



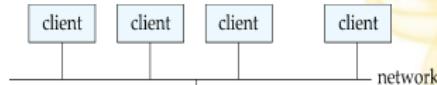
## UMD DATA605 - Big Data Systems

### 10.1: Parallel and Distributed Systems / DBs

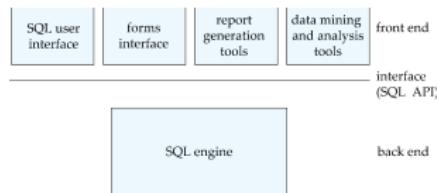
- **Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)

# Client-Server Architecture

- **Client-server:** Model for distributed applications partitioning tasks between:
  - *Clients:* Request service (e.g., dashboard, GUI, client applications)
  - *Servers:* Provide resource or service (e.g., database)



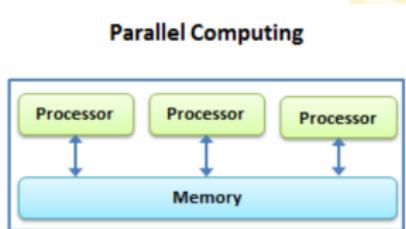
- **Architecture of a database system:**
  - *Back-end (Server):* manage access, query evaluation, optimization, concurrency control, recovery
  - *Front-end (Clients):* tools like forms, report-writers, GUI
- Interface between front-end and back-end:
  - SQL
  - Application programming interface (API)



# Parallel vs Distributed Computing

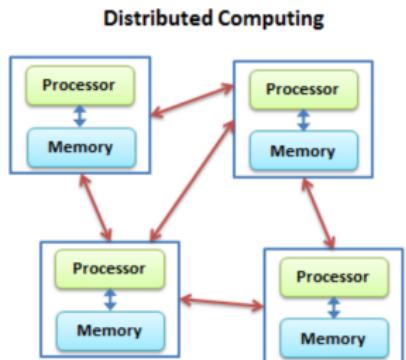
- **Parallel computing**

- One computer, multiple CPUs
- Cluster: many computers, multiple CPUs
- Homogenous, geographically close nodes
- Work on one task



- **Distributed computing**

- Autonomous, geographically separate systems
- Heterogeneous and distant
- Perform separate tasks



# Parallel Systems

---

- **Parallel systems** consist of:
  - Multiple processors
  - Multiple memories
  - Multiple disks
  - Fast interconnection network
- **Coarse-grain parallel machine**
  - Small number of powerful processors
  - E.g., your laptop with multiple CPUs
- **Fine-grain parallel machine**
  - Aka massively parallel
  - Thousands of smaller processors
  - Larger degree of parallelism
  - With or without shared memory
  - E.g., GPUs, The Connection Machine



*The Connection Machine, MIT, 1980s*

# Parallel Databases: Introduction

---

- Parallel DBs were the standard approach before MapReduce
- Parallel machines have become common and affordable
  - Prices of microprocessors, memory, and disks drop sharply
  - Desktop/laptop computers feature multiple processors
  - Trend will continue
- DBs are growing increasingly large
  - Large volumes of transaction data collected and stored for analysis
  - Multimedia objects increasingly stored in databases
- Large-scale parallel DBs increasingly used for:
  - Storing large volumes of data
  - Processing time-consuming queries
  - Providing high throughput for transaction processing

# Parallel Databases

---

- Internet / Big Data created need for large, fast DBs
  - Store petabytes of data
  - Process thousands of transactions per second (e.g., commerce website)
- **Databases can be parallelized**
  - Set-oriented nature of DB queries suits parallelization
  - Some operations are embarrassingly parallel
    - E.g., join between R and S on  $R.b = S.b$  as MapReduce task
- **Parallel DBs**
  - More transactions per second or less time per query
  - Throughput vs response time
  - Speed-up vs scale-up
- **Perfect speedup doesn't happen** due to:
  - Start-up costs
  - Task interference
  - Skew

# How to Measure Parallel Performance

- **Throughput**

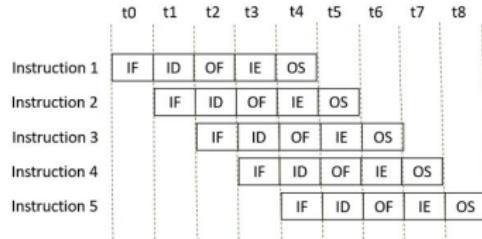
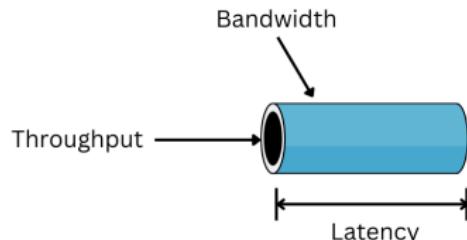
- Number of tasks completed in given time
- Increase by processing tasks in parallel

- **Latency**

- Time to complete single task from submission
- Decrease by performing subtasks in parallel

- **Throughput and latency are related but not the same**

- Increase throughput by reducing latency
- Increase throughput by pipelining (overlapping task execution)
  - E.g., building a car takes weeks, but one car is completed per hour
  - Pipelining of microprocessor instructions



# Speed-Up and Scale-Up: Intuition

---

- You have a workload to execute
  - Change workload  $M$ 
    - Number of DB transactions
    - Amount of DB data to query
- You need to execute the workload on a machine
  - Change computing power  $N$ 
    - Better CPU (scale vertically, scale up)
    - More CPUs (scale horizontally, scale out)
- Two ways to measure efficiency when increasing workload and computing power
  - Speed-up
    - Keep constant problem size  $M$
    - Increase machine power  $N$
  - Scale-up
    - Increase problem size  $M$
    - Increase machine power  $N$

# Speed-Up vs Scale-Up

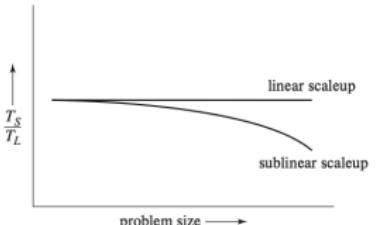
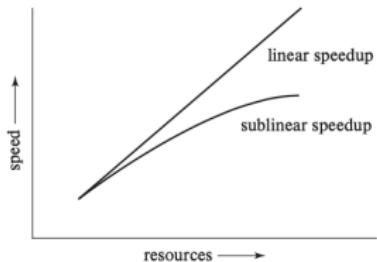
- The amount of computing power  $N$  can be changed
- The amount of work  $M$  can be changed
- Speed-up:** fixed-sized problem on a small system given to a system  $N$ -times larger

$$\text{speed-up} = \frac{\text{small system elapsed time}}{\text{large system elapsed time}}$$

- Speed-up is linear if equation equals  $N$
- Scale-up:** increase size of both problem  $M$  and system  $N$ 
  - $N$ -times larger system to perform  $M$ -times larger job

$$\text{scale-up} = \frac{\text{small system-problem time}}{\text{big system-problem time}}$$

- Scale-up is linear if equation equals 1

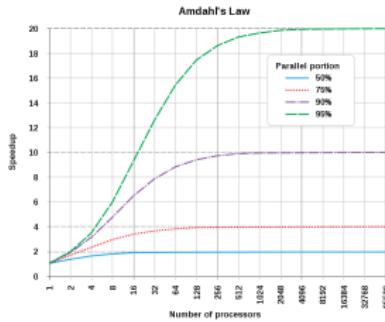


# Factors Limiting Speed-up and Scale-up

- Speed-up and scale-up are often sub-linear due to several issues
  - E.g., some computation is parallel, others sequential
- **Amdahl's Law**

- $p$  = fraction parallelizable
- $s$  = number of nodes
- $T$  = execution time serially
- $T(p) = \text{execution time on } s \text{ nodes} = (1 - p)T + (p/s)T$

$$\text{Speedup}(s) = \frac{T}{T(s)} = \frac{1}{(1 - p) + \frac{p}{s}}$$



- E.g.,
  - 90% parallelizable, max speed-up 10x
  - 50% parallelizable, max speed-up 2x (even with infinite nodes)

# Factors Limiting Speed-up and Scale-up

---

- **Startup costs**

- Starting many processes may dominate computation time
- E.g., DBs create a thread pool at startup

- **Interference**

- Processes compete for shared resources (e.g., system bus, disks, locks)
- Time spent waiting on other processes
- E.g., devs touching the same code create merge conflicts

- **Cost of synchronization**

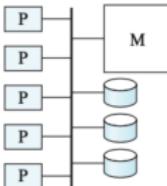
- Smaller work pieces increase synchronization complexity
- E.g., hiring many developers in a company

- **Skew**

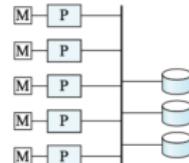
- Splitting work increases variance in task response time
- Difficult to split tasks equally
- Execution time determined by slowest task

# Topology of Parallel Systems

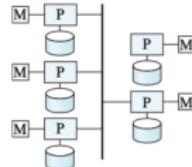
- Several ways to organize computation and storage
  - $M$  = memory
  - $P$  = processors
  - $D$  = disks
- **Topology**
  - Shared memory
  - Shared disk
  - Shared nothing
  - Hierarchical
- **Problems** are:
  - Cache coherency
  - Data communication
  - Fault tolerance
  - Resource congestion



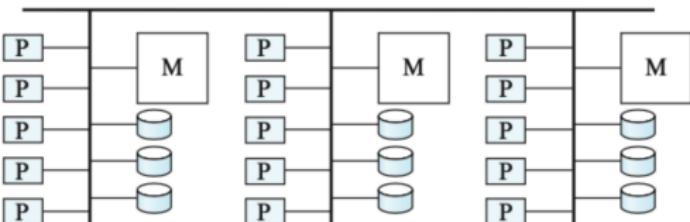
(a) shared memory



(b) shared disk



(c) shared nothing

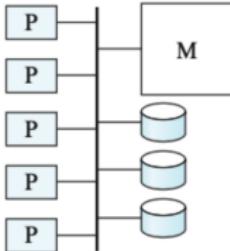


(d) hierarchical

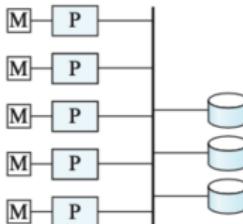


# Topology of Parallel Systems: Comparison

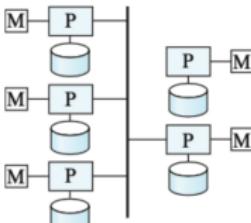
	Shared Memory	Shared Disk	Shared Nothing
Communication between processors	Extremely fast	Disk interconnect is very fast	Over a LAN, so slowest
Scalability?	Not beyond 32 or 64 or so (memory bus is the bottleneck)	Not very scalable (disk interconnect is the bottleneck)	Very very scalable
Notes	Cache-coherency an issue	Transactions complicated; natural fault-tolerance.	Distributed transactions are complicated (deadlock detection etc);
Main use	Low degrees of parallelism	Not used very often	Everywhere



(a) shared memory

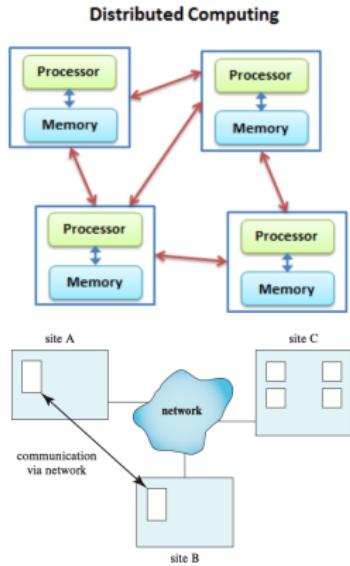


(b) shared disk



# Distributed Databases

- **Distributed DBs**
  - DB stored on nodes at geographically separated sites
  - Communicate through high-speed private networks or Internet
- Done due to necessity, e.g.,
  - Large corporation with global offices
  - Redundancy and disaster recovery
  - E.g., natural disasters, power outage, hacker attacks
  - Achieve high-availability despite failures
- Typically not for performance reasons
  - Use parallel DB for high performance
- Wide-area networks (WAN) vs Local-area networks (LAN)
  - Lower bandwidth
  - Higher latency
  - Greater probability of failures
  - Network-link failures may cause network partition

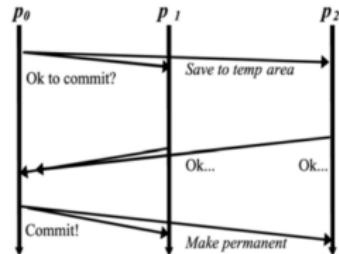


• Non-sharable of memory or disks

• Communication delay is high

# Consistency Issues in Distributed DB Systems

- Parallel and distributed DBs work well for query processing
  - Only reading data
- Updating a parallel or distributed DB requires consistency enforcement
- **Atomicity issues**
  - *Problem:* Transaction is all-or-nothing across multiple nodes
  - Two-phase commit (2PC) is a centralized approach
    - Commit decision delegated to a single coordinator node
    - Each node executes the transaction, reaching a “ready state”
    - If each node reaches ready state, coordinator commits
    - If a node fails in ready state, it can recover from failure (e.g., write-ahead logs on stable storage)
    - If a node aborts, coordinator aborts transaction
  - Distributed consensus, e.g.,
    - Paxos
    - Blockchain



**Coordinator :**

- multicast: *ok to commit?*
- collect replies
- all *ok* => *send commit*
- else => *send abort*

**Participant:**

- ok to commit* =>
- save to temp area, reply *ok*
- commit* =>
- make change permanent
- abort* =>
- delete temp area

# Consistency Issues in Distributed DB Systems

---

- **Concurrency issues**

- *Problem:* Multiple processes writing and reading simultaneously
  - Locks / deadlock management

- **Autonomy issues**

- *Problem:* Units/departments protective of their systems
  - E.g., administering systems, patching, updating