



# Lecture 03: 云分布式计算

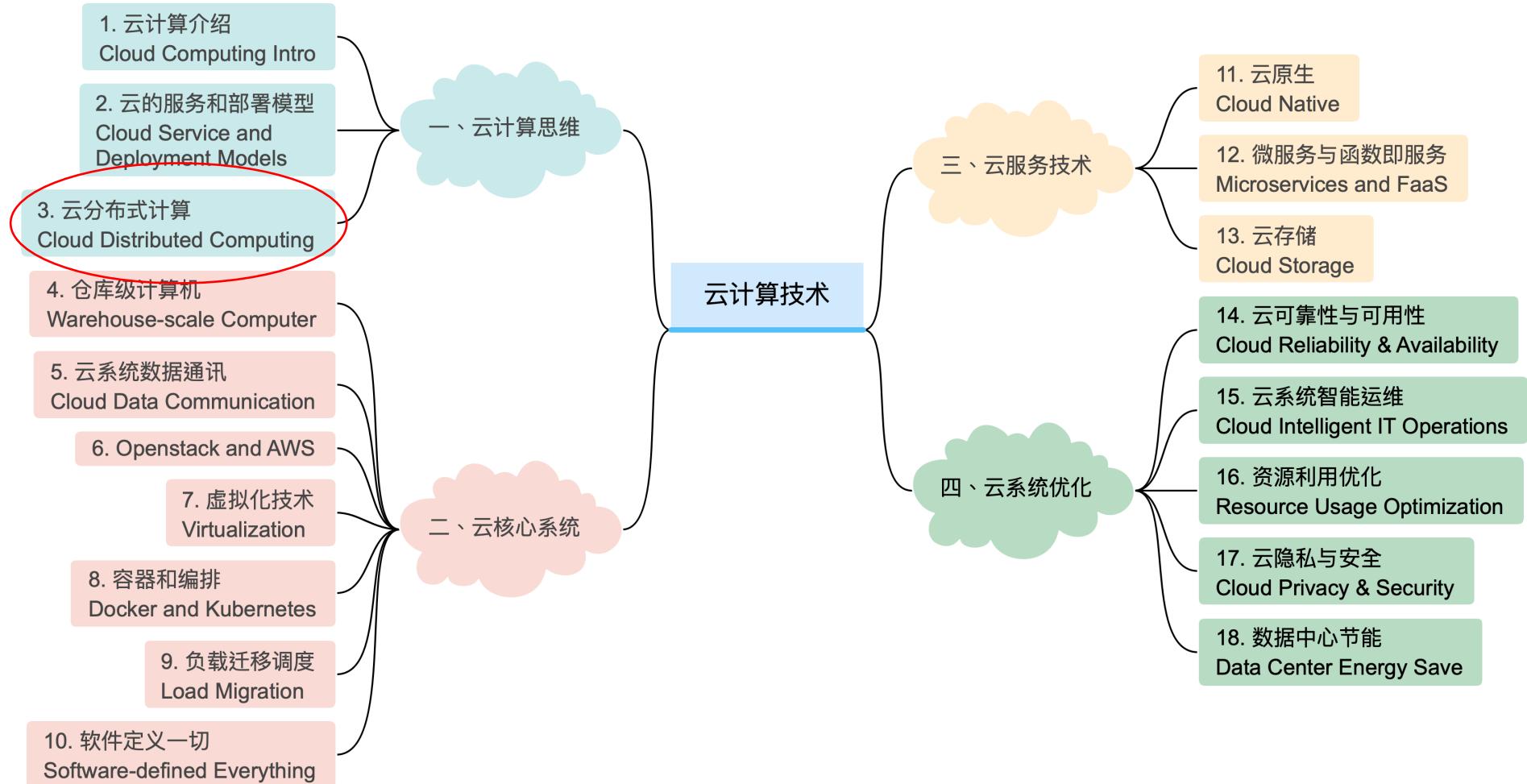
SSE316: 云计算技术  
Cloud Computing Technologies

---

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn



# Today's topics

□ 分布式计算概要

□ MapReduce

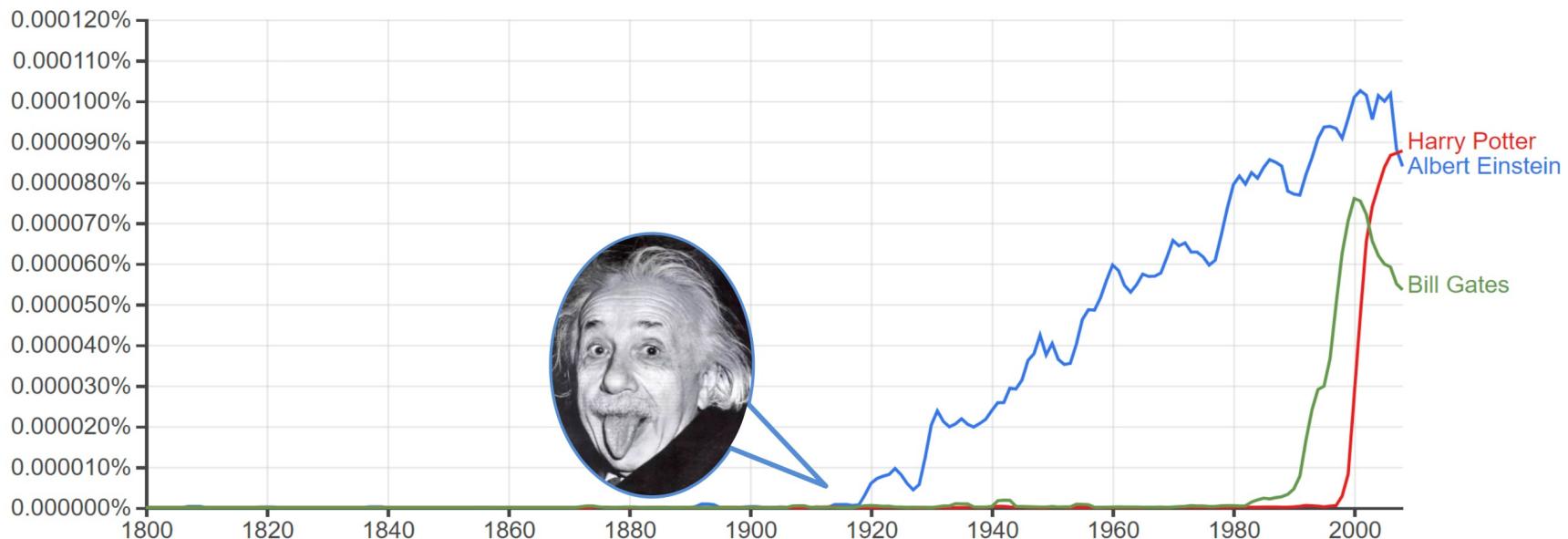
□ MapReduce 深度优化

- 任务延迟问题
- 长尾问题

# Google Books Ngram Viewer

Graph these comma-separated phrases:   case-insensitive

between  and  from the corpus  ▾ with smoothing of  ▾ **Search lots of books**



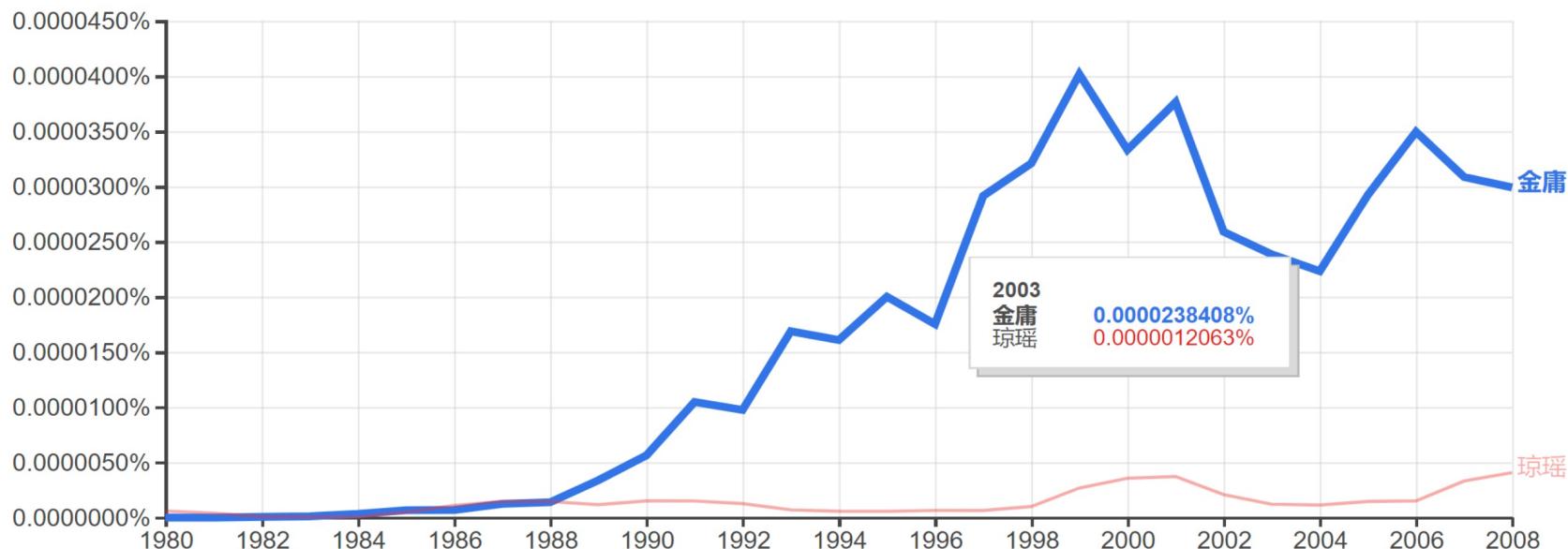
1800年 -- 2000年  
约全世界 4% 图书

超过 5 亿个词汇  
中英等多国语言

# Google Books Ngram Viewer

Graph these comma-separated phrases: 金庸,琼瑶  case-insensitive

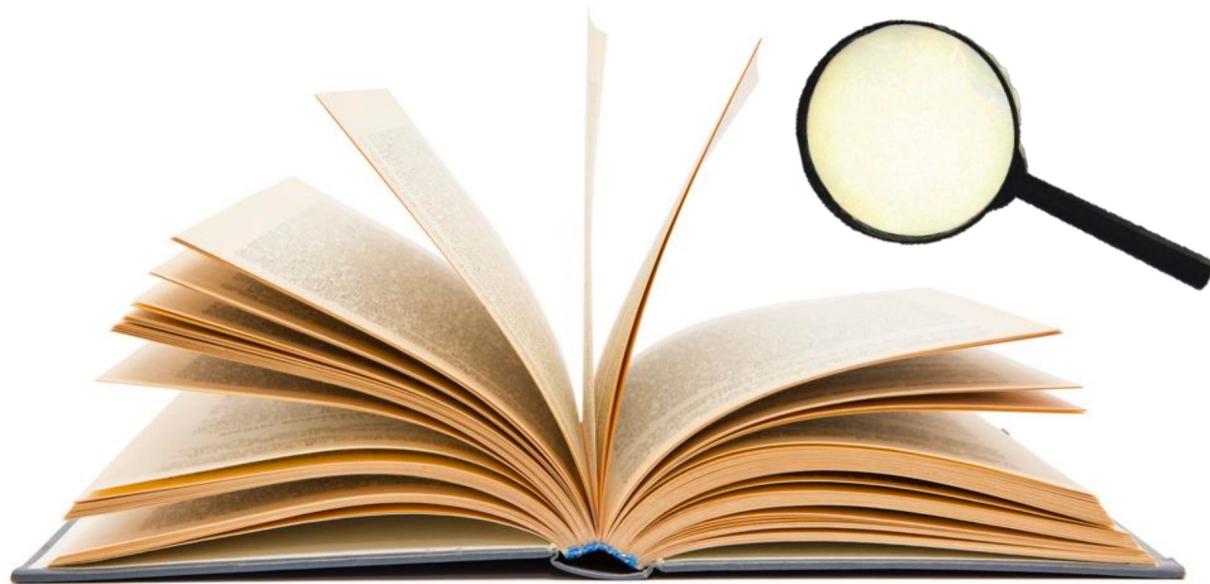
between 1980 and 2008 from the corpus Chinese (simplified) ▾ with smoothing of 1 ▾ [Search lots of books](#)



1800年 -- 2000年  
约全世界 4% 图书

超过 5 亿个词汇  
中英等多国语言

# 怎么统计文献中单词出现的次数？



# 计数问题

1

**Hello World Bye World  
Hello SYSU Bye SYSU  
Bye SYSU Hello SYSU**

挨个计数

2

**Hello (1) World (1) Bye (1) World (1)  
Hello (1) SYSU (1) Bye (1) SYSU (1)  
Bye (1) SYSU (1) Hello (1) SYSU (1)**

# 计数问题

Hello (1) World (1) Bye (1) World (1)  
Hello (1) SYSU (1) Bye (1) SYSU (1)  
Bye (1) SYSU (1) Hello (1) SYSU (1)

## 归纳统计

Hello (1) Hello (1) Hello (1)  
World (1) World (1)  
Bye (1) Bye (1) Bye (1)  
SYSU (1) SYSU (1) SYSU (1) SYSU (1)

# 计数问题

Hello (1) Hello (1) Hello (1)

World (1) World (1)

Bye (1) Bye (1) Bye (1)

SYSU (1) SYSU (1) SYSU (1) SYSU (1)

求和输出

Hello: (1+1+1=3)

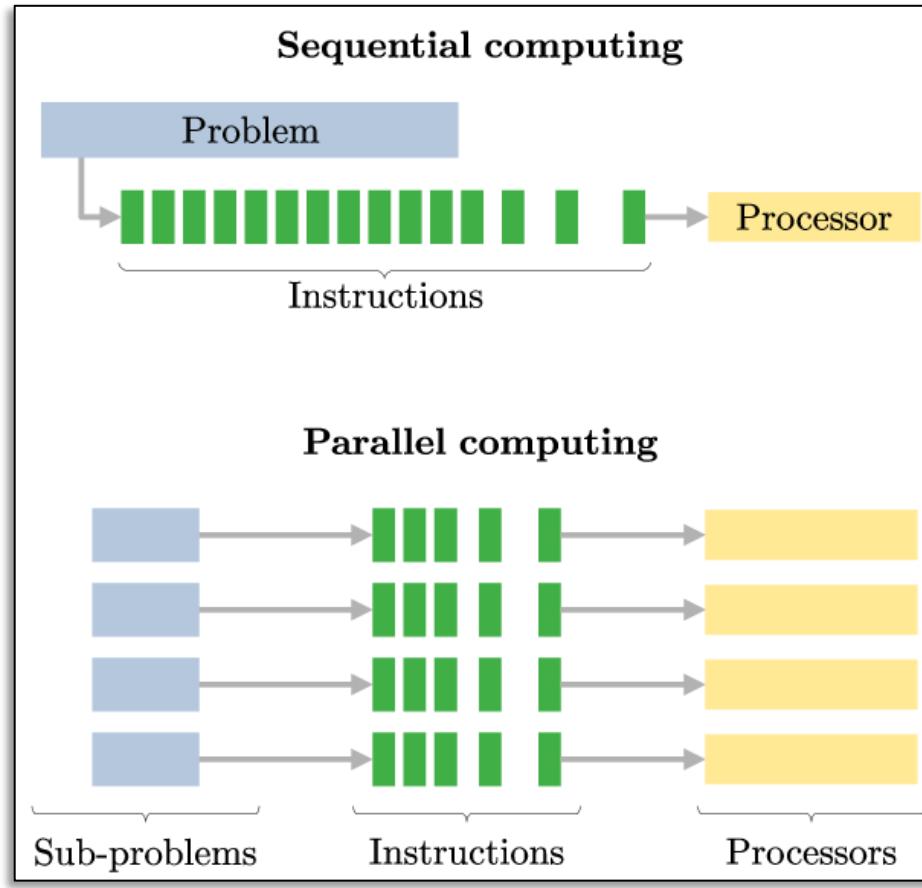
World: (1+1=2)

Bye: (1+1+1=3)

SYSU: (1+1+1+1=4)

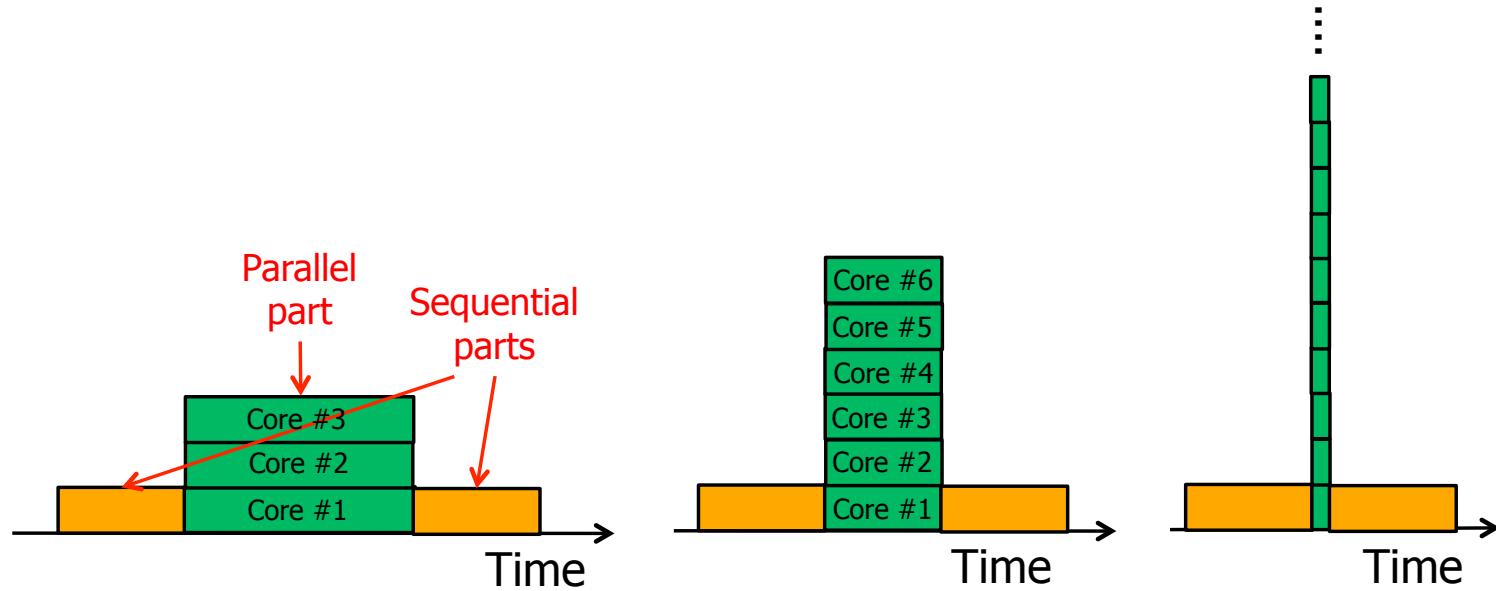
# 如何提高计数的效率?

## 并行计算! Parallel Computing!



并行计算是一种**计算模型**，指同时使用多种计算资源解决计算问题，以提高计算速度和效率

# 阿姆达尔定律 Amdahl' s Law



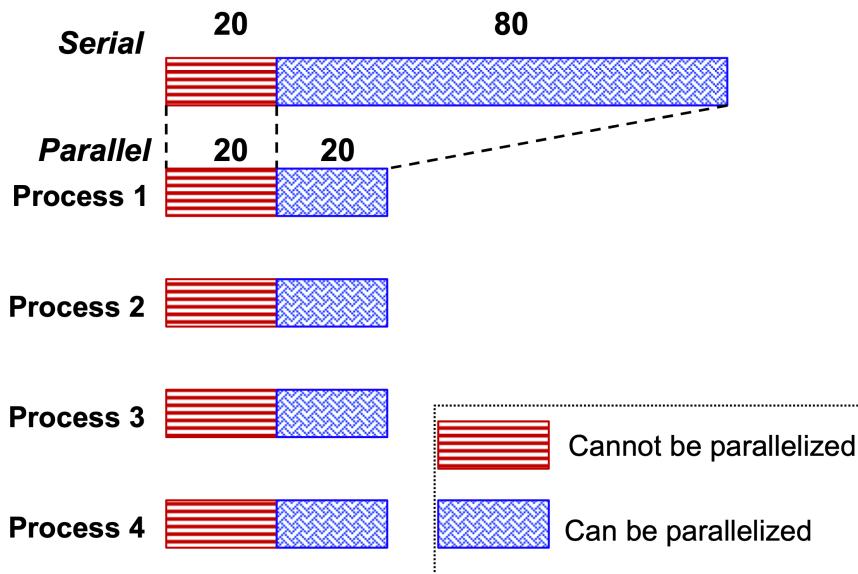
- 通常，一个程序并不是所有部分都可以并行化处理
- 假设某程序可并行处理的部分占比为  $f$ ，并行处理的节点个数为  $p$ ，则并行的加速比  $S$  为：

$$S = \frac{1}{(1 - f) + \frac{f}{p}}$$

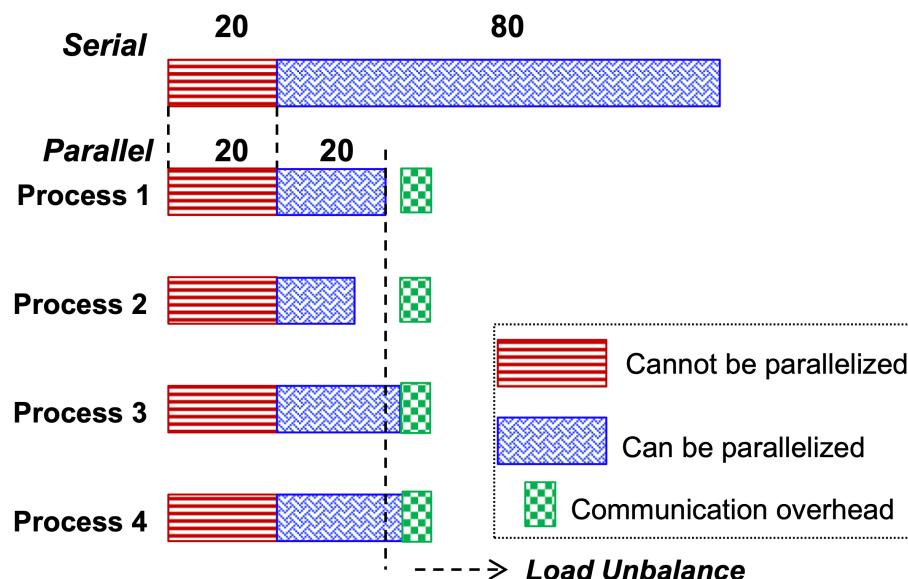
# 阿姆达尔定律

## □ 理想与真实的情况

- 阿尔达姆定律太过简单了，无法被应用到实际中
- 当我们运行一个并行程序时，一般来说，进程之间存在**通信开销、同步开销和负载不平衡**

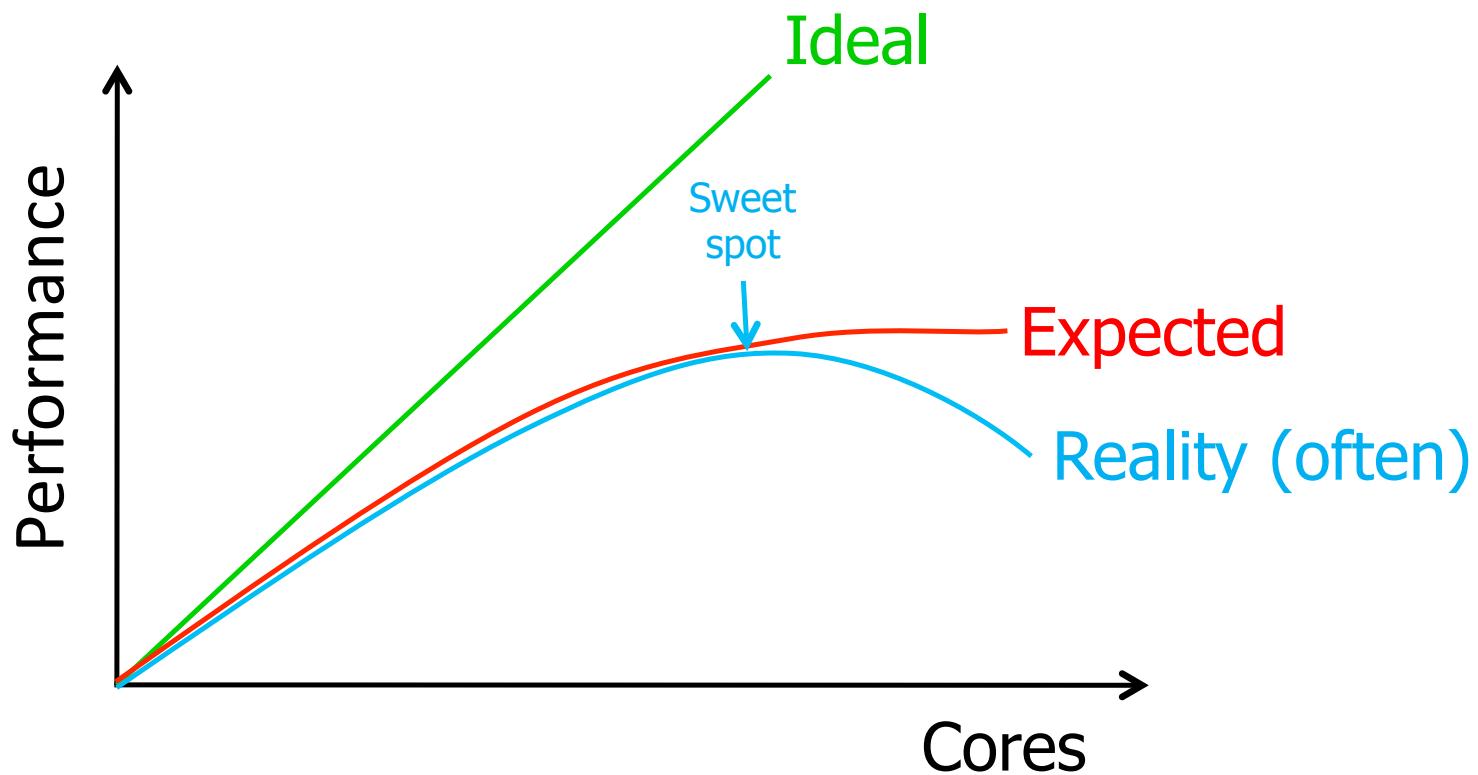


1. Parallel Speed-up: An Ideal Case



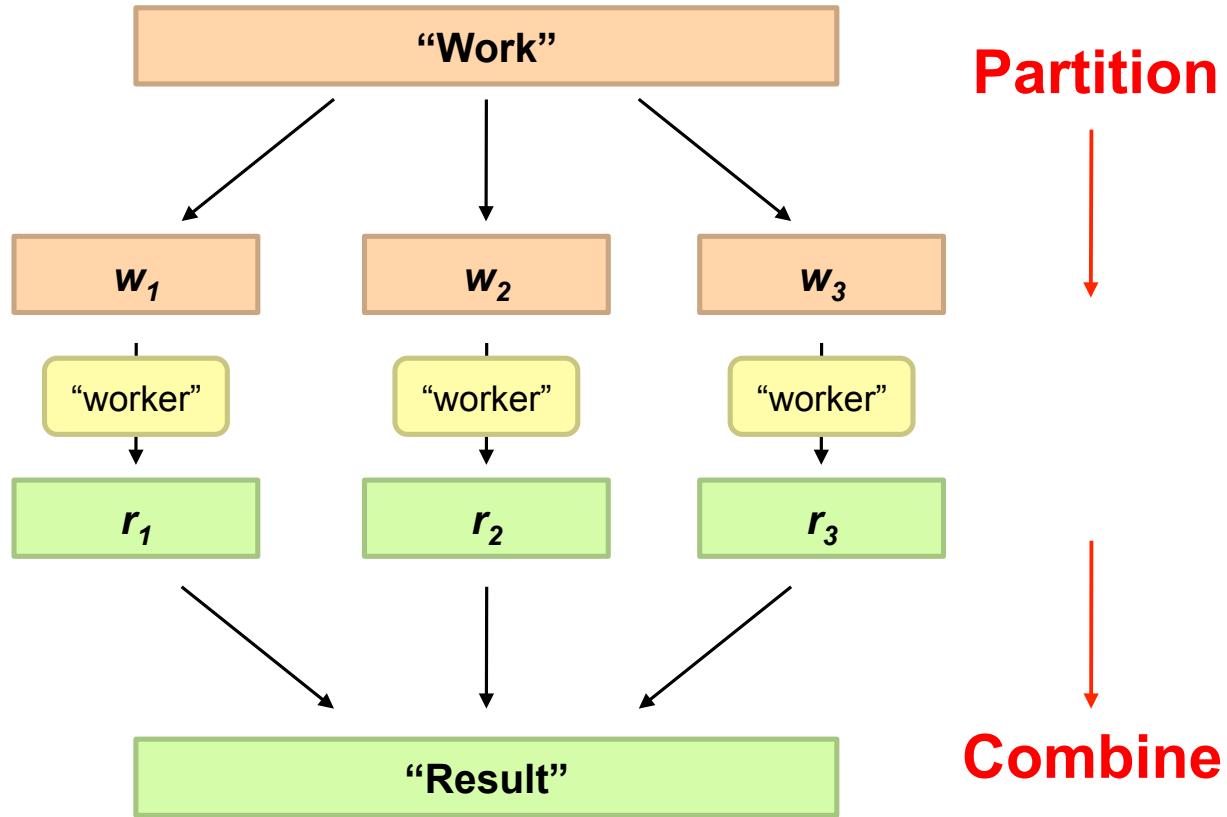
2. Parallel Speed-up: An Actual Case

因此，当并行超过一定程度时，性能可能会下降！



# **How do we scale-up for Web-Scale Information Analytics?**

# 分治 Divide and Conquer



# 并行化的挑战

- 如何将 task 分配给 worker?
- 如果 task 比 worker 多怎么办?
- 如果 worker 需要共享部分结果怎么办?
- 如何汇总部分结果?
- 如何知道所有 worker 已经完成了工作?
- 如果 worker 故障怎么办?

**What is the common theme of all of these problems?**

# 并行化的挑战

## Two common themes

1. Communication between workers (e.g., to exchange state)
2. Access to shared resources (e.g., data)



Source: Ricardo Guimarães Herrmann

# 管理多个 worker

□ 很困难，因为我们

- 不知道 worker 运行的顺序
- 不知道 worker 什么时候会互相干扰
- 不知道 worker 访问共享数据的顺序

□ 因此，我们需要

- 信号量 Semaphores (锁定 lock、解锁 unlock)
- 条件变量 Conditional variables (等待 wait、通知 notify、广播 broadcast)
- 同步屏障 Barriers

□ 但还是有很多问题

- 死锁 Deadlock, 活锁 Livelock, 竞争条件 Race conditions
- Dining philosophers, sleeping barbers, cigarette smokers...

# Where the rubber meets the road

## 养兵千日，用兵一时

□ 并发性 (concurrency) 很难推理

□ 以下场景使并发推理更困难

- 数据中心规模 (甚至跨数据中心)
- 出现故障时
- 多个服务交互

□ 更不用说调试 (debugging) 了

□ 现实情况

- 大量一次性的解决方案，自定义代码
- 编写自己的专用库，然后使用它进行编程
- 程序员显式管理一切负担



# What is the point?

隐藏细节

- 开发者不需要关心 race conditions, lock contention 等并发问题

分离 **What**  
和 **How**

- 开发者指定需要执行的计算
- 执行框架 (“runtime”) 处理实际的计算

# **Computational Model for Web-scale Information Processing: MapReduce**

# 大数据并行运算编程模型

## MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

[jeff@google.com](mailto:jeff@google.com), [sanjay@google.com](mailto:sanjay@google.com)

*Google, Inc.*

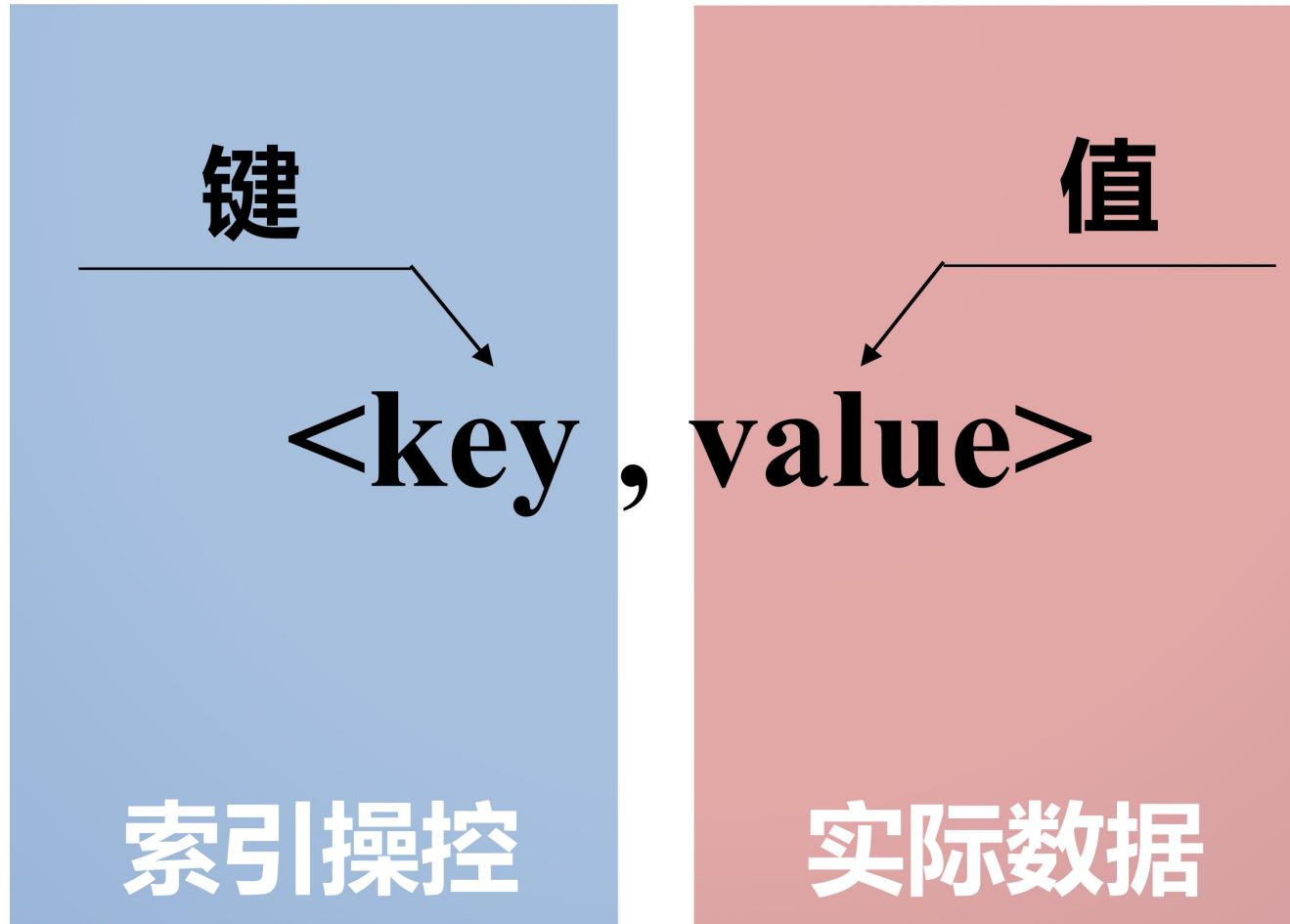


# 典型的大数据计算问题

- Iterate over a large number of records
- Map** ○ Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results **Reduce**
- Generate final output

Key idea: provide a functional abstraction  
for these two operations

数据流以特定结构呈现，即 **key-value** 分布式存储



# MapReduce

Map: 映射操作

定义

Reduce: 归纳化简

主要  
操作

Map

$(key_1, val_1) \rightarrow <key_2, val_2>^*$

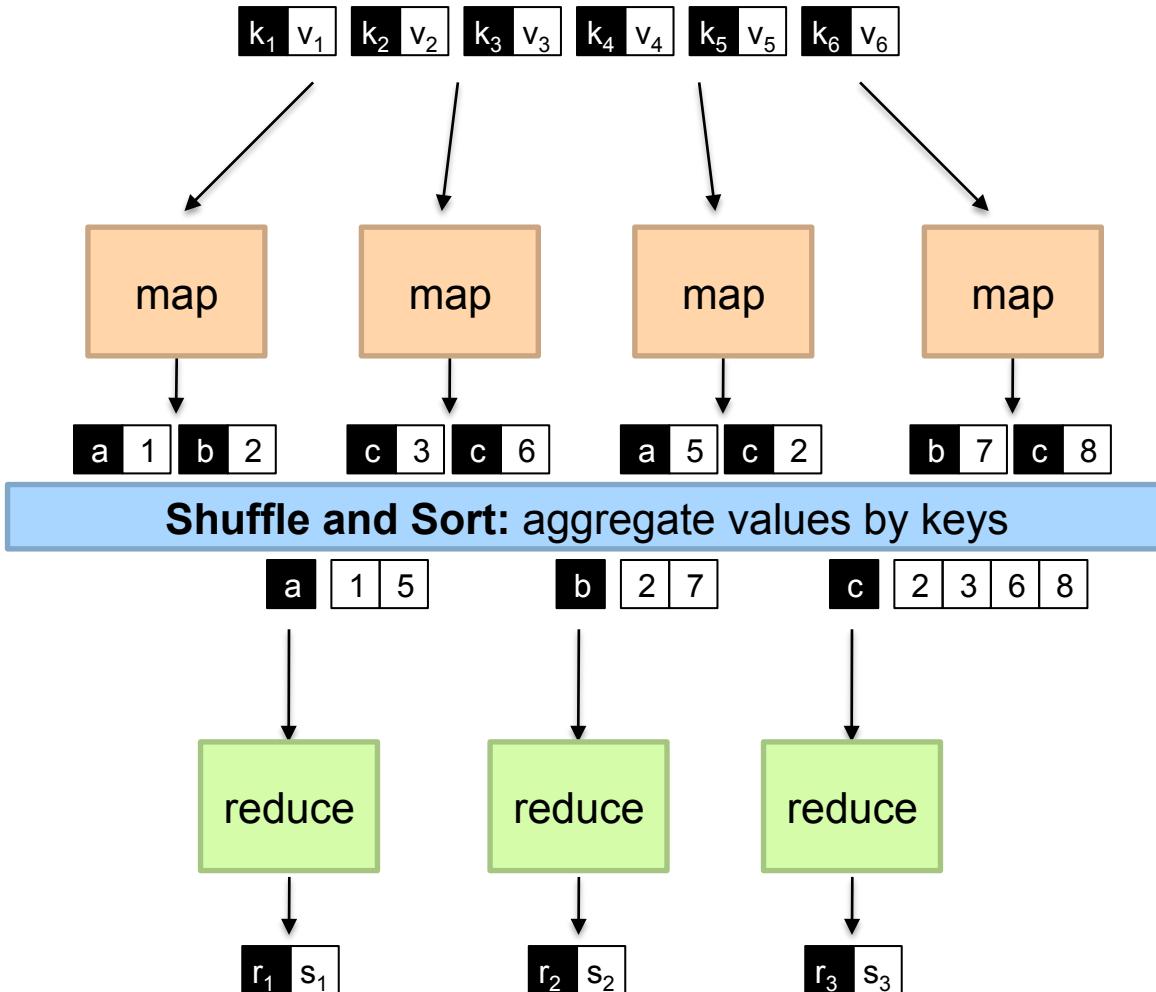
Reduce

$(key_2, val_2) \rightarrow <key_3, val_3>^*$

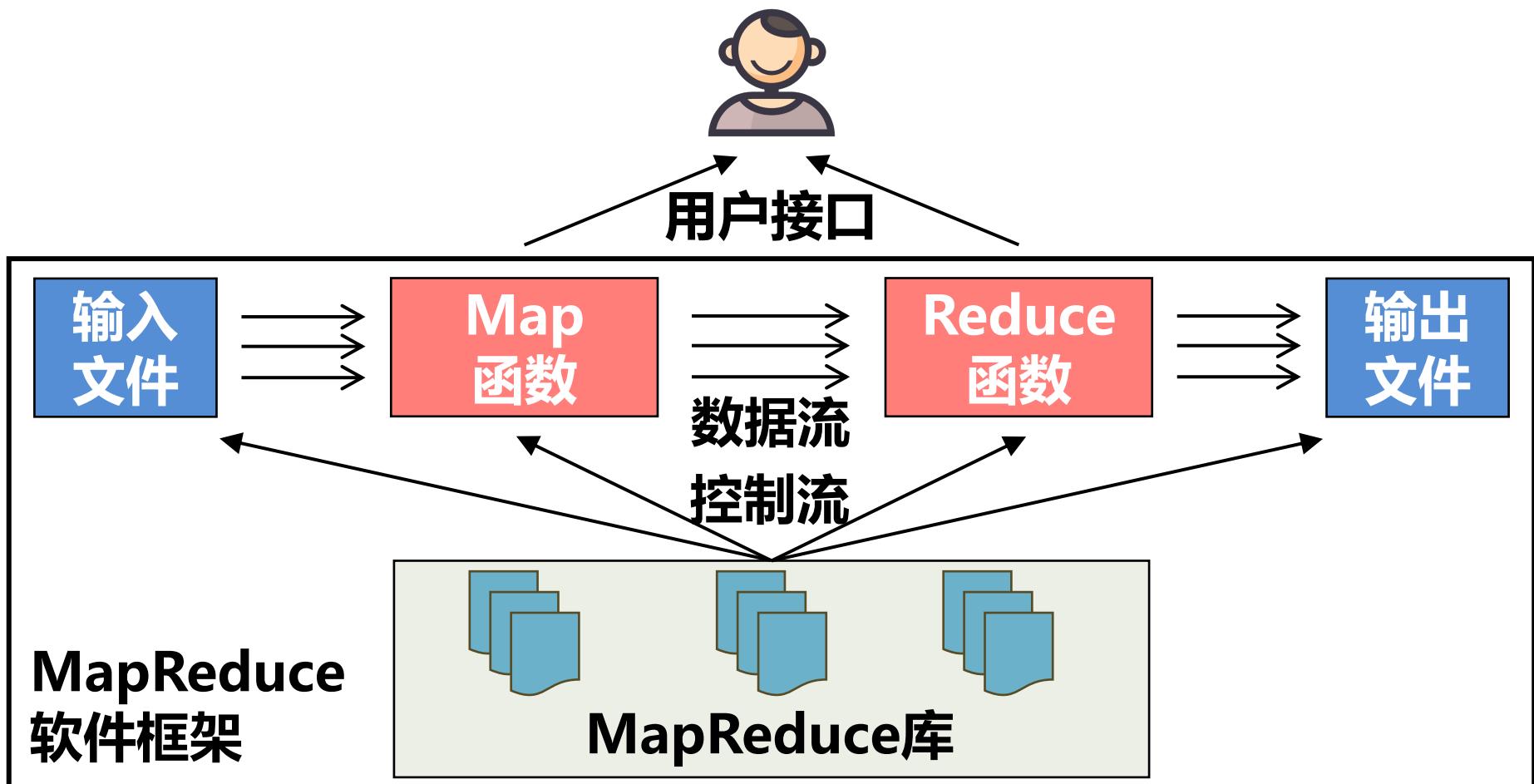
- All values with the same key are sent to the same reducer
- $<a, b>^*$  means a list of tuples in the form of (a, b)

开发人员仅需指定 Map 和 Reduce 函数，  
并行计算框架负责执行

# MapReduce



# MapReduce是一个软件框架，简化复杂编程



底层硬件基础设施 (对用户不可见)

# 回顾计数问题

Hello World Bye World  
Hello SYSU Bye SYSU  
Bye SYSU Hello SYSU

Map

Hello (1) World (1) Bye (1) World (1)  
Hello (1) SYSU (1) Bye (1) SYSU (1)  
Bye (1) SYSU (1) Hello (1) SYSU (1)

# 回顾计数问题

1

Hello (1) World (1) Bye (1) World (1)  
Hello (1) SYSU (1) Bye (1) SYSU (1)  
Bye (1) SYSU (1) Hello (1) SYSU (1)

Shuffle

2

Hello (1) Hello (1) Hello (1)  
World (1) World (1)  
Bye (1) Bye (1) Bye (1)  
SYSU (1) SYSU (1) SYSU (1) SYSU (1)

# 回顾计数问题

Hello (1) Hello (1) Hello (1)

World (1) World (1)

Bye (1) Bye (1) Bye (1)

SYSU (1) SYSU (1) SYSU (1) SYSU (1)

Reduce

Hello: (1+1+1=3)

World: (1+1=2)

Bye: (1+1+1=3)

SYSU: (1+1+1+1=4)

# MapReduce能够提供 数据并行 型计算的动态调度

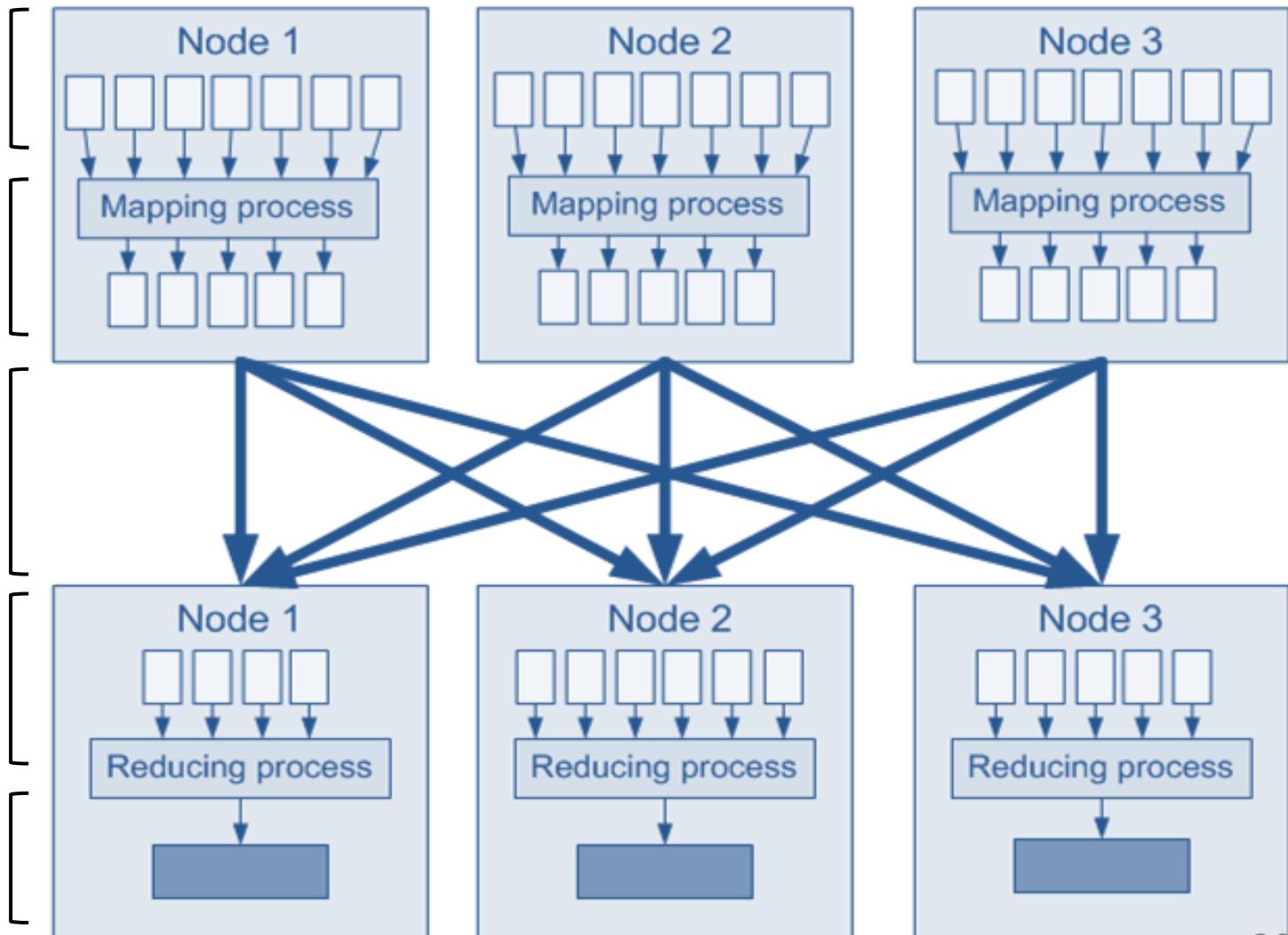
分布存储

Map  
数据映射

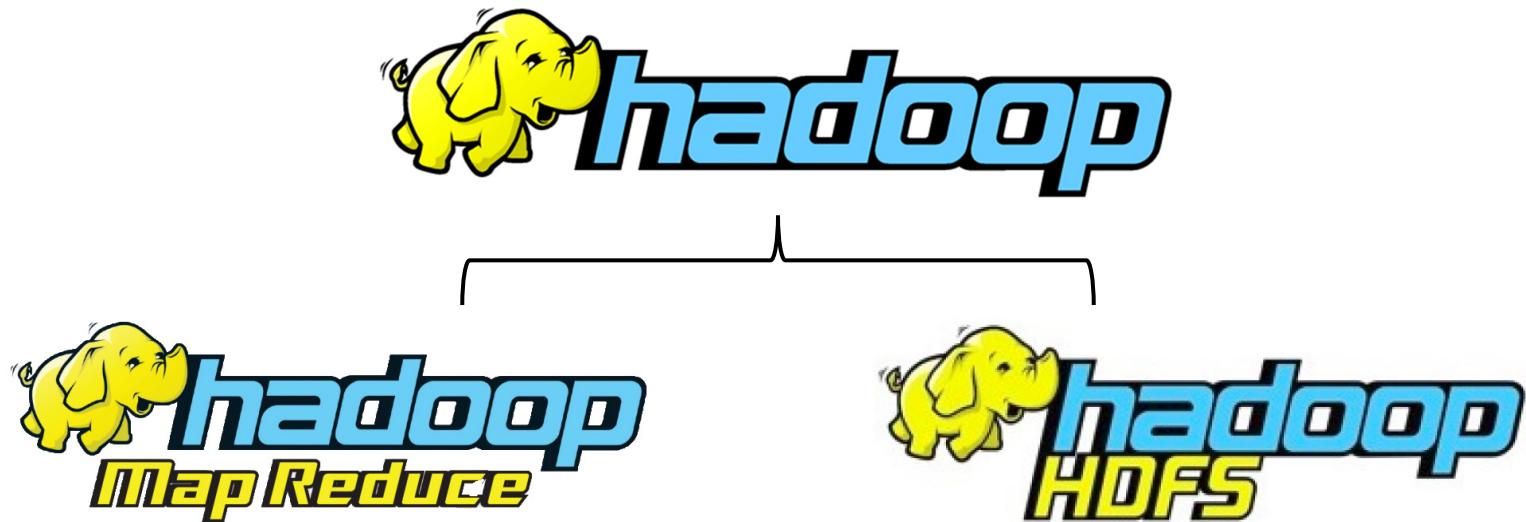
Shuffle  
数据混洗

Reduce  
数据化简

写会本地



# MapReduce 是开源框架 Hadoop 的核心



Hadoop的计算模型，提供一种基于写入-读取的并行编程模型，用于大规模数据集的并行处理

Hadoop的分布式文件系统，支持在成千上万的服务器节点上分布式存储大量的数据

The Google File System

# Fun "facts" about Jeff Dean

来自崇拜他的（前）谷歌员工们

- 编译器不会给 Jeff 警告，但 Jeff 会警告编译
- Jeff 直接用二进制写代码，他写源代码是为了给其他开发人员参考
- Jeff 写代码的速度在 2000 年底提高了 40 倍，原因是更新了 USB 2.0 的键盘
- 当 Jeff 失眠时，他的大脑用 MapReduce 数羊
- 所有的指针都指向了 Jeff
- Jeff 的手表会显示从 1970 年 1 月 1 日起的秒数，他从不迟到
- To Jeff Dean, "NP" means "No Problem"



# MapReduce 深度优化

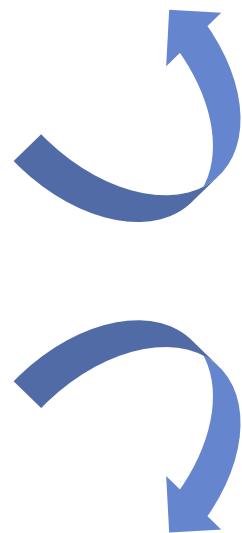
# MapReduce 深度优化

## 分布式计算的任务拖延问题

映射

计算任务

He was an  
old man  
who fished  
alone in a  
skiff in the  
Gulf Stream  
and he had  
gone...



化简

计算结果

task

task

task

task

task

task



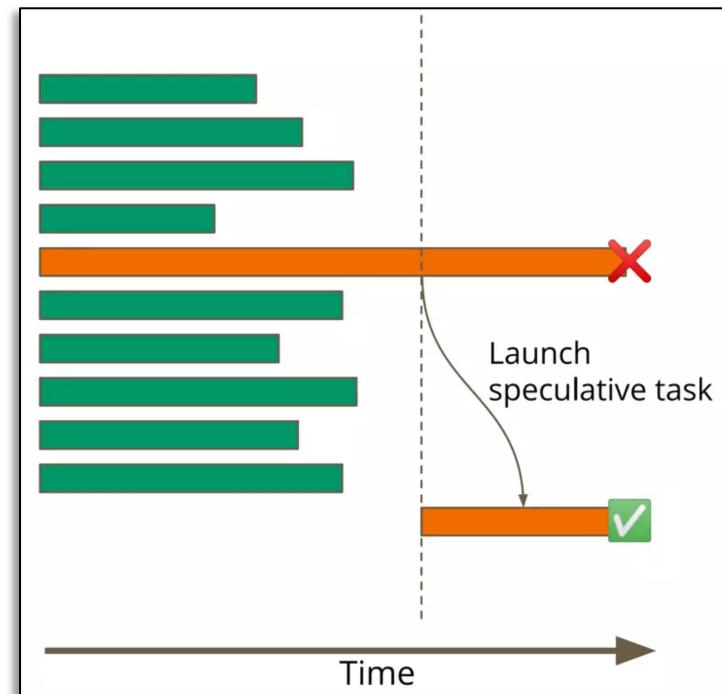
词	频
fish	3%
old	3%
man	3%
now	3%
down	2%
when	2%
line	2%

# MapReduce 深度优化

## 利用空闲节点避免应用进度滞后

MapReduce 的一个关键优点是自动处理故障，若一个节点崩溃，则在另一台机器上重新运行其任务

同样重要的是，若一个节点  
**可用但表现不佳**，称之为滞  
后节点 (straggler)，则会在  
另一台机器上运行其任务的  
副本（也称为“备份执行”），  
以更快地完成计算



# MapReduce 深度优化

## 利用空闲节点避免应用进度滞后

如何选择任务进行备份执行?

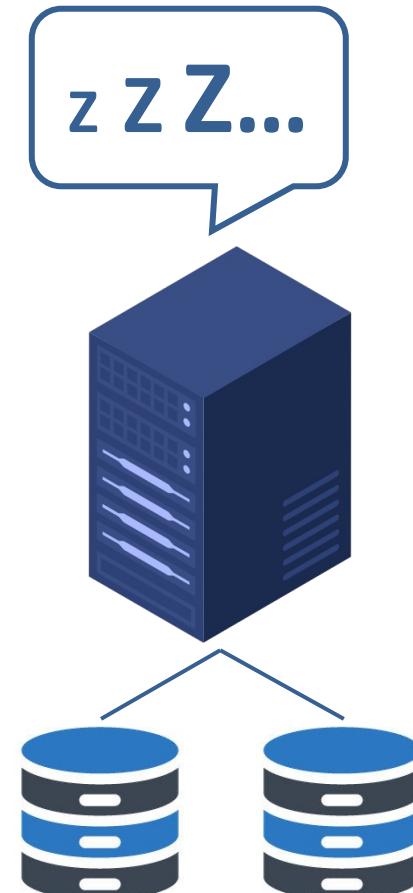
已半途失败的任务

未开始执行的任务

推测执行

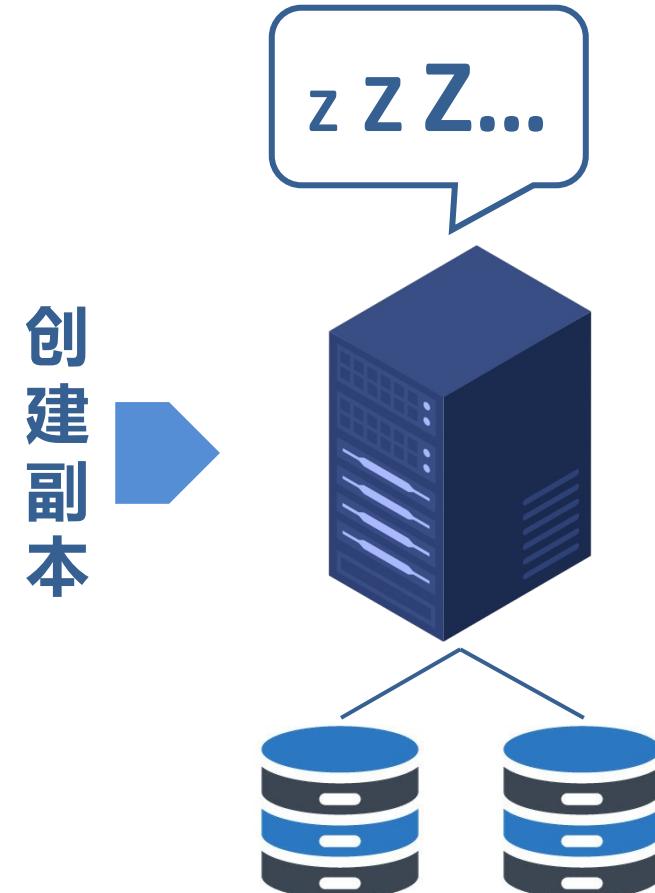
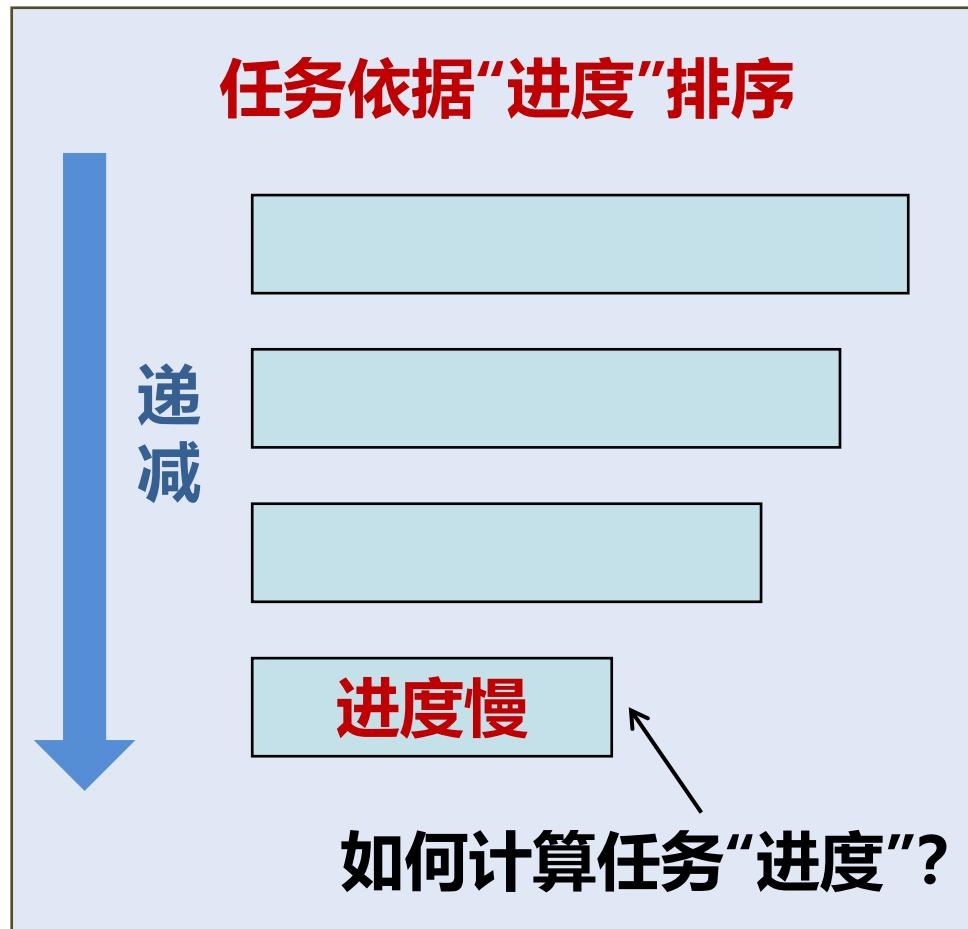
优先级

创建副本



# 如何推测哪些任务需要备份执行？

利用空闲节点避免应用进度滞后



# 深度优化

## 推测执行：Map 类任务打分

依据已完成输入的数据的比重

Data loading...



A 进度分：1/3



B 进度分：2/3

# 深度优化

## 推测执行：Reduce 类任务打分

依据以下三个部分的进度

1/3分数比重

数据复制阶段

1/3分数比重

数据混洗阶段

1/3分数比重

数据计算阶段

A

进行到一半的复制阶段

A 进度分： $1/3 * 50\% = 1/6$

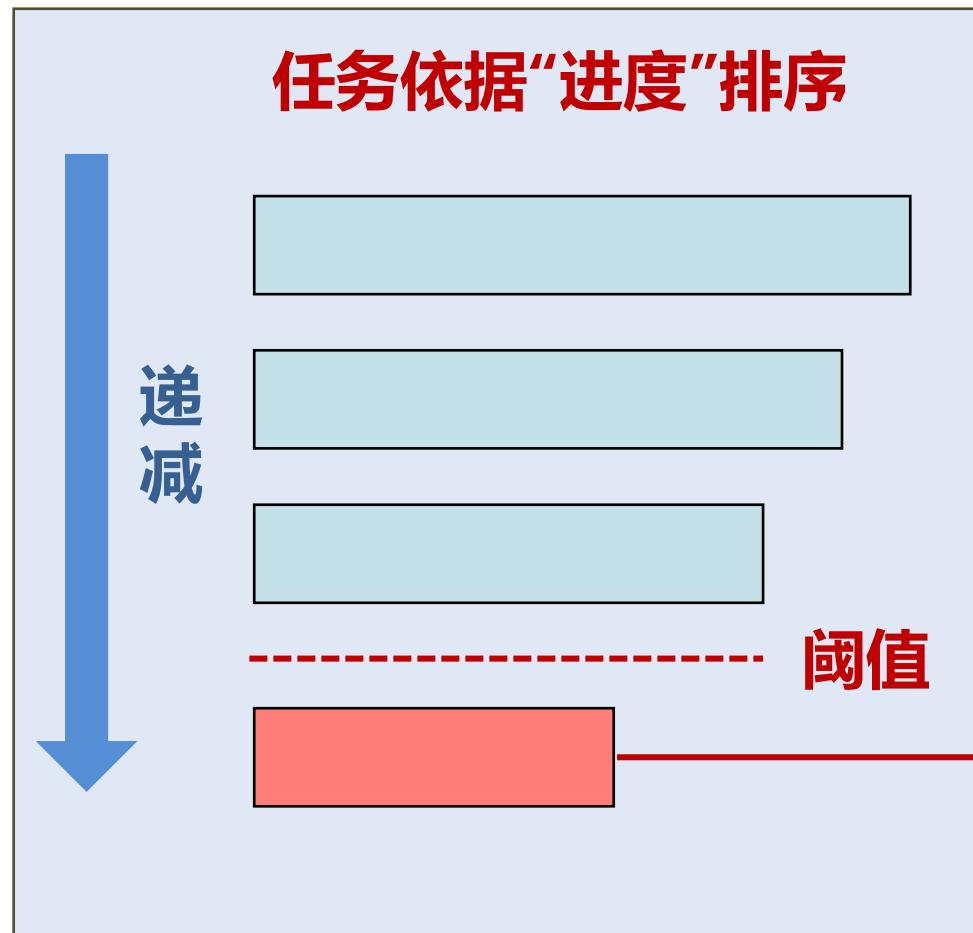
B

进行到一半的混洗阶段

B 进度分： $1/3 + 1/3 * 50\% = 1/2$

# MapReduce 深度优化

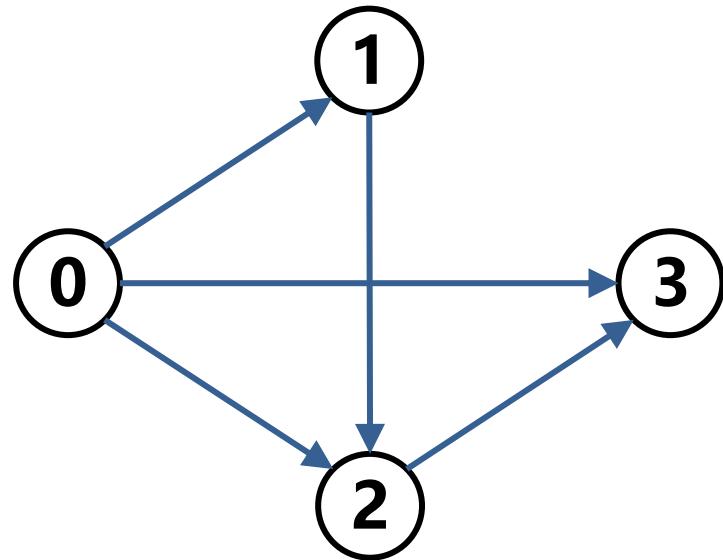
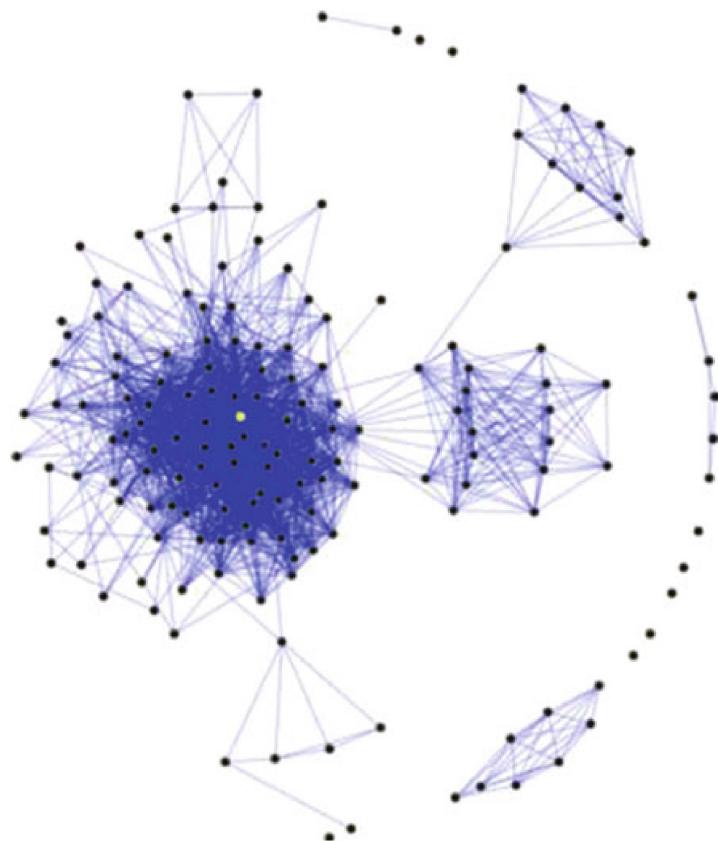
## 基于排序和阈值的推测执行



调动idle节点

# 开放思考

**图结构处理：数据计算成本小，数据访问开销大**



# 开放思考

## 图结构处理的GAS模型

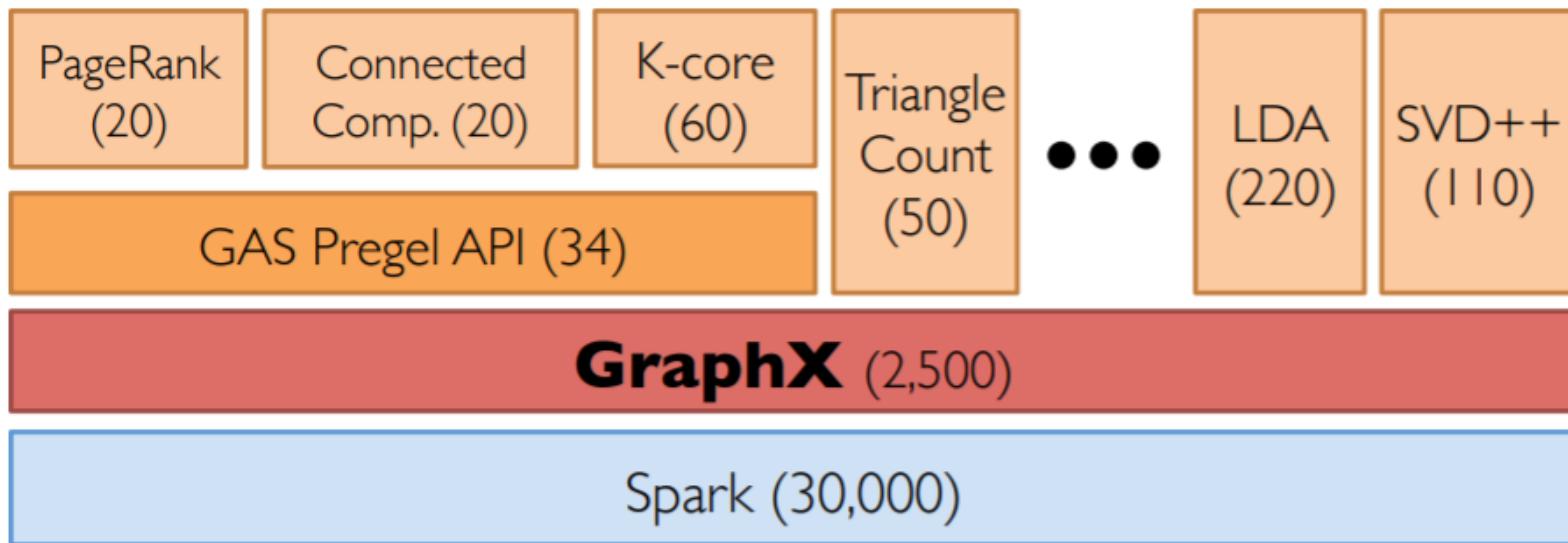
The GAS model represents three conceptual phases of a vertex-program: Gather, Apply, and Scatter

**Gather:** information about adjacent vertices and edges is collected

**Apply:** updates the value of the central vertex

**Scatter:** uses the central vertex to update the data on adjacent edges

# Spark 和 GraphX



GraphX is a thin layer on top of the Spark general-purpose dataflow framework (lines of code)

# 并行计算的长尾问题

# **99th Percentile Latency at Scale with Apache Kafka**

Technology < Apache Kafka

FEB 25, 2020 READ TIME: 23 MIN

Fraud detection, payment systems, and stock trading platforms are only a few of many Apache Kafka® use cases that require both *fast* and *predictable* delivery of data. For example, detecting fraud for online banking transactions has to happen in real time in order to deliver business value, without adding more than 50–100 ms of overhead to each transaction in order to maintain a good customer experience.

**同一起跑线，速度差数倍？**

## 《连线》杂志主编克里斯·安德森和他的“长尾理论”



# 计算中的长尾



George Zipf  
哈佛大学语言学家

“在自然语言的语料库里，一个单词出现的**频率**与它在频率表里的**排名**成**反比**”

乔治·齐夫【美】

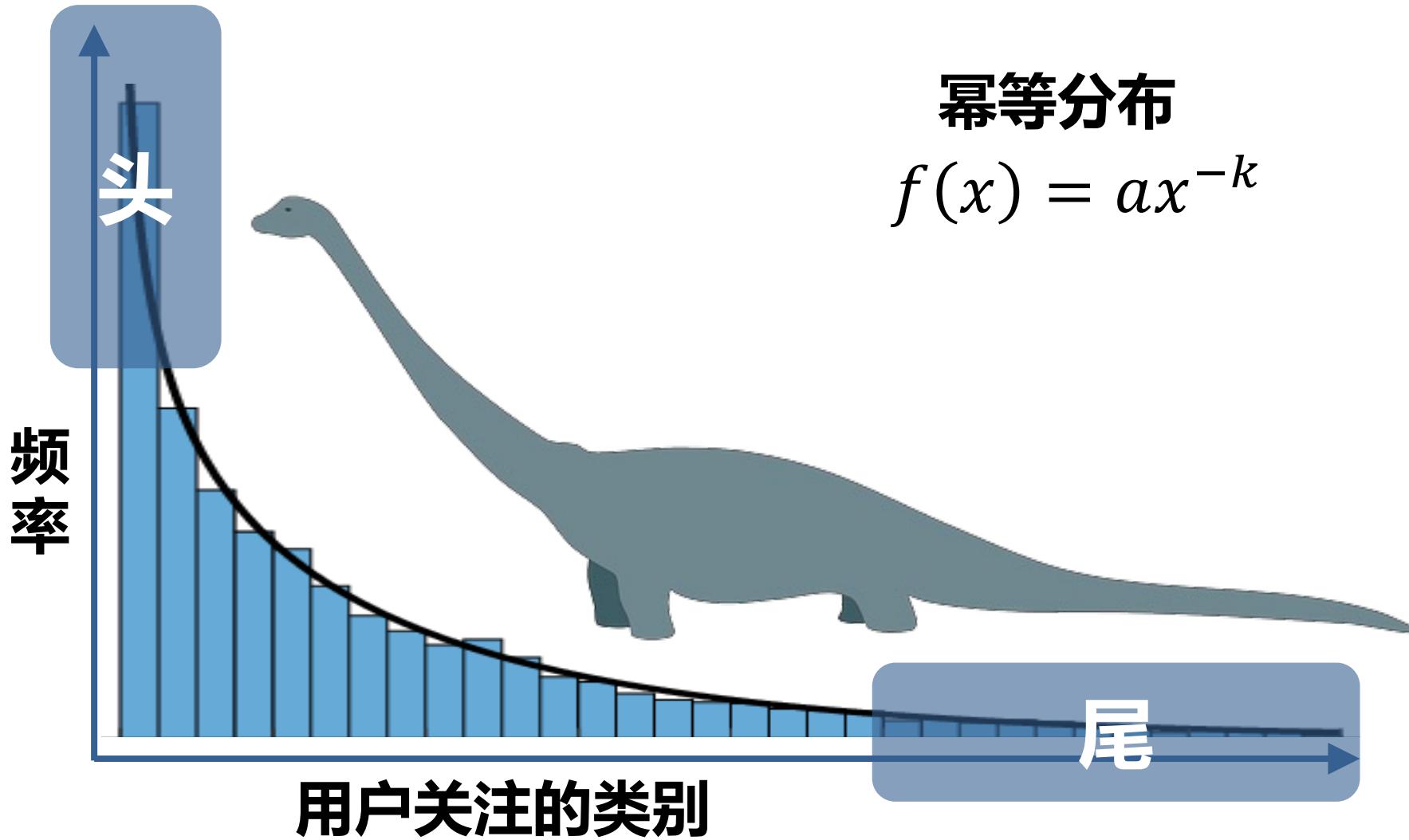
CURRENT WORD

WORDCOUNT

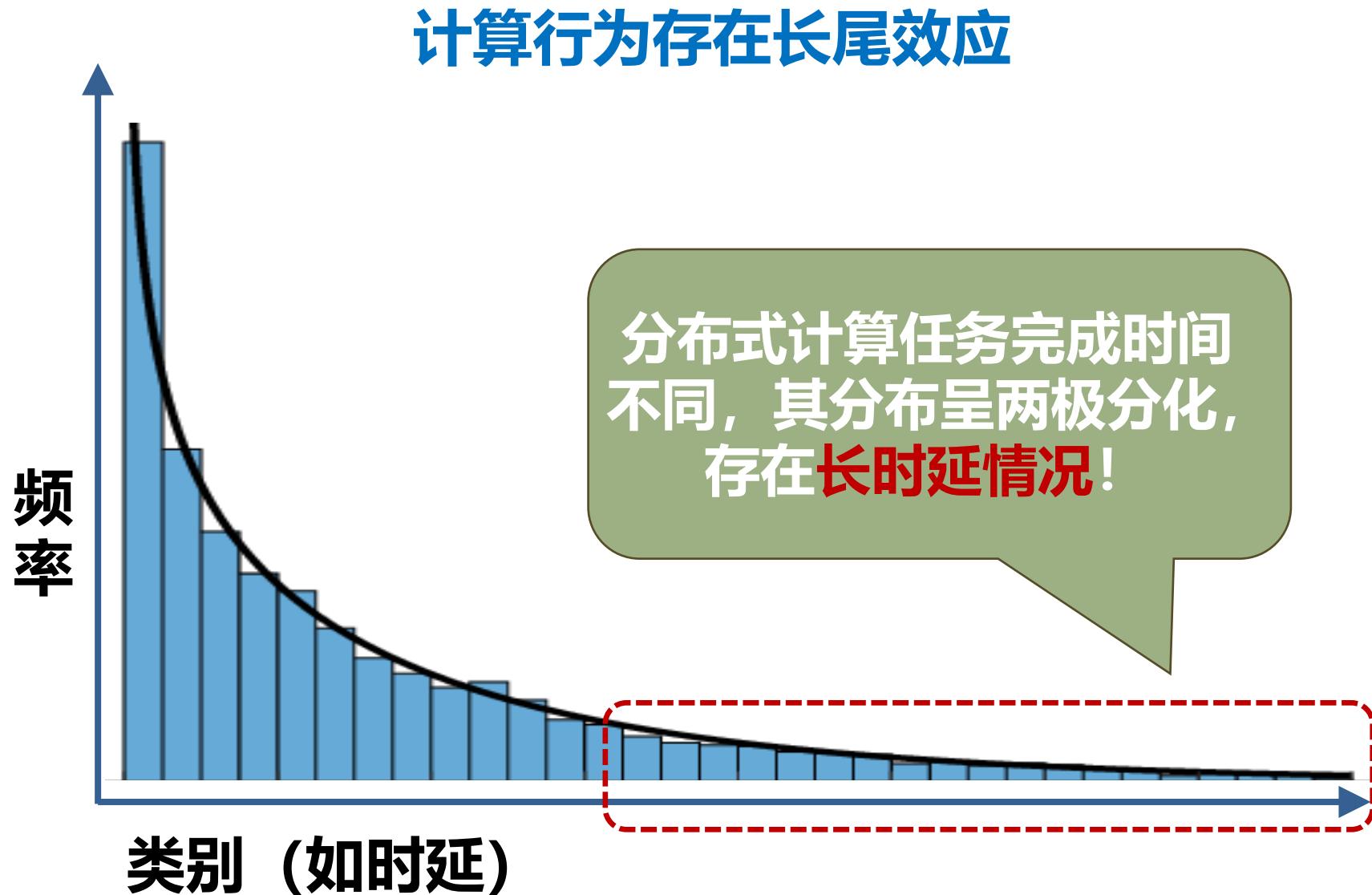
FIND WORD:  BY RANK:  REQUESTED WORD:  
RANK:

WORDS IN ARCHIVE  
ABOUT WORDCOUNT

# 计算中的长尾

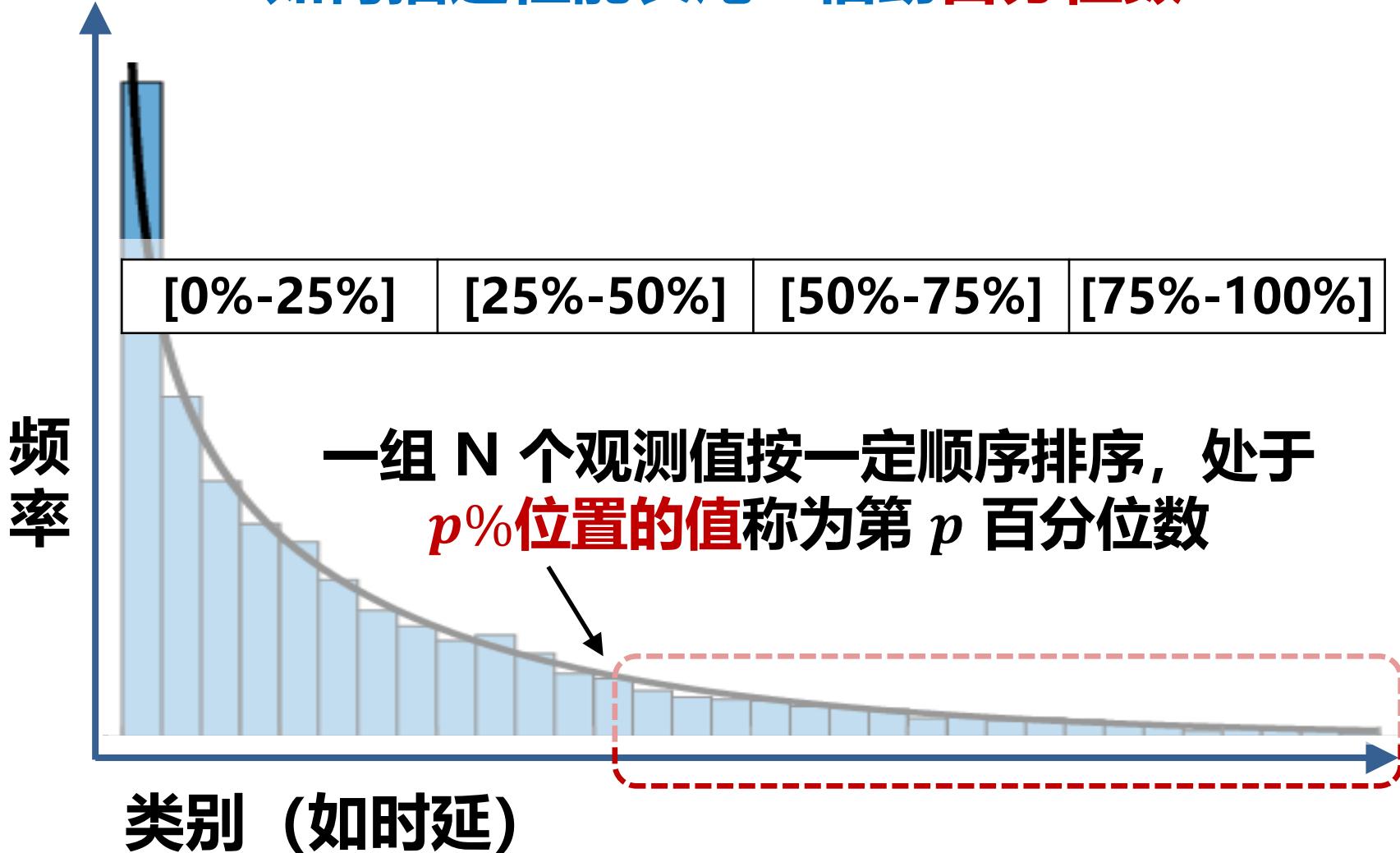


# 计算中的长尾

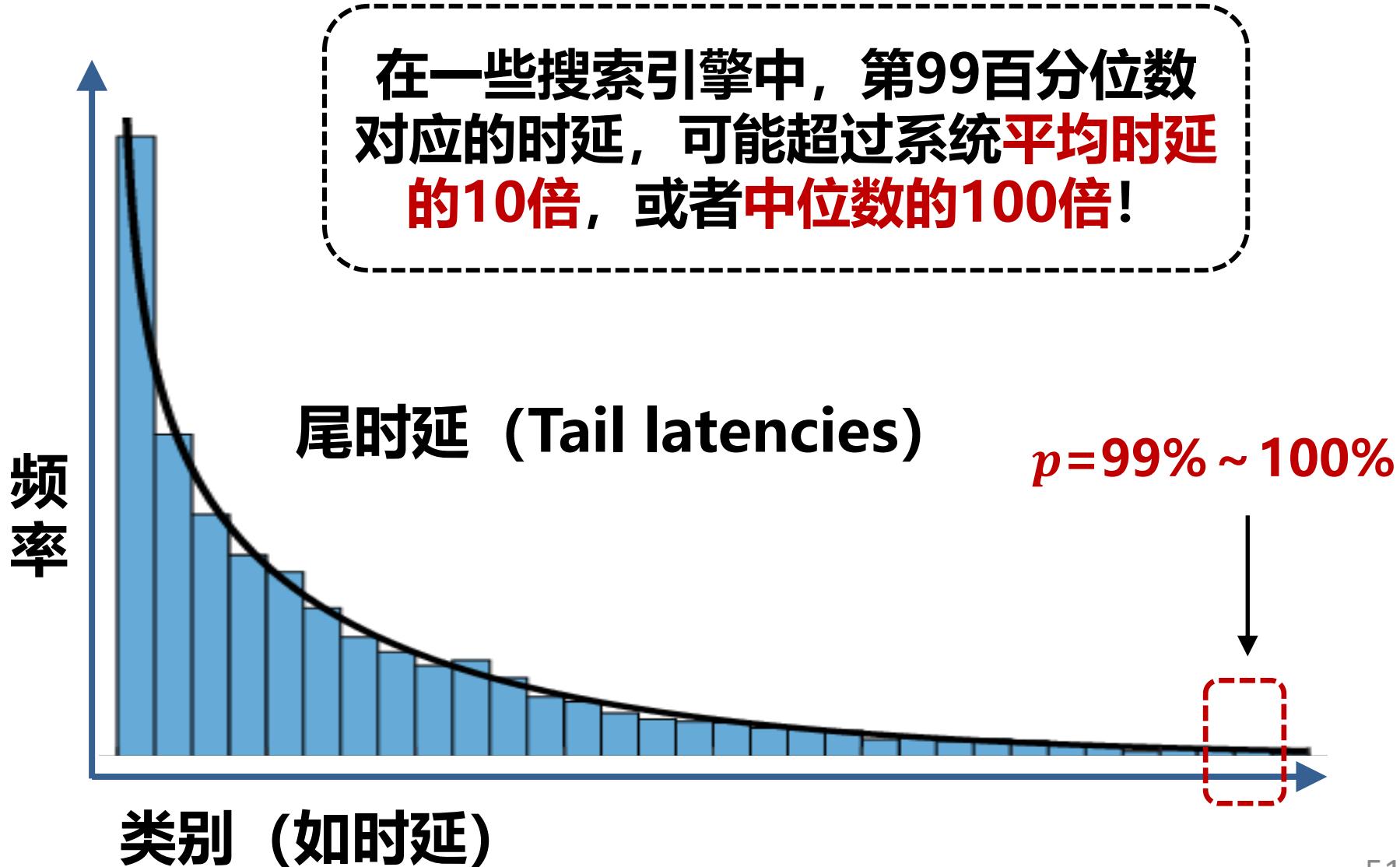


# 计算中的长尾

如何描述性能长尾：借助百分位数



# 计算中的长尾



# 长尾背后的原因

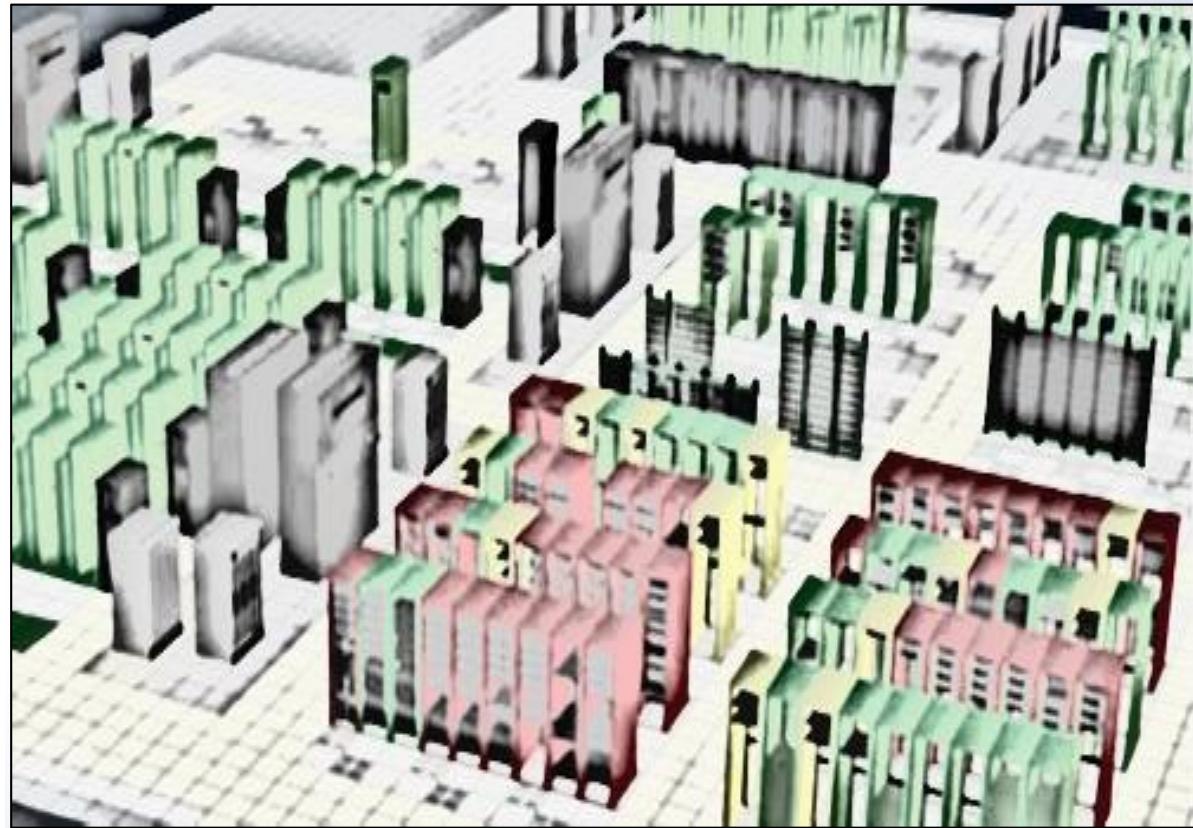
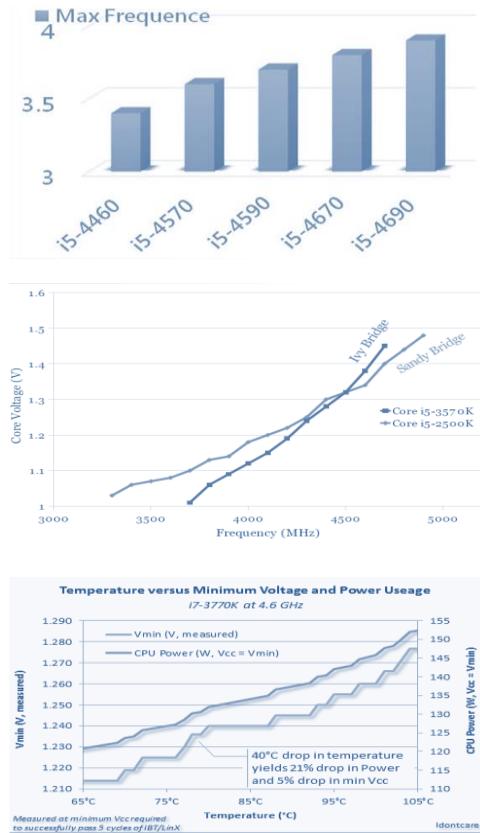
云计算中的性能长尾问题并不罕见



# 长尾背后的原因

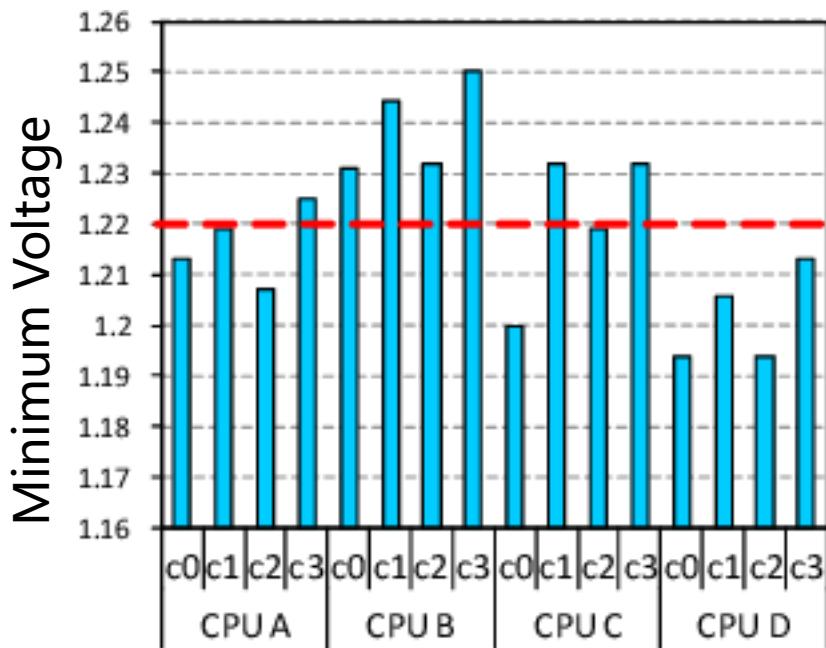
存在不同时期、规格、和运行环境的设备

速度差异 功耗差异 散热差异

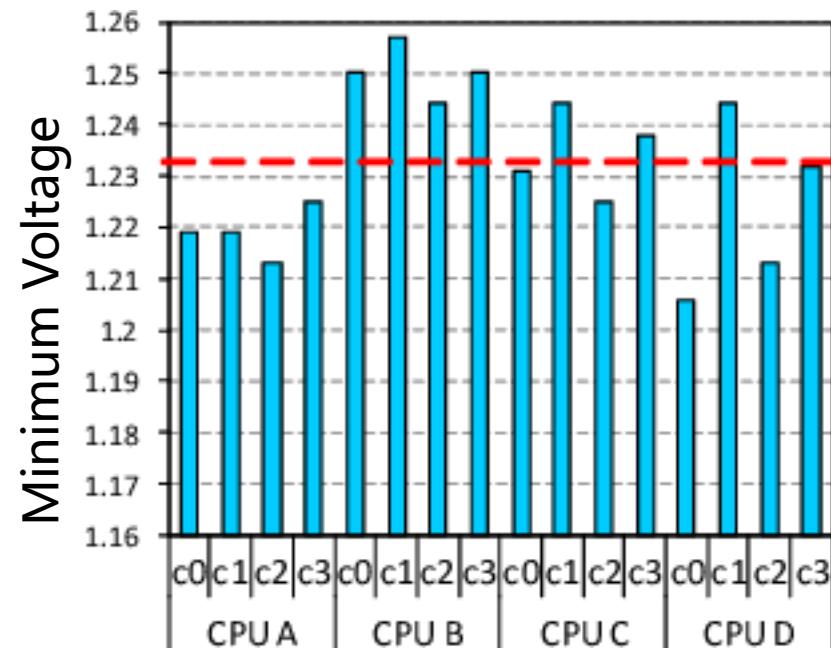


# 长尾背后的原因

同一批次的计算设备，其他特征也会存在差异



集成显卡关闭状态



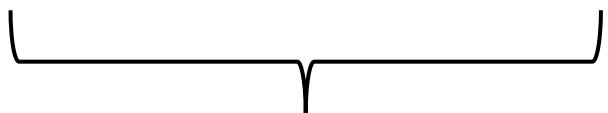
集成显卡打开状态

# 长尾背后的原因

网络通讯拥塞

计算资源竞争

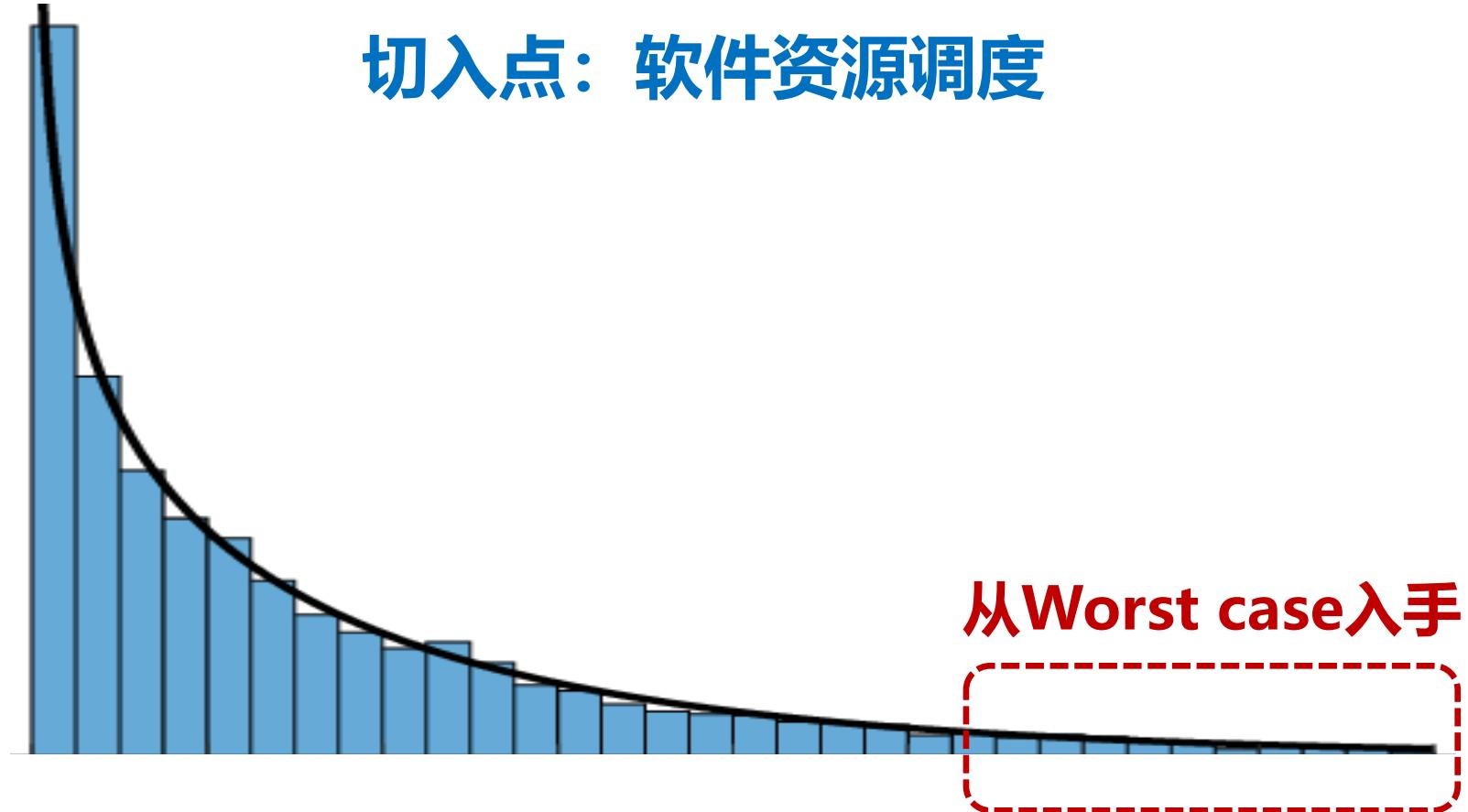
设备特征差异



具有不确定性，但可能缓解

难以避免

# 改善长尾问题



# 改善长尾问题

DOI:10.1145/2408776.2408794

**Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.**

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

# The Tail at Scale



# 改善长尾问题

口任务的特征

**短任务**

**长任务**

0ms

50ms

150ms

哪个更容易  
出现长尾?

**少任务**

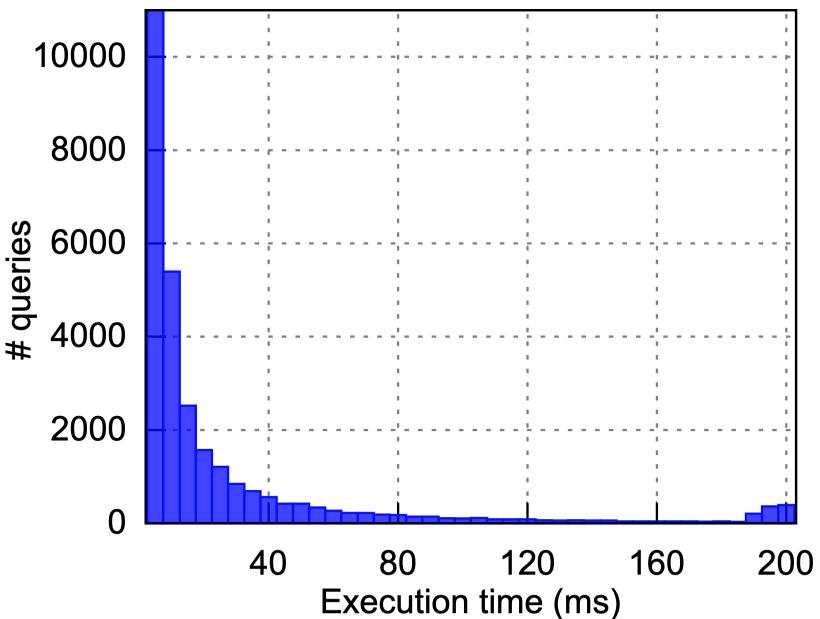
**多任务**

0ms

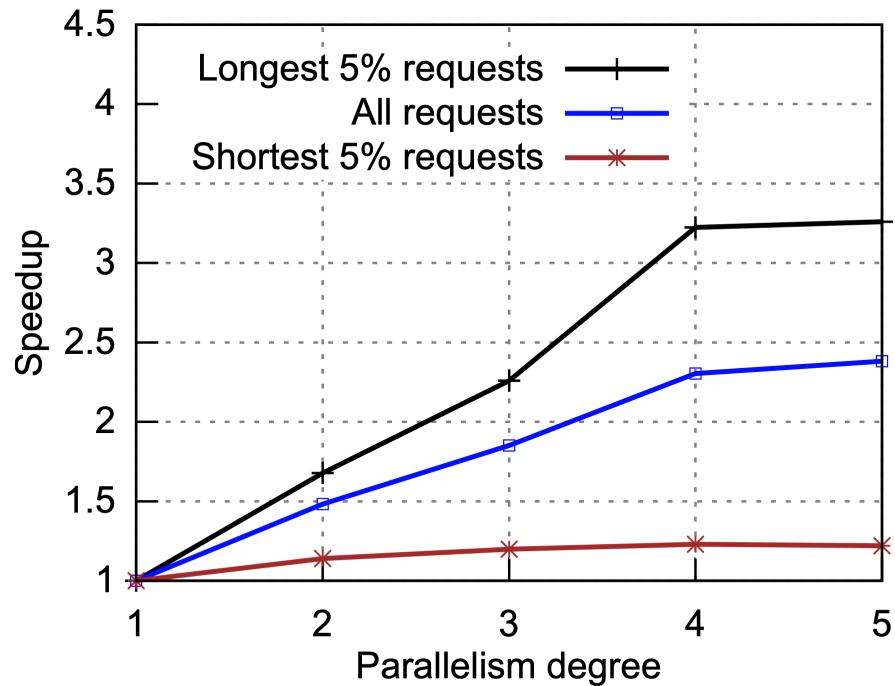
哪个更容易  
出现长尾?

# 长短任务的长尾特征

□ Bing demand distribution and average speedup



(a) Sequential execution time histogram of 30K requests

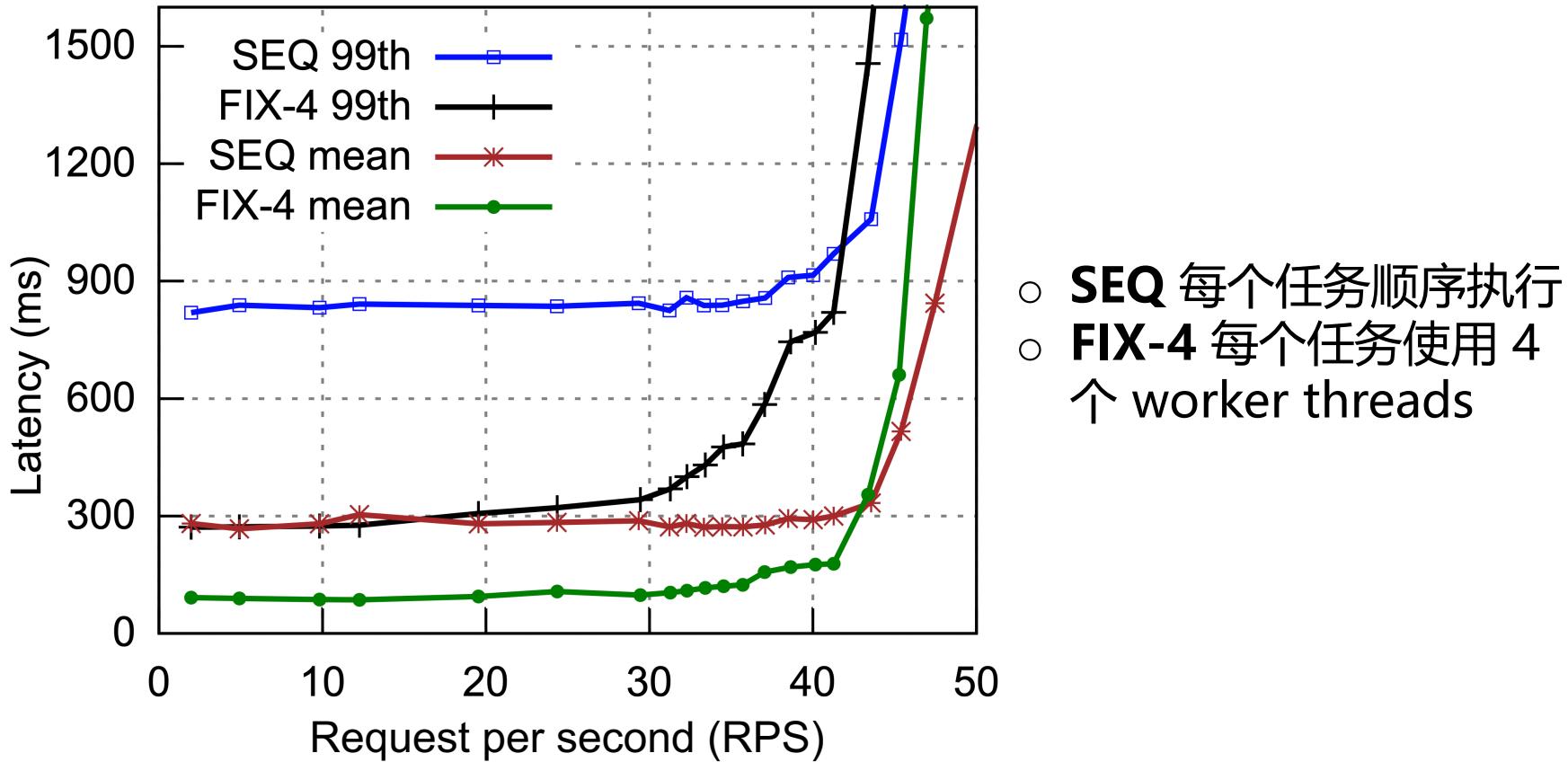


(b) Average speedup

提高长任务的并行度获得的加速比更明显

# 多少任务的长尾特征

## □ Effect of **fixed parallelism** on latency in Lucene



相同并行度下，**多任务更容易出现长尾**

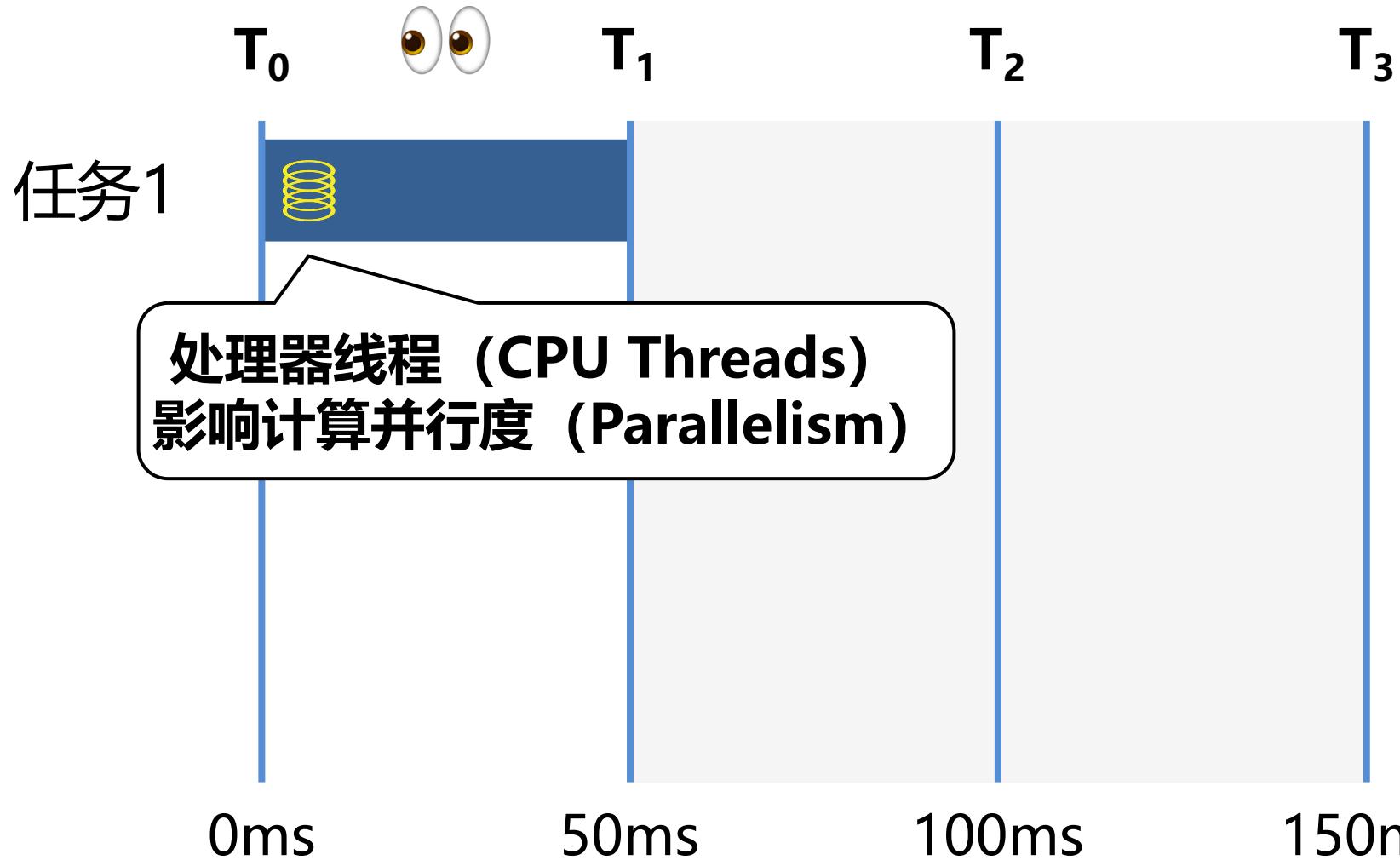
Increase the probability that short requests will finish with less parallelism, which saves resources, while assigns long requests more parallelism, which reduces tail latency.

问题是，我们无法预测任务的长短！

Few-to-Many (FM) 漸增式调度算法原理

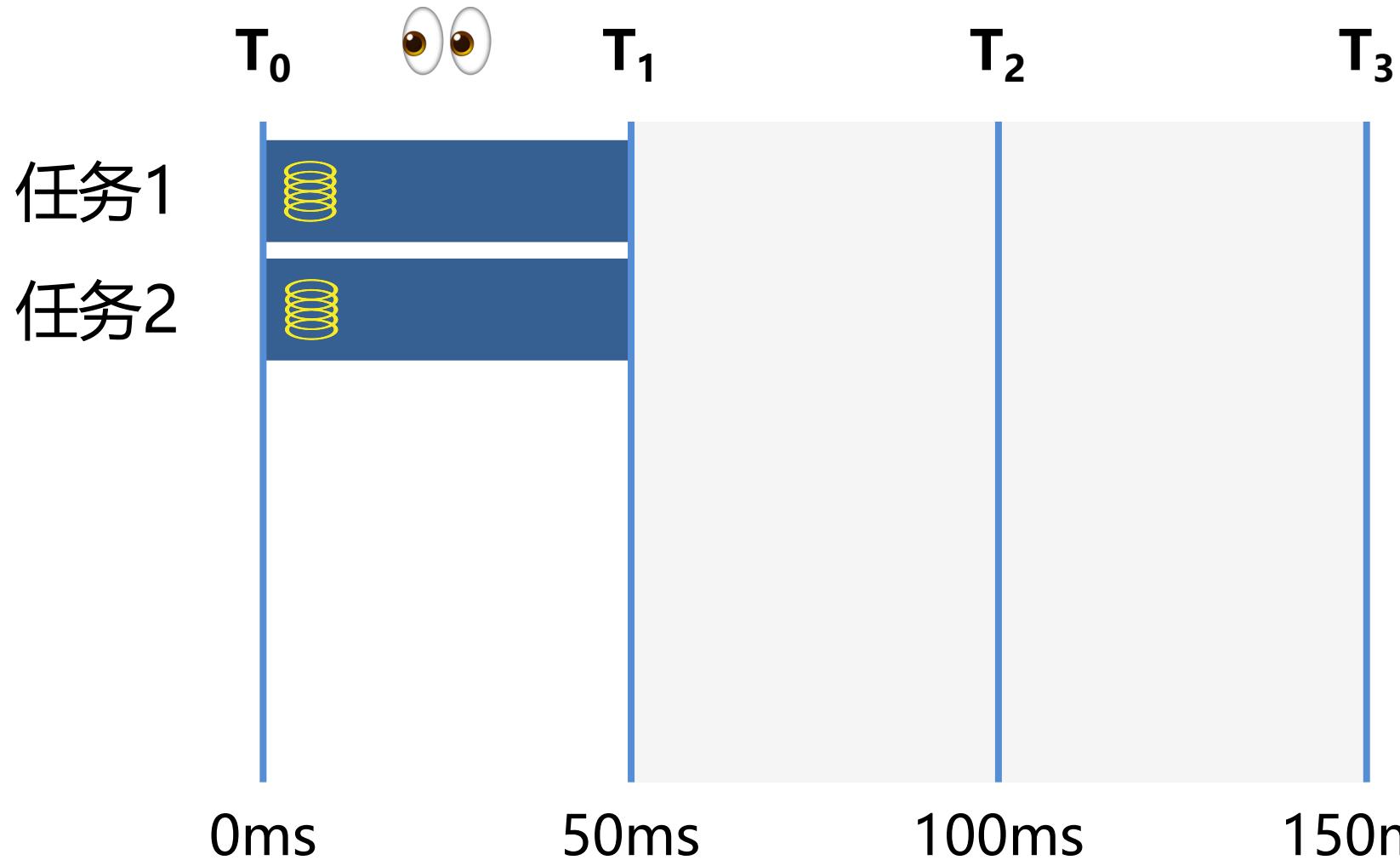
# 改善长尾问题

## Few-to-Many (FM) 演增式调度算法



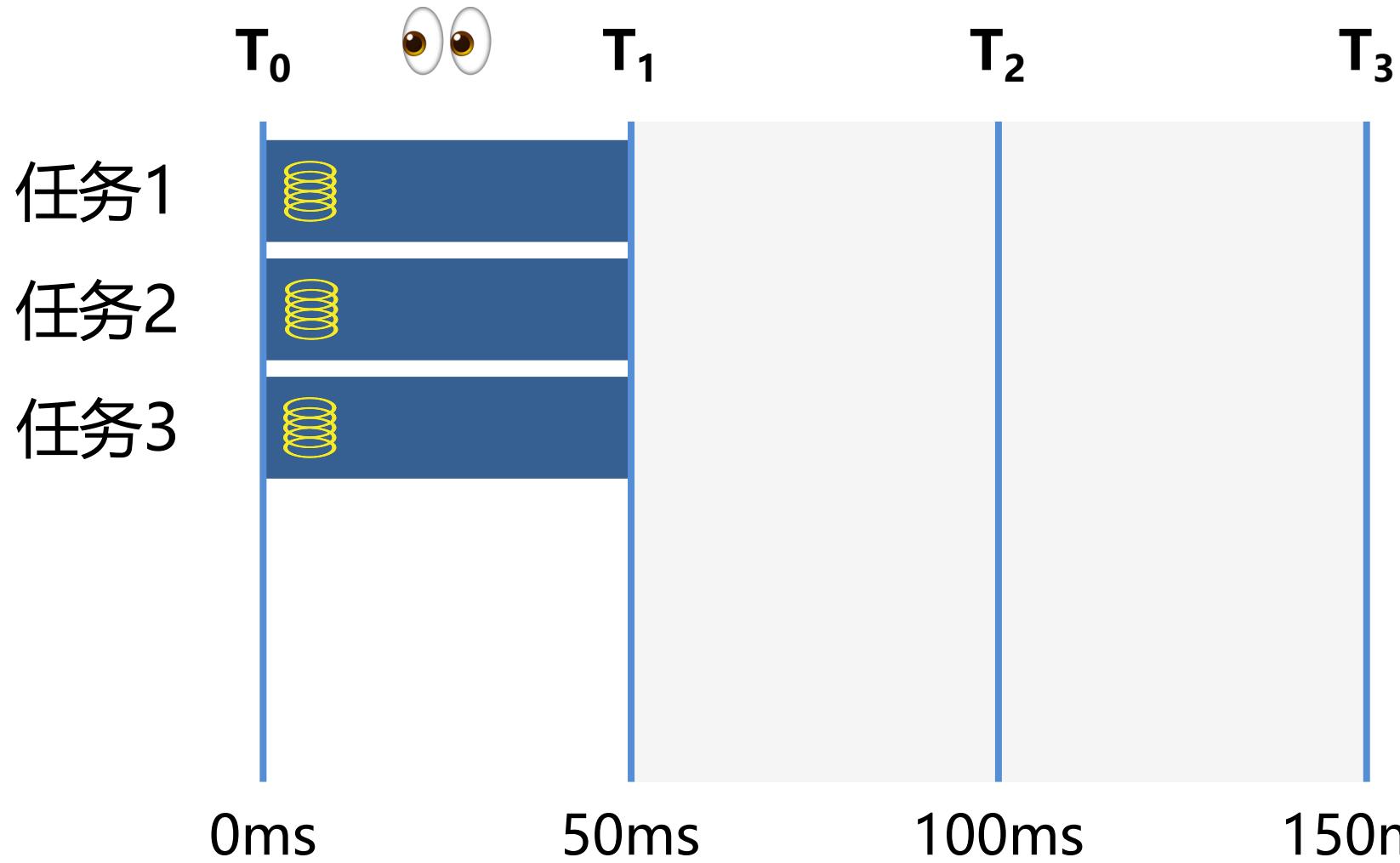
# 改善长尾问题

## Few-to-Many (FM) 漸增式调度算法



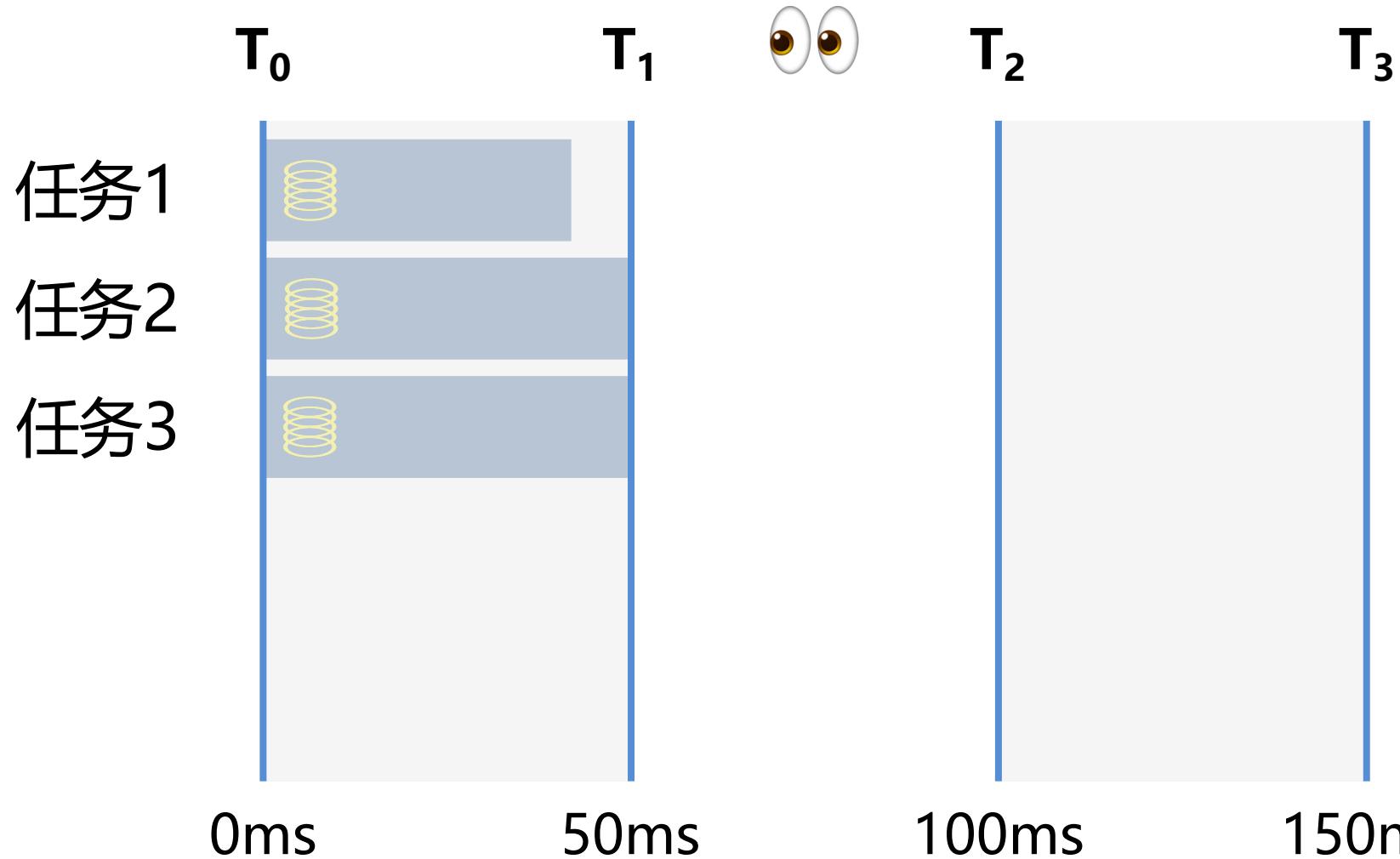
# 改善长尾问题

## Few-to-Many (FM) 漸增式调度算法



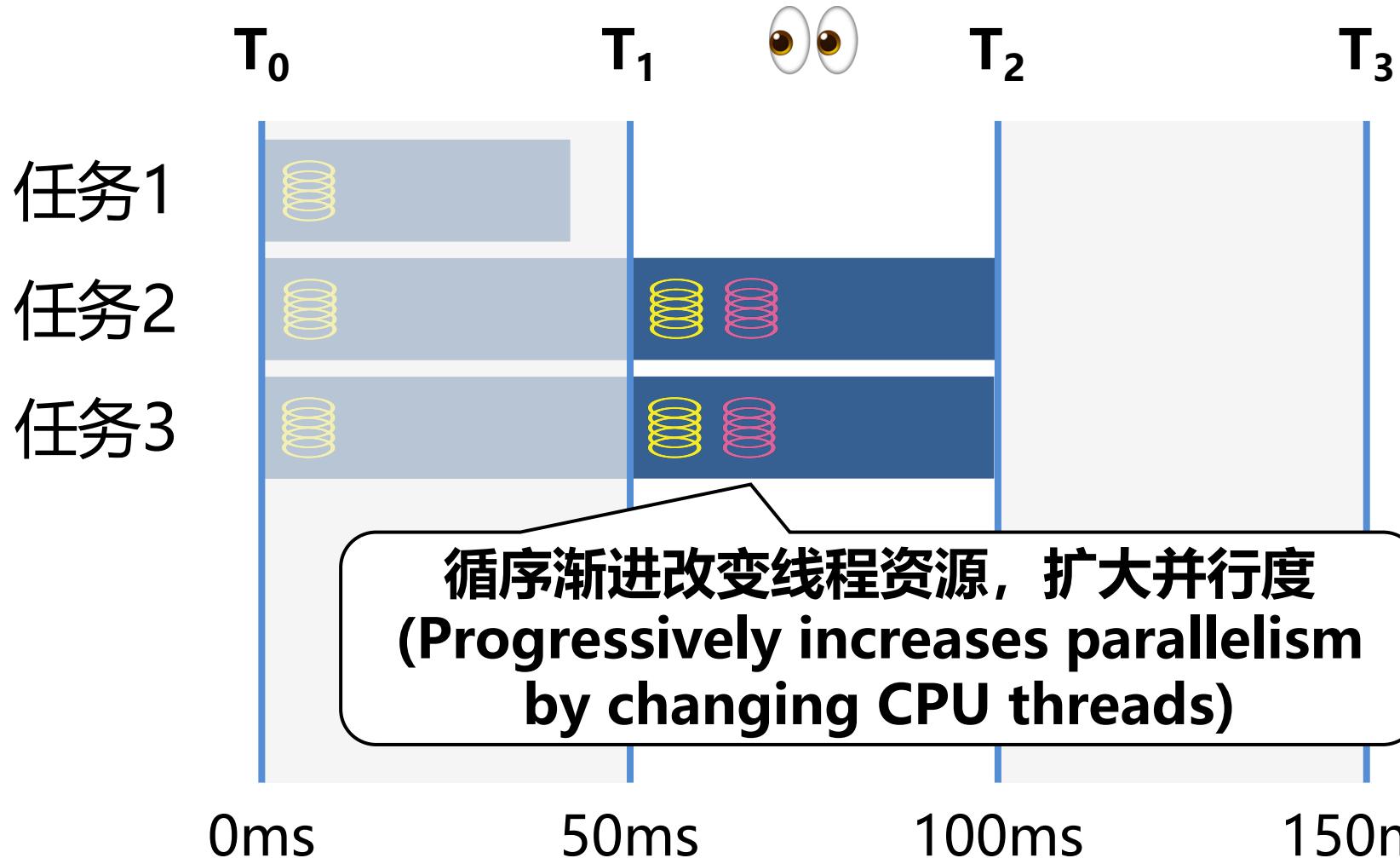
# 改善长尾问题

## Few-to-Many (FM) 漸增式调度算法



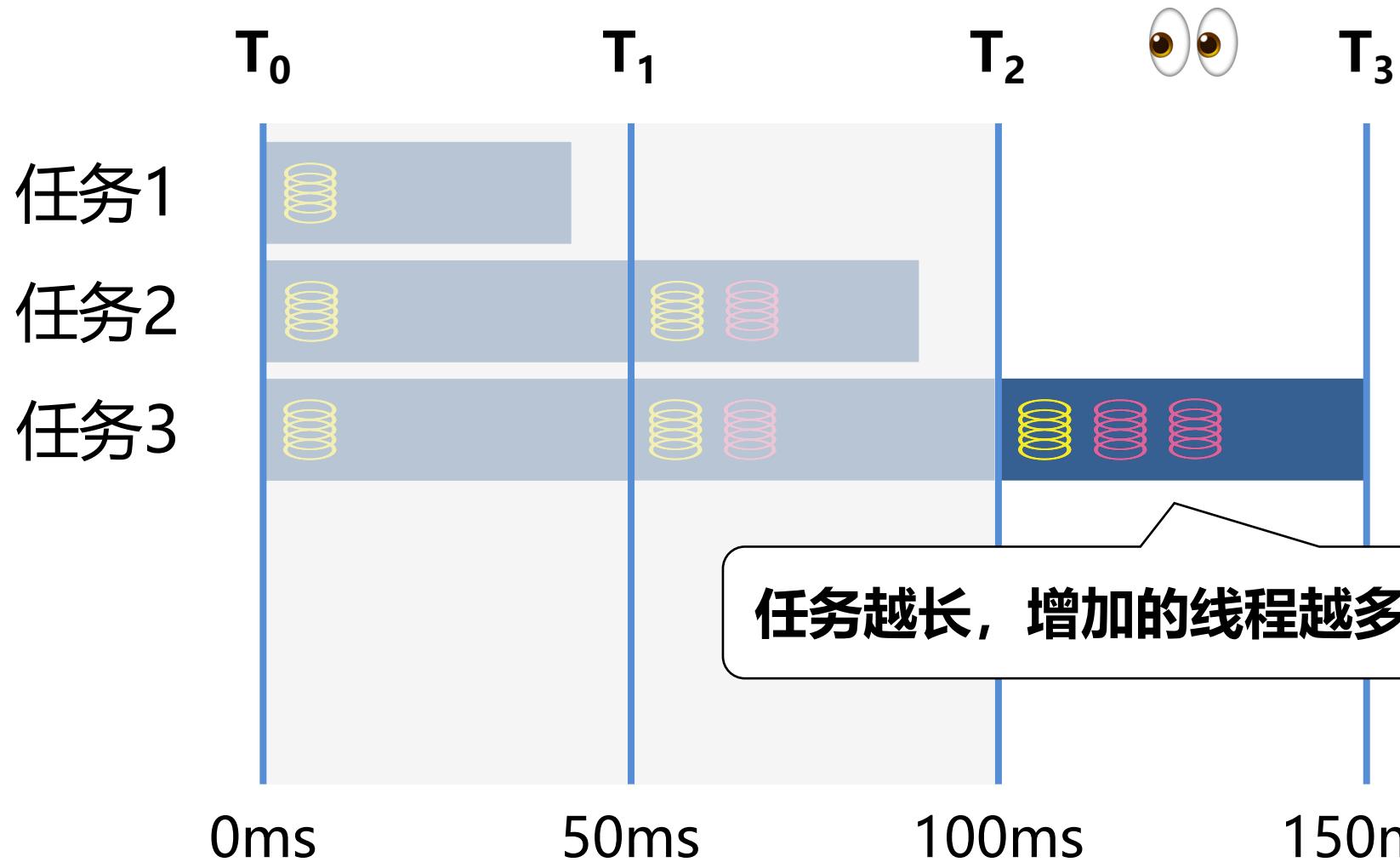
# 改善长尾问题

## Few-to-Many (FM) 漸增式调度算法



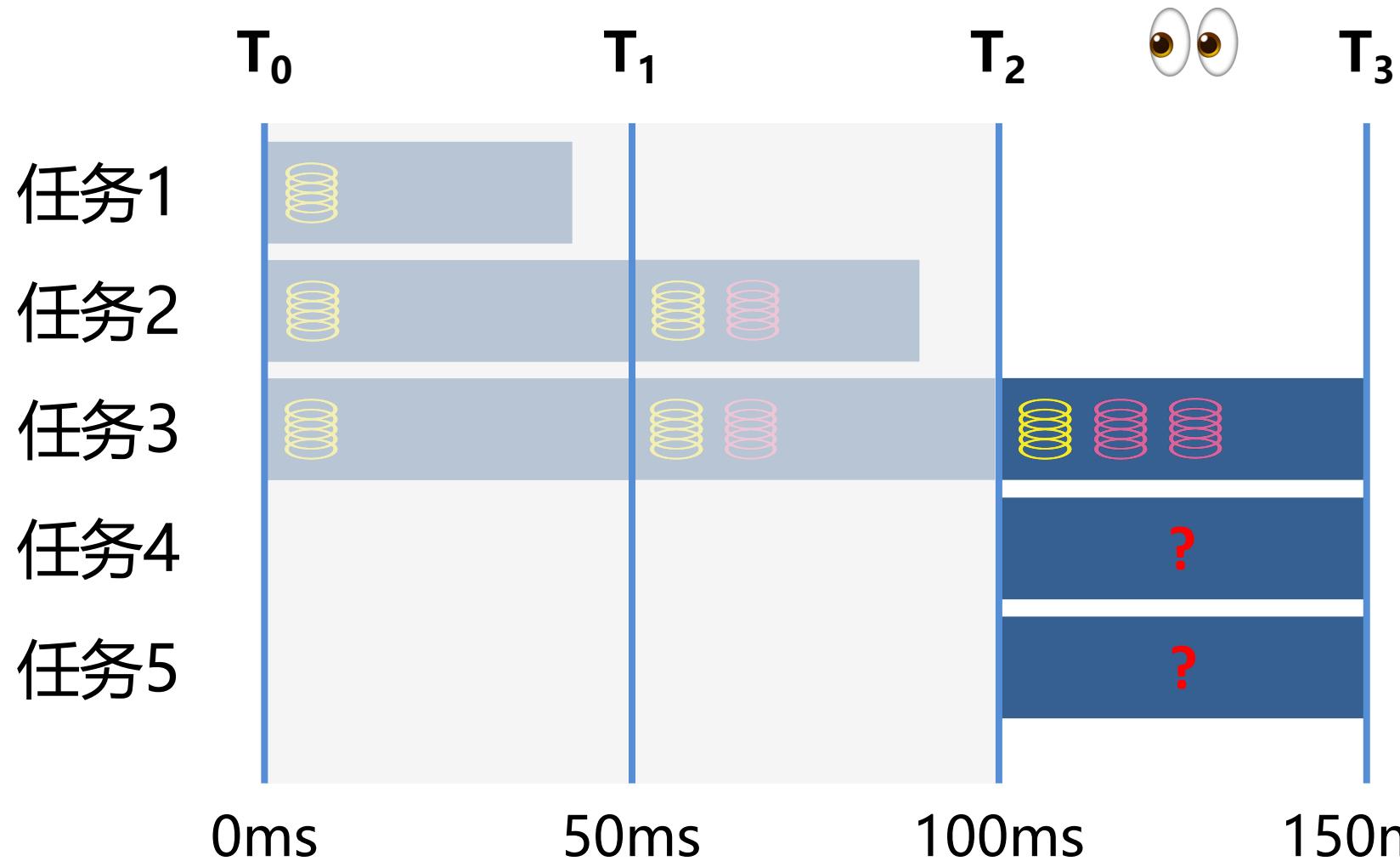
# 改善长尾问题

## Few-to-Many (FM) 漸增式调度算法



# 改善长尾问题

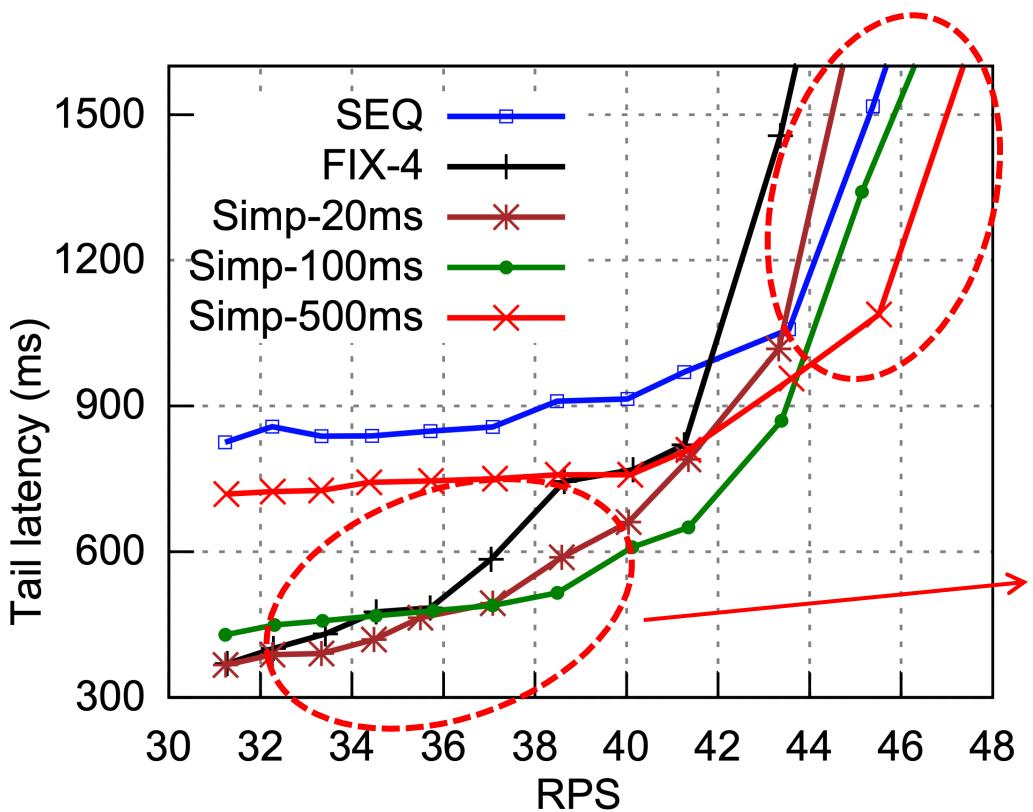
## Few-to-Many (FM) 漸增式调度算法



# 改善长尾问题

## □ Few-to-Many (FM) 渐增式调度算法最简单版本

Simply increase parallelism **periodically**, e.g., add one thread to each request after a fixed time interval.



- **Simp-Xms** 从1个worker thread开始，每隔 X ms 增加1个 worker thread

问题：难以选择合适的 interval，不同 interval 在不同 RPS 下有不一样的表现

# 改善长尾问题

## Few-to-Many (FM) 演增式调度算法实现

Offline阶段

基于历史数据构建 Interval Table

Online阶段

通过查询 Interval Table 实时调整并发度

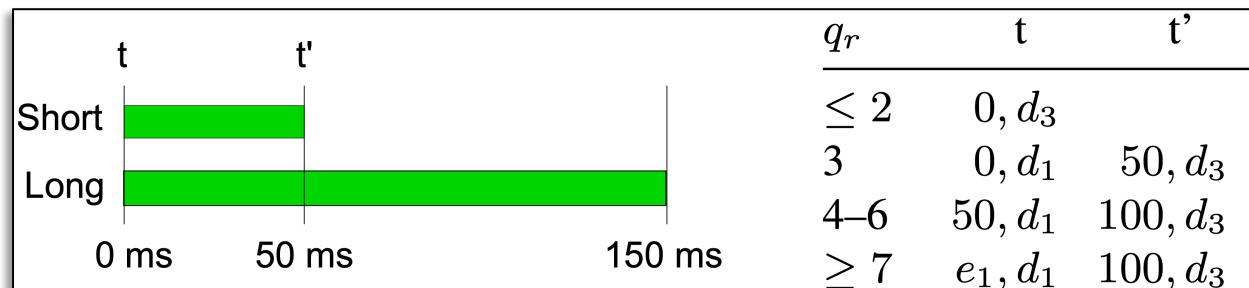
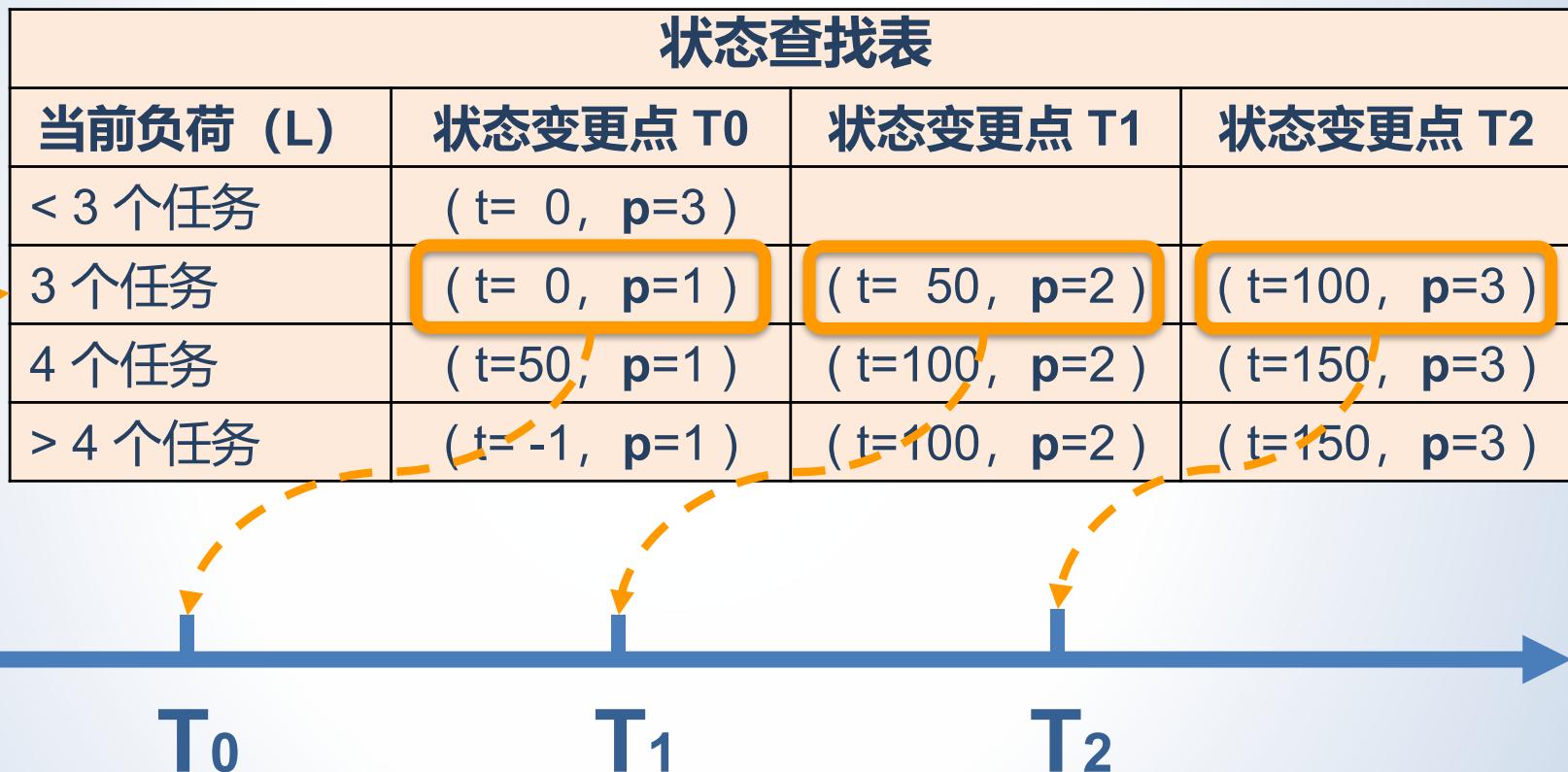


Figure 5: Simple workload and interval table for 50 ms intervals ( $t = 0, t' = 50$ ), number of requests ( $q_r$ ), parallelism degree ( $d_j$ ), for 6 cores with speedups  $s(2) = 1.5, s(3) = 2$ .

# 改善长尾问题

## Few-to-Many (FM) 演增式调度算法实现



FM: 将计算资源优先给予**低负载、长任务**的情况

# 改善长尾问题

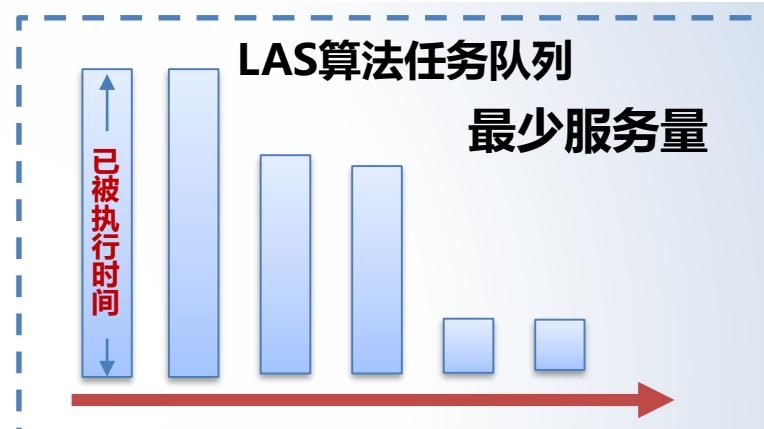
## 其他典型调度算法



Shortest Remaining Processing Time (SRPT) scheduling policy

SPRT 在平均响应时间方面可达最优

SPRT的关键问题是需要对任务的实际运行时间有感知



Least Attained Service (LAS) does not rely on a priori estimates of runtime

LAS机制适合长尾分布的任务

LAS一个挑战：需要避免任务抢占和恢复引发的性能开销



中山大學 软件工程学院  
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬  
软件工程学院  
[chenzhb36@mail.sysu.edu.cn](mailto:chenzhb36@mail.sysu.edu.cn)