分布式计算

04-可扩展性Scalability

Weixiong Rao 饶卫雄
Tongji University 同济大学软件学院
2023 秋季
wxrao@tongji.edu.cn

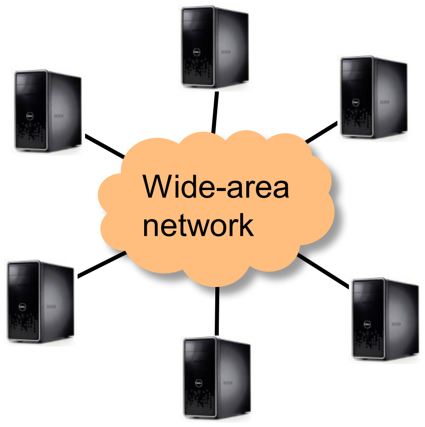
同济大学软件学院
School of Software Engineering, Tongji University

扩张规模导致系统的复杂性
Scale increases complexity

单核服务器
Single-core
machine

多核服务器
Multicore
server

集群
Cluster



复杂的分布式系统
数据中心(网络) Large-scale distributed system

更多困难

并发性
True
concurrency

网络
消息传递
失效模型
(故障节点 ...)
Network
Message passing
More failure modes
(faulty nodes, ...)

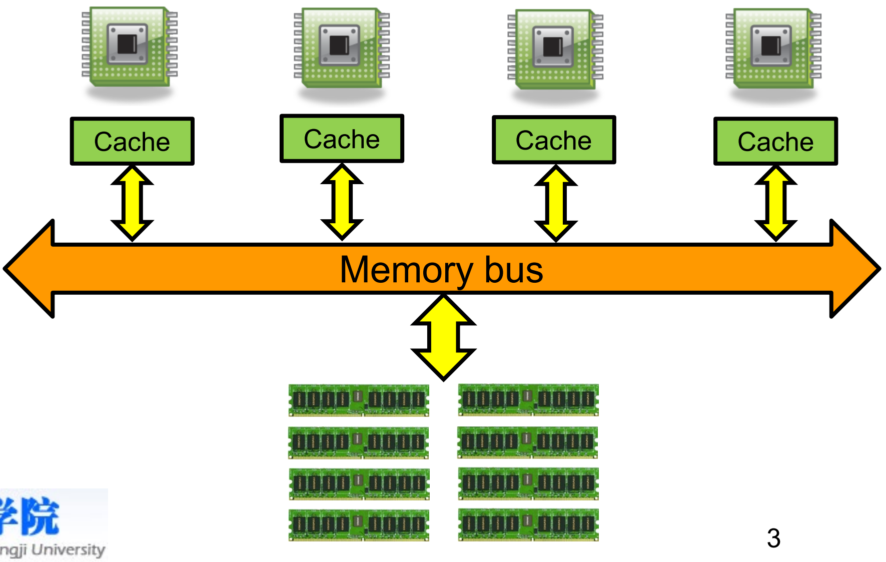
广域网
更多更复杂的失效模型
各种系统设计方法和定律…

Wide-area network
Even more failure
modes
Incentives, laws, ...

对称多处理器Symmetric Multiprocessing (SMP)

- 多核CPUs可以访问共享内存For now, assume we have multiple cores that can access the same shared memory
  - 假设: 每个核均可以访问任意一个字节数据, 相同的访问速度 (即每个字节，无论是读写操作，均花费同样的时间)
  - 注意：当然不是所有的机器是符合这样的假定，其他的模型将此后讨论

- Any core can access any byte; speed is uniform (no byte takes longer to read or write than any other)
- Not all machines are like that -- other models discussed later



同济大学软件学院
School of Software Engineering, Tongji University

内容目录

- 并行编程模型及存在的问题Parallel programming and its challenges
  - 并行化、可扩展性, Amdahl定律Parallelization and scalability, Amdahl's law
  - 同步，一致性
  - 互斥、锁、锁机制所产生的问题
  - 体系结构: SMP, NUMA, Shared-nothing
- 分布式编程及难点
  - 网络分隔故障、CAP定理
  - 故障,失效, 及应对方法

NEXT

什么是可扩展性？What is scalability?

- 一个计算机系统是可扩展的A system is scalable
  - 如果该系统可以灵活适应需求的增加和减少it can easily adapt to increased (or reduced) demand
  - 举例: 存储系统可以在开始阶段有10TB，还能够通过增加更多的节点使得存储能力提高到PB级别 Example: A storage system might start with a capacity of just 10TB but can grow to many PB by adding more nodes
  - 可扩展性总归还是存在一些瓶颈Scalability is usually limited by some sort of bottleneck
- 通常情况下, 可扩展性还意味... Often, scalability also means...
  - 相当规模的运行能力the ability to operate at a very large scale
  - 增长能力迅速the ability to grow efficiently
    - 比如: 4倍节点 → ~4倍的计算能力提升(不仅仅是2倍的提升!) Example: 4x as many nodes → ~4x capacity (not just 2x!)

可扩展性,效率及速度
Scalability, efficiency, and speed

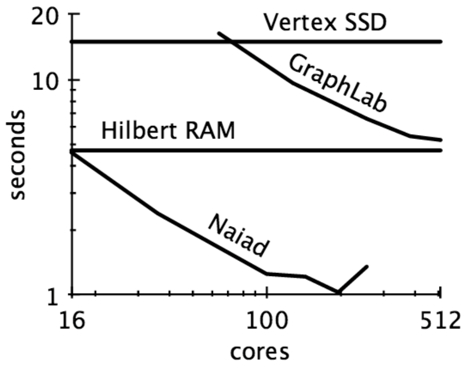
- 需要处理1TB数据Suppose you need to process 1TB of data
- 有资金去买100台机器
  You have money for up to 100 machines

土豪：买买买

- 三个可选的系统You can choose among three systems: :
  - 系统 #1: 在一台机器上跑: Runs on a single machine
  - 系统 #2: 系统的吞吐量按O(N)增加, 其中N是机器数量, N<=50: Throughput grows with O(N), where N is the number of machines, up to N=50
  - 系统 #3:系统的吞吐量按O(sqrt(N))增加, 其中N是机器数量 Throughput grows with O(sqrt(N)), where N is the number of machines
- 你会选择哪个系统呢？Which system should you use?

除了可扩展性 – 还有成本呢？ Scalability – but at what COST?

Data set	twitter_rv	uk-2007-05
顶点数 Nodes	41,652,230	105,896,555
边数 Edges	1,468,365,182	3,738,733,648
文件大小 Size	5.76 GB	14.72 GB



系统System	核数 cores	Twitter 数据处理时间	uk-2007-05数据处理时间
GraphChi	2	3160s	6972s
Stratosphere	16	2250s	-
X-Stream	16	1488s	-
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

F. McSherry, M. Isard, D. Murray: "Scalability! But at what COST?", HotOS 2015

- 实验结果: 跑多个可扩展的数据处理系统Experiment: Run several scalable data processing systems on some large data sets
  - twitter_rv & uk-2007-05: Crawls of Twitter and part of the UK web graph
  - 20 次迭代的 **PageRank 算法** 20 iterations of PageRank (more about this later)
- 结论? What should we conclude from this?

如何实现可扩展性算法?
How to build systems that scale

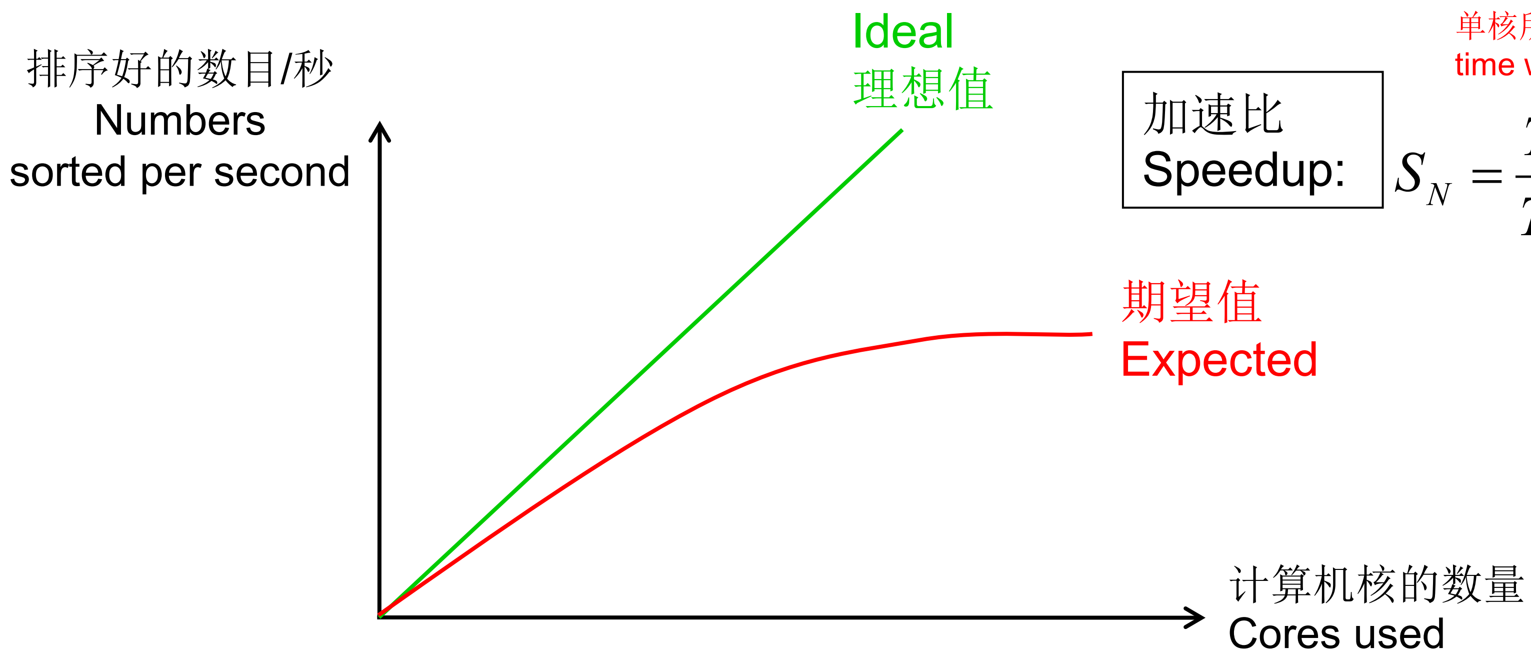
```
void bubblesort(int nums[]) {
  boolean done = false;
  while (!done) {
    done = true;
    for (int i=1; i<nums.length; i++) {
      if (nums[i-1] > nums[i]) {
        swap(nums[i-1], nums[i]);
        done = false;
      }
    }
  }
}
```

```
int[] mergesort(int nums[]) {
  int numPieces = 10;
  int pieces[][] = split(nums, numPieces);
  for (int i=0; i<numPieces; i++)
    sort(pieces[i]);
  return merge(pieces);
}
```

并行化处理
Can be done in parallel!

- 冒泡排序在单核机器效果很好The left algorithm works fine on one core
- 在多核机器上, 冒泡排序能否跑得更快呢?Can we make it faster on multiple cores?
  - 有难度 – 很难找到可以在其他的核上可以并行运行的代码 Difficult - need to find something for the other cores to do
- 其他的一些排序算法(并归排序)容易实现并行化There are other sorting algorithms where this is much easier
  - 不是所有的算法可以取得相同程度的并行化Not all algorithms are equally parallelizable
  - 可以取得可扩展性但是没有并行化机制的方式? Can you have scalability without parallelism?

实际情况下的可扩展性
Scalability in practice



加速比
Speedup:

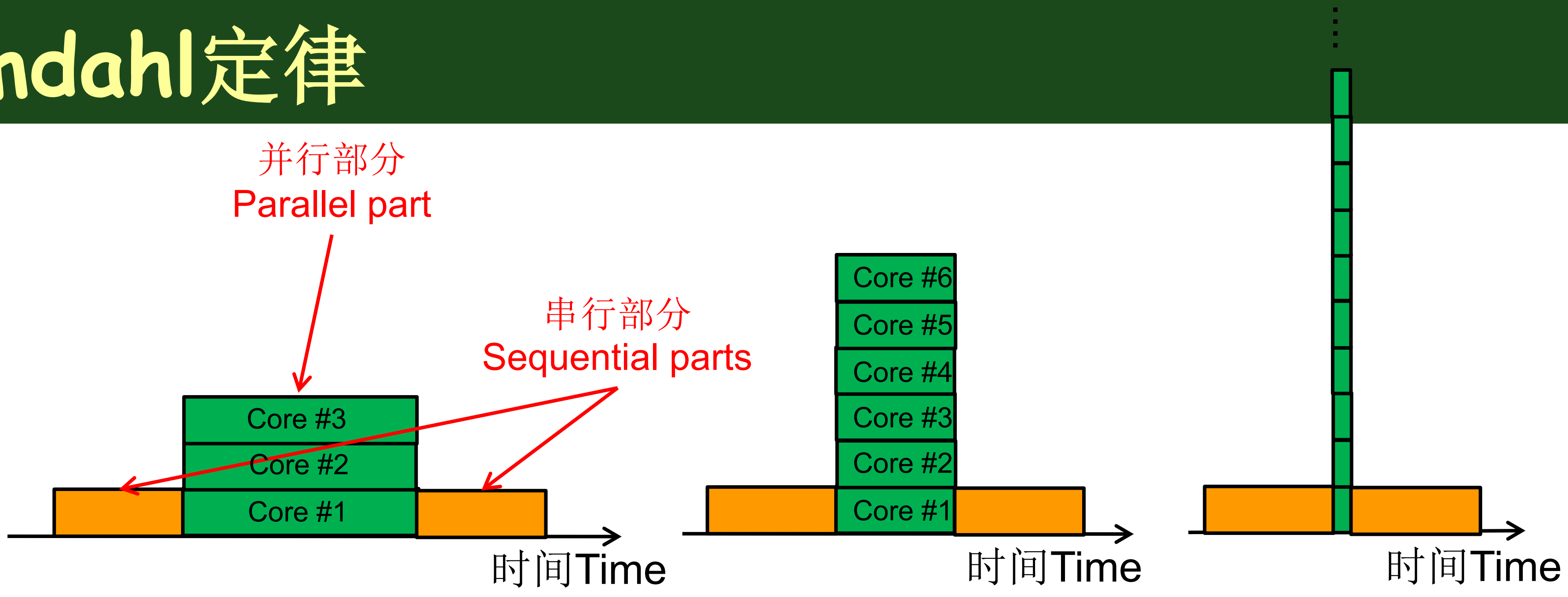
$$S_N = \frac{T_1}{T_n}$$

单核所需的时间 Completion time with one core

多核所需的时间 Completion time with n cores

- 在增加计算机核的数量情况下,计算速度是否亦随之增长? If we increase the number of processors, will the speed also increase?
  - **是的**, 但是在多数情况下只会增长到**一个特定的点** Yes, but (in almost all cases) only up to a point
  - Why?

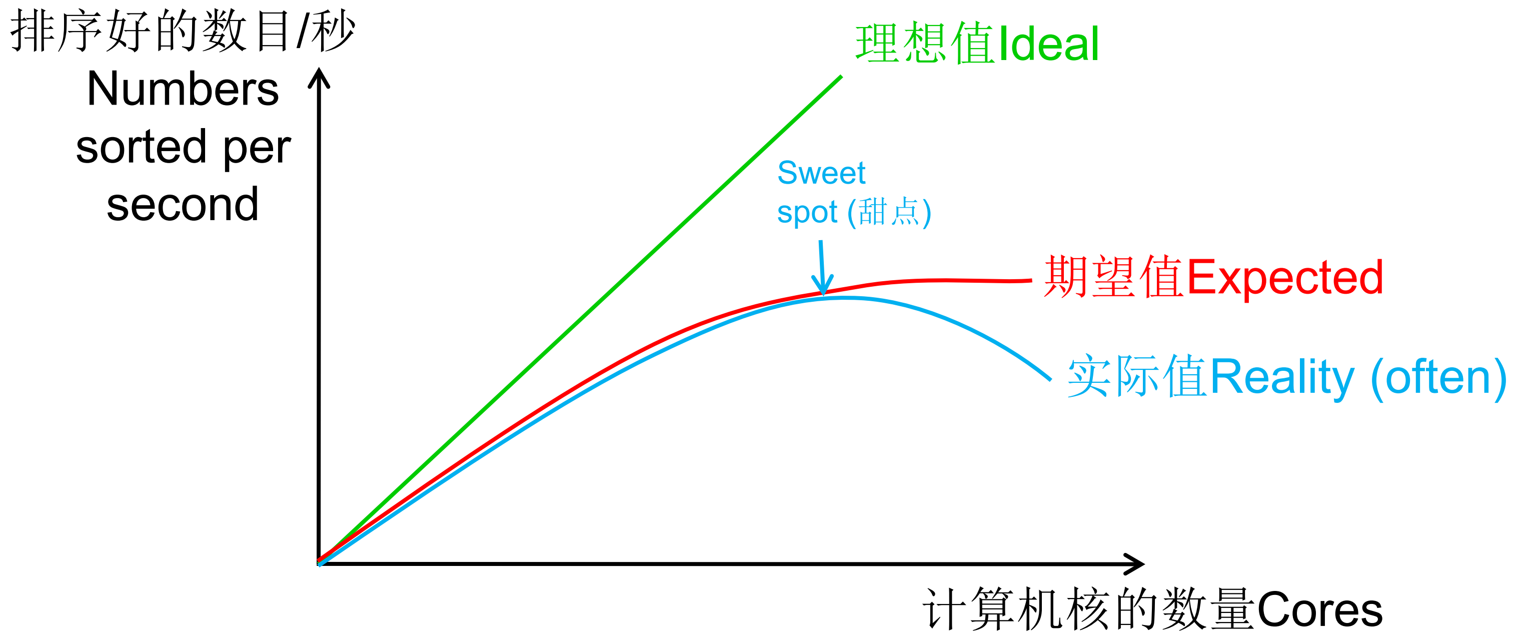
Amdahl定律



- 通常情况, 一个计算机算法不是每一个部分均可以进行并行化 Usually, not all parts of the algorithm can be parallelized
- 假定**f** 是该算法可以并行化的比例, **S_{part}** 是对应的加速比Let f be the fraction of the algorithm that can be parallelized, and let S_{part} be the corresponding speedup
- Then

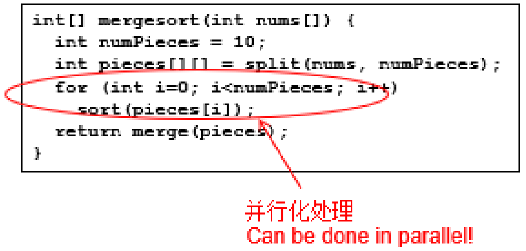
$$S_{overall} = \frac{1}{(1-f)+\frac{f}{S_{part}}}$$

并行度越高就越好吗？
Is more parallelism always better?



- 超过**甜点**继续增加并行度会导致性能下降! Increasing parallelism beyond a certain point can cause performance to decrease! Why?
- 算法中的顺序计算部分所需时间取决于核的数量Time for serial parts can depend on #cores
  - 举例: 需要给每个核发送消息，从而告诉每个核所执行的任务Example: Need to send a message to each core to tell it what to do

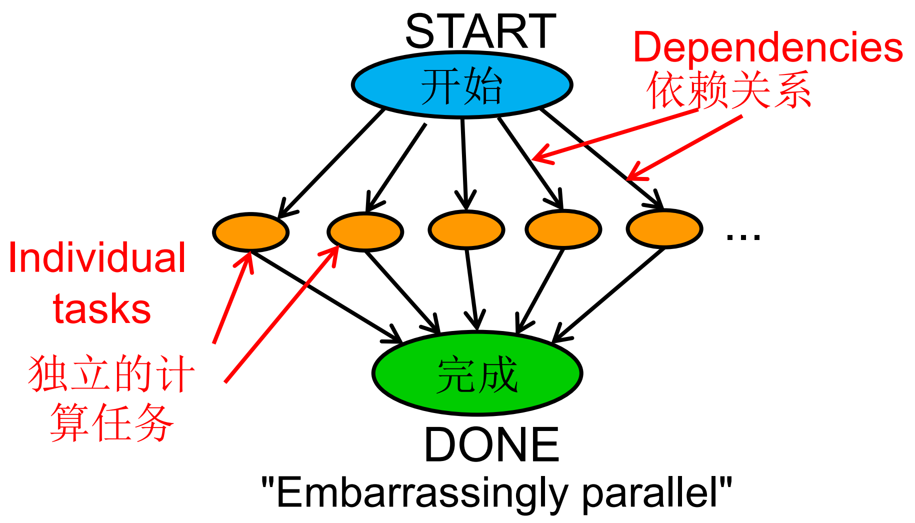
并行粒度Granularity

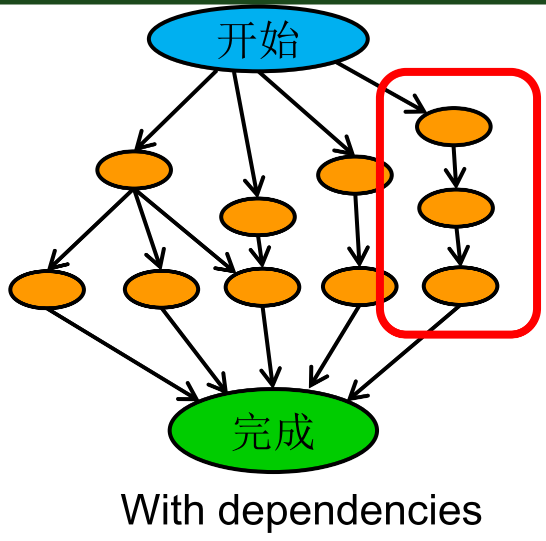


```
int[] mergesort(int nums[]) {
  int numPieces = 10;
  int pieces[][] = split(nums, numPieces);
  for (int i=0; i<numPieces; i++)
    sort(pieces[i]);
  return merge(pieces);
}
```

- 给定一个计算任务，每个核需要分配多大的计算内容？How big a task should we assign to each core?
  - 粗粒度、细粒度的并行化Coarse-grain vs. fine-grain parallelism
- 频繁的任务协调会导致额外的计算开销Frequent coordination creates overhead
  - 往返的消息发送、等待其他核... Need to send messages back and forth, wait for other cores...
  - 后果: 计算机核多数时间花费在核之间的通信、而非计算本身 Result: Cores spend most of their time communicating
- 粗粒度的并行化往往效率更高Coarse-grain parallelism is usually more efficient
  - Bad: 请求每个核对3个数进行排序Ask each core to sort three numbers
  - Good: 请求每一个核对1百万个数进行排序Ask each core to sort a million numbers

依赖关系Dependencies





- 如果计算任务之间存在互依赖关系该怎么办? What if tasks depend on other tasks?
  - 例: 在合并之前需要对各自的链表进行排序 Example: Need to sort lists <u>before</u> merging them
  - 互依赖关系限制了并行度 Limits the degree of parallelism
  - 最小的整体任务完成时间 (或者说最大的加速比) 取决于从开始到完成的最长路径Minimum completion time (and thus maximum speedup) is determined by the longest path from start to finish
    - Assumes resources are plentiful; actual speedup may be lower

异构性Heterogeneity

- 如果存在以下情况... What if...
  - 有些任务比其他任务计算量更大? some tasks are larger than others?
  - 有些任务比其他任务计算时间更长? some tasks are harder than others?
  - 有些任务优先级更高? some tasks are more urgent than others?
  - 并非所有的计算和都是均等速度? not all cores are equally fast, or have different resources?
- 求解调度问题的最优解Result: Scheduling problem
  - 非常非常的难Can be very difficult

小结：并行化Recap: Parallelization

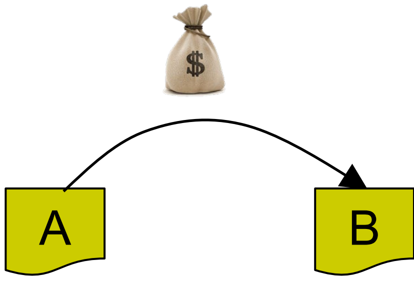
- 并行化并非易事Parallelization is hard
  - 不是所有的算法都是等同的并行化 – 必须很谨慎的选择并行化 Not all algorithms are equally parallelizable -- need to pick very carefully
- 可扩展性往往受到很多因素的制约Scalability is limited by many things
  - Amdahl定律Amdahl's law
  - 任务之间的依赖关系Dependencies between tasks
  - 通信的开销Communication overhead
  - 异构性Heterogeneity
  - ...

调度问题及其算法 Scheduling problem

内容目录

- 并行编程模型及存在的问题
  - 并行化、可扩展性, Amdahl定律 ✓
  - 同步，一致性Synchronization, consistency NEXT
  - 互斥、锁、锁机制所产生的问题
  - 体系结构: SMP, NUMA, Shared-nothing
- 分布式编程及难点
  - 网络分隔故障、CAP定理
  - 故障,失效, 及应对方法

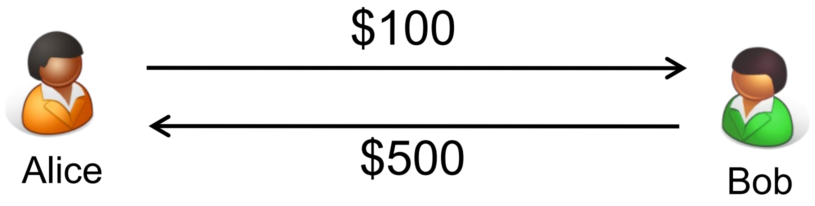
为什么需要进行同步？
Why do we need synchronization?



```
void transferMoney(customer A, customer B, int amount)
{
   showMessage("Transferring "+amount+" to "+B);
   int balanceA = getBalance(A);
   int balanceB = getBalance(B);
   setBalance(B, balanceB + amount);
   setBalance(A, balanceA - amount);
   showMessage("Your new balance: "+(balanceA-amount));
}
```

- 例: 银行的记账系统Simple example: Accounting system in a bank
  - 维护每个银行账户的收支信息 Maintains the current balance of each customer's account
  - 客户之间进行转账 Customers can transfer money to other customers

同步的缘由
Why do we need synchronization?

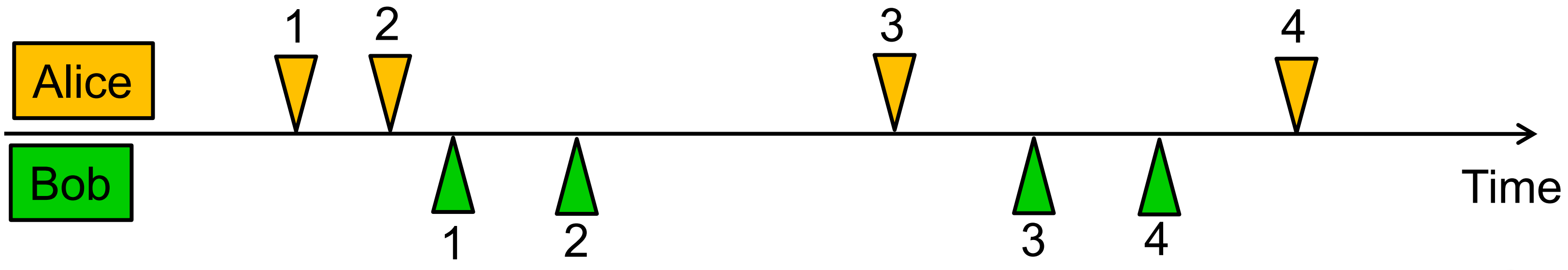


1) B=Balance(Bob)
2) A=Balance(Alice)
3) SetBalance(Bob,B+100)
4) SetBalance(Alice,A-100)

1) A=Balance(Alice)
2) B=Balance(Bob)
3) SetBalance(Alice,A+500)
4) SetBalance(Bob,B-500)

- 如果这两段代码同时执行，计算结果是怎样的?

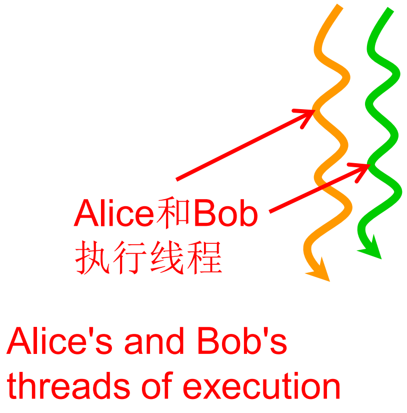
What can happen if this code runs concurrently?



Alice:	$200	$200	$700	$700	$100
Bob:	$800	$900	$900	$300	$300

问题之一 Problem: Race condition

: Race condition(竞争)



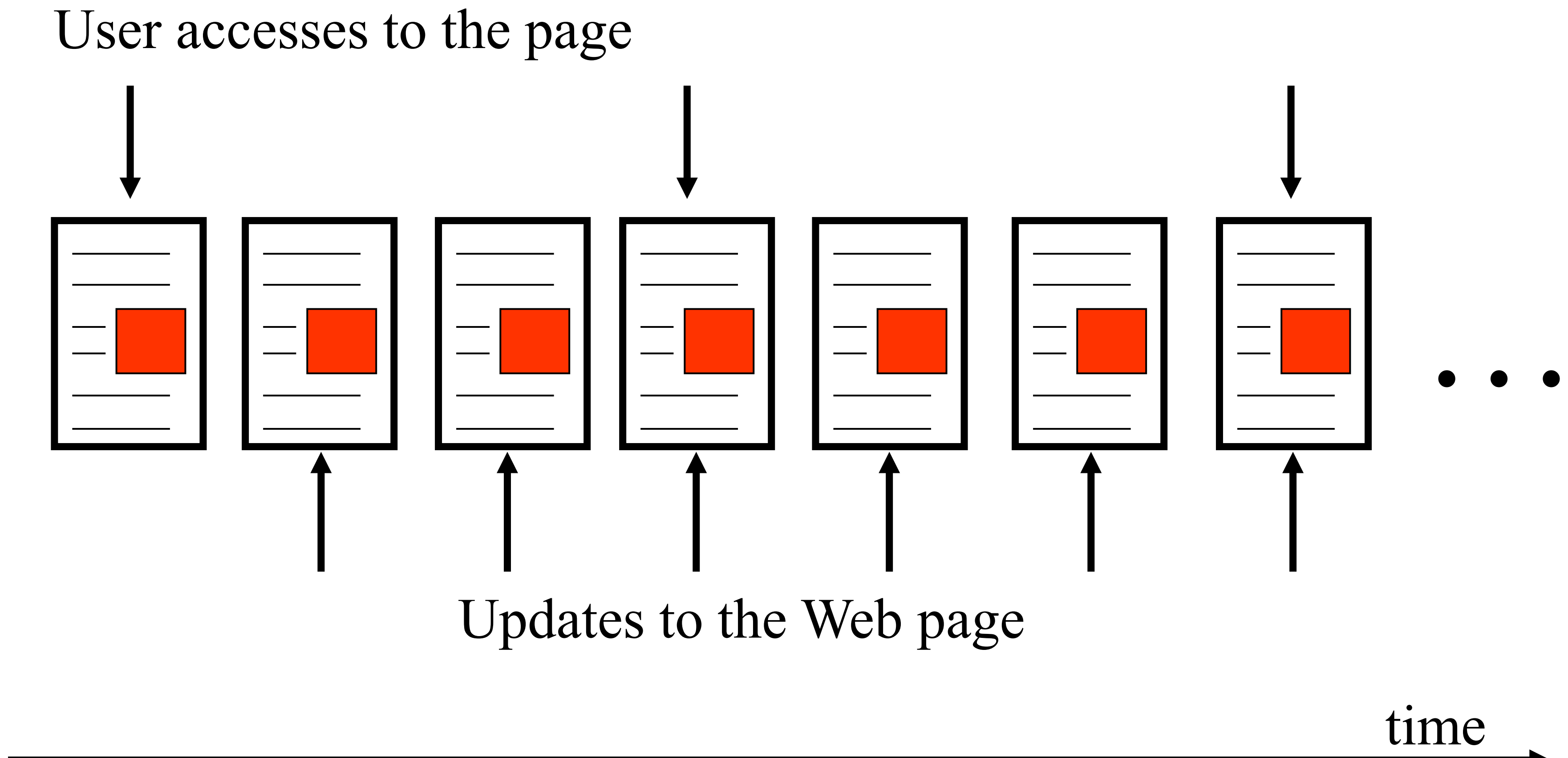
```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

- 竞争What happened?
  - 竞争: 计算结果取决于线程的执行时间, 即, 指令的执行顺序
    Race condition: Result of the computation depends on the exact timing of the two threads of execution, i.e., the order in which the instructions are executed
  - 缘由: 对一个状态的并发更新 Reason: Concurrent updates to the same state
    - 如果所有线程均读取数据、并没有更新操作，是否会发生竞争？Can you get a race condition when all the threads are reading the data, and none of them are updating it?

目标Goal: 一致性 Consistency

- 正确结果是什么？ What should have happened?
  - 不管请求是否并发的执行，计算结果均不应有差别 Intuition: It shouldn't make a difference whether the requests are executed concurrently or not
- 如何形式化描述? How can we formalize this?
  - 通过**一致性模型**描述系统在并发情况下的行为方式 Need a consistency model that specifies how the system should behave in the presence of concurrency

Consistency Issues – Access/Update Ratio



Reasons for Replication

- Performance:
  - Scalability (size and geographical)
- Reliability:
  - Mask failures
  - Mask corrupted data
- Examples:
  - Web caching
  - Horizontal server distribution

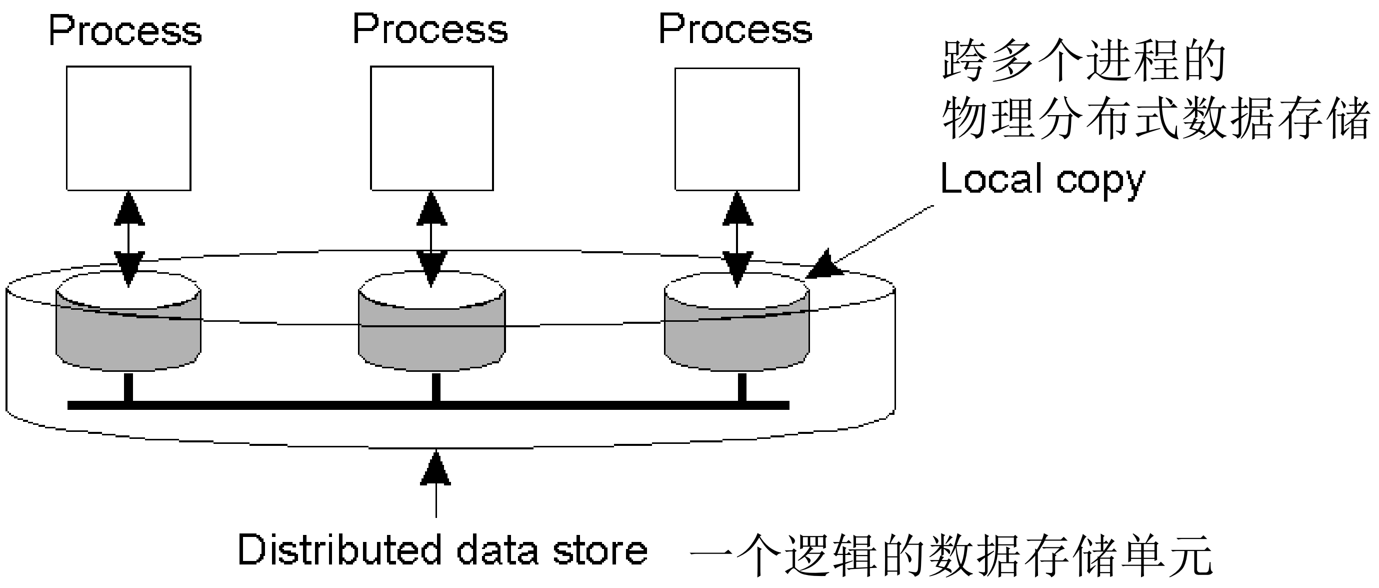
Cost of Replication

- Replicas must be kept consistent Dilemma:
- Replicate data for better performance
- Modification on one copy triggers modifications on all other replicas
- Propagating each modification to each replica can degrade performance

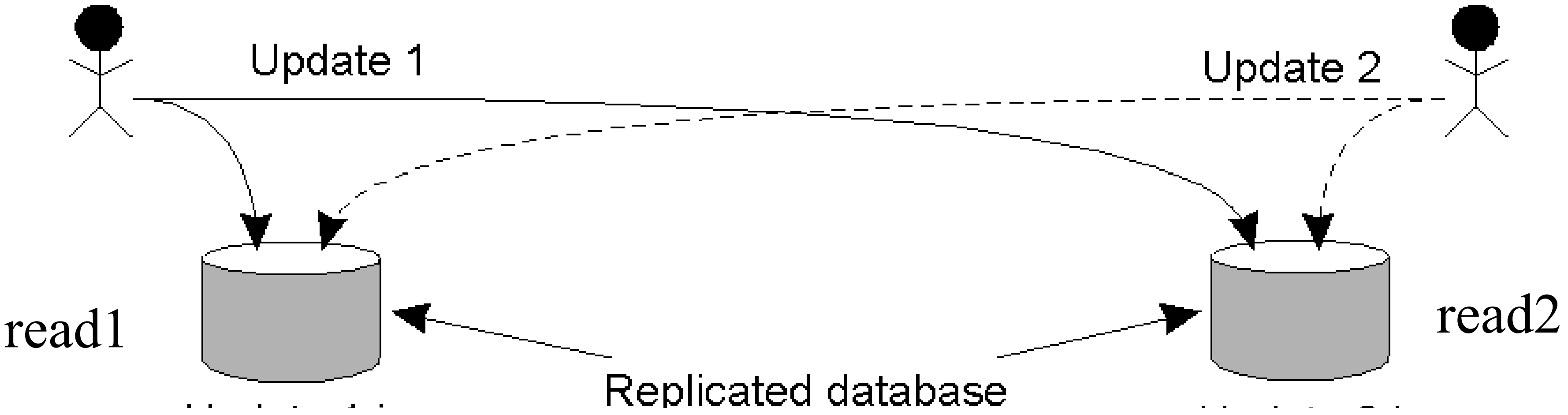
Consistency Model

- Consistency Model = When and how the modifications are made:
  - Weak versus strong consistency model

The general organization of a logical data store, physically distributed and replicated across multiple processes.



Consistency Models (cont)



- A process performs a read operation on a data item, expects the operation to return a value that shows the result of the last write operation on that data
- No global clock ⇒ difficult to define the last write operation
- Different consistency models have **different restrictions** on the values that a read operation can return

Summary of Consistency Models

Models not using synchronization operations.

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

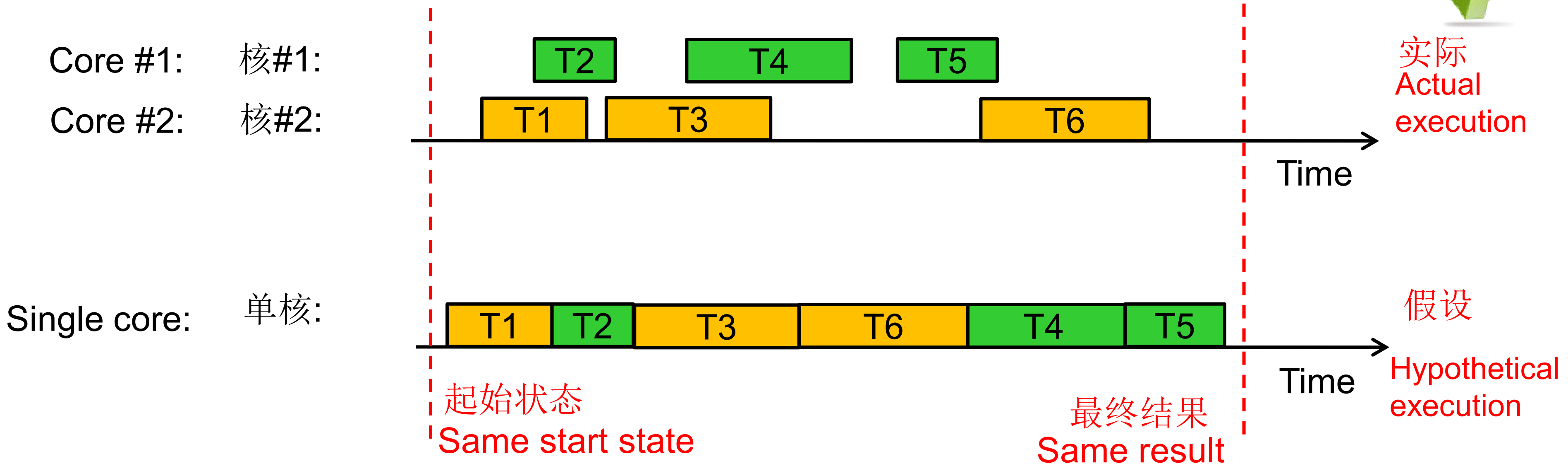
(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

Models with synchronizatio n operations.

顺序一致性Sequential consistency



- 最严格的模型: 顺序一致性 The strictest model: sequential consistency
  - 任意一种情况的执行结果均相同
  - 所有的核均以某个串行的顺序进行计算处理
  - 每一个核在其中的执行顺序均与程序指定的顺序均一致The result of any execution is the same as if the operations of all the cores had been executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program

其他的一致性模型
Other consistency models

- Strong consistency 强一致性模型
  - 仅仅当更新完成之后, 之后的访问才可以访问到更新的值After update completes, all subsequent accesses will return the updated value
- Weak consistency 弱一致性模型
  - 当更新完成之后,之后的访问不必返回已经更新的值；不过必须要满足一些特定的条件After update completes, accesses do not necessarily return the updated value; some condition must be safisfied first
    - 举例: 更新值必须要达到数据对象的副本 Example: Update needs to reach all the replicas of the object
- Eventual consistency最终一致性模型
  - 一种特性是的弱一致性: 如果没有更多的更新，则所有的读操作将最终访问最近的值Specific form of weak consistency: If no more updates are made to an object, then eventually all reads will return the latest value
    - 变形版本: 因果一致性、单调写, ... Variants: Causal consistency, read-your-writes, monotonic writes, ...
    - 我们将介绍更多细节!
- 为什么会这么多的模型呢？Why would we want any of these?
- 如何在系统设计的时候实现这些模型？How do we build systems that achieve them?

Backup Slides

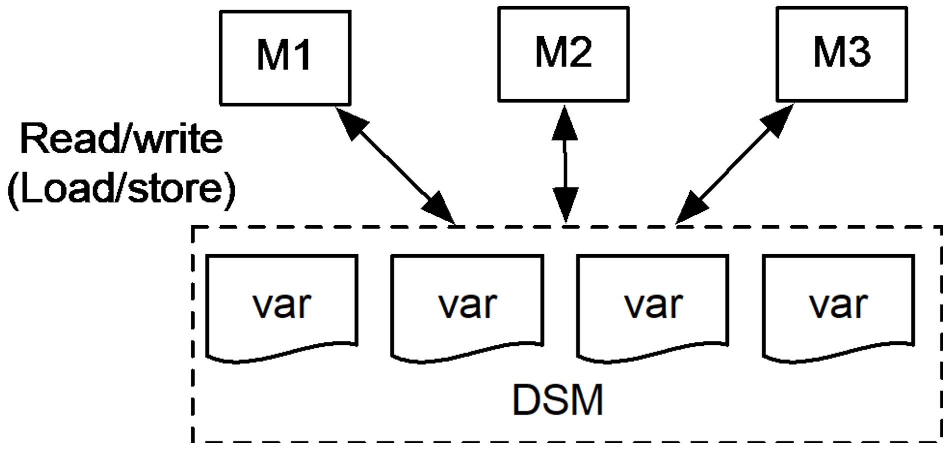
Consistency Models

What is Consistency

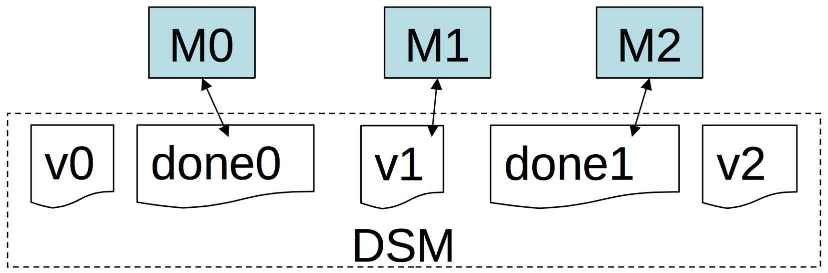
- Consistency
  - Meaning of concurrent reads and writes on shared, possibly replicated, state
  - Important in many designs
- Trade-offs between performance/scalability vs elegance of the design
- We will look at shared memory today
  - Similar concepts in other systems (e.g., storage, filesys)

Distributed Shared Memory (DSM)

- Two models for communication in distributed systems
  - Message passing
  - Shared memory
- Shared memory is often thought more intuitive to write parallel programs than message passing
  - Each machine can access a common address space

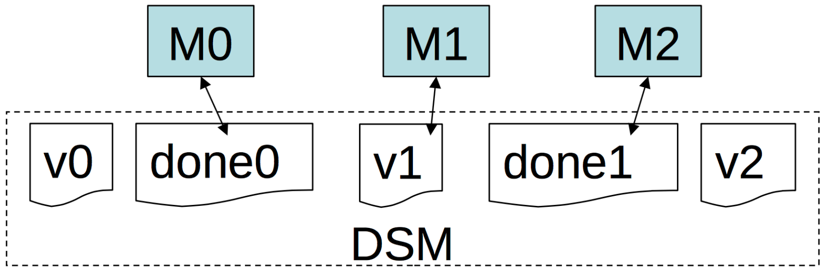


Distributed Shared Memory (DSM)



- M0 writes a value v0 and sets a variable done0 = 1
- After M0 finishes, M1 writes a value v1=f1(v0) and sets a variable done1 = 1
- After M1 finishes, M2 writes a value v2=f2(v0, v1)

Distributed Shared Memory (DSM)



v1 =f1(v0)

v2=f2(v0, v1)

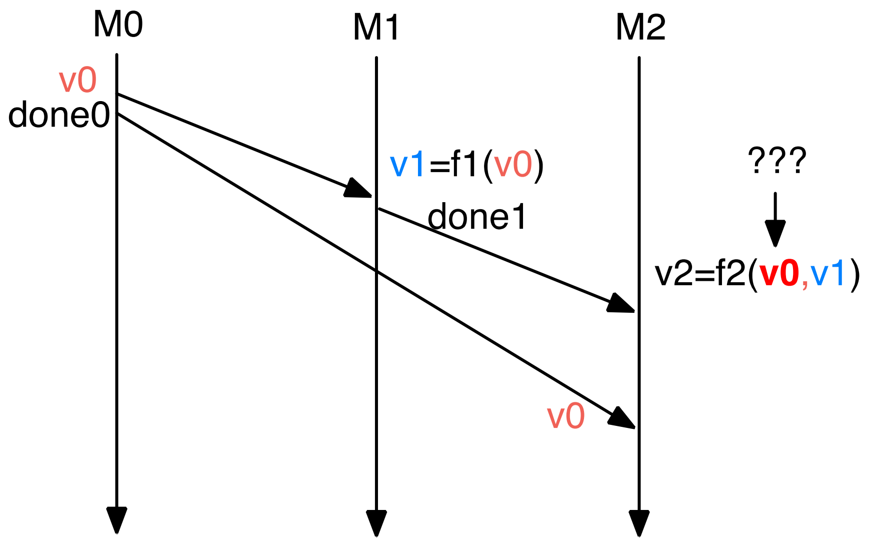
- What's the intuitive intent?
  - M2 should execute f2 based on v0 and v1, which are generated by M0 and M1
  - M2 needs to wait for M1 to finish
  - M1 needs to wait for M0 to finish

A Naïve Solution

- Each machine maintains a local copy of all of memory
- Operations
  - Read: from local memory
  - Write: send updates to all other machines
- Fast: never waits for communication
- Discussion
  - What’s the **issue**?

Problem with the naïve solution

- M2 only needs to wait for done1 signal to start writing v2
- But he doesn't have the latest value of v0 yet!
- M1 and M2 have inconsistent order of M0's write and M1's write



Naïve DSM

- Fast but has unexpected behavior
- A lot of consistency issues
- And we need consistency models to build a distributed system!
  - Depending on what we want

Consistency Models

- Memory system promises to behave according to certain rules, which constitute the system's "consistency model"
  - We write programs assuming those rules

- The rules are a "contract" between memory system and programmer

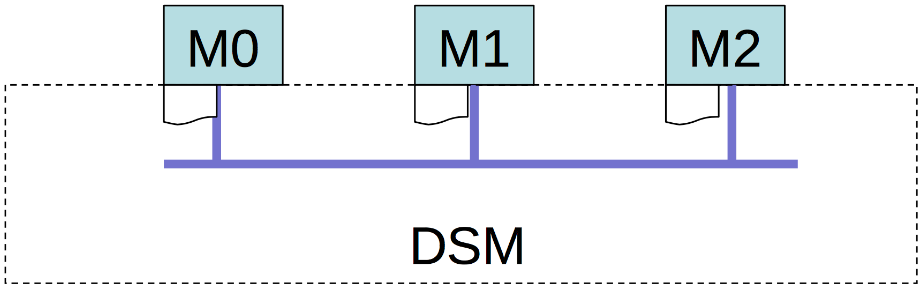
Consistency Models

- Discussion
  - What's the consistency model for a webpage, e.g., shopping, shared doc?

- Consistency is **hard** in (distributed) systems:
  - Data replication (caching)
  - Concurrency
  - Failures

Model 1: Strict Consistency

- Each operation is stamped with a global wall-clock time
- Rules:
  - Rule 1: Each read gets the latest written value
  - Rule 2: All operations at one CPU are executed in order of their timestamps

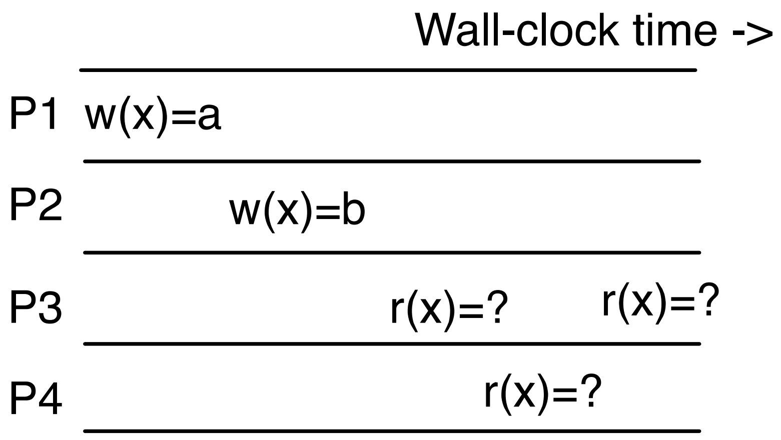


Model 1: Strict Consistency

- Suppose we already implement the rules
  - Rule 1: Each read gets the latest written value
  - Rule 2: All operations at one CPU are executed in order of their timestamps
- Problem 1: Can M1 ever see v0 unset but done0=1?
- Problem 2: Can M1 and M2 disagree on order of M0 and M1 writes?
- So it essentially has the same semantics as a uniprocessor

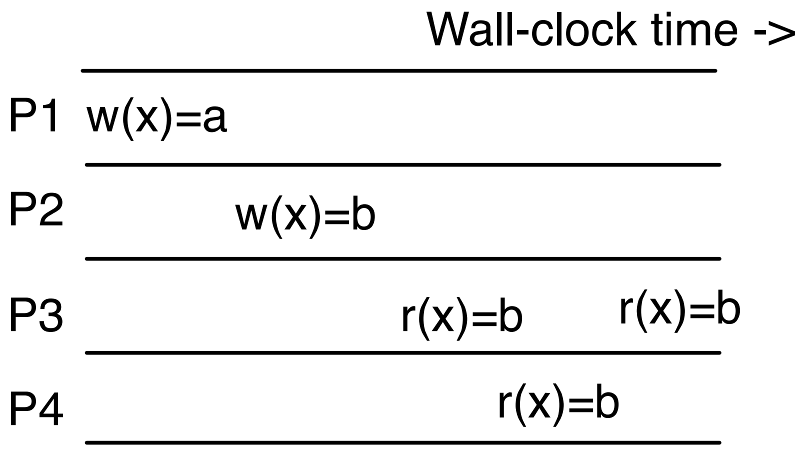
Model 1: Strict Consistency

- We are just like reading and writing on a single processor
- Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued

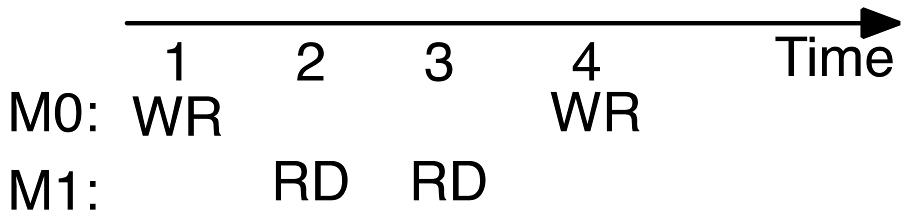


Model 1: Strict Consistency

- We are just like reading and writing on a single processor
- Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued



How to implement Strict Consistency?



- We need to ensure…
  - Each read must be aware of, and wait for, each write
    - RD@2 aware of WR@1; WR@4 must know how long to wait
  - Real-time clocks are strictly synchronized
- Unfortunately
  - Time between instructions << speed-of-light
  - Real-clock synchronization can be tough (even now)
- So, strict consistency is tough to implement efficiently

内容目录

- 并行编程模型及存在的问题
  - 并行化、可扩展性, Amdahl定律 ✓
  - 同步， 一致性 ✓
  - 互斥、锁、锁机制所产生的问题Mutual exclusion, locking, issues related to locking ← NEXT
  - 体系结构: SMP, NUMA, Shared-nothing

- 分布式编程及难点
  - 网络分隔故障、CAP定理
  - 故障,失效, 及应对方法

互斥Mutual exclusion

临界区

Critical section

```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

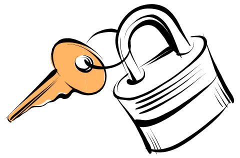
- 如何取得效果更好的一致性? How can we achieve better consistency?
  - 关键点: 通过设计一个**代码的临界区**，使得其他核运行这段代码的时候并访问同一资源的时候，就会出现问题；Key insight: Code has a critical section where accesses from other cores to the same resources will cause problems
- 方法: **互斥**Approach: Mutual exclusion
  - 限制在任意时刻只能有一个核/机器才能执行该临界区代码Enforce restriction that only one core (or machine) can execute the critical section at any given time
  - 问题：还有可扩展性么？What does this mean for scalability?

锁Locking

临界区
Critical section

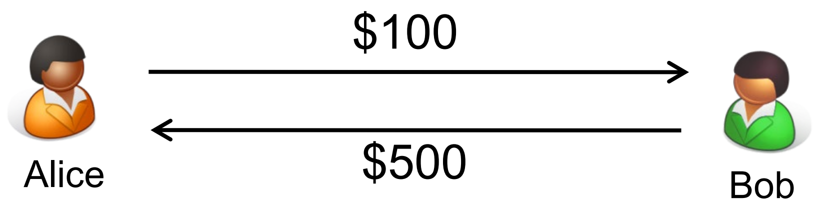
```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

- 思路:Idea: Implement locks
  - 如果**X还没有上锁,** 再调用LOCK(X), 可以**对X上锁**继续执行If LOCK(X) is called and X is not locked, lock X and continue
  - 如果**X已经上锁,** 再调用LOCK(X), <u>等待</u>**一直到X解锁**If LOCK(X) is called and X is locked, <u>wait</u> until X is unlocked
  - 如果**X已经上锁**, 再调用UNLOCK(X), 则可对X解锁If UNLOCK(X) is called and X is locked, unlock X

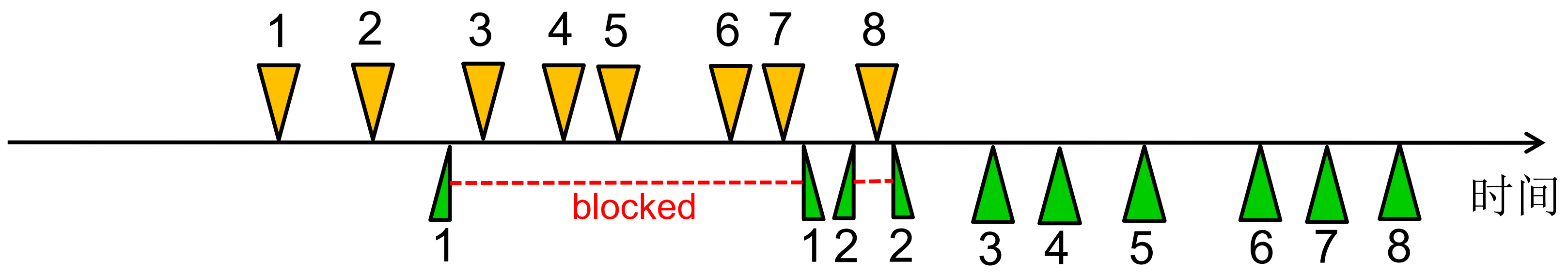


- 有多少锁呢，且在什么位置可以用锁? How many locks, and where do we put them?
  - 选项 #1: 在临界区周围用锁Option #1: One lock around the critical section
  - 选项 #2: 每个变量用一把锁 (即`balanceA`、`balanceB`) Option #2: One lock per variable (A's and B's balance)
  - 优点、缺陷？还有其他办法吗？Pros and cons? Other options?

锁Locking helps!

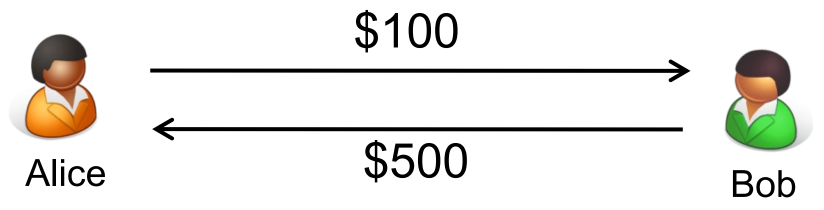


Alice	Bob
1) LOCK(Bob)	1) LOCK(Alice)
2) LOCK(Alice)	2) LOCK(Bob)
3) B=Balance(Bob)	3) A=Balance(Alice)
4) A=Balance(Alice)	4) B=Balance(Bob)
5) SetBalance(Bob,B+100)	5) SetBalance(Alice,A+500)
6) SetBalance(Alice,A-100)	6) SetBalance(Bob,B-500)
7) UNLOCK(Alice)	7) UNLOCK(Bob)
8) UNLOCK(Bob)	8) UNLOCK(Alice)



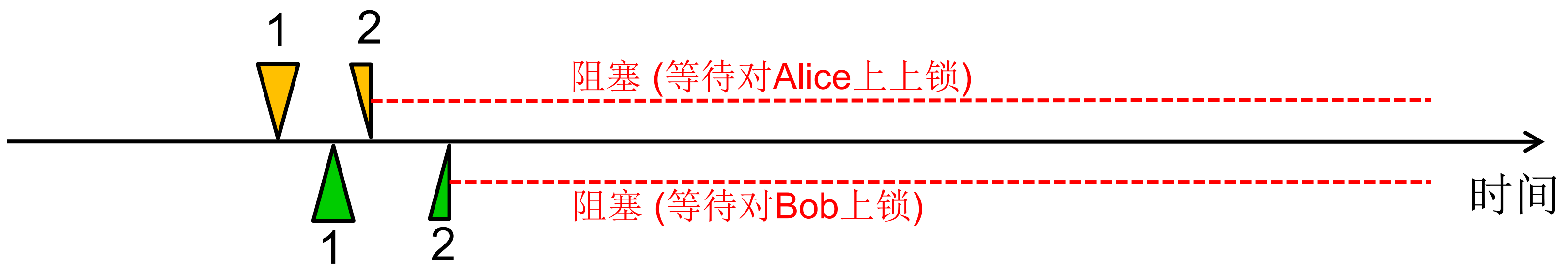
Alice的balance:	$200	$200	$100	$600	$600
Bob的 balance:	$800	$900	$900	$900	$400

死锁：Deadlock



```
1)  LOCK(Bob)
2)  LOCK(Alice)
3)  B=Balance(Bob)
4)  A=Balance(Alice)
5)  SetBalance(Bob,B+100)
6)  SetBalance(Alice,A-100)
7)  UNLOCK(Alice)
8)  UNLOCK(Bob)
```

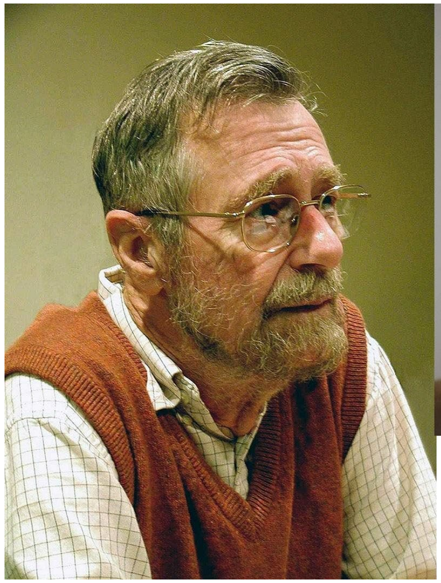
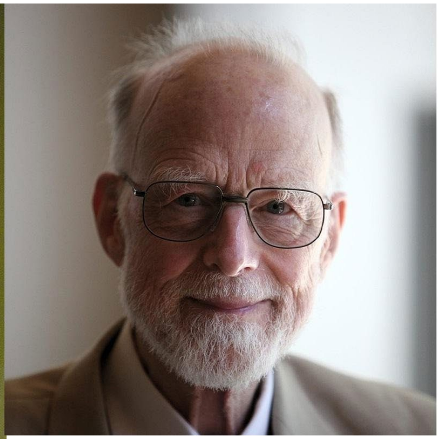
```
1)  LOCK(Alice)
2)  LOCK(Bob)
3)  A=Balance(Alice)
4)  B=Balance(Bob)
5)  SetBalance(Alice,A+500)
6)  SetBalance(Bob,B-500)
7)  UNLOCK(Bob)
8)  UNLOCK(Alice)
```



- 没有一个处理器可以继续往下做，均出现等待阻塞
  Neither processor can make progress!

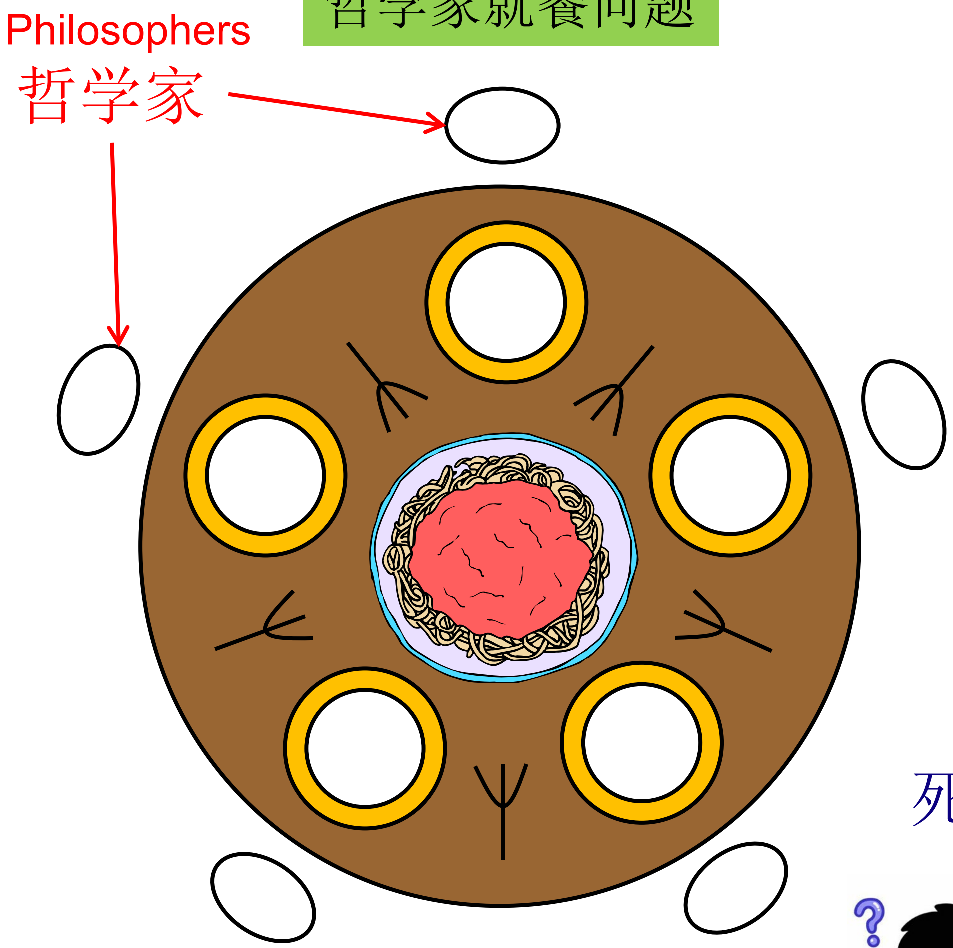
哲学家就餐问题Dining Philosophers Problem

- 最初是由荷兰计算机科学家**艾兹赫尔·韦伯·戴克斯特拉**Edsger Wybe Dijkstra于1971年提出的
- 如果有五台计算机想同时访问五个共享的磁带驱动器，该如何进行调度的问题
- 英国计算机科学家**安东尼·霍尔爵士**C. A. R. Hoare重新表述成“哲学家就餐问题”

The dining philosophers problem

哲学家就餐问题



- 餐桌中间有一大碗意大利面
- 一只餐叉很难吃到面条，必须用两只餐叉吃东西。
- 只能使用左右手的两只餐叉

```
Philosopher:

repeat
   think
   pick up left fork
   pick up right fork
   eat
   put down forks
forever
```

死锁

每个哲学家都拿着左手的餐叉
永远都在等右边的餐叉(或者相反)

发生死锁的根本原因是什么？

如何解决死锁问题
What to do about deadlocks

- Many possible solutions, including:
  - 锁管理器Lock manager: Hire a waiter and require that philosophers must ask the waiter before picking up any forks
    - 雇服务员, 哲学家在用叉之前必须向服务器发出申请
    - 结果是有利于可扩展性吗? Consequences for scalability?
  - 资源分级Resource hierarchy: Number forks 1-5 and require that each philosopher pick up the fork with the lower number first
    - 对叉子进行编号1-5，每个哲学家必须先拿编号最小的叉, 然后才拿编号大的叉，用餐好之后先放编号大的叉，然后是编号小的叉 → 保证没有两个哲学家同时使用同一个叉
    - 效率? Problem?
  - **Chandy/Misra 解法** Chandy/Misra solution: **这个解法允许很大的并行性，适用于任意大的问题**
    - 把叉子凑对, 让要吃的人先吃, 没叉子的人得到一张券。Forks can either be dirty or clean; initially all forks are dirty
    - 饿了, 把券交给有叉子的人, 有叉子的人吃饱了会把叉子交给发券的人. After the philosopher has eaten, all his forks are dirty
    - 有了券的人不会再得到第二张券。When a philosopher needs a fork he can't get, he asks his neighbor
    - 保证有叉子的都有得吃。If a philosopher is asked for a dirty fork, he cleans it and gives it up
    - If a philosopher is asked for a clean fork, he keeps it

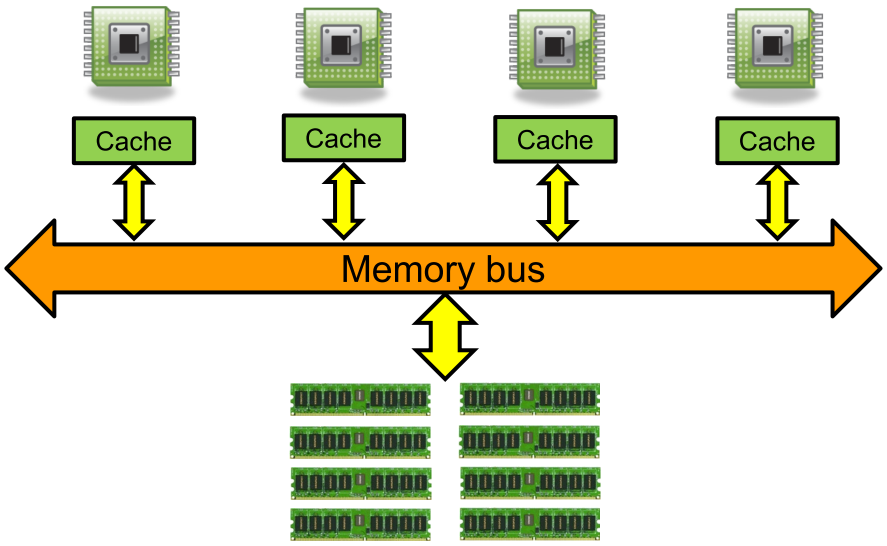
其他方法Alternatives to locks

- 锁并非唯一的途径! Locks aren't the only solution!
  - 还有其他方法解决并发问题... There are many ways to handle concurrency...
  - ... 但是必须要正确使用! ... but you DO have to handle it properly!
- 举例: 乐观并发控制Example: "Optimistic" concurrency control
  - 允许并发更新Allow accounts to be updated concurrently
  - 试图合并并非更新的结果Try to "merge" the effects (e.g., reconcile the accounts)
  - 如果不能，则串行处理If not possible, do things in sequence
- 举例: Abort及重启的Example: Abort and restart
  - 如果所有的锁均用完，则释放所有的锁再重试If not all locks are available, release all and retry

内容目录

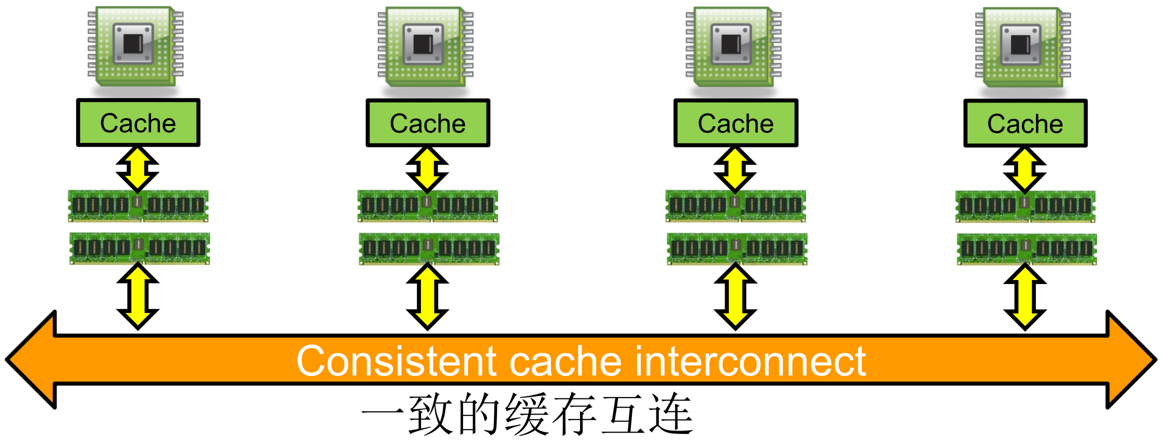
- 并行编程模型及存在的问题
  - 并行化、可扩展性, Amdahl定律 ✓
  - 同步， 一致性 ✓
  - 互斥、锁、锁机制所产生的问题 ✓
  - 体系结构: SMP, NUMA, Shared-nothing NEXT
- 分布式编程及难点
  - 网络分隔故障、CAP定理
  - 故障,失效, 及应对方法

Symmetric Multiprocessing (SMP) 对称多处理器



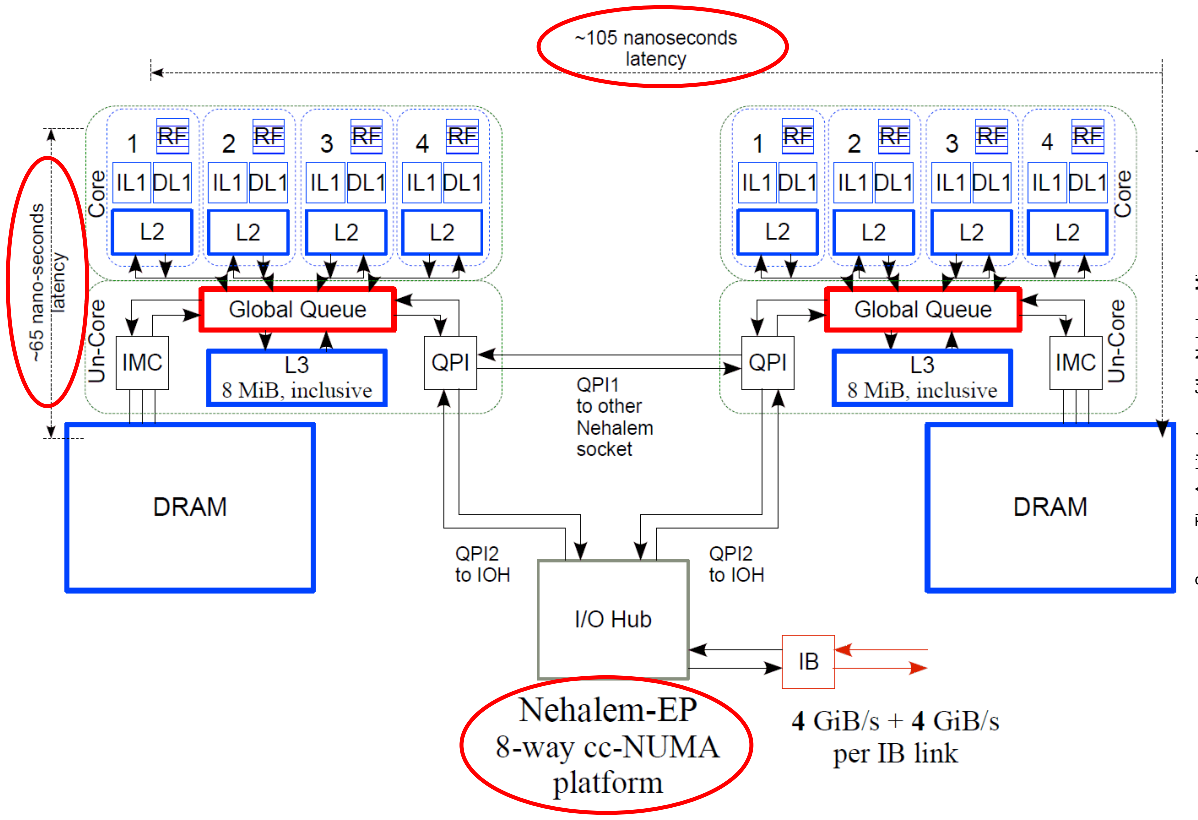
- 所有的处理均共享同一内存All processors share the same memory
  - 任何CPU都可以访问任一个内存字节; 延迟均相同Any CPU can access any byte; latency is always the same
  - 优点: 简单、容易取得负载均衡Pros: Simplicity, easy load balancing
  - 缺点: 扩展性有限 (~10处理器), 贵Cons: Limited scalability (~10 processors), expensive

Non-Uniform Memory Architecture (NUMA) 非一致性内存架构



- 内存只对临近的特定处理器可访问Memory is local to a specific processor
  - 每个CPU仍然可以访问任意内存字节, 但是本地内存的访问速度更快 (2-3x), 远端内存访问速度明显慢Each CPU can still access any byte, but accesses to 'local' memory are considerably faster (2-3x)
  - 优点: 扩展性更优Pros: Better scalability
  - 缺陷: 编程复杂、扩展性任然有限Cons: Complicates programming a bit, scalability still limited

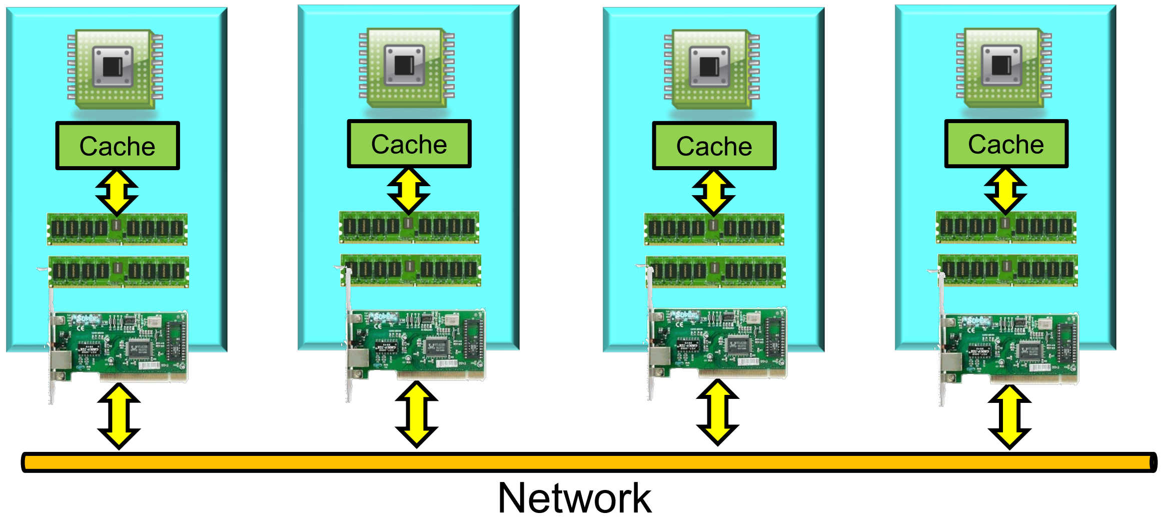
举例：Intel Nehalem志强芯片



Source: The Architecture of the Nehalem Microprocessor and Nehalem-EP SMP Platforms, M. Thomadakis, Texas A&M

- 访问远处的内存更慢
  - 105ns vs 65ns
- Access to remote memory is slower
  - In this case, 105ns vs 65ns

Shared-Nothing无共享架构



- 通过网络互相的完全独立的计算机 Independent machines connected by network
  - 每个CPU只能访问本地内存; 如需要访问远端计算机的数据, 则发送消息 Each CPU can only access its local memory; if it needs data from a remote machine, it must send a message there
  - 优点: 更优的扩展性Pros: Much better scalability
  - 缺陷: 编码模型困难Cons: Nontrivial programming model

内容目录

- 并行编程模型及存在的问题
  - 并行化、可扩展性, Amdahl定律 ✓
  - 同步，一致性 ✓
  - 互斥、锁、锁机制所产生的问题 ✓
  - 体系结构: SMP, NUMA, Shared-nothing ✓
- 分布式编程及难点
  - 网络分隔故障、CAP定理
  - 故障,失效, 及应对方法

敬请期待

下节课:

互联网基础; 故障及失效

Next time you will learn about:

Internet basics; faults and failures

同济大学软件学院
School of Software Engineering, Tongji University