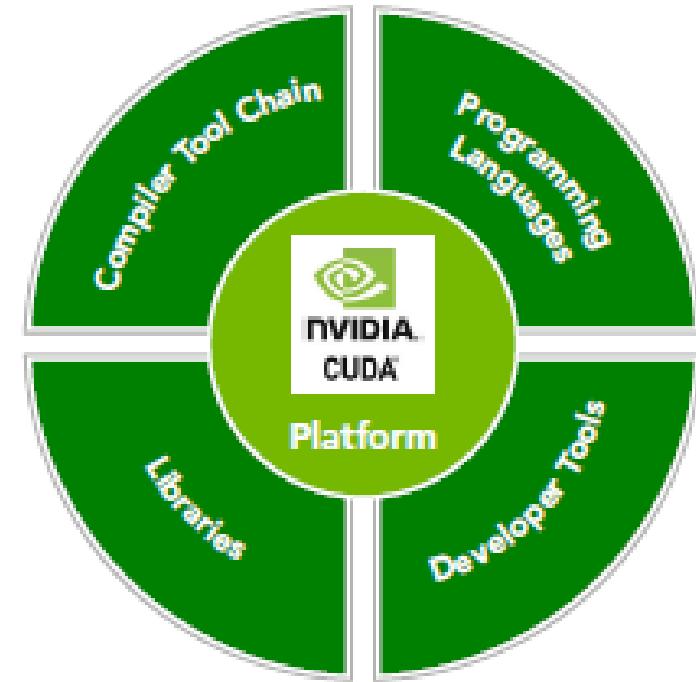
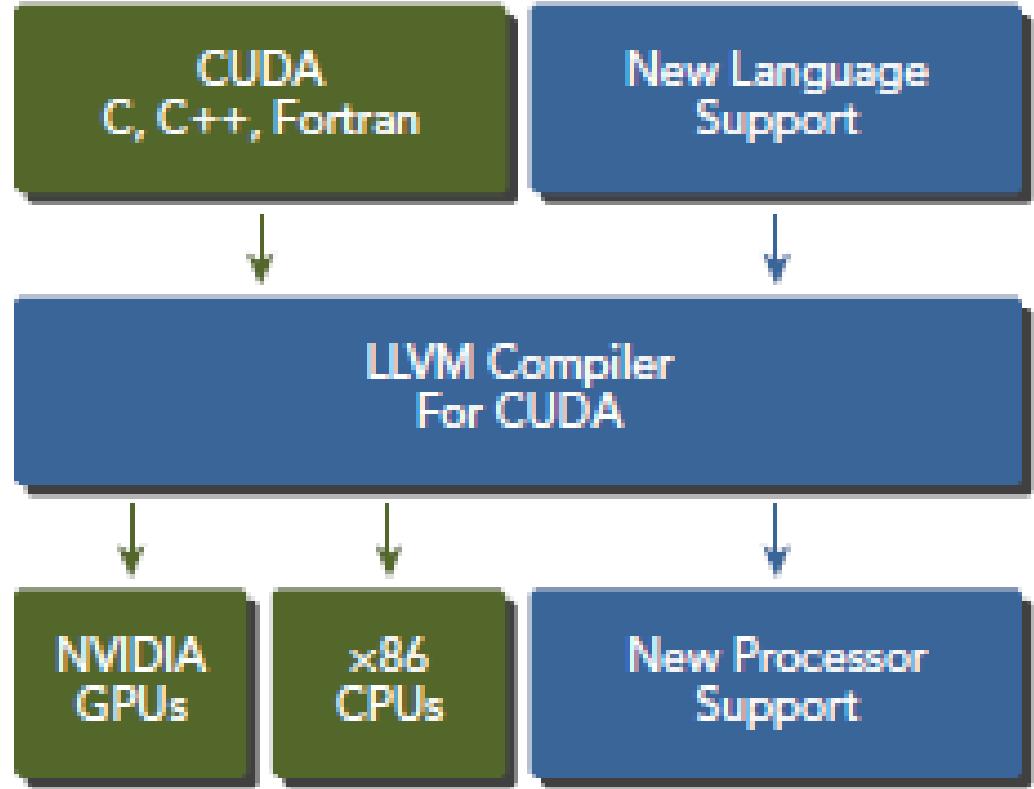
- The host code runs on CPU.
- The device code runs on GPU

# CUDA compiler

- NVIDIA's CUDA **nvcc compiler** separates the device code from the host code during the compilation process
- Host code is standard C code and is further compiled with C compilers
- Device code is written using CUDA C extended with keywords for labeling data-parallel functions, called kernels
- Device code is further compiled by nvcc
- During the link stage, CUDA runtime libraries are added for kernel procedure calls and explicit GPU device manipulation
- CUDA nvcc compiler is based on the widely used LLVM open source compiler infrastructure.
- Create or extend programming languages with support for GPU acceleration using the CUDA Compiler SDK



# CUDA SDK



- To write a CUDA C program
  1. Create a source code file with the special file name extension of cu.
  2. Compile the program using the CUDA nvcc compiler.
  3. Run the executable file from the command line, which contains the kernel code executable on the GPU

## CUDA PROGRAM STRUCTURE

A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA kernel to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

# IS CUDA C PROGRAMMING DIFFICULT?



- main difference between CPU programming and GPU programming is the level of programmer exposure to GPU architectural features
- basic knowledge of CPU architectures
- Locality is a very important concept in parallel programming
- Locality refers to the reuse of data so as to reduce memory access latency
- Two basic types of reference locality
  - Temporal locality refers to the reuse of data and/or resources within relatively small time durations
  - Spatial locality refers to the use of data elements within relatively close storage locations
- Modern CPU architectures use large caches to optimize for applications with good spatial and temporal locality

- To write a CUDA C program
  1. Create a source code file with the special file name extension of cu.
  2. Compile the program using the CUDA nvcc compiler.
  3. Run the executable file from the command line, which contains the kernel code executable on the GPU

## CUDA PROGRAM STRUCTURE

A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA kernel to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## CUDA Programming Model

**Dr. P. Padmakumari**

**CSE/SoC/SASTRA**



PresenterMedia

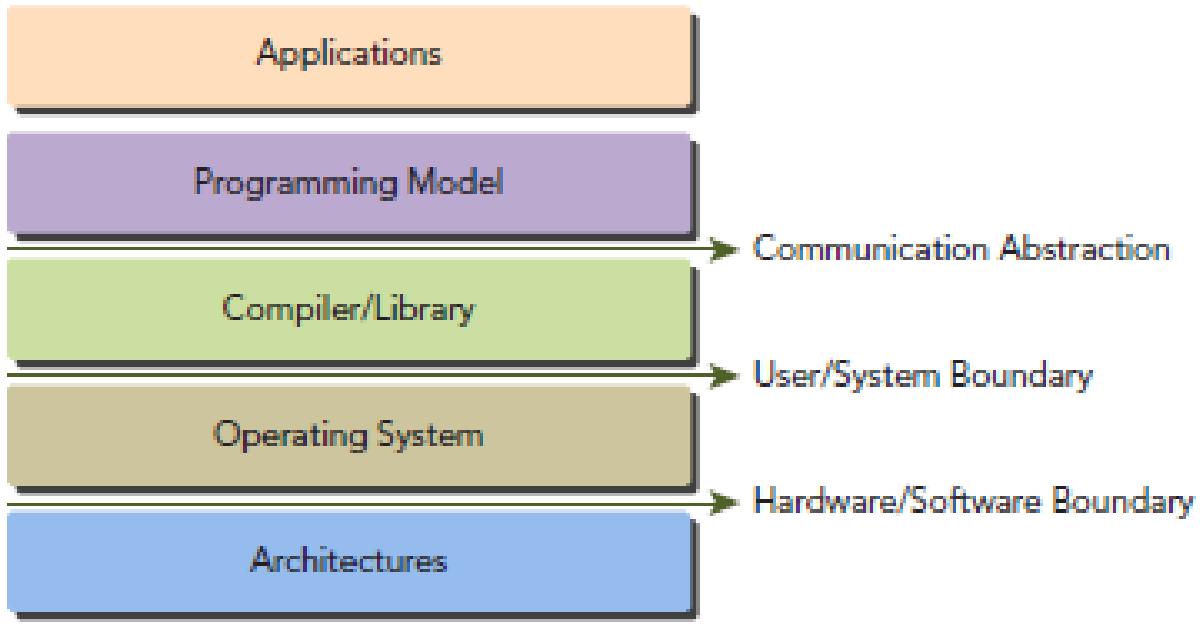
# Outline



- Writing a CUDA program
- Executing a kernel function
- Organizing threads with grids and blocks
- Measuring GPU performance

# Introducing The CUDA Programming Model

- Programming models present an abstraction of computer architectures that act as a bridge between an application and its implementation on available hardware



# CUDA Programming Model

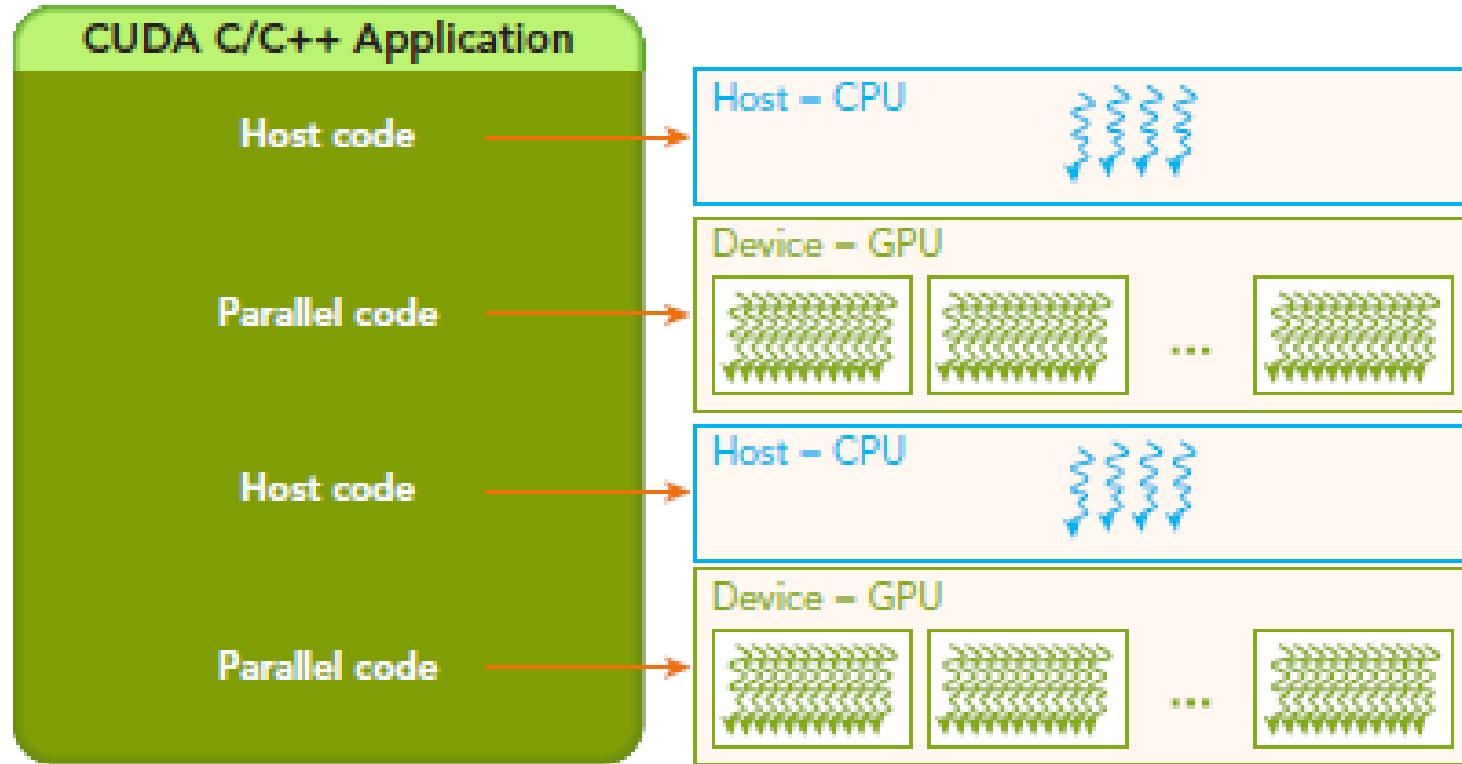


- CUDA programming model provides the following special features to harness the computing power of GPU architectures
  - A way to organize threads on the GPU through a hierarchy structure
  - A way to access memory on the GPU through a hierarchy structure
- From the perspective of a programmer view parallel computation from different levels, such as:
  - Domain level
  - Logic level
  - Hardware level

# CUDA Programming Structure

- A heterogeneous environment consists of CPUs complemented by GPUs, each with its own memory separated by a PCI-Express bus
- ► Host: the CPU and its memory (host memory)
- ► Device: the GPU and its memory (device memory)
- NVIDIA introduced a programming model improvement called **Unified Memory**, which bridges the divide between host and device memory spaces
- Allows to access both the CPU and GPU memory using a single pointer, while the system automatically migrates the data between the host and device
- A key component of the CUDA programming model is the **kernel** — the code that runs on the GPU device

# GPU kernel execution



# Managing Memory



## Host and Device Memory Functions

### STANDARD C FUNCTIONS

`malloc`

`memcpy`

`memset`

`free`

### CUDA C FUNCTIONS

`cudaMalloc`

`cudaMemcpy`

`cudaMemset`

`cudaFree`

- The function used to perform GPU memory allocation is **cudaMalloc**, and its function signature is:

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

- This function allocates a linear range of device memory with the specified size in bytes
- The allocated memory is returned through devPtr
- Malloc and the standard C runtime library malloc

# Compute Capabilities



- Function used to transfer data between the host and device is: **cudaMemcpy**, and its function signature is:

**cudaError\_t cudaMemcpy ( void\* dst, const void\* src, size\_t count, cudaMemcpyKind kind )**

- Function copies the specified bytes from the source memory area, pointed to by src, to the destination memory area, pointed to by dst, with the direction specified by kind, where kind takes one of the following types:

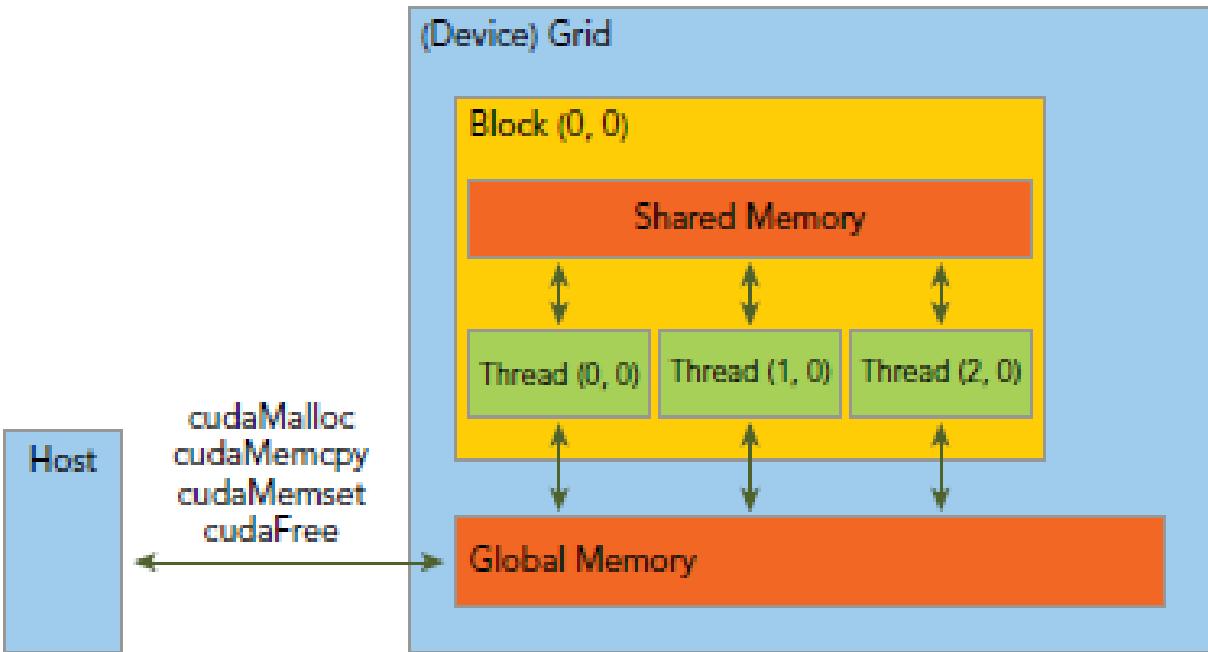
- **cudaMemcpyHostToHost**
- **cudaMemcpyHostToDevice**
- **cudaMemcpyDeviceToHost**
- **cudaMemcpyDeviceToDevice**

Function exhibits synchronous behavior because the host application blocks until cudaMemcpy returns and the transfer is complete

Every CUDA call, except kernel launches, returns an error code of an enumerated type **cudaError\_t**

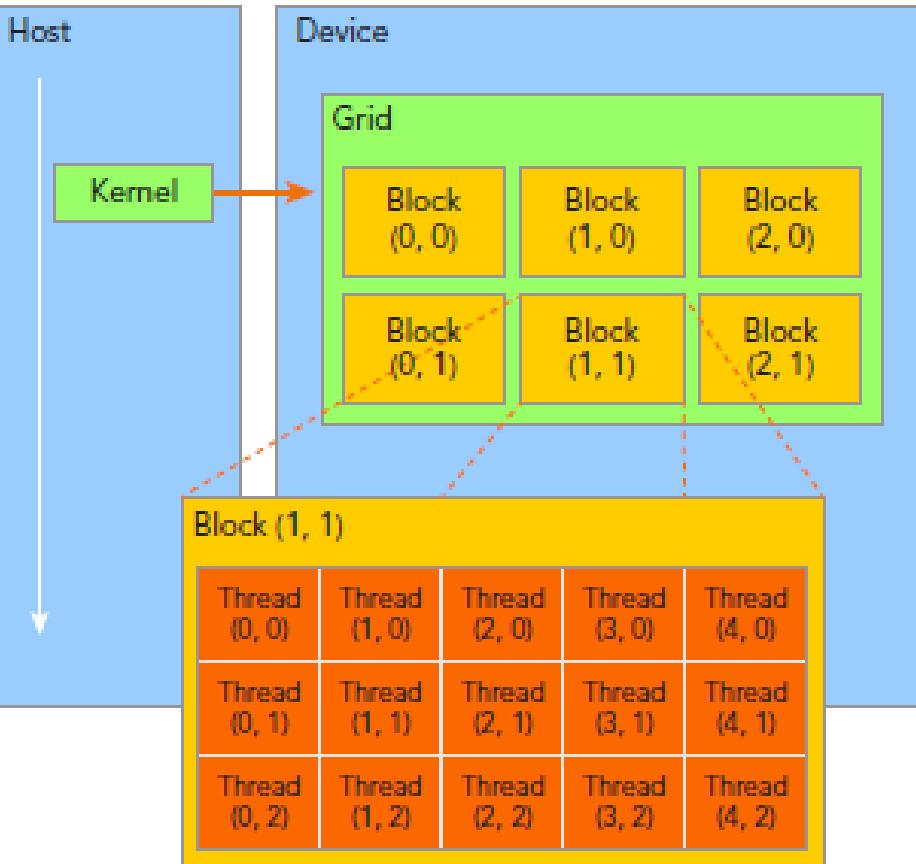
# Paradigm of Heterogeneous Computing

- Simplified GPU memory structure, containing two major ingredients: global memory and shared memory



# Organizing Threads

- CUDA exposes a thread hierarchy abstraction to enable you to organize your threads
- Two-level thread hierarchy decomposed into
  - ✓ Blocks of threads
  - ✓ Grids of blocks
- All threads spawned by a single kernel launch are collectively called a grid.
- All threads in a grid share the same global memory space. A grid is made up of many thread blocks
- A thread block is a group of threads that can cooperate with each other using:
  - ✓ Block-local synchronization
  - ✓ Block-local shared memory



# Organizing Threads



- Threads from different blocks cannot cooperate
- Threads rely on the following two unique coordinates to distinguish themselves from each other:
  - ✓ BlockIdx (block index within a grid)
  - ✓ ThreadIdx (thread index within a block)
- variables appear as built-in, pre-initialized variables that can be accessed within kernel functions.
- When a kernel function is executed, the coordinate variables blockIdx and threadIdx are assigned to each thread by the CUDA runtime
- Based on the coordinates, y assign portions of data to different threads
- The coordinate variable is of type uint3, a CUDA built-in vector type, derived from the basic integer type
- It is a structure containing three unsigned integers, and the 1st, 2nd, and 3rd components are accessible through the fields x, y, and z respectively
- blockIdx.x, blockIdx.y ,blockIdx.z, threadIdx.x ,threadIdx.y, threadIdx.z

# Organizing Threads



- CUDA organizes grids and blocks in three dimensions
- Dimensions of a grid and a block are specified by the following two built-in variables:
  - `blockDim` (block dimension, measured in threads)
  - `gridDim` (grid dimension, measured in blocks)
- Variables are of type `dim3`, an integer vector type based on `uint3` that is used to specify dimensions
- When defining a variable of type `dim3`, any component left unspecified is initialized to 1
- Each component in a variable of type `dim3` is accessible through its `x`, `y`, and `z` fields, respectively, as shown in the following example:
  - ✓ `blockDim.x`
  - ✓ `blockDim.y`
  - ✓ `blockDim.z`

## GRID AND BLOCK DIMENSIONS

Usually, a grid is organized as a 2D array of blocks, and a block is organized as a 3D array of threads.

Both grids and blocks use the `dim3` type with three unsigned integer fields. The unused fields will be initialized to 1 and ignored.

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## CUDA Programming Model

**Dr. P. Padmakumari**

**CSE/SoC/SASTRA**



PresenterMedia

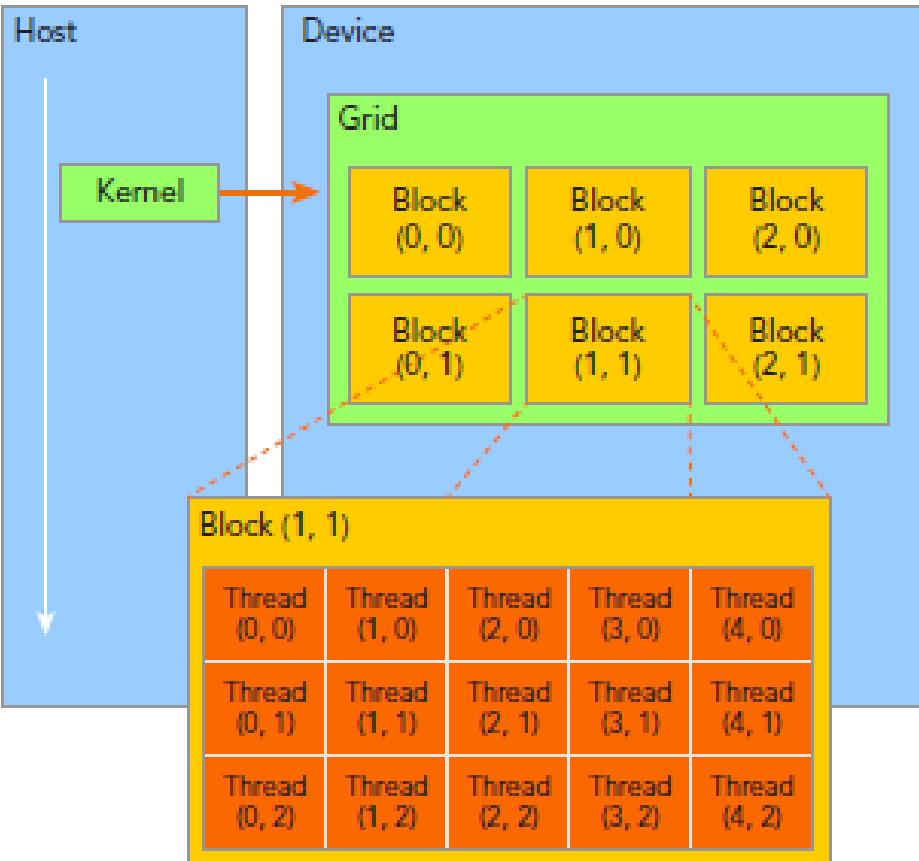
# Outline



- Writing a CUDA program
- Executing a kernel function
- Organizing threads with grids and blocks
- Measuring GPU performance

# Organizing Threads

- CUDA exposes a thread hierarchy abstraction to enable you to organize your threads
- Two-level thread hierarchy decomposed into
  - ✓ Blocks of threads
  - ✓ Grids of blocks
- All threads spawned by a single kernel launch are collectively called a grid.
- All threads in a grid share the same global memory space. A grid is made up of many thread blocks
- A thread block is a group of threads that can cooperate with each other using:
  - ✓ Block-local synchronization
  - ✓ Block-local shared memory



# Organizing Threads



- Threads from different blocks cannot cooperate
- Threads rely on the following two unique coordinates to distinguish themselves from each other:
  - ✓ BlockIdx (block index within a grid)
  - ✓ ThreadIdx (thread index within a block)
- variables appear as built-in, pre-initialized variables that can be accessed within kernel functions.
- When a kernel function is executed, the coordinate variables blockIdx and threadIdx are assigned to each thread by the CUDA runtime
- Based on the coordinates, y assign portions of data to different threads
- The coordinate variable is of type uint3, a CUDA built-in vector type, derived from the basic integer type
- It is a structure containing three unsigned integers, and the 1st, 2nd, and 3rd components are accessible through the fields x, y, and z respectively
- blockIdx.x, blockIdx.y ,blockIdx.z, threadIdx.x ,threadIdx.y, threadIdx.z

# Organizing Threads



- CUDA organizes grids and blocks in three dimensions
- Dimensions of a grid and a block are specified by the following two built-in variables:
  - `blockDim` (block dimension, measured in threads)
  - `gridDim` (grid dimension, measured in blocks)
- Variables are of type `dim3`, an integer vector type based on `uint3` that is used to specify dimensions
- When defining a variable of type `dim3`, any component left unspecified is initialized to 1
- Each component in a variable of type `dim3` is accessible through its `x`, `y`, and `z` fields, respectively, as shown in the following example:
  - ✓ `blockDim.x`
  - ✓ `blockDim.y`
  - ✓ `blockDim.z`

- Grid is organized as a **2D array of blocks**, and a block is organized as a **3D array of threads**
- Both grids and blocks use the **dim3** type with three unsigned integer fields. Unused fields will be initialized to 1 and ignored

Check grid and block indices and dimensions

```
grid.x 2 grid.y 1 grid.z 1 block.x 3 block.y 1 block.z 1
threadIdx:(0, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

# Organizing Threads



## ACCESS GRID/BLOCK VARIABLES FROM THE HOST AND DEVICE SIDE

It is important to distinguish between the host and device access of grid and block variables. For example, using a variable declared as block from the host, you define the coordinates and access them as follows:

`block.x, block.y, and block.z`

On the device side, you have pre-initialized, built-in block size variable available as:

`blockDim.x, blockDim.y, and blockDim.z`

In summary, you define variables for grid and block on the host before launching a kernel, and access them there with the `x, y` and `z` fields of the vector structure from the host side. When the kernel is launched, you can use the pre-initialized, built-in variables within the kernel.

Define grid and block dimensions on the host

`grid.x 1 block.x 1024`  
`grid.x 2 block.x 512`  
`grid.x 4 block.x 256`  
`grid.x 8 block.x 128`

## THREAD HIERARCHY

One of CUDA's distinguishing features is that it exposes a two-level thread hierarchy through the programming model. Because the grid and block dimensionality of a kernel launch affect performance, exposing this simple abstraction provides the programmer with an additional avenue for optimization.

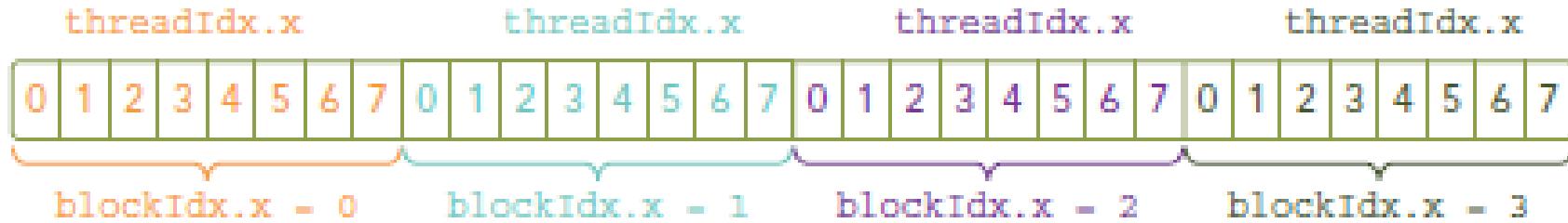
There are several restrictions on the dimensions of grids and blocks. One of the major limiting factors on block size is available compute resources, such as registers, shared memory, and so on. Some limits can be retrieved by querying the GPU device.

Grids and blocks represent a logical view of the thread hierarchy of a kernel function. In Chapter 3, you will see that this type of thread organization gives you the ability to efficiently execute the same application code on different devices, each with varying amounts of compute and memory resources.

# Launching a CUDA Kernel

- C function call syntax:  
`function_name (argument list);`
- A CUDA kernel call is a direct extension to the C function syntax that adds a kernel's execution configuration inside triple-angle-brackets:  
`kernel_name <>>(argument list);`
- CUDA programming model exposes the thread hierarchy
- With the execution configuration, specify how the threads will be scheduled to run on the GPU
- The first value in the execution configuration is the **grid dimension**, the number of blocks to launch
- The second value is the **block dimension**, the number of threads within each block
- By specifying the grid and block dimensions, you configure:
  - The total number of threads for a kernel
  - The layout of the threads you want to employ for a kernel
- Threads within the same block can easily communicate with each other, and threads that belong to different blocks cannot cooperate

# Launching a CUDA Kernel



Because the data is stored linearly in global memory, you can use the built-in variables `blockIdx.x` and `threadIdx.x` to:

- Identify a unique thread in the grid
- Establish a mapping between threads and data elements.

If you group all 32 elements into one block, then you just have one block as follows:

`kernel_name<<1,32>>(argument list);`

If you let each block just have one element, you have 32 blocks :

`kernel_name<<32,1>>(argument list);`

A kernel call is asynchronous with respect to the host thread

After a kernel is invoked, control returns to the host side immediately

Function to force the host application to wait for all kernels to complete

`cudaError_t cudaDeviceSynchronize(void);`

Some CUDA runtime APIs perform an implicit synchronization between the host and the device

## ASYNCHRONOUS BEHAVIORS

Unlike a C function call, all CUDA kernel launches are asynchronous. Control returns to the CPU immediately after the CUDA kernel is invoked.

- Kernel function is the code to be executed on the device side
- In a kernel function, define the computation for a single thread, and the data access for that thread
- When the kernel is called, many different CUDA threads perform the same computation in parallel
- A kernel is defined using the \_global\_ declaration specification as shown:  
\_global\_ void kernel\_name(argument list)
- A kernel function must have a void return type

# Writing Your Kernel

QUALIFIERS	EXECUTION	CALLABLE	NOTES
<code>__global__</code>	Executed on the device	Callable from the host Callable from the device for devices of compute capability 3	Must have a <code>void</code> return type
<code>__device__</code>	Executed on the device	Callable from the device only	
<code>__host__</code>	Executed on the host	Callable from the host only	Can be omitted

`__device__` and `__host__` qualifiers can be used together, in which case the function is compiled for both the host and the device

## CUDA KERNELS ARE FUNCTIONS WITH RESTRICTIONS

The following restrictions apply for all kernels:

- Access to device memory only
- Must have `void` return type
- No support for a variable number of arguments
- No support for static variables
- No support for function pointers
- Exhibit an asynchronous behavior

# Writing Your Kernel

Adding two vectors A and B of size N

The C code for vector addition on the host is given below:

```
void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

This is a sequential code that iterates N times

Peeling off the loop would produce the following kernel function:

```
_global_ void sumArraysOnGPU(float *A, float *B, float *C)
{ int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

# Verifying Your Kernel

need a host function to verify the result from the kernel

## VERIFYING KERNEL CODE

---

Besides many useful debugging tools, there are two very basic but useful means by which you can verify your kernel code.

First, you can use `printf` in your kernel for Fermi and later generation devices.

Second, you can set the execution configuration to `<<<1, 1>>>`, so you force the kernel to run with only one block and one thread. This emulates a sequential implementation. This is useful for debugging and verifying correct results. Also, this helps you verify that numeric results are bitwise exact from run-to-run if you encounter order of operations issues.

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## Compiling and Executing

**Dr. P. Padmakumari**

**CSE/SoC/SASTRA**



PresenterMedia

- Measure kernel performance
- Simplest method is to use either a CPU timer or a GPU timer to measure kernel executions from the host side

## Timing with CPU Timer

A CPU timer can be created by using the `gettimeofday` system call to get the system's wall-clock time, which returns the number of seconds since the epoch.

Need to include the `sys/time.h` header file

```
double cpuSecond()
{
    struct timeval tp;
    gettimeofday(&tp,NULL);
    return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);
}
```

# Timing with CPU Timer



- Measure kernel with cpuSecond in the following way:

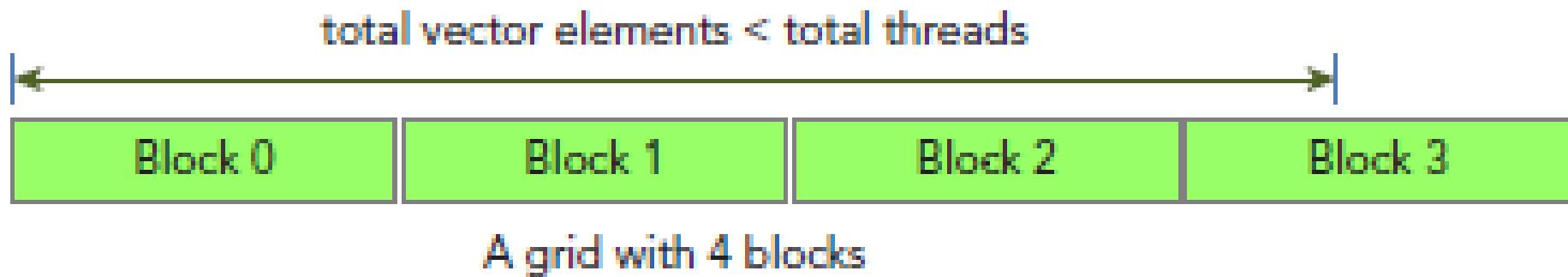
```
double iStart = cpuSecond();
kernel_name<<>>(argument list);
cudaDeviceSynchronize();
double iElaps = cpuSecond() - iStart;
```

- kernel call is asynchronous with respect to the host, you need to use cudaDeviceSynchronize to wait for all GPU threads to complete
- variable iElaps reports the time spent as if you had measured kernel execution with your wristwatch (in seconds)

# Timing with CPU Timer

- kernel for GPU scalability by calculating a row-major array index i using the block and thread indices, and by adding a test ( $i < N$ ) that checks for those indices that may exceed array bounds, as follows:

```
_global_ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```



# Measuring the vector summation kernel (sumArraysOnGPU-timer.cu)

LISTING 2-5: Measuring the vector summation kernel (sumArraysOnGPU-timer.cu)

```
#include <cuda_runtime.h>
#include <stdio.h>
#include <sys/time.h>

int main(int argc, char **argv) {
    printf("%s Starting...\n", argv[0]);

    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up date size of vectors
    int nElem = 1<<24;
    printf("Vector size %d\n", nElem);

    // malloc host memory
    size_t nBytes = nElem * sizeof(float);

    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A      = (float *)malloc(nBytes);
    h_B      = (float *)malloc(nBytes);
    hostRef = (float *)malloc(nBytes);
    gpuRef  = (float *)malloc(nBytes);

    double iStart,iElaps;

    // initialize data at host side
    iStart = cpuSecond();
    initData (h_A, nElem);
    initData (h_B, nElem);
    iElaps = cpuSecond() - iStart;

    memset (hostRef, 0, nBytes);
    memset (gpuRef, 0, nBytes);

    // add vector at host side for result checks
    iStart = cpuSecond();
    sumArraysOnHost (h_A, h_B, hostRef, nElem);
    iElaps = cpuSecond() - iStart;

    // malloc device global memory
    float *d_A, *d_B, *d_C;
    cudaMalloc((float**)&d_A, nBytes);
    cudaMalloc((float**)&d_B, nBytes);
    cudaMalloc((float**)&d_C, nBytes);
```

# Measuring the vector summation kernel (sumArraysOnGPU-timer.cu)

LISTING 2-5 *(continued)*

```
// transfer data from host to device
cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice);

// invoke kernel at host side
int iLen = 1024;
dim3 block (iLen);
dim3 grid ((nElem+block.x-1)/block.x);

iStart = cpuSecond();
sumArraysOnGPU <<<grid, block>>>(d_A, d_B, d_C,nElem);
cudaDeviceSynchronize();
iElaps = cpuSecond() - iStart;
printf("sumArraysOnGPU <<<%d,%d>>> Time elapsed %f" \
"sec\n", grid.x, block.x, iElaps);

// copy kernel result back to host side
cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost);

// check device results
checkResult(hostRef, gpuRef, nElem);

// free device global memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

// free host memory
free(h_A);
free(h_B);
free(hostRef);
free(gpuRef);

return(0);
}
```

## KNOW YOUR LIMITATIONS

---

A key concept to understand while tweaking the execution configuration is the limitations on grid and block dimensions. The maximum size at each level of the thread hierarchy is device dependent.

CUDA provides the ability to query the GPU for these limits. More information about this topic is covered in the “Managing Devices” section of this chapter.

For Fermi devices, the maximum number of threads per block is 1,024, and the maximum grid dimension for each *x*, *y*, and *z* dimension is 65,535.

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS ORGANIZING PARALLEL THREADS

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

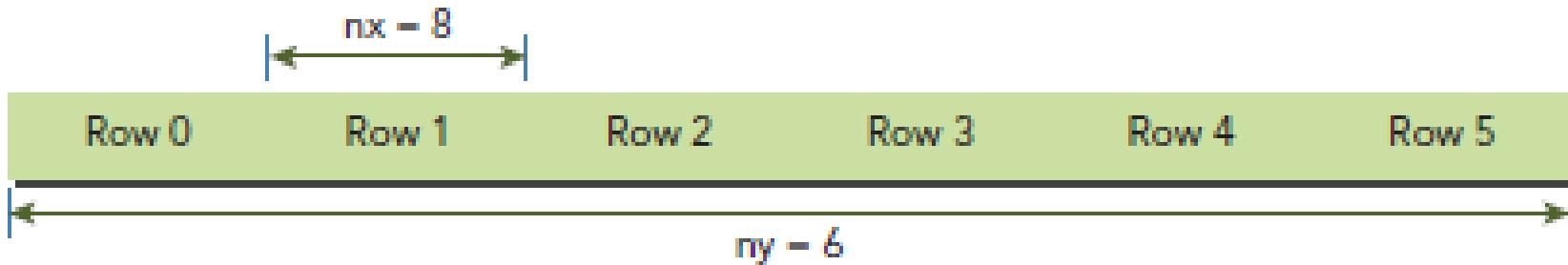
# Organize threads



- Organize threads using the right grid and block size, it can make a big impact on kernel performance
- Matrix addition:
  - ▶ 2D grid with 2D blocks
  - ▶ 1D grid with 1D blocks
  - ▶ 2D grid with 1D blocks

# Indexing Matrices with Blocks and Threads

- Matrix is stored linearly in global memory with a row-major approach
- Matrix addition kernel, a thread is usually assigned one data element to process



- Three kinds of indices for a 2D case need to manage:
  - Thread and block index
  - Coordinate of a given point in the matrix
  - Offset in linear global memory

- For a given thread, to obtain the offset in global memory from the block and thread index by

- ✓ First mapping the thread and block index to coordinates in the matrix
  - ✓ Map the thread and block index to the coordinate of a matrix with the following formula:

$ix = threadIdx.x + blockIdx.x * blockDim.x$

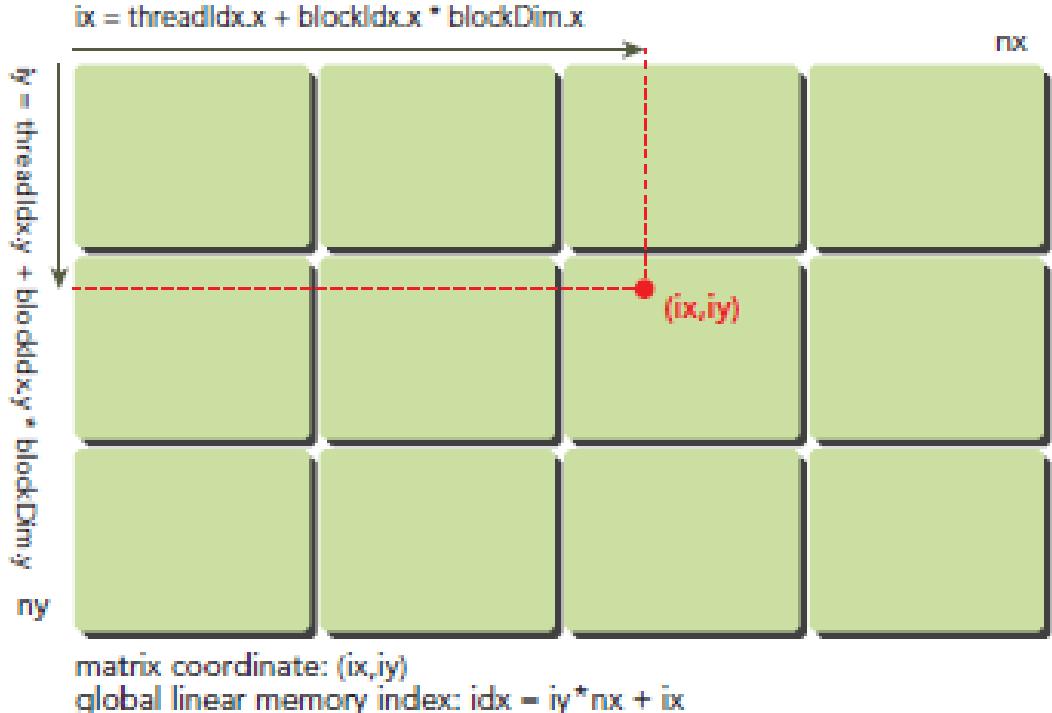
$iy = threadIdx.y + blockIdx.y * blockDim.y$

- ✓ Mapping those matrix coordinates to a global memory location
  - ✓ Map a matrix coordinate to a global memory location/index with the following formula:

$idx = iy * nx + ix$

# Relationship among block and thread indices

Relationship among block and thread indices, matrix coordinates, and linear global memory indices



# printThreadInfo



Function **printThreadInfo** is used to print out the following information about each thread:

- Thread index
- Block index
- Matrix coordinate
- Global linear memory offset
- Value of corresponding elements

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **ORGANIZING PARALLEL THREADS**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

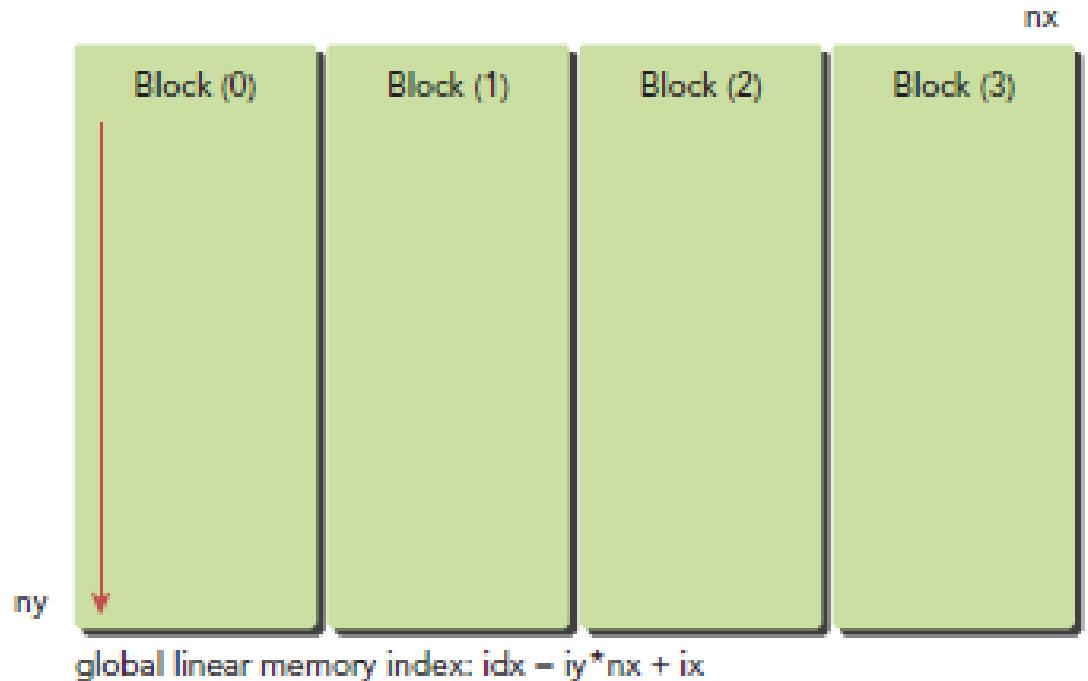
# Organize threads



- Organize threads using the right grid and block size, it can make a big impact on kernel performance
- Matrix addition:
  - ▶ 2D grid with 2D blocks
  - ▶ 1D grid with 1D blocks
  - ▶ 2D grid with 1D blocks

# Summing Matrices with a 1D Grid and 1D Blocks

- Use a 1D grid with 1D blocks, need to write a new kernel in which each thread processes  $ny$



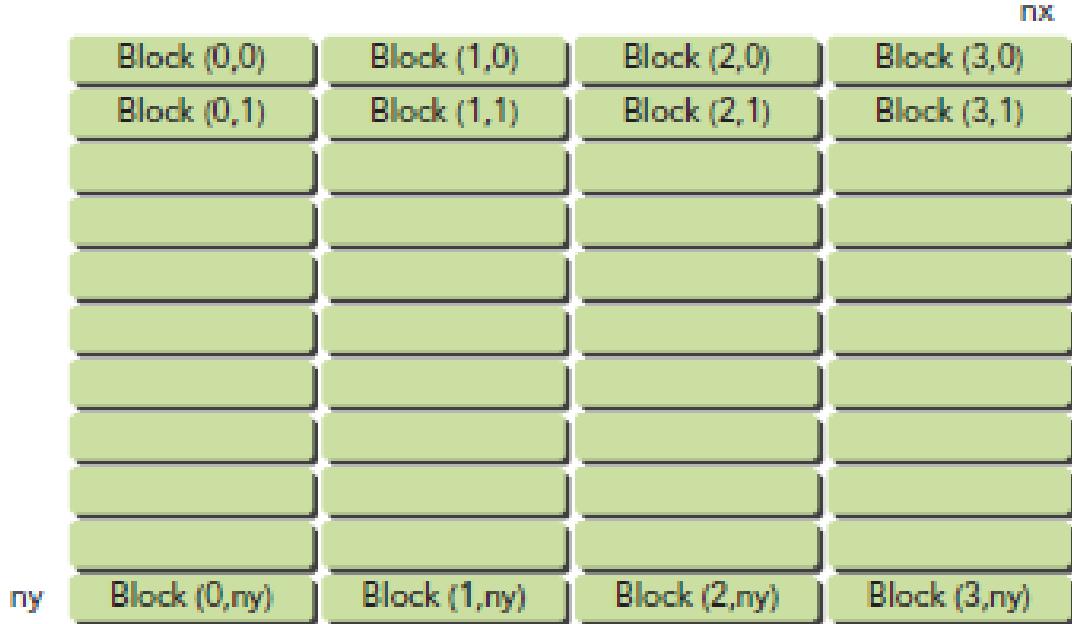
only `threadIdx.x` is useful and a loop inside the kernel is used to handle  $ny$  elements in each thread

# Summing Matrices with a 1D Grid and 1D Blocks

```
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC,
    int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < nx ) {
        for (int iy=0; iy<ny; iy++) {
            int idx = iy*nx + ix;
            MatC[idx] = MatA[idx] + MatB[idx];
        }
    }
}
```

# Summing Matrices with a 2D Grid and 1D Blocks

- When using a 2D grid that contains 1D blocks, each thread takes care of only one data element and the second dimension of grid equals ny



# Summing Matrices with a 2D Grid and 1D Blocks



```
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *MatC,
        int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy*nx + ix;

    if (ix < nx && iy < ny)
        MatC[idx] = MatA[idx] + MatB[idx];
}
```

# Comparison of Different Kernel Implementations

KERNEL	EXECUTION CONFIGURE	TIME ELAPSED
sumMatrixOnGPU2D	(512,1024), (32,16)	0.038041
sumMatrixOnGPU1D	(128,1), (128,1)	0.044701
sumMatrixOnGPUMix	(64,16384), (256,1)	0.030765

- Two basic and powerful means to query and manage GPU devices
  - The CUDA runtime API functions
  - The NVIDIA Systems Management Interface (nvidia-smi) command-line utility

## Determining the Best GPU

```
int numDevices = 0;
cudaGetDeviceCount(&numDevices);
if (numDevices > 1) {
    int maxMultiprocessors = 0, maxDevice = 0;
    for (int device=0; device<numDevices; device++) {
        cudaDeviceProp props;
        cudaGetDeviceProperties(&props, device);
        if (maxMultiprocessors < props.multiProcessorCount) {
            maxMultiprocessors = props.multiProcessorCount;
            maxDevice = device;
        }
    }
    cudaSetDevice(maxDevice);
}
```

# Using nvidia-smi to Query GPU Information



- Command-line tool nvidia-smi assists with managing and monitoring GPU devices, and allows to query and modify device state

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS GLOBAL MEMORY**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



# CUDA memory model



- Learning the CUDA memory model
  - Managing CUDA memory
  - Programming with global memory
  - Exploring global memory access patterns
  - Probing global memory data layout
  - Programming with unified memory
  - Maximizing global memory throughput

# INTRODUCING THE CUDA MEMORY MODEL



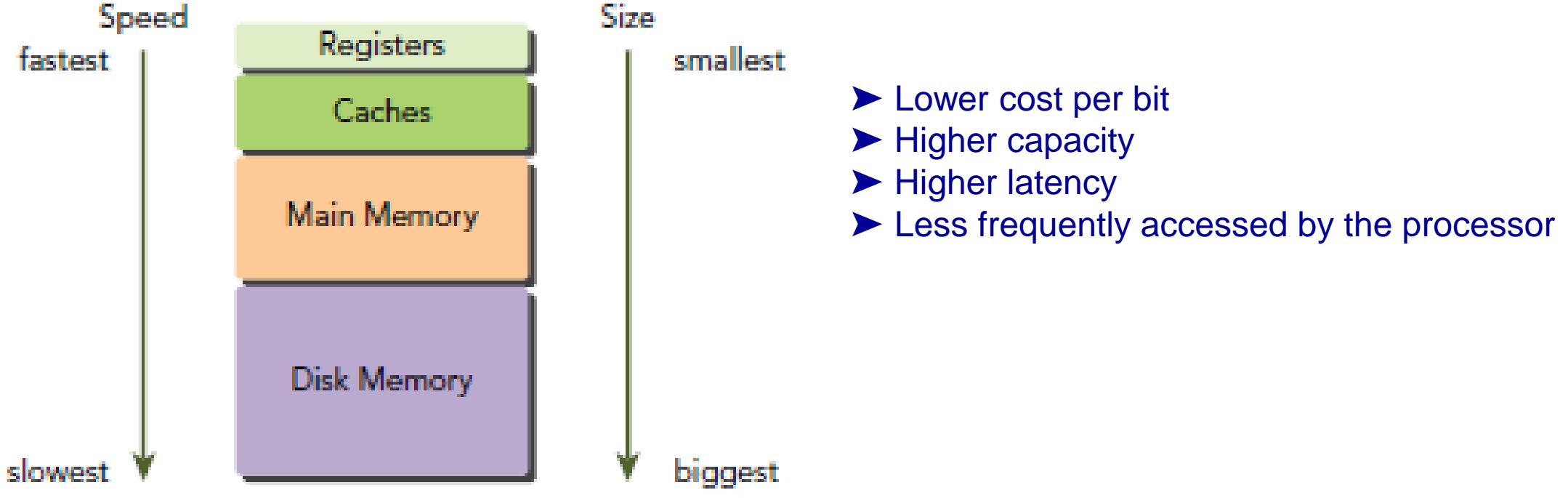
- Memory access and management are important parts of any programming language
- Memory management has a particularly **large impact on high performance computing** in modern accelerators
- Because many workloads are limited by how rapidly they can
  - load and store data,
  - having a large amount of low-latency,
  - high-bandwidth memory can be very beneficial to performance
- Procuring large capacity, high-performance memory is not always possible or economical
  - memory model to achieve optimal latency and bandwidth, given the hardware memory subsystem
- CUDA memory model unifies separate host and device memory systems and exposes the full memory hierarchy - **explicitly control data placement** for optimal performance

# Benefits of a Memory Hierarchy



- Principle of locality - suggests that they access a relatively small and localized portion of their address space at any point-in-time
- Two different types of locality:
  - ▶ Temporal locality (locality in time)  
if a data location is referenced, then it is more likely to be referenced again within a short time period and less likely to be referenced as more and more time passes
  - ▶ Spatial locality (locality in space)  
if a memory location is referenced, nearby locations are likely to be referenced as well

# Benefits of a Memory Hierarchy

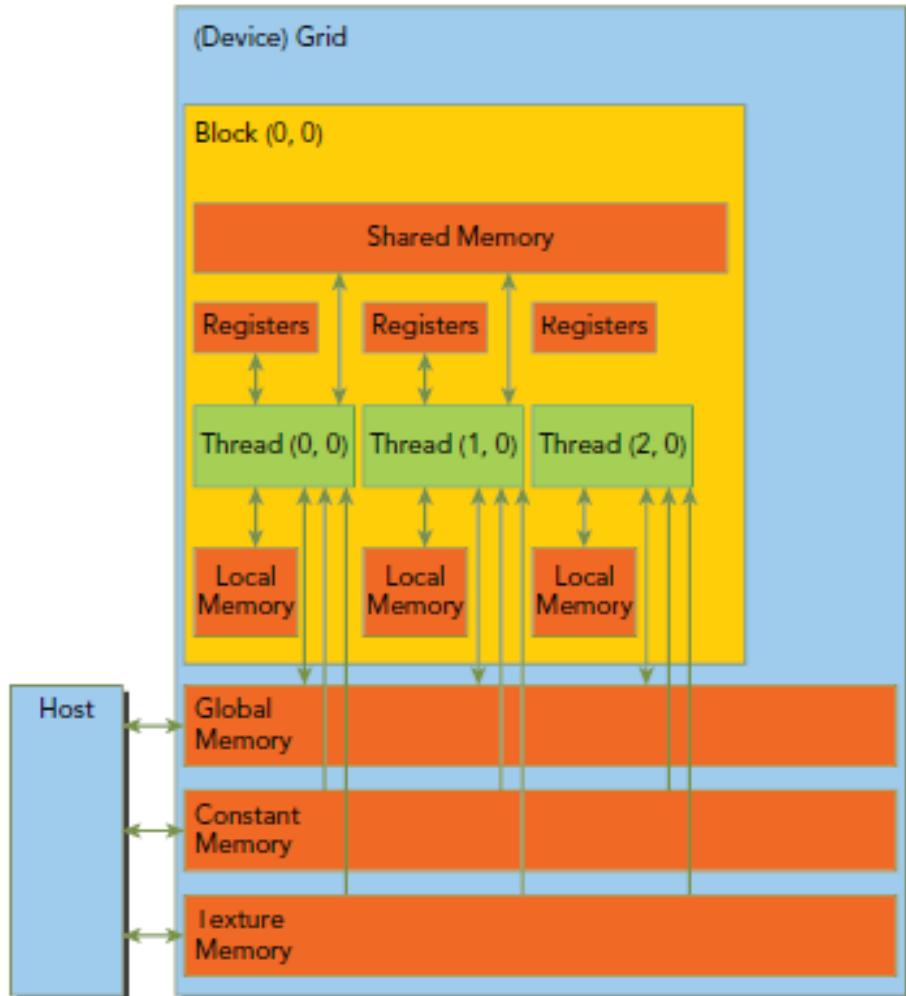


# CUDA Memory Model



- To programmers, there are generally two classifications of memory:
  - ▶ Programmable: explicitly control what data is placed in programmable memory
  - ▶ Non-programmable: no control over data placement, and rely on automatic techniques to achieve good performance
- CUDA memory model exposes many types of programmable memory
  - ▶ Registers
  - ▶ Shared memory
  - ▶ Local memory
  - ▶ Constant memory
  - ▶ Texture memory
  - ▶ Global memory

# Hierarchy of CUDA memory spaces



# CUDA Memory Model



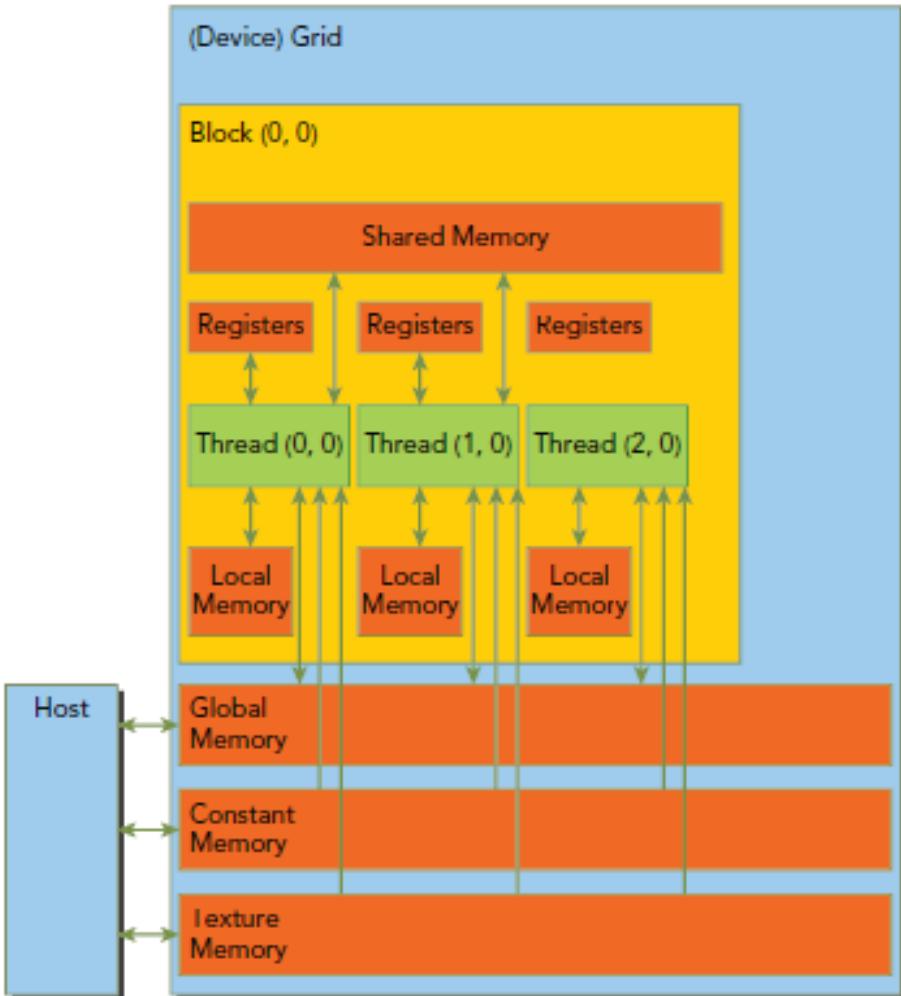
- To programmers, there are generally two classifications of memory:
  - ▶ Programmable: explicitly control what data is placed in programmable memory
  - ▶ Non-programmable: no control over data placement, and rely on automatic techniques to achieve good performance
- CUDA memory model exposes many types of programmable memory
  - ▶ Registers
  - ▶ Shared memory
  - ▶ Local memory
  - ▶ Constant memory
  - ▶ Texture memory
  - ▶ Global memory

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS GLOBAL MEMORY**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



# Hierarchy of CUDA memory spaces



# Registers



- Fastest memory space on a GPU
- Automatic variable declared in a kernel without any other type qualifiers is generally stored in a register
- Arrays declared in a kernel may also be stored in registers, but only if the indices used to reference the array are constant and can be determined at compile time
- Register variables are private to each thread
- Kernel typically uses registers to hold frequently accessed thread-private variables
- Register variables share their lifetime with the kernel
- Once a kernel completes execution, a register variable cannot be accessed again
- On Fermi GPUs, there is a hardware limit of 63 registers per thread
- Kepler expands limit to 255 registers per thread
- If a kernel uses more registers than the hardware limit, the excess registers will spill over to local memory. This **register spilling** can have adverse performance consequences.

# Local Memory



- Variables in a kernel that are eligible for registers but cannot fit into the register space allocated for that kernel will spill into local memory
- Variables that the compiler is likely to place in local memory are:
  - ► Local arrays referenced with indices whose values cannot be determined at compile-time
  - ► Large local structures or arrays that would consume too much register space
  - ► Any variable that does not fit within the kernel register limit
- The name “local memory” is misleading: Values spilled to local memory reside in the same physical location as global memory, so local memory accesses are characterized by high latency and low bandwidth and are subject to the requirements for efficient memory access
- For GPUs with compute capability 2.0 and higher, local memory data is also cached in a per-SM L1 and per-device L2 cache.

# Shared Memory



- Variables decorated with the following attribute in a kernel are stored in shared memory:  
`_shared`
- shared memory is on-chip, it has a much higher bandwidth and much lower latency than local or global memory
- Used similarly to CPU L1 cache, but is also programmable
- Each SM has a limited amount of shared memory that is partitioned among thread blocks
- Shared memory serves as a basic means for inter-thread communication
- Threads within a block can cooperate by sharing data stored in shared memory
- Access to shared memory must be synchronized using the following CUDA runtime call introduced  
`void __syncthreads();`
- Function creates a barrier which all threads in the same thread block must reach before any other thread is allowed to proceed

# Constant Memory



- Constant memory resides in device memory and is cached in a dedicated, per-SM constant cache.
- A constant variable is decorated with the following attribute:

\_constant\_

- Constant variables must be declared with global scope, outside of any kernels
- A limited amount of constant memory can be declared — 64 KB for all compute capabilities
- Constant memory is statically declared and visible to all kernels in the same compilation unit
- Kernels can only read from constant memory
- Constant memory must therefore be initialized by the host using:

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src, size_t count);
```

This function copies count bytes from the memory pointed to by src to the memory pointed to by symbol, which is a variable that resides on the device in global or constant memory. This function is synchronous in most cases.

# Constant Memory



- Constant memory performs best when all threads in a warp read from the same memory address.
- For example, a coefficient for a mathematical formula is a good use case for constant memory because all threads in a warp will use the same coefficient to conduct the same calculation on different data
- If each thread in a warp reads from a different address, and only reads once, then constant memory is not the best choice because a single read from constant memory broadcasts to all threads in a warp

# Texture Memory



- Texture memory resides in device memory and is cached in a per-SM, read-only cache
- Texture memory is a type of global memory that is accessed through a dedicated read-only cache
- The readonly cache includes support for hardware filtering, which can perform floating-point interpolation as part of the read process
- Texture memory is optimized for 2D spatial locality, so threads in a warp that use texture memory to access 2D data will achieve the best performance
- For some applications, this is ideal and provides a performance advantage due to the cache and the filtering hardware
- However, for other applications using texture memory can be slower than global memory

# Global Memory



- Global memory is the largest, highest-latency, and most commonly used memory on a GPU
- The name global refers to its scope and lifetime
- Its state can be accessed on the device from any SM throughout the lifetime of the application
- A variable in global memory can either be declared statically or dynamically
- Declare a global variable statically in device code using the following qualifier:  
`_device_`
- Global memory is allocated by the host using `cudaMalloc` and freed by the host using `cudaFree`
- Pointers to global memory are then passed to kernel functions as parameters
- Global memory allocations exist for the lifetime of an application and are accessible to all threads of all kernels

- GPU Caches Like CPU caches, GPU caches are non-programmable memory
- There are four types of cache in GPU devices:
  - L1
  - L2
  - Read-only constant
  - Read-only texture
- One L1 cache per-SM and one L2 cache shared by all SMs
- Both L1 and L2 caches are used to store data in local and global memory, including register spills
- On Fermi GPUs and Kepler K40 or later GPUs, CUDA allows you to configure whether reads are cached in both L1 and L2, or only in L2

# GPU Caches



- CPU, both memory loads and stores can be cached
- GPU only memory load operations can be cached; memory store operations cannot be cached
- Each SM also has a read-only constant cache and read-only texture cache that are used to improve read performance from their respective memory spaces in device memory

# CUDA Variable Declaration Summary

TABLE 4-1: CUDA Variable and Type Qualifier

QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	float var	Register	Thread	Thread
	float var[100]	Local	Thread	Thread
<code>_shared_</code>	float var†	Shared	Block	Block
<code>_device_</code>	float var†	Global	Global	Application
<code>_constant_</code>	float var†	Constant	Global	Application

# CUDA Variable Declaration Summary

TABLE 4-2: Salient Features of Device Memory

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **MEMORY MANAGEMENT**

**Dr. P. Padmakumari**

**CSE/SoC/SASTRA**



# Introduction



- Memory management in CUDA programming is similar to C programming - added programmer responsibility of **explicitly managing data movement between the host and device**
- Explicitly manage memory and data movement using CUDA functions to:
  - Allocate and deallocate device memory
  - Transfer data between the host and device
- Achieve maximum performance - CUDA provides functions that prepare device memory on the host and explicitly transfer data to and from the device

# Memory Allocation and Deallocation



- Allocate global memory on the host using the following function

```
cudaError_t cudaMalloc(void **devPtr, size_t count)
```

- Function allocates count bytes of global memory on the device and returns the location of that memory in pointer devPtr

- Deallocated using:

```
cudaError_t cudaFree(void *devPtr)
```

- Function frees the global memory pointed to by devPtr

- Device memory allocation and deallocation are expensive operations, so device memory should be reused by applications whenever possible to minimize the impact on overall performance

# Memory Transfer



- Global memory is allocated, transfer data to the device from the host using the following function:

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind);
```

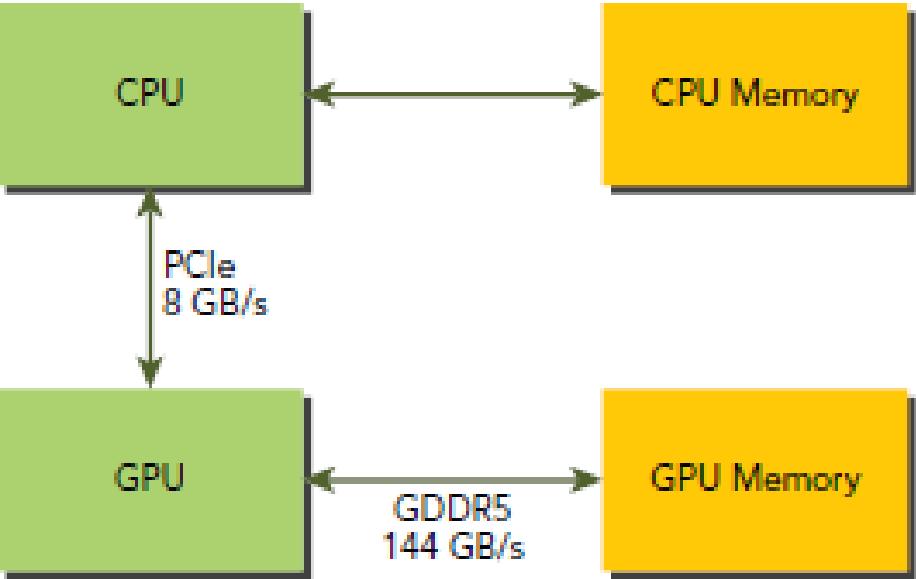
- Function copies count bytes from the memory location src to the memory location dst

- Variable kind specifies the direction of the copy and can have the following values:

- ✓ cudaMemcpyHostToHost
  - ✓ cudaMemcpyHostToDevice
  - ✓ cudaMemcpyDeviceToHost
  - ✓ cudaMemcpyDeviceToDevice

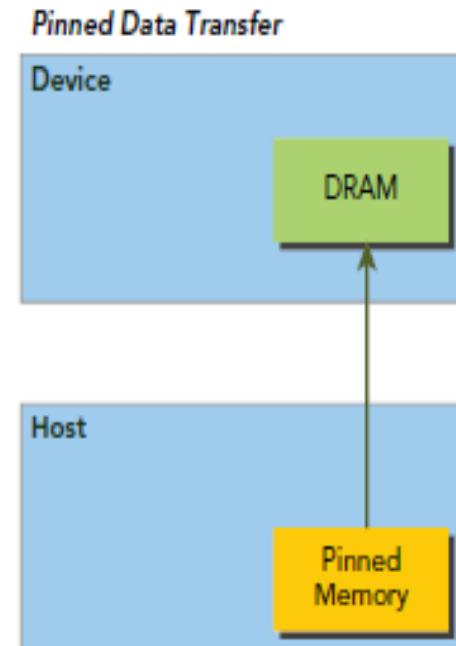
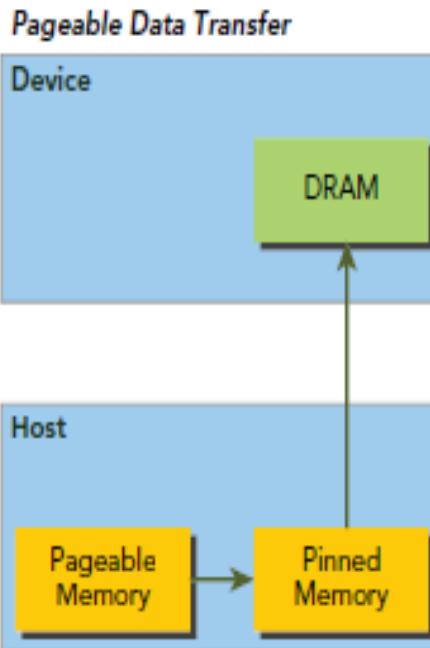
# Connectivity of CPU and GPU memory

- Peak bandwidth between the GPU chip and the on-board GDDR5 GPU memory is very high, 144 GB/sec for a Fermi C2050 GPU
- Link between CPU and GPU through the PCI Express (PCIe) Gen2 bus shows a much lower theoretical peak bandwidth of 8 GB/sec (the PCIe Gen3 maximum theoretical limit is 16 GB/sec)
- Disparity means that data transfers between the host and device can throttle overall application performance if not managed properly
- Basic principle of CUDA programming, you should always be thinking of ways to minimize host device transfers



# Pinned Memory

- The CUDA runtime allows you to directly allocate pinned host memory using:  
`cudaError_t cudaMallocHost(void **devPtr, size_t count);`
- Pinned memory is more expensive to allocate and deallocate than pageable memory
- Provides higher transfer throughput for large data transfers
- Speedup achieved when using pinned memory relative to pageable memory depends on device compute capability
- For example, on Fermi devices it is generally beneficial to use pinned memory when transferring more than 10 MB of data
- Batching many small transfers into one larger transfer improves performance because it reduces per-transfer overhead



# Zero-Copy Memory



- Host cannot directly access device variables, and the device cannot directly access host variables
- Both the host and device can access zero-copy memory
- GPU threads can directly access zero-copy memory
- Several advantages to using zero-copy memory in CUDA kernels, such as:
  - Leveraging host memory when there is insufficient device memory
  - Avoiding explicit data transfer between the host and device
  - Improving PCIe transfer rates
- When using zero-copy memory to share data between the host and device, must synchronize memory accesses across the host and device
- Modifying data in zero-copy memory from both the host and device at the same time will result in undefined behavior

# Zero-copy memory



- Zero-copy memory is pinned (non-pageable) memory that is mapped into the device address space
- Create a mapped, pinned memory region with the following function:

```
cudaError_t cudaHostAlloc(void **pHost, size_t count, unsigned int flags)
```

- Function allocates count bytes of host memory that is page-locked and accessible to the device
- Memory allocated by this function must be freed with `cudaFreeHost`
- The flags parameter enables further configuration of special properties of the allocated memory:

- ▶ `cudaHostAllocDefault`
- ▶ `cudaHostAllocPortable`
- ▶ `cudaHostAllocWriteCombined`
- ▶ `cudaHostAllocMapped`

# Zero-copy memory

## ZERO-COPY MEMORY

---

There are two common categories of heterogeneous computing system architectures: Integrated and discrete.

In integrated architectures, CPUs and GPUs are fused onto a single die and physically share main memory. In this architecture, zero-copy memory is more likely to benefit both performance and programmability because no copies over the PCIe bus are necessary.

For discrete systems with devices connected to the host via PCIe bus, zero-copy memory is advantageous only in special cases.

Because the mapped pinned memory is shared between the host and device, you must synchronize memory accesses to avoid any potential data hazards caused by multiple threads accessing the same memory location without synchronization.

Be careful to not overuse zero-copy memory. Device kernels that read from zero-copy memory can be very slow due to its high-latency.

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## SHARED MEMORY

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



# Overview



- Learning how data is arranged in shared memory
- Mastering index conversion from 2D shared memory to linear global memory
- Resolving bank conflicts for different access modes
- Caching data in shared memory to reduce global memory accesses
- Avoiding non-coalesced global memory access using shared memory
- Understanding the difference between the constant cache and the read-only cache
- Programming with the warp shuffle instruction

# CUDA SHARED MEMORY



- GPUs are equipped with two types of memory:
  - On-board memory
  - On-chip memory
- Global memory is large, on-board memory and is characterized by relatively high latencies
- Shared memory is smaller, low-latency on-chip memory that offers much higher bandwidth than global memory
- Shared memory is generally useful as:
  - ➤ An intra-block thread communication channel
  - ➤ A program-managed cache for global memory data
  - ➤ Scratch pad memory for transforming data to improve global memory access

# Shared Memory

- Shared memory (SMEM) is one of the key components of the GPU
- Physically, each SM contains a small low-latency memory pool shared by all threads in the thread block currently executing on that SM
- Shared memory enables threads within the same thread block to cooperate, facilitates reuse of on-chip data, and can greatly reduce the global memory bandwidth needed by kernels
- Because the contents of shared memory are explicitly managed by the application, it is often described as a program-managed cache
- Fermi and Kepler GPUs have similar memory hierarchies, except Kepler includes an additional compiler-directed cache for read-only data

# Shared Memory

- All load and store requests to global memory go through the L2 cache, which is the primary point of data unification between SM units
- Note that shared memory and L1 cache are physically closer to the SM than both the L2 cache and global memory
- As a result, shared memory latency is roughly 20 to 30 times lower than global memory, and bandwidth is nearly 10 times higher.



## PROGRAM-MANAGED CACHE

In C programming, loop transformations are a common cache optimization. Loop transformations can improve cache locality during loop traversal by re-arranging the order of iterations. At the algorithm level, you need to manually adjust loops to achieve better spatial locality while considering cache size. The cache is transparent to your program, and the compiler handles all data movement. You have no ability to control cache eviction.

Shared memory is a program-managed cache. You have full control over when data is moved into shared memory, and when data is evicted. By allowing you to manually manage shared memory, CUDA makes it easier for you to optimize your application code by providing more fine-grained control over data placement and improving on-chip data movement.

# Shared Memory Allocation



- Several ways to allocate or declare shared memory variables depending on your application requirements
- Allocate shared memory variables either statically or dynamically
- Shared memory can also be declared as either local to a CUDA kernel or globally in a CUDA source code file
- CUDA supports declaration of 1D, 2D, and 3D shared memory arrays
- A shared memory variable is declared with the following qualifier:

**\_shared\_**

# Shared Memory Allocation



- The following code segment statically declares a shared memory 2D float array.
- If declared inside a kernel function, the scope of this variable is local to the kernel. If declared outside of any kernels in a file, the scope of this variable is global to all kernels

```
_shared_ float tile[size_y][size_x];
```

- If the size of shared memory is unknown at compile time, you can declare an un-sized array with the extern keyword
- For example, the following code segment declares a shared memory 1D un-sized int array.
- This declaration can be made either inside a kernel or outside of all kernels

```
extern _shared_ int tile[];
```

- Because the size of this array is unknown at compile-time, you need to dynamically allocate shared memory at each kernel invocation by specifying the desired size in bytes as a third argument inside the triple angled brackets, as follows:

```
kernel<<<grid, block, isize * sizeof(int)>>>(...)
```

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## Shared Memory Banks and Access Mode

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

# Shared Memory Banks and Access Mode



- Two key properties to measure when optimizing memory performance:  
latency and bandwidth
- Impact on kernel performance of latency and bandwidth caused by different global memory access patterns
- Shared memory can be used to hide the performance impact of global memory latency and bandwidth
- To fully exploit these resources, it is helpful to understand how shared memory is arranged

# Memory Banks



- To achieve high memory bandwidth, shared memory is divided into **32** equally-sized memory modules, called banks, which can be accessed simultaneously
- There are **32 banks** because there are **32 threads in a warp**
- Shared memory is a **1D address space**
- Depending on the compute capability of a GPU, the addresses of shared memory are mapped to different banks in different patterns
- If a shared memory load or store operation issued by a **warp** does not access more than one memory location per bank, the operation can be serviced by one memory transaction
- Otherwise, the operation is serviced by multiple memory transactions, thereby decreasing memory bandwidth utilization

# Bank Conflict

- When multiple addresses in a shared memory request fall into the same memory bank, a bank conflict occurs, causing the request to be replayed
- Hardware splits a request with a bank conflict into as many separate conflict-free transactions as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory transactions required
- Three typical situations occur when a request to shared memory is issued by a warp:
  - Parallel access: multiple addresses accessed across multiple banks
  - Serial access: multiple addresses accessed within the same bank
  - Broadcast access: a single address read in a single bank

Parallel access is the most common pattern:

- Multiple addresses accessed by a warp that fall into multiple banks
- Implies that some, if not all, of the addresses can be serviced in a single memory transaction
- Optimally, a conflict-free shared memory access is performed when every address is in a separate bank

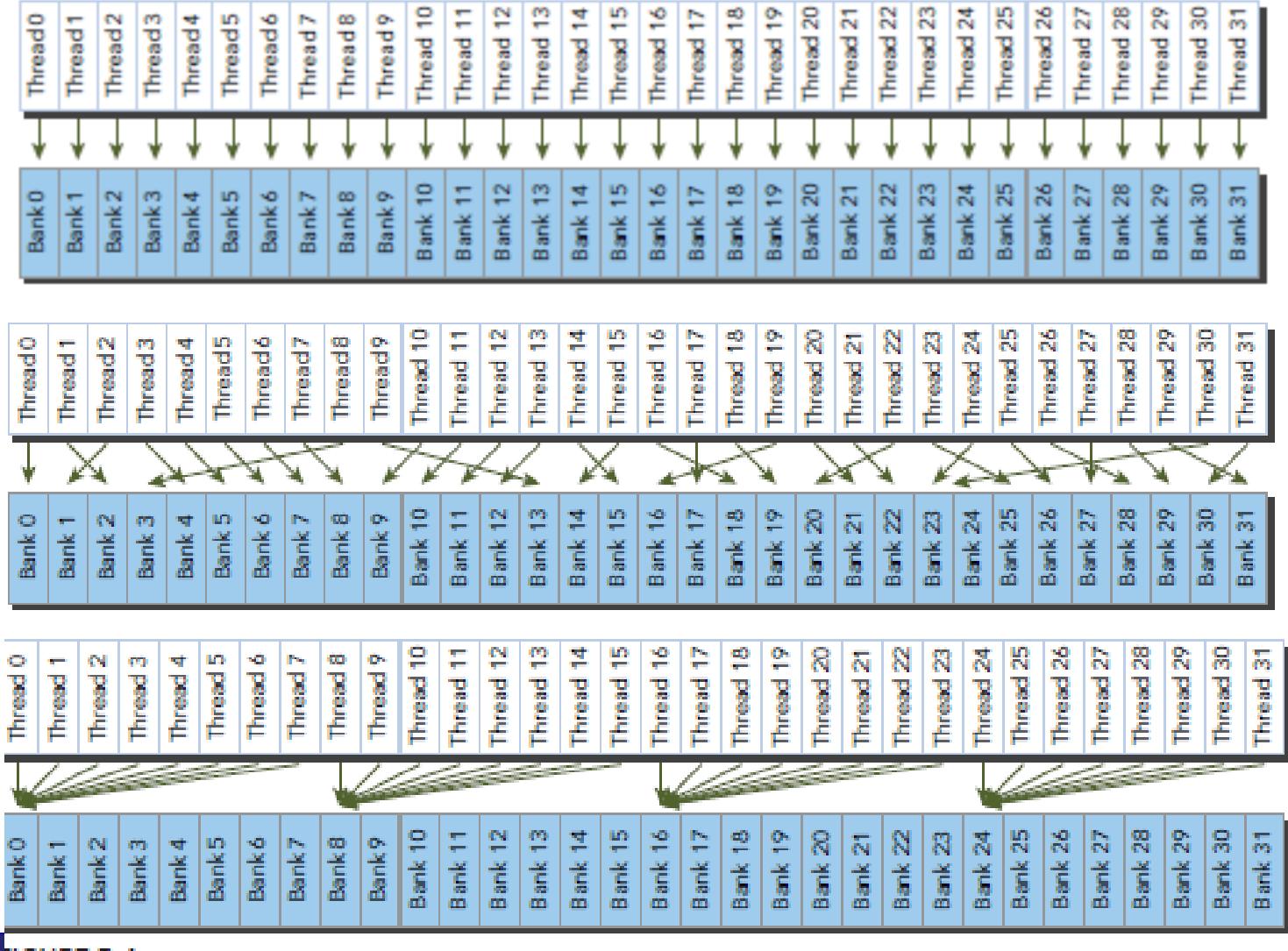
Serial access is the worst pattern:

- When multiple addresses fall into the same bank, the request must be serialized
- If all 32 threads in a warp access different memory locations in a single bank, 32 memory transactions will be required and satisfying those accesses will take 32 times as long as a single request

Broadcast access:

- All threads in a warp read the same address within a single bank
- One memory transaction is executed, and the accessed word is broadcast to all requesting threads
- While only a single memory transaction is required for a broadcast access, bandwidth utilization is poor because only a small number of bytes are read

# Optimal parallel access pattern



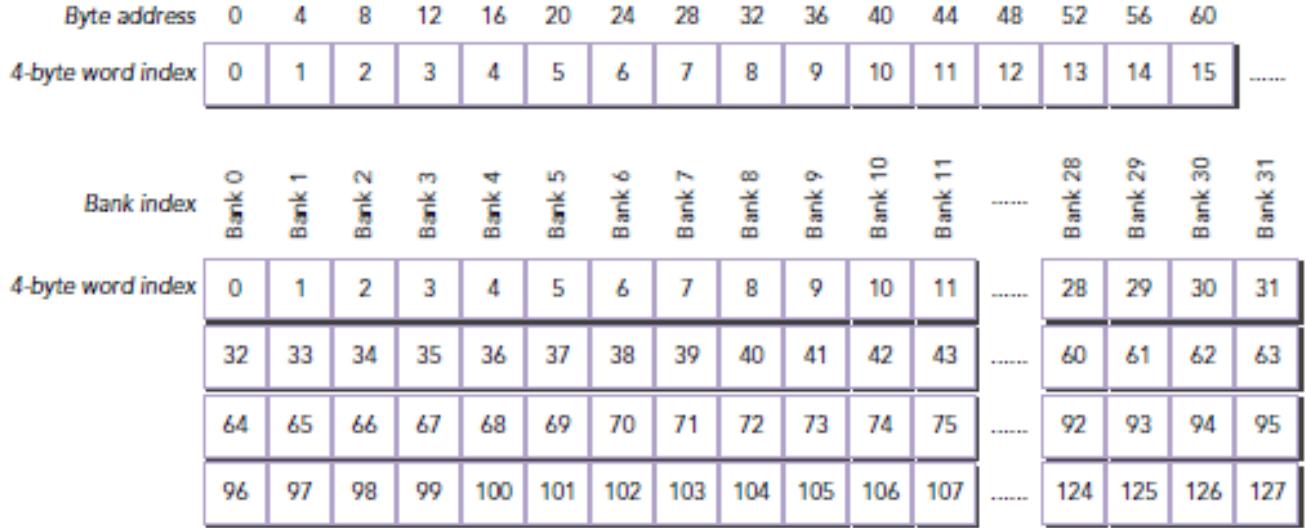
# Access Mode



- Shared memory bank width defines which shared memory addresses are in which shared memory banks.
- Memory bank width varies for devices depending on compute capability
- There are two different bank widths:
  - ▶ 4 bytes (32-bits) for devices of compute capability 2.x
  - ▶ 8 bytes (64-bits) for devices of compute capability 3.x
- For a Fermi device, the bank width is 32-bits and there are 32 banks
- Each bank has a bandwidth of 32 bits per two clock cycles
- Successive 32-bit words map to successive banks
- Mapping from shared memory address to bank index can be calculated as follows

$$\text{bank index} = (\text{byte address} \div 4 \text{ bytes/bank}) \% 32 \text{ banks}$$

# Access Mode



- A bank conflict does not occur when two threads from the same warp access the same address
- In that case, for read accesses, the word is broadcast to the requesting threads, and for write accesses, the word is written by only one of the threads — which thread performs the write is undefined

# Conflict-free access in 64-bit mode



- Shared memory bank width defines which shared memory addresses are in which shared memory banks.
- Memory bank width varies for devices depending on compute capability
- There are two different bank widths:
  - ▶ 4 bytes (32-bits) for devices of compute capability 2.x
  - ▶ 8 bytes (64-bits) for devices of compute capability 3.x
- For a Fermi device, the bank width is 32-bits and there are 32 banks
- Each bank has a bandwidth of 32 bits per two clock cycles
- Successive 32-bit words map to successive banks
- Mapping from shared memory address to bank index can be calculated as follows

$$\text{bank index} = (\text{byte address} \div 4 \text{ bytes/bank}) \% 32 \text{ banks}$$

# Access Mode

- For Kepler devices, shared memory has 32 banks with the following two address modes:
  - 64-bit mode
  - 32-bit mode

In 64-bit mode, successive 64-bit words map to successive banks

Each bank has a bandwidth of 64 bits per clock cycle.

The mapping from shared memory address to bank index can be calculated as follows:

$$\text{bank index} = (\text{byte address} \div 8 \text{ bytes/bank}) \% 32 \text{ banks}$$

Byte address	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	.....
4-byte word index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	.....

Bank index	Bank 0		Bank 1		Bank 2		Bank 3		Bank 4		Bank 5		Bank 30		Bank 31	
4-byte word index	0	32	1	33	2	34	3	35	4	36	5	37	28	62	31	63
	64	96	65	97	66	98	67	99	68	100	69	101	94	126	95	127
	128	160														
	192	224														

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## Shared Memory

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

# Synchronization



- Synchronization among parallel threads is a key mechanism for any parallel computing language
- Shared memory can be simultaneously accessed by multiple threads within a thread block
- Cause inter-thread conflicts when the same shared memory location is modified by multiple threads without synchronization
- CUDA provides several runtime functions to perform intra-block synchronization
- There are two basic approaches to synchronization:
  - ✓ Barriers
  - ✓ Memory fences
- At a barrier, all calling threads wait for all other calling threads to reach the barrier point
- At a memory fence, all calling threads stall until all modifications to memory are visible to all other calling threads

# Weakly-Ordered Memory Model



- Memory accesses are **not necessarily executed in the order** in which they appear in the program
- CUDA adopts a weakly-ordered memory model to enable more aggressive compiler optimizations
- Order in which a GPU thread writes data to different memories, such as shared memory, global memory, page-locked host memory, or the memory of a peer device, is not necessarily the same order of those accesses in the source code
- Order **thread's writes** - become visible to other threads may not match the actual order in which those writes were performed
- Order **thread reads** - data from different memories is not necessarily the order in which the read instructions appear in the program if instructions are independent of each other
- Explicitly force a certain ordering for program correctness, **memory fences and barriers** must be inserted in application code – guarantee the **correct behavior** of a kernel that shares resources with other threads

# Explicit Barrier

- only possible to perform a barrier among threads in the same thread block  
`void __syncthreads();`
- `__syncthreads` acts as a barrier point at which threads in a block must wait until all threads have reached that point
- `__syncthreads` also ensures that all global and shared memory accesses made by these threads prior to the barrier point are visible to all threads in the same block
- `__syncthreads` is used to coordinate communication between the threads of the same block
- When some threads within a block access the same addresses in shared or global memory, there are potential hazards (read-after-write, write-after-read, and write-after-write) which will result in undefined application behavior and undefined state at those memory locations
- Undesirable behavior can be avoided by synchronizing threads between conflicting accesses
- only valid to call `__syncthreads` if a conditional is guaranteed to evaluate identically across the entire thread block

# Memory Fence Functions



- Memory fence functions ensure that any memory write before the fence is visible to other threads after the fence.
- Three variants of memory fences depending on the desired scope: block, grid, or system
- Create a memory fence within a thread block using the following intrinsic function:

**void \_\_threadfence\_block();**

`__threadfence_block` ensures that all writes to shared memory and global memory made by a calling thread before the fence are visible to other threads in the same block after the fence

Recall that memory fences do not perform any thread synchronization, and so it is not necessary for all threads in a block to actually execute this instruction

Create a memory fence at the grid level using the following intrinsic function:

**void \_\_threadfence();**

`__threadfence` stalls the calling thread until all of its writes to global memory are visible to all threads in the same grid

# Memory Fence Functions



- Set a memory fence across the system (including host and device) using the following intrinsic function:

**void \_\_threadfence\_system()**

`__threadfence_system` stalls the calling thread to ensure all its writes to global memory, pagelocked host memory, and the memory of other devices are visible to all threads in all devices and host threads

- Declaring a variable in global or shared memory using the volatile qualifier prevents compiler optimization which might temporally cache data in registers or local memory
- With the volatile qualifier, the compiler assumes that the variable's value can be changed or used at any time by any other thread
- Any reference to this variable is compiled to a global memory read or global memory write instruction that skips the cache

## SHARED MEMORY VERSUS GLOBAL MEMORY

GPU global memory resides in device memory (DRAM), and it is much slower to access than GPU shared memory. Compared to DRAM, shared memory has:

- 20 to 30 times lower latency than DRAM
- Greater than 10 times higher bandwidth than DRAM

The access granularity of shared memory is also smaller. While the access granularity of DRAM is either 32 bytes or 128 bytes, the access granularity of shared memory is as follows:

- Fermi: 4 bytes bank width
- Kepler: 8 bytes bank width

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## Streams and Concurrency

**Dr. P. Padmakumari**

**CSE/SoC/SASTRA**



PresenterMedia

# Introducing Streams and Concurrency



- A CUDA stream refers to a **sequence of asynchronous CUDA operations** that execute on a device in the order issued by the host code
- A stream encapsulates these operations, maintains their ordering, **permits operations to be queued** in the stream to be executed after all preceding operations, and allows for querying the status of queued operations
- Operations can include host-device data transfer, kernel launches, and most other commands that are issued by the host but handled by the device
- Execution of an operation in a stream is always asynchronous with respect to the host
- CUDA runtime will determine when that operation is eligible for execution on the device
- By using multiple streams to launch multiple simultaneous kernels, can implement **grid level concurrency**

# Streams and Concurrency



- Because all operations queued in a CUDA stream are asynchronous, it is possible to overlap their execution with other operations in the host-device system
- Typical pattern in CUDA programming has been:
  1. Move input data from the host to the device.
  2. Execute a kernel on the device.
  3. Move the result from the device back to the host
- More time is spent executing the kernel than transferring data
- Completely hide CPU-GPU communication latency- By dispatching kernel execution and data transfer into separate streams, these operations can be overlapped, and the total elapsed time of the program can be shortened
- Streams can be used to implement pipelining or double buffering at the granularity of CUDA API calls

# Streams and Concurrency



- CUDA API can generally be classified as either **synchronous or asynchronous**
- Functions with synchronous behavior **block the host** thread until they complete
- Functions with asynchronous behavior **return control to the host** immediately after being called
- Asynchronous functions and streams are the two basic pillars on which you build grid-level concurrency in CUDA
- While from a software point of view CUDA operations in different streams run concurrently; that may not always be the case on physical hardware
- Depending on PCIe bus contention or the availability of per-SM resources, different CUDA streams may still need to wait for each other in order to complete

# CUDA Streams

- All CUDA operations (both kernels and data transfers) either explicitly or implicitly run in a stream
  - There are two types of streams:
    - ▶ Implicitly declared stream (NULL stream)
    - ▶ Explicitly declared stream (non-NUL stream)
  - NULL stream is the default stream that kernel launches and data transfers use if you do not explicitly specify a stream
  - Non-null streams are explicitly created and managed- to overlap different CUDA operations, must use non-null streams
  - Asynchronous, stream-based kernel launches and data transfers enable the following types of coarse-grain concurrency:
    - ▶ Overlapped host computation and device computation
    - ▶ Overlapped host computation and host-device data transfer
    - ▶ Overlapped host-device data transfer and device computation
    - ▶ Concurrent device computation

# Default Stream

```
✓ cudaMemcpy( . . . , cudaMemcpyHostToDevice ) ;  
kernel<<<grid, block>>>( . . . ) ;  
cudaMemcpy( . . . , cudaMemcpyDeviceToHost ) ;
```

- Data transfers can also be issued asynchronously; explicitly set the CUDA stream to place them in. The CUDA runtime provides the following asynchronous version of cudaMemcpy:

```
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,  
cudaMemcpyKind kind, cudaStream_t stream = 0);
```

# Default Stream

```
cudaMemcpy(..., cudaMemcpyHostToDevice);  
kernel<<<grid, block>>>(...);  
cudaMemcpy(..., cudaMemcpyDeviceToHost);
```

- Data transfers can also be issued asynchronously; explicitly set the CUDA stream to place them in. The CUDA runtime provides the following asynchronous version of cudaMemcpy:

```
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,  
    cudaMemcpyKind kind, cudaStream_t stream = 0);
```

```
cudaError_t cudaStreamCreate(cudaStream_t* pStream);
```

# Default Stream



- Stream returned in pStream can then be used as the stream argument to `cudaMemcpyAsync` and other asynchronous CUDA API functions

```
cudaError_t cudaMallocHost(void **ptr, size_t size);  
cudaError_t cudaHostAlloc(void **pHost, size_t size, unsigned int flags);
```

- One common point of confusion when using asynchronous CUDA functions is that they may return error codes from previously launched asynchronous operations
- API call returning an error is not necessarily the call that caused the error
- `cudaStreamCreate` creates a non-null stream that you manage explicitly
- stream returned in pStream can then be used as the stream argument to `cudaMemcpyAsync` and other asynchronous CUDA API functions
- One common point of confusion when using asynchronous CUDA functions is that they may return error codes from previously launched asynchronous operations
- API call returning an error is not necessarily the call that caused the error

# Non-default Stream

- To launch a kernel in a non-default stream, you must provide a stream identifier as the fourth parameter in the kernel execution configuration:

```
kernel_name<<<grid, block, sharedMemSize, stream>>>(argument list);
```

- Non-default stream is declared as follows

```
cudaStream_t stream;
```

- Non-default streams can be created using:

```
cudaStreamCreate(&stream);
```

- Resources of a stream can be released using:

```
cudaError_t cudaStreamDestroy(cudaStream_t stream);
```

# Non-default Stream



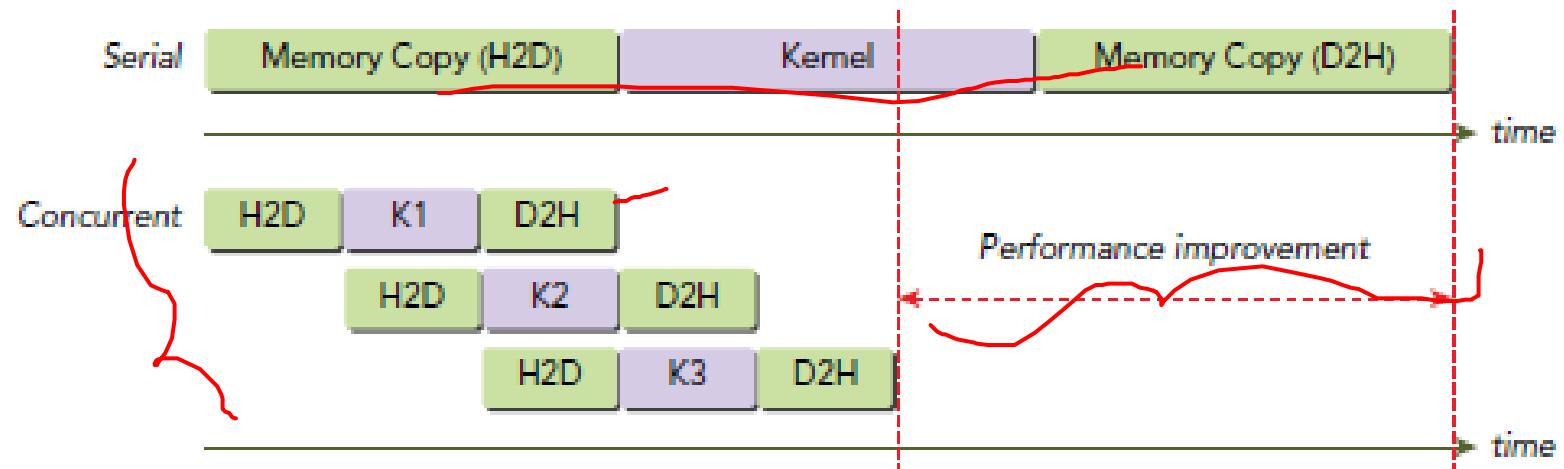
Pending work in a stream when `cudaStreamDestroy` is called on that stream, `cudaStreamDestroy` returns immediately and the resources associated with the stream are released automatically when all work in the stream has completed

All CUDA stream operations are asynchronous, the CUDA API provides two functions that check if all operations in a stream have completed

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);  
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

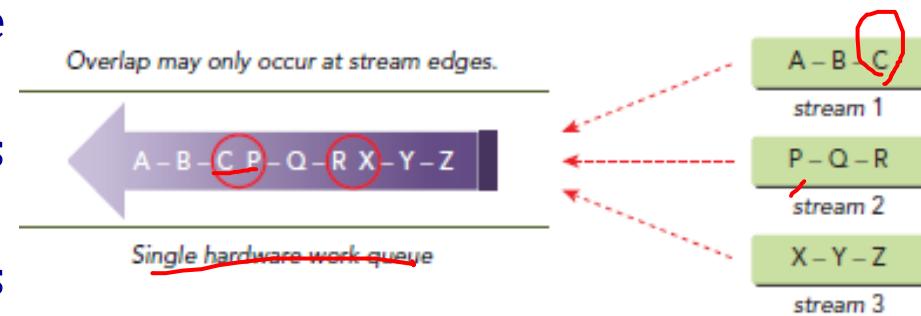
# Timeline of CUDA operations using three streams

```
for (int i = 0; i < nStreams; i++) {  
    int offset = i * bytesPerStream;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], bytePerStream, streams[i]);  
    kernel<<grid, block, 0, streams[i]>>(&d_a[offset]);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], bytesPerStream, streams[i]);  
}  
  
for (int i = 0; i < nStreams; i++) {  
    cudaStreamSynchronize(streams[i]);  
}
```



# Stream Scheduling

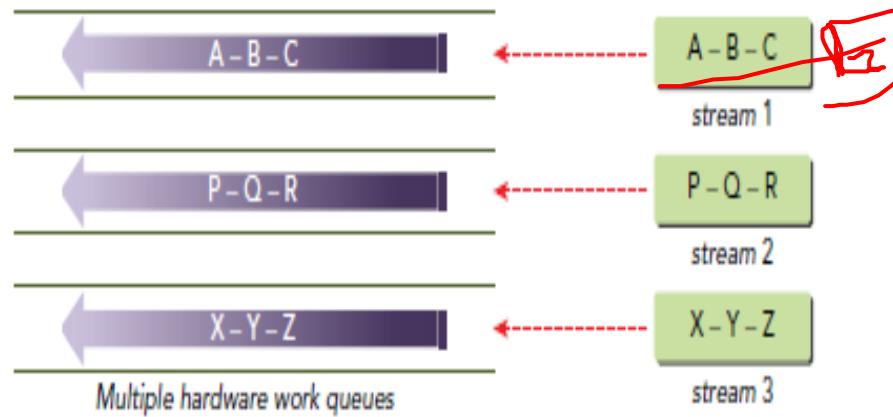
- All streams can run simultaneously
  - How concurrent kernel operations in multiple CUDA streams are scheduled by hardware?
- False Dependencies**
- All streams are ultimately multiplexed into a single hardware work queue
  - When selecting a grid to execute, the task at the front of the queue is scheduled by the CUDA runtime
  - Runtime checks for task dependencies, and waits for any tasks that this task depends on to complete if they are still executing
  - Finally, when all dependencies are satisfied the new task is dispatched to available SMs
  - Single pipeline may result in a false dependency
  - Circled task pairs will eventually be executed concurrently because the runtime will block before launching every other grid
  - Blocked operation in the queue blocks all subsequent operations in the queue, even when they belong to different streams



# Stream Scheduling

## Hyper-Q

- False dependencies are reduced in the Kepler family of GPUs using multiple hardware work queues, a technology called Hyper-Q
- Hyper-Q allows multiple CPU threads or processes to launch work on a single GPU simultaneously by maintaining multiple hardware-managed connections between the host and the device
- Existing applications that were limited by Fermi's false dependencies can see a dramatic performance increase without changing any existing code
- Kepler GPUs use 32 hardware work queues and allocate one work queue per stream
- If more than 32 streams are created, multiple streams will share a single hardware work queue
- Result is full stream-level concurrency with minimal false inter-stream dependencies

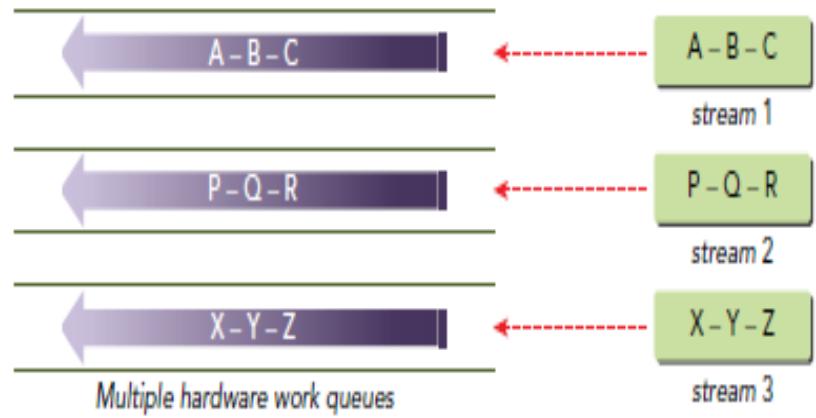


Three streams on three hardware work queues

# Stream Priorities

## Hyper-Q

- False dependencies are reduced in the Kepler family of GPUs using multiple hardware work queues, a technology called Hyper-Q
- Hyper-Q allows multiple CPU threads or processes to launch work on a single GPU simultaneously by maintaining multiple hardware-managed connections between the host and the device
- Existing applications that were limited by Fermi's false dependencies can see a dramatic performance increase without changing any existing code
- Kepler GPUs use 32 hardware work queues and allocate one work queue per stream
- If more than 32 streams are created, multiple streams will share a single hardware work queue
- Result is full stream-level concurrency with minimal false inter-stream dependencies



Three streams on three hardware work queues

# Stream Priorities

- For devices with compute capability 3.5 or higher, streams can be assigned priorities. A stream is created with a specific priority using the following function:  
`cudaError_t cudaStreamCreateWithPriority (cudaStream_t*pStream, unsigned int flags, int priority);`
- Function creates a stream with the specified integer priority and returns a handle in pStream
- Priority is associated with the work scheduled in pStream
- Grids queued to a higher priority stream may preempt work already executing in a low priority stream
- Stream priorities have no effect on data transfer operations, only on compute kernels
  - Allowable range of priorities for a given device can be queried using the following function:  
`cudaError_t cudaDeviceGetStreamPriorityRange(int *leastPriority, int *greatestPriority);`
  - Function returns values in `leastPriority` and `greatestPriority` that correspond to the lowest and highest priorities for the current device
  - By convention, lower integer values indicate a higher stream priority
  - `cudaDeviceGetStreamPriorityRange` returns zero in both parameters if the current device does not support stream priorities

- An event in CUDA is essentially a marker in a CUDA stream associated with a certain point in the flow of operations in that stream.
- Use events to perform the following two basic tasks:
  - ▶ Synchronize stream execution
  - ▶ Monitor device progress
- CUDA API provides functions that allow you to insert events at any point in a stream as well as query for event completion
- Event recorded on a given stream will only be satisfied (that is, complete) when all preceding operations in the same stream have completed
- Events specified on the default stream apply to all preceding operations in all CUDA streams

- Creation and Destruction An event is declared as follows:

cudaEvent\_t event;

- Event can be created using:

cudaError\_t cudaEventCreate(cudaEvent\_t\* event);

- Event can be destroyed using:

cudaError\_t cudaEventDestroy(cudaEvent\_t event);

- Event is queued to a CUDA stream using the following function:

cudaError\_t cudaEventRecord(cudaEvent\_t event, cudaStream\_t stream = 0);

- Event is queued to a CUDA stream using the following function:

cudaError\_t cudaEventRecord(cudaEvent\_t event, cudaStream\_t stream = 0);

- Test if an event has completed without blocking the host application using:

```
cudaError_t cudaEventQuery(cudaEvent_t event);
```

cudaEventQuery is similar to cudaStreamQuery, but for events

- Measure the elapsed time of CUDA operations marked by two events using the following function:

```
cudaError_t cudaEventElapsedTime(float* ms, cudaEvent_t start, cudaEvent_t stop);
```

# CUDA Events

The following code sample illustrates how events are typically used to time device operations:

```
// create two events
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// record start event on the default stream
cudaEventRecord(start);

// execute kernel
kernel<<<grid, block>>>(arguments);

// record stop event on the default stream
cudaEventRecord(stop);

// wait until the stop event completes
cudaEventSynchronize(stop);

// calculate the elapsed time between two events
float time;
cudaEventElapsedTime(&time, start, stop);

// clean up the two events
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

# Stream Synchronization



- All operations in non-default streams are non-blocking with respect to the host thread, will run across situations where you need to synchronize the host with operations running in a stream
- From the host point-of-view, CUDA operations can be classified into two main categories:
  - Memory-related operations
  - Kernel launches Kernel launches are always asynchronous with respect to the host
- Many memory operations are inherently synchronous (such as cudaMemcpy), but the CUDA runtime also provides asynchronous functions for performing memory operations
- Two types of streams:
  - Asynchronous streams (non-NULL streams)
  - Synchronous streams (the NULL/default stream)
- Non-null stream is an asynchronous stream with respect to the host; all operations applied to it do not block host execution
- On the other hand, the NULL-stream, declared implicitly, is a synchronous stream with respect to the host

# Blocking and Non-Blocking Streams



- Non-NULL streams can be further classified into the following two types:
  - Blocking streams
  - Non-blocking streams

```
kernel_1<<<1, 1, 0, stream_1>>>(); ✓  
kernel_2<<<1, 1>>>(); ✓  
kernel_3<<<1, 1, 0, stream_2>>>(); ✓
```

CUDA runtime provides a function that allows customization of a non-NULL stream's behavior in relation to the NULL stream

```
cudaError_t cudaStreamCreateWithFlags(cudaStream_t* pStream, unsigned int flags);
```

# Blocking and Non-Blocking Streams

## Implicit Synchronization

- Implicit synchronization is of special interest in CUDA programming because runtime functions with implicit synchronization behavior may cause unwanted blocking, usually at the device level
- Many memory-related operations imply blocking on all previous operations on the current device, such as:
  - A page-locked host memory allocation
  - A device memory allocation
  - A device memset
  - A memory copy between two addresses on the same device
  - A modification to the L1/shared memory configuration

# Blocking and Non-Blocking Streams

## Explicit Synchronization

CUDA runtime supports several ways of explicitly synchronizing a CUDA program at the grid level:

- Synchronizing the device
- Synchronizing a stream
- Synchronizing an event in a stream
- Synchronizing across streams using an event

Block a host thread until the device has completed all preceding tasks with the following function:

```
cudaError_t cudaDeviceSynchronize(void);
```

# Configurable Events



- CUDA runtime provides a way to customize the behavior and properties of events:

```
cudaError_t cudaEventCreateWithFlags(cudaEvent_t* event, unsigned int flags);
```

- Valid flags include:

- ✓ `cudaEventDefault`
- ✓ `cudaEventBlockingSync`- specifies that synchronizing on this event with `cudaEventSynchronize` will block the calling thread
- ✓ `cudaEventDisableTiming` - Passing `cudaEventDisableTiming` indicates that the created event is only used for synchronization and does not need to record timing data
- ✓ `cudaEventInterprocess` -e flag `cudaEventInterprocess` indicates that the created event may be used as an inter-process event.

# Blocking and Non-Blocking Streams

- Block the host thread until all operations in a stream complete using `cudaStreamSynchronize`, or perform a non-blocking test for completion using

`cudaStreamQuery: cudaError_t cudaStreamSynchronize(cudaStream_t stream);`

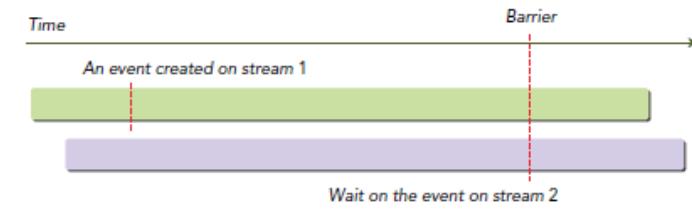
`cudaError_t cudaStreamQuery(cudaStream_t stream);`

- CUDA events can also be used for fine-grain blocking and synchronization using `cudaEventSynchronize` and `cudaEventQuery`:

`cudaError_t cudaEventSynchronize(cudaEvent_t event);`

`cudaError_t cudaEventQuery(cudaEvent_t event);`

- `cudaStreamWaitEvent` offers a flexible way to introduce inter-stream dependencies using CUDA events:  
`cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);`



# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## CUDA Instructions

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

# Introducing CUDA Instructions



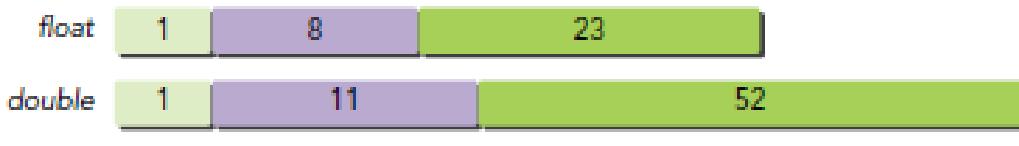
- Instructions generated for a CUDA kernel:
  - ✓ Floating-point operations
  - ✓ Intrinsic and standard functions
  - ✓ Atomic operations
- Floating-point calculations operate on non-integral values and affect both the accuracy and performance of CUDA programs
- Intrinsic and standard functions implement overlapping sets of mathematical operations but offer different guarantees of accuracy and performance
- Atomic instructions guarantee correctness when concurrently performing operations on a variable from multiple threads

# Floating-Point Instructions

- Binary floating-point data is encoded in three fields: A one-bit sign field, multiple exponent bits, and multiple bits encoding the significand (or fraction)



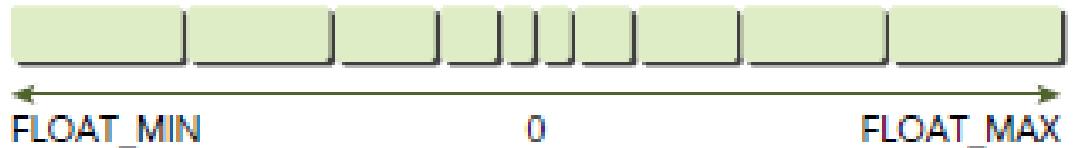
- IEEE-754 defines 32- and 64-bit floating-point formats, which correspond to the C data types float and double



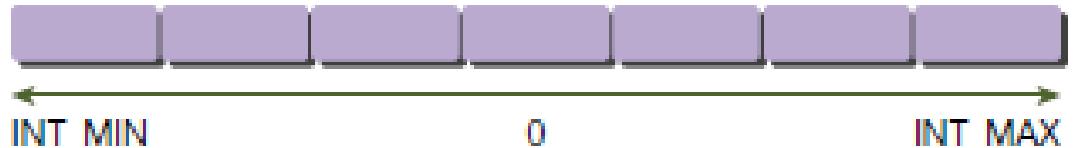
```
float a = 3.1415927f;
float b = 3.1415928f;
if (a == b) {
    printf("a is equal to b\n");
} else {
    printf("a does not equal b\n");
}
```

# Floating-Point Instructions

Floating-point granularity



Integer granularity



# Floating-Point Instructions

- IEEE 754 support two tiers of accuracy in floating-point: 32-bit and 64-bit
- Different formats are also referred to as single-precision and double-precision, respectively
- Because double-precision variables use twice as many bits as single-precision variables, double-precision values can correctly represent many more values
- Set of double-precision values has both a finer granularity and wider range than single-precision values
- Floating-point accuracy example that uses double- instead of single-precision floats:

```
double a = 3.1415927;  
double b = 3.1415928;  
if (a == b) {  
    printf("a is equal to b\n");  
} else {  
    printf("a does not equal b\n");  
}
```

compute-capable NVIDIA GPUs support single-precision floating-point, you will need an NVIDIA GPU of compute capability 1.3 or higher to use double-precision values

This code produces the expected output:

a does not equal b

- nearest representable values for a and b are not the same when stored using doubleprecision variables

# Intrinsic and Standard Functions



- CUDA also categorizes all arithmetic functions as either intrinsic or standard functions
- Standard functions are used to support operations that are accessible from, and standardized across, the host and device
- Standard functions include arithmetic operations from the C standard math library such as `sqrt`, `exp`, and `sin`
- Single-instruction operations like multiplication and addition are also included as standard functions

# CUDA intrinsic functions



- CUDA intrinsic functions can only be accessed from device code
- Function being intrinsic, or built-in, implies that the compiler has special knowledge about its behavior, which enables more aggressive optimization and specialized instruction generation
- This is true for CUDA intrinsic functions
- Many trigonometric functions are directly implemented in hardware on GPUs because they are used heavily in graphics applications
- Standard function for performing a double-precision floating-point square root is `sqrt`
- Intrinsic version implementing the same functionality is `_dsqrt_rn`

# CUDA intrinsic functions



- Intrinsic functions decompose into fewer instructions than their equivalent standard functions
- Intrinsic functions are faster than their equivalent standard functions but less numerically precise
- Gives the capability to use standard and intrinsic functions interchangeably, but produce different program behavior in terms of both performance and numerical accuracy
- Standard and intrinsic functions add a significant amount of flexibility to any CUDA application
- Serve as fine-grained knobs that you can turn to tweak performance and numerical accuracy on an operation-by-operation basis

# Atomic Instructions

- An atomic instruction performs a mathematical operation, but does so in a single uninterrupted operation with no interference from other threads
- When a thread successfully completes an atomic operation on a variable, it can be certain that the variable's state change has completed no matter how many other threads are accessing that variable
- Because atomic instructions prevent multiple threads from interfering with each other, they enable read-modify-write operations for data shared across threads (for example, read the current value, increment it, and write the new value)
- Guaranteeing the atomicity of read-modify-write operations is especially important in highly concurrent environments, like the GPU
- CUDA provides atomic functions that perform read-modify-write atomic operations on 32-bits or 64-bits of global memory or shared memory
- While any device with compute capability 1.1 or higher supports atomic operations, Kepler-based global atomic memory operations are faster than Fermi-based operations, leading to dramatically higher throughput

# Atomic Instructions

- Like standard and intrinsic functions, each atomic function implements a basic mathematical operation such as addition, multiplication, or subtraction
- Unlike any other instruction type described so far, atomic instructions have a defined behavior when operating on a memory location shared by two competing threads
- Problem here is caused by more than one thread writing to the same memory location is called a **data race**, or **unsafe access** to memory
- Data race is formally defined as two or more independent threads of execution accessing the same location, where at least one of those accesses is modifying that location

```
__global__ void incr(int *ptr) {
    int temp = *ptr;
    temp = temp + 1;
    *ptr = temp;
}
```



```
__global__ void incr(__global__ int *ptr) {
    int temp = atomicAdd(ptr, 1);
}
```



# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## Unit –II Parallel Processing Terminology

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



## UNIT - II

**15 Periods**

**Parallel Processing:** Introduction - Parallel Processing Terminology - The Sieve of Eratosthenes - **PRAM Algorithms:** Parallel Reduction - Prefix sums - List Ranking - Pre-order Tree Traversal - Merging of two sorted Lists - Graph coloring - **Matrix Multiplication:** Algorithms for processor Arrays - **Sorting:** Enumeration sort - Odd Even transposition sort- Parallel Quick sort - Hyper quick sort

# Parallel processing



## Parallel Processing:

- Concurrent manipulation of data elements belonging to one or more processes solving a single problem.

## Parallel Computer:

- Multiple Processor computer
- Capable of Parallel processing

## Supercomputer:

- Solving individual problems @ high computational speed compared with other computer.
- having small no. of extremely powerful processors + large no. of microprocessors.

# Parallel processing

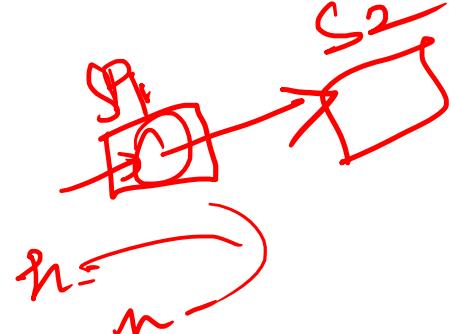
- Throughput:
  - No. of results produced/unit time.

To improve throughput:

- Increase Speed
- Concurrency (increase no. of operations/ unit time)

Pipelining

Data Parallelism



- use of multiple functional units to apply the same operation simultaneously to elements of a data set.

Computation divided into no. of steps  $\Rightarrow$  Segments or Stages.

- **Speedup:**

Execution Time taken by Sequential Computation

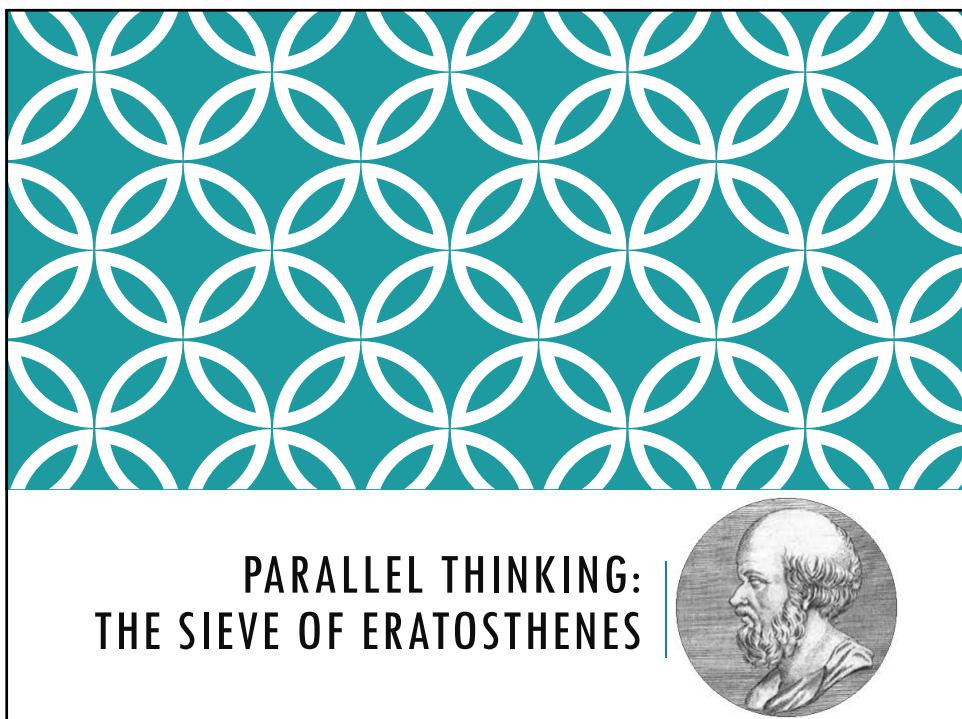
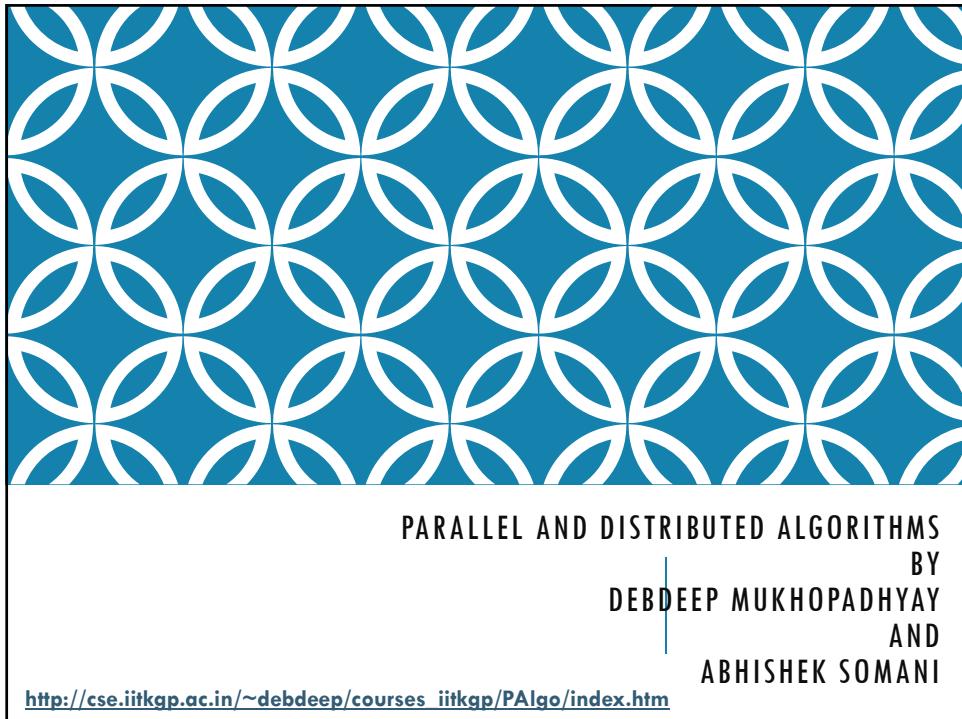
Speedup = -----

Execution Time taken by Pipelined or Parallel M/c

- **Contrasting Pipelining and Data Parallelism:**
- There are 3 steps involved to assemble a widget.
- For each step, it requires 1 second(s)

### Sequential:

- **2 widgets completed in 6 seconds.**



## THE SIEVE OF ERATOSTHENES

Classic prime finding algorithm:

- Want to find the number of primes less than or equal to some positive integer  $n$ .
- A prime has exactly two factors: itself and one.
- The Sieve of Eratosthenes begins with a list of natural numbers  $2, 3, 4, \dots, n$ , and removes composite numbers from the list by striking multiples of  $2, 3, 5$ , and successive primes. The sieve terminates after multiples of the largest prime less than or equal to  $\sqrt{n}$  have been struck.

- Prime is next unmarked natural number - ie. 2
- Strike all multiples of 2, starting with  $2^2$
- Prime is next unmarked natural number - ie. 3
- Strike all multiples of 3, starting with  $3^2$
- Prime is next unmarked natural number - ie. 5
- Strike all multiples of 5, starting with  $5^2$
- Prime is next unmarked natural number - ie. 7. Since  $7^2$  is greater than  $n=30$ , algorithm terminates. All unmarked natural numbers are also prime.

(a)	<table border="1"><tr><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td></tr></table>	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30		
(b)	<table border="1"><tr><td>2</td><td>3</td><td><del>4</del></td><td><del>5</del></td><td><del>6</del></td><td><del>7</del></td><td><del>8</del></td><td><del>9</del></td><td><del>10</del></td><td><del>11</del></td><td><del>12</del></td><td><del>13</del></td><td><del>14</del></td><td><del>15</del></td><td><del>16</del></td><td><del>17</del></td><td><del>18</del></td><td><del>19</del></td><td><del>20</del></td><td><del>21</del></td><td><del>22</del></td><td><del>23</del></td><td><del>24</del></td><td><del>25</del></td><td><del>26</del></td><td><del>27</del></td><td><del>28</del></td><td><del>29</del></td><td><del>30</del></td></tr></table>	2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>
2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>		
(c)	<table border="1"><tr><td>2</td><td>3</td><td><del>4</del></td><td><del>5</del></td><td><del>6</del></td><td><del>7</del></td><td><del>8</del></td><td><del>9</del></td><td><del>10</del></td><td><del>11</del></td><td><del>12</del></td><td><del>13</del></td><td><del>14</del></td><td><del>15</del></td><td><del>16</del></td><td><del>17</del></td><td><del>18</del></td><td><del>19</del></td><td><del>20</del></td><td><del>21</del></td><td><del>22</del></td><td><del>23</del></td><td><del>24</del></td><td><del>25</del></td><td><del>26</del></td><td><del>27</del></td><td><del>28</del></td><td><del>29</del></td><td><del>30</del></td></tr></table>	2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>
2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>		
(d)	<table border="1"><tr><td>2</td><td>3</td><td><del>4</del></td><td><del>5</del></td><td><del>6</del></td><td><del>7</del></td><td><del>8</del></td><td><del>9</del></td><td><del>10</del></td><td><del>11</del></td><td><del>12</del></td><td><del>13</del></td><td><del>14</del></td><td><del>15</del></td><td><del>16</del></td><td><del>17</del></td><td><del>18</del></td><td><del>19</del></td><td><del>20</del></td><td><del>21</del></td><td><del>22</del></td><td><del>23</del></td><td><del>24</del></td><td><del>25</del></td><td><del>26</del></td><td><del>27</del></td><td><del>28</del></td><td><del>29</del></td><td><del>30</del></td></tr></table>	2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>
2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>		
(e)	<table border="1"><tr><td>2</td><td>3</td><td><del>4</del></td><td><del>5</del></td><td><del>6</del></td><td><del>7</del></td><td><del>8</del></td><td><del>9</del></td><td><del>10</del></td><td><del>11</del></td><td><del>12</del></td><td><del>13</del></td><td><del>14</del></td><td><del>15</del></td><td><del>16</del></td><td><del>17</del></td><td><del>18</del></td><td><del>19</del></td><td><del>20</del></td><td><del>21</del></td><td><del>22</del></td><td><del>23</del></td><td><del>24</del></td><td><del>25</del></td><td><del>26</del></td><td><del>27</del></td><td><del>28</del></td><td><del>29</del></td><td><del>30</del></td></tr></table>	2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>
2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>		
(f)	<table border="1"><tr><td>2</td><td>3</td><td><del>4</del></td><td><del>5</del></td><td><del>6</del></td><td><del>7</del></td><td><del>8</del></td><td><del>9</del></td><td><del>10</del></td><td><del>11</del></td><td><del>12</del></td><td><del>13</del></td><td><del>14</del></td><td><del>15</del></td><td><del>16</del></td><td><del>17</del></td><td><del>18</del></td><td><del>19</del></td><td><del>20</del></td><td><del>21</del></td><td><del>22</del></td><td><del>23</del></td><td><del>24</del></td><td><del>25</del></td><td><del>26</del></td><td><del>27</del></td><td><del>28</del></td><td><del>29</del></td><td><del>30</del></td></tr></table>	2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>
2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>		
(g)	<table border="1"><tr><td>2</td><td>3</td><td><del>4</del></td><td><del>5</del></td><td><del>6</del></td><td><del>7</del></td><td><del>8</del></td><td><del>9</del></td><td><del>10</del></td><td><del>11</del></td><td><del>12</del></td><td><del>13</del></td><td><del>14</del></td><td><del>15</del></td><td><del>16</del></td><td><del>17</del></td><td><del>18</del></td><td><del>19</del></td><td><del>20</del></td><td><del>21</del></td><td><del>22</del></td><td><del>23</del></td><td><del>24</del></td><td><del>25</del></td><td><del>26</del></td><td><del>27</del></td><td><del>28</del></td><td><del>29</del></td><td><del>30</del></td></tr></table>	2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>
2	3	<del>4</del>	<del>5</del>	<del>6</del>	<del>7</del>	<del>8</del>	<del>9</del>	<del>10</del>	<del>11</del>	<del>12</del>	<del>13</del>	<del>14</del>	<del>15</del>	<del>16</del>	<del>17</del>	<del>18</del>	<del>19</del>	<del>20</del>	<del>21</del>	<del>22</del>	<del>23</del>	<del>24</del>	<del>25</del>	<del>26</del>	<del>27</del>	<del>28</del>	<del>29</del>	<del>30</del>		

3

## FEW POINTS ON THE SEQUENTIAL ALGORITHM

The Sieve of Eratosthenes is impractical for testing primality of numbers with hundreds of digits.

- The time complexity of the algorithm is  $\Omega(n)$ , and  $n$  increases exponentially with the number of digits.
- However modern sieving techniques use the sieving techniques through other suitable manipulations.

A sequential implementation of the Sieve of Eratosthenes manages 3 key data structures:

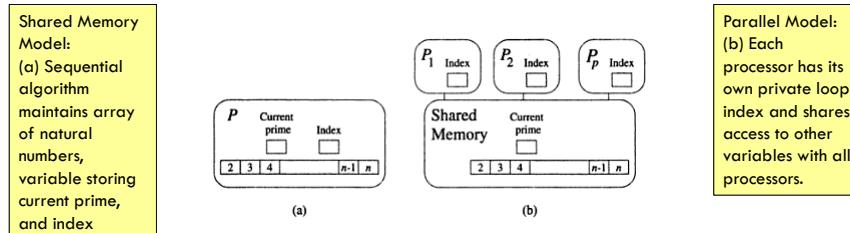
- An array whose elements correspond to the natural numbers being sieved.
- An integer corresponding to latest prime number found.
- An integer used as a loop index, incremented as multiples of the latest (current) prime are marked as composite.

4

## SHARED MEMORY MODEL FOR PARALLEL ERATOSTHENES ALGORITHM

### Control Parallel Approach:

- Every processor goes through the two step process of finding the next prime number
- Striking from the list multiples of that prime, beginning with its square.
- The processors continue until a prime is found whose value is greater than  $\sqrt{n}$



5

## INEFFICIENCIES

### Two processors may asynchronously end up using the same prime to sieve.

- The first processor will access the value of the current prime and start sieving with it.
- The second processor will start from the next unmarked cell, which it updates as the current prime.
- If another processor starts before this update then it also starts sieving with the same prime.

### Also a processor may end up sieving with composite numbers!

- The first processor starts sieving with multiples of 2.
- Before it marks any cell, a second processor starts sieving with the next prime, which is 3.
- A third processor starts sieving with the next unmarked cell, which is 4 (and has not been marked yet by the first processor)!

Our implementations hence needs to ensure that these time wasting situations do not happen.

6

## ESTIMATING THE MAX SPEEDUP

Assumptions:

1. The above situations do not occur.
2. Ignore the time spent finding the next prime, and concentrate on the operations of marking the cells.

First analyze the sequential algorithm:

7

## ESTIMATING THE MAX SPEEDUP

Assume it takes one unit of time for a processor to mark a cell.

Suppose there are  $k$  primes less than or equal to  $\sqrt{n}$

Denote them by  $\pi_1, \pi_2, \dots, \pi_k$ . Thus,  $\pi_1 = 2, \pi_2 = 3, \pi_3 = 5, \dots$

The total time required by a single processor is:

$$\left\lceil \frac{(n+1)-\pi_1^2}{\pi_1} \right\rceil + \left\lceil \frac{(n+1)-\pi_2^2}{\pi_2} \right\rceil + \left\lceil \frac{(n+1)-\pi_3^2}{\pi_3} \right\rceil + \dots + \left\lceil \frac{(n+1)-\pi_k^2}{\pi_k} \right\rceil = \\ \left\lceil \frac{n-3}{2} \right\rceil + \left\lceil \frac{n-8}{3} \right\rceil + \left\lceil \frac{n-24}{5} \right\rceil + \dots + \left\lceil \frac{(n+1)-\pi_k^2}{\pi_k} \right\rceil$$

For  $n=1000$ , the sum is 1,411.

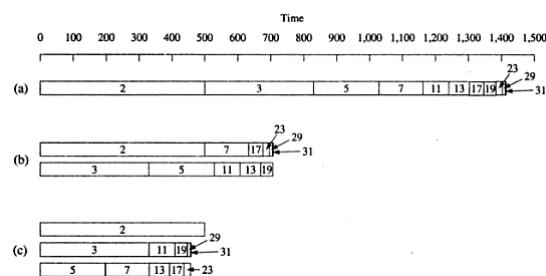
8

## TIME TAKEN BY THE PARALLEL ALGORITHM

Time reduction with addition of processors ( $n=1000$ ):

- a) Single Processor strikes out all composite numbers in 1,411 units of time.
- b) Two processors reduce the execution time to 706 units of time. This corresponds to a speedup of  $1411/706=2$
- c) Three processors reduce the time to 499 time units, which leads to speedup of 2.83.

**Note adding more processors does not help here, because with more than 2 processors the time required to sieve all multiples of 2 determine the parallel execution time.**



9

## DATA PARALLEL APPROACH

Let us consider another approach.

In this case, the approach is data parallel: that is different processor elements perform the same operation on different data sets.

- Each processor will be responsible for a segment of the array representing the natural numbers.
- All the processors perform the same operation (ie. strikes off multiples of the same prime) on its own segment of data.

Analyzing the speedup is straight-forward and is left as an exercise.

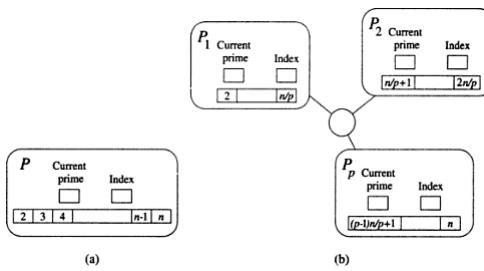
10

## MODEL WITH NO SHARED MEMORY: MESSAGE PASSING PARADIGM

Consider a different model for parallel computing:

- There is no shared memory
- Processors interact by message passing

**Shared Memory Model:**  
 (a) Sequential algorithm maintains array of natural numbers, variable storing current prime, and index



**Parallel Model:**  
 (b) Each processor has its own copy of the variables containing the current prime and the loop index.  
 Processor 1 finds prime and communicates them to other processors.  
 Each processor iterates through its own portion of the array.

Assume the number of processors  $p << \sqrt{n}$ . Thus the list controlled by the first processor has all primes less than  $\sqrt{n}$  and the first prime greater than  $\sqrt{n}$ . Termination of the program happens when processor 1 reaches a prime greater than  $\sqrt{n}$

11

## ANALYSIS

We need to consider the time spent communicating the value of the current prime from processor 1 to all other processors.

Assume it takes  $\chi$  time units for a processor to mark a multiple of a prime as being a composite number.

Suppose there are  $k$  primes as before, less than or equal to  $\sqrt{n}$ .

### Computation Time:

The total time a processor spends striking out composite numbers is:

$$\left( \left\lceil \frac{\pi(2)}{2} \right\rceil + \left\lceil \frac{\pi(3)}{3} \right\rceil + \left\lceil \frac{\pi(5)}{5} \right\rceil + \dots + \left\lceil \frac{\pi(p)}{p} \right\rceil \right) \chi$$

**Communication Time:** Assume each time processor 1 finds a new prime it communicates the value to each of the  $(p-1)$  processors in turn.

If processor 1 spends  $\lambda$  amount of time it passes a number to another process, total communication time for  $k$  primes is  $k(p-1)\lambda$ .

12

## ANALYSIS FOR N=1,000,000

It turns out there are 168 primes less than 1,000 (square root of  $10^6$ ).

The largest is 997.

Therefore maximum computation time:

$$\left( \left\lceil \frac{\left\lceil \frac{(1,000,000)}{p} \right\rceil}{2} \right\rceil + \left\lceil \frac{\left\lceil \frac{(1,000,000)}{p} \right\rceil}{3} \right\rceil + \left\lceil \frac{\left\lceil \frac{(1,000,000)}{p} \right\rceil}{5} \right\rceil + \dots + \left\lceil \frac{\left\lceil \frac{(1,000,000)}{p} \right\rceil}{997} \right\rceil \right) \chi$$

Total Communication Time:  $168(p-1)\lambda$

Assume  $\lambda = 100\chi$  and lets plot the speedup.

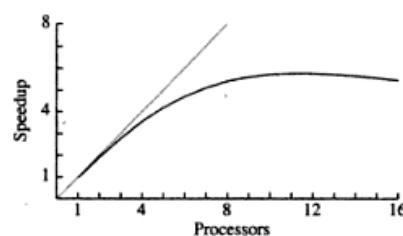
13

## ESTIMATED SPEEDUP

Note that speedup is not directly proportional to the number of processors used.

Speedup is highest at 11 processors.

Why does the decline in speedup happen?



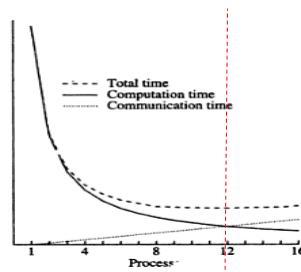
14

## COMPUTATION TIME, COMMUNICATION TIME AND PROCESSORS

Computation time is inversely proportional with the number of processors.

Communication time increases linearly with the number of processors.

After 11 processors, increase in communication time is greater than the decrease in computation time.

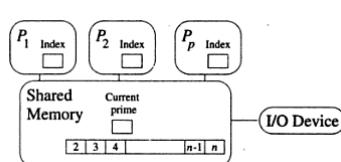


15

## DATA PARALLEL APPROACH WITH I/O

The algorithms also need to store and print their results before termination.

Let us consider the data parallel implementation of the sieving method with an output on the shared memory model for parallel computation.

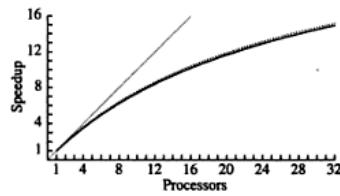


Let  $i\beta$  denote the time required for a processor to transmit  $i$  prime numbers to that device.

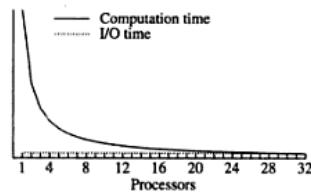
There are 78,498 primes less than 1,000,000.  
Thus the time for the I/O is  $78,498\beta$ .

16

## SPEEDUP ANALYSIS



Assuming,  $\beta = \chi$  we plot the speedup.  
The plot shows the variation of speedup for 1,2, ..., 32 processors.  
There is a damping effect on the speedup.  
**This is because the output to the I/O device must be performed sequentially.**



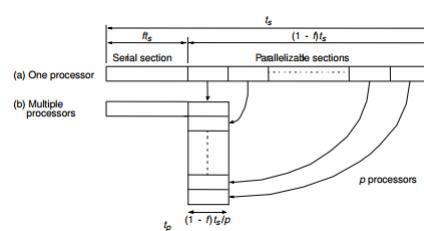
**I/O time is a part of the operation which does not depend on the number of processors**

17

## AMDAHL'S LAW

Let,  $f$  be the fraction of operations in a computation that must be performed sequentially, where  $0 \leq f \leq 1$

Maximum speedup  $S$  achievable by a parallel computer with  $p$  processors is: 
$$\frac{1}{f + (1-f)/p}$$



18

## APPLYING AMDAHL'S LAW

When  $n=1,000,000$  the sequential algorithm marks 2,122,048 cells and outputs 78,498 primes.

Assuming both these operations take same amount of time, total time required is  $2,122,048 + 78,498 = 2,200,546$ .

Thus,  $f = 78,498 / 2,200,546 = 0.0357$ .

Thus, the upper bound on the speedup with  $p$  processors is:

$$\frac{1}{0.0357 + 0.9643/p}$$

The dotted curve in the speedup plot, shows this upper bound for different values of  $p$ .

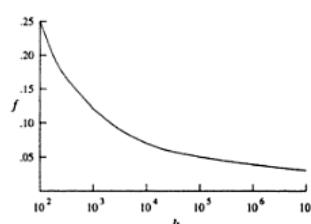
19

## AMDAHL EFFECT

As the size of the problem increases, the fraction  $f$  of inherently sequential operation decreases.

- This phenomenon is called as Amdahl Effect.

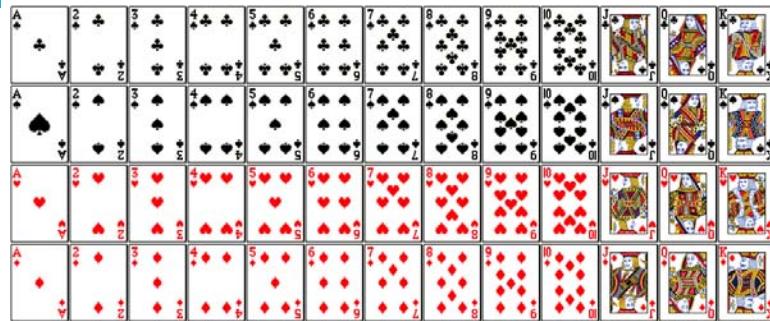
**An ameliorating fact: This makes the problem more amenable for parallelization.**



Plot of  $f$  with  $n$  for the data-parallel sieve algorithm with output, assuming  $\beta = \chi$ .

20

## QUESTION



Shuffle a deck of cards and then determine how long it takes to sort the cards as above. Assume it is faster to sort the cards initially by suit, and then by an insertion sort to arrange each suit.

1. How long does it take for  $p$  people to sort  $p$  decks of shuffled cards?
2. How long does it take  $p$  people to sort one deck of cards?

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **Unit –II PRAM Algorithm**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

## UNIT - II

**15 Periods**

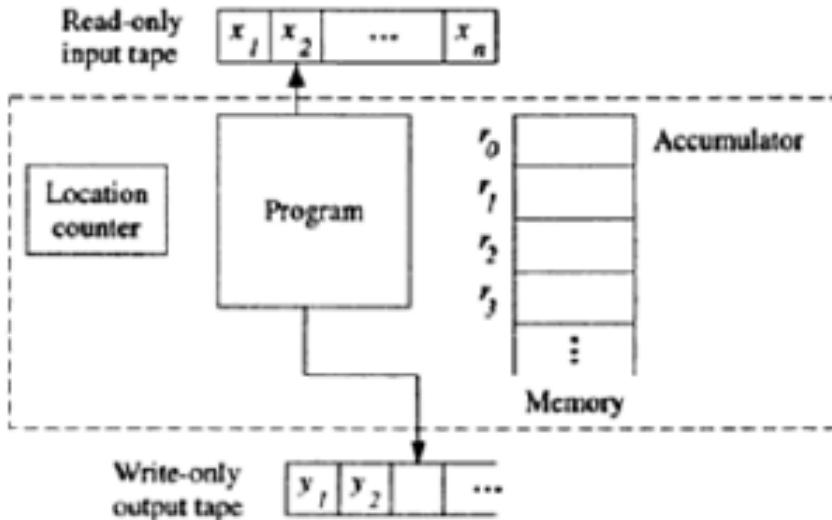
**Parallel Processing:** Introduction - Parallel Processing Terminology - The Sieve of Eratosthenes - **PRAM Algorithms:** Parallel Reduction - Prefix sums - List Ranking - Pre-order Tree Traversal - Merging of two sorted Lists - Graph coloring - **Matrix Multiplication:** Algorithms for processor Arrays - **Sorting:** Enumeration sort - Odd Even transposition sort- Parallel Quick sort - Hyper quick sort

# RAM: A MODEL OF SERIAL COMPUTATION

Random Access Machine (RAM) is a model of a **one-address computer**

- ✓ Consists of a memory
- ✓ Read-only input tape
- ✓ Write-only output tape
- ✓ Program

Input tape consists of a sequence of integers.  
Every time an input value is read, the input head advances one square.  
Likewise, the output head advances after every write.



Memory consists of unbounded set of registers,  $r_0, r_1, \dots$

Each register holds a single integer.

Register  $r_0$  is the accumulator, where computations are performed.

# COST MODELS

**Uniform Cost Criterion:** each RAM instruction requires one unit of time to execute. Every register requires one unit of space.

**Logarithmic Cost Criterion:** Assumes that every instruction takes a logarithmic number of time units (wrt. the length of the operands), and that every register requires a logarithmic number of units of space.

Thus, uniform cost criteria count the number of operations and logarithmic cost criteria count the number of bit operations.

The uniform cost criterion is applicable if the values manipulated by a program always fit into one computer word.

Consider an 8 bit adder.

In the uniform cost criteria to analyze the run time of the adder, we would say that the adder takes 1 unit of time, ie.  $T(N)=1$ .

However, in the logarithmic model you would consider that the 1's position bits are added, followed by the 2's position bits, and so on. In this model, thus there are 8 smaller additions (for every bit positions) and each requires a unit of time. Thus,  $T(N)=8$ . Generalizing,  $T(N)=\log(N)$ .

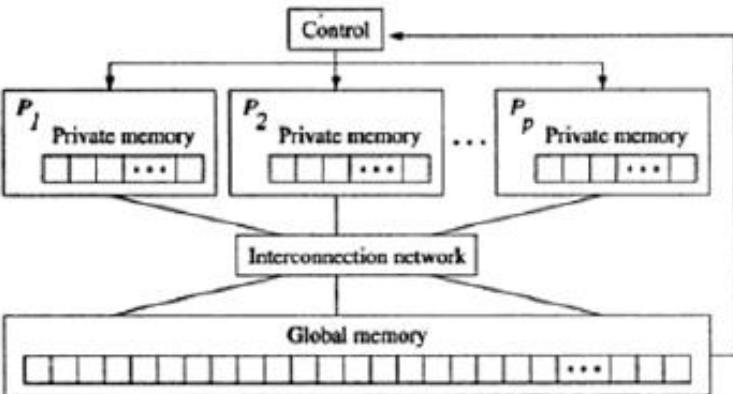
# TIME COMPLEXITIES IN THE RAM MODEL



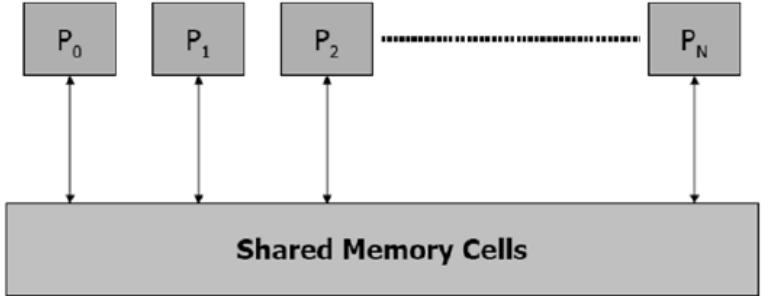
- **Worst case time complexity:** The function  $f(n)$ , the maximum time taken by the program to execute over all inputs of size  $n$ .
- **Expected time complexity:** It is the average time over the execution times for all inputs of size  $n$ .
- Analogous definitions hold for the space complexities (just replace the time word by space)

# THE PRAM MODEL

- A PRAM consists of a control unit, global memory, an unbounded set of processors, each with its own private memory
- Active processors execute identical instructions
- Every processor has a unique index, and the value can be used to enable or disable the processor, or influence which memory locations it accesses



# THE PRAM MODEL



**Cost of a PRAM computation** is the product of the parallel time complexity and the number of processors used. For example, a PRAM algorithm that has time complexity  $\Theta(\log p)$  using  $p$  processors has cost  $\Theta(p \log p)$ .

- All processing elements (PE) execute synchronously the same algorithm and work on distinct memory areas.
- Neither the number of PEs nor the size of memory is bounded.
- Any PE can access any memory location in one unit of time.
  - The last two assumptions are unrealistic!

# PRAM COMPUTATION STEPS



- A PRAM computation starts with the input stored in global memory and a single active processing element
- During each step of the computation an active, enabled processor may read a value from a single private or global memory location, perform a single RAM operation, and write into one local or global memory location
- Alternatively, during a computation step a processor may activate another processor
- All active, enabled processors must execute the same instruction, albeit on different memory locations
- Computation terminates when the last processor halts

# PRAM MODELS

- The models differ in how they handle read or write conflicts, ie. when two or more processors attempt to read from or write to the same global memory location.
- 1. **EREW (Exclusive Read Exclusive Write)** Read or write conflicts are not allowed.
- 2. **CREW (Concurrent Read Exclusive Write)** Concurrent reading allowed, ie. Multiple processors may read from the same global memory location during the same instruction step. Write conflicts are not allowed.
  - 1. During a given time, ie. During a given step of an algorithm, arbitrarily many PEs can read the value of a cell simultaneously while at most one PE can write a value into a cell.
- 3. **CRCW (Concurrent Read Concurrent Write):** Concurrent reading and writing are allowed.
- A variety of CRCW models exist with different policies for handling concurrent writes to the same global address:
  - 1. **Common:** All processors concurrently writing into the same global address must be writing the same value.
  - 2. **Arbitrary:** If multiple processors concurrently write to the same global address, one of the competing processors is arbitrarily chosen as the winner, and its value is written.
  - 3. **Priority:** The processor with the lowest index succeeds in writing its value.

The EREW model is the weakest.

- A CREW PRAM can execute any EREW PRAM algorithm in the same time. This is obvious, as the concurrent read facility is not used.
- Similarly, a CRCW PRAM can execute any EREW PRAM algorithm in the same amount of time.

The PRIORITY PRAM model is the strongest.

- Any algorithm designed for the COMMON PRAM model will execute in the same time complexity in the ARBITRARY or PRIORITY PRAM models.
  - If the processors writing to the same location write the same value choosing an arbitrary processor would cause the same result.
  - Likewise, it also produces the same result when the processor with the lowest index is chosen the winner.

Because the PRIORITY PRAM model is stronger than the EREW PRAM model, an algorithm to solve a problem on the EREW PRAM can have higher time complexity than an algorithm solving the same problem on the PRIORITY PRAM model.

# CSE409 - PARALLEL & DISTRIBUTED SYSTEMS

## Unit –II Parallel Reduction & Prefix Sum

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

## UNIT - II

**15 Periods**

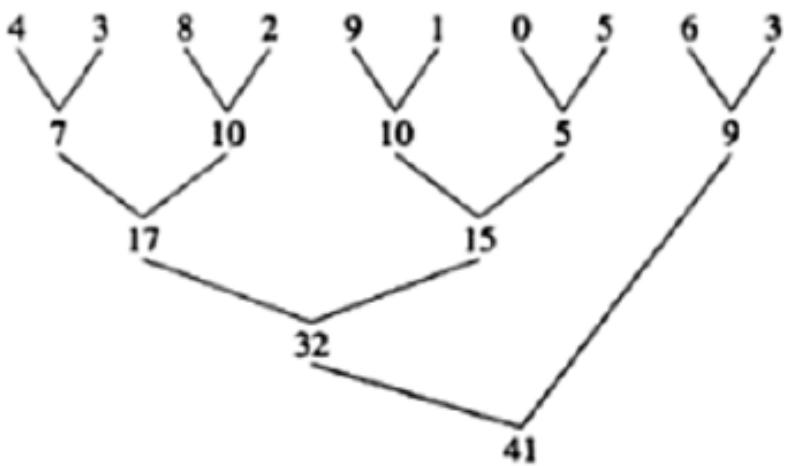
**Parallel Processing:** Introduction - Parallel Processing Terminology - The Sieve of Eratosthenes - **PRAM Algorithms:** Parallel Reduction - Prefix sums - List Ranking - Pre-order Tree Traversal - Merging of two sorted Lists - Graph coloring - **Matrix Multiplication:** Algorithms for processor Arrays - **Sorting:** Enumeration sort - Odd Even transposition sort- Parallel Quick sort - Hyper quick sort

# PARALLEL REDUCTION



- Binary tree is one of the most important paradigms of parallel computing
- In the algorithms - an inverted binary tree
  - Data flows from the leaves to the root
- Fan-in or reduction operations
- More formally, given a set of n values  $a_1, a_2, \dots, a_n$  and an associative binary operator  $\oplus$ , reduction is the process of computing  $a_1 \oplus a_2 \oplus \dots \oplus a_n$ .
- Parallel Sum is an example of a reduction operation

# PARALLEL SUMMATION IS AN EXAMPLE OF REDUCTION



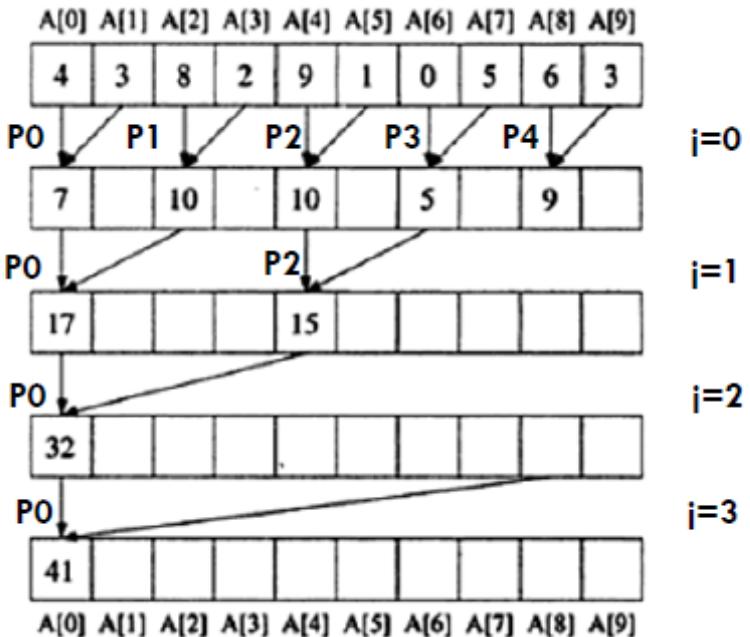
How do we write the PRAM algorithm for doing this summation?

# GLOBAL ARRAY BASED EXECUTION

The processors in a PRAM algorithm manipulate data stored in global registers.

For adding n numbers we spawn  $\lfloor \left(\frac{n}{2}\right) \rfloor$  processors.

Consider the example to generalize the algorithm.



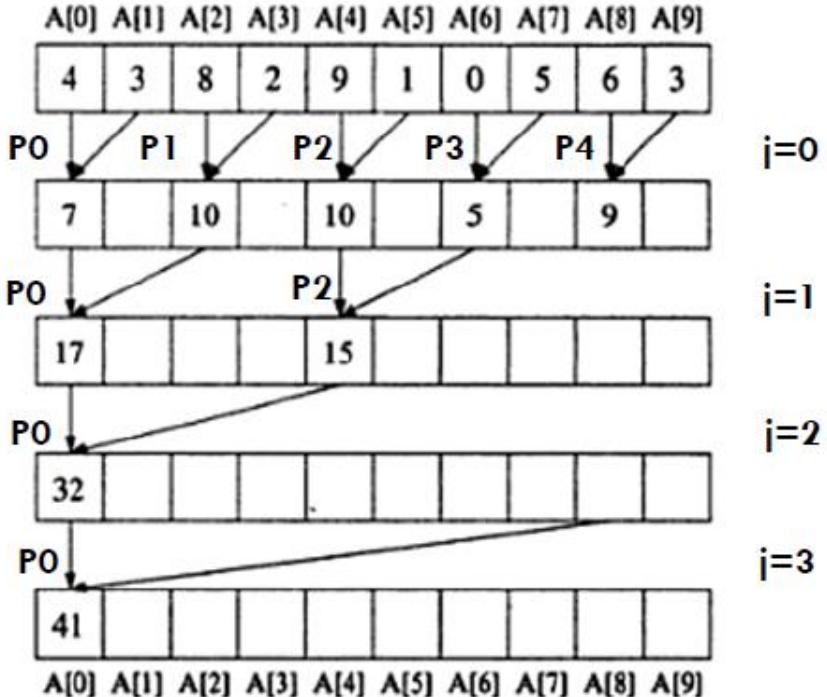
# GLOBAL ARRAY BASED EXECUTION

Each addition corresponds to:

$$A[2i] + A[2i+2^i].$$

Note, the processor which is active has an  $i$  such that:  
 $i \bmod 2^i = 0$  (ie. keep only those processors active).

Also check that the array does not go out of bound.  
▪ ie,  $2i + 2^i < n$



EREW PRAM algorithm to sum  $n$  elements using  $\lfloor n/2 \rfloor$  processors.

### SUM (EREW PRAM)

Initial condition: List of  $n \geq 1$  elements stored in  $A[0 \dots (n - 1)]$

Final condition: Sum of elements stored in  $A[0]$

Global variables:  $n$ ,  $A[0 \dots (n - 1)]$ ,  $j$

begin

```
spawn ( $P_0, P_1, P_2, \dots, P_{\lfloor n/2 \rfloor - 1}$ )
for all  $P_i$  where  $0 \leq i \leq \lfloor n/2 \rfloor - 1$  do
    for  $j \leftarrow 0$  to  $\lceil \lg n \rceil - 1$  do
        if  $i$  modulo  $2^j = 0$  and  $2i + 2^j < n$  then
             $A[2i] \leftarrow A[2i] + A[2i + 2^j]$ 
        endif
    endfor
endfor
end
```

# COMPLEXITY

The SPAWN routine requires  $\lceil \log \left\lfloor \frac{n}{2} \right\rfloor \rceil$  doubling steps.

The sequential for loop executes  $\lceil \log n \rceil$  times.

- Each iteration takes constant time.

Hence overall time complexity is  $\Theta(\log n)$  given  $n/2$  processors.

# PREFIX SUM



Given a set of  $n$  values  $a_1, a_2, \dots, a_n$ , and an associative operation  $\oplus$ , the prefix sum problem is to calculate the  $n$  quantities:

$$a_1,$$

$$a_1 \oplus a_2,$$

...

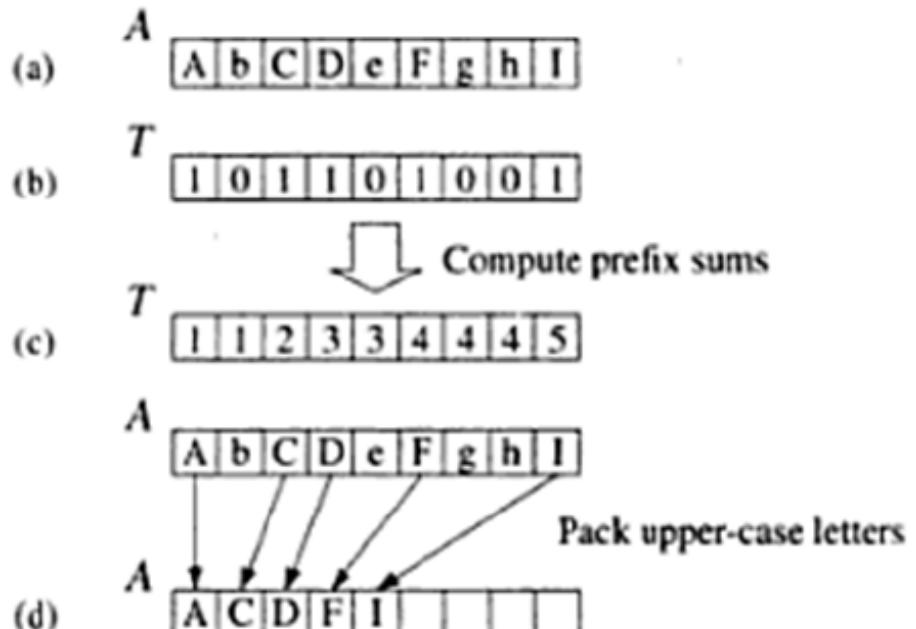
$$a_1 \oplus a_2 \oplus \dots \oplus a_n$$

# AN APPLICATION OF PREFIX SUM



We are given an array A of n letters. We want to pack the uppercase letters in the initial portion of A while maintaining their order. The lower case letters are deleted.

- a) Array A contains both uppercase and lowercase letters. We want to pack uppercase letters into beginning of A.
- b) Array T contains a 1 for every uppercase letter, and 0 for lowercase.
- c) Array T after prefix sum. For every element of A containing an uppercase letter, the corresponding element of T is the element's index in the packed array.
- d) Array A after packing.

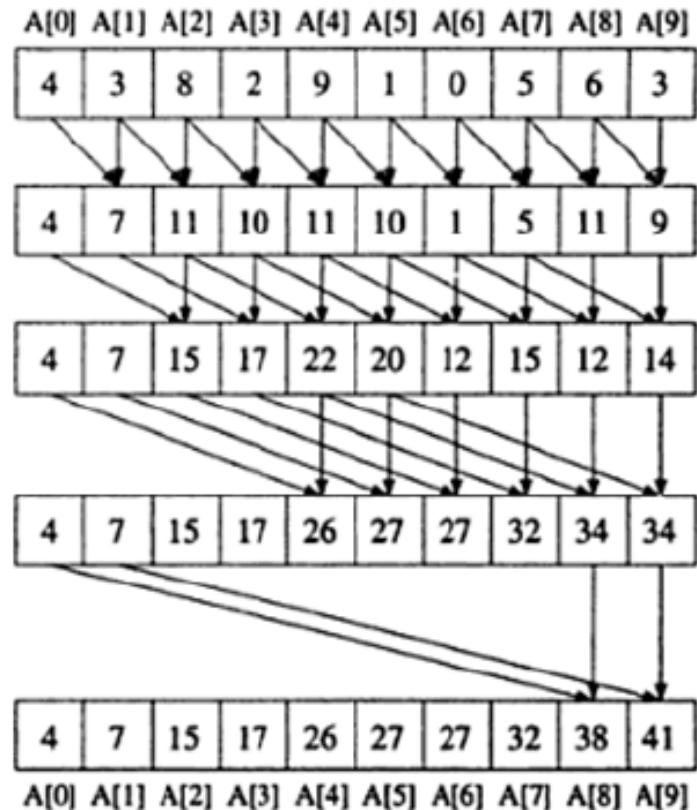


# GLOBAL ARRAY BASED EXECUTION IN EREW

There are  $n-1$  processors activated.

Each one accesses  $A[i]$ , then accesses  $A[i-2^j]$ , where  $j$  is the depth ( $j$  varies from 0 to  $\lceil \log n \rceil - 1$ ).

Of course, the bounds need to be checked.



# THE PRAM PSEUDOCODE



## PREFIX.SUMS (CREW PRAM):

Initial condition: List of  $n \geq 1$  elements stored in  $A[0 \dots (n - 1)]$

Final condition: Each element  $A[i]$  contains  $A[0] \oplus A[1] \oplus \dots \oplus A[i]$

Global variables:  $n$ ,  $A[0 \dots (n - 1)]$ ,  $j$

```
begin
    spawn ( $P_1, P_2, \dots, P_{n-1}$ )
    for all  $P_i$  where  $1 \leq i \leq n - 1$  do
        for  $j \leftarrow 0$  to  $\lceil \log n \rceil - 1$  do
            if  $i - 2^j \geq 0$  then
                 $A[i] \leftarrow A[i] + A[i - 2^j]$ 
            endif
        endfor
    endfor
end
```

# COMPLEXITY



Running time is  $t(n) = O(\lg n)$

Cost is  $c(n) = p(n) \times t(n) = O(n \lg n)$

Note not cost optimal, as RAM takes  $O(n)$

# COMPLEXITY



Running time is  $t(n) = O(\lg n)$

Cost is  $c(n) = p(n) \times t(n) = O(n \lg n)$

Note not cost optimal, as RAM takes  $O(n)$

# Prefix sum tracing

P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>	P <sub>8</sub>	P <sub>9</sub>	T <sub>n</sub>
<u>j=0</u> $i-j \geq 0$ $A[i] = A[0] + A[1]$	$i-j \geq 0$ $A[i] = A[0] + A[1] + A[2]$	$i-j \geq 0$ $A[i] = A[0] + A[1] + A[2] + A[3]$	$i-j \geq 0$ $A[i] = A[0] + A[1] + A[2] + A[3] + A[4]$	$i-j \geq 0$ $A[i] = A[0] + A[1] + A[2] + A[3] + A[4] + A[5]$	$i-j \geq 0$ $A[i] = A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6]$	-	-	-	$8+3+7+9+1+9$
<u>j=1</u> $i-j \neq 0$ $A[1] < A[0]$	$i-j \geq 0$ $A[1] < A[0] + A[2]$	$i-j \geq 0$ $3 \leftarrow 3+1$	$i-j \geq 0$ $4 \leftarrow 4+2, 5 \leftarrow 5+1$	$i-j \geq 0$ $6 \leftarrow 6+4, 7 \leftarrow 7+5$	$i-j \geq 0$ $8 \leftarrow 8+6$	$i-j \geq 0$ $9 \leftarrow 9+7$	-	-	-
<u>j=2</u> $i-j \neq 0$ $A[2] < A[0]$	$i-j \neq 0$ $A[2] < A[0] + A[1]$	$i-j \neq 0$ $3 \leftarrow 3+1$	$i-j \geq 0$ $4 \leftarrow 4+0, 5 \leftarrow 5+1$	$i-j \geq 0$ $6 \leftarrow 6+2, 7 \leftarrow 7+3$	$i-j \geq 0$ $8 \leftarrow 8+4$	$i-j \geq 0$ $9 \leftarrow 9+5$	-	-	-
<u>j=3</u> $i-j \neq 0$ $A[3] < A[0]$	$i-j \neq 0$ $A[3] < A[0] + A[1]$	$i-j \neq 0$ $A[3] < A[0] + A[1] + A[2]$	$i-j \neq 0$ $A[3] < A[0] + A[1] + A[2] + A[3]$	$i-j \neq 0$ $A[3] < A[0] + A[1] + A[2] + A[3] + A[4]$	$i-j \neq 0$ $A[3] < A[0] + A[1] + A[2] + A[3] + A[4] + A[5]$	$i-j \neq 0$ $A[3] < A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6]$	$i-j \neq 0$ $A[3] < A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7]$	$i-j \neq 0$ $A[3] < A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7] + A[8]$	$i-j \neq 0$ $8+8+0+9+1$

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **Unit –II**

### **List Ranking & Pre-order Tree Traversal**

**Dr. P. Padmakumari**

**CSE/SoC/SASTRA**



**PresenterMedia**

## UNIT - II

**15 Periods**

**Parallel Processing:** Introduction - Parallel Processing Terminology - The Sieve of Eratosthenes - **PRAM Algorithms:** Parallel Reduction - Prefix sums - List Ranking - Pre-order Tree Traversal - Merging of two sorted Lists - Graph coloring - **Matrix Multiplication:** Algorithms for processor Arrays - **Sorting:** Enumeration sort - Odd Even transposition sort- Parallel Quick sort - Hyper quick sort

Consider the problem of finding, for each element of  $n$  elements on a linked list, the suffix sums of the last  $i$  elements of the list, where

$$i \leq i \leq n.$$

The suffix sum problem is a variant of the prefix sum problem.

- Array is replaced by a linked list.
- Sums are computed from the end.

If the elements of the list are 0 or 1, and the associative operation is addition, the problem is called the list ranking problem.

One way to solve this is to traverse the list and count the number of links traversed between the list element and the end of the list.

Only a single pointer can be followed in one step, and there are  $n-1$  pointers between the first element and the end of the list.

- How can any algorithm traverse such a list in less than  $\Theta(n)$  time?

- The distance to the end of the list is cut in half through the instruction
  - :

$$next[i] \leftarrow next[next[i]]$$

Hence, a logarithmic number of pointer jumpings are sufficient to collapse the list so that every element points to the last list element.

If a processor adds to its own link traversal count,  $position[i]$ , the current link traversal count of the successors it encounters, the list position will be correctly determined.

# ILLUSTRATING THE PROCESS OF LIST RANKING



## List ranking problem

- Given a singly linked list L with n objects, for each node, compute the distance to the end of the list

If d denotes the distance

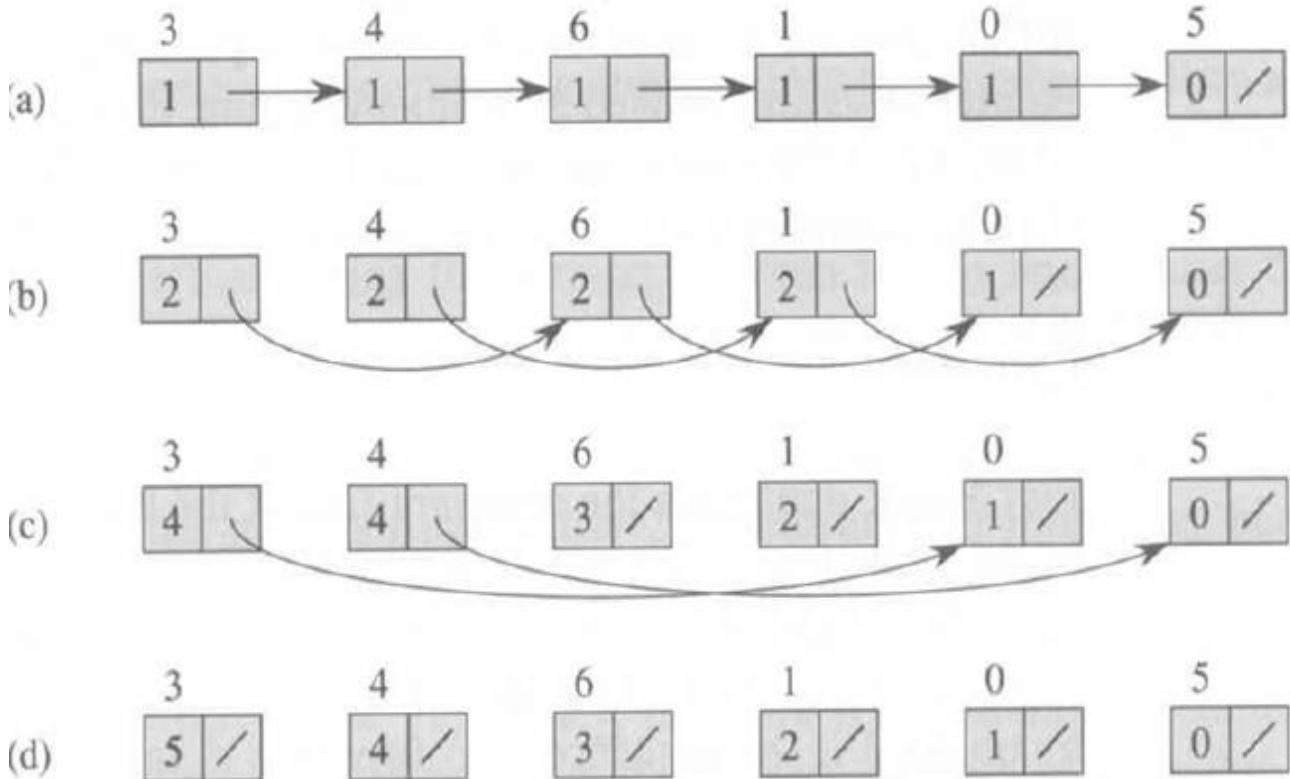
- $$\text{node.d} = \begin{cases} 0 & \text{if } \text{node.next} = \text{nil} \\ \text{node.next.d} + 1 & \text{otherwise} \end{cases}$$

Serial algorithm:  $O(n)$

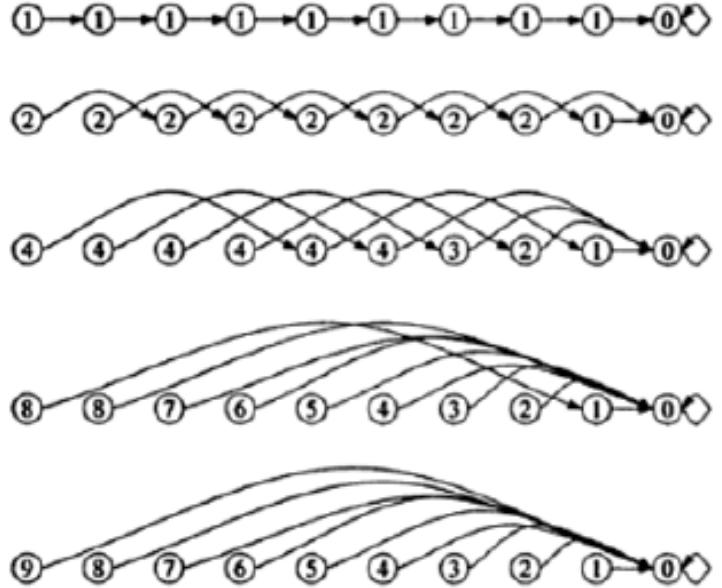
## Parallel algorithm

- Assign one processor for each node
- Assume there are as many processors as list objects
- For each node i, perform
  - $i.d = i.d + i.next.d$
  - $i.next = i.next.next$  // pointer jumping

# LIST RANKING – EXAMPLE 1



# LIST RANKING – EXAMPLE 2



**The position of each item on the  $n$ -element list can be determined in  $\lceil \log n \rceil$  pointer jumping steps.**

# PRAM ALGORITHM

PRAM algorithm to compute, for each element of a singly-linked list, its distance from the end of the list.

## LIST.RANKING (CREW PRAM):

Initial condition: Values in array *next* represent a linked list

Final condition: Values in array *position* contain original distance of each element from end of list

```
Global variables: n, position[0... $(n - 1)$ ], next[0... $(n - 1)$ ], j
begin
    spawn ( $P_0, P_1, P_2, \dots, P_{n-1}$ )
    for all  $P_i$  where  $0 \leq i \leq n - 1$  do
        if next[ $i$ ] =  $i$  then position[ $i$ ]  $\leftarrow 0$ 
        else position[ $i$ ]  $\leftarrow 1$ 
        endif
        for  $j \leftarrow 1$  to  $\lceil \log n \rceil$  do
            position[ $i$ ]  $\leftarrow$  position[ $i$ ] + position[next[ $i$ ]]
            next[ $i$ ]  $\leftarrow$  next[next[ $i$ ]]
        endfor
    endfor
end
```

Note this step does not depend on  $j$ .

There are  $\lceil \log n \rceil$  steps.

There are  $n$  processors.

So total cost is:

$$\Theta(n \log n)$$

Not cost optimal!

## Synchronization is important

- In step 6 ( $i \rightarrow next = i \rightarrow next \rightarrow next$ ), all processors must read right hand side before any processor write left hand side

## The list ranking algorithm is EREW

- If we assume in step 5 ( $i.d = i.d + i.next.d$ ) all processors read  $i.d$  and then read  $i.next.d$
- If  $j.next = i$ ,  $i$  and  $j$  do not read  $i.d$  concurrently

## Work performance

- performs  $O(n \log n)$  work since  $n$  processors in  $O(\log n)$  time

## Work efficient

- A PRAM algorithm is work efficient w.r.t another algorithm if two algorithms are within a constant factor
- Is the link ranking algorithm work-efficient w.r.t the serial algorithm?
  - No, because  $O(n \log n)$  versus  $O(n)$

## Speedup

- $S = n / \log n$

# PREORDER TREE TRAVERSAL



Sometimes it is appropriate to reduce a complicated looking problem into a simpler form for which a parallel algorithm is already known.

Let us consider **the problem of numbering the vertices of a rooted tree in preorder (depth first search order).**

At first glance this problem looks sequential!

# RECURSIVE PREORDER TRAVERSAL



```
PREORDER.TRAVERSAL(nodeptr):  
  
Begin  
  
    if nodeptr≠null then  
  
        nodecount ← nodecount + 1  
  
        nodeptr.label ← nodecount  
  
        PREORDER.TRAVERSAL(nodeptr.left)  
  
        PREORDER.TRAVERSAL(nodeptr.right)  
  
    endif  
  
End
```

Where is the parallelism?  
The fundamental operation  
assigns a label to a node.

We cannot assign labels to the  
vertices in the right subtree of  
the left subtree, until we know  
how many vertices are on the  
left subtree of the left subtree,  
and so on.

The algorithm seems inherently  
sequential!

Can we parallelize this?

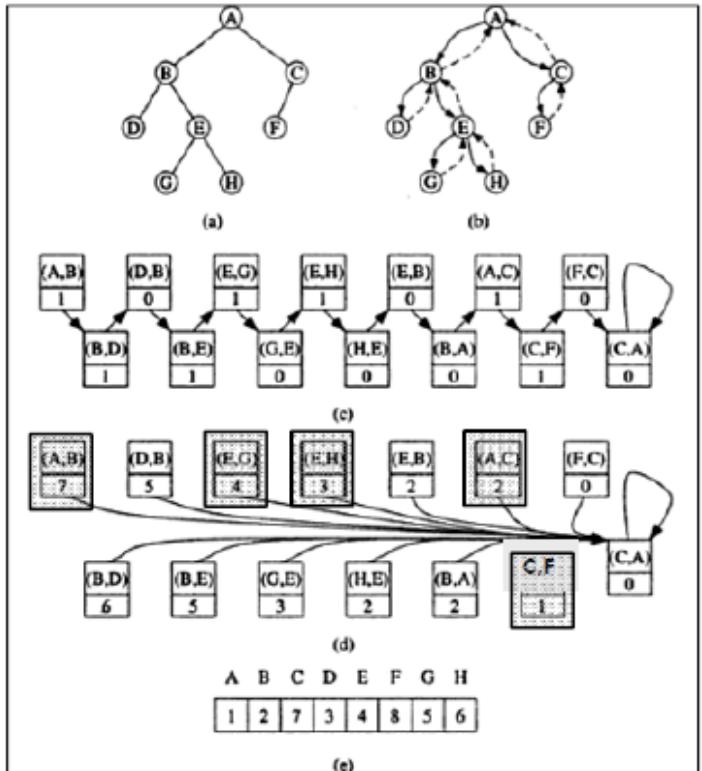
Instead of focusing on the vertices, let us look into the edges.

When we perform a preorder traversal, we systematically work our way through the edges of the tree.

- We pass along every vertex twice: one heading down from the parent to the child, and one going from the child to the parent.
- *If we divide each tree edge into two edges, one corresponding to the downward traversal, and one corresponding to the upward traversal, then the problem of traversing a tree turns into the problem of traversing a single linked list.*

# Example

## EXAMPLE



- a) Tree
- b) Double Tree Edges, distinguishing downward edges from upward edges.
- c) Build linked list out of directed tree edges. Associate 1 with downward edges, and 0 with upward edges.
- d) Use pointer jumping to compute total weight from each vertex to end of list.

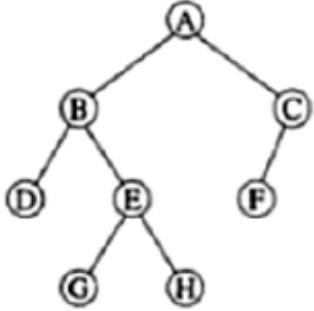
The elements of the linked list which correspond to downward edges, have been shaded.

Processors managing these elements assign preorder values.

For example, (E,G) has a weight 4, meaning tree node G is 4<sup>th</sup> node from end of preorder traversal list.

The tree has 8 nodes, so it can compute that tree node G has label 5 in preorder traversal ( $=8-4+1$ )

# DATA STRUCTURE FOR THE TREE



	A	B	C	D	E	F	G	H
parent	null	A	A	B	B	C	E	E
sibling	null	C	null	E	null	null	H	null
child	B	D	F	null	G	null	null	null

For every tree node, the data structure stores the node's parent, the node's immediate sibling to the right, and the node's leftmost child.

Representing the node this way keeps the amount of data stored a constant for each tree node and simplifies the tree traversal.

# PROCESSOR ALLOCATION



The PRAM algorithm spawns  $2(n-1)$  processors.

A tree with  $n$  nodes have  $(n-1)$  edges.

We are dividing each edge into two edges, one for the downward traversal and one for the upward traversal.

***So, the algorithm needs  $2(n-1)$  processors to manipulate each of the  $2(n-1)$  edges of the singly-linked list of elements corresponding to the edge traversals.***

# CONSTRUCTION OF THE LINKED LIST

Once all the processors have been activated they construct the linked list:

- $P(i,j)$ : The processor for the edge  $(i,j)$
- Note  $(j,i)$  has a different processor  $P(j,i)$

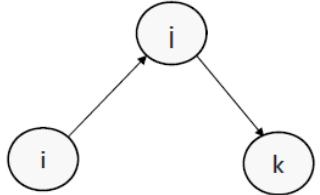
Given an edge  $(i,j)$ ,  $P(i,j)$  must compute the successor of  $(i,j)$  and store in a global array:  $\text{succ}[1 \dots 2(n-1)]$ .

- If the successor of  $(i,j)$  is  $(j,k)$ , then  $\text{succ}[(i,j)] \leftarrow (j,k)$

# HANDLING EDGES

## HANDLING UPWARD EDGES

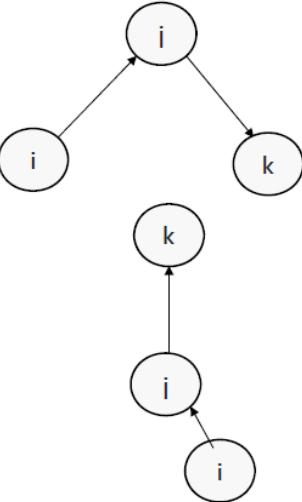
Edge  $(i,j)$ , such that  $\text{parent}(i)=j$



If  $\text{sibling}[i] \neq \text{NULL}$   
 $\text{succ}[(i,j)] \leftarrow (i, \text{sibling}[i])$

## HANDLING UPWARD EDGES

Edge  $(i,j)$ , such that  $\text{parent}(i)=j$



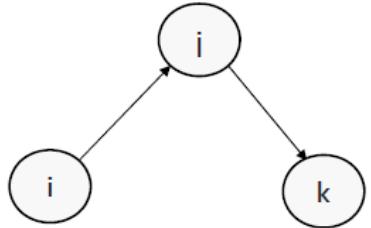
If  $\text{sibling}[i] \neq \text{NULL}$   
 $\text{succ}[(i,j)] \leftarrow (i, \text{sibling}[i])$

Else If  $\text{parent}[i] \neq \text{NULL}$   
 $\text{succ}[(i,j)] \leftarrow (i, \text{parent}[i])$

# CONSTRUCTION OF THE LINKED LIST

## HANDLING UPWARD EDGES

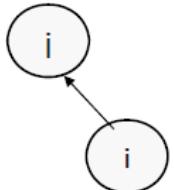
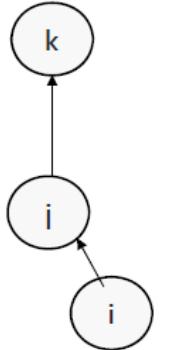
Edge  $(i,j)$ , such that  $\text{parent}(i)=j$



If  $\text{sibling}[i] \neq \text{NULL}$   
 $\text{succ}[(i,i)] \leftarrow (i,\text{sibling}[i])$

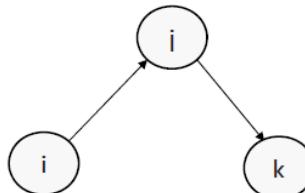
Else If  $\text{parent}[i] \neq \text{NULL}$   
 $\text{succ}[(i,i)] \leftarrow (i,\text{parent}[i])$

Else  
 $\text{succ}[(i,i)] \leftarrow (i,i)$   
The edge is at the end of  
the tree traversal, so we  
put a loop at the end of  
the element list.



## HANDLING UPWARD EDGES

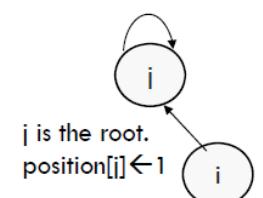
Edge  $(i,j)$ , such that  $\text{parent}(i)=j$



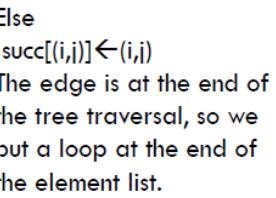
If  $\text{sibling}[i] \neq \text{NULL}$   
 $\text{succ}[(i,i)] \leftarrow (i,\text{sibling}[i])$

Else If  $\text{parent}[i] \neq \text{NULL}$   
 $\text{succ}[(i,i)] \leftarrow (i,\text{parent}[i])$

$\text{position}[1 \dots 2(n-1)]$   
is a global array to  
hold the edge ranks.



j is the root.  
 $\text{position}[i] \leftarrow 1$

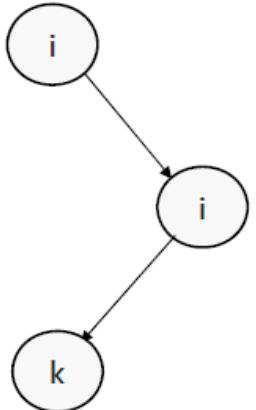


Else  
 $\text{succ}[(i,i)] \leftarrow (i,i)$   
The edge is at the end of  
the tree traversal, so we  
put a loop at the end of  
the element list.

# CONSTRUCTION OF THE LINKED LIST

## HANDLING DOWNWARD EDGES

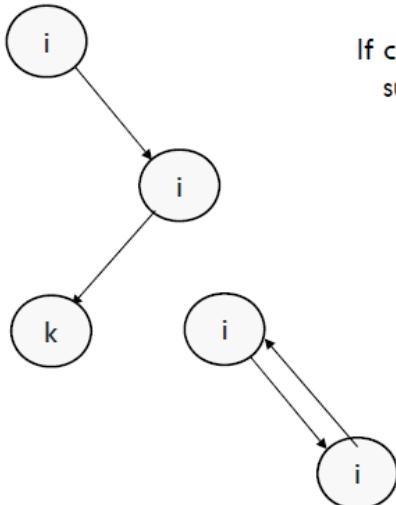
Edge  $(i,j)$ , such that  $\text{parent}[i] \neq j$ .



If  $\text{child}[j] \neq \text{NULL}$   
 $\text{succ}[(i,j)] \leftarrow (j, \text{child}[j])$

## HANDLING DOWNWARD EDGES

Edge  $(i,j)$ , such that  $\text{parent}[i] \neq j$ .



If  $\text{child}[j] \neq \text{NULL}$   
 $\text{succ}[(i,j)] \leftarrow (j, \text{child}[j])$

else  
 $\text{succ}[(i,j)] \leftarrow (j, i)$

i.e.  $j$  is a leaf and the successor is the edge back from the child to the parent.

## ASSIGNING EDGE RANKS

After the processors construct the list, they assign position values:

- 1 to those elements corresponding to downward edges
- 0 to those elements corresponding to upward edges.
- Note the root is already handled.

```
if parent[i]=j, position[(i,j)]←0
Else position[(i,j)]←1
```

# Assigning Edge Ranks

The pointer jumping follows subsequently to compute the suffix sum.

The final position values indicate the number of preorder traversal nodes between the list element and the end of the list.

To compute each node's preorder traversal label compute  $(n\text{-position}+1)$ .

# PRAM Program

## PRAM PROGRAM

PREORDER.TREE.TRAVERSAL (CREW PRAM):

```
Global n          {Number of vertices in tree}
    parent[1... n] {Vertex number of parent node}
    child[1... n]  {Vertex number of first child}
    sibling[1... n] {Vertex number of sibling}
    succ[1... (n - 1)] {Index of successor edge}
    position[1... (n - 1)] {Edge rank}
    preorder[1... n] {Preorder traversal number}

begin
    spawn (set of all  $P(i, j)$  where  $(i, j)$  is an edge)
    for all  $P(i, j)$  where  $(i, j)$  is an edge do
        {Put the edges into a linked list}
        if parent[i] = j then
            if sibling[i] ≠ null then
                succ[(i, j)] ← (j, sibling[i])
            else if parent[j] ≠ null then
                succ[(i, j)] ← (j, parent[j])
            else
                succ[(i, j)] ← (i, j)
                preorder[j] ← 1 {j is root of tree}
            endif
        else
            if child[j] ≠ null then succ[(i, j)] ← (j, child[j])
            else succ[(i, j)] ← (j, i)
            endif
        endif
    endfor
```

# PRAM Program

```
if parent[i] = j then position[(i, j)] ← 0
else position[(i, j)] ← 1
endif
{Perform suffix sum on successor list}
for k ← 1 to ⌈ log(2(n - 1))⌉ do
    position[(i, j)] ← position[(i, j)] + position[succ[(i, j)]]
    succ[(i, j)] ← succ[succ[(i, j)]]]
endfor
{Assign preorder values}
if i = parent[j] then preorder[j] ← n + 1 - position[(i, j)]
endif
endfor
end
```

Time Complexity: $O(\lceil \log(n) \rceil)$

Processors:  $O(n)$

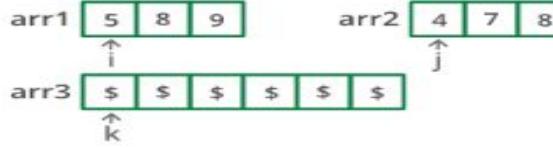
Cost:  $O(n \log n)$

# *PRAM Algorithm*

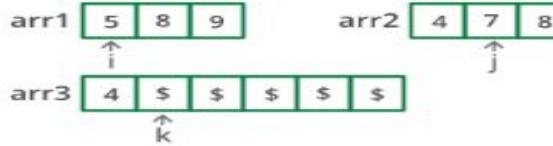
## *Merging of Two sorted Lists*

# Sequential Algorithm

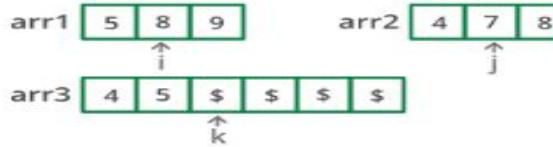
Initially:



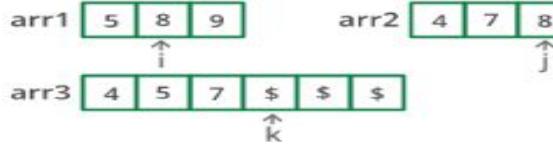
Step 1:



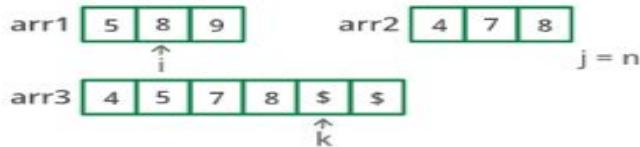
Step 2:



Step 3:

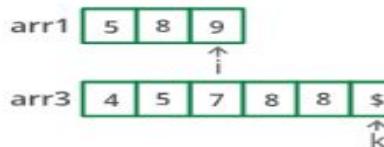


Step 4:

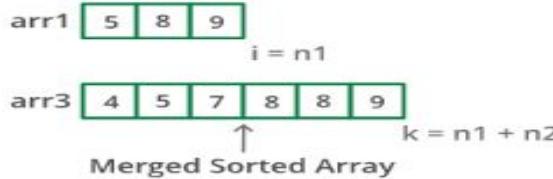


The first while loop breaks  
Second while loop copies all elements from arr1 to arr3

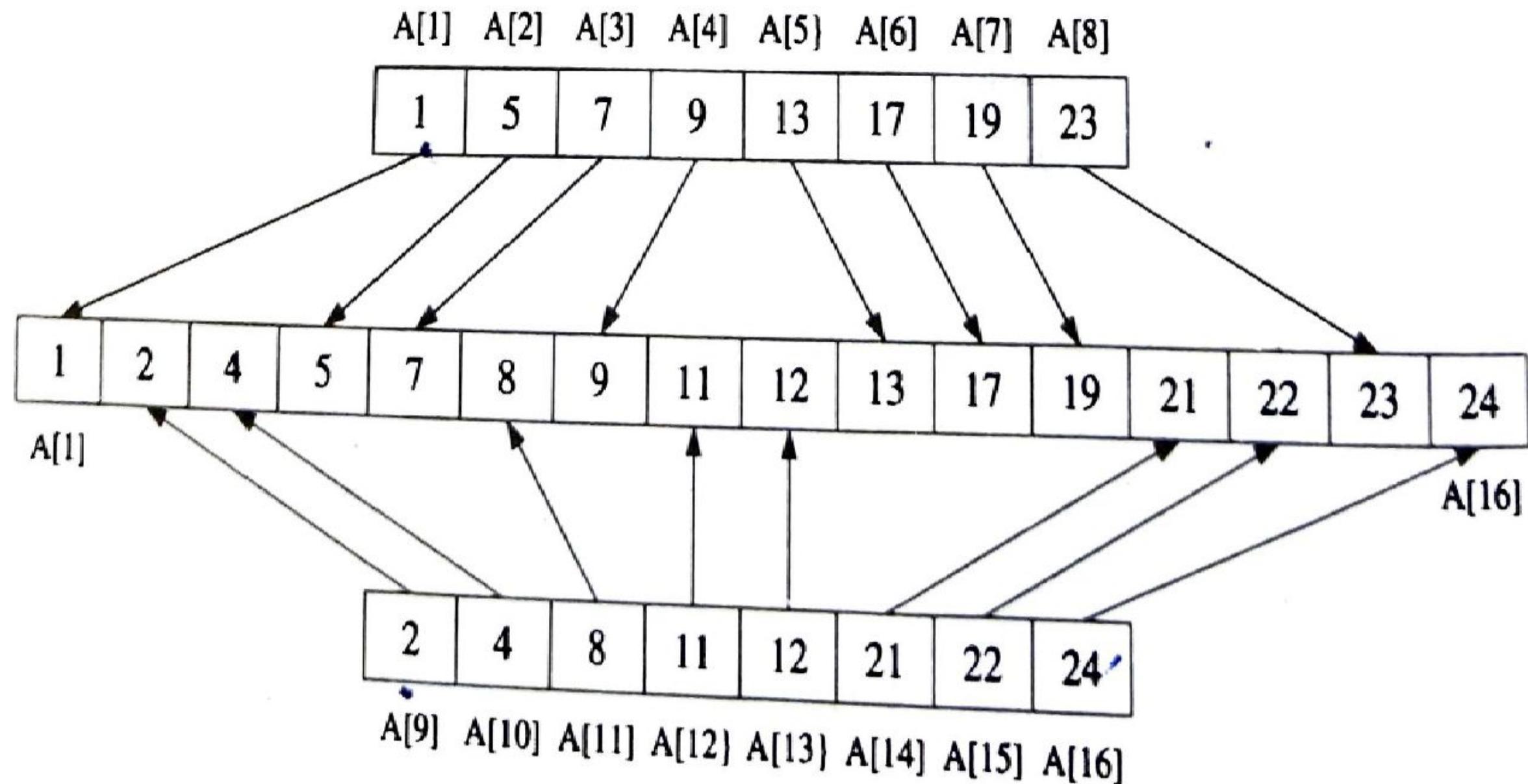
Step 5:



Step 6:



Two lists having  $n/2$  elements each can be merged in  $\Theta(\log n)$  time.



Given: Two sorted lists of  $n/2$  elements each, stored in  
 $A[1] \dots A[n/2]$  and  $A[(n/2) + 1] \dots A[n]$   
The two lists and their unions have disjoint values

Final condition: Merged list in locations  $A[1] \dots A[n]$

Global  $A[1 \dots n]$

Local  $x, low, high, index$

begin

spawn ( $P_1, P_2, \dots, P_n$ )

for all  $P_i$  where  $1 \leq i \leq n$  do

```
{ Each processor sets bounds for binary search }
if  $i \leq n/2$  then
    low  $\leftarrow (n/2) + 1$ 
    high  $\leftarrow n$ 
else
    low  $\leftarrow 1$ 
    high  $\leftarrow n/2$ 
endif
{ Each processor performs binary search }
 $x \leftarrow A[i]$ 
repeat
    index  $\leftarrow \lfloor (low + high)/2 \rfloor$ 
    if  $x < A[index]$  then
        high  $\leftarrow index - 1$ 
    else
        low  $\leftarrow index + 1$ 
    endif
until  $low > high$ 
{ Put value in correct position on merged list }
 $A[high + i - n/2] \leftarrow x$ 
endfor
end
```

P1	P2	P3	P4
1	5	7	9

Index 3

1	2	4	5	7	8	9	11
---	---	---	---	---	---	---	----

2	4	8	11
---	---	---	----

P5 P6 P7 P8

## Job of P6

Low = 1

4 < 5

4 < 1

**1+6-4 = 3**

High=4

High=1

High=1

**Model: CREW PRAM**

X=4

Low=1

Low=2

**Time: O(log n)**

Index=2

X=4

stop

**Cost: O(n log n)**

$x \leftarrow A[i]$

**repeat**

$index \leftarrow \lfloor (low + high)/2 \rfloor$

if  $x < A[index]$  then

$high \leftarrow index - 1$

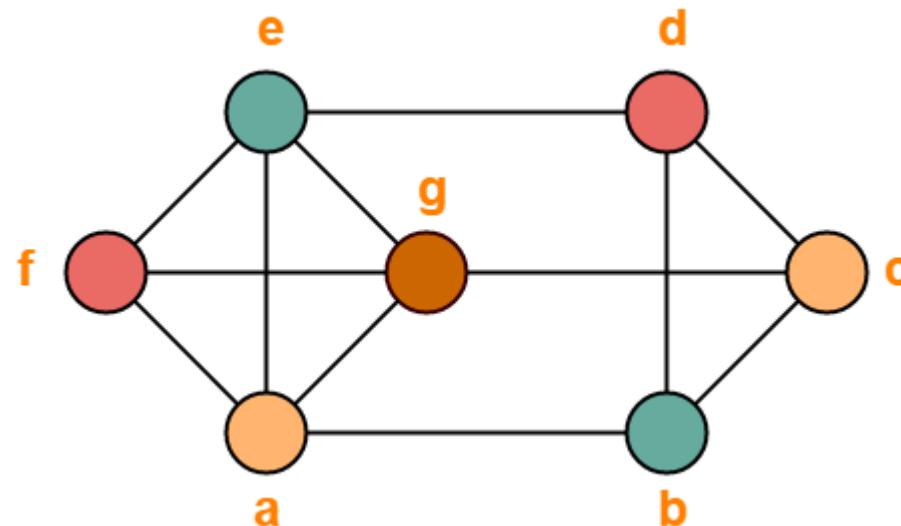
**else**

$low \leftarrow index + 1$

**endif**

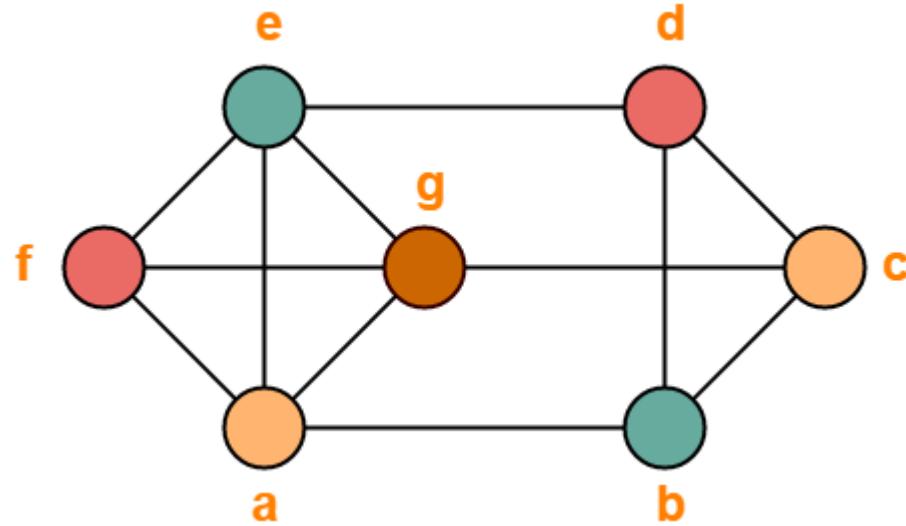
until  $low > high$

# *PRAM Algorithm for Graph Coloring*



# What is Coloring?

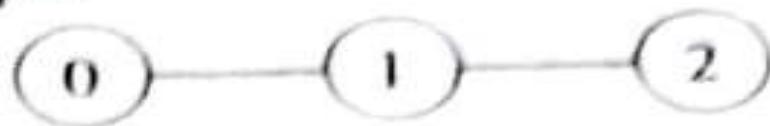
Graph Coloring is an assignment of colors (or any distinct marks) to the vertices of a graph. Strictly speaking, a coloring is a proper coloring if no two adjacent vertices have the same color.



Graph with  
7 Vertices  
4 Colors

# Color a 3-Vertex Graph with 2- Colors

Graph:



Colorings: Initial values: After checking:

0,0,0

0,0,1

0,1,0

0,1,1

1,0,0

1,0,1

1,1,0

1,1,1

1
1
1
1
1
1
1
1

0
0
1
0
0
1
0
0

Number of  
legal colorings:

+ 2

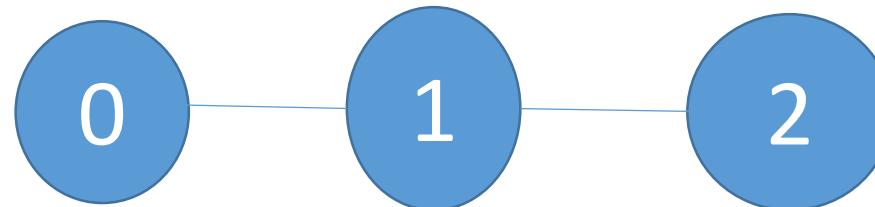
# PRAM Algorithm for Graph Coloring

<b>Global</b>	<i>n</i>	{Number of vertices}
	<i>c</i>	{Number of colors}
	<i>A</i> [1... <i>n</i> ][1... <i>n</i> ]	{Adjacency matrix}
	<i>candidate</i> [1... <i>c</i> ][1... <i>c</i> ] ... [1... <i>c</i> ]	{ <i>n</i> -dimensional boolean matrix}
	<i>valid</i>	{Number of valid colorings}
	<i>j</i> , <i>k</i>	

```
begin
    spawn ( $P(i_0, i_1, \dots, i_{n-1})$ ) where  $0 \leq i_v < c$  for  $0 \leq v < n$ 
    for all  $P(i_0, i_1, \dots, i_{n-1})$  where  $0 \leq i_v < c$  for  $0 \leq v < n$  do
        candidate[ $i_0, i_1, \dots, i_{n-1}$ ]  $\leftarrow 1$ 
        for j  $\leftarrow 0$  to  $n - 1$  do
            for k  $\leftarrow 0$  to  $n - 1$  do
                if A[j][k] and  $i_j = i_k$  then
                    candidate[ $i_0, i_1, \dots, i_n$ ]  $\leftarrow 0$ 
                endif
            endfor
        endfor
        valid  $\leftarrow \Sigma \text{candidate}$  {Sum of all elements of candidate}
    endfor
    if valid  $> 0$  then print "Valid coloring exists"
    else printif "Valid coloring does not exist"
    endif
end
```

# Adjacency Matrix

	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0



0            0    Invalid

0            1    Valid

```
for j ← 0 to n - 1 do
    for k ← 0 to n - 1 do
        if A[j][k] and ij=1 = ik then
            candidate[i0, i1, ..., in] ← 0
        endif
    endfor
    ..
```

If A[j][k]==1  
Sum=2 > 0  
Valid coloring  
exists

0
0
1
0
0
0
1
0
0

# Complexity and Which model of PRAM

- CREW PRAM
- Processors  $c^n$
- Spawn  $O(\log c^n)$
- $O(n^2)$  for loops
- $O(\log c^n)$  with  $c^n$  processors - summation
- Overall complexity
- $O(\log c^n + n^2) = O(n^2 + n \log c)$
- Bcoz  $c < n$ , reduced to  $O(n^2)$

# Applications of Graph Coloring

- Many problems can be formulated as a graph coloring problem including Time Tabling, Scheduling, Register Allocation, Channel Assignment.
- A lot of research has been done in this area so much is already known about the problem space.

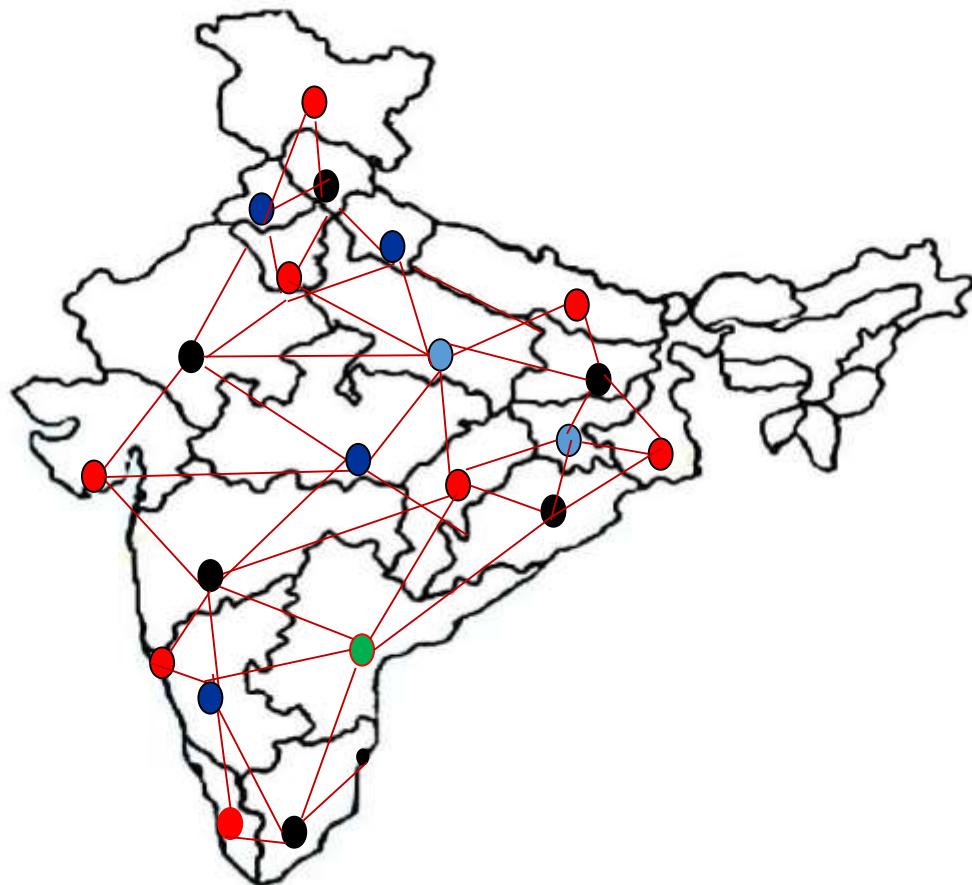
# Solution to Problem



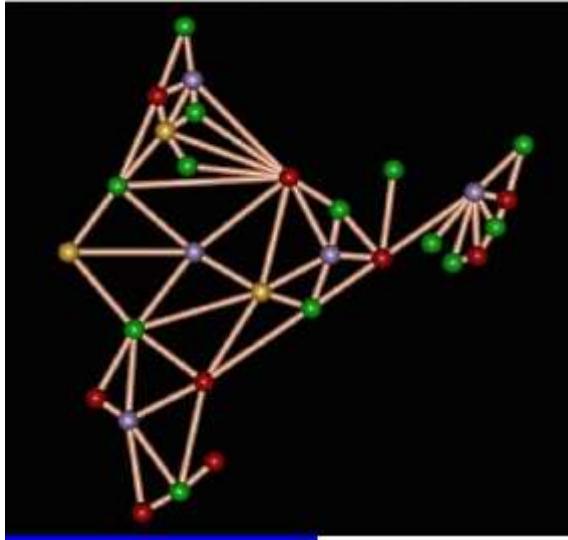
# Explanation

- The standard approach to coloring a map is to use a single color for a state and never use the same color for two states.
- Two states whose common border is just one point can be colored, if we so choose, with the same color.

# Explanation (cont)



# Explanation (cont)



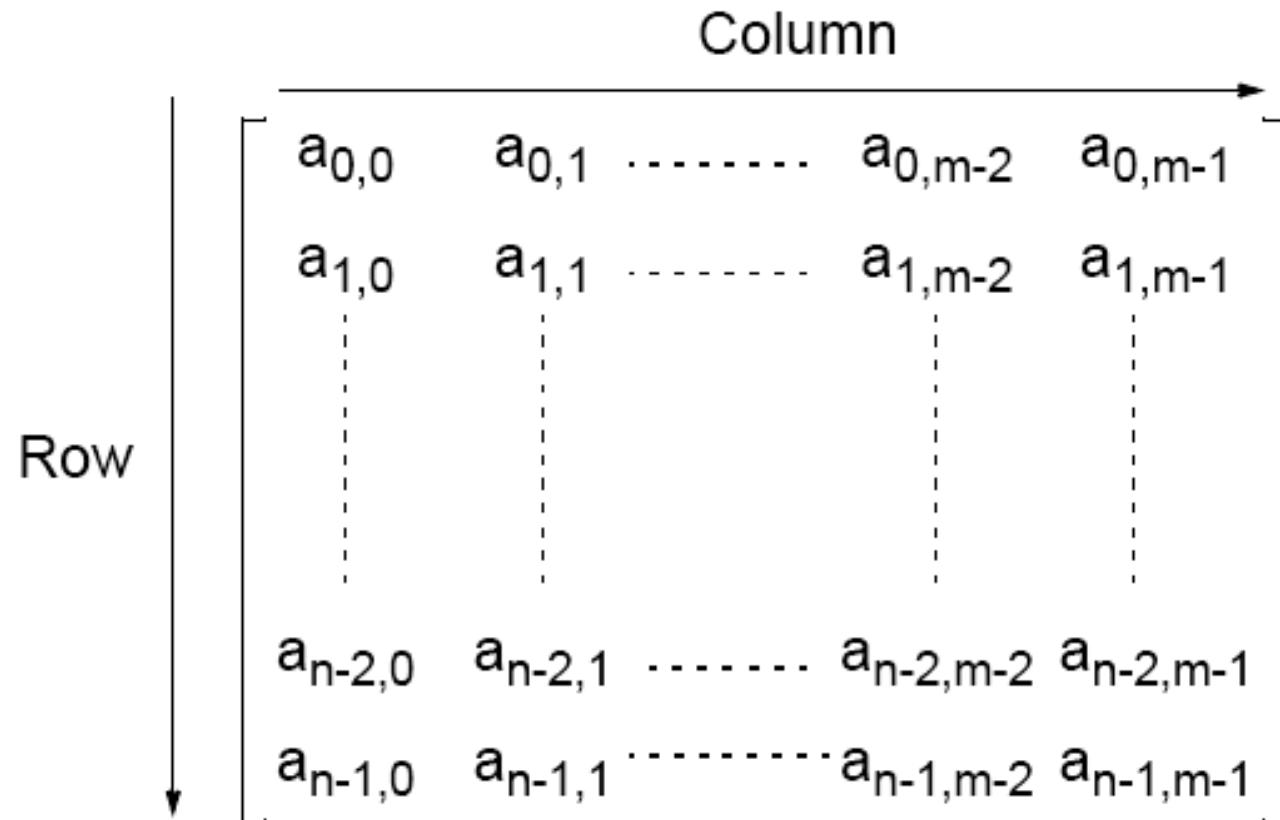
GSM networks operate in only four different frequency ranges. The reason why only four different frequencies because any country can colored with four different colors.

*Thank you*

# **Parallel Matrix Multiplication**

# Matrices — A Review

An  $n \times m$  matrix



# Matrix Addition

Involves adding corresponding elements of each matrix to form the result matrix.

Given the elements of **A** as  $a_{i,j}$  and the elements of **B** as  $b_{i,j}$ , each element of **C** is computed as

$$c_{i,j} = a_{i,j} + b_{i,j}$$

$$(0 \leq i < n, 0 \leq j < m)$$

# Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces the matrix **C** whose elements,  $c_{i,j}$  ( $0 \leq i < n$ ,  $0 \leq j < m$ ), are computed as follows:

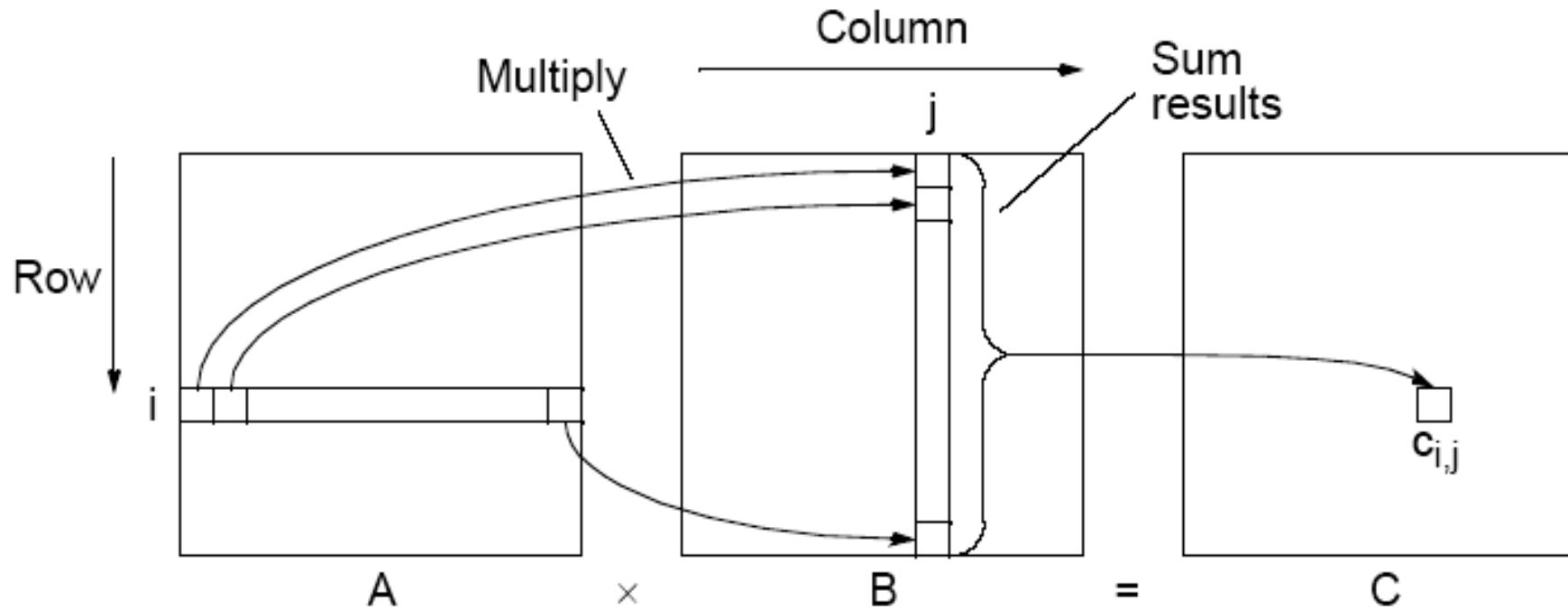
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

where **A** is an  $n \times l$  matrix and **B** is an  $l \times m$  matrix.

## Sequential algorithm

```
> for (i = 0; i < n; i++)  
    for (j = 0; j < m; j++)  
        c[i][j] = 0;  
        for (k = 0; k < l; k++)  
            c[i][j] += a[i][k] * b[k][j]  
    end for  
end for
```

# Matrix multiplication, $C = A \times B$



# Cannon Algorithm (2D Mesh SIMD)

- This is a memory efficient algorithm.
- Both n matrices A & B are partitioned among P processors.

$\mathbf{A}_{0,0}$	$\mathbf{A}_{0,1}$	$\mathbf{A}_{0,2}$	$\mathbf{A}_{0,3}$
$\mathbf{A}_{1,0}$	$\mathbf{A}_{1,1}$	$\mathbf{A}_{1,2}$	$\mathbf{A}_{1,3}$
$\mathbf{A}_{2,0}$	$\mathbf{A}_{2,1}$	$\mathbf{A}_{2,2}$	$\mathbf{A}_{2,3}$
$\mathbf{A}_{3,0}$	$\mathbf{A}_{3,1}$	$\mathbf{A}_{3,2}$	$\mathbf{A}_{3,3}$

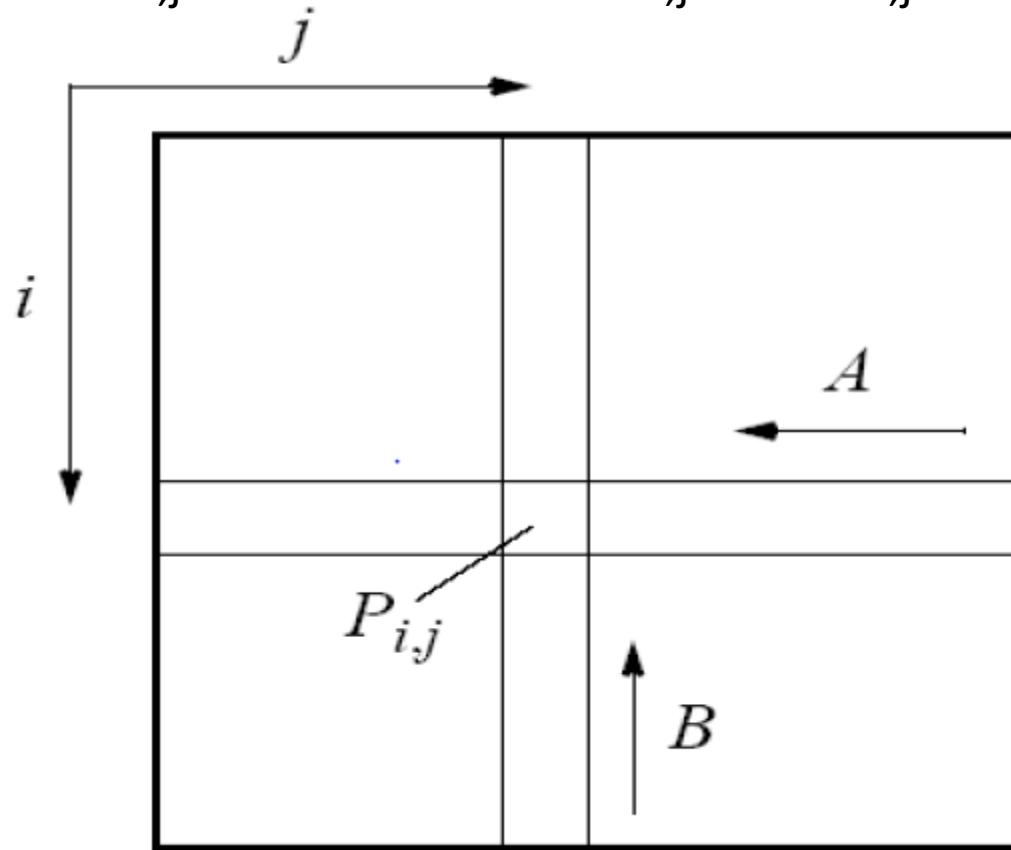
A

$\mathbf{B}_{0,0}$	$\mathbf{B}_{0,1}$	$\mathbf{B}_{0,2}$	$\mathbf{B}_{0,3}$
$\mathbf{B}_{1,0}$	$\mathbf{B}_{1,1}$	$\mathbf{B}_{1,2}$	$\mathbf{B}_{1,3}$
$\mathbf{B}_{2,0}$	$\mathbf{B}_{2,1}$	$\mathbf{B}_{2,2}$	$\mathbf{B}_{2,3}$
$\mathbf{B}_{3,0}$	$\mathbf{B}_{3,1}$	$\mathbf{B}_{3,2}$	$\mathbf{B}_{3,3}$

B

# Cannon Algorithm –Cont.

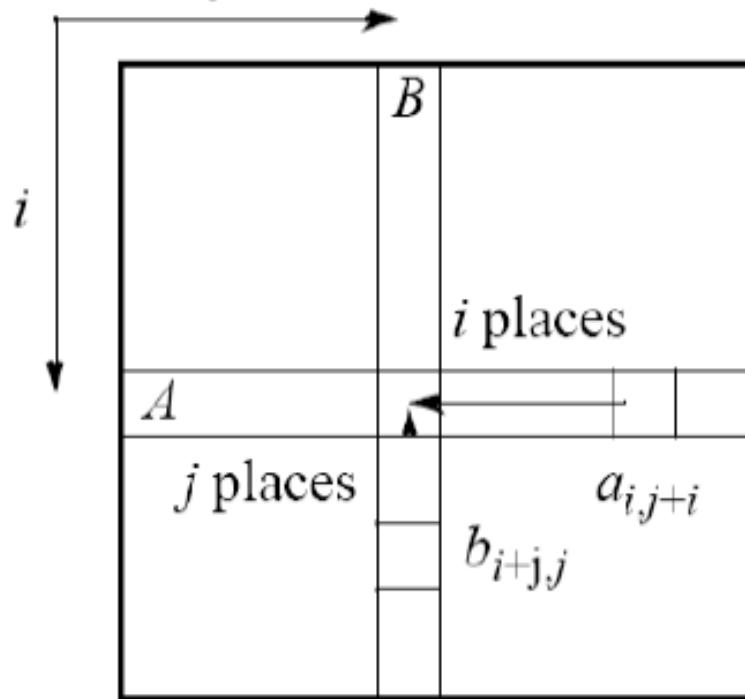
1. Initially processor  $P_{i,j}$  has elements  $a_{i,j}$  and  $b_{i,j}$  ( $0 \leq i < n$ ,  $0 \leq j < n$ )



# Cannon Algorithm –Cont.

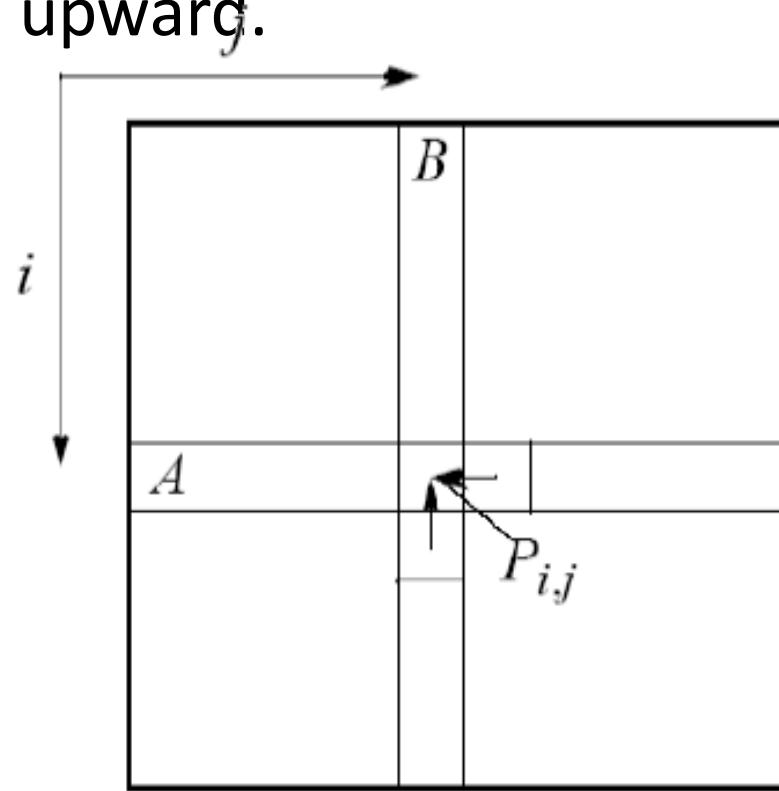
- Elements are moved from their initial position to an aligned position.

The complete  $i^{\text{th}}$  row of A is shifted  $i$  places left and the complete  $j^{\text{th}}$  column of B is shifted  $j$  places upward.



# Cannon Algorithm –Cont.

3. Each processor  $P_{i,j}$  multiply its elements.
4. The  $i^{\text{th}}$  row of A is shifted **one** place left, and the  $j^{\text{th}}$  column of B is shifted **one** place upward.



## Cannon Algorithm –Cont.

5. Each processor  $P_{i,j}$  multiplies its elements brought to it and adds the results to the accumulating sum.
6. Step 4 and 5 are repeated until the final result is obtained ( $n-1$  shifts with  $n$  rows and  $n$  columns of elements).

# Cannon Algorithm –Cont.

Processor P1,2

$$C_{12} = a_{10} * b_{02} + a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32}$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$\underbrace{\phantom{aaaaaaaaaaaa}}$   
 $A$

$\underbrace{\phantom{aaaaaaaaaaaa}}$   
 $B$

$\underbrace{\phantom{aaaaaaaaaaaa}}$   
 $C$

			$b_{0,1}$
		$b_{0,2}$	$b_{1,2}$
	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$
	$b_{0,0}$	$b_{1,1}$	$b_{2,2}$
			$b_{3,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
	$b_{1,0}$	$b_{2,1}$	$b_{3,2}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
		$b_{2,0}$	$b_{3,1}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
		$b_{3,0}$	

(c)

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$b_{0,0}$	$b_{1,1}$	$b_{2,2}$	$b_{3,3}$
$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,0}$
$b_{1,0}$	$b_{2,1}$	$b_{3,2}$	$b_{0,3}$
$a_{2,2}$	$a_{2,3}$	$a_{2,0}$	$a_{2,1}$
$b_{2,0}$	$b_{3,1}$	$b_{0,2}$	$b_{1,3}$
$a_{3,3}$	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$
$b_{3,0}$	$b_{0,1}$	$b_{1,2}$	$b_{2,3}$

# Cannon Algorithm –Cont.

$$\begin{matrix} A_{0,0} \\ B_{0,0} \end{matrix}$$

$$\begin{matrix} A_{0,1} \\ B_{0,1} \end{matrix}$$

$$\begin{matrix} A_{0,2} \\ B_{0,2} \end{matrix}$$

$$\begin{matrix} A_{0,3} \\ B_{0,3} \end{matrix}$$

$$\begin{matrix} A_{0,0} \\ B_{0,0} \end{matrix}$$

$$\begin{matrix} A_{0,1} \\ B_{1,1} \end{matrix}$$

$$\begin{matrix} A_{0,2} \\ B_{2,2} \end{matrix}$$

$$\begin{matrix} A_{0,3} \\ B_{3,3} \end{matrix}$$

$$\begin{matrix} A_{1,0} \\ B_{1,0} \end{matrix}$$

$$\begin{matrix} A_{1,1} \\ B_{1,1} \end{matrix}$$

$$\begin{matrix} A_{1,2} \\ B_{1,2} \end{matrix}$$

$$\begin{matrix} A_{1,3} \\ B_{1,3} \end{matrix}$$

$$\begin{matrix} A_{1,1} \\ B_{1,0} \end{matrix}$$

$$\begin{matrix} A_{1,2} \\ B_{2,1} \end{matrix}$$

$$\begin{matrix} A_{1,3} \\ B_{3,2} \end{matrix}$$

$$\begin{matrix} A_{1,0} \\ B_{0,3} \end{matrix}$$

$$\begin{matrix} A_{2,0} \\ B_{2,0} \end{matrix}$$

$$\begin{matrix} A_{2,1} \\ B_{2,1} \end{matrix}$$

$$\begin{matrix} A_{2,2} \\ B_{2,2} \end{matrix}$$

$$\begin{matrix} A_{2,3} \\ B_{2,3} \end{matrix}$$

$$\begin{matrix} A_{2,2} \\ B_{2,0} \end{matrix}$$

$$\begin{matrix} A_{2,3} \\ B_{3,1} \end{matrix}$$

$$\begin{matrix} A_{2,0} \\ B_{0,2} \end{matrix}$$

$$\begin{matrix} A_{2,1} \\ B_{1,3} \end{matrix}$$

$$\begin{matrix} A_{3,0} \\ B_{3,0} \end{matrix}$$

$$\begin{matrix} A_{3,1} \\ B_{3,1} \end{matrix}$$

$$\begin{matrix} A_{3,2} \\ B_{3,2} \end{matrix}$$

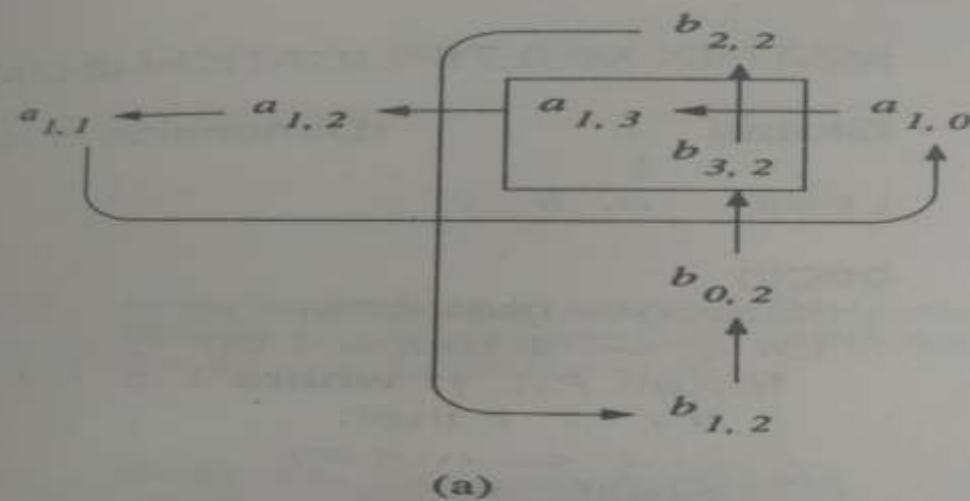
$$\begin{matrix} A_{3,3} \\ B_{3,3} \end{matrix}$$

$$\begin{matrix} A_{3,3} \\ B_{3,0} \end{matrix}$$

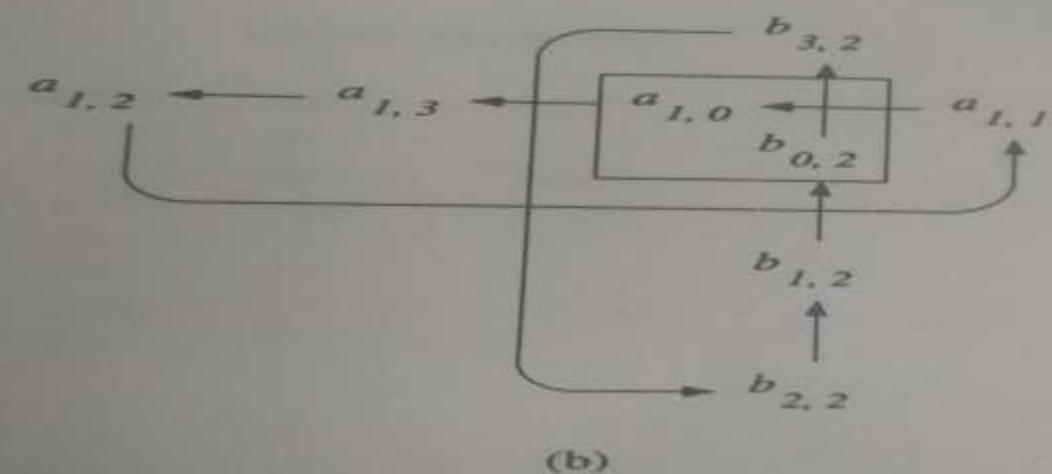
$$\begin{matrix} A_{3,0} \\ B_{0,1} \end{matrix}$$

$$\begin{matrix} A_{3,1} \\ B_{1,2} \end{matrix}$$

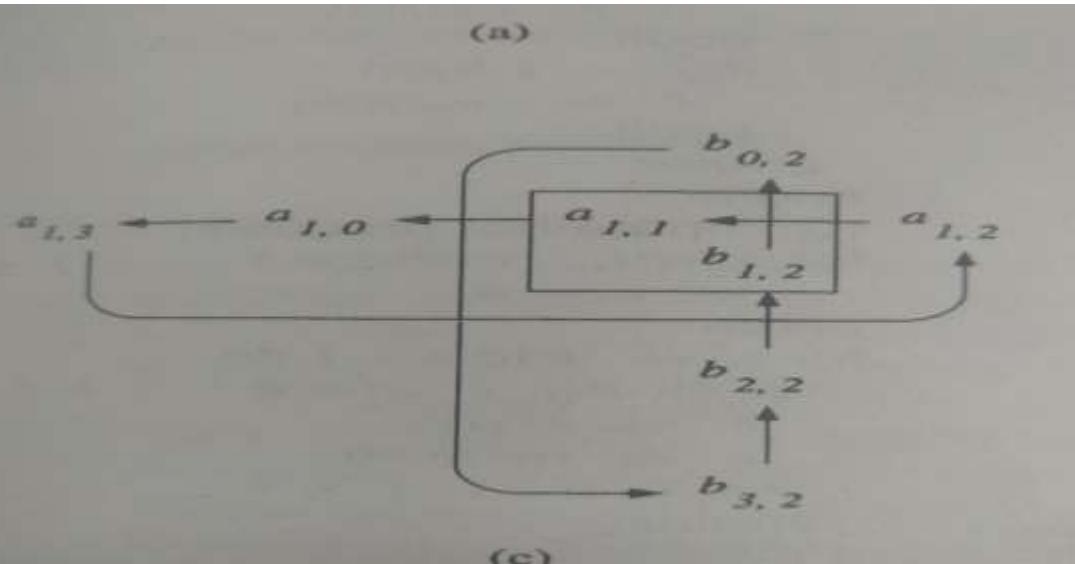
$$\begin{matrix} A_{3,2} \\ B_{2,3} \end{matrix}$$



(a)

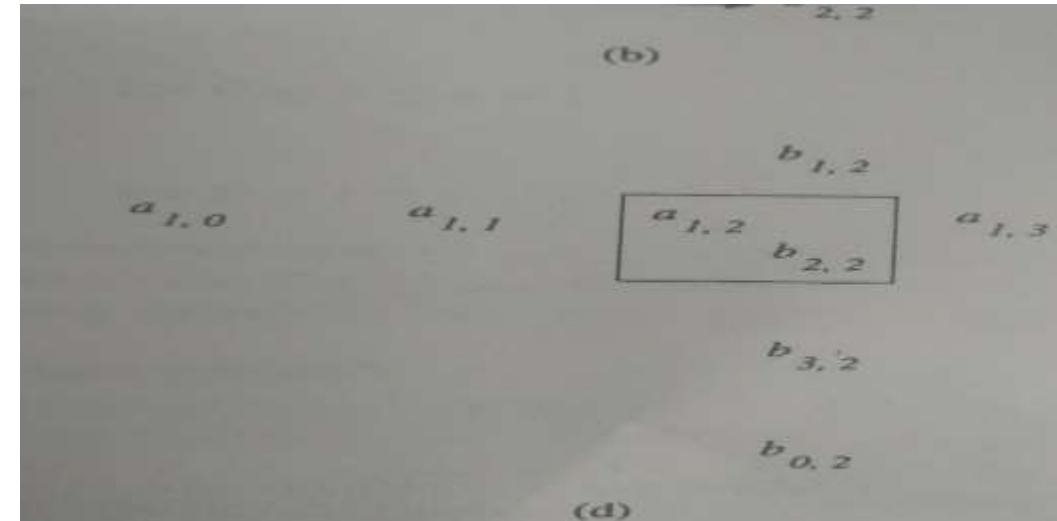


(b)



(c)

RE 7-4 2-D mesh SIMD matrix multiplication  $P(1,2)$ . The matrices have already been



(d)

from the point of view of Processor 0. (a) First scalar multiplication step. The other elements of A are cycled to the left and the second scalar multiplication step occurs after second

# MATRIX MULTIPLICATION (2-D MESH SIMD)

Global     $n$                               {Dimension of matrices}  
               $k$   
Local      $a, b, c$

```
begin
  {Stagger matrices}
  for  $k \leftarrow 1$  to  $n - 1$  do
    for all  $P(i, j)$  where  $1 \leq i, j \leq n$  do
      if  $i > k$  then
         $a \leftarrow east(a)$ 
      endif
      if  $j > k$  then
         $b \leftarrow south(b)$ 
      endif
    endfor
  endfor
  {Compute dot products}
  for all  $P(i, j)$  where  $1 \leq i, j \leq n$  do
     $c \leftarrow a \times b$ 
  endfor
  for  $k \leftarrow 1$  to  $n - 1$  do
    for all  $P(i, j)$  where  $1 \leq i, j \leq n$  do
       $a \leftarrow east(a)$ 
       $b \leftarrow south(b)$ 
       $c \leftarrow c + a \times b$ 
    endfor
  endfor
end
```

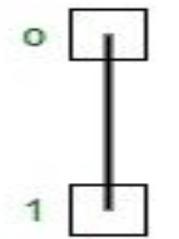
on of parallel matrix  
algorithm on the  
MD model.

Thank u

# *Matrix Multiplication on Hypercube SIMD Model*

Given the Hypercube SIMD Model with  $n^3=2^{3q}$  Processors, with  $n \times n$  matrices can be multiplied in  $O(\log n)$  time.

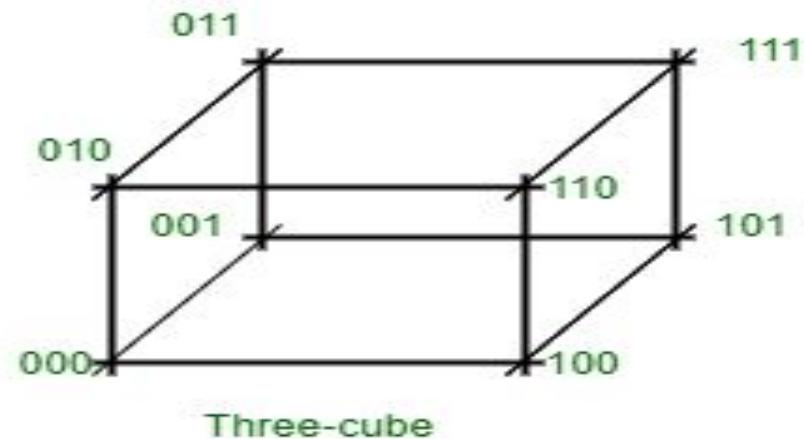
# Hypercube



One-cube

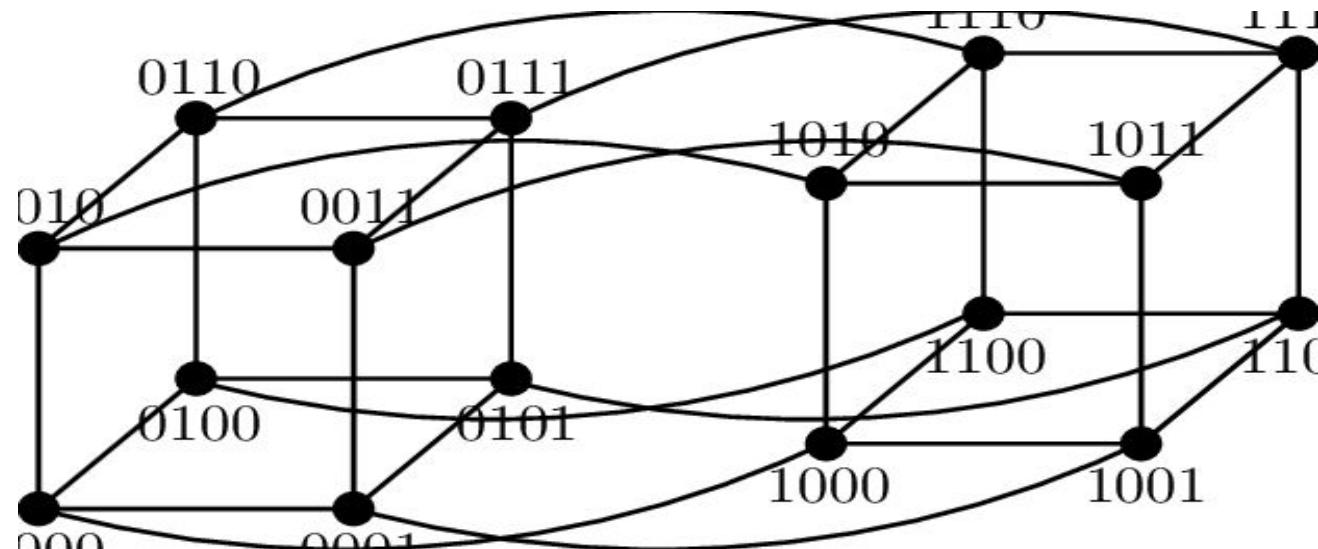


Two-cube



Three-cube

Figure - Hypercube Structures For  $n = 1, 2, 3$



For  $n=4$

# Important Functions

Functions BIT and BIT.COMPLEMENT.

**BIT(9, 0) = 1**

**BIT(9, 1) = 0**

**BIT(9, 3) = 1**

**BIT(9, 4) = 0**

**BIT(9, 5) = 0**

**BIT.COMPLEMENT(9, 0) = 8**

**BIT.COMPLEMENT(9, 1) = 11**

**BIT.COMPLEMENT(9, 3) = 1**

**BIT.COMPLEMENT(9, 4) = 25**

**BIT.COMPLEMENT(9, 5) = 41**

**BIT(9,0) -> 00001001**

**BIT(9,0) = 1 BIT(9,1) = 0**

**BIT.COMPLEMENT(9,0) = 8**

**00001001->00001000 -> 8**

**BIT.COMPLEMENT(9,3)**

**00001001->00000001 -> 1**

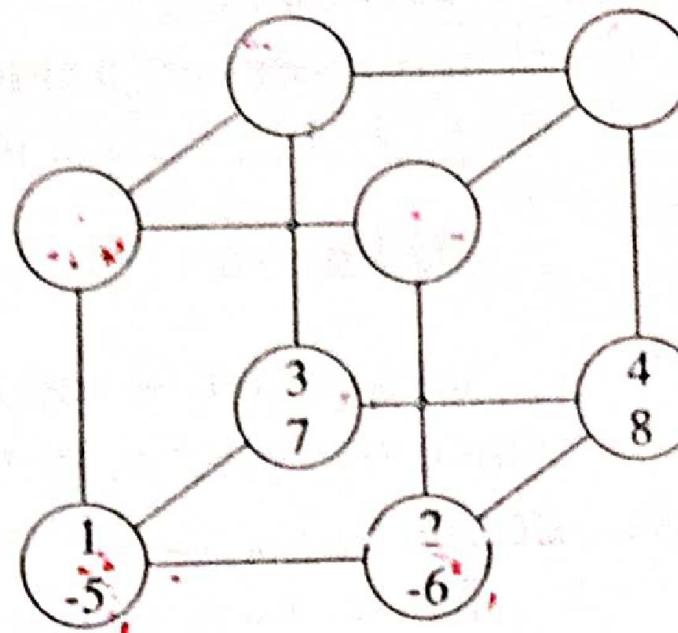
# Initial Allocation

$$\begin{matrix} \mathbf{A} & \mathbf{B} & \mathbf{C} \\ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & \times & \begin{pmatrix} -5 & -6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix} \end{matrix}$$

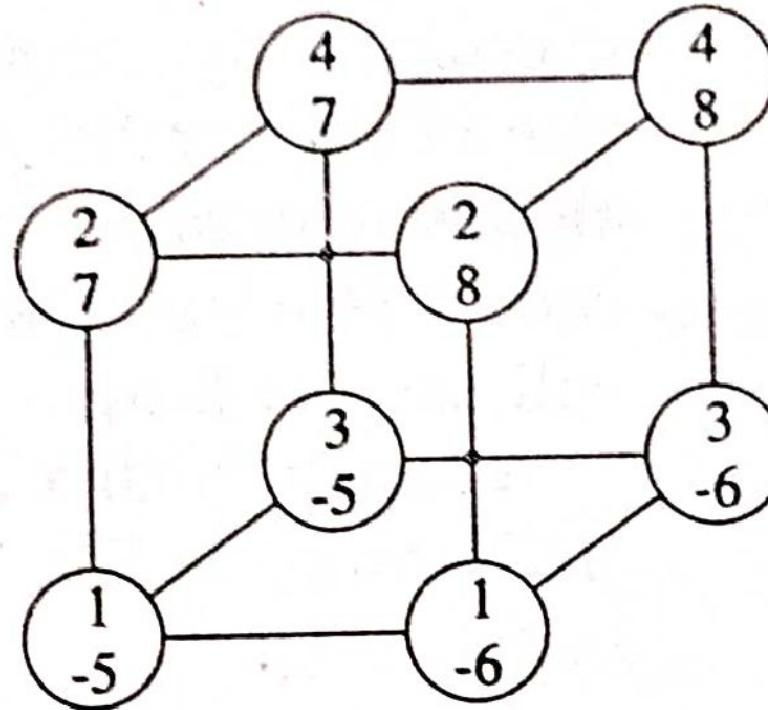
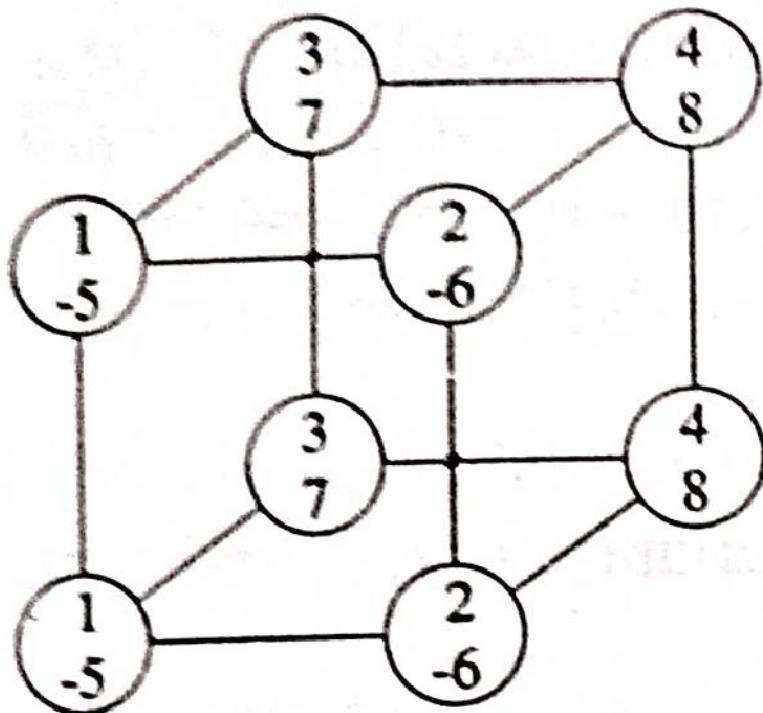
## Local Variables

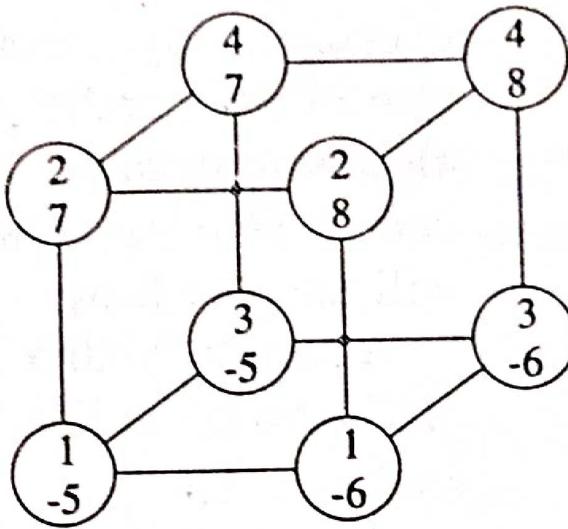
	$a$	$b$
$P(0)$	$a_{0,0}$	$b_{0,0}$
$P(1)$	$a_{0,1}$	$b_{0,1}$
$P(2)$	$a_{1,0}$	$b_{1,0}$
$P(3)$	$a_{1,1}$	$b_{1,1}$
$P(4)$		
$P(5)$		
$P(6)$		
$P(7)$		

Initial allocation of matrix elements to processing elements on hypercube SIMD model, where  $n = 2$ ,  $q = 1$ , and  $p = 8$ .

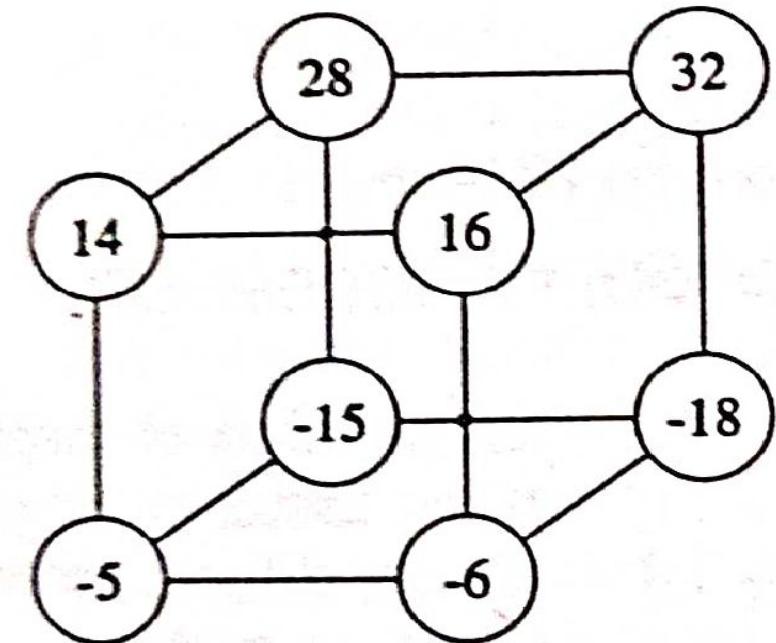


# Broadcast the matrices A and B

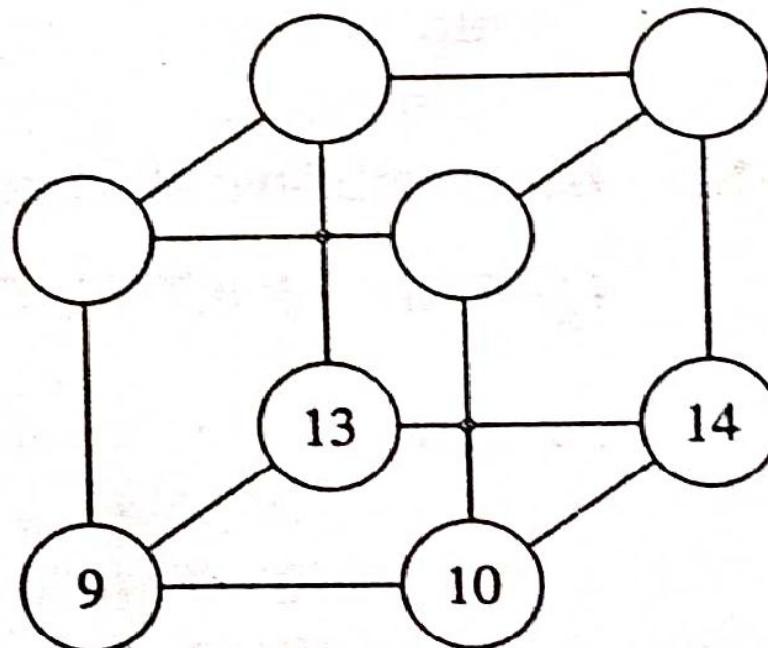




**Do the Multiplication in parallel**



**Sum the products**



# Complete Algorithm

---

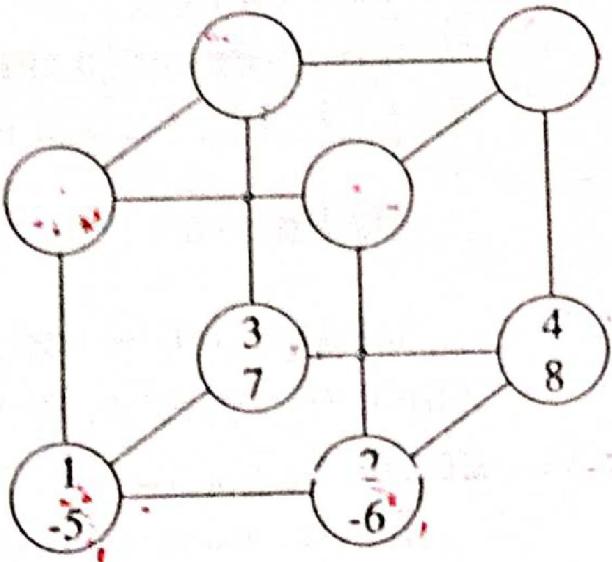
```
Parameter q { matrix size is  $2^q \times 2^q$ }
Global I
Local a, b, c, s, t
Begin
{Phase 1: Broadcast matrices A and B}
for  $I \leftarrow 3q-1$  down to  $2q$  do
    for all  $P_m$  where  $\text{BIT}(m, I) = I$  do
         $t \leftarrow \text{BIT.COMPLEMENT}(m, I)$ 
         $a \Leftarrow [t]a$ 
         $b \Leftarrow [t]b$ 
    end for
end for
for  $I \leftarrow q-1$  down to  $0$  do
    for all  $P_m$ , where  $\text{BIT}(m, I) \neq \text{BIT}(m, 2q+I)$  do
         $t \leftarrow \text{BIT.COMPLEMENT}(m, I)$ 
         $a \Leftarrow [t]a$ 
    end for
end for
for  $I \leftarrow 2q-1$  down to  $q$  do
    for all  $P_m$  do
         $t \leftarrow \text{BIT.COMPLEMENT}(m, I)$ 
         $b \Leftarrow [t]b$ 
    end for
end for
{Phase 2: Do the multiplications in parallel}
For all  $P_m$  do
     $c \leftarrow a \times b$ 
end for
{Phase 3: Sum the products}
for  $I \leftarrow 2q$  to  $3q-1$  do
    for all  $P_m$  do
         $t \leftarrow \text{BIT.COMPLEMENT}(m, I)$ 
         $s \Leftarrow [t]c$ 
         $c \leftarrow c + s$ 
    end for
end for
End
```

---

Figure 1. Pseudocódigo Hypercube Algorithm for matrix multiplication (Hypercube SIMD).

Source Quinn, Michael J., [4]

# Broadcast the matrices A and B



```
begin
{Phase 1: Broadcast matrices A and B}
for  $l \leftarrow 3q - 1$  downto  $2q$  do
  for all  $P_m$ , where  $\text{BIT}(m, l) = 1$  do
     $t \leftarrow \text{BIT.COMPLEMENT}(m, l)$ 
     $a \Leftarrow [t]a$ 
     $b \Leftarrow [t]b$ 
  endfor
endfor
```

P0 P1 P2 P3 P4 P5 P6 P7  
000 001 010 011 100 101 110 111

$l = 2$   $M=0$

$\text{BIT}(0,2) \rightarrow 000 \rightarrow 0 \neq 1$

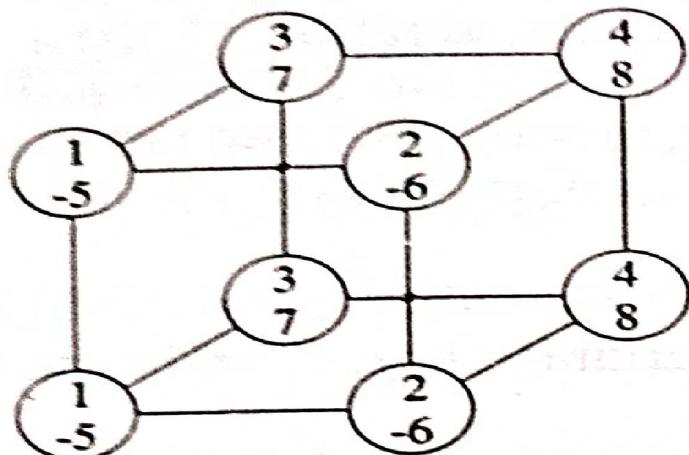
Only P4,5,6,7  $\text{BIT}(m,l)=1$

P4

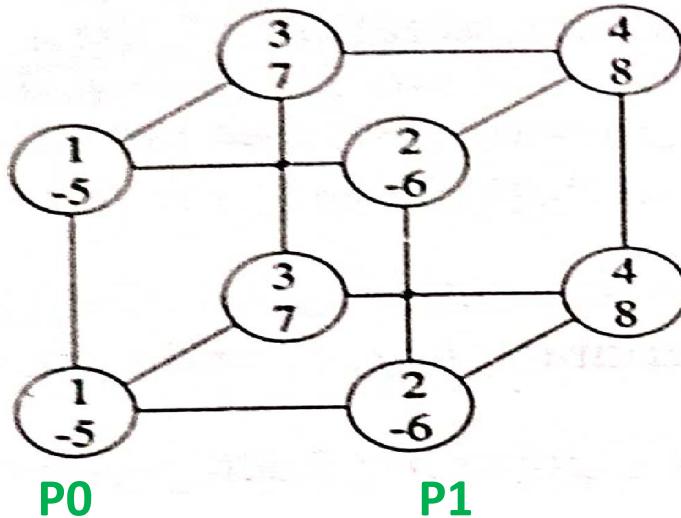
$t = \text{BIT} > \text{COMPLEMENT}(4,2) \rightarrow 100 \rightarrow 000 \rightarrow P0$

$a = [0]a$

$B = [0]b$

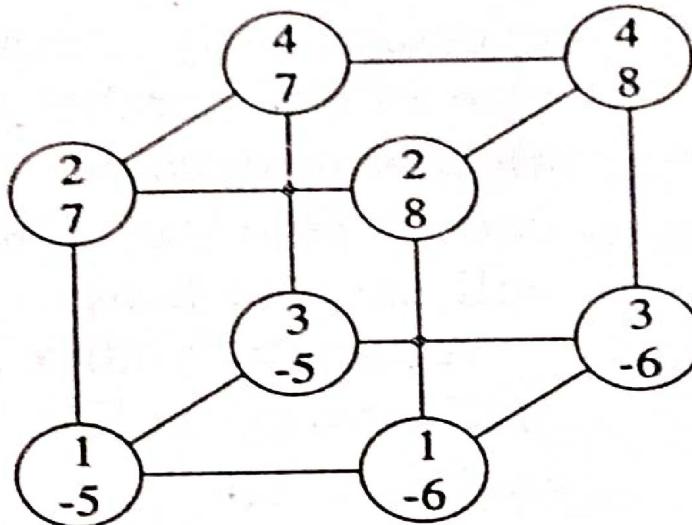


# Broadcast the matrices A and B



P0

P1



```
for l ← q - 1 downto 0 do
    for all  $P_m$ , where  $\text{BIT}(m, l) \neq \text{BIT}(m, 2q + l)$  do
        t ←  $\text{BIT.COMPLEMENT}(m, l)$ 
        a ← [t]a
    endfor
endfor
for l ← 2q - 1 downto q do
    for all  $P_m$ , where  $\text{BIT}(m, l) \neq \text{BIT}(m, q + l)$  do
        t ←  $\text{BIT.COMPLEMENT}(m, l)$ 
        b ← [t]b
    endfor
endfor
```

l=0

P1

$\text{BIT}(1,0)=001=1$

$\text{BIT}(m,2q+l)=\text{BIT}(1,2)=001=0$

$1 \neq 0$

$T = \text{BIT.COMPLEMENT}(1,0)=001=000=0$

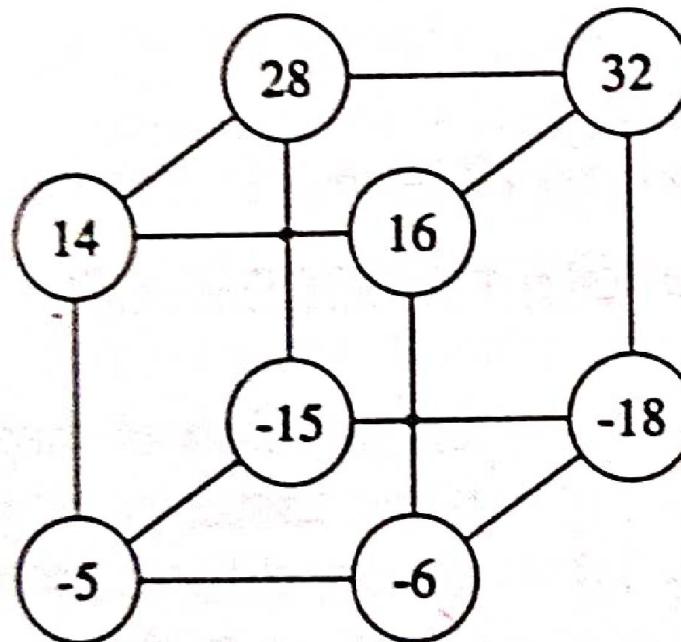
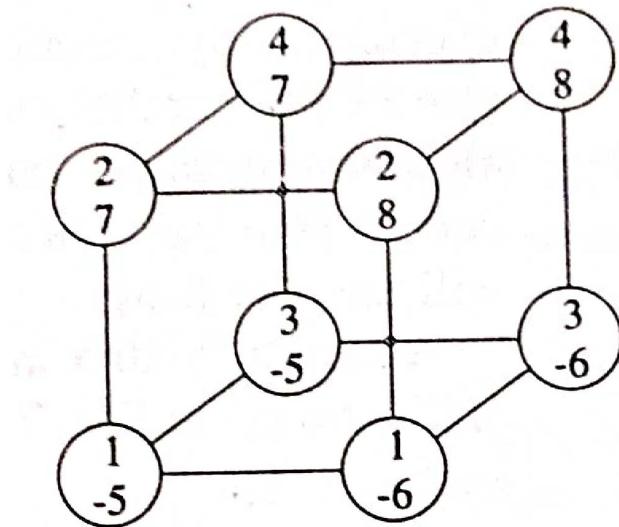
$a=[0]a$

## Do the Multiplication in parallel

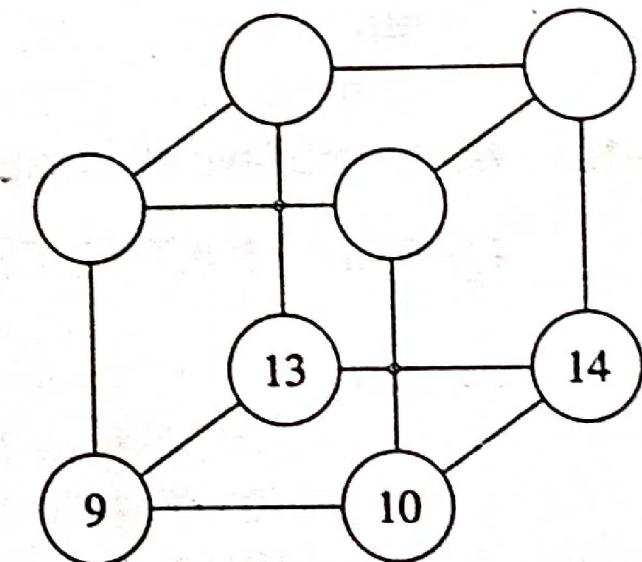
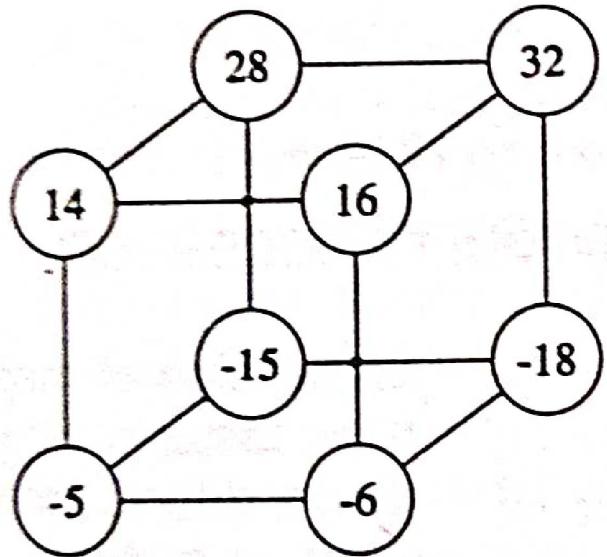
{Phase 2: Do the multiplications in parallel}  
for all  $P_m$  do

$$c \leftarrow a \times b$$

endfor



## Sum of products



{Phase 3: Sum the products}

for  $l \leftarrow 2q$  to  $3q - 1$  do

for all  $P_m$  do

$t \leftarrow \text{BIT.COMPLEMENT}(m, l)$

$s \leftarrow [t]c$

$c \leftarrow c + s$

endfor

endfor

end

P3

$l=2$  to  $2$

$t=\text{BIT.COMPLEMENT}(3,2)=011=111=7$

$S=[7]c$

$C=-18+32=14$

*Thank YOU*

# Overview – Sorting algorithms

- Enumeration sort
- Odd Even Transposition sort
- Parallel Quick sort
- Hypercube Quick sort

# Enumeration sort

- Compute the final position of each element in the sorted list by comparing with other elements and counting the number of elements having smaller value.
- CRCW PRAM model
  - If multiple processors simultaneously write values to a single memory location, the sum of the values is assigned to that location.
- Number of processors
  - Given  $n^2$  processors, CRCW PRAM model can compare every pair of elements and compute each element's list position in constant time.



## ENUMERATION SORT (SPECIAL CRCW PRAM):

Parameter  $n$  {Number of elements}  
Global  $a[0...(n - 1)]$  {Elements to be sorted}  
 $position[0...(n - 1)]$  {Sorted positions}  
 $sorted[0...(n - 1)]$  {Contains sorted elements}

begin

spawn ( $P_{i,j}$ , for all  $0 \leq i, j < n$ )

for all  $P_{i,j}$ , where  $0 \leq i, j < n$  do

$position[i] \leftarrow 0$

if  $a[i] < a[j]$  or ( $a[i] = a[j]$  and  $i < j$ ) then

$position[i] \leftarrow 1$

endif .

endfor

for all  $P_{i,0}$ , where  $0 \leq i < n$  do

$sorted[position[i]] \leftarrow a[i]$

endfor

end

Change in the condition  
given in book:  
If  $a_i > a_j$

$A [ \begin{matrix} 0 & 1 & 2 & 3 \\ 12 & 9 & 2 & 5 \end{matrix} ]$ 
 $n^2$  processes  
 16 "

$p_{00}$	$p_{01}$	$p_{02}$	$p_{03}$	$p_{10}$	$p_{11}$	$p_{12}$	$p_{13}$	$p_{20}$	$p_{21}$	$p_{22}$	$p_{23}$	$p_{30}$	$p_{31}$	$p_{32}$	$p_{33}$
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

$p_{00}^0$  0                          0                          0                          0

if ( $a_i^0 > a_j^0$ ) || ( $a_i^0 = a_j^0$  and  $i < j$ )

- $a_0 > a_1$   $a_0 > a_2$   $a_0 > a_3$  ... - - - - -

$\underbrace{-1}_{p_{01}^0}$   $\underbrace{-1}_{p_{02}^0}$   $\underbrace{-1}_{p_{03}^0}$   $\underbrace{-1}_{p_{10}^0}$   $\underbrace{-1}_{p_{11}^0}$   $\underbrace{-1}_{p_{12}^0}$   $\underbrace{-1}_{p_{13}^0}$

$p_{01}^0$  3                          2                          0                          1

sorted [  $p_{01}^0$  ] =  $a[0]$   
 sorted [ 3 ] =  $a_0$ , sorted [ 2 ] =  $a_1$ , sorted [ 0 ] =  $a_2$ , sorted [ 1 ] =  $a_3$

sorted array = { 2, 5, 9, 12 }

- Time complexity
  - Spawn time -  $\theta(\log n)$  for  $n^2$  processors
  - Comparison -  $\theta(1)$
  - Overall complexity is  $\theta(\log n)$
- Cost ( number of comparisons)
  - Sequential algorithm  $\theta(n \log n)$
  - Parallel algorithm  $\theta(n^2)$  . Therefore not cost optimal

# Odd-Even Transposition sort

- Processor array model - one dimensional mesh
- Input:  $A = (a_0, a_1, \dots, a_{n-1})$  – set of  $n$  elements to be sorted
- Number of processors:  $n$
- Each of  $n$  processing elements contains two local variables
  - $a$  : unique element of array  $A$
  - $t$  : a variable containing a value retrieved from neighbouring processing element
- Number of iterations:  $n/2$

- Phases:
  - Odd-even exchange
    - ✓ The value of  $a$  in every odd numbered processor (except processor  $n-1$ ) is compared with the value of  $a$  stored in the successor processor
    - ✓ The values are exchanged, if necessary, so that low-numbered processor contains the smaller value
  - Even-odd exchange
    - ✓ The value of  $a$  in every even-numbered processor is compared with the value of  $a$  stored in the successor processor
    - ✓ As in the first phase, the values are exchanged, if necessary, so that low-numbered processor contains the smaller value



25

## ODD-EVEN TRANSPOSITION SORT (ONE-DIMENSIONAL MESH PROCESSOR ARRAY):

```
Parameter      n          (Number of elements in array)
Global         i          (Index variable)
Local          a          (Element to be sorted)
                  t          (Element taken from adjacent processor)

begin
    for i ← 1 to n/2 do
        for all  $P_j$ , where  $0 \leq j \leq n - 1$  do
            if  $j < n - 1$  and  $\text{odd}(j)$  then
                t ← successor(a)           (Odd-even exchange)
                successor(a) ← max(a, t)   (Get value from successor)
                a ← min(a, t)             (Give away larger value)
            endif
            if even(j) then
                t ← successor(a)           (Even-odd exchange)
                successor(a) ← max(a, t)   (Get value from successor)
                a ← min(a, t)             (Give away larger value)
            endif
        endfor
    endfor
end
```

Odd-even transposition sort algorithm for the one-dimensional mesh processor array model.

Odd-even transposition sort of eight values on the one-dimensional mesh processor array model.

Indices:                    0    1    2    3    4    5    6    7

Initial values:            G    H    F    D    E    C    B    A

After odd-even exchange: G    F < H    D < E    B < C    A

After even-odd exchange: F < G    D < H    B < E    A < C

After odd-even exchange: F    D < G    B < H    A < E    C

After even-odd exchange: D < F    B < G    A < H    C < E

After odd-even exchange: D    B < F    A < G    C < H    E

After even-odd exchange: B < D    A < F    C < G    E < H

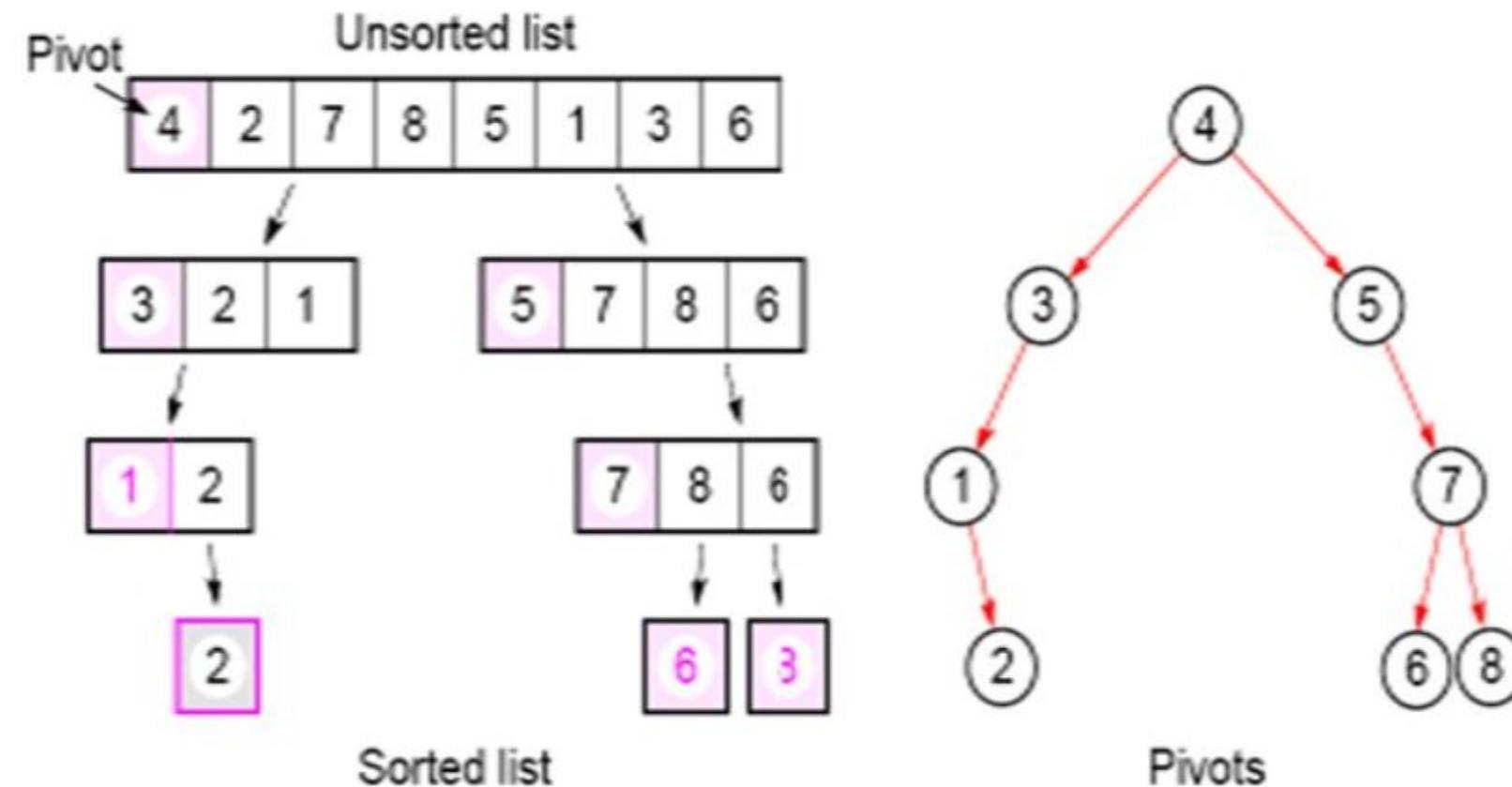
After odd-even exchange: B    A < D    C < F    E < G    H

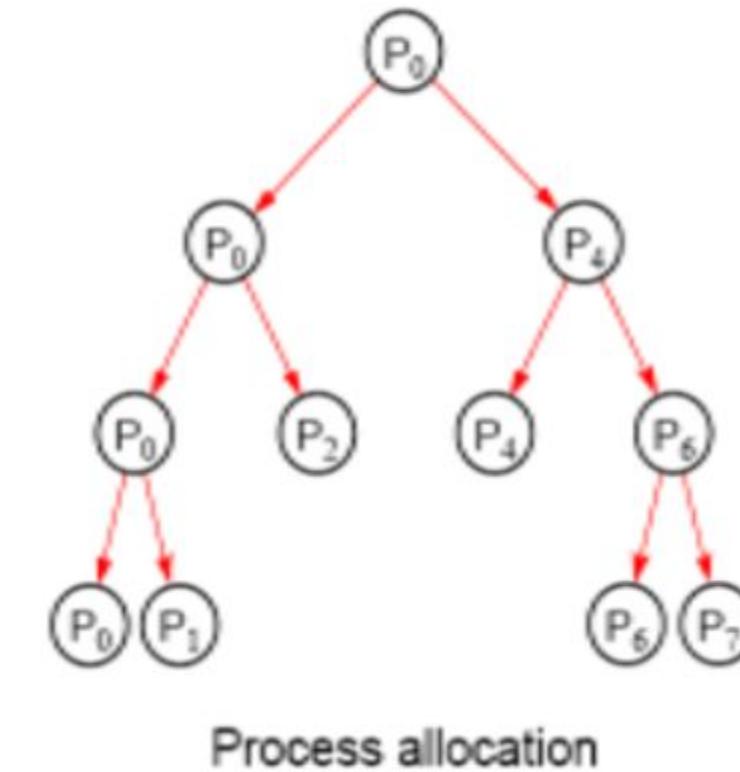
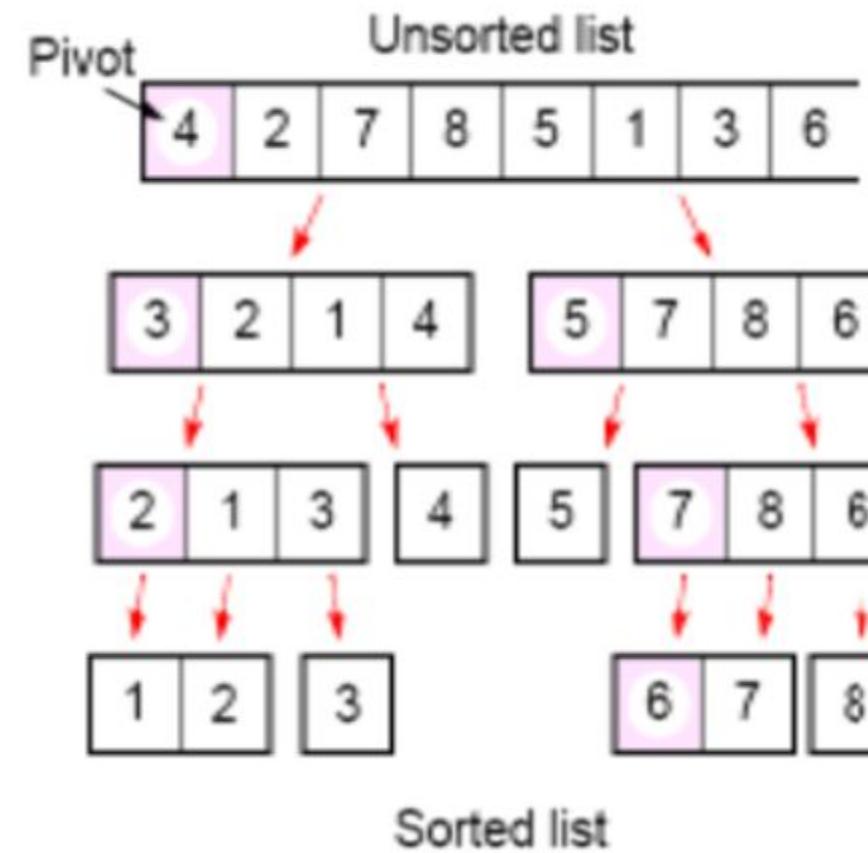
After even-odd exchange: A < B    C < D    E < F    G < H

- The complexity of sorting n elements on a one-dimensional mesh processor array with n processors using odd-even transposition sort is  $\theta(n)$

# Parallel Quicksort

- A number of identical processes, one per processor, execute the parallel algorithm
- Input: Stored as array in global memory
- A stack in global memory stores the indices of the subarrays that are still unsorted
- When a process is without work, it attempts to pop the indices for an unsorted subarray off the global stack.
- If it is successful, the processor partitions the subarray, based on a supposed median element, into two smaller arrays, containing elements less than or equal to the supposed median value or greater the supposed median value, respectively
- After the partitioning step, identical to the partitioning step performed by the serial quicksort algorithm, the process pushes the indices for one subarray onto the global stack of the unsorted subarrays and repeats the partitioning process on the other subarray





# Parallel Quicksort

## QUICKSORT (UMA MULTIPROCESSOR):

```
Global    n          {Size of array of unsorted elements}
          a[0...(n - 1)] {Array of elements to be sorted}
          sorted        {Number of elements in sorted position}
          min.partition {Smallest subarray that is partitioned rather than
                           sorted directly}
Local     bounds      {Indices of unsorted subarray}
          median       {Final position in subarray of partitioning key}
begin
  sorted ← 0
  INITIALIZE.STACK()

  for all  $P_i$ , where  $0 \leq i < p$  do
    while (sorted < n) do
      bounds ← STACK.DELETE()

      while (bounds.low < bounds.high) do
        if (bounds.high - bounds.low < min.partition) then
          INSERTION.SORT (a, bounds.low, bounds.high)
          ADD.TO.SORTED (bounds.high - bounds.low + 1)
        exit while
```

# Parallel Quicksort...

```
else
    median ← PARTITION (bounds.low, bounds.high)
    STACK.INSERT (median + 1, bounds.high)
    bounds.high ← median - 1

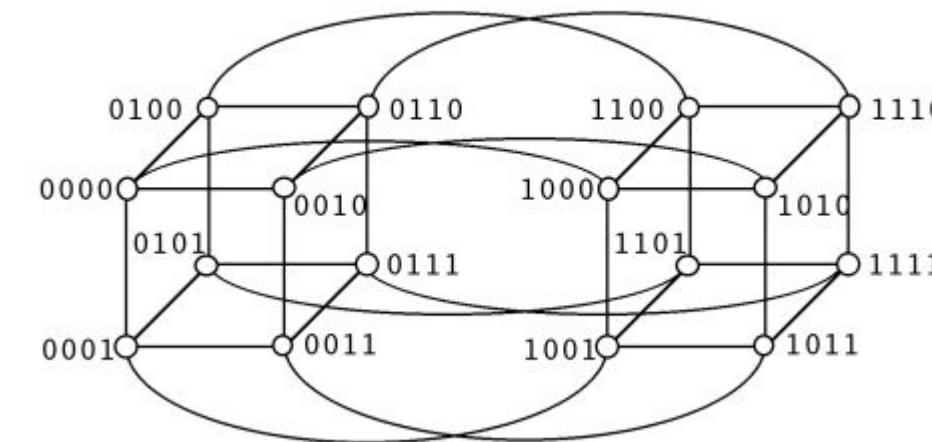
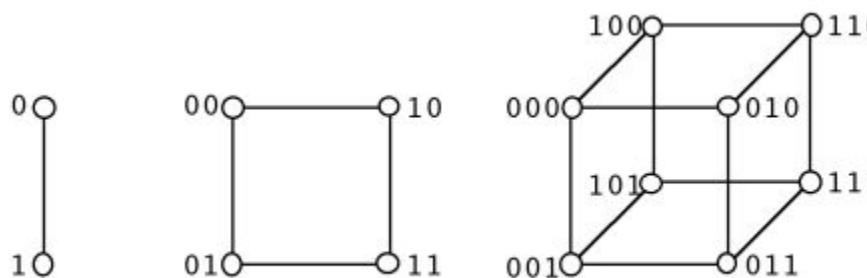
    if bounds.low = bounds.high then
        ADD.TO.SORTED (2)
    else
        ADD.TO.SORTED (1)
    endif
    endif
endwhile
endwhile
endfor
end
```

Multiprocessor-oriented parallel quicksort algorithm. A shared stack contains the indices of unsorted subarrays. Processes must execute functions **STACK.DELETE()**, **ADD.TO.SORTED()**, and **STACK.INSERT()** inside critical sections to ensure mutual exclusion.

# Hypercube Quicksort

- HYPERCUBE TOPOLOGY
  - A  $d$ -dimensional hypercube consists of  $n = 2^d$  processors.
  - Each processor has a number whose binary representation has  $d$  digits.
- Hamming distance: The total number of bit positions at which two binary numbers differ.
- In a hypercube, two processors are connected if their Hamming distance is 1. The connectivity of a  $d$ -dimensional hypercube is thus  $d$ . Since each link can change only one digit, the diameter is  $d$ , or  $\log_2 n$ .

- A hypercube topology is constructed recursively as follows.
  - (1) First a one-dimensional hypercube has two connected processors 0 and 1.
  - (2) A  $(d+1)$ -dimensional hypercube is defined from a  $d$ -dimensional hypercube as follows:
    - a. Duplicate the  $d$ -dimensional hypercube including processor numbers.
    - b. Create links between processors with the same number in the original and duplicate.
    - c. Append a binary 1 to the left of each processor number in the duplicate, and a binary 0 to left of each processor number in the original.



- Divide-and-conquer algorithms are often implemented conveniently with the hypercube topology.
- **NETWORK TOPOLOGY**
  - **Topology:** Which processors are directly connected to which other processors.
  - **Distance:** The number of communication links a message must traverse between two processors in the most direct path.
  - **Diameter:** The maximum distance between any two processors in the network. The diameter measures the maximum delay for transmitting a message from one processor to another.
  - **Connectivity:** The number of incident links on each interface. High connectivity is desirable, because it lowers contention for communication resources, but it also increases the cost.
  - **Bisection width:** The minimum number of communication links that have to be removed to partition the network into two equal halves. The bisection width measures the largest number of messages, which can be sent simultaneously.

- Let  $n$  be the number of elements to be sorted and  $p = 2^d$  be the number of processors in a  $d$ -dimensional hypercube.
- Each processor is assigned a block of  $n/p$  elements.
- The algorithm starts by selecting a common pivot value, which is broadcast to all processors.
- Each processor partitions its local elements into two blocks, one with elements smaller than the pivot, and the other with elements larger than the pivot.
- Then the processors connected along the  $d$ -th communication link exchange blocks: Each processor with a 0 in the  $d$ -th bit retains the smaller elements, and each processor with a 1 in the  $d$ -th bit retains the larger elements.

- After this step, each processor in the  $(d-1)$ -dimensional hypercube whose  $d$ -th bit is 0 has elements smaller than the pivot, and each processor in the other  $(d-1)$ -dimensional hypercube has elements larger than the pivot.
- At the next level, a pivot is chosen in each  $(d-1)$ -dimensional hypercube separately, and it is broadcast to all the processors in each hypercube.
- Each processor partitions its local elements into two blocks, one smaller and the other larger than the pivot.
- Appropriate blocks are exchanged through the  $(d-1)$ -th communication link so that each processor with a 0 in the  $(d-1)$ -th bit retains the smaller elements than the pivot, and each processor with a 1 in the  $(d-1)$ -th bit retains the larger. This procedure is performed recursively.

- After  $d$  such splits, the sequence is sorted with respect to the global ordering imposed on the processors.
- Then each processor sorts its local elements by using sequential quicksort.
  - Dimension      Master of Subcube
    - 3            000
    - 2            x00
    - 1            xx0
  - Partner is obtained by flipping the  $d$ -th bit.

## • PIVOT SELECTION

- Master of each subcube determines the pivot value and broadcasts it to all the processors in the subcube.
- Bad choice of pivot at early stages degrades the performance significantly (no recovery from it).
- Let us use the average value elements in the master processor as a pivot.

$$\text{pivot} = (\Sigma \text{ elements}) / (\text{number of elements})$$

## HYPERQUICKSORT (HYPERCUBE MULTICOMPUTER):

```

Global   n          (Initial number of elements per processor)
        d          (Dimension of hypercube)
        i          (Dimension number of current hypercube)

Local    logical.num (Unique processor number)
        partner    (Processor's partner in the exchange)
        root       (Root processor of current hypercube)
        splitter   (Median of root processor's sorted list)

begin
  for all  $P_j$ , where  $0 \leq j < 2^d$  do
    Sort  $n$  values using sequential quicksort algorithm
    if  $d > 0$  then
      for  $i \leftarrow d$  downto 1 do
        root  $\leftarrow$  root of the binary  $i$ -cube containing processor logical.num
        if logical.num = root then
          splitter  $\leftarrow$  median of the sorted list held by processor logical.num
        endif
        Processor root broadcasts splitter to other processors in binary  $i$ -cube
        Use splitter to partition sorted values into low list, high list
        partner  $\leftarrow$  logical.num  $\otimes 2^{(i-1)}$  { Bitwise exclusive "or" }
        if logical.num > partner then
          Send low list to processor partner
          Receive another high list from processor partner
        else { logical.num < partner }
          Send high list to processor partner
          Receive another low list from processor partner
        endif
        Merge two lists into a single sorted list of values
      endfor
    endif
  endfor
end

```

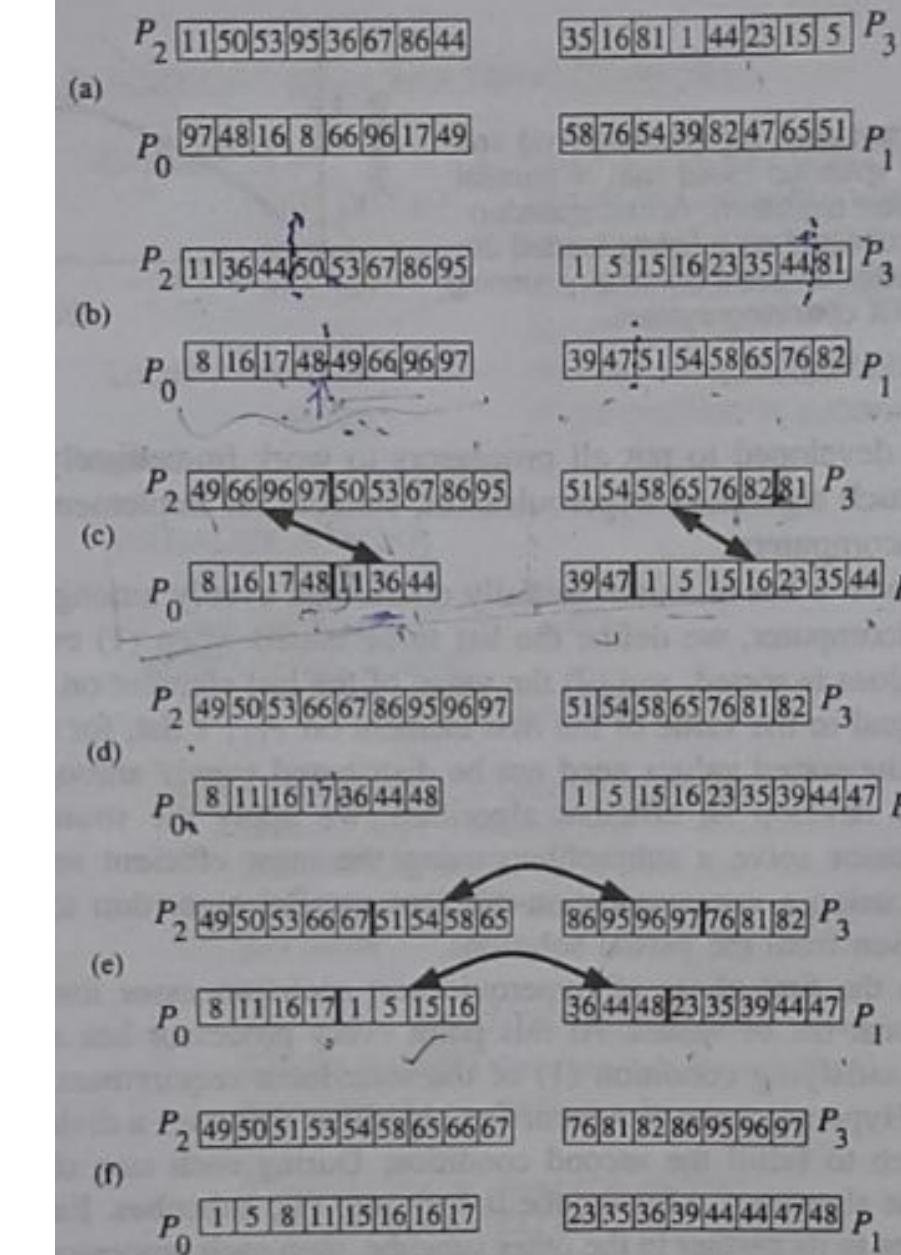


Illustration of the hyperquicksort algorithm. In this example 32 elements are being sorted on a two-dimensional hypercube. (a) Initially, each processor has eight elements. (b) Each processor performs sequential quicksort on its own list. Processor 0 broadcasts its median value, 48, to the other processors. (c) Processors in the lower half of the hypercube send values greater than 48 to processors in the upper half of the hypercube. The processors in the upper half send values less than or equal to 48. (d) Each processor merges the elements it kept with the elements it received. Processor 0 broadcasts its median value to processor 1, and processor 2 broadcasts its median value to processor 3. (e) Processors swap values across another hypercube dimension. (f) Each processor merges the elements it kept with the elements it received. At this point the list is sorted.

- Two-dimensional hypercube  $d=2$
- Processors: 0 to  $2^d-1 \Rightarrow 0$  to 3
- Initial sort using quick sort
- $d>0$ 
  - $i = 2$  to 1
  - $i = 2$ 
    - ✓ Root = 0
    - ✓ Splitter = 48
    - ✓ Root broadcasts 48 to all parallel processors
    - ✓ Processors partition list into low list and high list based on 48
    - ✓ Processors identify partner: logical\_num XOR  $2^{2-1}$ 
      - P0 partner is p2, p1 partner is p3, p2 partner is p0 and p3 partner is p1
    - ✓ Logical\_num > partner (p2 and p3): sends lowlist to partner and receives highlist from partner
    - ✓ Logical\_num < partner (p0 and p1): sends highlist to partner and receives lowlist from partner
    - ✓ Merge list

- $i = 1$ 
  - ✓ Root = 0 and 2
  - ✓ Splitter = 17 and 67
  - ✓ p0 broadcasts 17 to p1 and p2 broadcast 67 to p3
  - ✓ Processors partition list into low list and high list based on splitter
  - ✓ Processors identify partner: logical\_num  $\text{XOR } 2^{1-1}$ 
    - P0 partner is p1, p1 partner is p0, p2 partner is p3 and p3 partner is p2
  - ✓ Logical\_num > partner :
    - p1 sends lowlist to p0, receives highlist from p0
    - p3 sends lowlist to p2, receives highlist from p2
  - ✓ Logical\_num < partner :
    - p0 sends highlist to p1, receives lowlist from p1
    - p2 sends highlist to p3, receives lowlist from p3
  - ✓ Merge list

- Expected case running time

$$\Theta\left(N \log N + \frac{d(d+1)}{2} + dN\right)$$

- $N \log N$  term represents the sequential running time
- $d(d + 1)/2$  term represents the broadcast step
- $dN$  term represents the time required for the exchanging and merging of the sets of elements
- Limitations
  - The number of processors has to be a power of 2
  - Very High communication overhead

**Complexity based on processor (p) and input size (n)**

Iterations =  $\log_2 p$

Select a pivot =  $O(n)$

– keep sublist sorted

Broadcast pivot =  $O(\log_2 p)$

Split the sequence

– split own sequence =  $O(\log n/p)$

– exchange blocks with neighbor =  $O(n/p)$

– merge blocks =  $O(n/p)$



**SASTRA**

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY  
(U/S 3 OF THE UGC ACT, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

# Thank you

## UNIT - III

**Introduction:** Design goals - **Types of distributed systems:** High performance distributed computing - Distributed information systems - Pervasive systems -  
**Architecture:** System architecture

**Communication:** Message-oriented communication - Simple transient messaging with sockets - Advanced transient messaging - Message-oriented persistent communication -  
**Multicast communication:** Application-level tree-based multicasting - Flooding-based multicasting - Gossip-based data dissemination

# Distributed System

- A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

1. A distributed system is a collection of computing elements each being able to behave independently of each other.

A computing element → a node

- It can be either a hardware device or a software process.

2. Users (be they people or applications) believe that they are dealing with a single system.

## Characteristic 1: Collection of autonomous computing elements

- Modern distributed systems consist of all kinds of nodes ranging from very big high-performance computers to smaller devices.
- In practice, nodes are programmed to achieve common goals, which are realized by exchanging messages with each other.
- A node reacts to incoming messages, which are then processed and, in turn, leading to further communication through message passing.

### Consequence of dealing with independent nodes

- each one will have its own notion of time
  - synchronization and coordination

- we are dealing with a collection of nodes implies that we may also need to manage the membership and organization of that collection.
- **Open group:** Any node is allowed to join the distributed system, effectively meaning that it can send messages to any other node in the system.
- **Closed group:** Only the members of that group can communicate with each other and a separate mechanism is needed to let a node join or leave the group.

→A mechanism is needed to authenticate a node -- > **Confidentiality & trust issues**

- A distributed system is often organized as an **overlay network**.

**Structured overlay:** In this case, each node has a well-defined set of neighbors with whom it can communicate.

For example, the nodes are organized in a tree or logical ring.

**Unstructured overlay:** In these overlays, each node has a number of references to randomly selected other nodes.

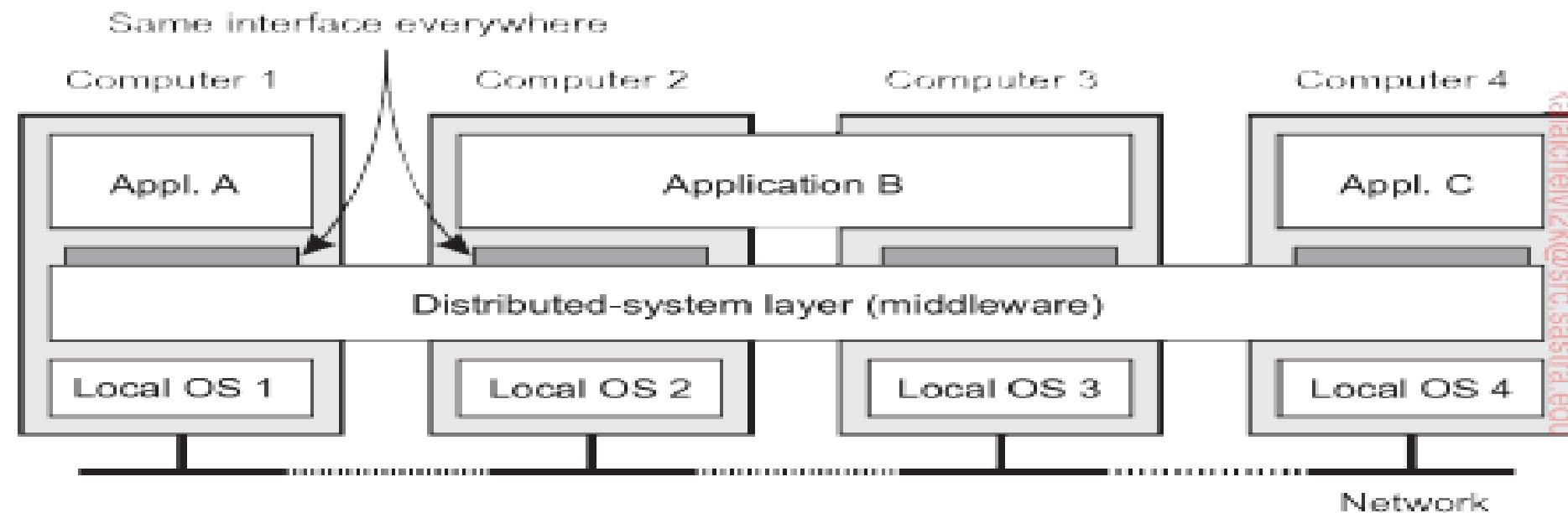
- In any case, an overlay network should always be **connected**, meaning that between any two nodes there is always a communication path allowing those nodes to route messages from one to the other.

## Characteristic 2: Single coherent system

- In a single coherent system the collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place.

## Middleware and distributed systems:

- Distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system.



**Figure 1.1: A distributed system organized in a middleware layer, which extends over multiple machines, offering each application the same interface.**

- Facilities for interapplication communication.
- Security services.
- Accounting services.
- Masking of and recovery from failures

## 1. Communication:

→ RPC

## 2. Transactions:

- Middleware generally offers special support for executing such services in an **all-or-nothing fashion**, commonly referred to as an atomic **transaction**.

## 3. Service composition:

- Web-based middleware can help by standardizing the way Web services are accessed and providing the means to generate their functions in a specific order.

- A simple example of how service composition is deployed is formed by **mashups**:
- Web pages that combine and aggregate data from different sources.

### Example:

**Google maps** in which maps are enhanced with extra information such as trip planners or real-time weather forecasts

**4. Reliability :** To build an application as a group of processes such that any message sent by one process is guaranteed to be received by all or no other process.

# DESIGN GOALS

1. Make Resources Accessible
2. Transparency
3. Openness
4. Scalability

## 1. Make Resources Accessible:

- Access resources and share them in a controlled and efficient way.

Printers, computers, storage facilities, data, files, Web pages, and networks, ...

- Easy for users to access remote resources.

## 2. Transparency:

- To hide the fact that **processes** and **resources** are physically distributed across multiple computers.
- Concealment from the users and the application programmers of the fact that the processes and resources of a distributed system are physically distributed across multiple computers.

- Access Transparency
- Location Transparency
- Migration Transparency
- Relocation Transparency
- Replication Transparency
- Concurrency Transparency
- Failure Transparency

## 1. Access Transparency:

- Hide differences in data representation and how a resource is accessed
- enables local and remote resources to be accessed using identical operations

Example: An API for files that uses the same operations to access both local and remote files.

## 2. Location Transparency:

- Hide where a resource is located
- enables resources to be accessed without knowledge of their physical location
- Resources are referred by location transparent logical names that contain no information about the physical location of the resource

Example: URLs of Web pages are location transparent

### 3. Migration Transparency:

- Hide that a resource may move to another location.
- enables resources to be moved without affecting how they can be accessed

#### Example:

A Web page can be moved to a different location without having its URL changed.

### 4. Relocation Transparency:

- hide that a resource may be moved to another location while in use.
- Relocation transparency enables resources to move while in use without being noticed by users and applications

#### Example:

Mobile users can continue to use their laptops while moving from place to place without being disconnected from the Internet

## 5. Replication Transparency:

- hide that a resource is replicated.
- enables multiple instances of resources to be used to increase availability and performance without knowledge of the replicas by users.

### Example:

- replicated web contents

## 6. Concurrency Transparency:

- hide that a resource may be shared by several competitive users.
- enables users and applications to access shared resources without interference between each other
  - Concurrent access to a shared resource should leave that resource in a consistent state
  - Consistency can be achieved using locks or transactions

## 7. Failure Transparency:

- hide the failure and recovery of a resource
- enables users and application programs to complete their tasks despite the failure of hardware or software components

### Example:

e-mail delivery

### 3. Openness:

- An open distributed system is a system that *offers services* according to published standards that describe the *syntax and semantics* of those services

#### Example:

Internet is an open system as the specifications of Internet protocols are published in RFCs

- Services in distributed systems are generally specified through interfaces, *which are often described in an Interface Definition Language (IDL)*
- Interface definitions written in an IDL specify the
  - Syntax of the services (i.e., the names of the functions that are available, the types of the parameters, return values, and possible exceptions that can be raised)
  - Semantics of interfaces are specified in an informal way by means of natural language.

## Benefits of Open Distributed Systems:

- **Interoperability:** Components written by different programmers can easily work together.
- **Portability:** Applications can be easily ported between different distributed systems that implement the same interfaces.
- **Extensibility:** New services can be easily added and old services can be easily re-implemented.

## 4. Scalability

- Distributed system operate **effectively** and **efficiently** at many different scales, ranging from a small intranet to the internet.
- A system is described as scalable if will remain effective when there is a **significant increase** in the number of resources and the number of users.

### Scalability problems:

#### Size scalability:

- As the number of users and resources increase, the system may become overloaded.

#### Geographical scalability:

- As the distance between nodes increases, communication delay becomes significant.

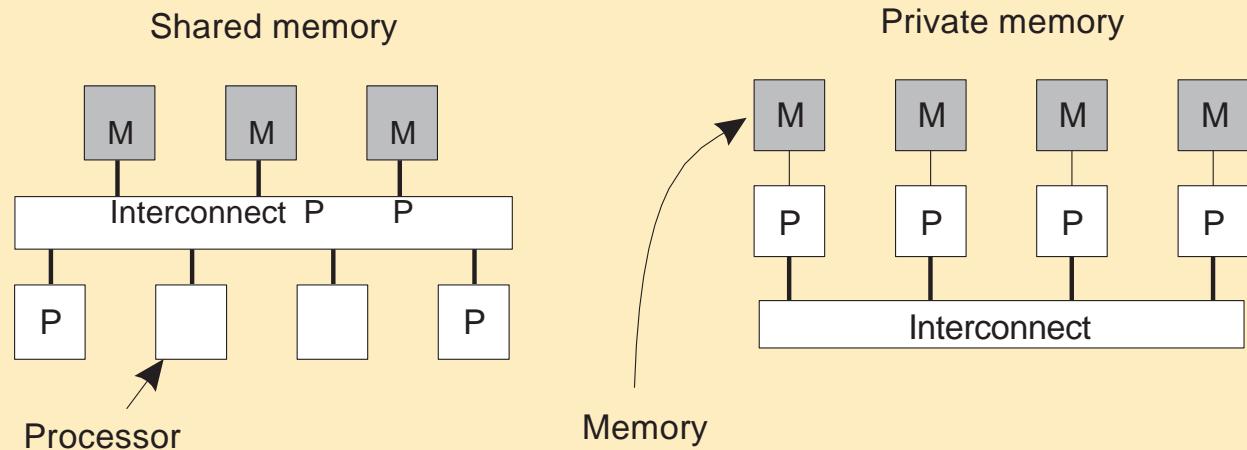
# Types of distributed systems

- **High performance distributed computing**
- **Distributed information systems**
- **Pervasive systems**

## Observation

**High-performance distributed computing started with parallel computing**

### Multiprocessor and multicore versus multicompiler



# High performance distributed computing:

## Cluster computing

- Underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high-speed local-area network.
- In addition, each node runs the same operating system.

## Grid computing:

- Each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.

## Cloud Computing:

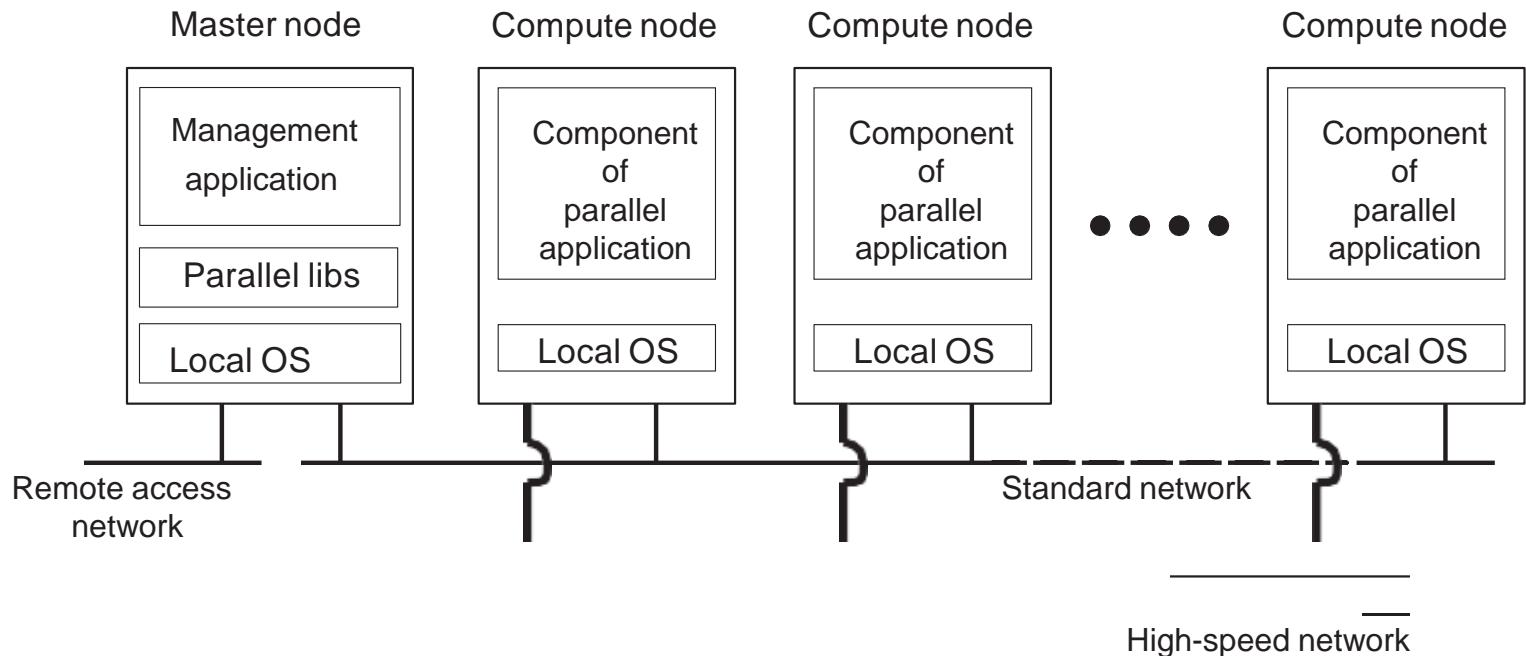
- Simply outsource the entire infrastructure that is needed for compute-intensive applications.
- Providing the facilities to dynamically construct an infrastructure and compose what is needed from available services.

# Cluster computing

Essentially a group of high-end systems connected through a LAN

- **Homogeneous:** same OS, near-identical hardware
- Single managing node

- Cluster computing is used for parallel programming in which a single program is run in parallel on multiple machines.
- Each cluster consists of a collection of compute nodes that are controlled and accessed by means of a single master node.
- Master handles the allocation of nodes to a particular parallel program, maintains a batch queue of submitted jobs, and provides an interface for the users of the system.



# Grid computing

The next step: lots of nodes from everywhere

- Heterogeneous
- Dispersed across several organizations
- Can easily span a wide-area network

- A key issue in a grid-computing system is that resources from different organizations are brought together to allow the collaboration of a group of people from different institutions, indeed forming a federation of systems. Such a collaboration is realized in the form of a **virtual organization**.
- The processes belonging to the same virtual organization have access rights to the resources that are provided to that organization.

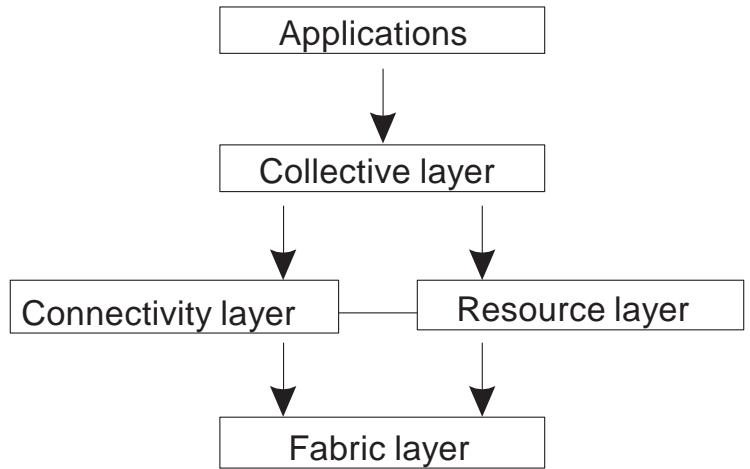
## Note

To allow for collaborations, grids generally use **virtual organizations**.

In essence, this is a grouping of users that will allow for authorization on resource allocation.

# Architecture for grid computing

## The layers



**Fabric:** Provides interfaces to local resources (for querying state and capabilities, locking, etc.)

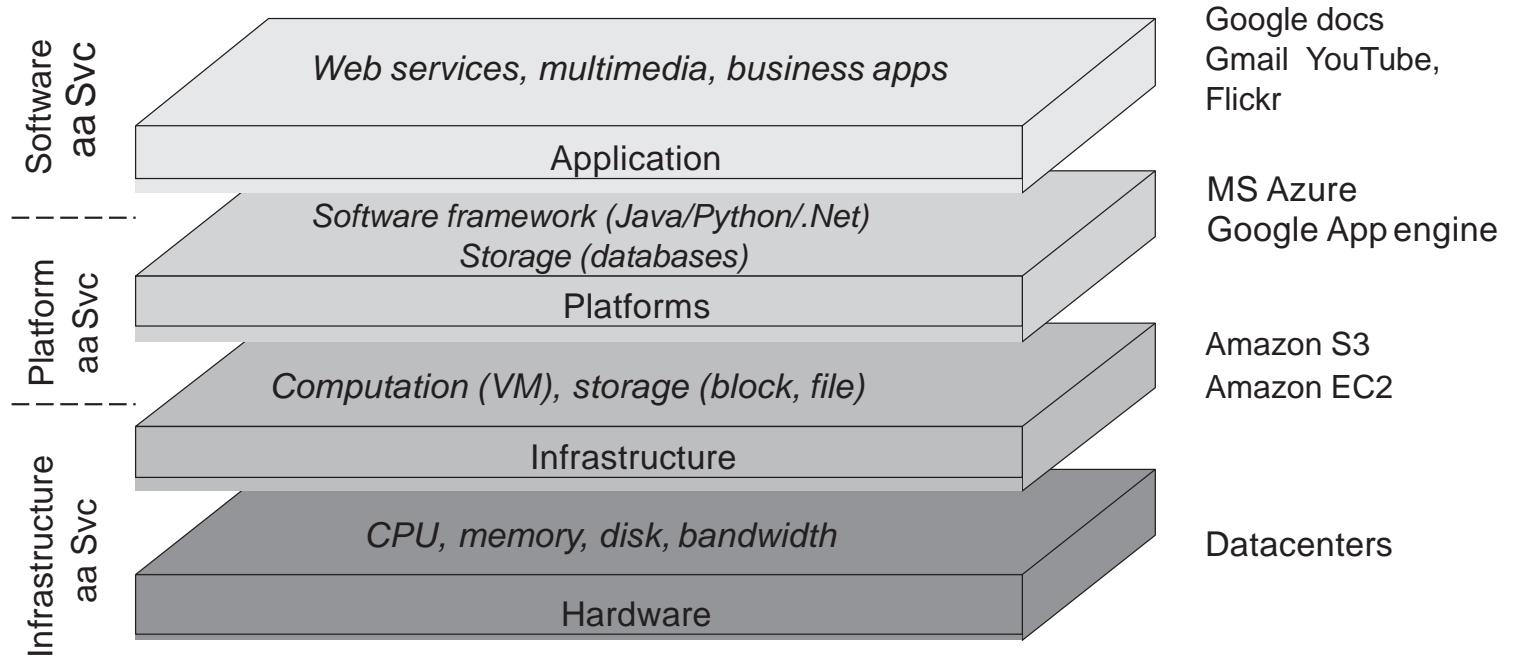
**Connectivity:** Communication/transaction protocols, e.g., for moving data between resources. Also various authentication protocols.

**Resource:** Manages a single resource, such as creating processes or reading data.

**Collective:** Handles access to multiple resources: discovery, scheduling, replication.

**Application:** Contains actual grid applications in a single organization.

# Cloud computing



## Make a distinction between four layers

**Hardware:** Processors, routers, power and cooling systems. Customers normally never get to see these.

**Infrastructure:** Deploys virtualization techniques. Evolves around allocating and managing virtual storage devices and virtual servers.

**Platform:** Provides higher-level abstractions for storage and such. Example: Amazon S3 storage system offers an API for (locally created) files to be organized and stored in so-called **buckets**.

**Application:** Actual applications, such as office suites (text processors, spreadsheet applications, presentation applications). Comparable to the suite of apps shipped with OSes.

# Distributed information systems

## Integrating applications

### Situation

Organizations confronted with many **networked applications**, but achieving interoperability was painful.

### Basic approach

A networked application is one that runs on a **server** making its services available to remote **clients**. Simple integration: clients combine requests for (different) applications; send that off; collect responses, and present a coherent result to the user.

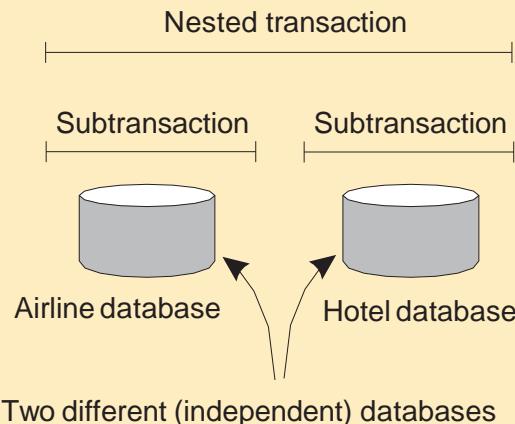
Allow direct application-to-application communication, leading to **Enterprise Application Integration**.

# Example EAI: (nested) transactions

## Transaction

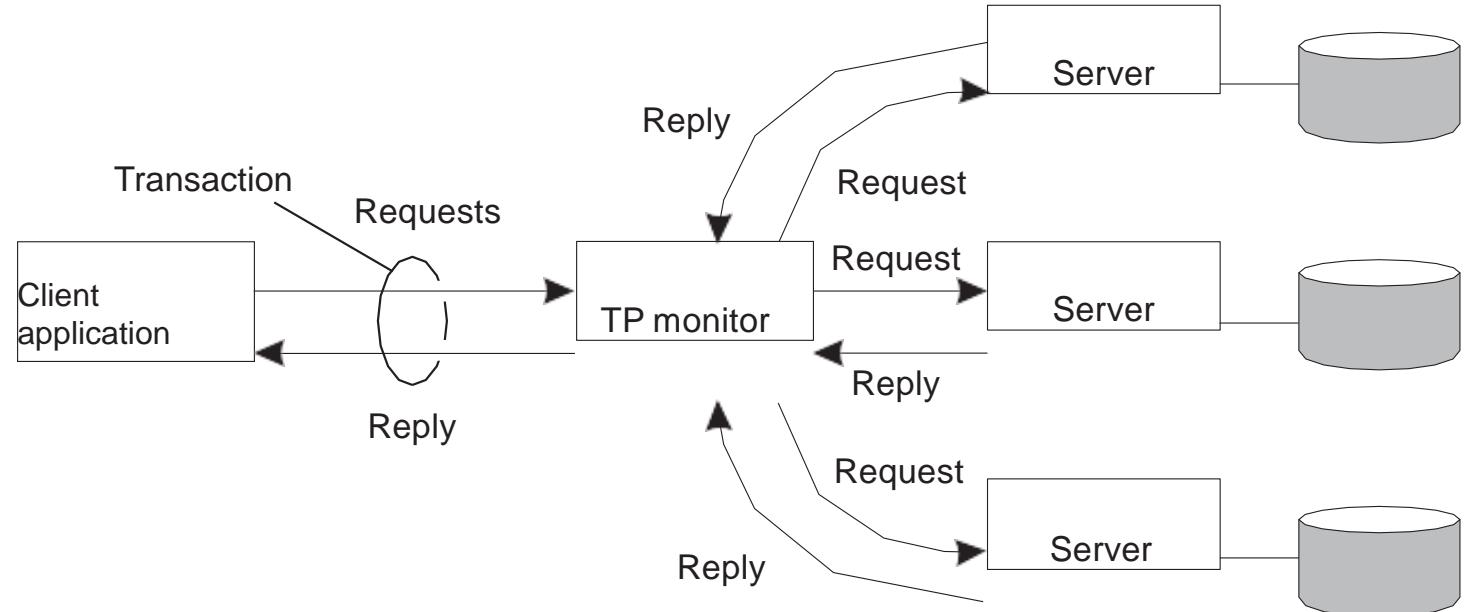
Primitive	Description
<i>BEGIN TRANSACTION</i>	Mark the start of a transaction
<i>END TRANSACTION</i>	Terminate the transaction and try to commit
<i>ABORT TRANSACTION</i>	Kill the transaction and restore the old values
<i>READ</i>	Read data from a file, a table, or otherwise
<i>WRITE</i>	Write data to a file, a table, or otherwise

## Issue: all-or-nothing



- **Atomic:** happens indivisibly (seemingly)
- **Consistent:** does not violate system invariants
- **Isolated:** not mutual interference
- **Durable:** commit means changes are permanent

# TPM: Transaction Processing Monitor

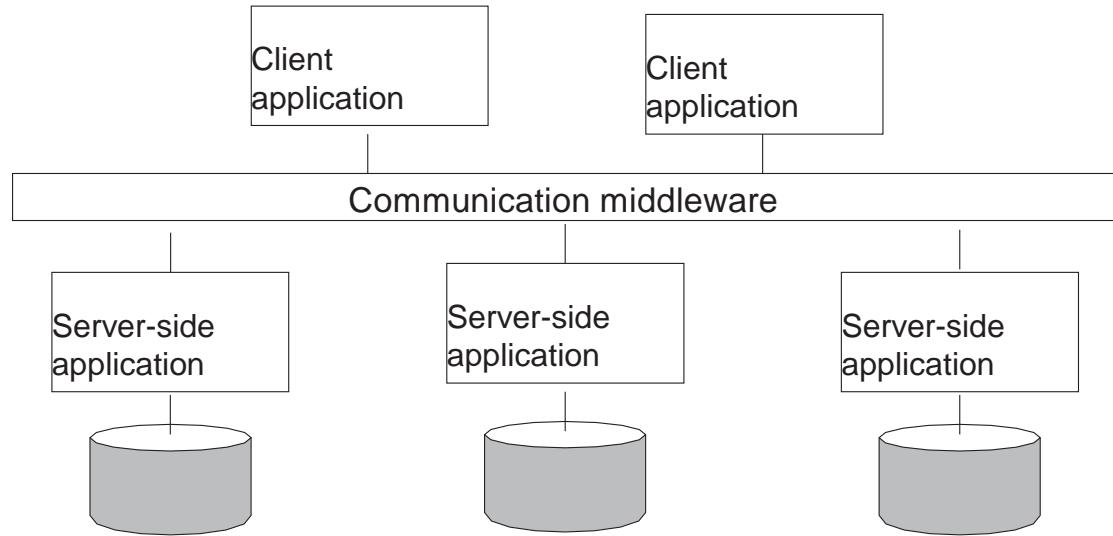


## Observation

In many cases, the data involved in a transaction is distributed across several servers.

A **TP Monitor** is responsible for coordinating the execution of a transaction.

# Middleware and EAI



**Middleware offers communication facilities for integration**

**Remote Procedure Call (RPC):** Requests are sent through local procedure call, packaged as message, processed, responded through message, and result returned as return from call.

**Message Oriented Middleware (MOM):** Messages are sent to logical contact point ([published](#)), and forwarded to [subscribed](#) applications.

# Distributed Pervasive systems

## Observation

Emerging next-generation of distributed systems in which nodes are small, mobile, and often embedded in a larger system, characterized by the fact that the system **naturally blends into the user's environment**.

## Three (overlapping) subtypes

**Ubiquitous computing systems**: pervasive and **continuously present**, i.e., there is a continuous interaction between system and user.

**Mobile computing systems**: pervasive, but emphasis is on the fact that devices are **inherently mobile**.

**Sensor (and actuator) networks**: pervasive, with emphasis on the actual (collaborative) **sensing** and **actuation** of the environment.

# Ubiquitous systems

## Core elements

**Distribution:** Devices are networked, distributed, and accessible in a transparent manner

**Interaction:** Interaction between users and devices is highly unobtrusive

**Context awareness:** The system is aware of a user's context in order to optimize interaction

**Autonomy:** Devices operate autonomously without human intervention, and are thus highly self-managed

**Intelligence:** The system as a whole can handle a wide range of dynamic actions and interactions

# Mobile computing

## Distinctive features

- A myriad of different mobile devices (smartphones, tablets, GPS devices, remote controls, active badges).
- Mobile implies that a device's location is expected to change over time ⇒ change of local services, reachability, etc. Keyword: **discovery**
- Communication may become more difficult: no stable route, but also perhaps no guaranteed connectivity ⇒ **disruption-tolerant networking**

# Sensor networks

## Characteristics

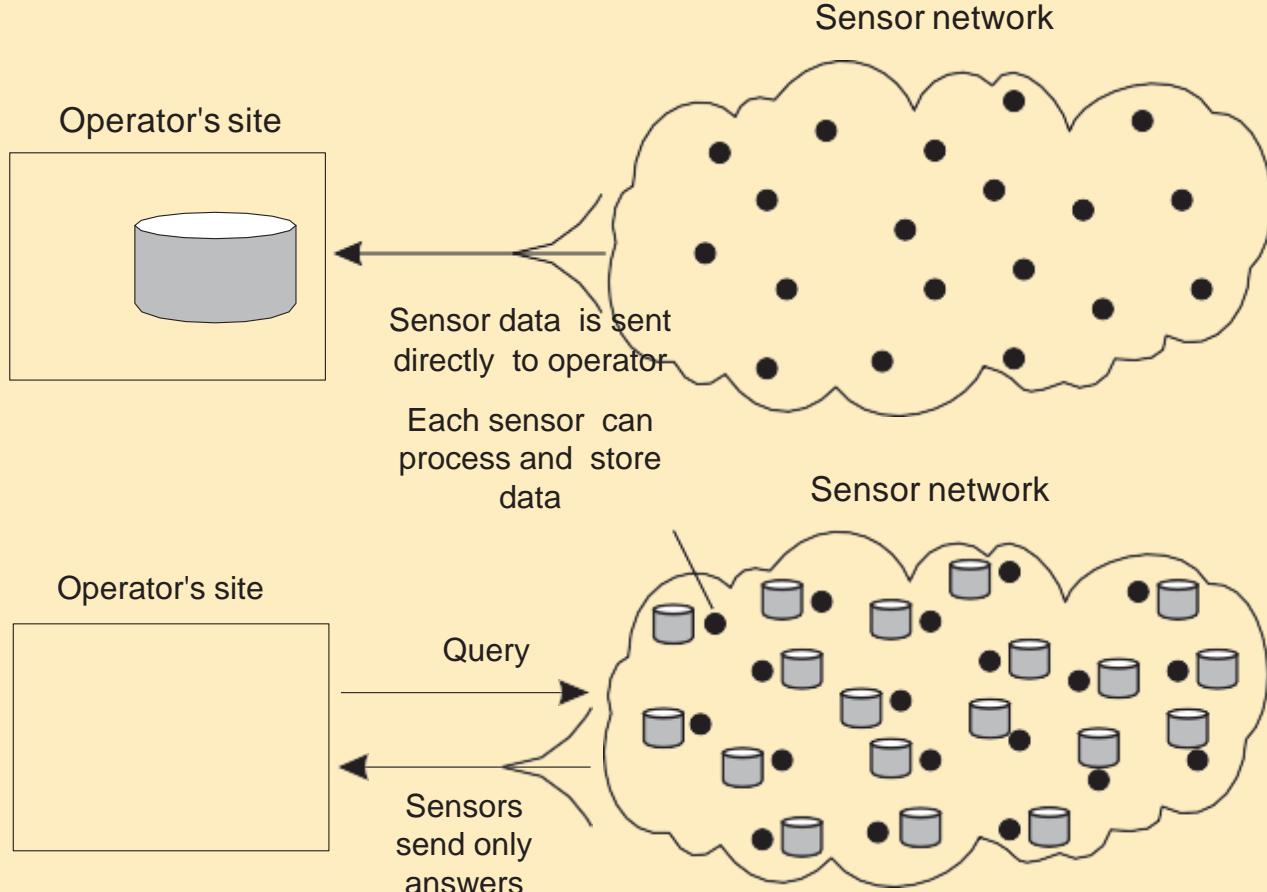
The **nodes** to which sensors are attached are:

Many (10s-1000s)

Simple (small memory/compute/communication capacity) Often battery-powered (or even battery-less)

# Sensor networks as distributed databases

## Two extremes



# System Architecture

- Deciding software components, their interaction, and their placement leads to an instance of a software architecture, also known as a **system architecture**.

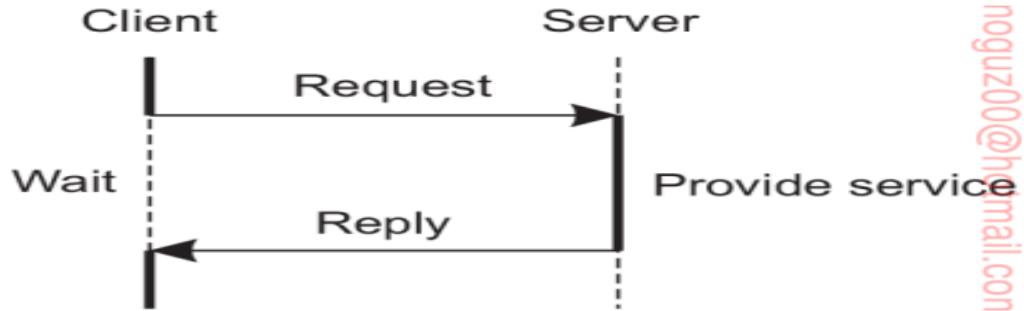
- **Centralized organizations**
- **Decentralized organizations**
- **Hybrid Architectures**



# Centralized organizations

## Simple client-server architecture:

- Processes in a distributed system are divided into two groups.
- A **server** is a process implementing a specific service
  - Example: a file system service or a database service.
- A **client** is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply.
- This client-server interaction, also known as **request-reply behavior**.



noguz00@gmail.com

**Figure 2.15:** General interaction between a client and a server.

Communication between a client and a server can be implemented by means of a simple connectionless protocol.

- When a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data.
- The message is then sent to the server.
- Server process it, and package the results in a reply message that is then sent to the client.

- As long as messages do not get lost or corrupted, the request/reply protocol works fine.
- Unfortunately, making the protocol resistant to occasional transmission **failures** is not trivial.
  - Client resend the request when no reply message comes in.
  - But, client cannot detect whether the **original request message was lost**, or that **transmission of the reply failed**.
  - **If the reply was lost, then resending a request may result in performing the operation twice.**
- When an operation can be repeated multiple times without harm, it is said to be **idempotent**.

- As an alternative, many client-server systems use a reliable connection oriented protocol.
- whenever a client requests a service, it first sets up a connection to the server before sending the request.
- The server uses that same connection to send the reply message, after which the connection is torn down.



### Problems:

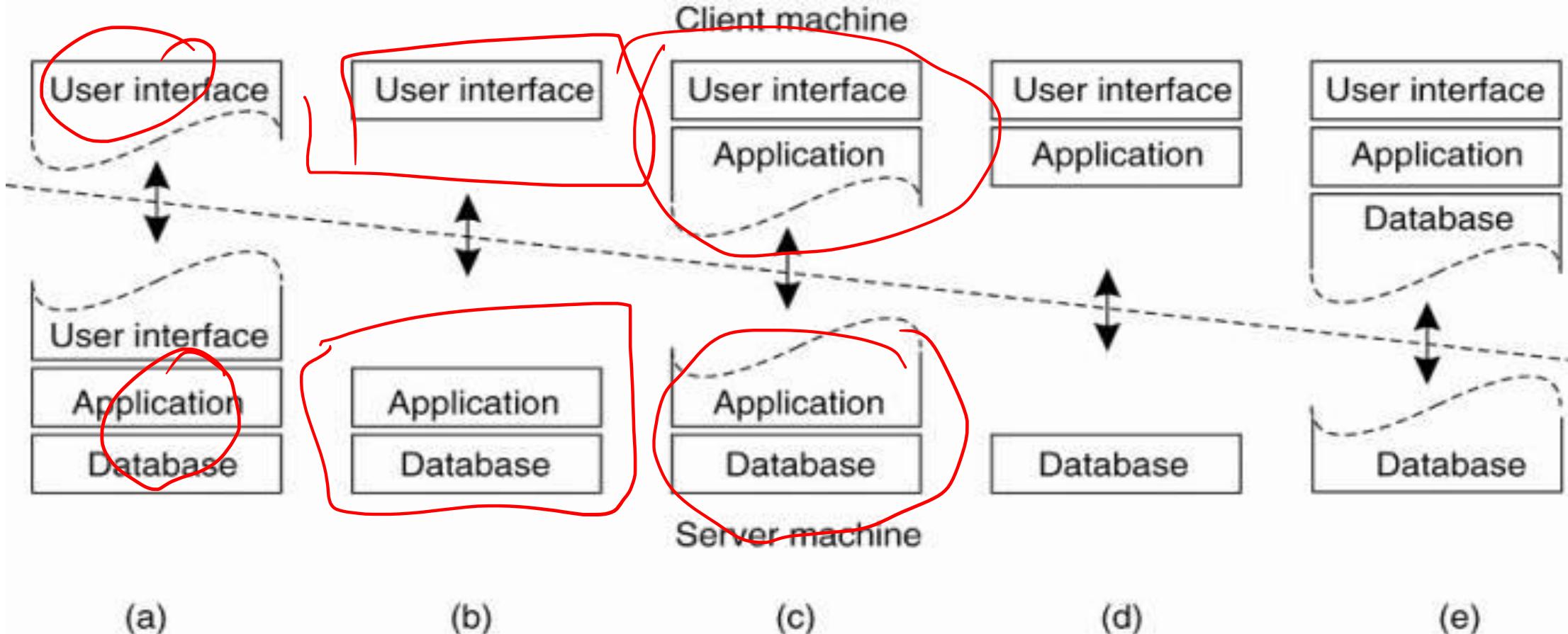
- Setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small.
- No clear distinction between a client and a server.

### Example:

- A server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables.
- In such a case, the database server itself only processes the queries.

# Multi-tiered Architectures

- Two types of machines:
- A client machine containing only the programs implementing the user-interface level.
- A server machine containing the rest, that is, the programs implementing the processing and data level.
- Distributed applications are divided into the three layers:
  - (1) User interface layer
  - (2) Processing layer
  - (3) Data layer
- Two kinds of machines:
- Client machines and server machines → referred to as a **Two-tiered architecture.**



**Client-server organizations in a two-tiered architecture.**

## Cases:

**A:** only the terminal-dependent part of the user interface on the client machine

**B:** Place the entire user-interface software on the client side

- o Divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol.
- O the front end (the client software) does no processing other than necessary for presenting the application's interface

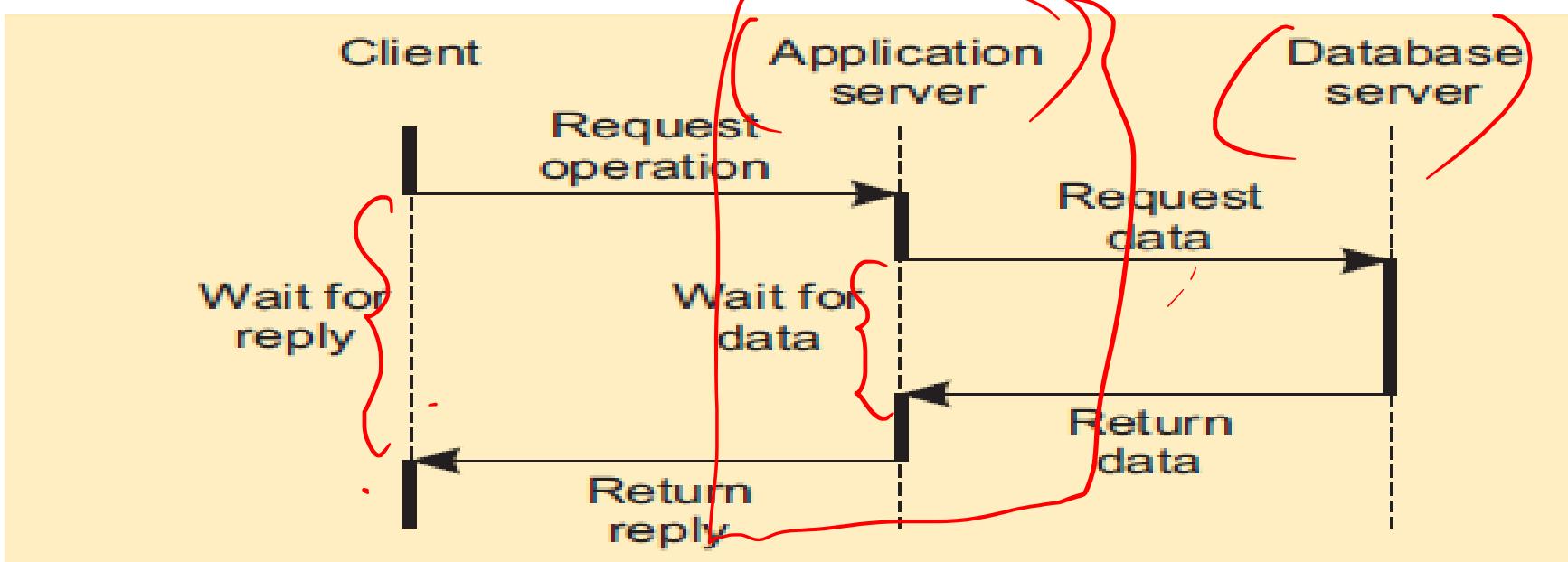
**C:** Move part of the application to the front end

- o e.g. the application makes use of a form that needs to be filled in entirely before it can be processed
- o front end can then check the correctness and consistency of the form, and where necessary interact with the user

- D. Used where the client machine is a PC or workstation, connected through a network to a distributed file system or database
- o most of the application is running on the client machine, but all operations on files or database entries go to the server
  - o e.g. many banking applications run on an end-user's machine where the user prepares transactions and such once finished, the application contacts the database on the bank's server and uploads the transactions for further processing

E: Used where the client machine is a PC or workstation, connected through a network to a distributed file system or database the situation where the client's local disk contains part of the data

## • Three-tier architecture



An example of a server acting as client.

- Server may sometimes need to act as a client
- In this architecture, part of the processing layer are executed by a separate server, but may additionally be partly distributed across the client and server machines
- Ex: TPM, Website Organization

# Decentralized organizations

- Distributed processing is equivalent to organizing a client-server application as a multi-tiered architecture.

## Vertical distribution:

Achieved by placing logically different components on different machines.

- Term is related to the concept of vertical fragmentation as used in distributed relational databases, where it means that tables are split column-wise, and subsequently distributed across multiple machines

## Horizontal distribution :

- A client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set.

e.g **peer-to-peer systems.**

- Processes are all equal: the functions that need to be carried out are represented by every process → each process will act as a client and a server at the same time (i.e., acting as a servant)



### Structured Peer-to-Peer Architectures:

The P2P overlay network consists of all the participating peers as **network nodes**.

- There are links between any two nodes that know each other:  
i.e. if a participating peer knows the location of another peer in the P2P network, then there is a directed edge from the former node to the latter in the overlay network.
- Based on how the nodes in the overlay network are linked to each other, we can classify the P2P networks as **unstructured or structured**.
- Some well known **structured P2P networks** are Chord, Pastry, Tapestry, CAN, and Tulip

# Structured peer-to-peer systems

- In a structured peer-to-peer system, the nodes are organized in an overlay that adheres to a specific, deterministic topology: a ring, a binary tree, a grid, etc
- This topology is used to efficiently look up data.
- Characteristic for structured peer-to-peer systems:  
 → based on semantic-free index i.e., each data item that is to be maintained by the system, is uniquely associated with a key  
 $\text{key}(\text{data item}) = \text{hash}(\text{data item's value}).$
- The peer-to-peer system as a whole is now responsible for storing (key,value) pairs.  
 Most-used procedure - organize the processes through a Distributed Hash Table (DHT).

- Now, structured peer-to-peer systems being able to look up a data item by means of its key.
- i.e., the system provides an efficient implementation of a function lookup that maps a key to an existing node:

existing node = lookup(key).



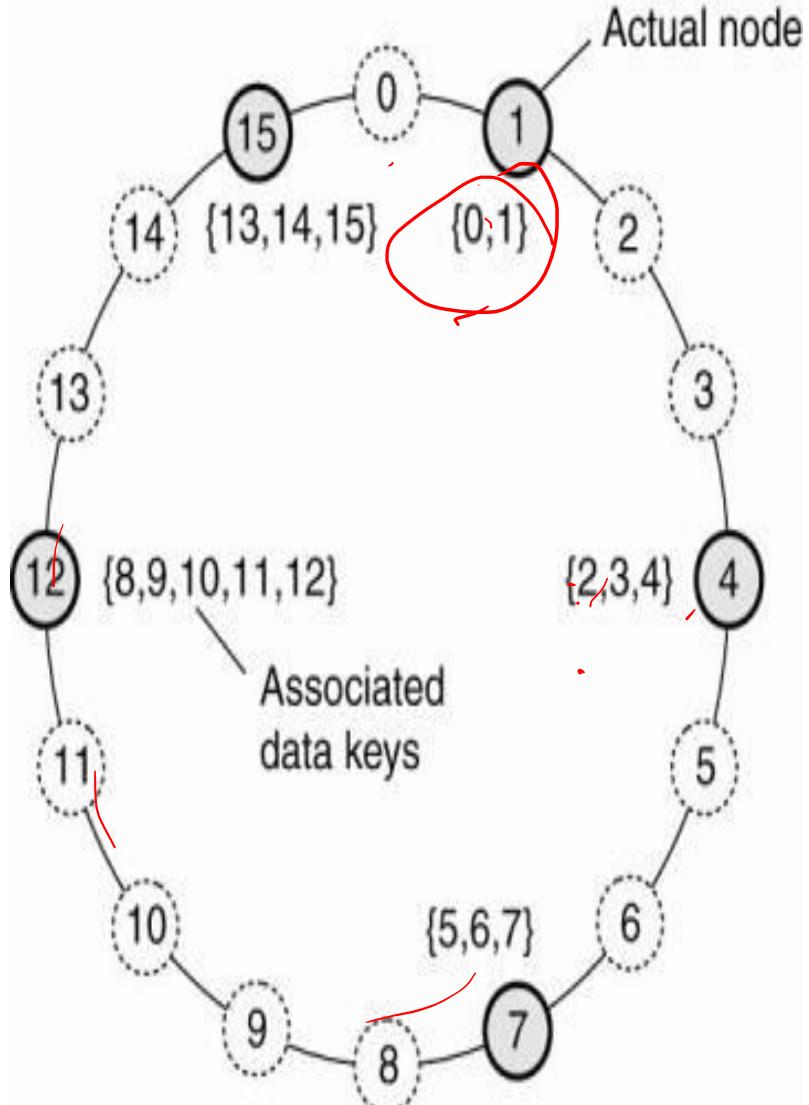
- **Chord system:** the nodes are logically organized in a ring such that a data item with key  $k$  is mapped to the node with the smallest identifier  $\text{id}_k$ .

- This node is referred to as the successor of key  $k$  and denoted as  $\text{succ}(k)$

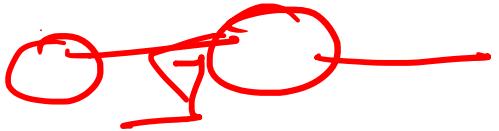
*succ(k)*

- To look up the data item an application running on an arbitrary node would then call the function LOOKUP( $k$ ) which would subsequently return the network address of  $\text{succ}(k)$ .

- At that point, the application can contact the node to obtain a copy of the data item.
- The mapping of data items onto nodes in Chord.



- Looking up a key does not follow the logical organization of nodes in the ring.
- Each node will maintain shortcuts to other nodes → lookups can generally be done in  $O(\log(N))$  number of steps, where  $N$  is the number of nodes participating in the overlay

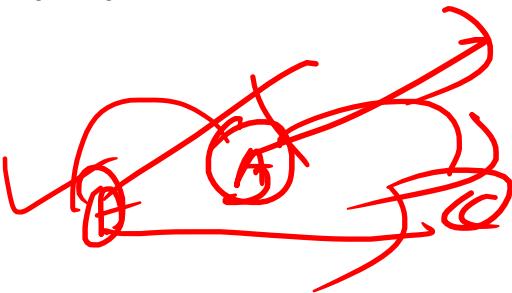


## Joining the P2P Network

1. When a node wants to join the system, it starts with generating a random identifier id.
2. Then, the node can simply do a lookup on id, which will return the network address of  $\text{succ}(\text{id})$
3. The joining node then contacts  $\text{succ}(\text{id})$  and its predecessor and insert itself in the ring.
  - a. This scheme requires that each node also stores information on its predecessor.
  - b. Insertion also yields that each data item whose key is now associated with node id, is transferred from  $\text{succ}(\text{id})$ .

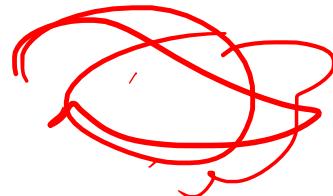
## Leaving the P2P Network

1. node id informs its departure to its predecessor and successor
2. transfers its data items to  $\text{succ}(\text{id})$ .

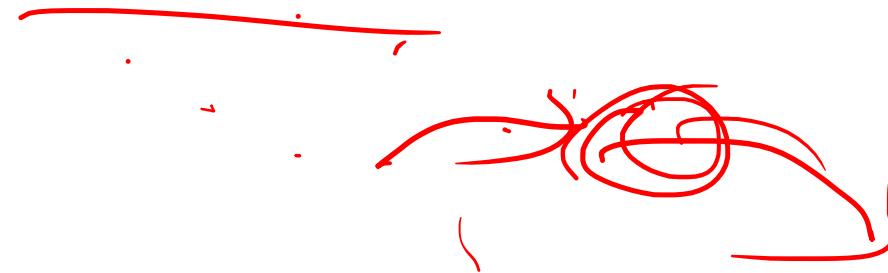


# Unstructured Peer-to-Peer Architectures

- An unstructured P2P network is formed when the overlay links are established arbitrarily.
- Such networks can be easily constructed as a new peer that wants to join the network can copy existing links of another node and then form its own links over time.
- In an unstructured P2P network, if a peer wants to find a desired piece of data in the network, the query has to be **flooded** through the network in order to find as many peers as possible that share the data.
- Main disadvantage - queries may not always be resolved.
- Flooding also causes a **high amount of signaling traffic in the network** and hence such networks typically have ~~very poor search efficiency~~.

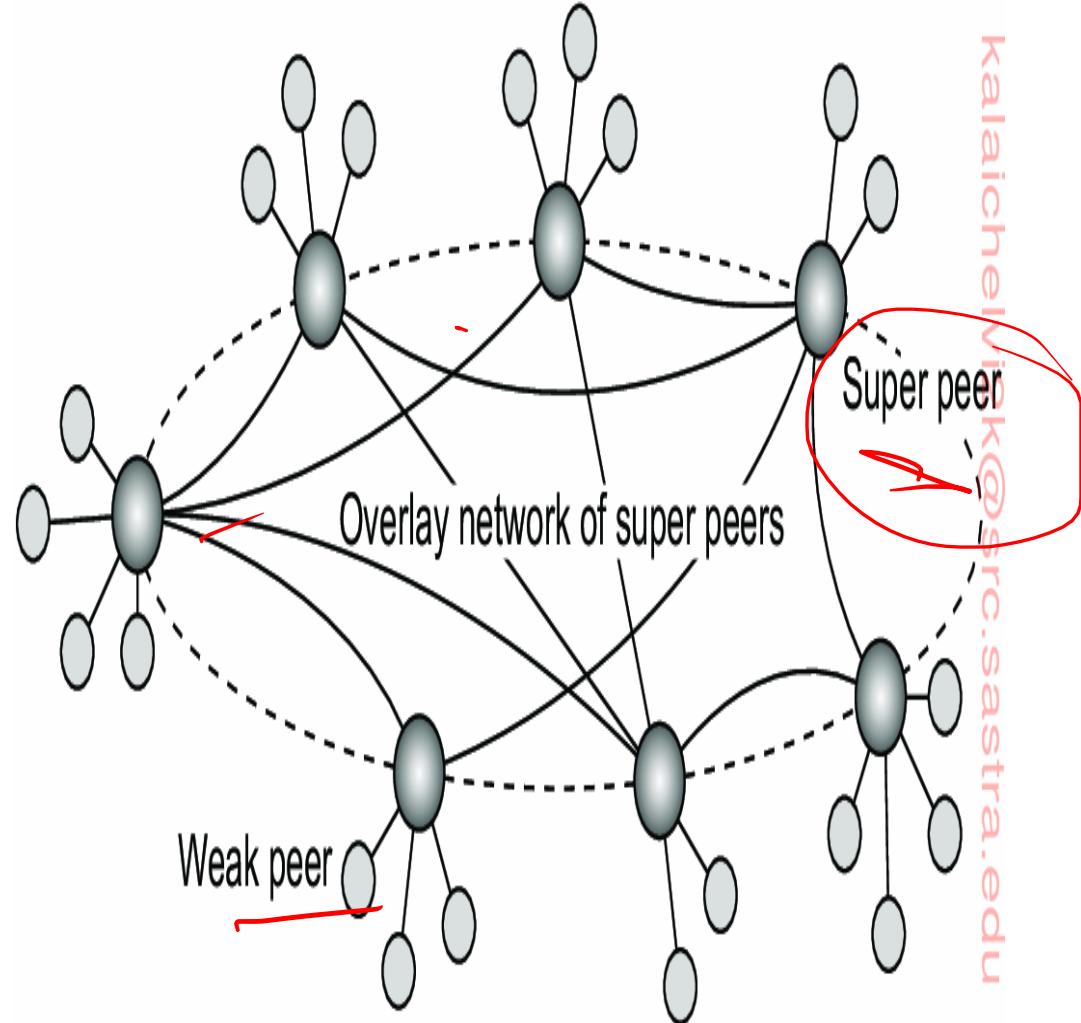


- At the other end of the search spectrum, an issuing node  $u$  can simply try to find a data item by asking a randomly chosen neighbor, say  $v$ .
  - If  $v$  does not have the data, it forwards the request to one of its randomly chosen neighbors, and so on.
  - The result is known as a **random walk**
- less network traffic



# Hierarchically organized peer-to-peer network

- Consider a collaboration of nodes that offer resources to each other.
- Example:
- In a collaborative **content delivery network (CDN)**, nodes may offer storage for hosting copies of Web documents allowing Web clients to access pages nearby, and thus to access them quickly.
- In that case, making use of a **broker** that collects data on resource usage and availability for a number of nodes that are in each other's proximity will allow to quickly select a node with sufficient resources.
- Those maintaining an index or acting as a broker are generally referred to as **super peers**.



- In this organization, every regular peer, now referred to as a **weak peer**, is connected as a client to a super peer.
- All communication from and to a weak peer proceeds through that peer's associated super peer.
- The client-super peer relation is fixed n many cases.
- whenever a regular peer joins the network, it attaches to one of the super peers and remains attached until it leaves the network.
- Expected that super peers are long-lived processes with a high availability.
- To compensate for potential unstable behavior of a super peer, backup schemes can be deployed, such as pairing every super peer with another one and requiring clients to attach to both.

# Hybrid Architectures

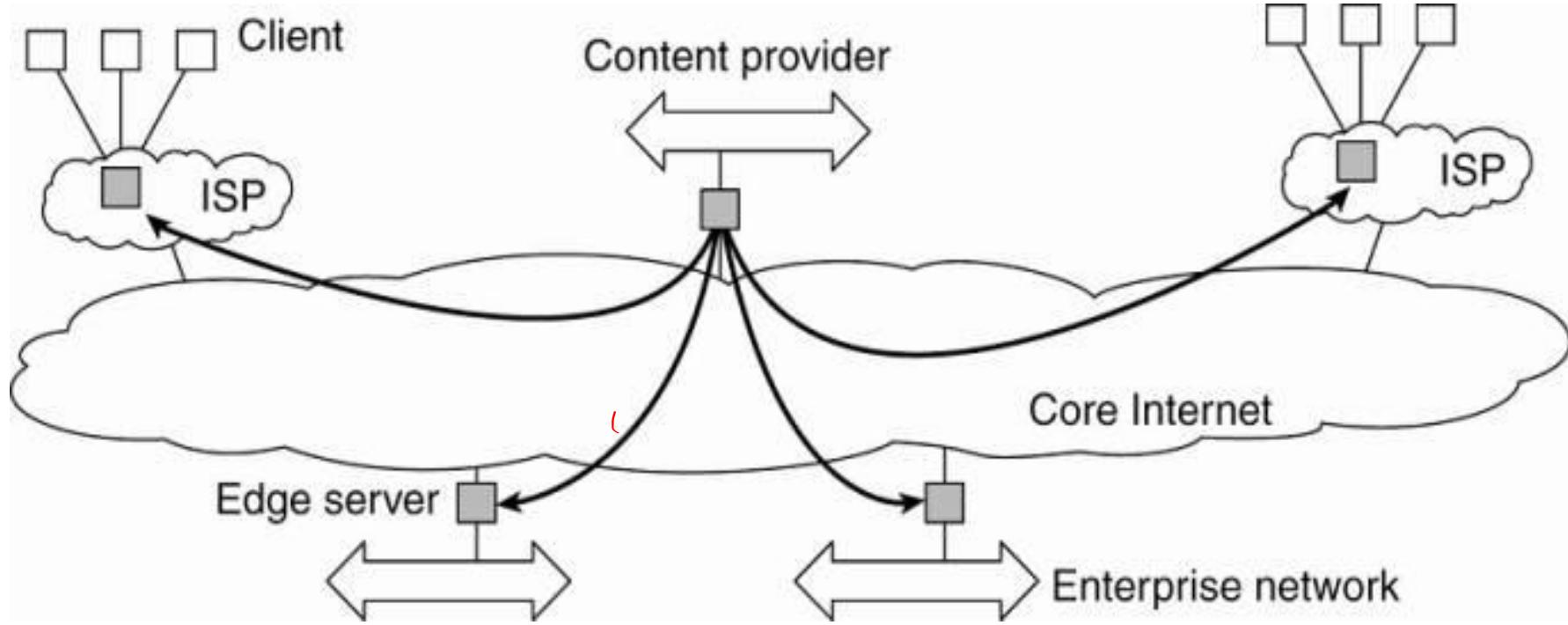
- Distributed systems in which client-server solutions are combined with decentralized architectures.

## Edge-Server Systems

- Deployed on the Internet where servers are placed "at the edge" of the network.
- Purpose is to serve content, possibly after applying filtering and transcoding functions
- a collection of edge servers can be used to optimize content and application distribution
- This edge is formed by the boundary between enterprise networks and the actual Internet



- e.g, an Internet Service Provider (ISP).
- e.g. end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet.



**Viewing the Internet as consisting of a collection of edge servers.**

- one edge server acts as an origin server from which all content originates.
- That server can use other edge servers for replicating Web pages

## Collaborative Distributed Systems

- Hybrid architectures are deployed in collaborative distributed systems.

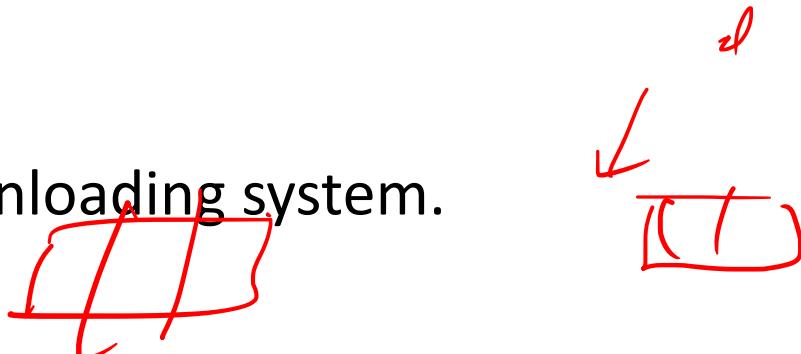
### Two step process:

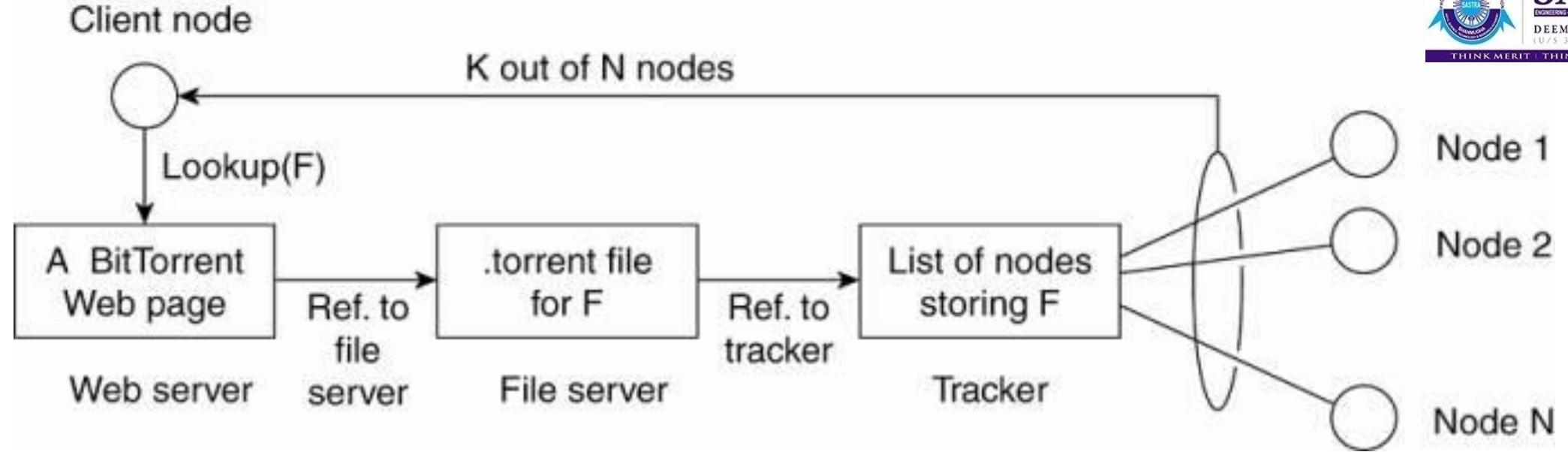
- Join system using a traditional client-server scheme.
- Once a node has joined the system - use a fully decentralized scheme for collaboration.

### Example:

BitTorrent file-sharing system

- BitTorrent is a peer-to-peer file downloading system.
- An end user downloads chunks of a file from other users until the downloaded chunks can be assembled together yielding the complete file.
- BitTorrent combines centralized with decentralized solutions.





### The principal working of BitTorrent

#### File download process:

1. Access a global directory of one of a few well-known Web sites.
  - Directory contains references to what are called .torrent files.
  - A torrent file contains the information that is needed to download a specific file.
  - It refers to a tracker - a server that keeps an accurate account of active nodes that have (chunks) of the requested file.
  - An active node is one that is currently downloading another file.
  - Many different trackers - but only a single tracker per file (or collection of files).

2. Once the nodes have been identified from where chunks can be downloaded - the downloading node becomes active.

- This node will be forced to help others by providing chunks of the file it is downloading that others do not yet have.
- Enforcement comes from a very simple rule: if node P notices that node Q is downloading more than it is uploading, P can decide to decrease the rate at which it sends data to Q.
- This scheme works well provided P has something to download from Q.
- For this reason, nodes are often supplied with references to many other nodes putting them in a better position to trade data.

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **Unit-III Distributed Computing**

### **Message-oriented communication**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

# Persistent versus Transient Communication

- **Persistent:**

- Messages are held by the middleware comm. service until they can be delivered (e.g., email)
- Sender can terminate after executing send
- Receiver will get message next time it runs

- **Transient:**

- Messages exist only while the sender and receiver are running
- Communication errors or inactive receiver cause the message to be discarded

# Berkeley Sockets

- Socket interface introduced in 1970's in Berkeley Unix
- Socket – communication end point, application can read/write data

# Berkeley Sockets (contd.)

<b>Primitive</b>	<b>Meaning</b>
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Figure 4-14. The socket primitives for TCPIIP.

# Berkeley Sockets (contd.)

- Server – Socket, bind, listen, Accept
- Client – Socket , connect
- Both – send, receive, close

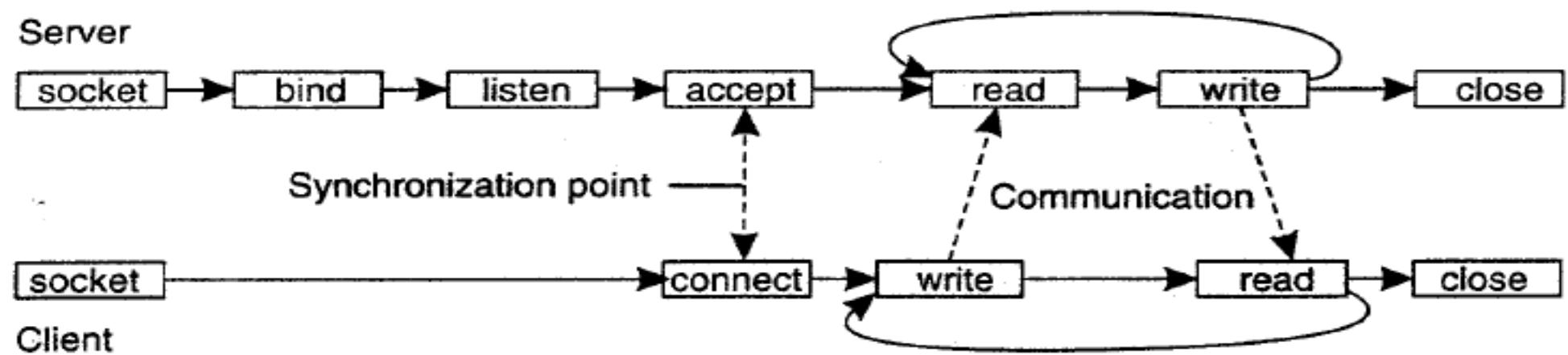


Figure 4-15. Connection-oriented communication pattern using sockets.

```

1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.bind((HOST, PORT))
4 s.listen(1)
5 (conn, addr) = s.accept() # returns new socket and addr. client
6 while True: # forever
7     data = conn.recv(1024) # receive data from client
8     if not data: break # stop if client stopped
9     conn.send(str(data)+"*") # return sent data plus an "*"
10 conn.close() # close the connection

```

**Figure 4.20:** (a) A simple socket-based client-server system: the server.

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # connect to server (block until accepted)
4 s.send('Hello, world') # send same data
5 data = s.recv(1024)    # receive the response
6 print data           # print the result
7 s.close()             # close the connection
```

Figure 4.20: (b) A simple socket-based client-server system: the client.

# Berkeley Socket - demerits

1. Support only send and receive primitives
2. Not suitable for high speed networks ( mostly used with TCP/IP )

# Advanced Transient Messaging

## ZeroMQ:

- Provides a message queue, but unlike message oriented middleware a ZeroMQ system can run **without a dedicated message broker**
- Like in the Berkeley approach, ZeroMQ also provides sockets through which all communication takes place

Three most important communication patterns supported by ZeroMQ:

✓ **Request-reply**

- Client application uses a request socket (of type REQ) to send a request message to a server and expects the latter to respond with an appropriate response
- Server is assumed to use a reply socket (of type REP)

✓ **Publish-subscribe**

- Clients subscribe to specific messages that are published by servers

✓ **Pipeline pattern**

- Process wants to push out its results, assuming that there are other processes that want to pull in those results

# Message Passing Interface (MPI)

- Designed for parallel applications using *transient* communication
- MPI is
  - Standardized and portable message-passing system designed by a group of researchers from academia and industry
  - Used in many environments, e.g., clusters
  - Platform independent
- Assume that communication takes place within a known group of processes
- Each process within group – local id
- Process – (groupid, localid) – instead of transport level address

# MPI (contd.)

Primitive	Meaning
<code>MPI_bsend</code>	Append outgoing message to a local send buffer
<code>MPI_send</code>	Send a message and wait until copied to local or remote buffer
<code>MPI_ssend</code>	Send a message and wait until receipt starts
<code>MPI_sendrecv</code>	Send a message and wait for reply
<code>MPI_isend</code>	Pass reference to outgoing message, and continue
<code>MPI_issend</code>	Pass reference to outgoing message, and wait until receipt starts
<code>MPI_recv</code>	Receive a message; block if there is none
<code>MPI_irecv</code>	Check if there is an incoming message, but do not block

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.

`MPI_Send(buffer,count,type,destination, communication)`

`MPI_Recv(buffer,count,type,source, communication,status)`

# Message oriented persistent communication

Processes communicate through message queues

- Each application – own private queue – other applications send message to that queue
- Multiple application can share single queue
- Queues are maintained by the message-queuing system
- Sender appends to queue, receiver removes from queue
- Neither the sender nor receiver needs to be on-line when the message is transmitted

# Queuing model (contd.)

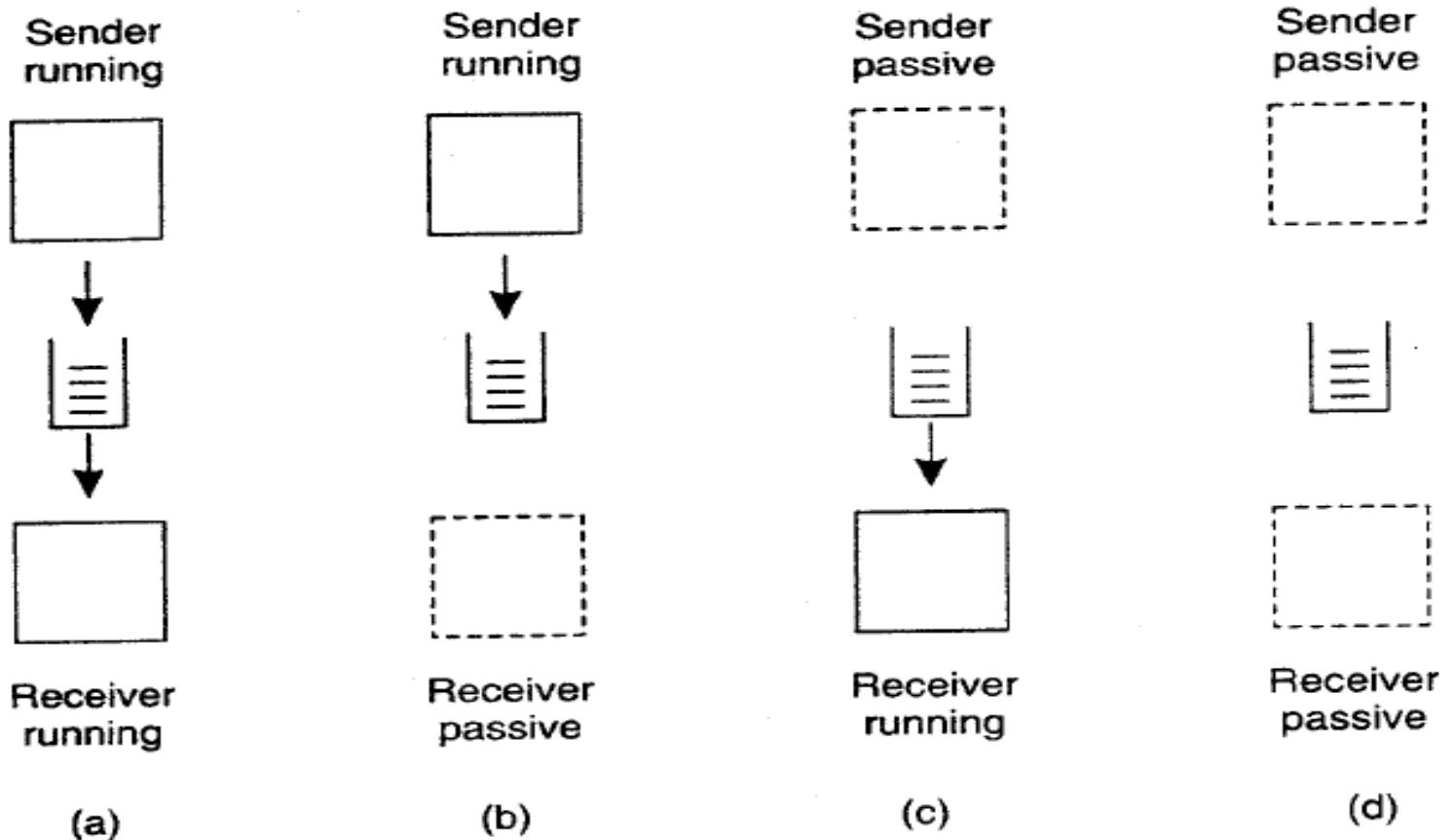


Figure 4-17. Four combinations for loosely-coupled communications using queues.

# Queuing model (contd.)

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Figure 4-18. Basic interface to a queue in a message-queuing system.

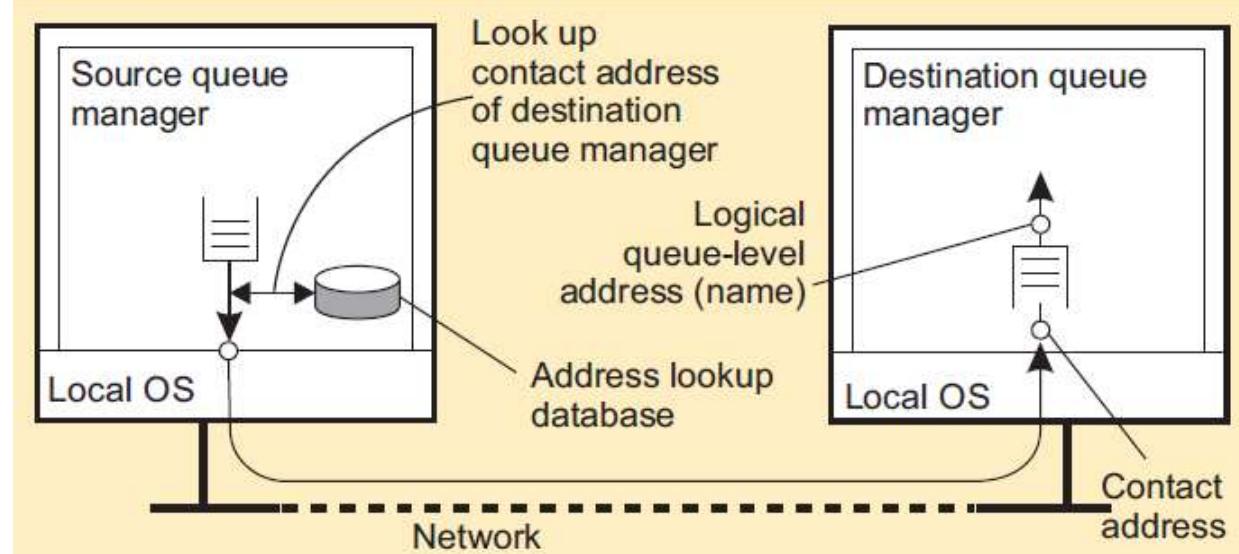
# Message-oriented middleware

- Asynchronous persistent communication through support of middleware-level queues
- Queues correspond to buffers at communication servers

Operation	Description
put	Append a message to a specified queue
get	Block until the specified queue is nonempty, and remove the first message
poll	Check a specified queue for messages, and remove the first. Never block
notify	Install a handler to be called when a message is put into the specified queue

# Queue managers

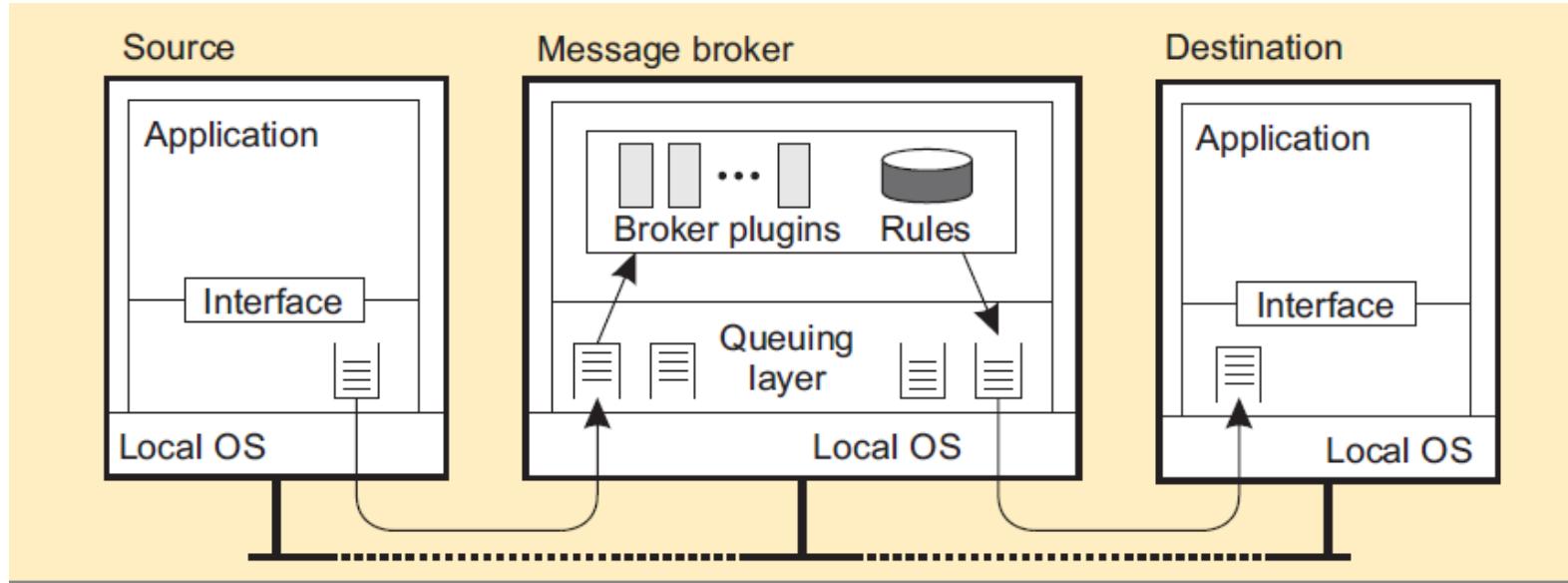
- Queues are managed by queue managers
- An application can put messages only into a local queue
- Getting a message is possible by extracting it from a local queue only ⇒ queue managers need to route messages



# Message broker

- Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)
- Broker handles application heterogeneity in an MQ system
- ✓ Transforms incoming messages to target format
- ✓ Very often acts as an application gateway
- ✓ May provide subject-based routing capabilities (i.e., publish -subscribe capabilities)

# Message broker: general architecture



# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **Unit-III Distributed Computing**

### **Multicast communication**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



# Multicast Communication

- Sending data to multiple receivers
- Major issues:-
  - setting up the communication path
  - involve huge management effort and human intervention
- Application level tree based multicasting
- Flood based multicasting
- Gossip based multicasting

# Application-level multicasting

## Essence

Organize nodes of a distributed system into an **overlay network** and use that network to disseminate data:

- Oftentimes a **tree**, leading to unique paths
- Alternatively, also **mesh networks**, requiring a form of **routing**

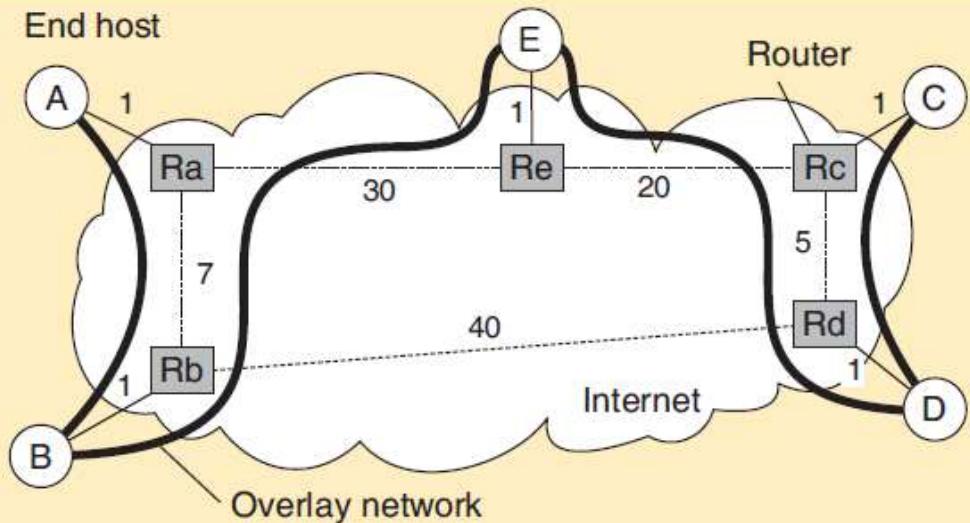
# Application-level multicasting in Chord

## Basic approach

- 1 Initiator generates a **multicast identifier  $mid$** .
- 2 Lookup  $\text{succ}(mid)$ , the node responsible for  $mid$ .
- 3 Request is routed to  $\text{succ}(mid)$ , which will become the **root**.
- 4 If  $P$  wants to join, it sends a **join** request to the root.
- 5 When request arrives at  $Q$ :
  - $Q$  has not seen a join request before  $\Rightarrow$  it becomes **forwarder**;  $P$  becomes child of  $Q$ . **Join request continues to be forwarded**.
  - $Q$  knows about tree  $\Rightarrow$   $P$  becomes child of  $Q$ . **No need to forward join request anymore**.

# ALM: Some costs

## Different metrics



- **Link stress:** How often does an ALM message cross the same physical link? **Example:** message from  $A$  to  $D$  needs to cross  $\langle Ra, Rb \rangle$  twice.
- **Stretch:** Ratio in delay between ALM-level path and network-level path. **Example:** messages  $B$  to  $C$  follow path of length 73 at ALM, but 47 at network level  $\Rightarrow$  stretch = 73/47.

# Flooding

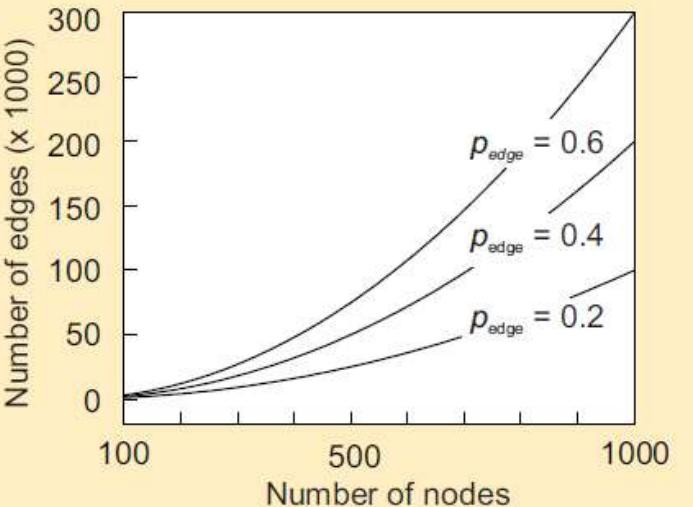
## Essence

$P$  simply sends a message  $m$  to each of its neighbors. Each neighbor will forward that message, except to  $P$ , and only if it had not seen  $m$  before.

## Performance

The more edges, the more expensive!

The size of a random overlay as function of the number of nodes



## Variation

Let  $Q$  forward a message with a certain probability  $p_{flood}$ , possibly even dependent on its own number of neighbors (i.e., **node degree**) or the degree of its neighbors.

# Gossip based multicasting

## Epidemic protocols

Assume there are no write–write conflicts

- Update operations are performed at a single server
- A replica passes updated state to only a few neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

Two forms of epidemics

- **Anti-entropy**: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards
- **Rumor spreading**: A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).

# Gossip based multicasting

## Anti-entropy

### Principle operations

- A node  $P$  selects another node  $Q$  from the system at random.
- **Pull**:  $P$  only pulls in new updates from  $Q$
- **Push**:  $P$  only pushes its own updates to  $Q$
- **Push-pull**:  $P$  and  $Q$  send updates to each other

### Observation

For push-pull it takes  $\mathcal{O}(\log(N))$  rounds to disseminate updates to all  $N$  nodes  
(**round** = when every node has taken the initiative to start an exchange).

# Gossip based multicasting

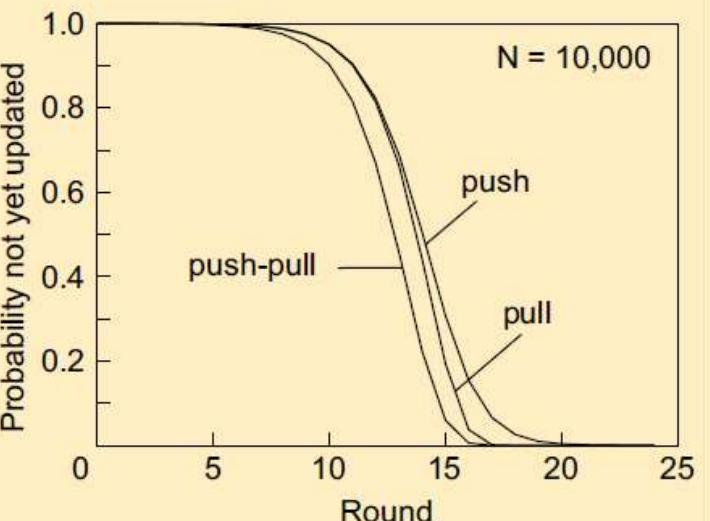
## Anti-entropy: analysis

### Basics

Consider a single source, propagating its update. Let  $p_i$  be the probability that a node has not received the update after the  $i^{th}$  round.

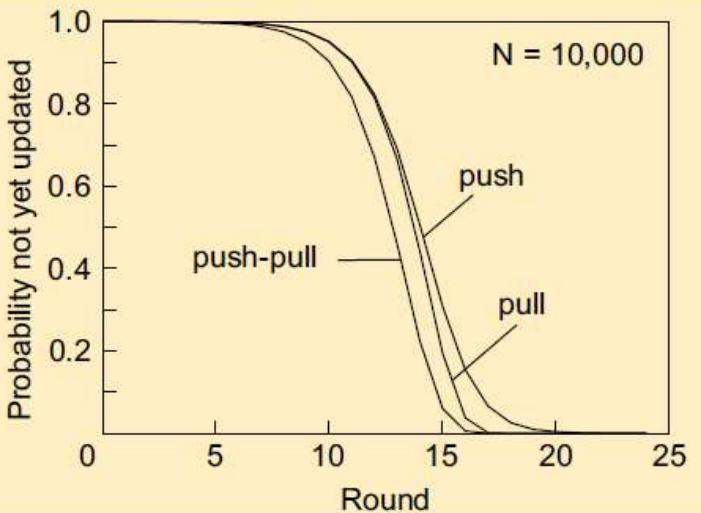
### Analysis: staying ignorant

- With **pull**,  $p_{i+1} = (p_i)^2$ : the node was not updated during the  $i^{th}$  round and should contact another ignorant node during the next round.
- With **push**,  
$$p_{i+1} = p_i \left(1 - \frac{1}{N}\right)^{N(1-p_i)} \approx p_i e^{-1}$$
 (for small  $p_i$  and large  $N$ ): the node was ignorant during the  $i^{th}$  round and no updated node chooses to contact it during the next round.
- With **push-pull**:  $(p_i)^2 \cdot (p_i e^{-1})$



# Gossip based multicasting

## Anti-entropy performance



# Gossip based multicasting

## Rumor spreading

### Basic model

A server  $S$  having an update to report, contacts other servers. If a server is contacted to which the update has already propagated,  $S$  stops contacting other servers with probability  $p_{stop}$ .

### Observation

If  $s$  is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers

$$s = e^{-(1/p_{stop}+1)(1-s)}$$

# Gossip based multicasting

## Formal analysis

### Notations

Let  $s$  denote fraction of nodes that have not yet been updated (i.e., **susceptible**);  
 $i$  the fraction of updated (**infected**) and active nodes; and  $r$  the fraction of  
updated nodes that gave up (**removed**).

### From theory of epidemics

$$(1) \ ds/dt = -s \cdot i$$

$$(2) \ di/dt = s \cdot i - p_{stop} \cdot (1 - s) \cdot i$$

$$\Rightarrow di/ds = -(1 + p_{stop}) + \frac{p_{stop}}{s}$$

$$\Rightarrow i(s) = -(1 + p_{stop}) \cdot s + p_{stop} \cdot \ln(s) + C$$

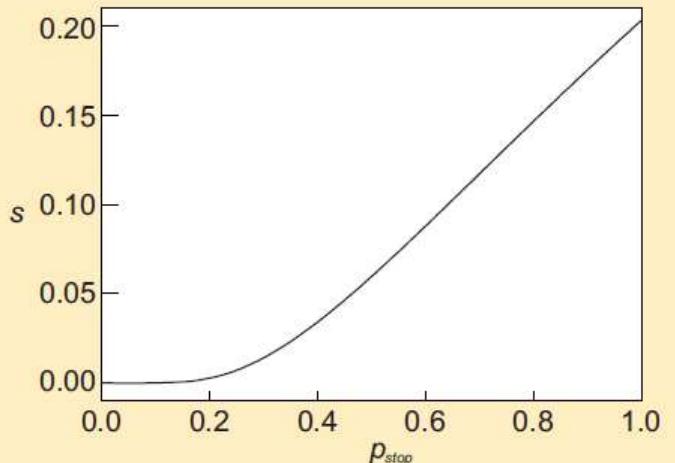
### Wrapup

$i(1) = 0 \Rightarrow C = 1 + p_{stop} \Rightarrow i(s) = (1 + p_{stop}) \cdot (1 - s) + p_{stop} \cdot \ln(s)$ . We are looking for the case  $i(s) = 0$ , which leads to  $s = e^{-(1/p_{stop}+1)(1-s)}$

# Gossip based multicasting

## Rumor spreading

### The effect of stopping

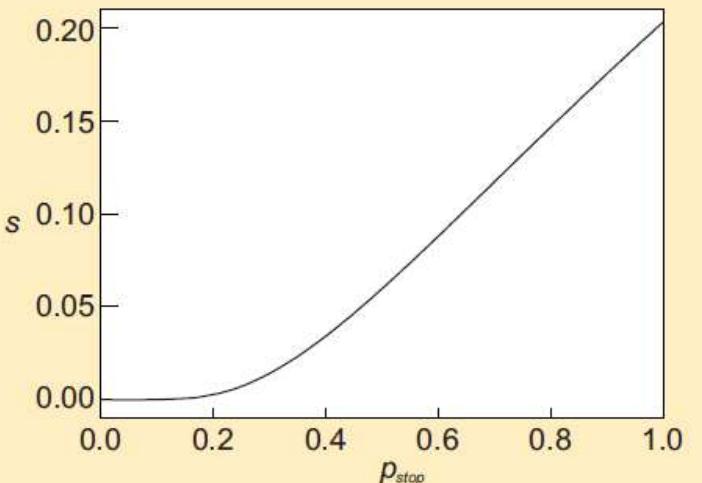


Consider 10,000 nodes		
$1/p_{stop}$	$s$	$N_s$
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

# Gossip based multicasting

## Rumor spreading

### The effect of stopping



Consider 10,000 nodes		
$1/p_{stop}$	$s$	$N_s$
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

### Note

If we really have to ensure that all servers are eventually updated, rumor spreading alone is not enough

# Gossip based multicasting

## Deleting values

### Fundamental problem

We cannot remove an old value from a server and expect the removal to propagate. Instead, mere removal will be undone in due time using epidemic algorithms

### Solution

Removal has to be registered as a special update by inserting a [death certificate](#)

#### When to remove a death certificate (it is not allowed to stay for ever)

- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)
- Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers)

#### Note

It is necessary that a removal actually reaches all servers.

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **Unit-IV Distributed Computing**

### **Clock Synchronization**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

# Physical clocks

## Physical clocks

### Problem

Sometimes we simply need the exact time, not just an ordering.

### Solution: Universal Coordinated Time (UTC)

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

### Note

UTC is **broadcast** through short-wave radio and satellite. Satellites can give an accuracy of about  $\pm 0.5$  ms.

# Clock Synchronization

## Precision

The goal is to keep the deviation between two clocks on any two machines within a specified bound, known as the precision  $\pi$ :

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

with  $C_p(t)$  the computed clock time of machine  $p$  at UTC time  $t$ .

## Accuracy

In the case of accuracy, we aim to keep the clock bound to a value  $\alpha$ :

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

## Synchronization

- Internal synchronization: keep clocks precise
- External synchronization: keep clocks accurate

# Clock Synchronization

## Clock drift

### Clock specifications

- A clock comes specified with its **maximum clock drift rate**  $\rho$ .
- $F(t)$  denotes oscillator frequency of the hardware clock at time  $t$
- $F$  is the clock's ideal (constant) frequency  $\Rightarrow$  living up to specifications:

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

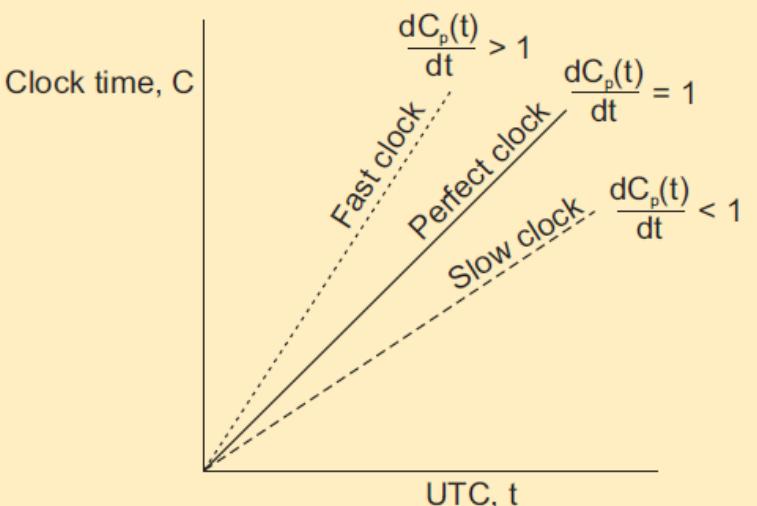
### Observation

By using hardware interrupts we couple a software clock to the hardware clock, and thus also its clock drift rate:

$$C_p(t) = \frac{1}{F} \int_0^t F(t) dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

$$\Rightarrow \forall t : 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

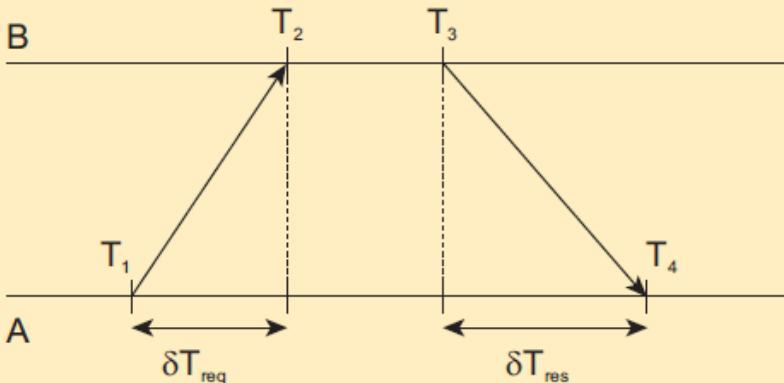
### Fast, perfect, slow clocks



# Clock Synchronization

## Detecting and adjusting incorrect times

Getting the current time from a time server



Computing the relative offset  $\theta$  and delay  $\delta$

**Assumption:**  $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

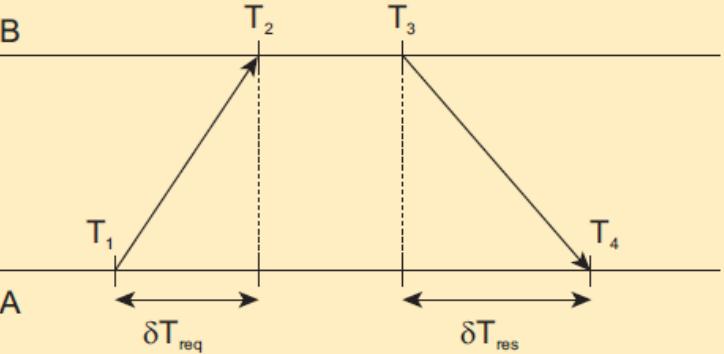
$$\theta = T_3 + ((T_2 - T_1) + (T_4 - T_3)) / 2 - T_4 = ((T_2 - T_1) + (T_3 - T_4)) / 2$$

$$\delta = ((T_4 - T_1) - (T_3 - T_2)) / 2$$

# Clock Synchronization

Detecting and adjusting incorrect times

Getting the current time from a time server



Computing the relative offset  $\theta$  and delay  $\delta$

**Assumption:**  $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

$$\theta = T_3 + ((T_2 - T_1) + (T_4 - T_3)) / 2 - T_4 = ((T_2 - T_1) + (T_3 - T_4)) / 2$$

$$\delta = ((T_4 - T_1) - (T_3 - T_2)) / 2$$

Network Time Protocol

Collect eight  $(\theta, \delta)$  pairs and choose  $\theta$  for which associated delay  $\delta$  was minimal.

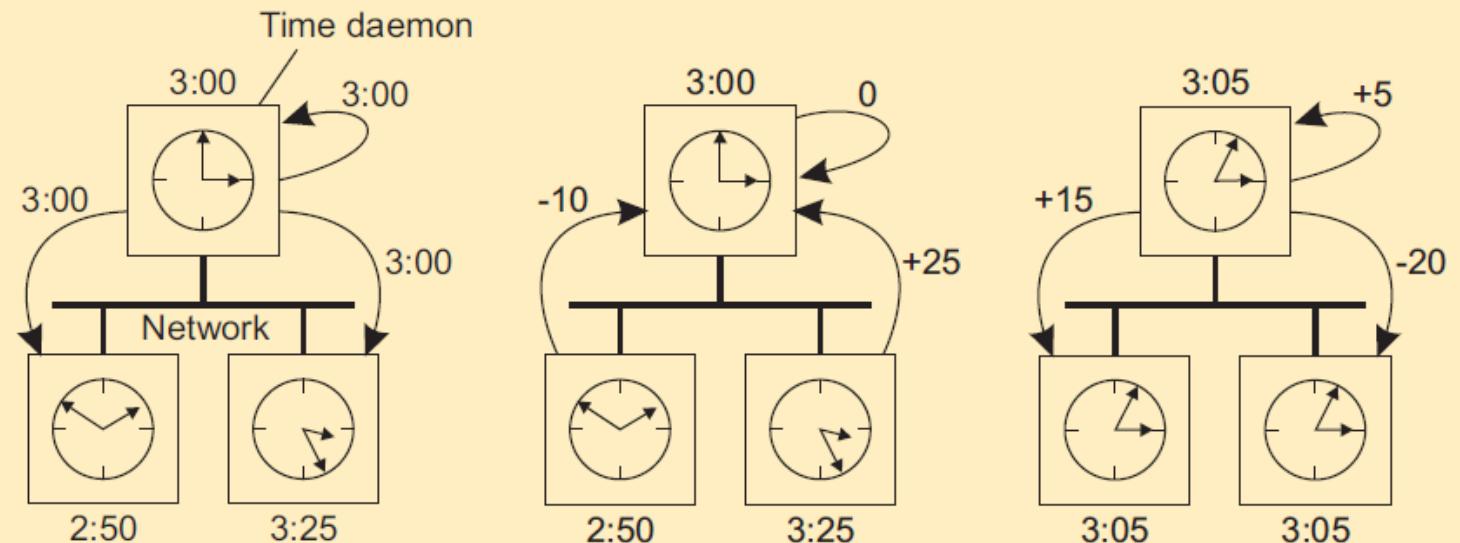
# Clock Synchronization

## Keeping time without UTC

### Principle

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time **relative to its present time**.

### Using a time server



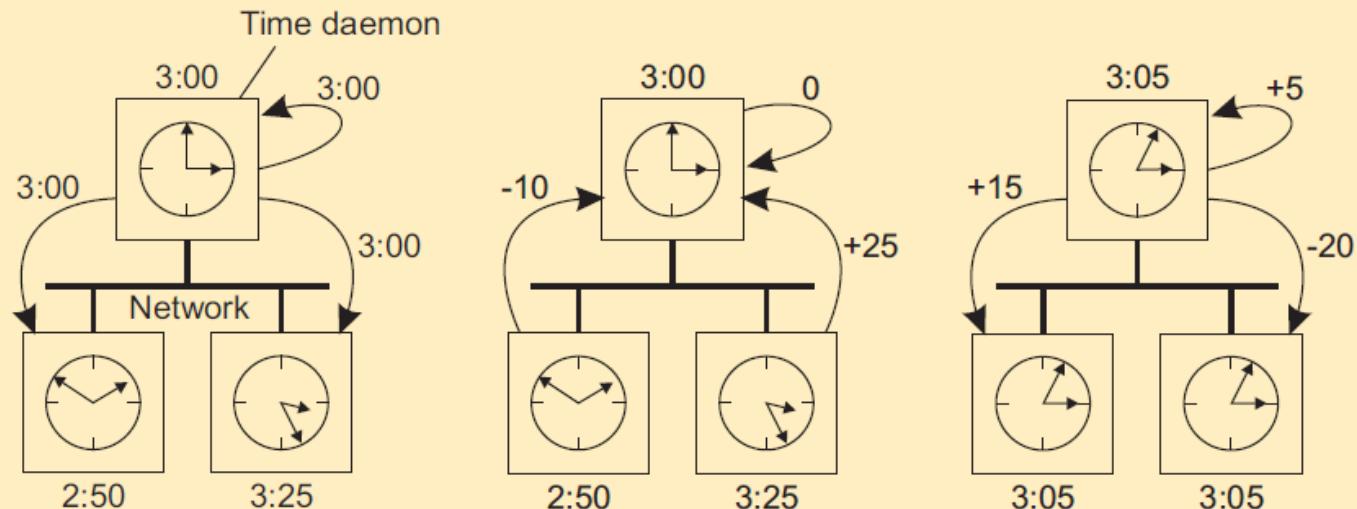
# Clock Synchronization

## Keeping time without UTC

### Principle

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time **relative to its present time**.

### Using a time server



### Fundamental

You'll have to take into account that setting the time back is **never** allowed ⇒ smooth adjustments (i.e., run faster or slower).

# Clock Synchronization

## Reference broadcast synchronization

### Essence

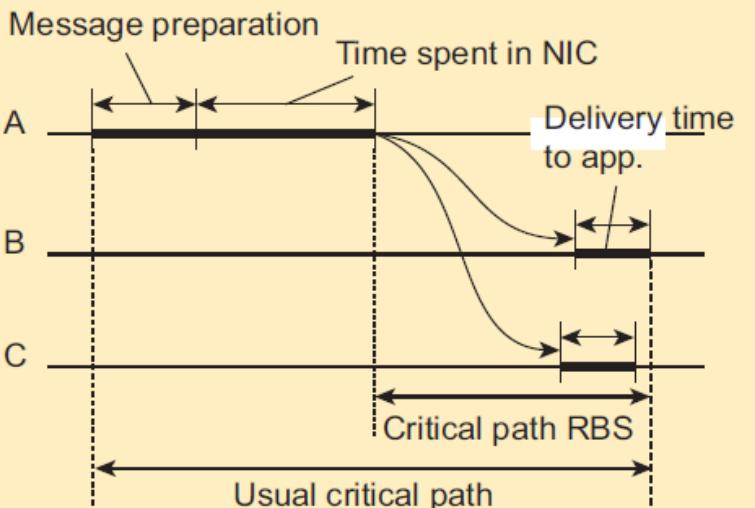
- A node broadcasts a reference message  $m \Rightarrow$  each receiving node  $p$  records the time  $T_{p,m}$  that it received  $m$ .
- Note:  $T_{p,m}$  is read from  $p$ 's local clock.

Problem: averaging will not capture drift  $\Rightarrow$  use linear regression

NO:  $\text{Offset}[p, q](t) = \frac{\sum_{k=1}^M (T_{p,k} - T_{q,k})}{M}$

YES:  $\text{Offset}[p, q](t) = \alpha t + \beta$

### RBS minimizes critical path



# Logical clocks

## The Happened-before relationship

### Issue

What usually matters is not that all processes agree on exactly what time it is, but that they agree on the **order in which events occur**. Requires a notion of ordering.

### The **happened-before** relation

- If  $a$  and  $b$  are two events in the same process, and  $a$  comes before  $b$ , then  $a \rightarrow b$ .
- If  $a$  is the sending of a message, and  $b$  is the receipt of that message, then  $a \rightarrow b$
- If  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$

### Note

This introduces a **partial ordering of events** in a system with concurrently operating processes.

# Logical clocks

## Problem

How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Attach a timestamp  $C(e)$  to each event  $e$ , satisfying the following properties:

- P1 If  $a$  and  $b$  are two events in the same process, and  $a \rightarrow b$ , then we demand that  $C(a) < C(b)$ .
- P2 If  $a$  corresponds to sending a message  $m$ , and  $b$  to the receipt of that message, then also  $C(a) < C(b)$ .

## Problem

How to attach a timestamp to an event when there's no global clock  $\Rightarrow$  maintain a **consistent** set of logical clocks, one per process.

# Logical clocks

## Logical clocks: solution

Each process  $P_i$  maintains a local counter  $C_i$  and adjusts this counter

- ① For each new event that takes place within  $P_i$ ,  $C_i$  is incremented by 1.
- ② Each time a message  $m$  is sent by process  $P_i$ , the message receives a timestamp  $ts(m) = C_i$ .
- ③ Whenever a message  $m$  is received by a process  $P_j$ ,  $P_j$  adjusts its local counter  $C_j$  to  $\max\{C_j, ts(m)\}$ ; then executes step 1 before passing  $m$  to the application.

### Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

# Logical clocks

## Logical clocks: example

Consider three processes with **event counters** operating at different rates

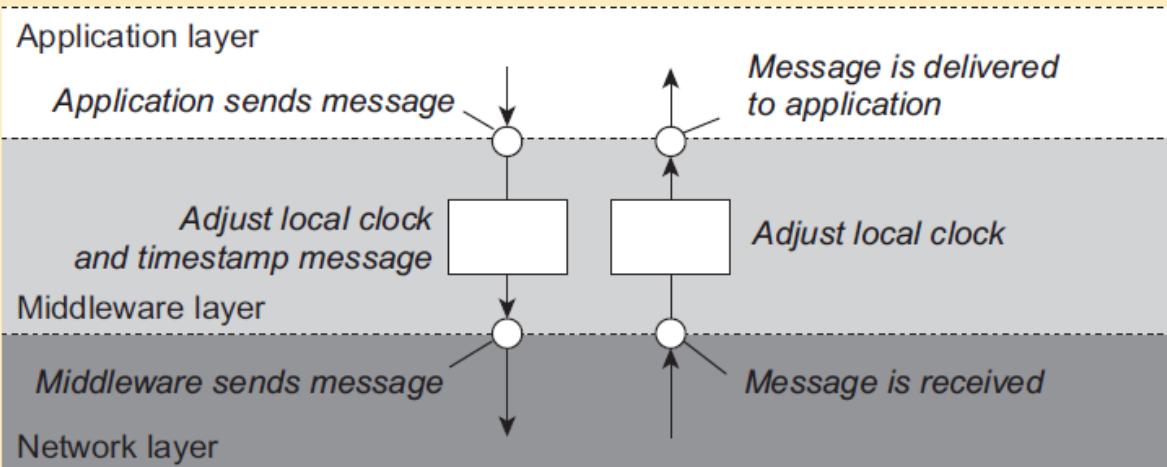
P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

# Logical clocks

## Logical clocks: where implemented

Adjustments implemented in middleware

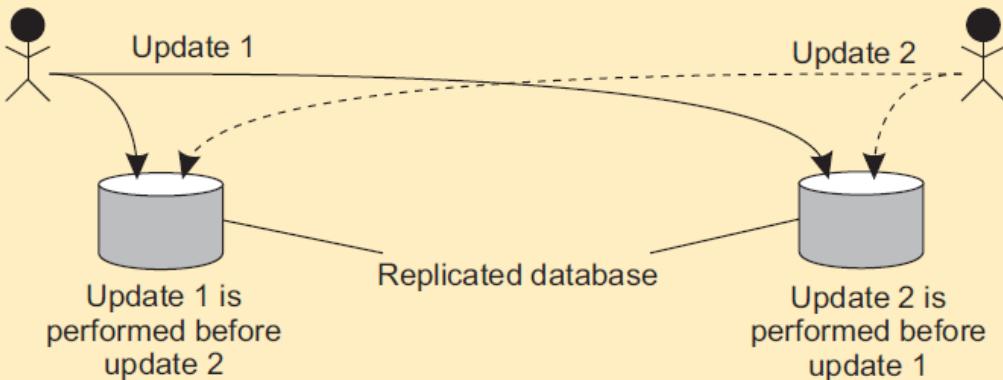


# Logical clocks

## Example: Total-ordered multicast

Concurrent updates on a replicated database are seen in the same order everywhere

- $P_1$  adds \$100 to an account (initial value: \$1000)
- $P_2$  increments account by 1%
- There are two replicas



## Result

In absence of proper synchronization:  
replica #1  $\leftarrow \$1111$ , while replica #2  $\leftarrow \$1110$ .

# Logical clocks

## Example: Total-ordered multicast

### Solution

- Process  $P_i$  sends **timestamped message**  $m_i$  to all others. The message itself is put in a local queue  $queue_i$ .
- Any incoming message at  $P_j$  is queued in  $queue_j$ , **according to its timestamp**, and **acknowledged** to every other process.

$P_j$  passes a message  $m_i$  to its application if:

- (1)  $m_i$  is at the head of  $queue_j$
- (2) for each process  $P_k$ , there is a message  $m_k$  in  $queue_j$  with a larger timestamp.

### Note

We are assuming that communication is **reliable** and **FIFO ordered**.

# Logical clocks

## Lamport's clocks for mutual exclusion

### Analogy with total-ordered multicast

- With total-ordered multicast, all processes build identical queues, delivering messages in the same order
- Mutual exclusion is about agreeing in which order processes are allowed to enter a critical section

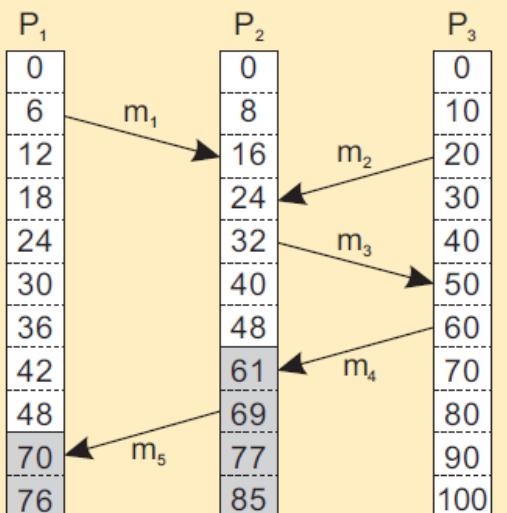
# Logical clocks

## Vector clocks

### Observation

Lamport's clocks do not guarantee that if  $C(a) < C(b)$  that  $a$  causally preceded  $b$ .

### Concurrent message transmission using logical clocks



### Observation

Event  $a$ :  $m_1$  is received at  $T = 16$ ;  
 Event  $b$ :  $m_2$  is sent at  $T = 20$ .

### Note

We **cannot** conclude that  $a$  causally precedes  $b$ .

# Logical clocks

## Causal dependency

### Definition

We say that  $b$  may causally depend on  $a$  if  $ts(a) < ts(b)$ , with:

- for all  $k$ ,  $ts(a)[k] \leq ts(b)[k]$  and
- there exists at least one index  $k'$  for which  $ts(a)[k'] < ts(b)[k']$

### Precedence vs. dependency

- We say that  $a$  causally precedes  $b$ .
- $b$  **may** causally depend on  $a$ , as there may be information from  $a$  that is propagated into  $b$ .

# Logical clocks

## Capturing causality

Solution: each  $P_i$  maintains a vector  $VC_i$

- $VC_i[i]$  is the local logical clock at process  $P_i$ .
- If  $VC_i[j] = k$  then  $P_i$  knows that  $k$  events have occurred at  $P_j$ .

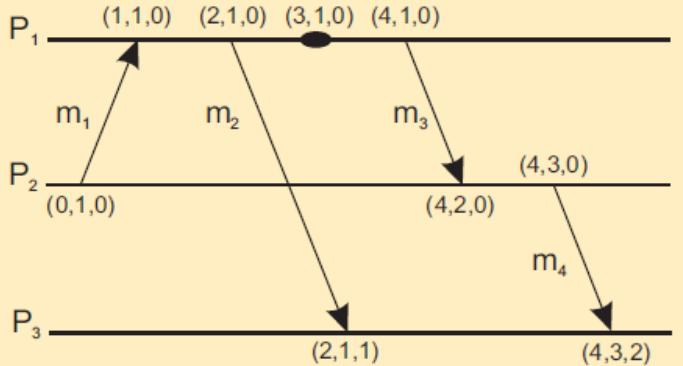
## Maintaining vector clocks

- ① Before executing an event  $P_i$  executes  $VC_i[i] \leftarrow VC_i[i] + 1$ .
- ② When process  $P_i$  sends a message  $m$  to  $P_j$ , it sets  $m$ 's (vector) timestamp  $ts(m)$  equal to  $VC_i$  after having executed step 1.
- ③ Upon the receipt of a message  $m$ , process  $P_j$  sets  $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$  for each  $k$ , after which it executes step 1 and then delivers the message to the application.

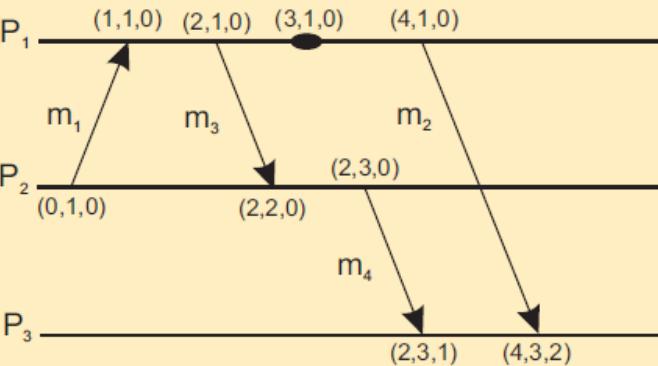
# Logical clocks

## Vector clocks: Example

Capturing potential causality when exchanging messages



(a)



(b)

## Analysis

Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Conclusion
(a)	(2, 1, 0)	(4, 3, 0)	Yes	No	$m_2$ may causally precede $m_4$
(b)	(4, 1, 0)	(2, 3, 0)	No	No	$m_2$ and $m_4$ may conflict

# Logical clocks

## Causally ordered multicasting

### Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

### Adjustment

$P_i$  increments  $VC_i[i]$  only when sending a message, and  $P_j$  “adjusts”  $VC_j$  when receiving a message (i.e., effectively does not change  $VC_j[j]$ ).

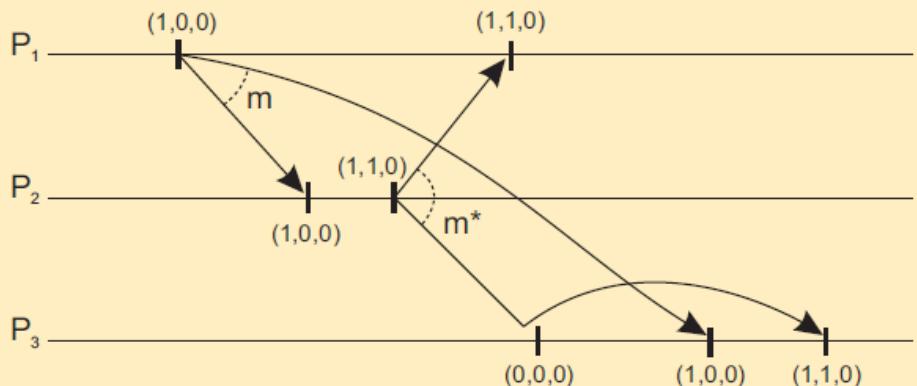
$P_j$  postpones delivery of  $m$  until:

- 1  $ts(m)[i] = VC_j[i] + 1$
- 2  $ts(m)[k] \leq VC_j[k]$  for all  $k \neq i$

# Logical clocks

## Causally ordered multicasting

Enforcing causal communication



# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **Unit-IV Distributed Computing**

### **Mutual Exclusion**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



PresenterMedia

## UNIT - IV

**Coordination:** Clock Synchronization - Logical clocks - **Mutual Exclusion:** Centralized algorithm - Distributed Algorithm - Token-ring algorithm - Decentralized Algorithm - **Election Algorithms:** Bully algorithm - Ring algorithm - Elections in wireless environment and large scale systems

**Fault Tolerance:** Introduction to fault tolerance - Concepts - Failure models - Failure masking by redundancy - **Reliable client server communication:** Point to point communication - RPC semantics in the presence of failures - **Reliable Group Communication:** Atomic multicast - Distributed commit - Recovery

# MUTUAL EXCLUSION

- When a process is accessing a shared variable, the process is said to be in Critical Section.
- No two process can be in the same critical section at the same time.

# Introduction

- To grant mutual exclusive access to resources by processes
- To prevent inconsistency , corruption of data due to simultaneous access

## Algorithm Categories :

1. Token-based
2. Permission-based

# Token-based

- Special message called token passed between processes
- Token holding process allowed to access shared resource
- After usage token given to next process
- If process not interested in resource access, pass it to next process

## Properties:

1. Avoid starvation
2. Avoid deadlock

*process should be organized in a manner that all of them  
should get the token*

# Permission-based

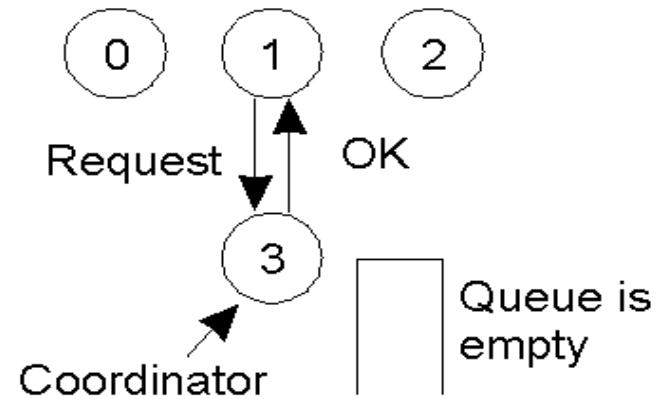
- Process wants to access resource requires permission of other processes

# Algorithms

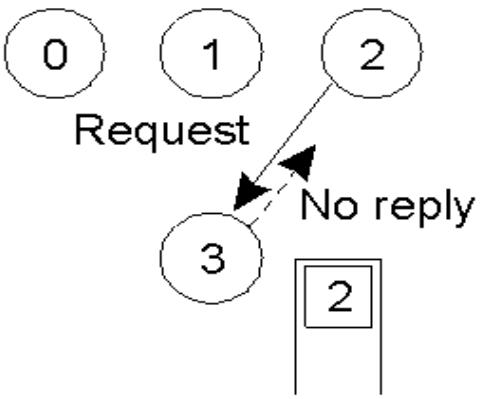
1. Centralized
2. Distributed
3. Token ring
4. Decentralized

# Centralized algorithm

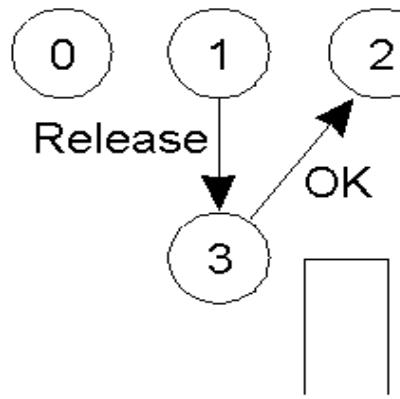
- Permission-based
- Process wants to enter critical section, send “request” msg to coordinator
- Coordinator check critical section
  - Already used by another process , block requesting process or send “permission denied” msg
  - Not used by another process, sent “grant” msg



(a)



(b)



(c)

- a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
- b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
- c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

## **Advantage:**

- Ensures presence of only one process in critical section
- Request granted in order in which they are received
- Easy to implement – only 3 messages  
(request, grant, release)

## **Disadvantage:**

- Coordinator crash – entire system down

# Distributed algorithm

- **Permission based**
- Process wants to enter CS send message to all other processes

*Msg(name of CS, process id, current time)*

- Receiving process
  - Not using CS, don't want CS , send “OK”
  - Already in CS, don't reply, queues request
  - Wants to enter CS, compare with its timestamp

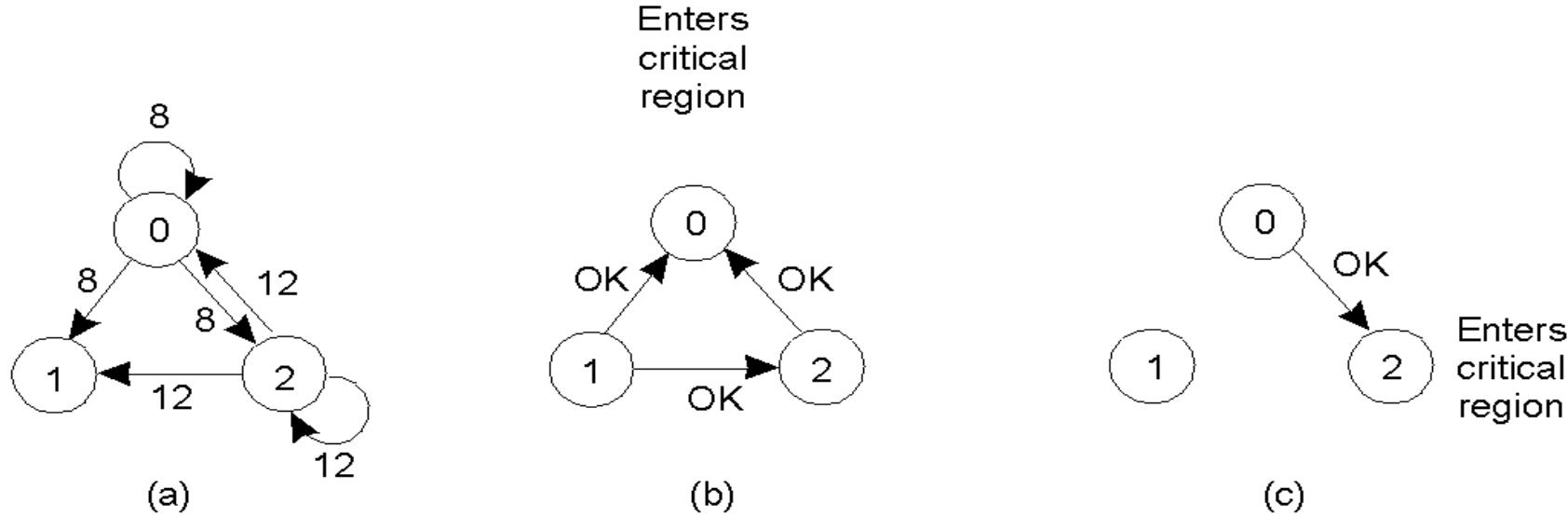
Timestamp of receiver < sender

- Queues request, don't reply

Timestamp of receiver > sender

- Send “OK” msg

# Distributed Algorithm



- a) Two processes want to enter the same critical region at the same moment.
- b) Process 0 has the lowest timestamp, so it wins.
- c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

## **Advantage :**

- No single point of failure

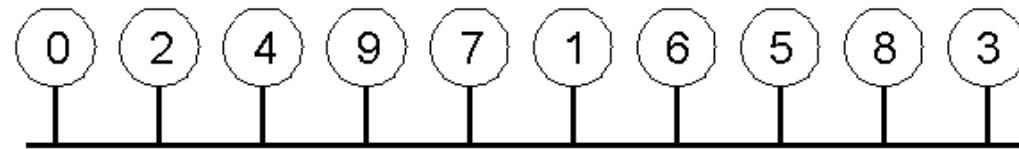
## **Disadvantage:**

- Need permission from all process – if anyone crash, no response
- Each process must maintain group members list

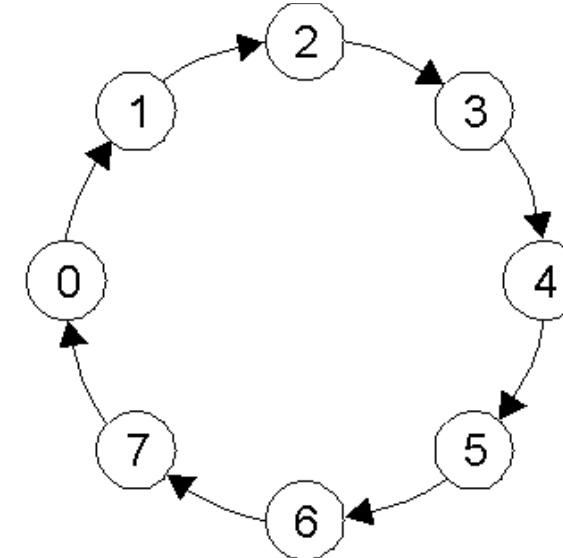
# Token ring algorithm

- **Token based**
- Processes – connected as logical ring
- Each process knows who is next?
- Token circulates around ring
- Initially token given to process0 and passed to next process
- Token holder is allowed to enter CS
- If any process don't want to enter CS, just pass token to next process

# Token Ring Algorithm



(a)



(b)

- a) An unordered group of processes on a network.
- b) A logical ring constructed in software.

## Disadvantage:

1. Token lost
2. Process crash

# Decentralized Algorithm

- Permission based
- More fault-tolerant than the centralized approach.
- Based on the Distributed Hash Table (DHT) system structure
  - Object names are hashed to find the node where they are stored
- $n$  replicas of each object are placed on  $n$  successive nodes
  - Hash object name to get addresses
- Every replica has a coordinator for controlling the access by concurrent processes

# Decentralized Algorithm

- Coordinators respond to requests at once: Yes or No
- For a process to use the resource it must receive permission from  $m > n/2$  coordinators.
  - If the requester gets fewer than  $m$  votes it will wait for a random time and then ask again.
- If a request is denied, or when the CS is completed, notify the coordinators who have sent OK messages, so they can respond again to another request.

# Analysis

- More robust than the central coordinator approach.
- If one coordinator goes down others are available.
  - If a coordinator fails and resets then it will not remember having granted access to one requestor, and may then give access to another.
  - According to the authors, it is highly unlikely that this will lead to a violation of mutual exclusion.

# Analysis

- If a resource is in high demand, multiple requests will be generated by different processes.
- High level of contention
- Processes may wait a long time to get permission
- Resource usage drops.

# Election algorithms

# Need for leader

- Many distributed algorithms require one process to act as coordinator
- If all processes are same with no distinguishing characteristics, highest numbered process will be selected as leader
- *Goal of election algorithm:*  
*find leader with agreement from all processes*

Assumption:

- Every process know process number of other process
- Do not know which one are currently up/down

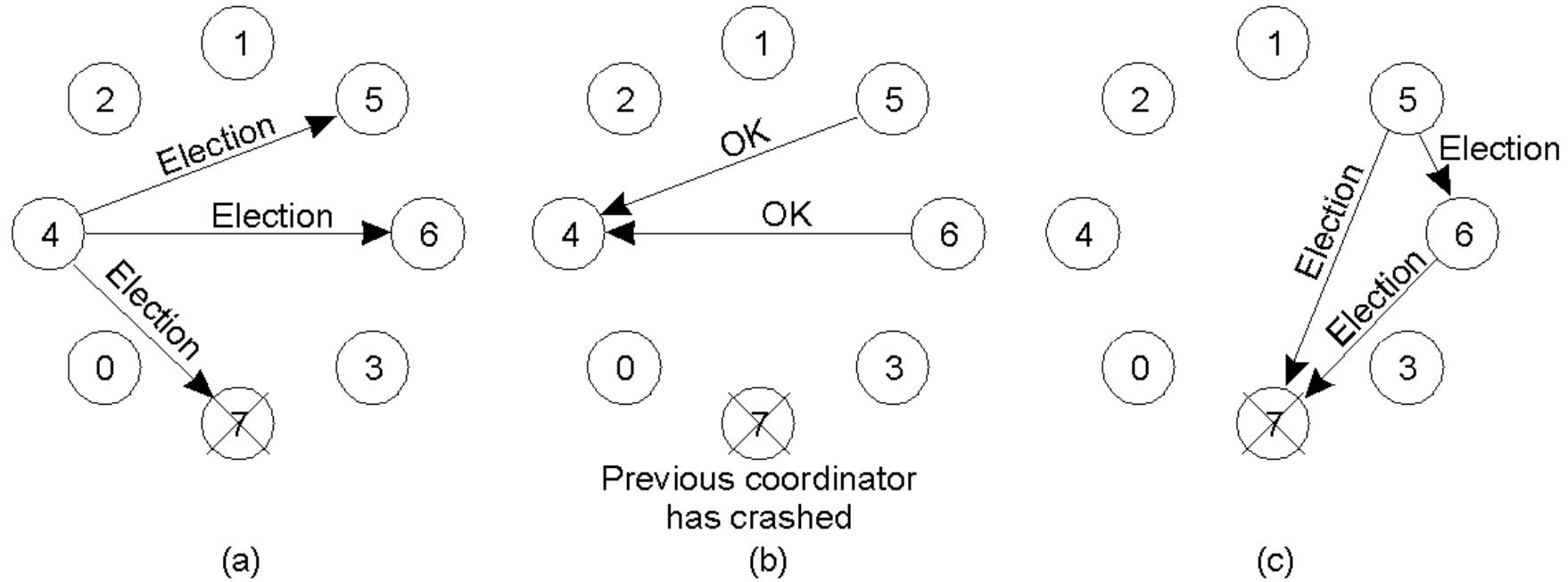
# Algorithms

- Traditional election algorithm
  - 1. Bully algorithm
  - 2. Ring algorithm
- Election in wireless environment

# Bully algorithm

- Highest numbered process serves as a coordinator
- Process (p) initiate election – if notice no longer responding coordinator
- Send election message to all process, with highest number
- If no one responds, (p) wins elections and become coordinator
- If higher-one answer, (p) silent and higher one conduct election
- If previous coordinator up – conduct election

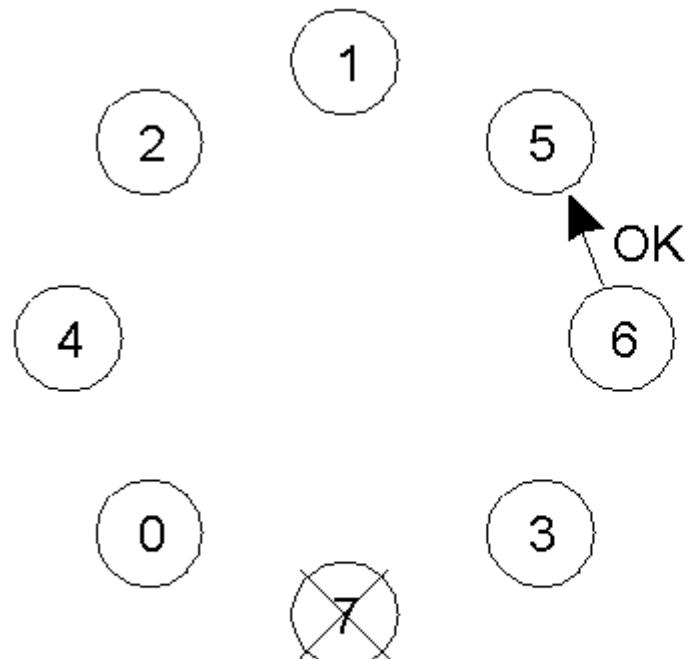
# The Bully Algorithm (1)



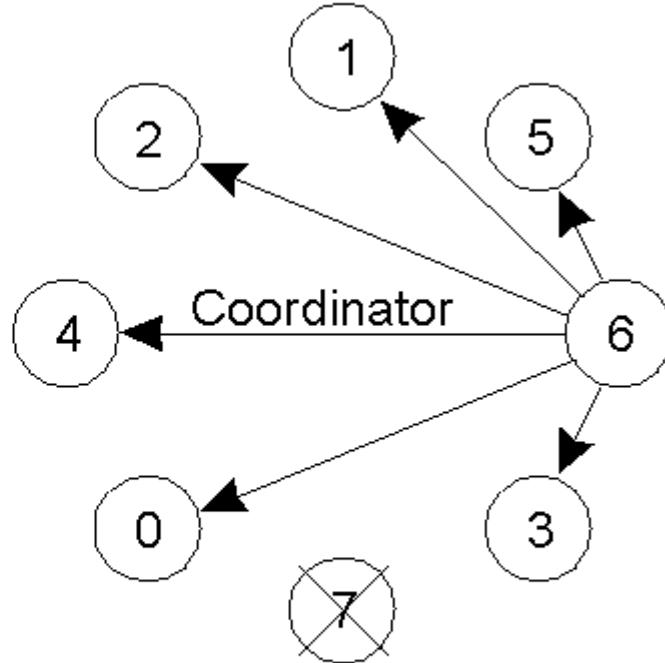
## The bully election algorithm

- **Process 4 holds an election**
- **Process 5 and 6 respond, telling 4 to stop**
- **Now 5 and 6 each hold an election**

# The Bully Algorithm (2)



(d)



(e)

- d) Process 6 tells 5 to stop
- e) Process 6 wins and tells everyone

# Issues

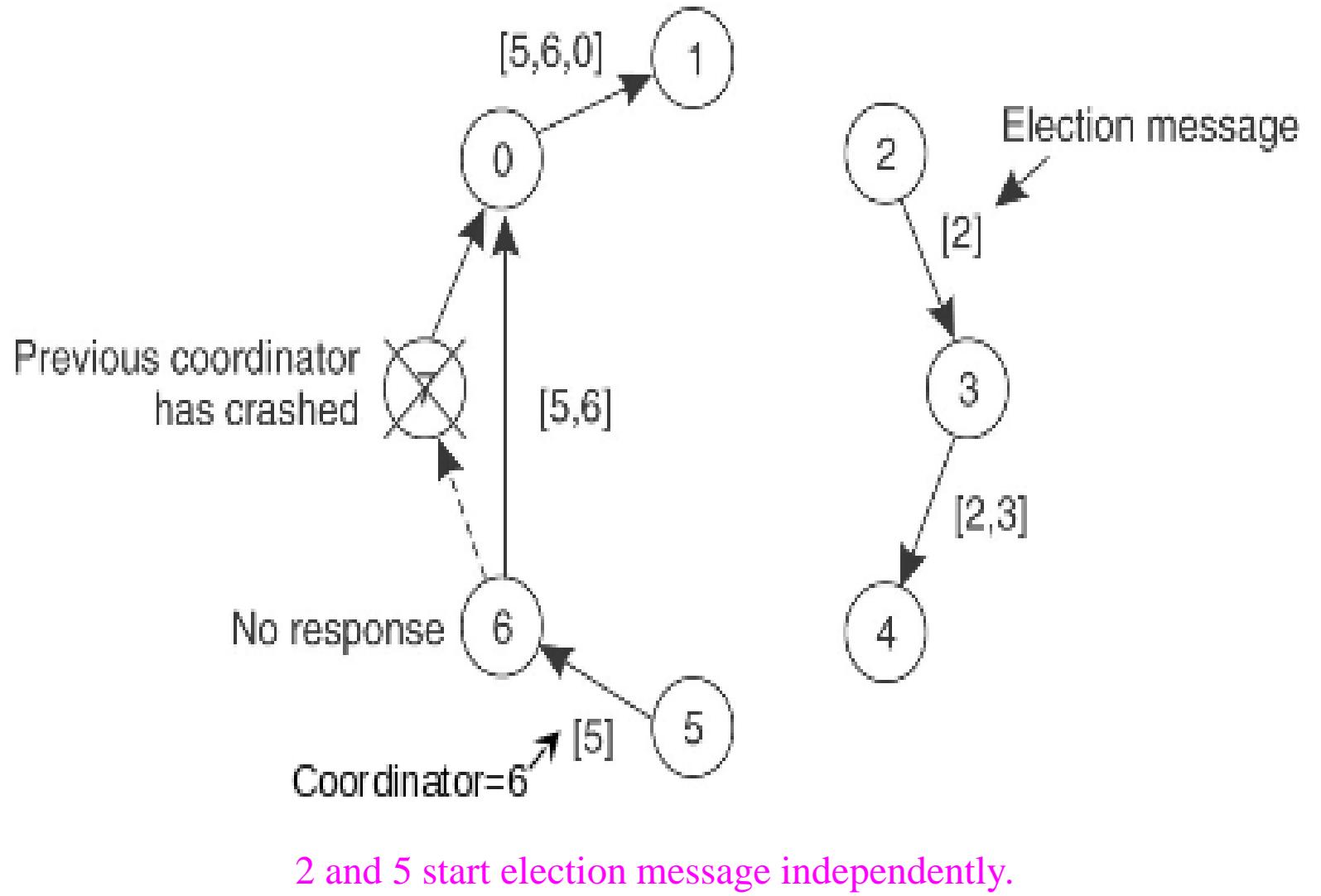
Suppose crashed nodes comes back on line:

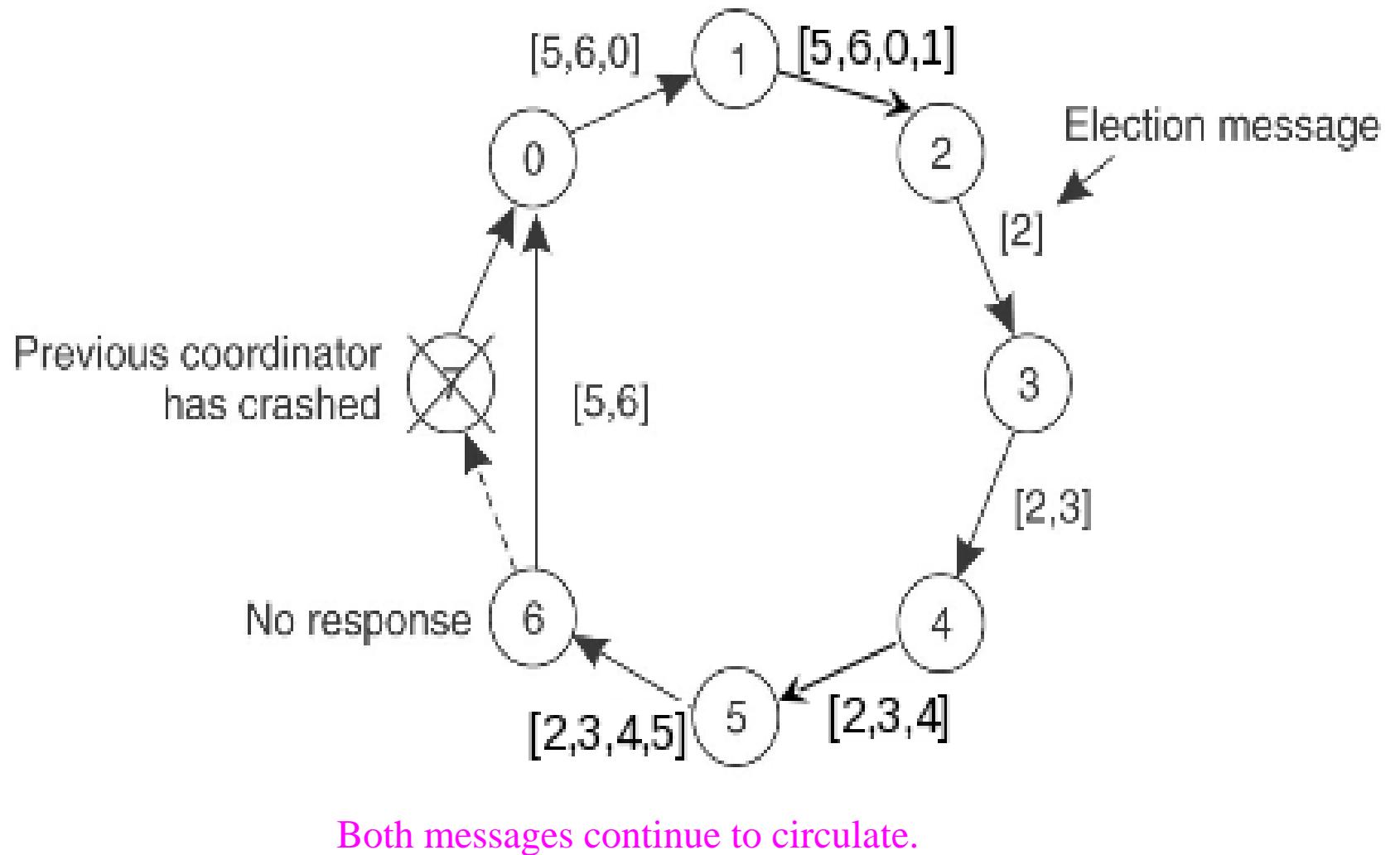
- Sends a new election message to higher numbered processes
- Repeat until only one process left standing
- Announces victory by sending message saying that it is coordinator (if not already coordinator)
- Existing (lower numbered) coordinator yields

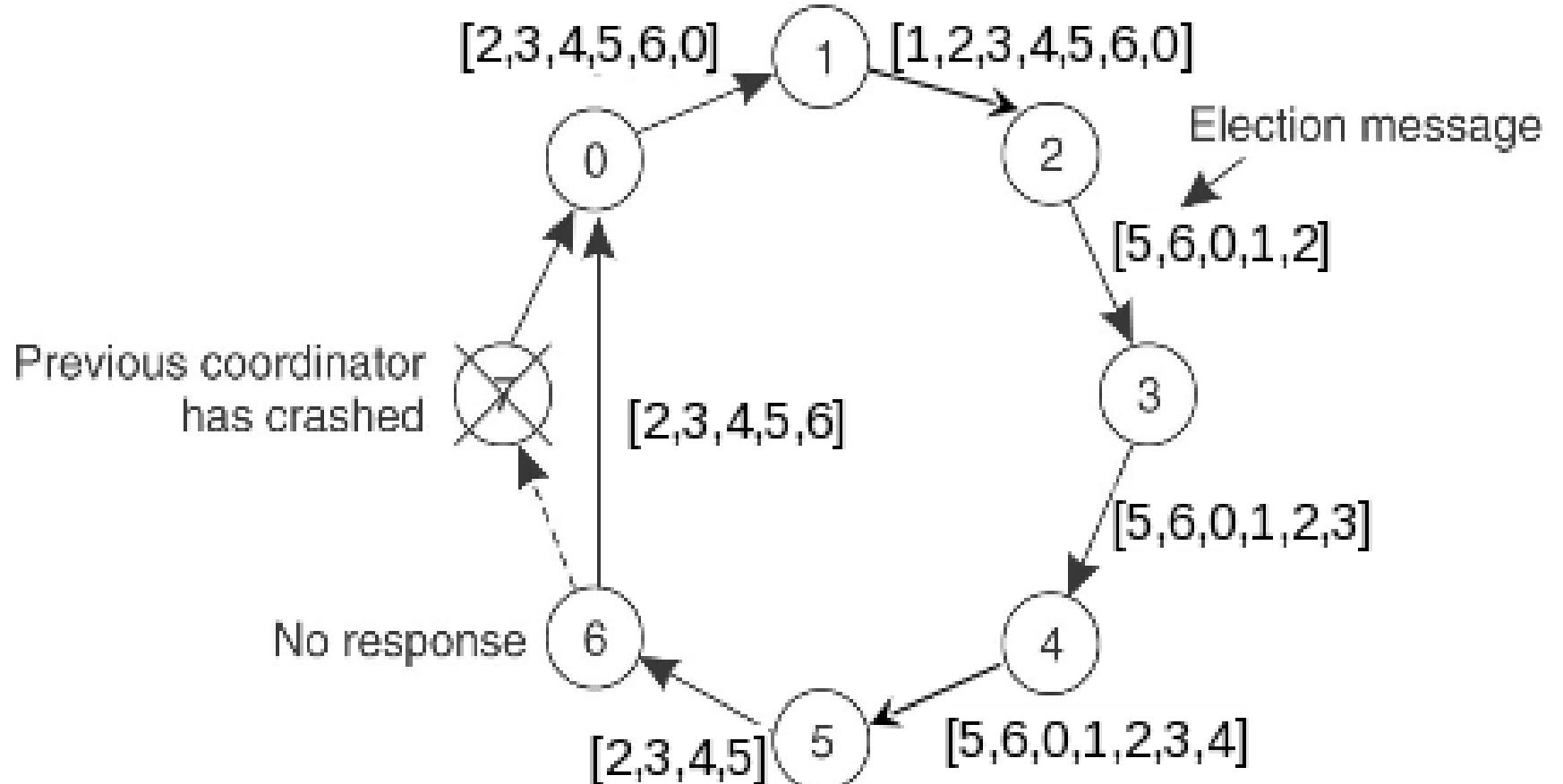
Hence the term 'bully'

# Ring algorithm

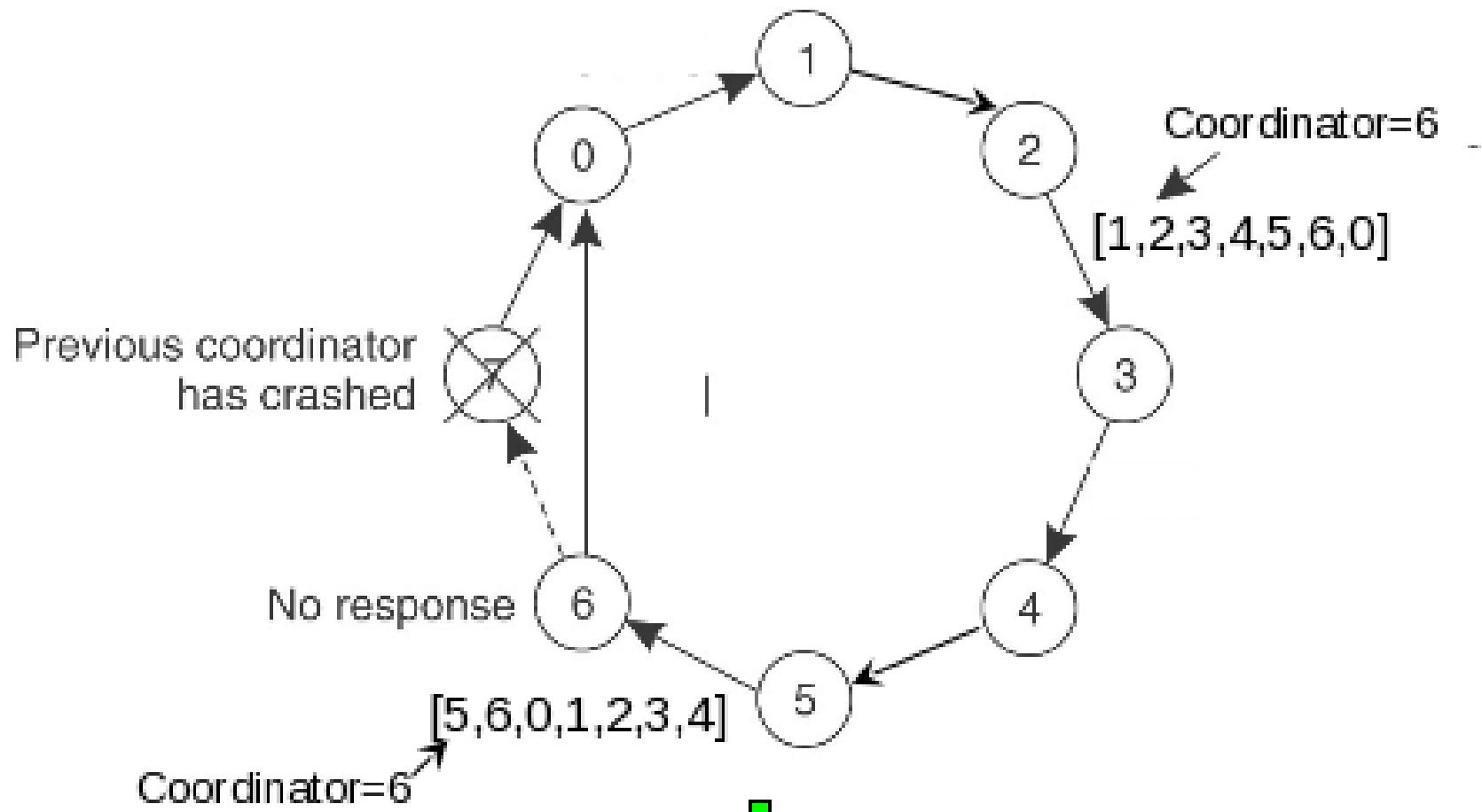
- Process physically / logically ordered
- Each process know its successor
- Process initiate election – if notice, no longer responding coordinator
- Send election message to successor, with its process number (in a list)
- If successor down, message goes to next member in ring, each process add its number to list
- Algorithm repeated until message reaches the initiated process
- Highest process number in the list becomes coordinator
- Selected coordinator will be circulated to each process







Eventually, both messages will go all the way around



2 and 5 will convert Election messages to COORDINATOR messages.  
 All processes recognize highest numbered process as new coordinator.

# Issues

1. Does it matter if two processes initiate an election?

*Just circulate extra messages and increase network traffic ( not harmful )*

2. What happens if a process crashes during the election?

# Election in Wireless networks

- Following assumptions not applicable
  - Message passing reliable
  - Topology of network doesn't change

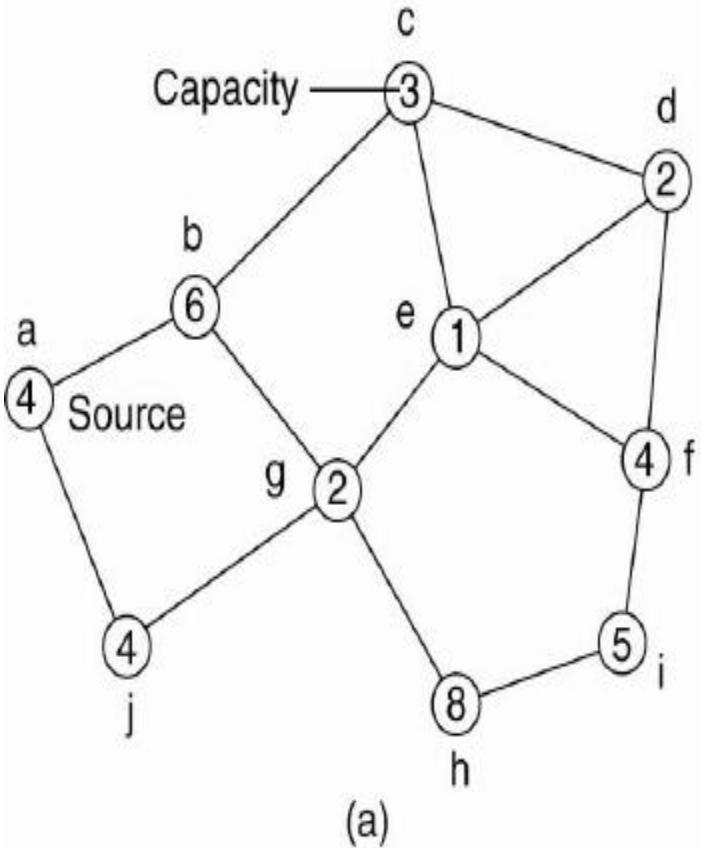
Algorithm:

*Goal:*

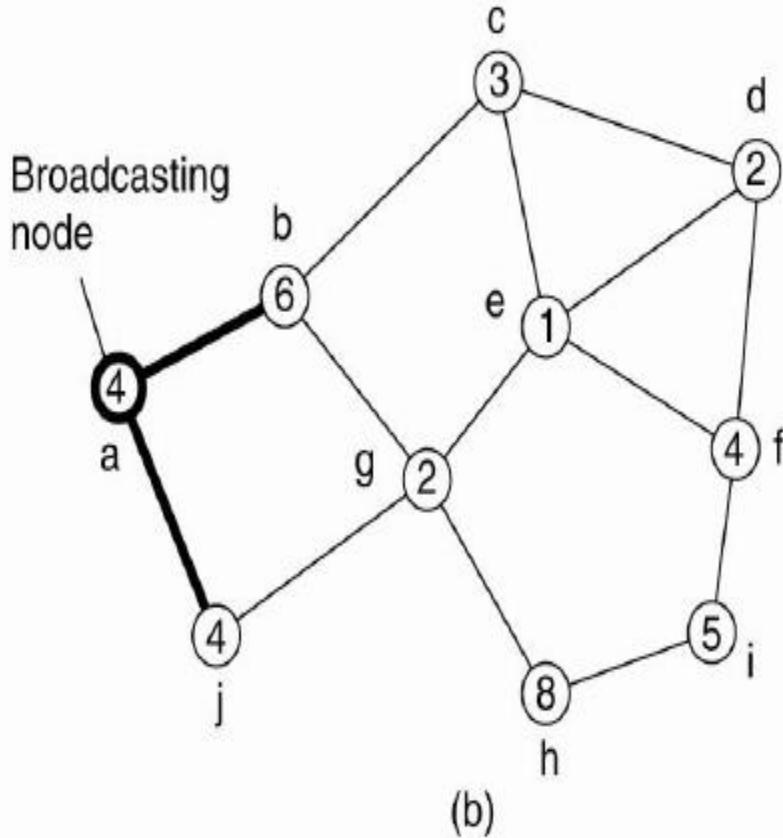
*Node with high capacity & battery lifetime elected as leader*

- Any node(source) start election by sending “ELECTION” message to its neighbors(within range)

- If a node receives election message for first time
  - designate sender as parent
  - forward election message to its neighbors except parent
  - wait for acknowledge from its neighbors before acknowledging its parent
- if a node receives election message other than its designated parent
  - acknowledge the receipt
  - also send report information(battery lifetime, resource capacity)
- While receiving acknowledgement from neighbors
  - compare and select high capacity neighbor
  - pass this information to parent along with ack
- In this way source will decide that which node can be selected as leader and broadcast this information to all other nodes



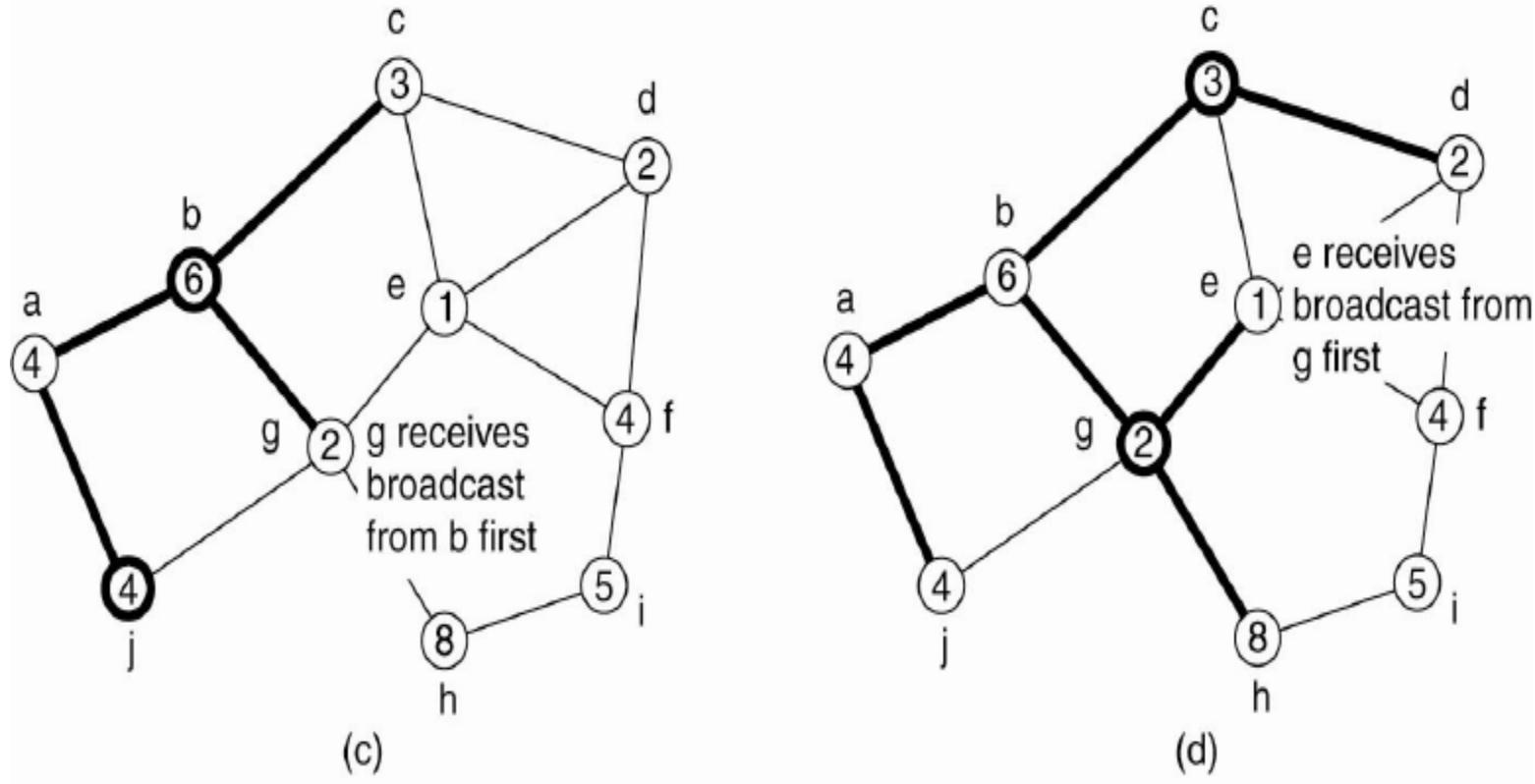
(a)



(b)

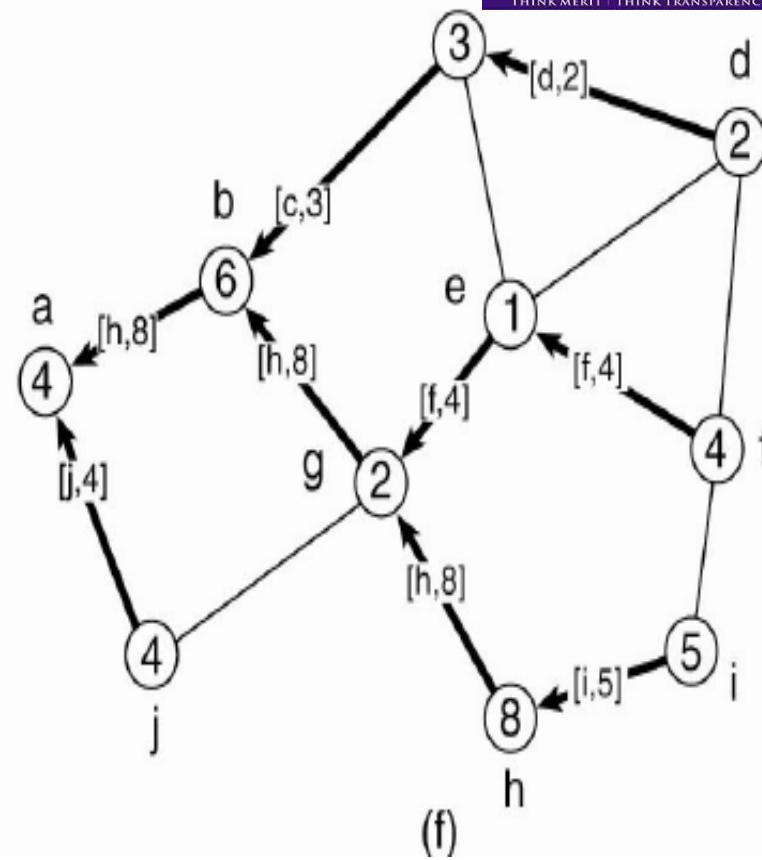
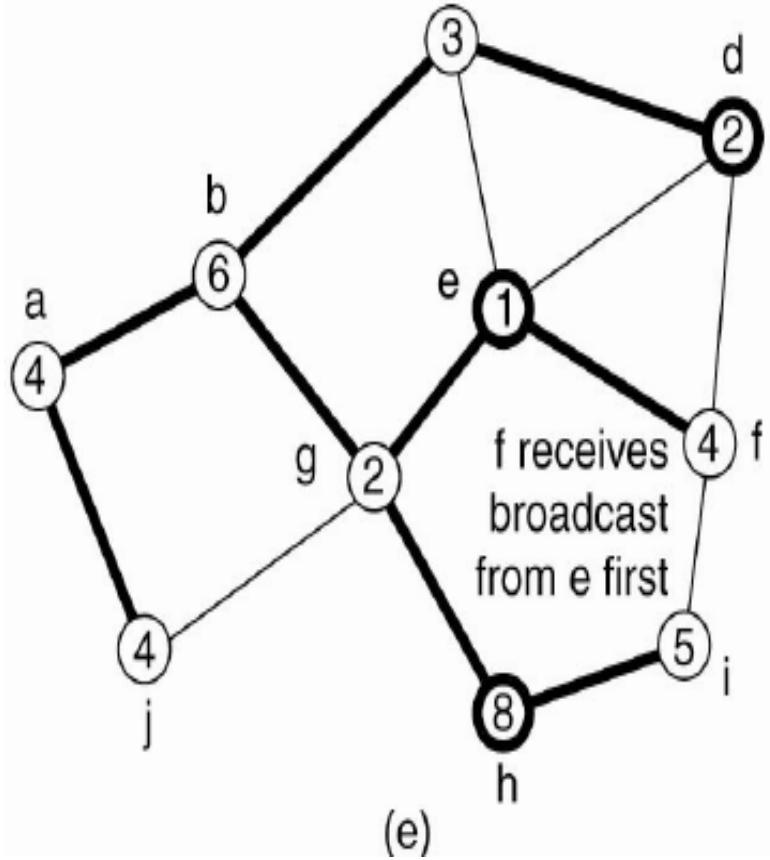
(a) – initial network

(b) - node 'a' send election message to its neighbors



(c) – node ‘b’ send election message to its neighbors

(d) – node ‘c’ and ‘g’ send election message to its neighbors



- (e) – node 'e' and 'h' send election message to its neighbors
- (f) – each node send acknowledgement along with report information  
source node 'a' will elect 'h' as leader ( since h has high capacity i.e 8 )  
and broadcast this information to all

# Issues

1. What happens if more than one process initiates election?
  - Each source tag its election message with unique identifier
  - Other processes will participate only in the election with highest identifier
2. What happens when network get partitioned?
3. What happens when nodes join/leave?

# **CSE409 - PARALLEL & DISTRIBUTED SYSTEMS**

## **Unit-IV Distributed Computing**

### **Fault Tolerance**

**Dr. P. Padmakumari**  
**CSE/SoC/SASTRA**



## UNIT - IV

**Coordination:** Clock Synchronization - Logical clocks - **Mutual Exclusion:** Centralized algorithm - Distributed Algorithm - Token-ring algorithm - Decentralized Algorithm - **Election Algorithms:** Bully algorithm - Ring algorithm - Elections in wireless environment and large scale systems

**Fault Tolerance:** Introduction to fault tolerance - Concepts - Failure models - Failure masking by redundancy - **Reliable client server communication:** Point to point communication - RPC semantics in the presence of failures - **Reliable Group Communication:** Atomic multicast - Distributed commit - Recovery

# Introduction to Fault Tolerance

## Dependability

### Basics

A **component** provides **services** to **clients**. To provide services, the component may require the services from other components  $\Rightarrow$  a component may **depend** on some other component.

### Specifically

A component  $C$  depends on  $C^*$  if the **correctness** of  $C$ 's behavior depends on the correctness of  $C^*$ 's behavior. (Components are processes or channels.)

### Requirements related to dependability

Requirement	Description
Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	How easy can a failed system be repaired

# Introduction to Fault Tolerance

## Reliability versus availability

### Reliability $R(t)$ of component $C$

Conditional probability that  $C$  has been functioning correctly during  $[0, t)$  given  $C$  was functioning correctly at time  $T = 0$ .

### Traditional metrics

- Mean Time To Failure (*MTTF*): The average time until a component fails.
- Mean Time To Repair (*MTTR*): The average time needed to repair a component.
- Mean Time Between Failures (*MTBF*): Simply  $MTTF + MTTR$ .

# Introduction to Fault Tolerance

## Reliability versus availability

Availability  $A(t)$  of component  $C$

Average fraction of time that  $C$  has been up-and-running in interval  $[0, t]$ .

- Long-term availability  $A$ :  $A(\infty)$
- Note:  $A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$

Observation

Reliability and availability make sense only if we have an accurate notion of what a failure actually is.

# Introduction to Fault Tolerance

## Terminology

Failure, error, fault

Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

# Introduction to Fault Tolerance

## Terminology

Handling faults

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don't hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

# Introduction to Fault Tolerance

## Terminology

Handling faults

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don't hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

# Failure Models

## Types of failures

Type	Description of server's behavior
Crash failure	Halts, but is working correctly until it halts
Omission failure	Fails to respond to incoming requests
<i>Receive omission</i>	Fails to receive incoming messages
<i>Send omission</i>	Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure	Response is incorrect
<i>Value failure</i>	The value of the response is wrong
<i>State-transition failure</i>	Deviates from the correct flow of control
Arbitrary failure	May produce arbitrary responses at arbitrary times

# Failure Models

## Dependability versus security

### Omission versus commission

Arbitrary failures are sometimes qualified as **malicious**. It is better to make the following distinction:

- **Omission failures**: a component fails to take an action that it should have taken
- **Commission failure**: a component takes an action that it should not have taken

### Observation

Note that **deliberate** failures, be they omission or commission failures are typically security problems. Distinguishing between deliberate failures and unintentional ones is, in general, impossible.

# Failure Models

## Halting failures

### Scenario

$C$  no longer perceives any activity from  $C^*$  — a halting failure? Distinguishing between a crash or omission/timing failure may be impossible.

### Asynchronous versus synchronous systems

- **Asynchronous system:** no assumptions about process execution speeds or message delivery times → cannot reliably detect crash failures.
- **Synchronous system:** process execution speeds and message delivery times are bounded → we can reliably detect omission and timing failures.
- In practice we have **partially synchronous systems:** most of the time, we can assume the system to be synchronous, yet there is no bound on the time that a system is asynchronous → can normally reliably detect crash failures.

# Failure Models

## Halting failures

Assumptions we can make

Halting type	Description
Fail-stop	Crash failures, but reliably detectable
Fail-noisy	Crash failures, eventually reliably detectable
Fail-silent	Omission or crash failures: clients cannot tell what went wrong
Fail-safe	Arbitrary, yet benign failures (i.e., they cannot do any harm)
Fail-arbitrary	Arbitrary, with malicious failures

# Failure Models

## Redundancy for failure masking

### Types of redundancy

- **Information redundancy**: Add extra bits to data units so that errors can be recovered when bits are garbled.
- **Time redundancy**: Design a system such that an action can be performed again if anything went wrong. Typically used when faults are transient or intermittent.
- **Physical redundancy**: add equipment or processes in order to allow one or more components to fail. This type is extensively used in distributed systems.

# Failure Models

## Redundancy for failure masking

### Types of redundancy

- **Information redundancy**: Add extra bits to data units so that errors can be recovered when bits are garbled.
- **Time redundancy**: Design a system such that an action can be performed again if anything went wrong. Typically used when faults are transient or intermittent.
- **Physical redundancy**: add equipment or processes in order to allow one or more components to fail. This type is extensively used in distributed systems.

# Distributed Systems

Fault Tolerance

Chapter 8

# Course/Slides Credits

Note: all course presentations are based on those developed by Andrew S. Tanenbaum and Maarten van Steen. They accompany their "Distributed Systems: Principles and Paradigms" textbook (1<sup>st</sup> & 2<sup>nd</sup> editions).

[http://www.prenhall.com/divisions/esm/app/author\\_tanenbaum/custom/dist\\_sys\\_1e/index.html](http://www.prenhall.com/divisions/esm/app/author_tanenbaum/custom/dist_sys_1e/index.html)

And additions made by Paul Barry in course CW046-4: Distributed Systems

<http://glasnost.itcarlow.ie/~barryp/net4.html>

# Fault Tolerance Basic Concepts

- Dealing successfully with partial failure within a Distributed System.
- Being fault tolerant is strongly related to what are called dependable systems.
- Dependability implies the following:
  1. Availability
  2. Reliability
  3. Safety
  4. Maintainability

# Dependability Basic Concepts

- *Availability* – the system is ready to be used immediately.
- *Reliability* – the system can run continuously without failure.
- *Safety* – if a system fails, nothing catastrophic will happen.
- *Maintainability* – when a system fails, it can be repaired easily and quickly (sometimes, without its users noticing the failure).

## But, What Is “Failure”?

- A system is said to “fail” when it *cannot meet* its promises.
- A failure is brought about by the *existence* of “errors” in the system.
- The *cause* of an error is a “fault”.

# Types of Fault

- There are three main types of ‘fault’:
  1. *Transient Fault* – appears once, then disappears.
  2. *Intermittent Fault* – occurs, vanishes, reappears; but: follows no real pattern (worst kind).
  3. *Permanent Fault* – once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally.

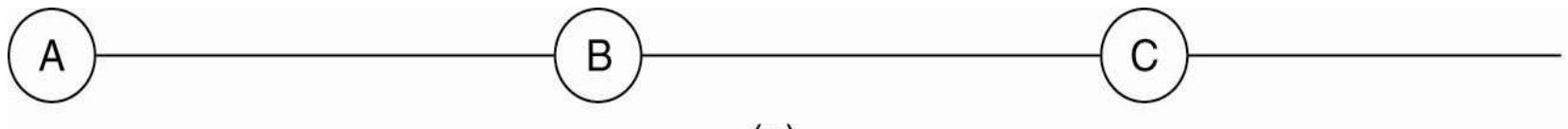
# Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
<i>Receive omission</i>	A server fails to receive incoming messages
<i>Send omission</i>	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	A server's response is incorrect
<i>Value failure</i>	The value of the response is wrong
<i>State transition failure</i>	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

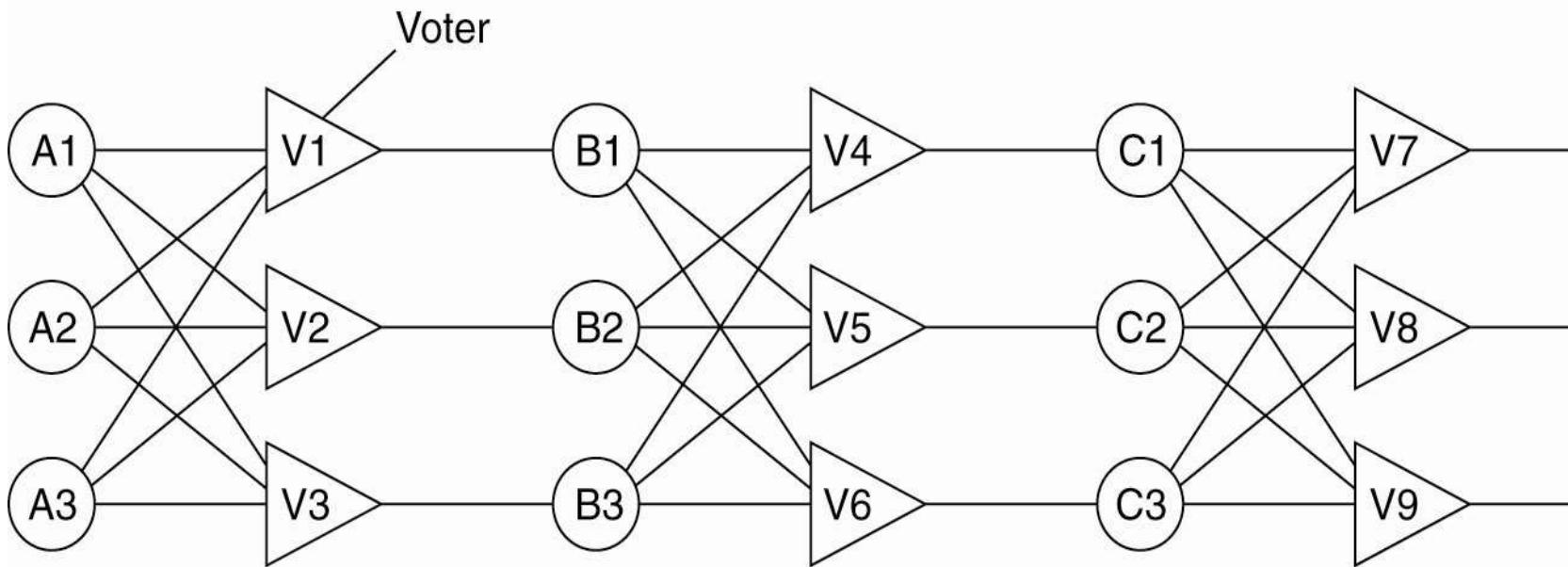
# Failure Masking by Redundancy

- **Strategy:** hide the occurrence of failure from other processes using *redundancy*.
- Three main types:
  1. *Information Redundancy* – add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).
  2. *Time Redundancy* – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).
  3. *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

# Failure Masking by Redundancy



(a)



(b)

Triple modular redundancy

# DS Fault Tolerance Topics

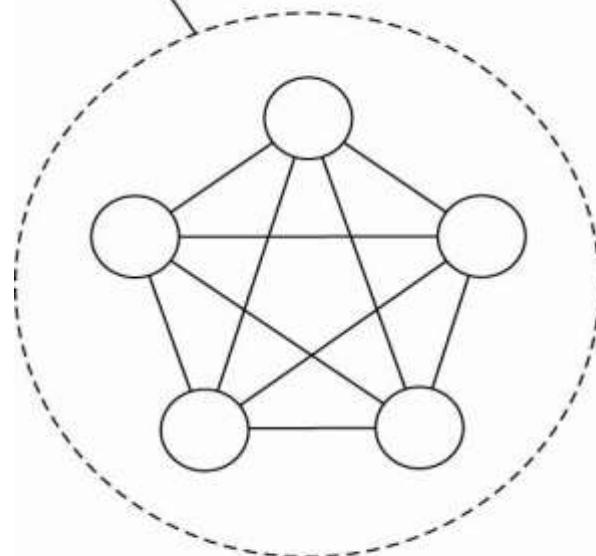
- Process Resilience
- Reliable Client/Server Communications
- Reliable Group Communication
- Distributed Commit
- Recovery Strategies

# Process Resilience

- Processes can be made fault tolerant by arranging to have a group of processes, with each member of the group being *identical*.
- A message sent to the group is delivered to all of the “copies” of the process (the group members), and then *only one* of them performs the required service.
- If one of the processes fail, it is assumed that one of the others will still be able to function (and service any pending request or operation).

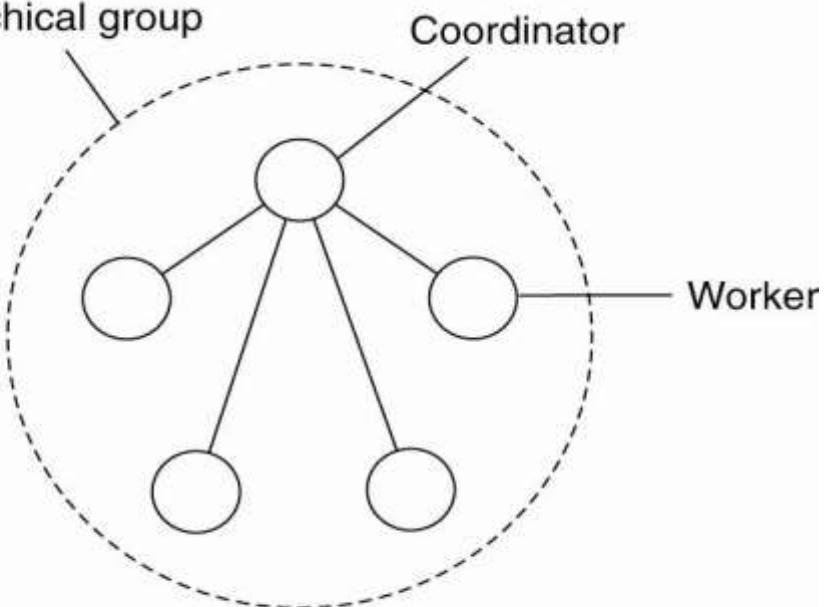
# Flat Groups versus Hierarchical Groups

Flat group



(a)

Hierarchical group



(b)

(a) Communication in a flat group.

(b) Communication in a simple hierarchical group.

# Failure Masking and Replication

- By organizing a *fault tolerant group of processes*, we can protect a single vulnerable process.
- There are two approaches to arranging the replication of the group:
  1. Primary (backup) Protocols
  2. Replicated-Write Protocols

# The Goal of Agreement Algorithms

- “To have all *non-faulty* processes reach consensus on some issue (quickly).”
- The **two-army problem**.
- Even with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.

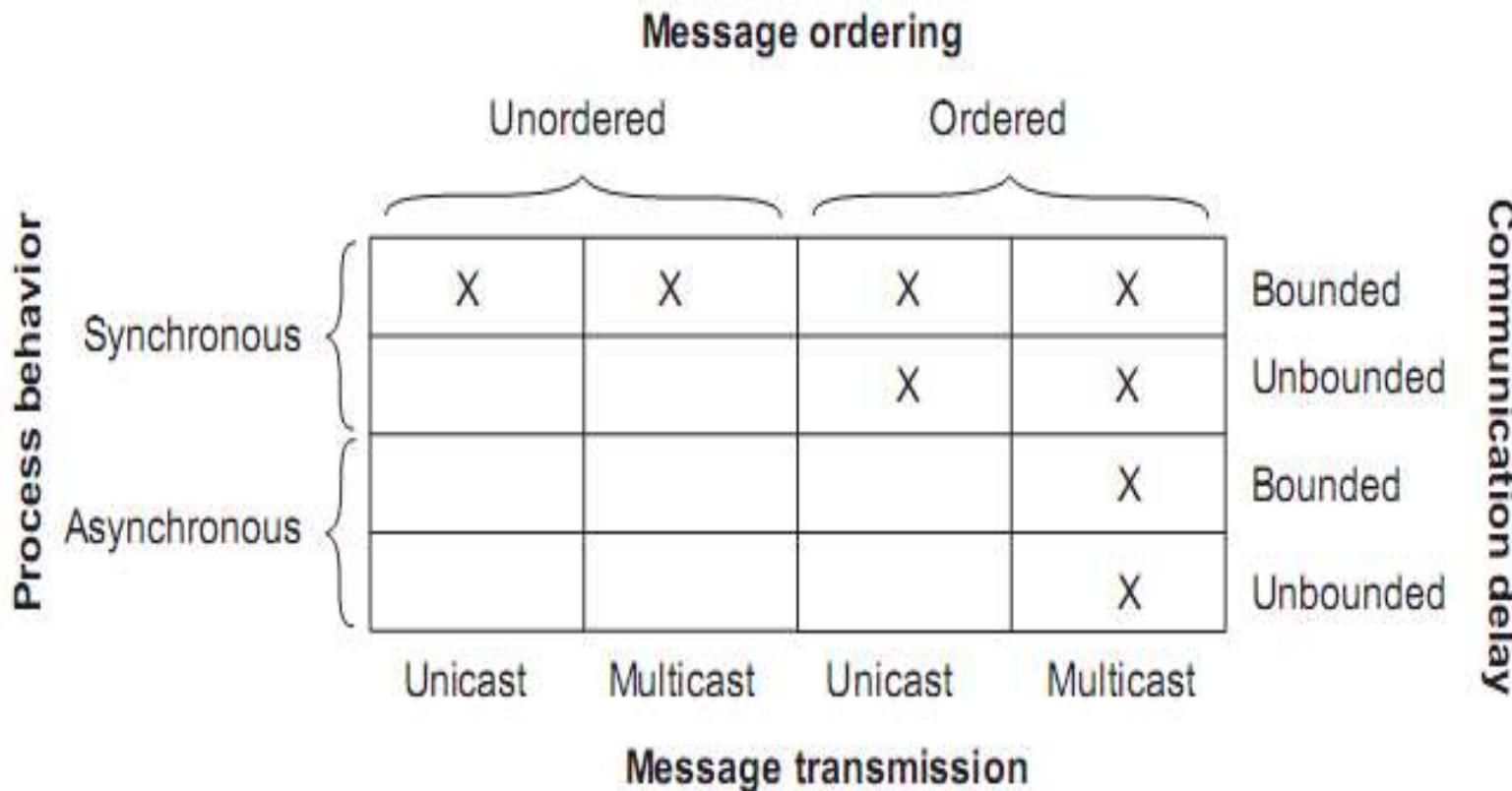
# History Lesson: The Byzantine Empire

- *Time:* 330-1453 AD.
- *Place:* Balkans and Modern Turkey.
- Endless conspiracies, intrigue, and untruthfulness were alleged to be common practice in the ruling circles of the day (*sounds strangely familiar ...* ).
- That is: it was typical for intentionally wrong and malicious activity to occur among the ruling group. A similar occurrence can surface in a DS, and is known as ‘Byzantine failure’.
- *Question:* how do we deal with such malicious group members within a distributed system?

# Agreement in Faulty Systems (1)

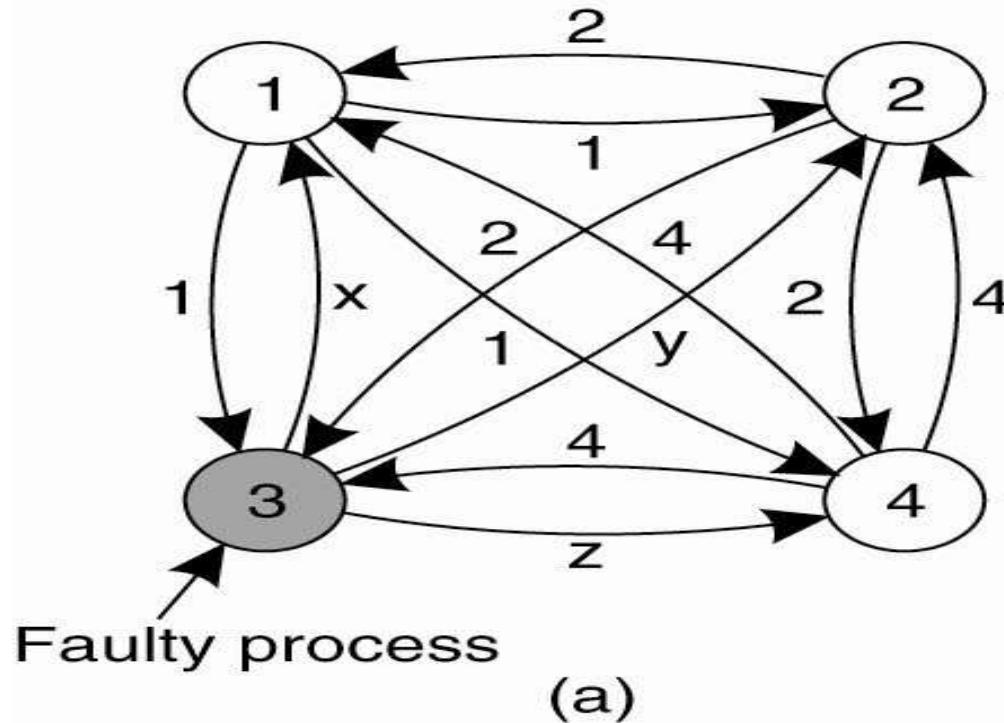
- Possible cases:
  1. Synchronous (lock-step) versus asynchronous systems.
  2. Communication delay is bounded (by globaly and predetermined maximum time) or not.
  3. Message delivery is ordered (in real-time) or not.
  4. Message transmission is done through unicasting or multicasting.

# Agreement in Faulty Systems (2)



Circumstances under which distributed agreement can be reached

# Agreement in Faulty Systems (3)



The Byzantine agreement problem for three non-faulty and one faulty process. (a) Each process sends their value to the others.

# Agreement in Faulty Systems (4)

1 Got(1, 2, x, 4)  
2 Got(1, 2, y, 4)  
3 Got(1, 2, 3, 4)  
4 Got(1, 2, z, 4)

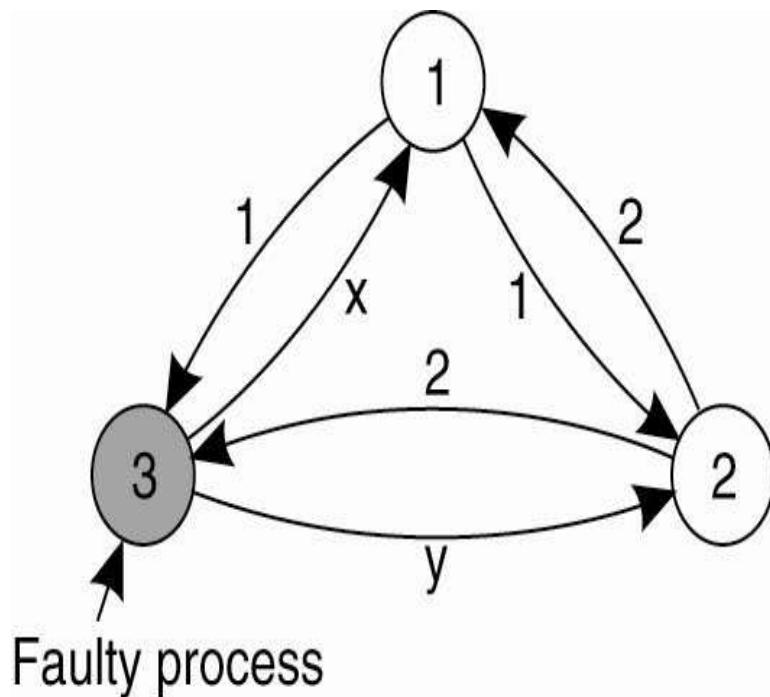
(b)

$\frac{1 \text{ Got}}{(1, 2, y, 4)}$      $\frac{2 \text{ Got}}{(1, 2, x, 4)}$      $\frac{4 \text{ Got}}{(1, 2, x, 4)}$   
 $(a, b, c, d)$      $(e, f, g, h)$      $(1, 2, y, 4)$   
 $(1, 2, z, 4)$      $(1, 2, z, 4)$      $(i, j, k, l)$

(c)

The Byzantine agreement problem for three non-faulty and one faulty process. (b) The vectors that each process assembles based on (a).  
(c) The vectors that each process receives in step 3.

# Agreement in Faulty Systems (5)



(a)

1 Got(1, 2, x)  
2 Got(1, 2, y)  
3 Got(1, 2, 3)

(b)

$$\frac{1 \text{ Got} \\ (1, 2, y)}{(a, b, c)} \quad \frac{2 \text{ Got} \\ (1, 2, x)}{(d, e, f)}$$

(c)

The same as before, except now with two correct process and one faulty process

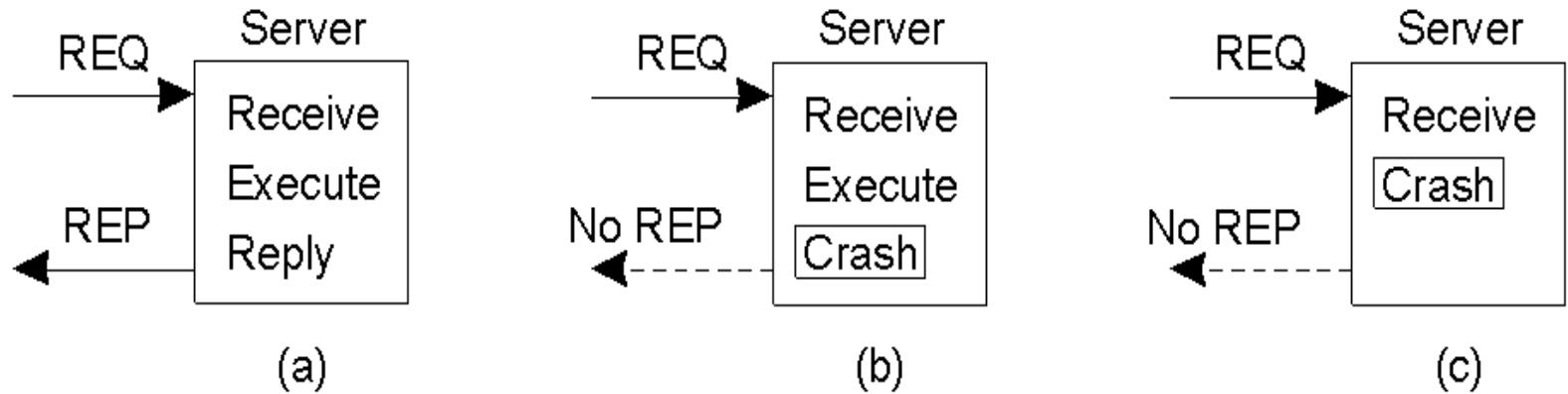
# Reliable Client/Server Communications

- In addition to process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures.
- In practice, the focus is on masking *crash* and *omission* failures.
- *For example:* the point-to-point TCP masks omission failures by guarding against lost messages using ACKs and retransmissions. However, it performs poorly when a crash occurs (although a DS may try to mask a TCP crash by automatically re-establishing the lost connection).

# RPC Semantics and Failures

- The RPC mechanism works well as long as both the client and server function perfectly.
- Five classes of RPC failure can be identified:
  1. *The client cannot locate the server*, so no request can be sent.
  2. *The client's request to the server is lost*, so no response is returned by the server to the waiting client.
  3. *The server crashes after receiving the request*, and the service request is left acknowledged, but undone.
  4. *The server's reply is lost on its way to the client*, the service has completed, but the results never arrive at the client
  5. *The client crashes after sending its request*, and the server sends a reply to a newly-restarted client that may not be expecting it.

# The Five Classes of Failure (1)



- A server in client-server communication:
  - a) The normal case.
  - b) Crash *after* service execution.
  - c) Crash *before* service execution.

# The Five Classes of Failure (2)

- An appropriate exception handling mechanism can deal with a missing server. However, such technologies tend to be very language-specific, and they also tend to be non-transparent (which is a big DS ‘no-no’).
- Dealing with lost request messages can be dealt with easily using timeouts. If no ACK arrives in time, the message is resent. Of course, the server needs to be able to deal with the possibility of duplicate requests.

# The Five Classes of Failure (3)

- Server crashes are dealt with by implementing one of three possible implementation philosophies:
  1. *At least once semantics*: a guarantee is given that the RPC occurred at least once, but (also) possibly more than once.
  2. *At most once semantics*: a guarantee is given that the RPC occurred at most once, but possibly not at all.
  3. *No semantics*: nothing is guaranteed, and client and servers take their chances!
- It has proved difficult to provide *exactly once semantics*.

# Server Crashes (1)

- Remote operation: print some text and (when done) send a completion message.
- Three events that can happen at the server:
  1. Send the completion message (M),
  2. Print the text (P),
  3. Crash (C).

# Server Crashes (2)

- These three events can occur in six different orderings:
  1.  $M \rightarrow P \rightarrow C$ : A crash occurs after sending the completion message and printing the text.
  2.  $M \rightarrow C (\rightarrow P)$ : A crash happens after sending the completion message, but before the text could be printed.
  3.  $P \rightarrow M \rightarrow C$ : A crash occurs after sending the completion message and printing the text.
  4.  $P \rightarrow C (\rightarrow M)$ : The text printed, after which a crash occurs before the completion message could be sent.
  5.  $C (\rightarrow P \rightarrow M)$ : A crash happens before the server could do anything.
  6.  $C (\rightarrow M \rightarrow P)$ : A crash happens before the server could do anything.

# Server Crashes (3)

Client	Strategy M → P			Strategy P → M		
	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Reissue strategy						
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once  
DUP = Text is printed twice  
ZERO = Text is not printed at all

Different combinations of client and server strategies in the presence of server crashes

# The Five Classes of Failure (4)

- Lost replies are difficult to deal with.
- *Why* was there no reply? Is the server *dead*, *slow*, or did the reply just go *missing*? Emmmmm?
- A request that can be repeated any number of times without any nasty side-effects is said to be *idempotent*. (For example: a read of a static web-page is said to be idempotent).
- *Nonidempotent* requests (for example, the electronic transfer of funds) are a little harder to deal with. A common solution is to employ *unique sequence numbers*. Another technique is the inclusion of additional bits in a retransmission to identify it as such to the server.

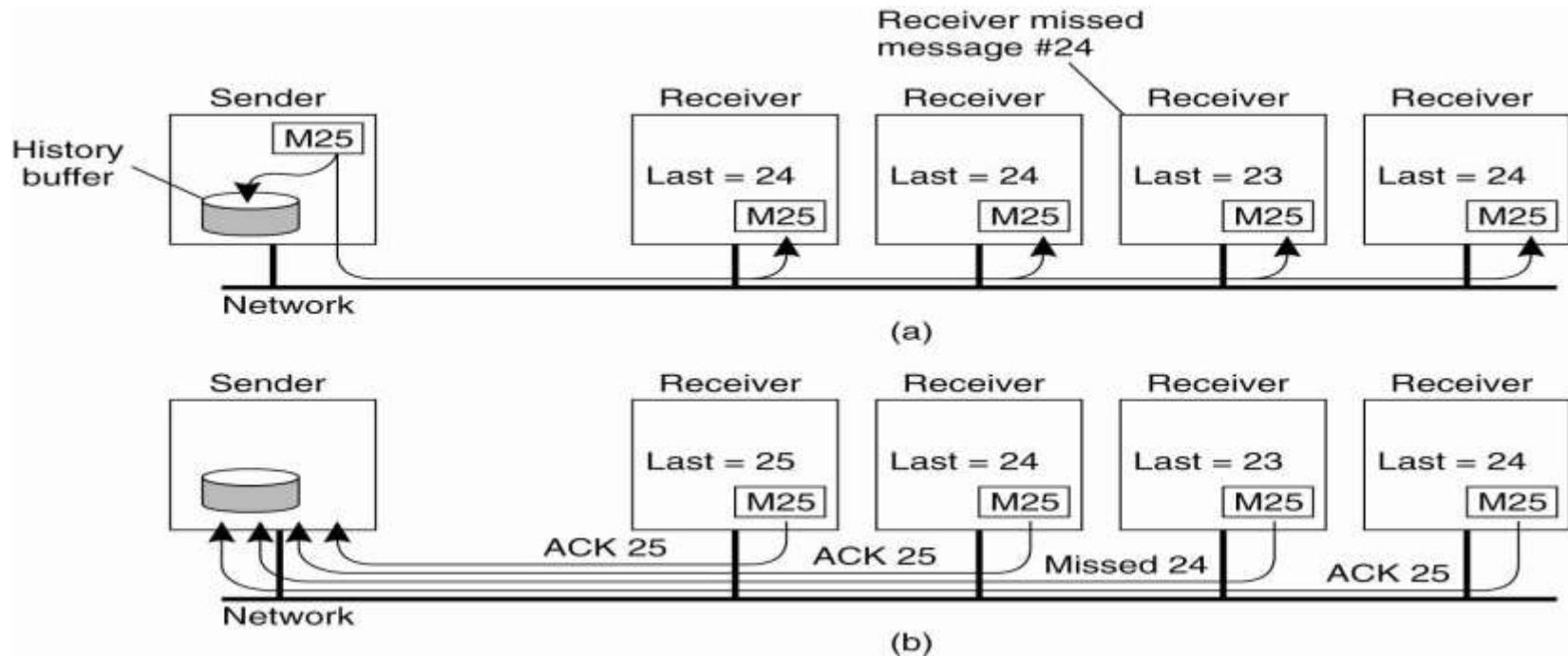
# The Five Classes of Failure (5)

- When a client crashes, and when an ‘old’ reply arrives, such a reply is known as an *orphan*.
- Four orphan solutions have been proposed:
  1. *extermination* (the orphan is simply killed-off).
  2. *reincarnation* (each client session has an *epoch* associated with it, making orphans easy to spot).
  3. *gentle reincarnation* (when a new epoch is identified, an attempt is made to locate a request's owner, otherwise the orphan is killed).
  4. *expiration* (if the RPC cannot be completed within a standard amount of time, it is assumed to have expired).
- In practice, however, none of these methods are desirable for dealing with orphans. Research continues ...

# Reliable Group Communication

- Reliable multicast services guarantee that all messages are delivered to all members of a process group.
- Sounds simple, but is surprisingly *tricky* (as multicasting services tend to be *inherently* unreliable).
- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution *scales poorly* as the group membership grows. Also:
  - What happens if a process *joins* the group during communication?
  - Worse: what happens if the sender of the multiple, reliable point-to-point channels *crashes* half way through sending the messages?

# Basic Reliable-Multicasting Schemes



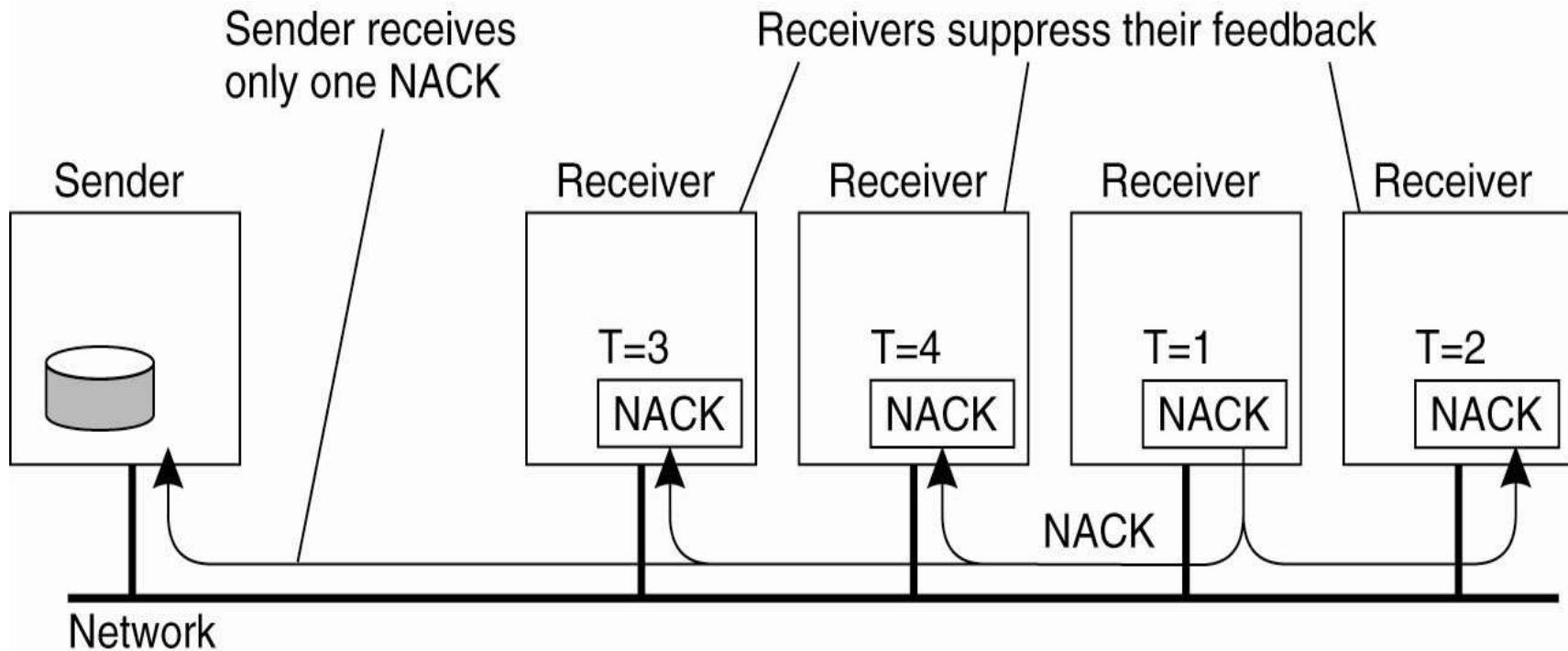
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.

(a) Message transmission. (b) Reporting feedback.

# SRM: Scalable Reliable Multicasting

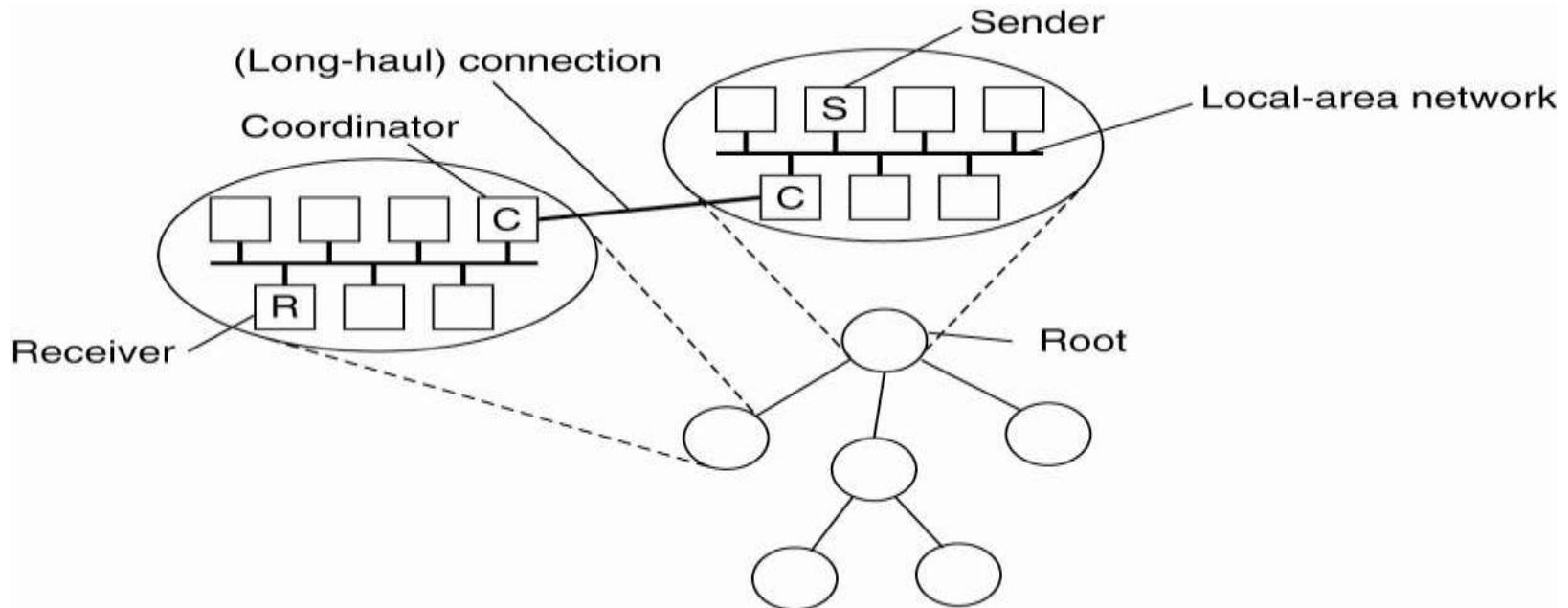
- Receivers *never* acknowledge successful delivery.
- **Only missing messages are reported.**
- NACKs are multicast to all group members.
- This allows other members to suppress their feedback, if necessary.
- To avoid “retransmission clashes”, each member is required to wait a random delay prior to NACKing.

# Nonhierarchical Feedback Control



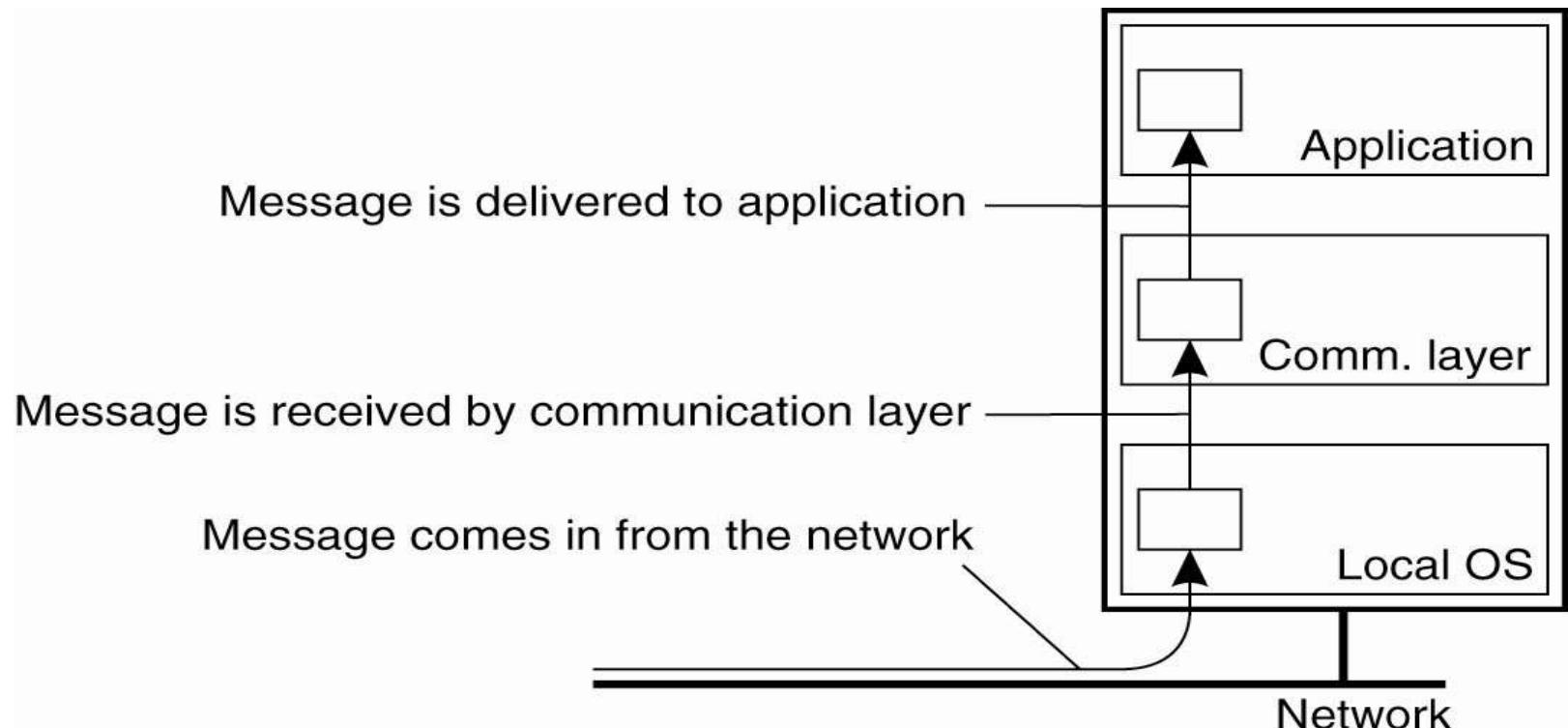
Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others

# Hierarchical Feedback Control



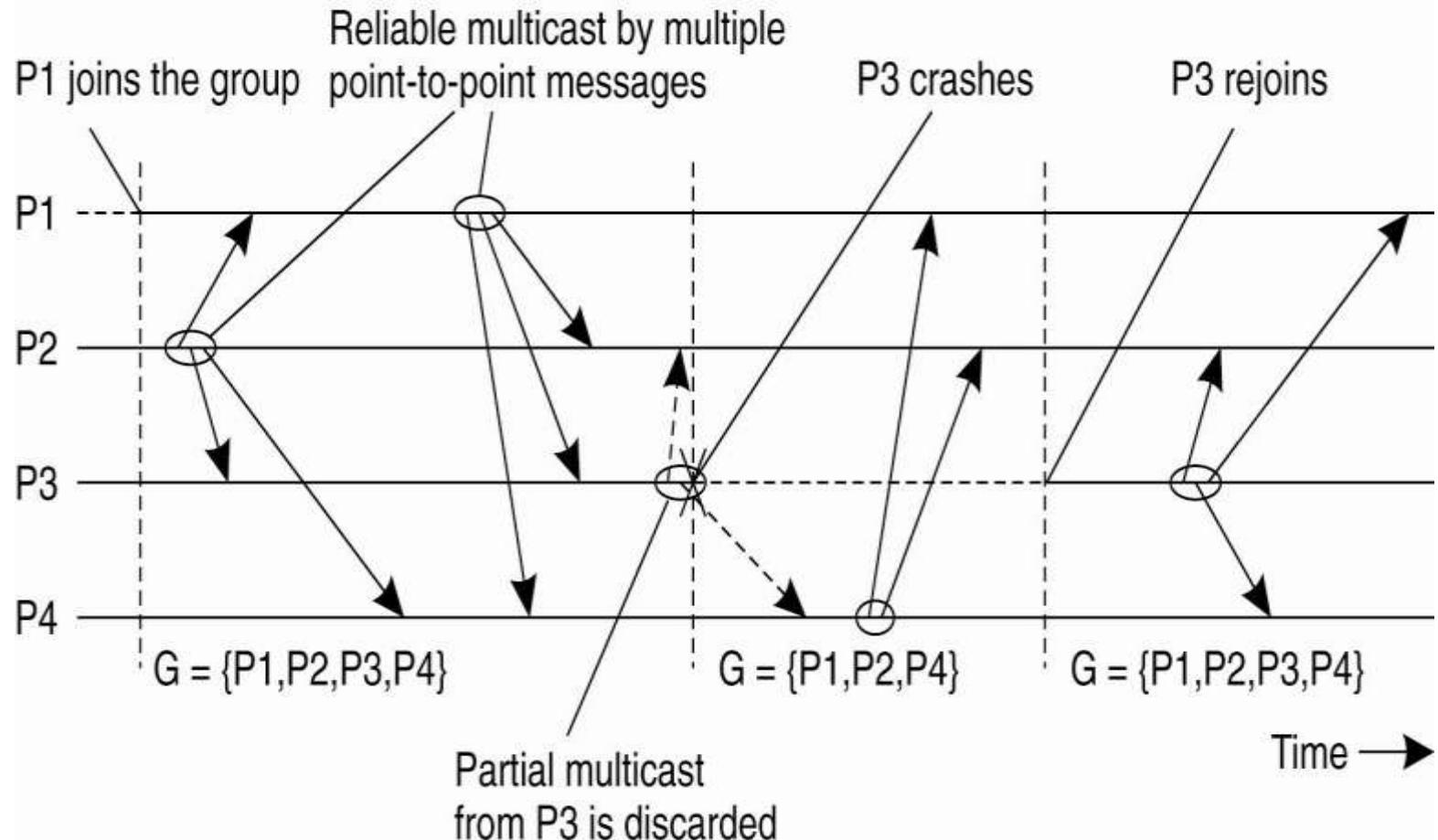
The essence of hierarchical reliable multicasting.  
Each local coordinator forwards the message to  
its children and later handles retransmission  
requests.

# Virtual Synchrony (1)



The logical organization of a distributed system to distinguish between message receipt and message delivery.

# Virtual Synchrony (2)



# Message Ordering (1)

- Four different orderings are distinguished:
  1. Unordered multicasts
  2. FIFO-ordered multicasts
  3. Causally-ordered multicasts
  4. Totally-ordered multicasts

## Message Ordering (2)

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

# Message Ordering (3)

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

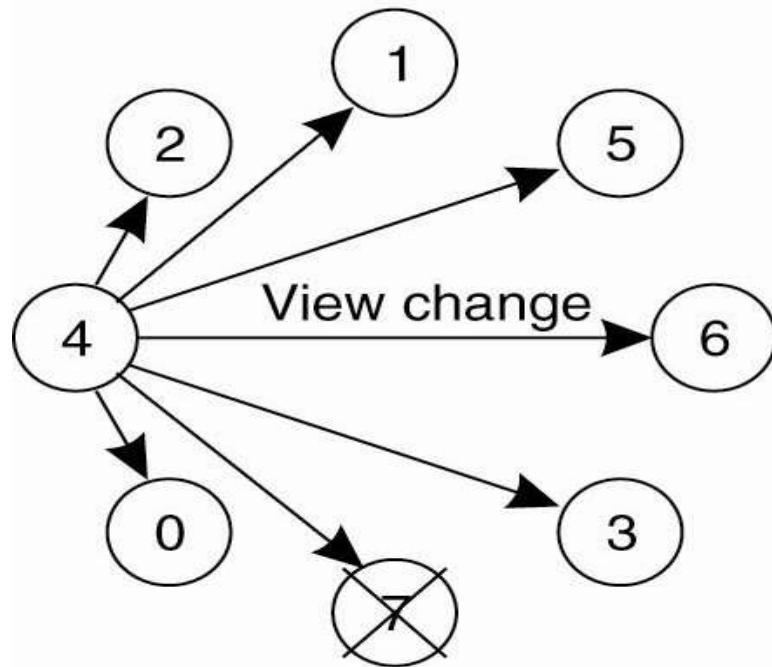
Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

# Implementing Virtual Synchrony (1)

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Six different versions of virtually  
synchronous reliable multicasting

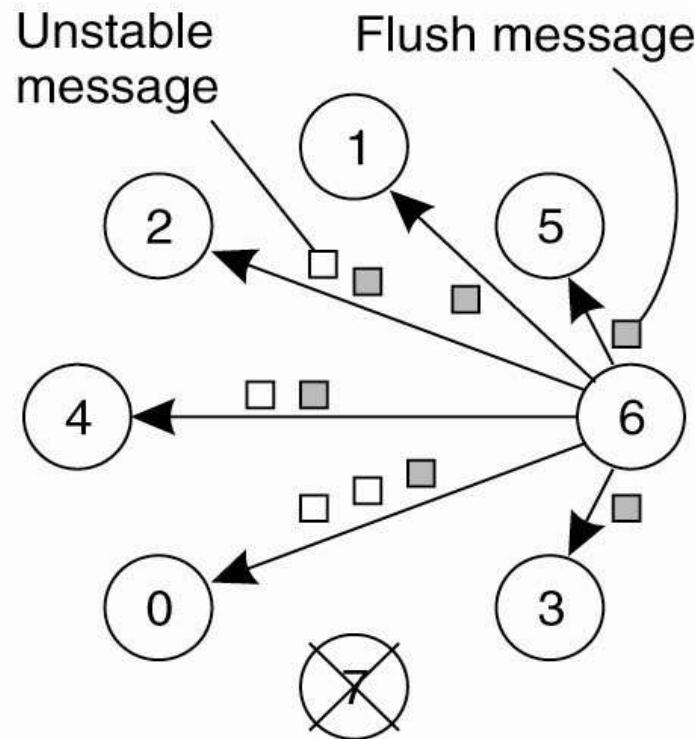
# Implementing Virtual Synchrony (2)



(a)

- (a) Process 4 notices that process 7 has crashed and sends a view change

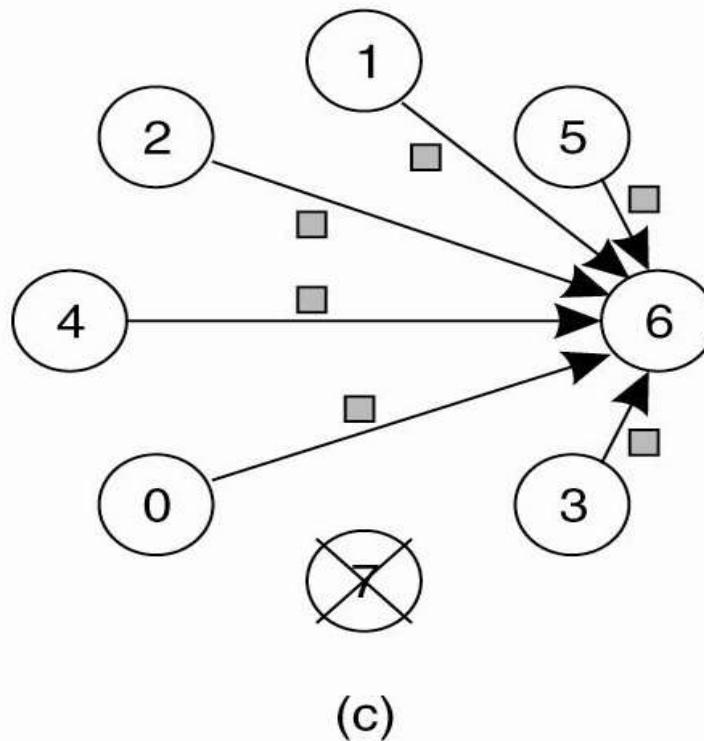
# Implementing Virtual Synchrony (3)



(b)

- (b) Process 6 sends out all its unstable messages, followed by a flush message

# Implementing Virtual Synchrony (4)



(c) Process 6 installs the new view when it has received a flush message from everyone else

# Distributed Commit

- **General Goal:**
  - *We want an operation to be performed by all group members, or none at all.*
  - [In the case of atomic multicasting, the operation is the delivery of the message.]

There are three types of “commit protocol”:

1. single-phase commit
2. two-phase commit
3. three-phase commit

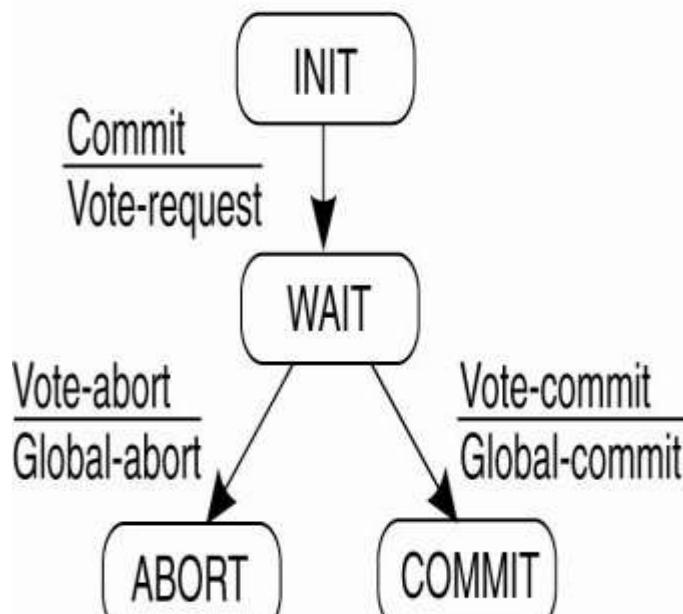
# Commit Protocols

- **One-Phase Commit Protocol:**
  - An elected coordinator tells all the other processes to perform the operation in question.
- But, what if a process cannot perform the operation? There's no way to tell the coordinator! Whoops ...
- **The solutions:**
  - The *Two-Phase* and *Three-Phase Commit Protocols*.

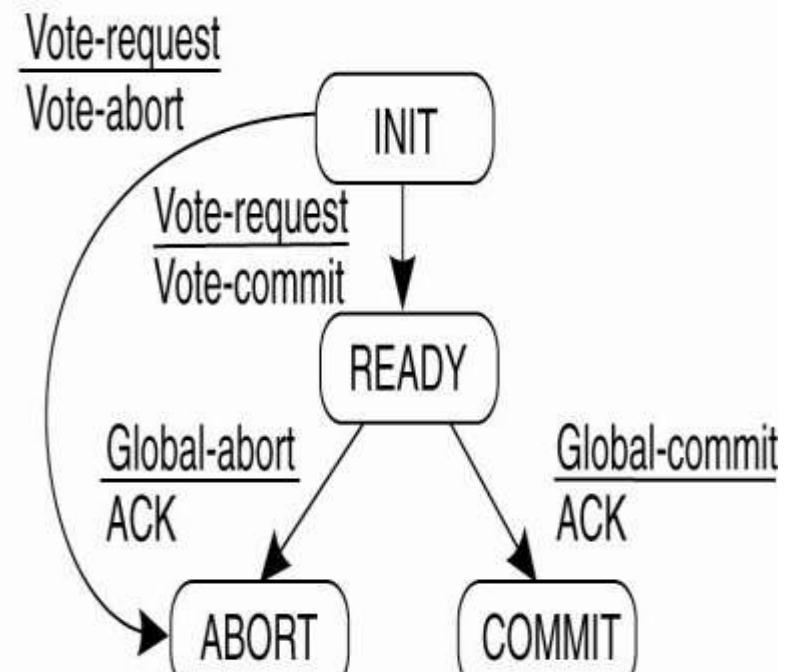
# The Two-Phase Commit Protocol

- First developed in 1978!!!
  - *Summarized: GET READY, OK, GO AHEAD.*
1. The coordinator sends a *VOTE\_REQUEST* message to all group members.
  2. The group member returns *VOTE\_COMMIT* if it can commit locally, otherwise *VOTE\_ABORT*.
  3. All votes are collected by the coordinator. A *GLOBAL\_COMMIT* is sent if all the group members voted to commit. If one group member voted to abort, a *GLOBAL\_ABORT* is sent.
  4. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

# Two-Phase Commit (1)



(a)



(b)

- (a) The finite state machine for the coordinator in 2PC.  
(b) The finite state machine for a participant.

## Two-Phase Commit (2)

<b>State of Q</b>	<b>Action by P</b>
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Actions taken by a participant P when residing in state READY and having contacted another participant Q

# Two-Phase Commit (3)

## Actions by coordinator:

```
write START_2PC to local log;  
multicast VOTE_REQUEST to all participants;  
while not all votes have been collected {  
    wait for any incoming vote;  
    if timeout {  
        write GLOBAL_ABORT to local log;  
        multicast GLOBAL_ABORT to all participants;  
        exit;  
    }  
    ...  
    record vote;  
}
```

Outline of the steps taken by the coordinator in a two-phase commit protocol

## Two-Phase Commit (4)

```
... if all participants sent VOTE_COMMIT and coordinator votes COMMIT {  
    write GLOBAL_COMMIT to local log;  
    multicast GLOBAL_COMMIT to all participants;  
} else {  
    write GLOBAL_ABORT to local log;  
    multicast GLOBAL_ABORT to all participants;  
}
```

Outline of the steps taken by the coordinator in a two-phase commit protocol

# Two-Phase Commit (5)

## **actions by participant:**

```
write INIT to local log;  
wait for VOTE_REQUEST from coordinator;  
if timeout {  
    write VOTE_ABORT to local log;  
    exit;  
}  
if participant votes COMMIT {  
    write VOTE_COMMIT to local log;  
    send VOTE_COMMIT to coordinator;  
    wait for DECISION from coordinator;  
    if timeout {  
        multicast DECISION_REQUEST to other participants;  
        wait until DECISION is received; /* remain blocked */  
        write DECISION to local log;  
    }  
    if DECISION == GLOBAL_COMMIT  
        write GLOBAL_COMMIT to local log;  
    else if DECISION == GLOBAL_ABORT  
        write GLOBAL_ABORT to local log;  
} else {  
    write VOTE_ABORT to local log;  
    send VOTE_ABORT to coordinator;  
}
```

(a) The steps taken by a participant process in 2PC

(a)

# Two-Phase Commit (6)

**Actions for handling decision requests:** /\* executed by separate thread \*/

```
while true {  
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */  
    read most recently recorded STATE from the local log;  
    if STATE == GLOBAL_COMMIT  
        send GLOBAL_COMMIT to requesting participant;  
    else if STATE == INIT or STATE == GLOBAL_ABORT  
        send GLOBAL_ABORT to requesting participant;  
    else  
        skip; /* participant remains blocked */  
}
```

(b)

- (b) The steps for handling incoming decision requests

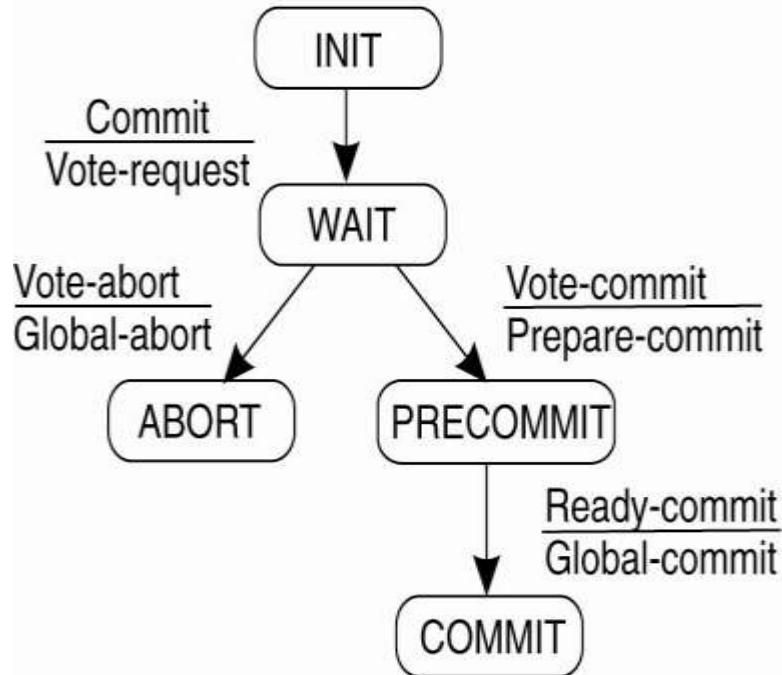
# Big Problem with Two-Phase Commit

- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.
- If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers* ...
- Two-Phase Commit is known as a **blocking-commit protocol** for this reason.
- The solution? *The Three-Phase Commit Protocol*.

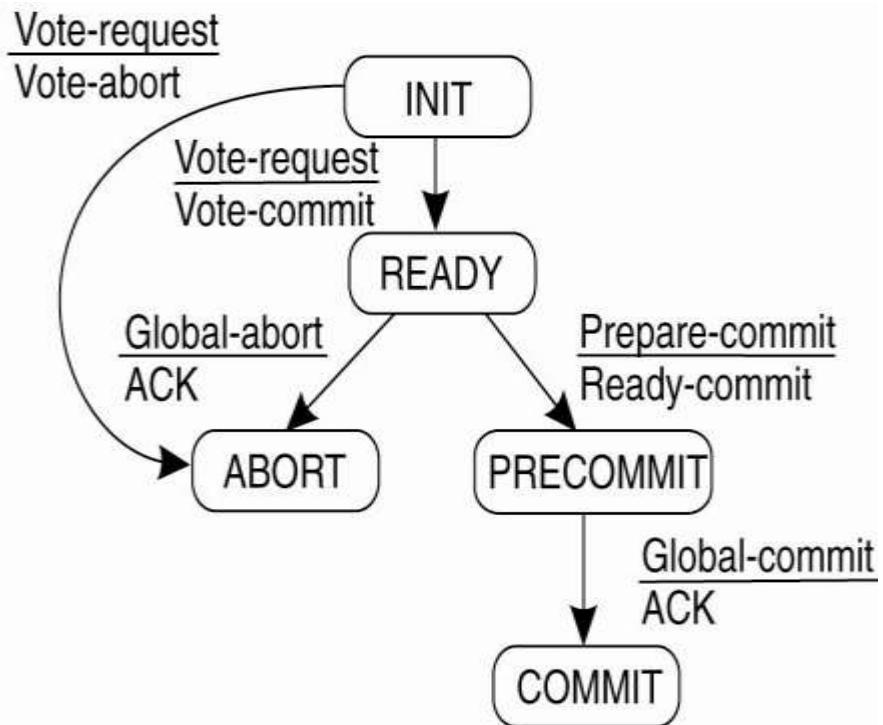
# Three-Phase Commit (1)

- The states of the coordinator and each participant satisfy the following two conditions:
  1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
  2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.

# Three-Phase Commit (2)



(a)



(b)

- (a) The finite state machine for the coordinator in 3PC.  
(b) The finite state machine for a participant.

# Recovery Strategies

- Once a failure has occurred, it is essential that the process where the failure happened *recovers* to a correct state.
- Recovery from an error is *fundamental* to fault tolerance.
- Two main forms of recovery:
  - Backward Recovery:** return the system to some previous correct state (using *checkpoints*), then continue executing.
  - Forward Recovery:** bring the system into a correct state, from which it can then continue to execute.

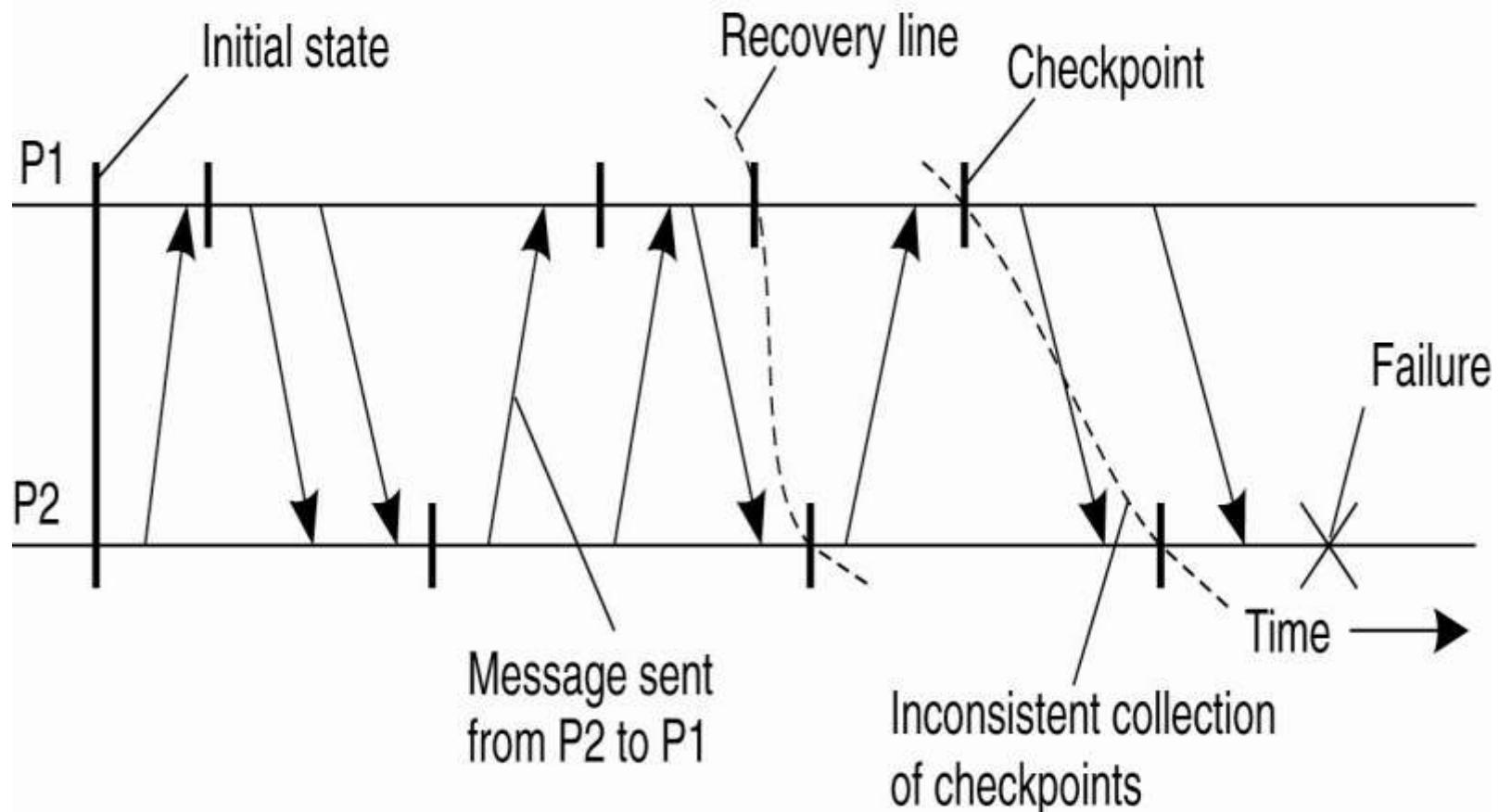
# Forward and Backward Recovery

- **Major disadvantage of Backward Recovery:**
  - Checkpointing can be very expensive (especially when errors are very rare).
  - [Despite the cost, backward recovery is implemented more often. The “logging” of information can be thought of as a type of checkpointing].
- **Major disadvantage of Forward Recovery:**
  - In order to work, all potential errors need to be accounted for *up-front*.
  - When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

# Recovery Example

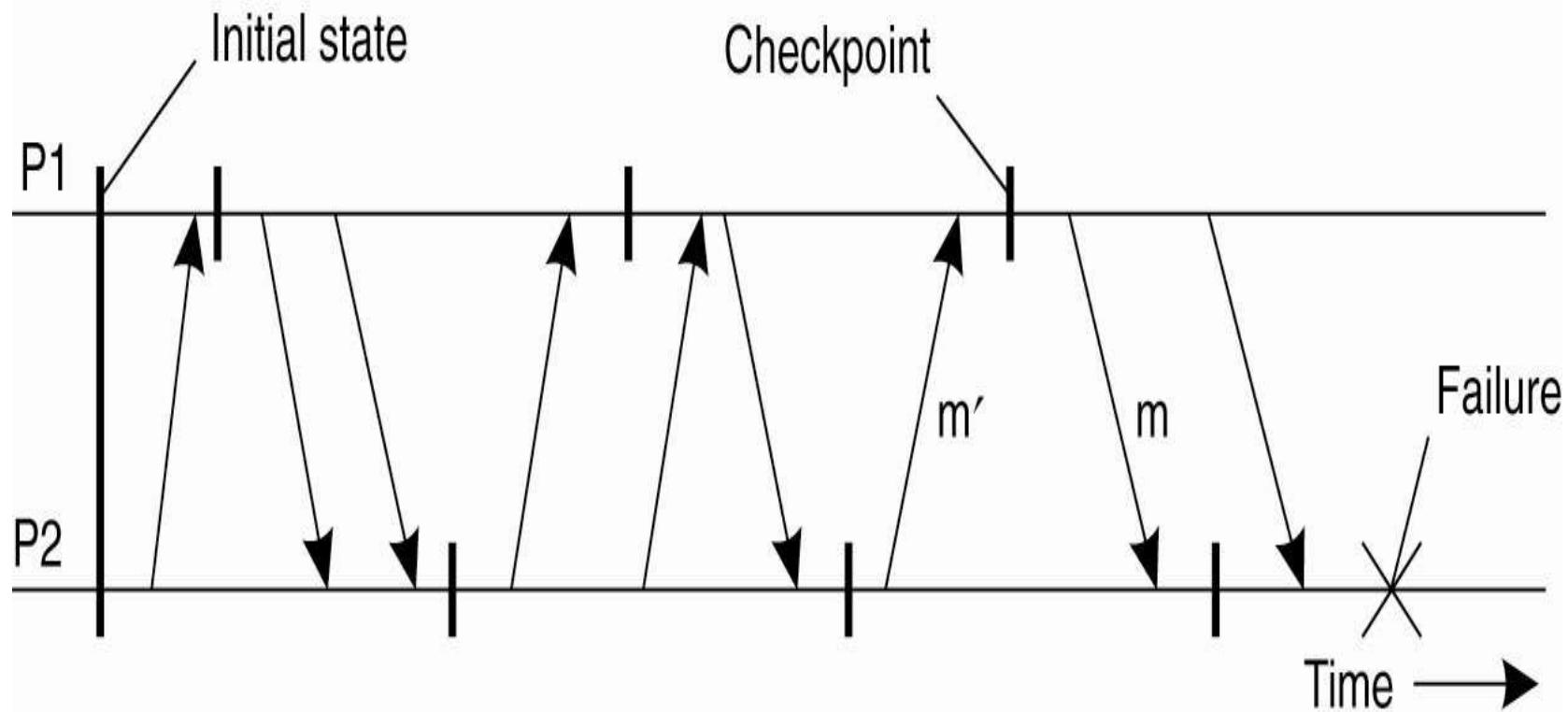
- Consider as an example:  
**Reliable Communications**
- *Retransmission* of a lost/damaged packet is an example of a backward recovery technique.
- When a lost/damaged packet can be reconstructed as a result of the receipt of other successfully delivered packets, then this is known as *Erasure Correction*. This is an example of a forward recovery technique.

# Checkpointing



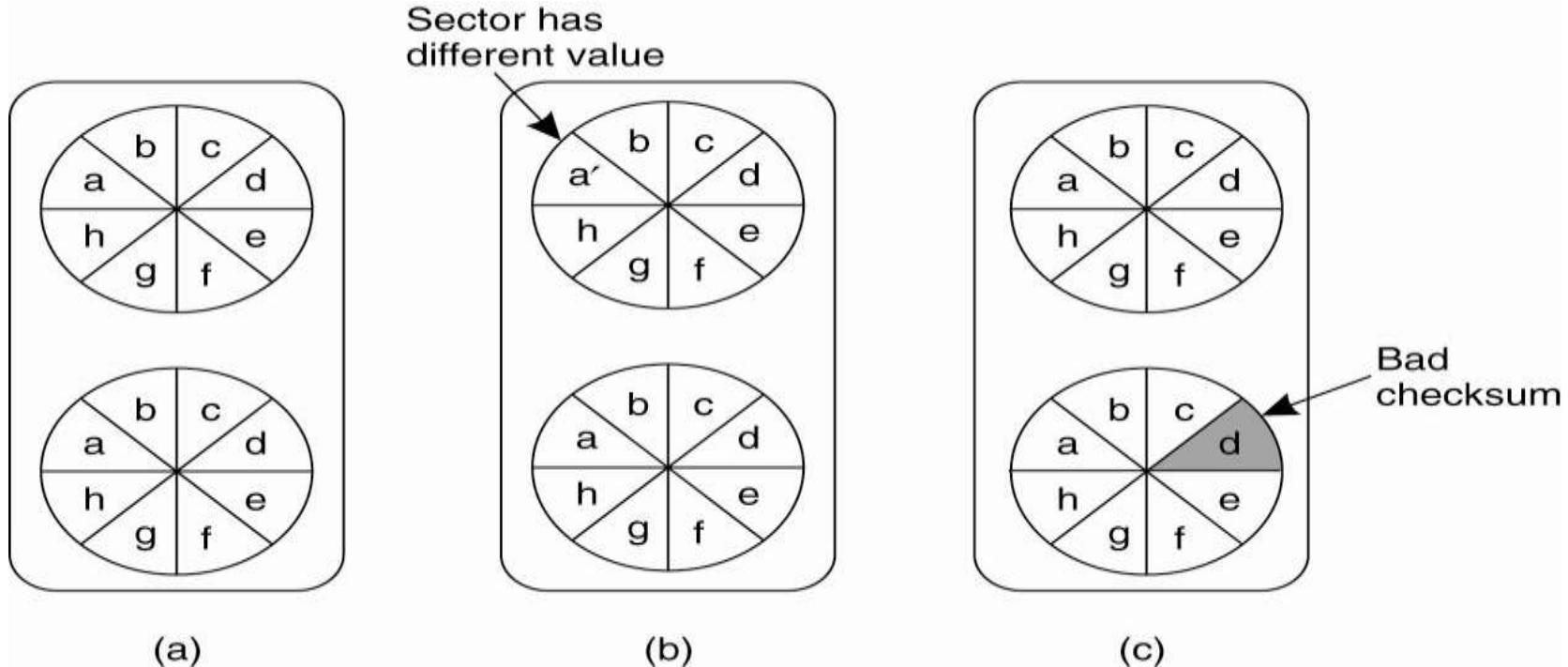
A recovery line

# Independent Checkpointing



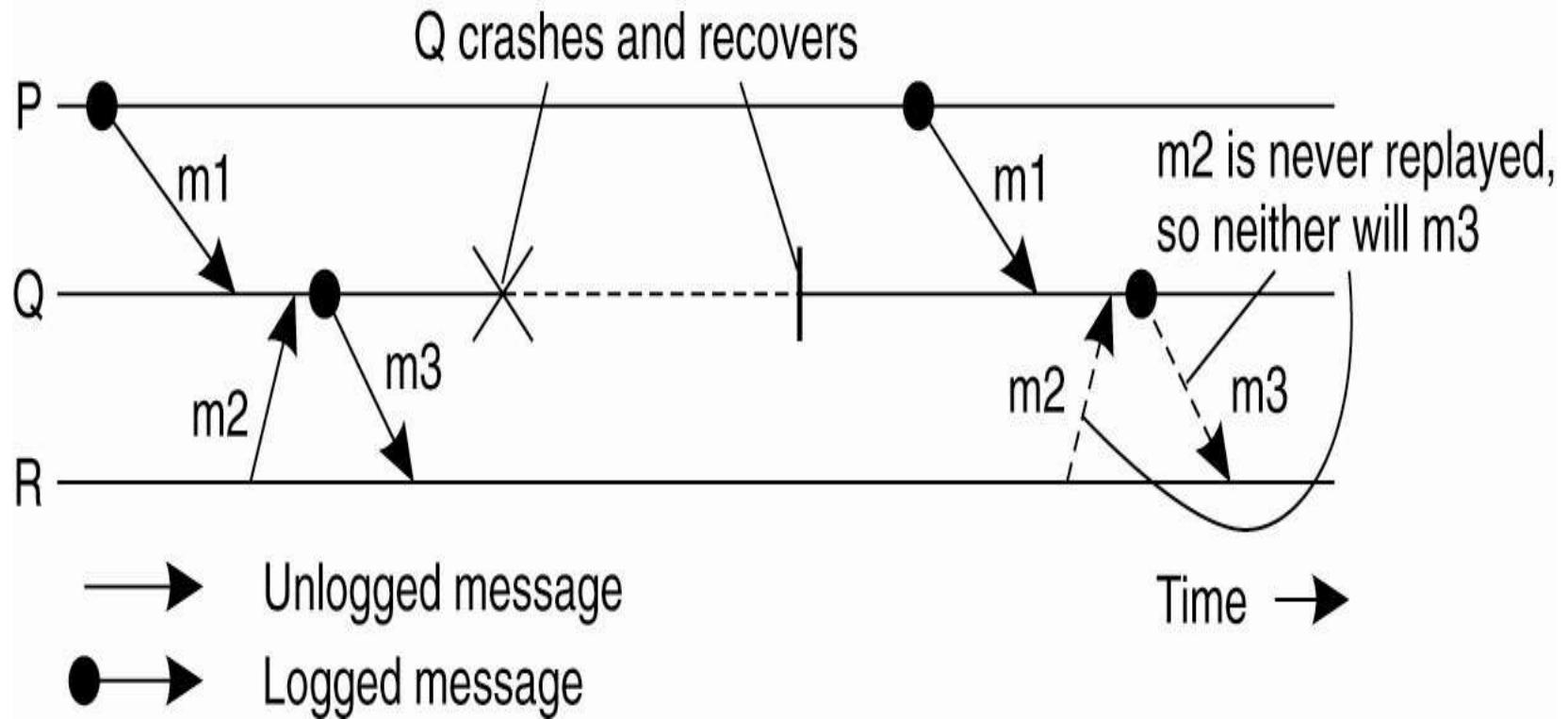
The domino effect – Cascaded rollback

# Recovery – Stable Storage



- (a) Stable storage.
- (b) Crash after drive 1 is updated.
- (c) Bad spot.

# Characterizing Message-Logging Schemes



Incorrect replay of messages after recovery,  
leading to an orphan process