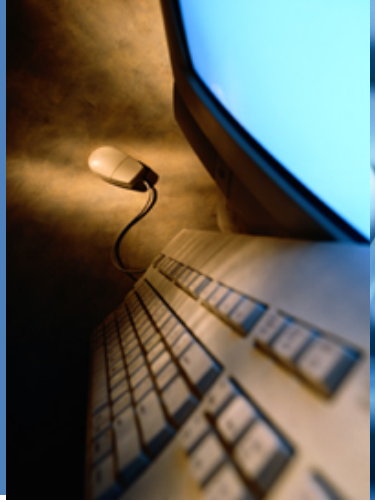
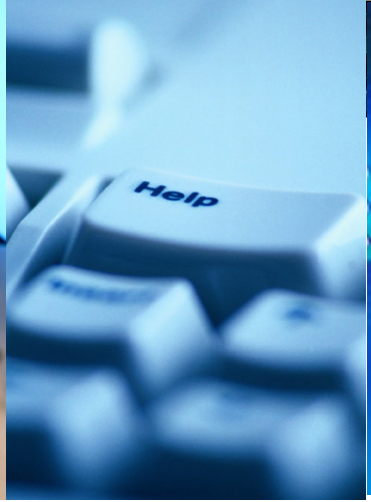
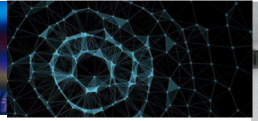
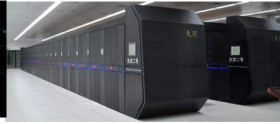



高级分布式系统

Advanced Distributed Systems

陈鹏飞
计算机学院
chenpf7@mail.sysu.edu.cn
主页：https://cse.sysu.edu.cn/teacher/ChenPengfei

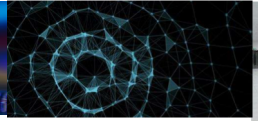
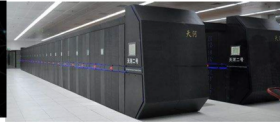




第一讲 — 分布式系统简介

2025.09.8

本课程主要课件来源于Ajay Kshemkalyani and Mukesh Singhal 为课程Distributed Computing: Principles, Algorithms, and Systems 所设计的PPT，由衷表示感谢！！

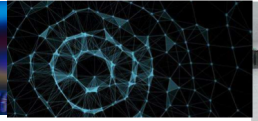




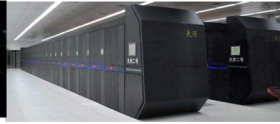
1 课程安排

2 课程考核

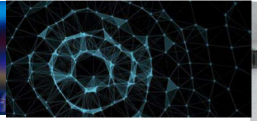
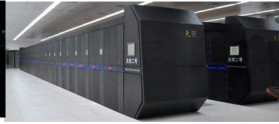
3 参考资料

4 分布式系统简介





1 课程安排

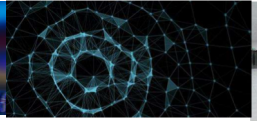
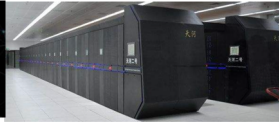
课程内容

➢ 课程描述

本课程主要讲授分布式**系统**的相关原理、算法、设计泛型以及前沿研究，涉及**分布式架构、通信模型、计算模型、逻辑时钟、快照算法、典型分布式算法、消息排序和组通信、终止检测、分布式互斥、分布式共享内存、容错共识、分布式追踪、分布式机器学习、大模型基础设施**等相关原理及技术，同时包括**云计算、分布式通信、分布式系统设计等操作实践**。课程以原理讲授为主，实践为辅，同时需要学生阅读相关领域的文献，并进行讨论和演讲。 此外，该课程还设计了多个实践项目，以强化学生对分布式系统原理和技术的理解与应用。

➢ 课程目的

1、 掌握布式系统的相关概念和原理；
2、 熟悉分布式系统的架构设计；
3、 掌握分布式系统的编程方法；
4、 了解分布式系统的管理方法；
5、 了解分布式系统领域的前沿技术；
6、 激发学生对分布式系统的探索兴趣；

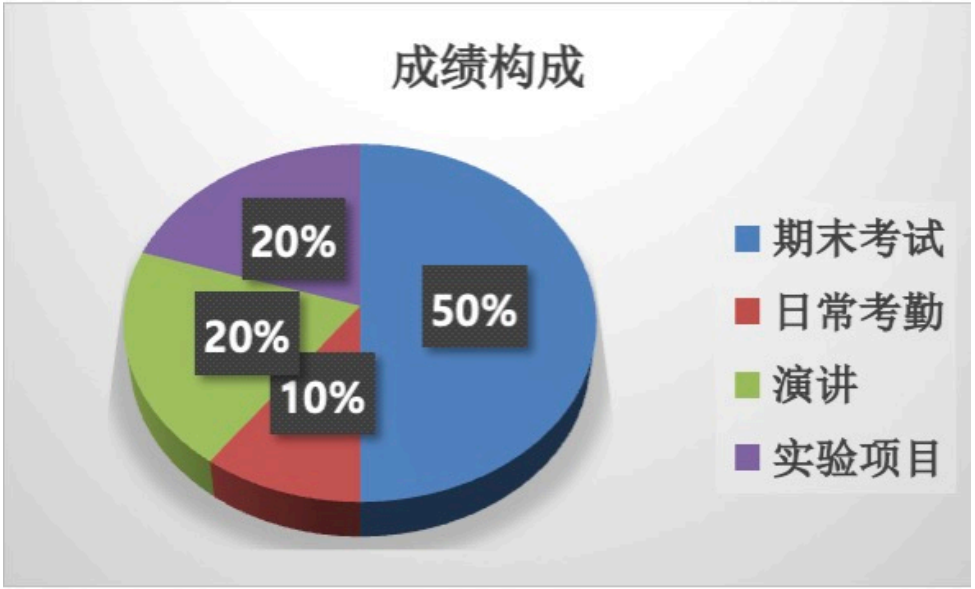




课程安排

周次	课程内容	周次	课程内容
第1周	分布式系统概述	第9周	分布式系统互斥方法
第2周	分布式计算模型	第10周	分布式共享内存
第3周	逻辑时钟	第11周	分布式系统容错
第4周	全局状态与快照	第12周	演讲
第5周	分布式算法	第13周	对等计算及覆盖网络
第6周	消息排序与组通信	第14周	微服务+Kubernetes
第7周	终止检测	第15周	分布式追踪
第8周	演讲	第16周	分布式机器学习
		第17周	大模型基础设施

课程考核标准

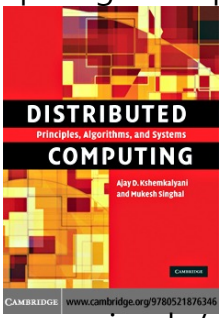
本门课程是计算机科学领域的重点课程，包括理论知识的学习和工程实践。 为了巩固知识、锻炼学生的实操能力，本门课程考核主要包括四部分：**期末考试（开卷）、日常考勤、演讲、实验项目/综述**。

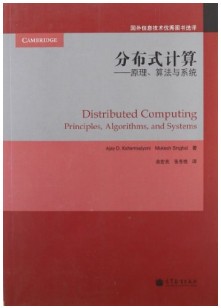


教材及参考资料

- 教材

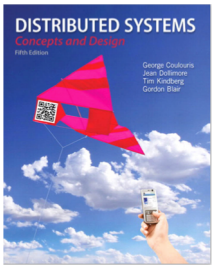
Distributed Computing: Principles, Algorithms, and Systems； 分布式计算翻译版



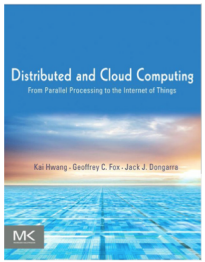


https://www.cs.uic.edu/~ajayk/DCS-Book

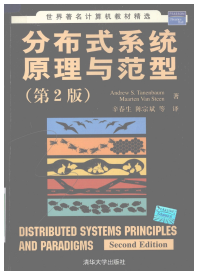
- 参考书



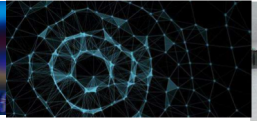
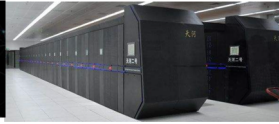
Distributed systems - Concepts and Design (5th)



Distributed and cloud computing



分布式系统原理与范型, 第二版

教材及参考资料

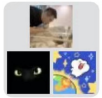
➢ 参考资料

- MIT courses: https://pdos.csail.mit.edu/6.824/schedule.html;
- CMU courses: http://www.andrew.cmu.edu/course/95-702/;
- 清华 courses: http://thu-cmu.cs.tsinghua.edu.cn/curriculum/dscourse/schedule.htm;
- 北京大学 courses: http://net.pku.edu.cn/~course/cs501/2008/schedule.html；
- Cornell courses: http://www.cs.cornell.edu/courses/cs5414/2016fa/;
- NYU courses: http://www.news.cs.nyu.edu/~jinyang/fa17-ds/schedule.html http://www.news.cs.nyu.edu/~jinyang/fa17-ds/schedule.html;
- MIT OpenCourseware: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-824-distributed-computer-systems-engineering-spring-2006/;
- 分布式系统资料： https://cloud.tencent.com/developer/article/1085803；
- Awesome-distributed-systems: https://github.com/zhenlohuang/awesome-distributed-systems;
- 知乎分布式系统：https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/；
- https://backendology.com/2018/09/19/distributed-systems-course-introduction/

教材及参考资料

MIT 6.824 分布式系统：https://www.bilibili.com/video/BV1qk4y197bB?from=search&seid=13146031809677171751

联系方式



群聊: 高级分布式系统 2024

该二维码7天内(9月20日前)有效，重新进入将更新

高级分布式系统微信群

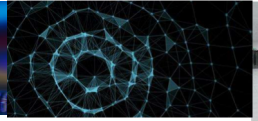
何竞凯 hejk25@mail2.sysu.edu.cn

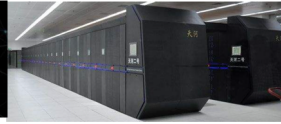
黄海宇 huanghy95@mail2.sysu.edu.cn

办公室：学院楼310

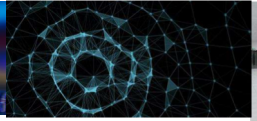
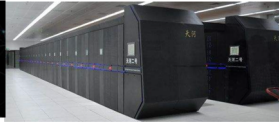
课程网站：

http://course.dds-sysu.tech/course





5 分布式系统简介

分布式系统定义

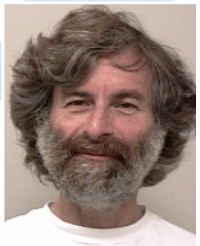
- Autonomous processors communicating over a communication network
- Some characteristics
  - No common physical clock
  - No shared memory
  - Geographical seperation
  - Autonomy and heterogeneity

分布式系统定义

- **《分布式系统原理与范型》定义：**

  分布式系统是若干**独立自主计算机**的集合（硬件），这些计算机对于用户来说像是**单个耦合**系统（软件）。

  物理分布，逻辑集中
  个体独立，整体统一



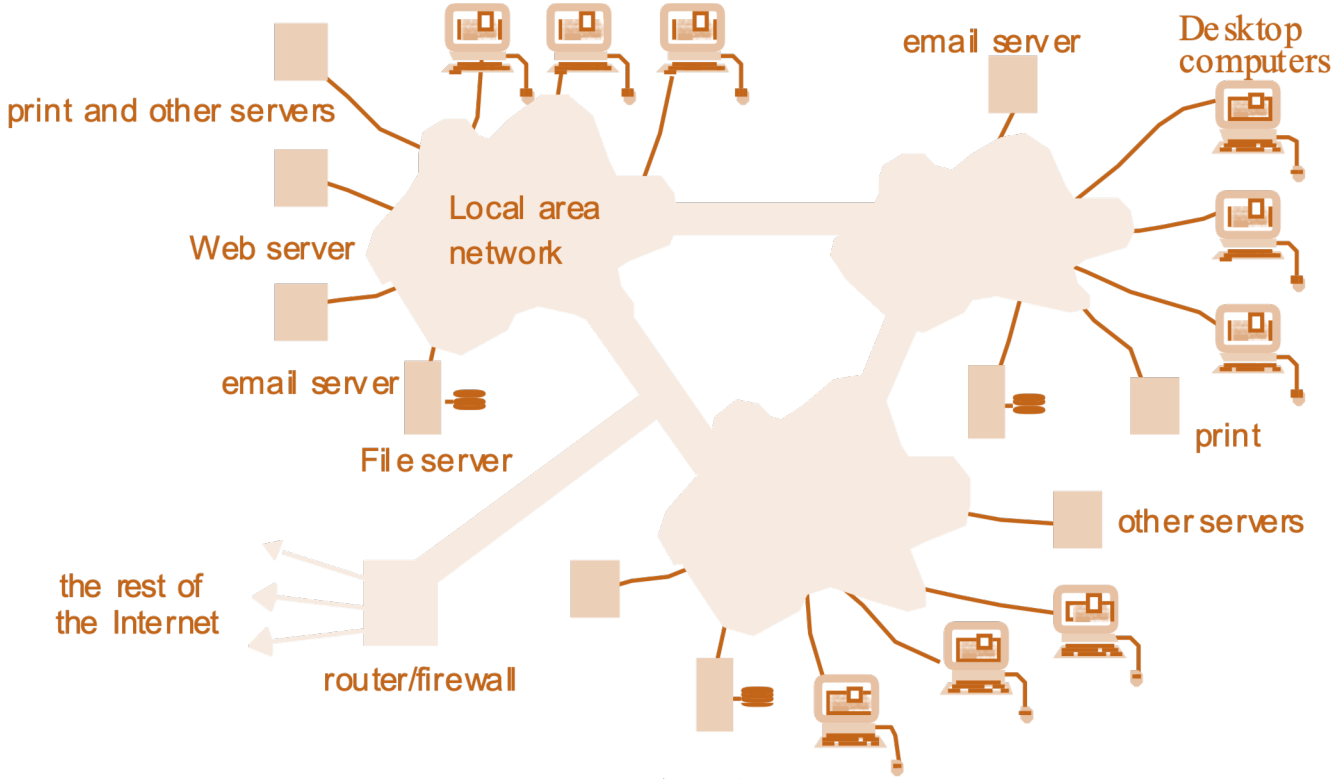
- **Leslie Lamport：** “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”

- **特性：**

  - 自主性：
    计算节点硬件或者软件进程是独立的；

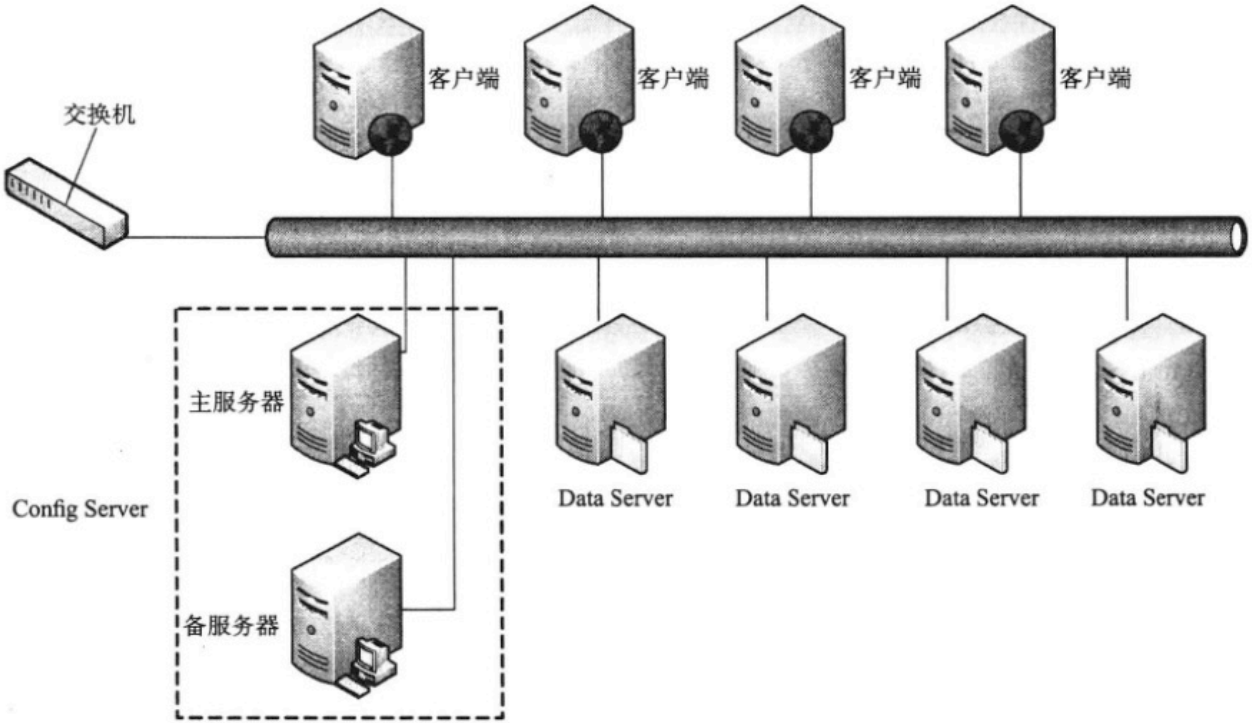
  - 耦合性：
    用户或者应用程序感觉系统是一个系统——节点之间需要相互协作；

分布式系统案例



局域网络

分布式系统案例



Tair分布式数据库架构

分布式系统案例

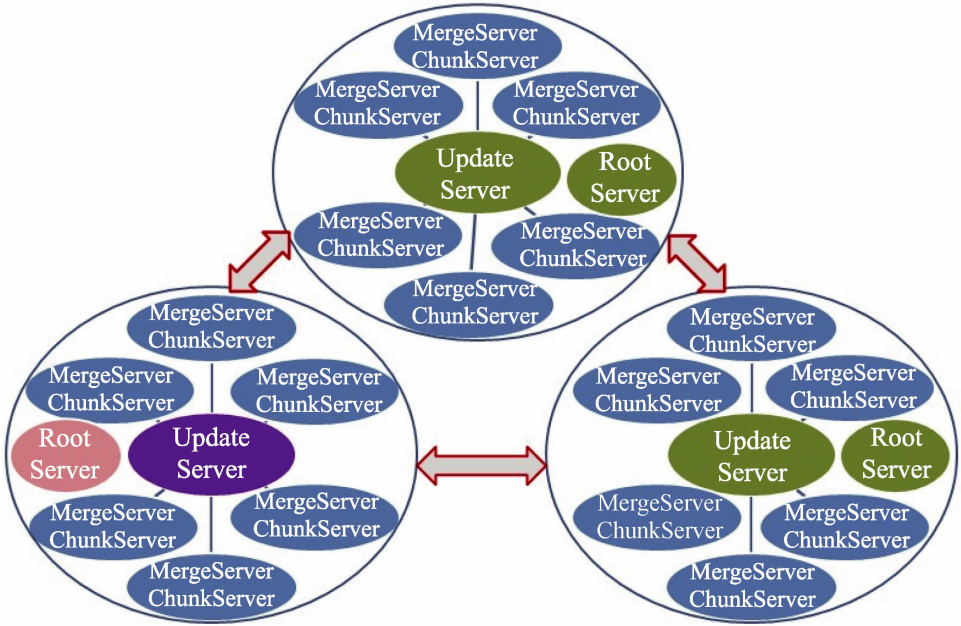
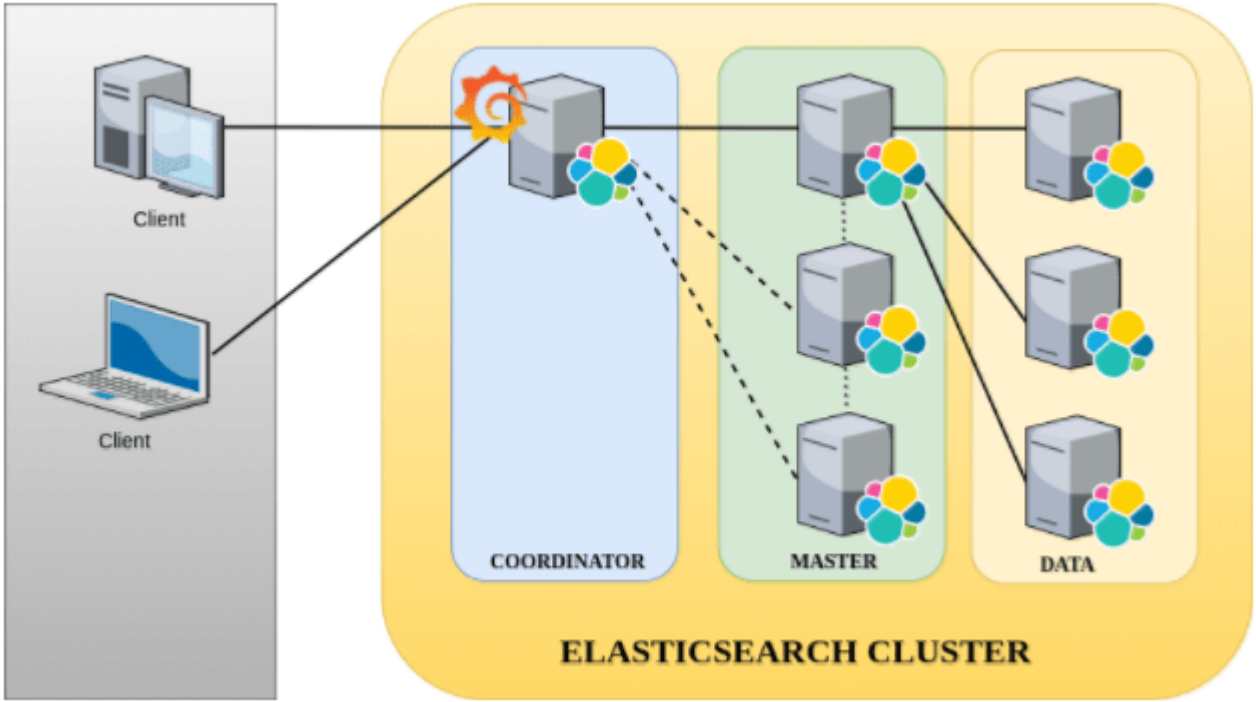


图 4　OceanBase 架构（三机群）

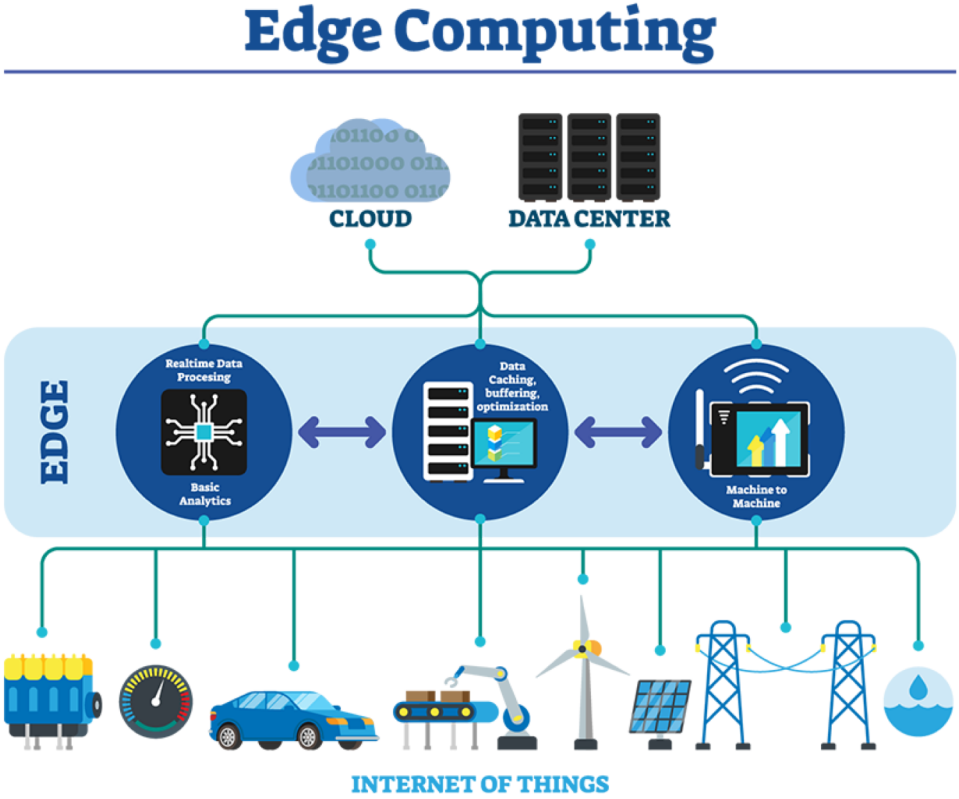
引自：阳振坤，阿里巴巴，《OceanBase关系数据库架构》

分布式系统案例

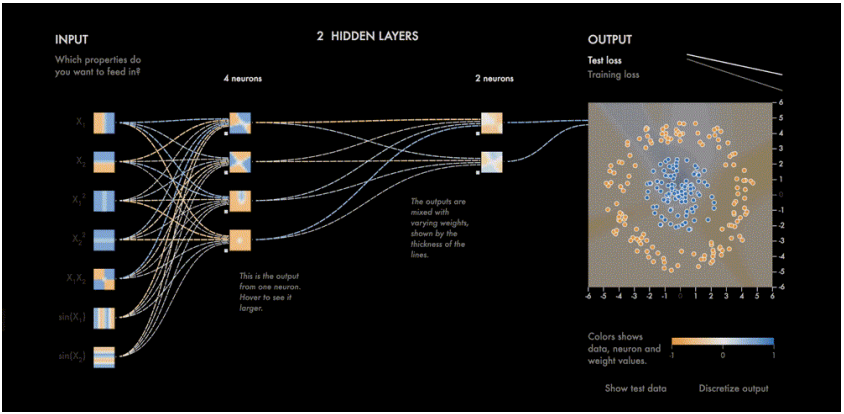


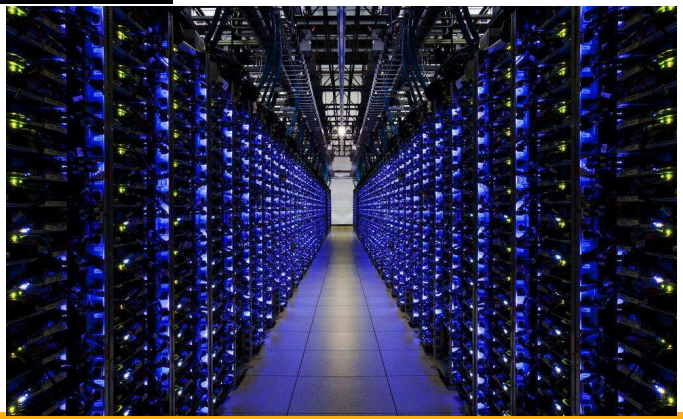
Elasticsearch 搜索集群

分布式系统案例


Edge Computing
CLOUD
DATA CENTER
EDGE
Realtime Data Procesing
Basic Analytics
Data Caching, buffering, optimization
Machine to Machine
INTERNET OF THINGS

分布式系统案例





大规模深度学习系统

分布式系统案例

Ray AIR enables simple scaling of AI workloads.

Data | Train | Tune | Serve | RLlib

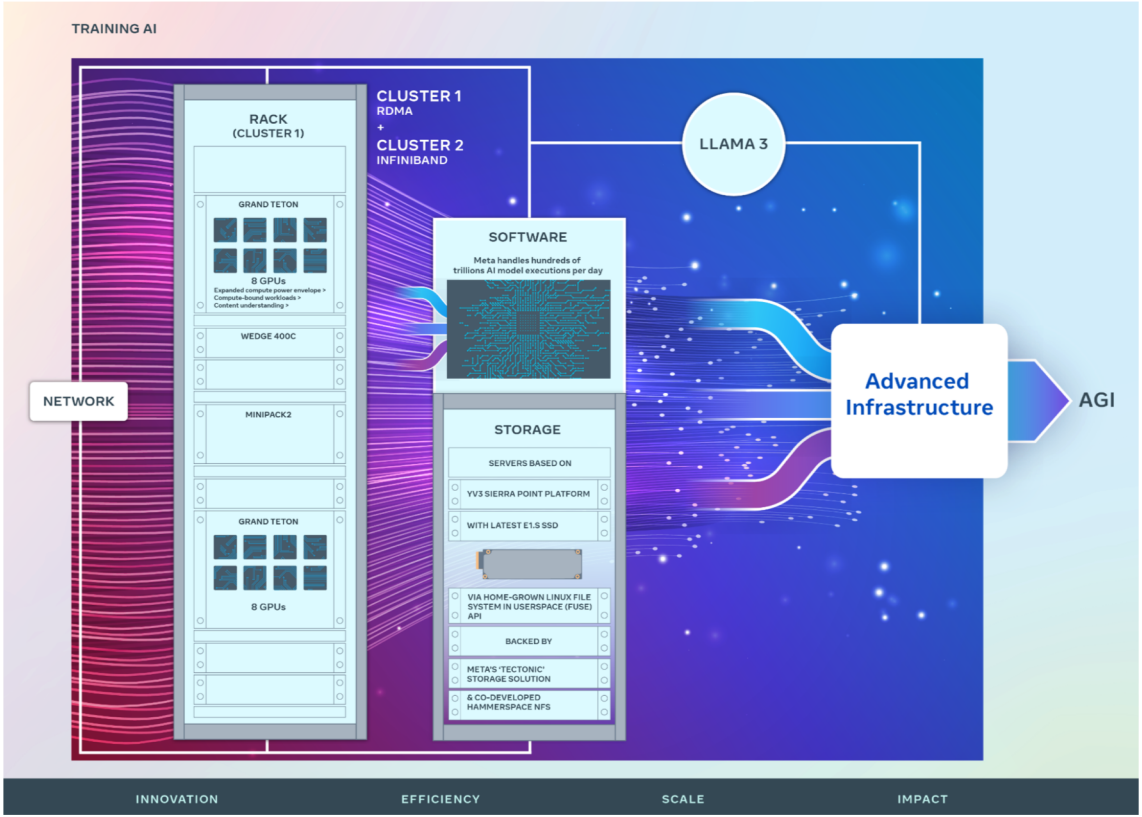
Ray Core enables scalable apps to be built in pure Python.

Custom Applications

Tasks | Actors | Objects

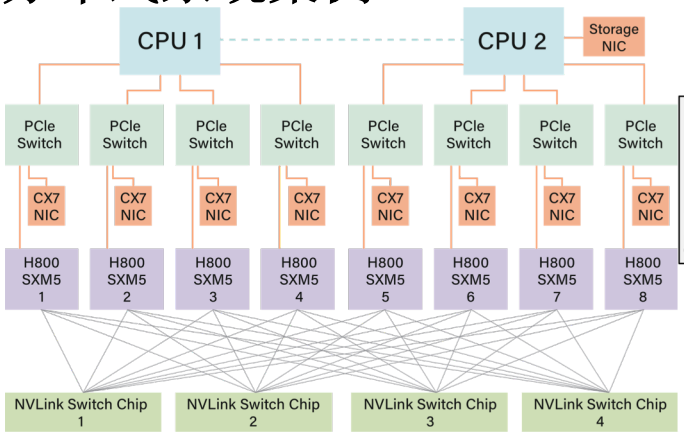
分布式系统案例

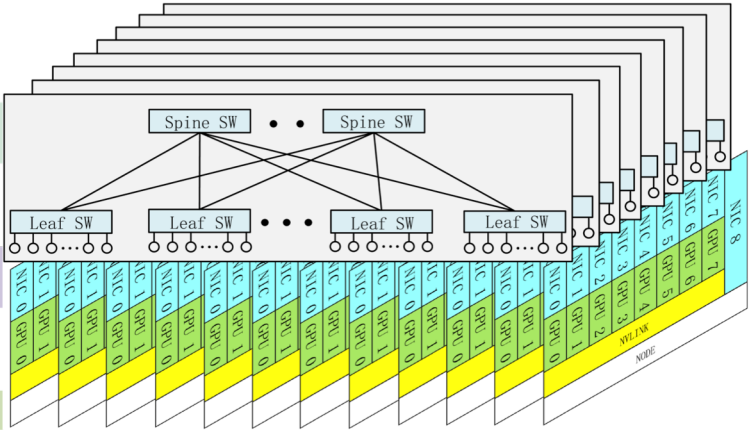
Meta的AI infrastructure



分布式系统案例

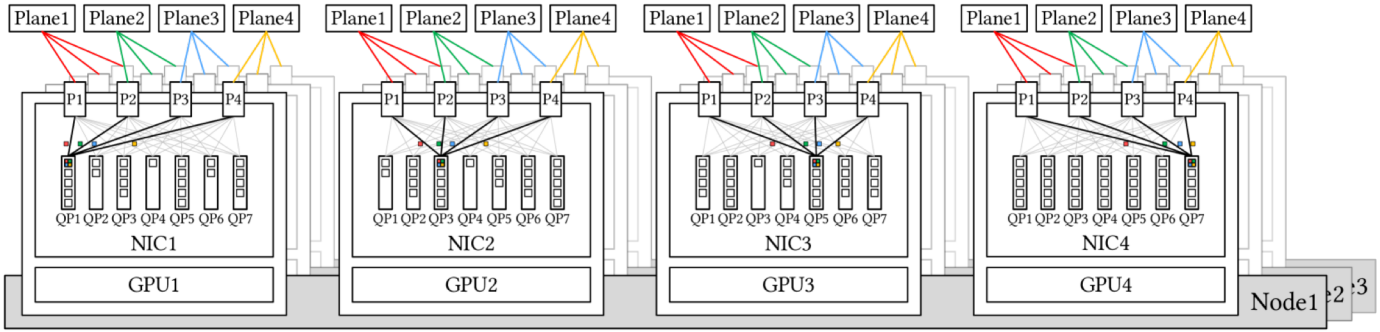
DeepSeek: Insights into DeepSeek-V3: Scaling Challenges and Reflections on Hardware for AI Architectures





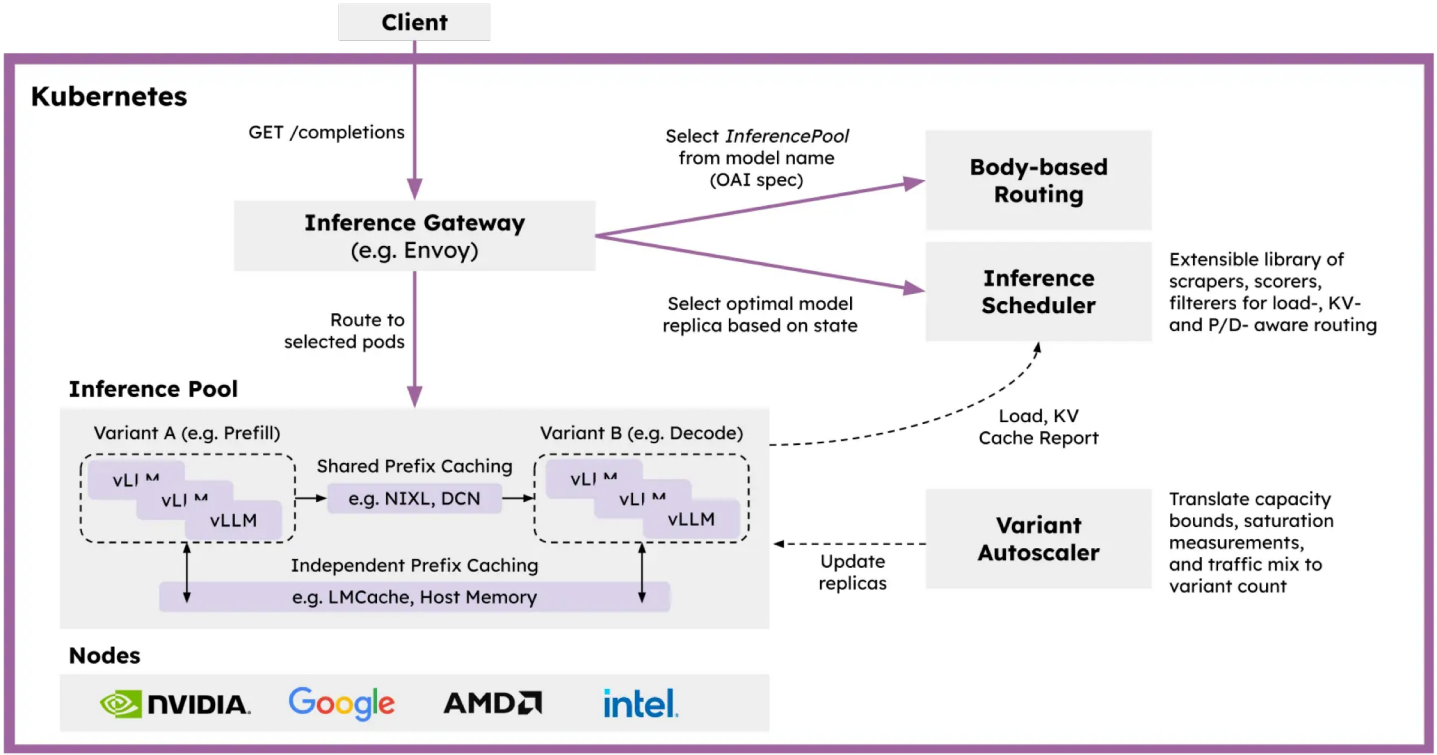
H800节点内互联

节点间连接网络



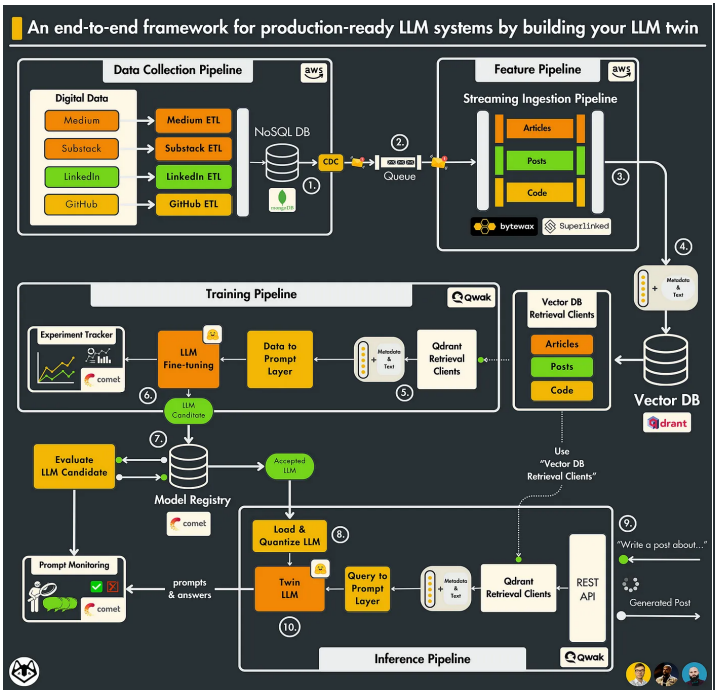
多节点多平面互联结构

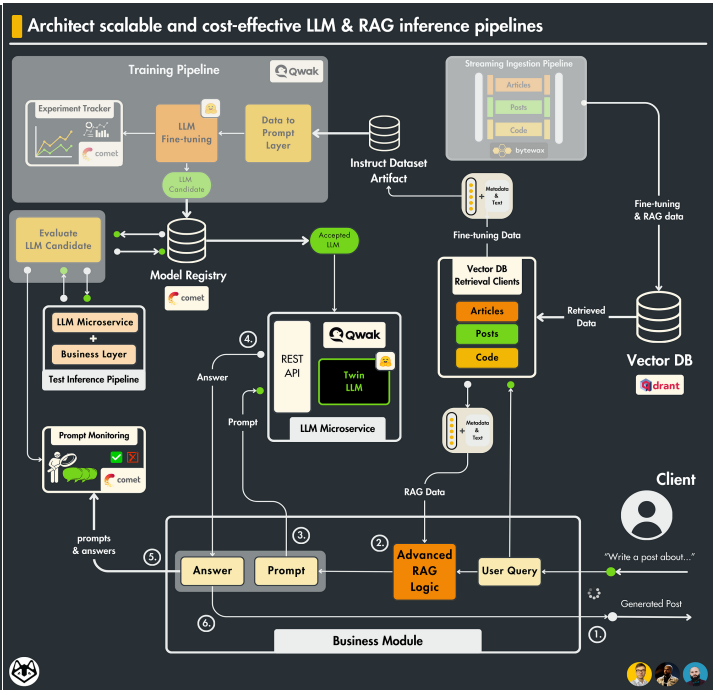
分布式系统案例



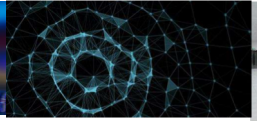
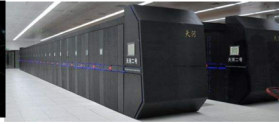
基于Kubernetes的可扩展的大模型推理系统架构

分布式系统案例



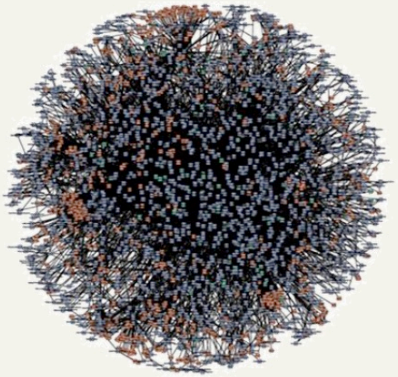


大模型训练和推理具有复杂的软件栈

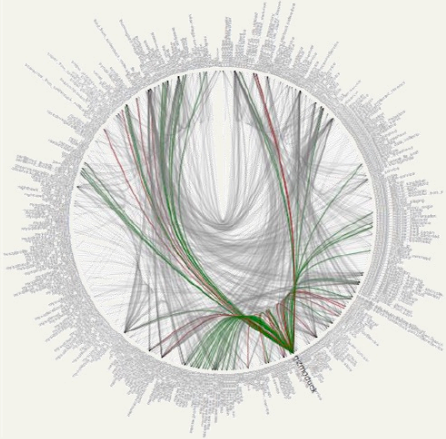
分布式系统案例

amazon.com



NETFLIX

twitter



大规模微服务系统

Relation between Software Components

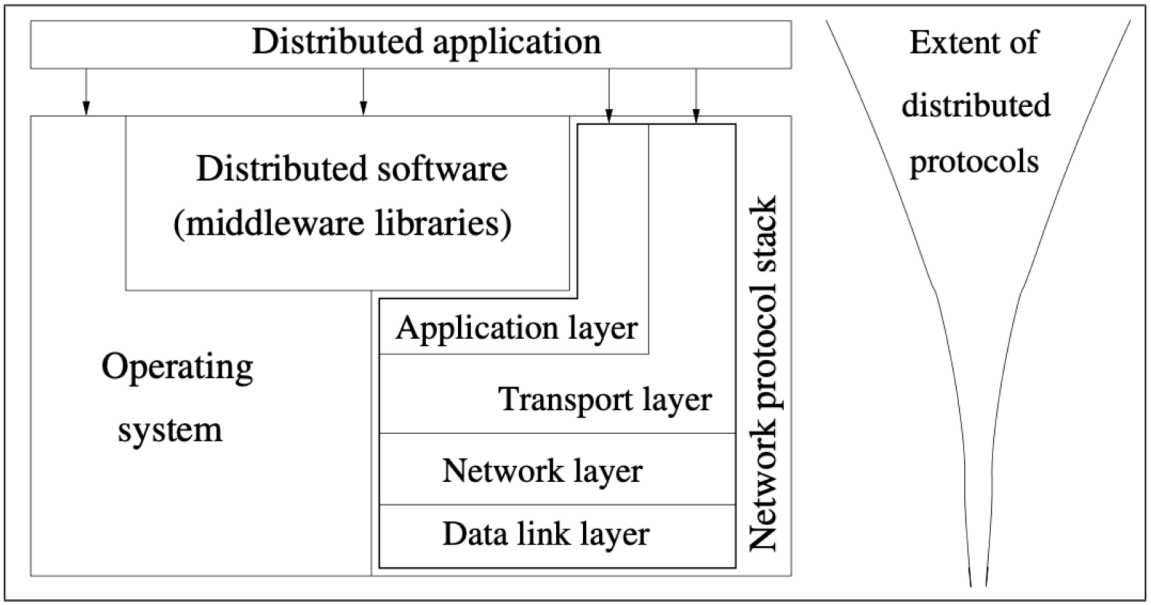
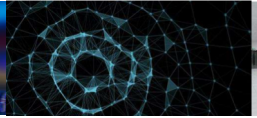
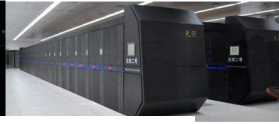
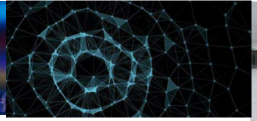
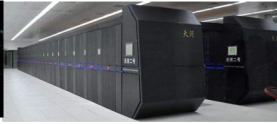


Figure 1.2: Interaction of the software components at each process.

Motivation for Distributed System

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

Parallel Systems

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

UMA vs. NUMA Models

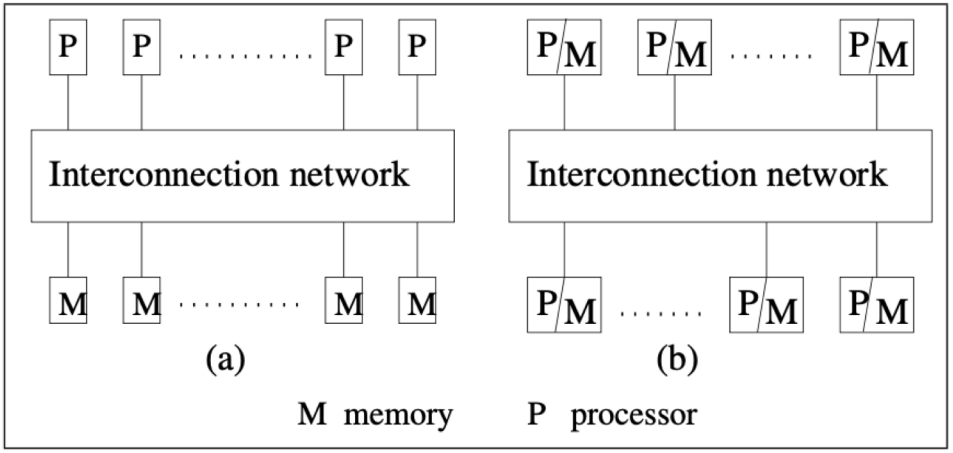


Figure 1.3: Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.

Omega, Butterfly Interconnects

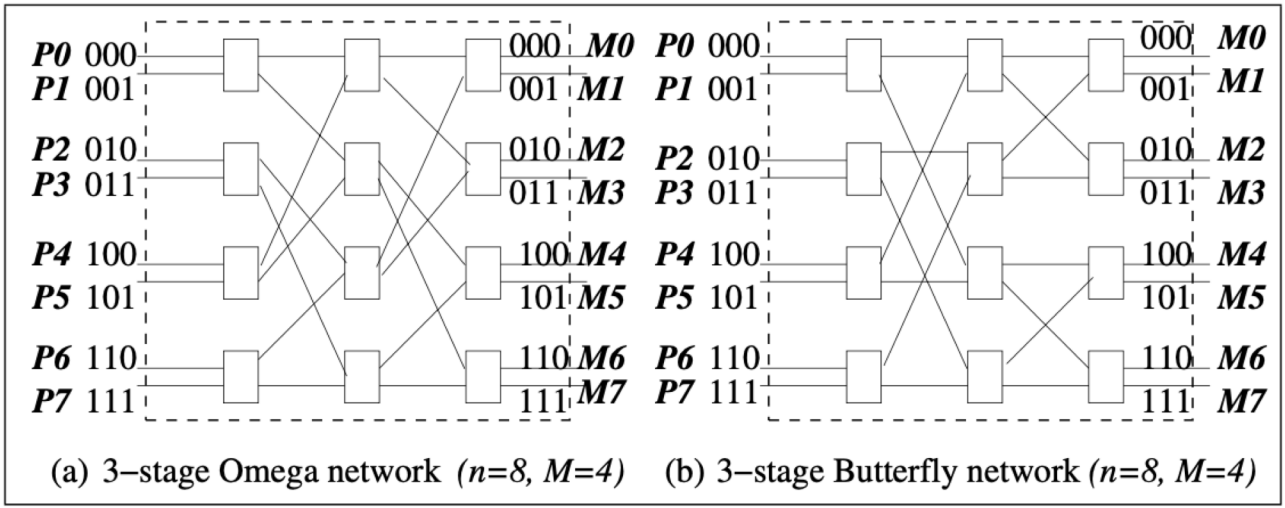
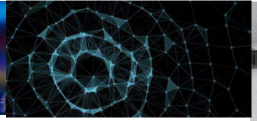
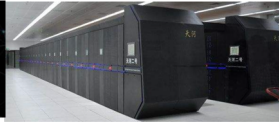


Figure 1.4: Interconnection networks for shared memory multiprocessor systems. (a) Omega network (b) Butterfly network.

Omega Network

- n processors, n memory banks
- $\log n$ stages: with $n/2$ switches of size 2x2 in each stage
- Interconnection function: Output i of a stage connected to input j of next stage:

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

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

Interconnection Topologies for Multiprocesors

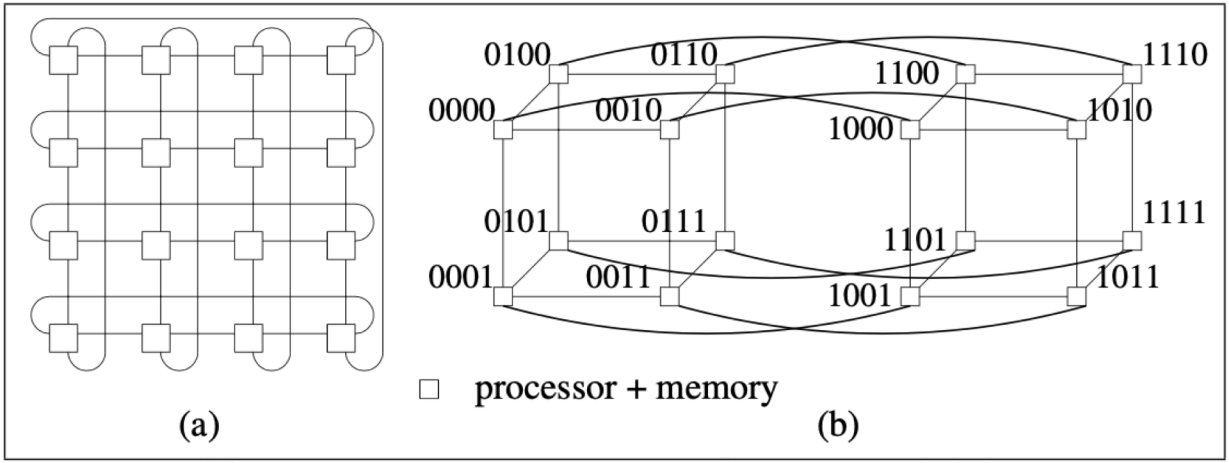
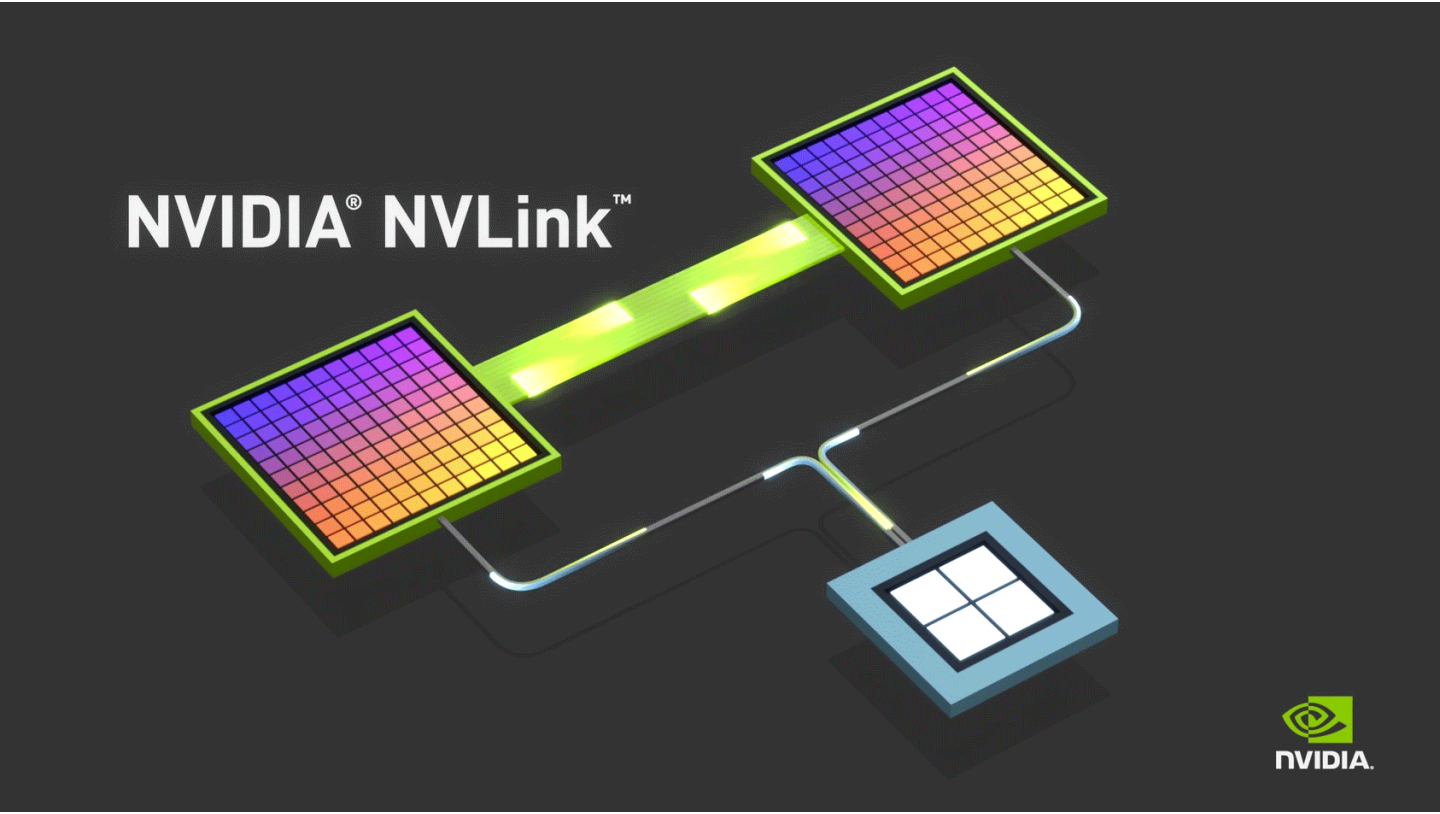
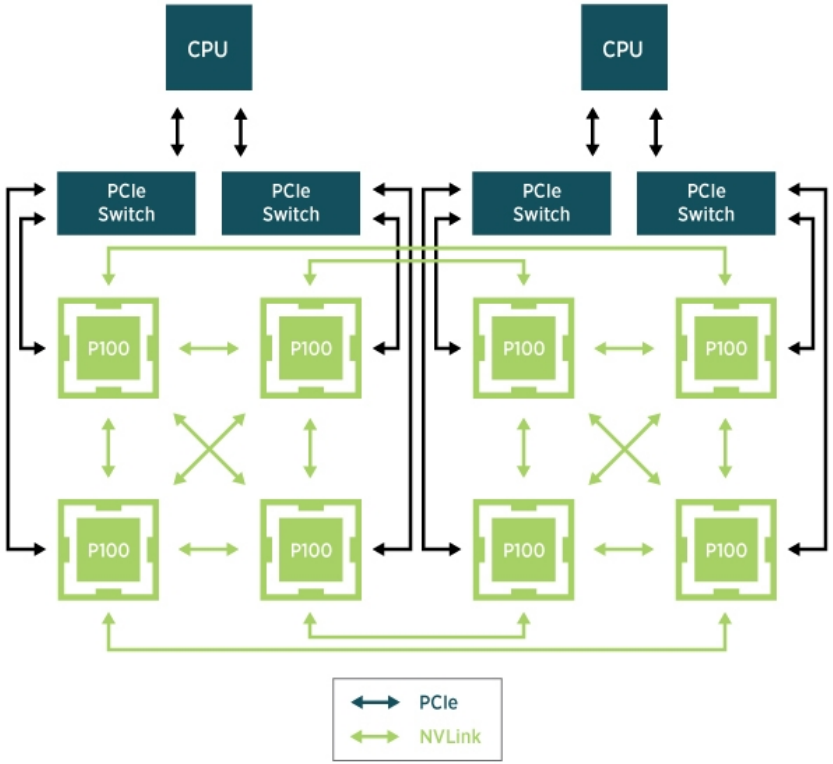


Figure 1.5: (a) 2-D Mesh with wraparound (a.k.a. torus) (b) 3-D hypercube

Nvidia NVlink



Nvidia NVlink



Flynn's Taxonomy

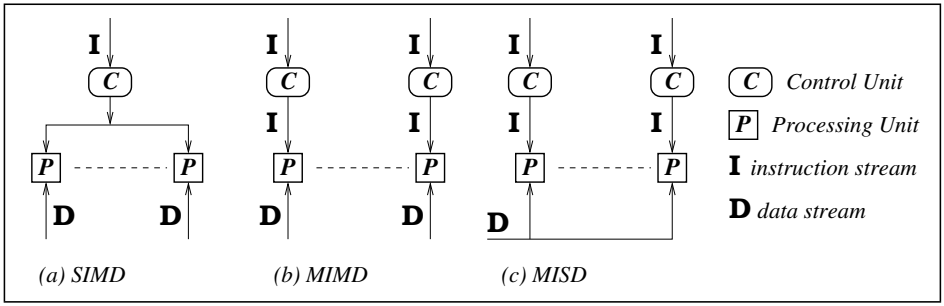
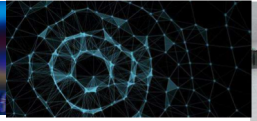
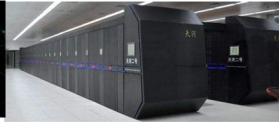


Figure 1.6: SIMD, MISD, and MIMD modes.

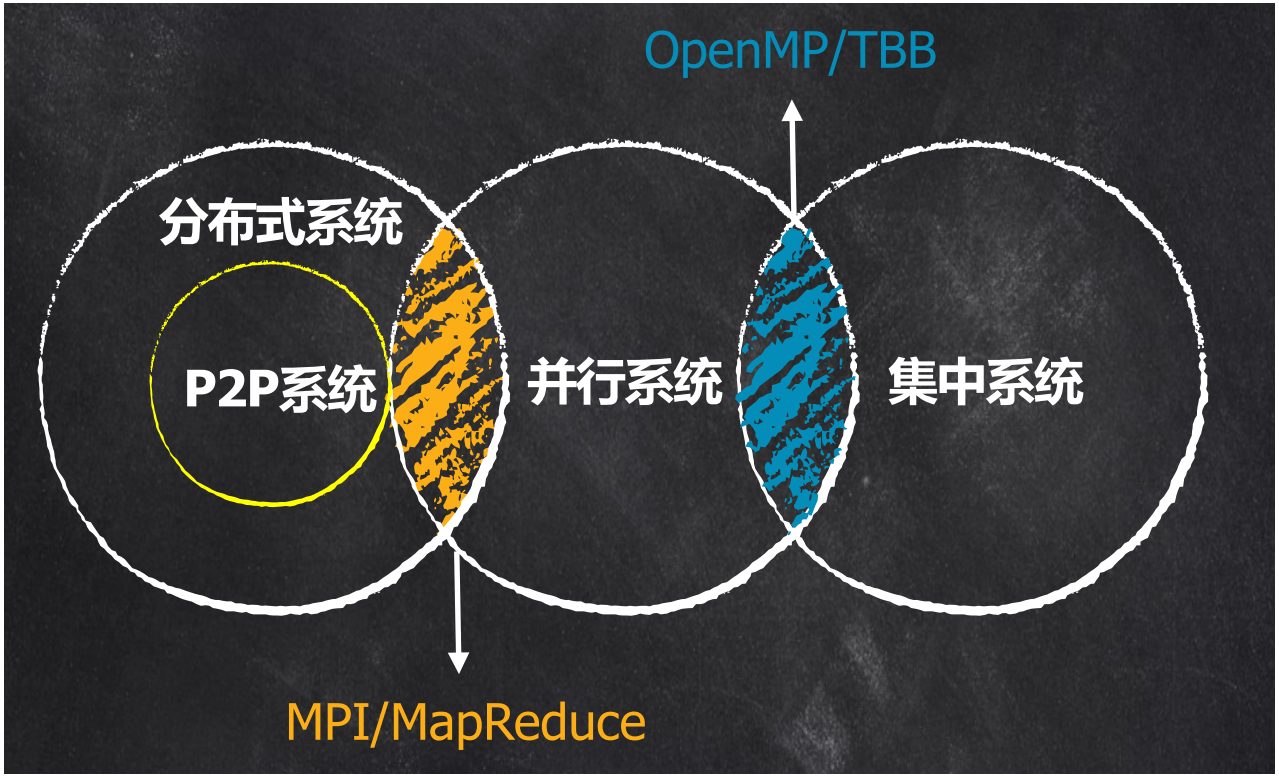
- SISD: Single Instruction Stream Single Data Stream (traditional)
- SIMD: Single Instruction Stream Multiple Data Stream
  - scientific applicaitons, applications on large arrays
  - vector processors, systolic arrays, Pentium/SSE, DSP chips
- MISD: Multiple Instruciton Stream Single Data Stream
  -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

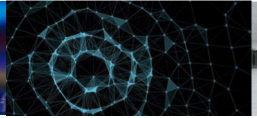
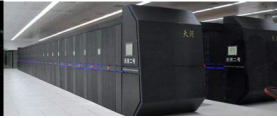




Terminology

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

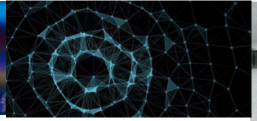
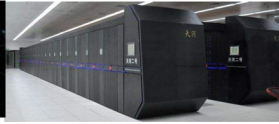
Comparisons



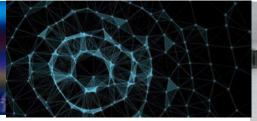
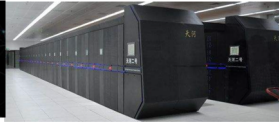
Message-passing vs. Shared Memory

- Emulating MP over SM:
  - Partition shared address space
  - Send/Receive emulated by writing/reading from special mailbox per pair of processes
- Emulating SM over MP:
  - Model each shared object as a process
  - Write to shared object emulated by sending message to owner process for the object
  - Read from shared object emulated by sending query to owner of shared object

Classification of Primitives (1)

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

Classification of Primitives (2)

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

Non-blocking Primitive

Send(X, destination, handle$_k$) //*handle$_k$* is a return parameter
...
...
Wait(*handle*$_1$, *handle*$_2$, ..., *handle$_k$*, ..., *handle$_m$*) //*Wait* always blocks

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

- Return parameter returns a system-generated handle
  - Use later to check for status of completion of call
  - Keep checking (loop or periodically) if handle has been posted
  - Issue Wait(handle1, handle2, ...) call with list of handles
  - Wait call blocks until one of the stipulated handles is posted

Blocking/nonblocking; Synchronous/asynchronous; send/receive primities

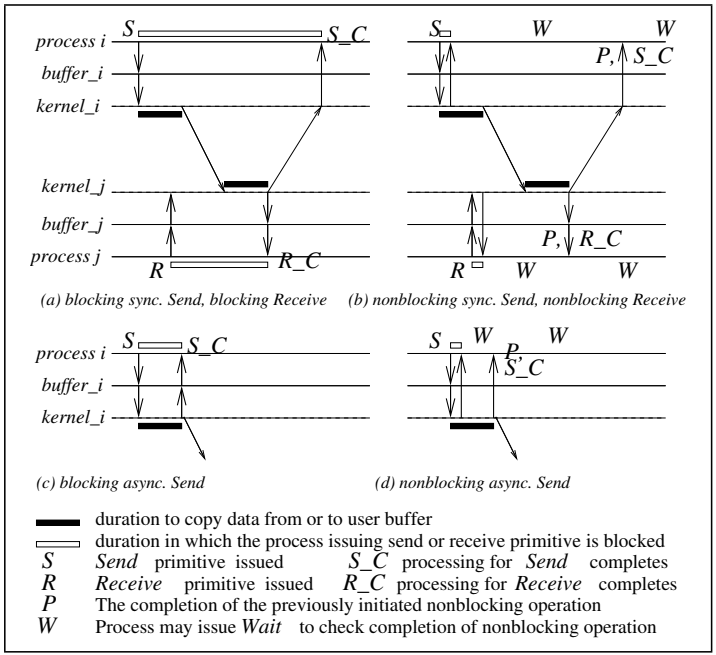
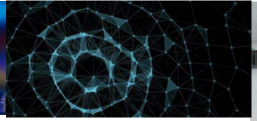
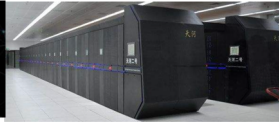


Figure 1.8:Illustration of 4 send and 2 receive primitives

Asynchronous Executions; Mesage-passing System

- An *asynchronous execution* is an execution in which (i) there is no processor synchrony and there is no bound on the drift rate of processor clocks, (ii) message delays (transmission + propagation times) are finite but unbounded, and (iii) there is no upper bound on the time taken by a process to execute a step. An example asynchronous execution with four processes P_0 to P_3 is shown in Figure 1.9. The arrows denote the messages; the tail and head of an arrow mark the *send* and *receive* event for that message, denoted by a circle and vertical line, respectively. Non-communication events, also termed as *internal* events, are shown by shaded circles.

Asynchronous Executions; Mesage-passing System

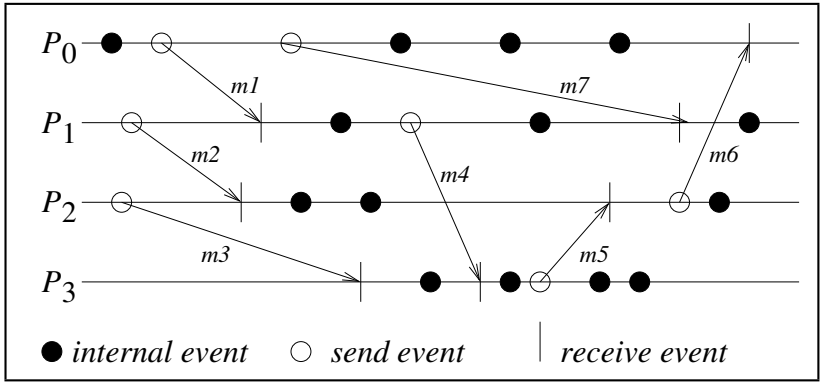
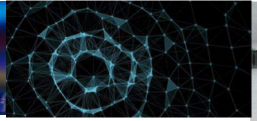
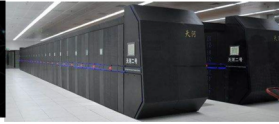


Figure 1.9: Asynchronous execution in a message-passing system

Synchronous Executions: Message-passing System

- A *synchronous execution* is an execution in which (i) processors are synchronized and the clock drift rate between any two processors is bounded, (ii) message delivery (transmission + delivery) times are such that they occur in one logical step or round, and (iii) there is a known upper bound on the time taken by a process to execute a step. An example of a synchronous execution with four processes P_0 to P_3 is shown in Figure 1.10. The arrows denote the messages.

Synchronous Executions: Message-passing System

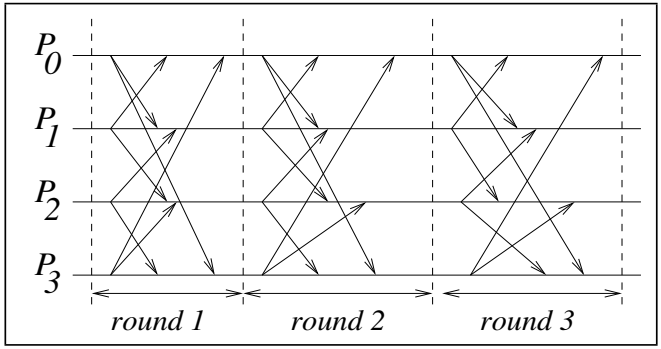
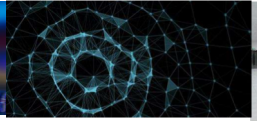
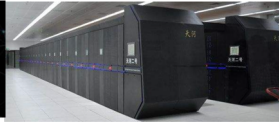


Figure 1.10: Synchronous execution in a message-passing system
In any round/step/phase: $(send \mid internal)^*(receive \mid internal)*$

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

Synchronous vs. Asynchronous Executions (2)

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

System Emulations

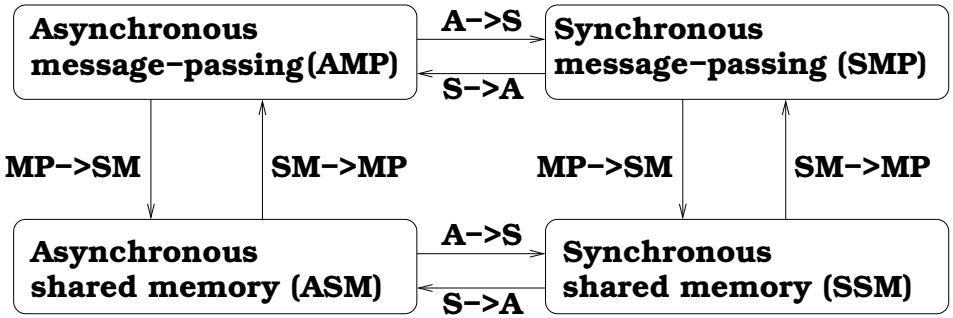
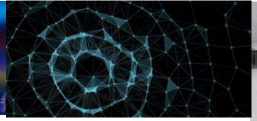
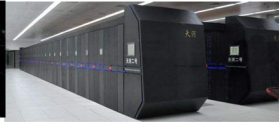


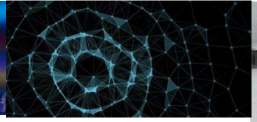
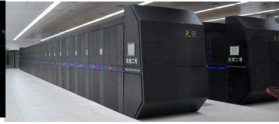
Figure 1.11: Sync $\leftrightarrow$ async, and shared memory $\leftrightarrow$ msg-passing emulations

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

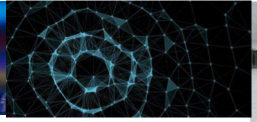
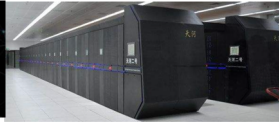
Challenges: System Perspective (1)

- Communication mechanisms: E.g., Remote Procedure Call (RPC), remote object invocation (ROI), message-oriented vs. stream-oriented communication
- Processes: Code migration, process/thread management at clients and servers, design of software and mobile agents
- Naming: Easy to use identifiers needed to locate resources and processes transparently and scalably
- Synchronization
- Data storage and access
  - Schemes for data storage, search, and lookup should be fast and scalable across network
  - Revisit file system design
- Consistency and replication
  - Replication for fast access, scalability, avoid bottlenecks
  - Require consistency management among replicas

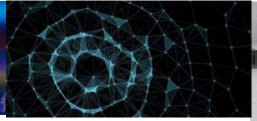
Challenges: System Perspective (2)

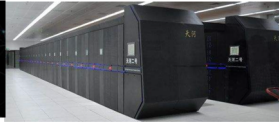
- Fault-tolerance: correct and efficient operation despite link, node, process failures
- Distributed systems security
  - Secure channels, access control, key management (key generation and key distribution), authorization, secure group management
- Scalability and modularity of algorithms, data, services
- Some experimental systems: Globe, Globus, Grid PlanetLab

Challenges: System Perspective (3)

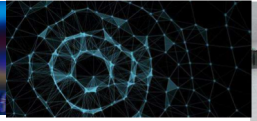
- API for communications, services: ease of use
- Transparency: hiding implementation policies from user
  - Access: hide differences in data rep across systems, provide uniform operations to access resources
  - Location: locations of resources are transparent
  - Migration: relocate resources without renaming
  - Relocation: relocate resources as they are being accessed
  - Replication: hide replication from the users
  - Concurrency: mask the use of shared resources
  - Failure: reliable and fault-tolerant operation

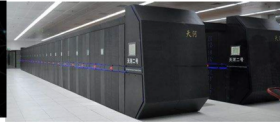




Challenges: Algorithm/Design (1)

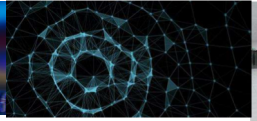
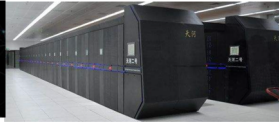
- Useful execution models and frameworks: to reason with and design correct distributed programs
  - Interleaving model
  - Partial order model
  - Input/Output automata
  - Temporal Logic of Actions
- Dynamic distributed graph algorithms and routing algorithms
  - System topology: distributed graph, with only local neighborhood knowledge
  - Graph algorithms: building blocks for group communication, data dissemination, object location
  - Algorithms need to deal with dynamically changing graphs
  - Algorithm efficiency: also impacts resource consumption, latency, traffic, congestion





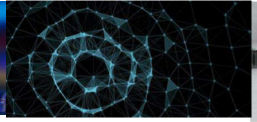
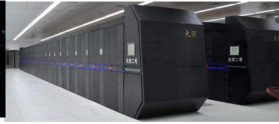
Challenges: Algorithm/Design (2)

- Time and global state
  - 3D space, 1D time
  - Physical time (clock) accuracy
  - Logical time captures inter-process dependencies and tracks relative time progression
  - Global state observation: inherent distributed nature of system
  - Concurrency measures: concurrency depends on program logic, execution speeds within logical threads, communication speeds

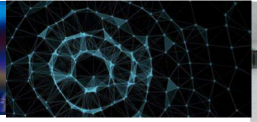
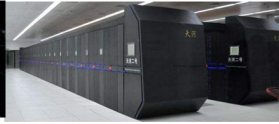
Challenges: Algorithm/Design (3)

- Synchronization/coordination mechanisms
  - Physical clock synchronization: hardware drift needs correction
  - Leader election: select a distinguished process, due to inherent symmetry
  - Mutual exclusion: coordinate access to critical resources
  - Distributed deadlock detection and resolution: need to observe global state; avoid duplicate detection, unnecessary aborts
  - Termination detection: global state of quiescence; no CPU processing and no in-transit messages
  - Garbage collection: Reclaim objects no longer pointed to by any process

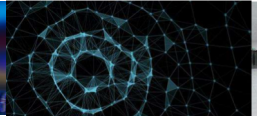
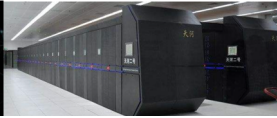
Challenges: Algorithm/Design (4)

- Group communication, multicast, and ordered message delivery
  - Group: processes sharing a context, collaborating
  - Multiple joins, leaves, fails
  - Concurrent sends: semantics of delivery order
- Monitoring distributed events and predicates
  - Predicate: condition on global system state
  - Debugging, environmental sensing, industrial process control, analyzing event streams
- Distributed program design and verification tools
- Debugging distributed programs

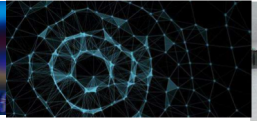
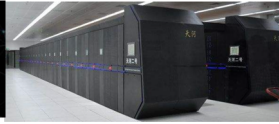
Challenges: Algorithm/Design (5)

- Data replication, consistency models, and caching
  - Fast, scalable access;
  - coordinate replica updates;
  - optimize replica placement
- World Wide Web design: caching, searching, scheduling
  - Global scale distributed system; end-users
  - Read-intensive; prefetching over caching
  - Object search and navigation are resource-intensive
  - User-perceived latency

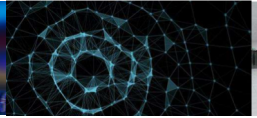
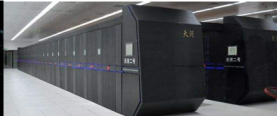
Challenges: Algorithm/Design (6)

- Distributed shared memory abstraction
  - Wait-free algorithm design: process completes execution, irrespective of actions of other processes, i.e., $n-1$ fault-resilience
  - Mutual exclusion
    - ⋆ Bakery algorithm, semaphores, based on atomic hardware primitives, fast algorithms when contention-free access
  - Register constructions
    - ⋆ Revisit assumptions about memory access
    - ⋆ What behavior under concurrent unrestricted access to memory? Foundation for future architectures, decoupled with technology (semiconductor, biocomputing, quantum ...)
  - Consistency models:
    - ⋆ coherence versus access cost trade-off
    - ⋆ Weaker models than strict consistency of uniprocessors

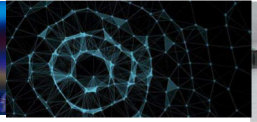
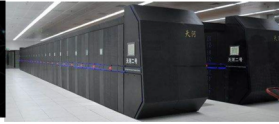
Challenges: Algorithm/Design (7)

- Reliable and fault-tolerant distributed systems
  - Consensus algorithms: processes reach agreement in spite of faults (under various fault models)
  - Replication and replica management
  - Voting and quorum systems
  - Distributed databases, commit: ACID properties
  - Self-stabilizing systems: "illegal" system state changes to "legal" state; requires built-in redundancy
  - Checkpointing and recovery algorithms: roll back and restart from earlier "saved" state
  - Failure detectors:
    - Difficult to distinguish a "slow" process/message from a failed process/ never sent message
    - algorithms that "suspect" a process as having failed and converge on a determination of its up/down status

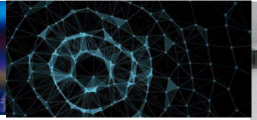
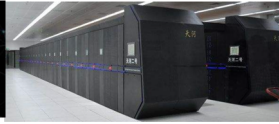
Challenges: Algorithm/Design (8)

- Load balancing: to reduce latency, increase throughput, dynamically. E.g., server farms
  - Computation migration: relocate processes to redistribute workload
  - Data migration: move data, based on access patterns
  - Distributed scheduling: across processors
- Real-time scheduling: difficult without global view, network delays make task harder
- Performance modeling and analysis: Network latency to access resources must be reduced
  - Metrics: theoretical measures for algorithms, practical measures for systems
  - Measurement methodologies and tools

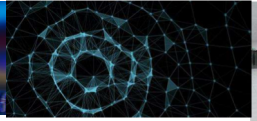
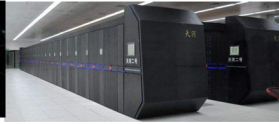
Applications and Emerging Challenges (1)

- Mobile systems
  - Wireless communication: unit disk model; broadcast medium (MAC), power management etc.
  - CS perspective: routing, location management, channel allocation, localization and position estimation, mobility management
  - Base station model (cellular model)
  - Ad-hoc network model (rich in distributed graph theory problems)
- Sensor networks: Processor with electro-mechanical interface
- Ubiquitous or pervasive computing
  - Processors embedded in and seamlessly pervading environment
  - Wireless sensor and actuator mechanisms; self-organizing; network-centric, resource-constrained
  - E.g., intelligent home, smart workplace

Applications and Emerging Challenges (2)

- Peer-to-peer computing
  - No hierarchy; symmetric role; self-organizing; efficient object storage and lookup;scalable; dynamic reconfig
- Publish/subscribe, content distribution
  - Filtering information to extract that of interest
- Distributed agents
  - Processes that move and cooperate to perform specific tasks; coordination, controlling mobility, software design and interfaces
- Distributed data mining
  - Extract patterns/trends of interest
  - Data not available in a single repository

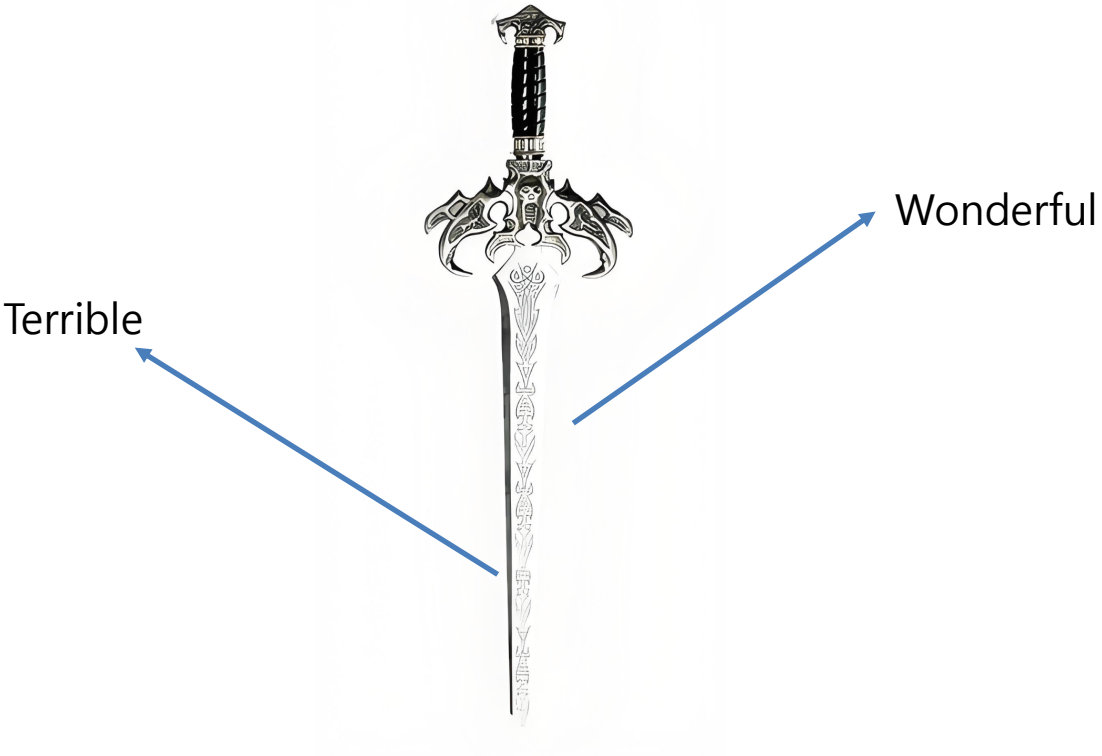




Applications and Emerging Challenges (3)

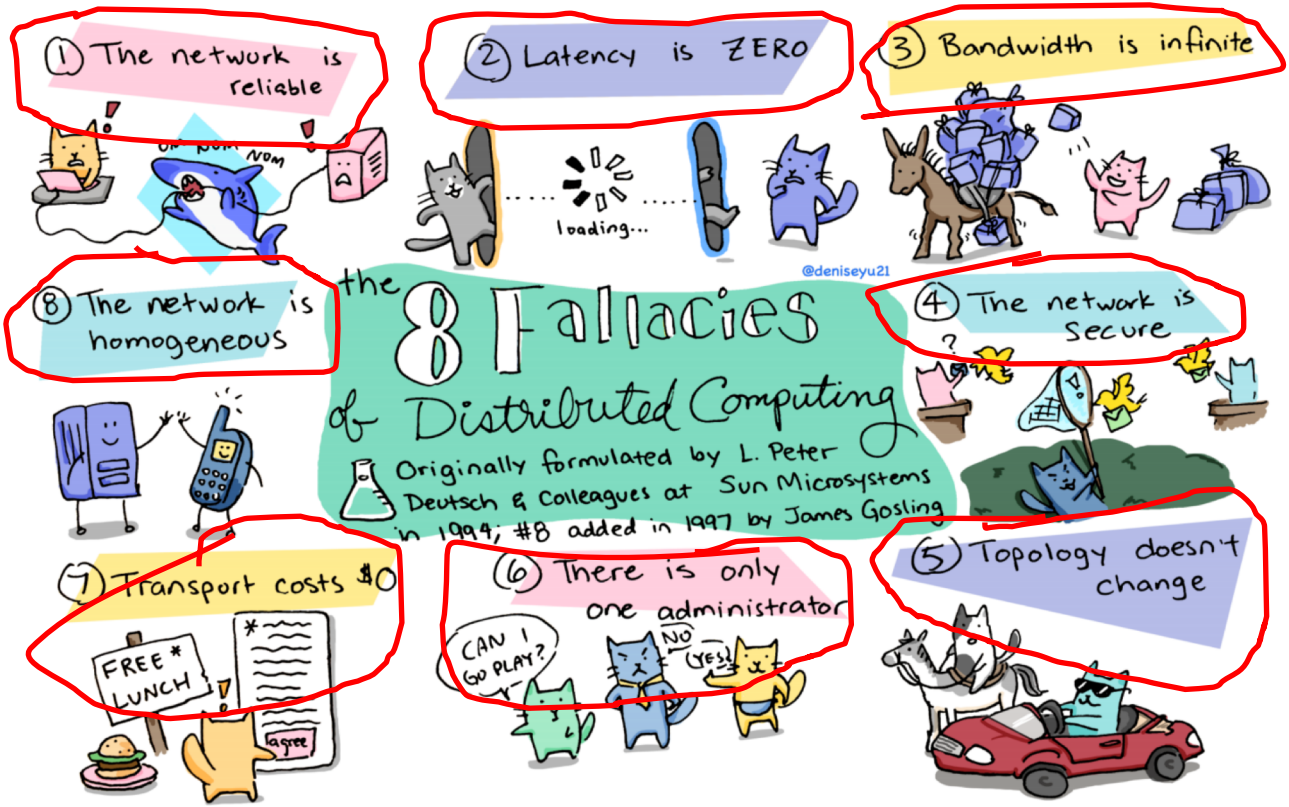
- Grid computing
  - Grid of shared computing resources; use idle CPU cycles
  - Issues: scheduling, QOS guarantees, security of machines and jobs
- Security
  - Confidentiality, authentication, availability in a distributed setting
  - Manage wireless, peer-to-peer, grid environments
    - ★ Issues: e.g., Lack of trust, broadcast media, resource-constrained, lack of structure

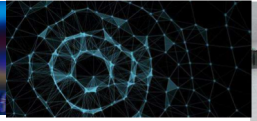
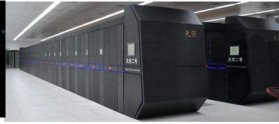
云计算/云原生、无服务器计算、在网计算、算力网络、智算中心、区块链、大模型

Double sides of Distributed systems



分布式系统中的8个谬误



分布式系统的目标

使资源可访问
让用户方便地访问资源

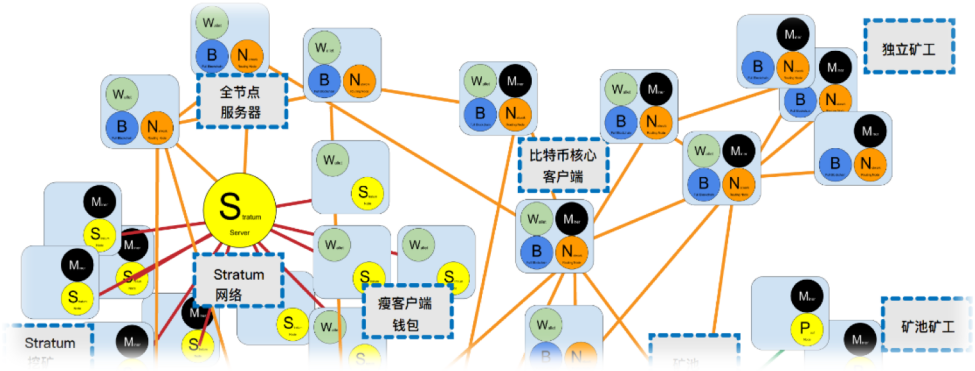
透明性
隐藏资源在网络上的分布

开放性
访问接口的标准化

可扩展性
系统在规模、地域、管理上的可扩展性

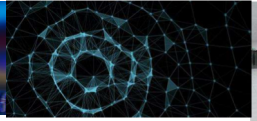
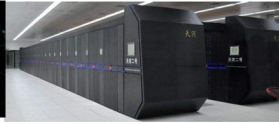
资源访问共享

- 典型样例
  - ✓ 基于云的存储和文件系统；
  - ✓ 基于P2P的流媒体系统；
  - ✓ 共享邮件系统（外包的邮件系统）；
  - ✓ 共享的Web支撑系统（CDN）；
  - ✓ ……



比特币网络

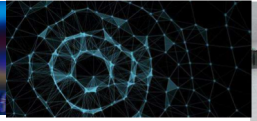
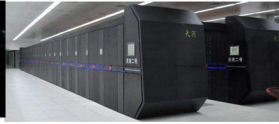
"**The network is the computer**"
—John Gage, then at Sun Microsystems

透明性

- 隐藏进程和资源在多台计算机上分布这一事实；
- 透明的类型

透明性	说明
访问	隐藏数据表示形式的不同以及资源访问方式的不同
位置	隐藏资源所在位置
迁移	异常资源是否移动到另一个位置
重定位	隐藏资源是否在使用过程中移动到另一个位置
复制	隐藏是否对资源进行复制
并发	隐藏资源是否由相互竞争的用户共享
故障	隐藏资源的故障和恢复
持久化	隐藏数据在主存和磁盘这一事实

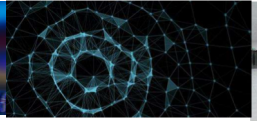
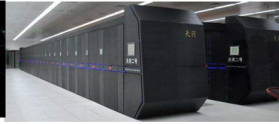




透明度

- 观点

  完全透明性是不可取的也是难以实现的，主要因为：

  - 可能掩盖通信的性能问题；
  - 完全隐藏网络和节点的失效是不可能的；
    - ✓ 不能区分失效和性能变慢的节点；
    - ✓ 不能确定系统失效之前的操作是什么；
  - 完全的透明性可能牺牲性能，暴露系统分布特征；
  - 保证复制节点与主节点的一致性需要时间（一致性问题）；
  - 为了容错需要立即将内存修改的内容同步到磁盘上；

透明度

- **暴露系统的分布特征有一定使用场景**
  - 利用基于位置的服务（如：找到附近的朋友）
  - 当与不同时区的用户交互时；
  - 当让用户理解系统发生了什么时，如当一台服务器不响应时，报告失效；
- **结论**

  分布式透明性是一个较好的属性，但是需要区别对待。

完全的透明性是不可取的！

分布式系统的开放性

- 什么是分布式系统的开放性？

分布式系统的开放性指：系统根据一系列准则来提供服务，这些准则描述了所提供服务的语法和语义 （标准化）。

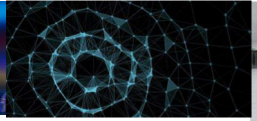
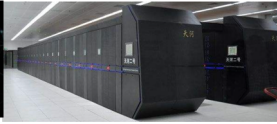
- 讨论分布式系统开放性的那些方面？

  - 系统应该具有良好定义的接口；
  - 系统应该容易实现互操作性；
  - 系统应该支持可移植性；
  - 系统应该容易实现可扩展性；

开放性之策略与机制

重点： 策略与机制分离

- **实现开放性：策略**
  - 需要为客户端的缓冲数据设置什么级别的一致性?
  - 我们允许下载的程序执行什么操作?
  - 当出现网络带宽波动的时候如何调整QoS需求？
  - 通信的安全水平设置多高？
- **实现开放性：机制**
  - 允许动态设定缓冲策略；
  - 支持为移动代码设置不同的信任级别；
  - 为每个数据流提供可调整的QoS参数；
  - 提供不同的加密算法；

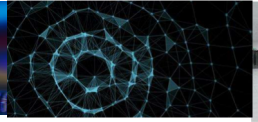
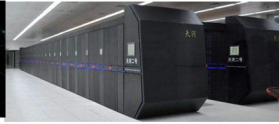
策略与机制严格分离

- **观察发现**

策略和机制之间分离的越严格，越需要设计合适的机制，这样会导致出现很多配置参数和复杂的管理。

- **策略和机制之间的平衡**

硬编码某些策略可以简化管理，减少复杂性，但是会导致灵活性降低。没有放之四海而皆准的方法。

分布式系统的可扩展性

➢ 观察

现代分布式系统的很多开发人员经常使用“可扩展性”，但不清楚为什么要进行扩展。

➢ 三个方面的扩展性

- 规模可扩展性：用户数量和进程数量增加；
- 地理可扩展性：节点之间的最大物理位置；
- 管理可扩展性：管理域的数量；

➢ 观点

- 大部分系统关心的是规模可扩展性；
- 解决方案：多个服务器独立并行运行；
- 地理可扩展性和管理可扩展性仍然充满挑战；

分布式系统面临的挑战

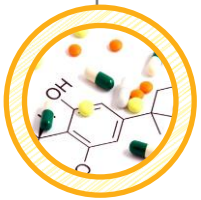
分布式系统挑战

系统设计

—正确的接口设计和抽象；
—如何换分功能和可扩展性；

一致性

—如何一致性共享数据；



容错

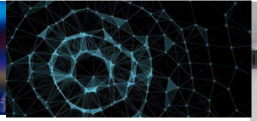
—如何保障系统在出现失效情况下正常运行；

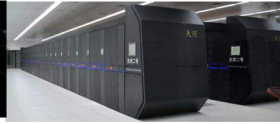
不同的部署场景

—集群；
—广域分布；
—传感网络；

实现

—如何最大化并行；
—性能瓶颈是什么；
—如何平衡负载；





谢谢