# 程序设计实习（实验班-2023春）
# 大数据并行计算模型：MPC


授课教师：姜少峰

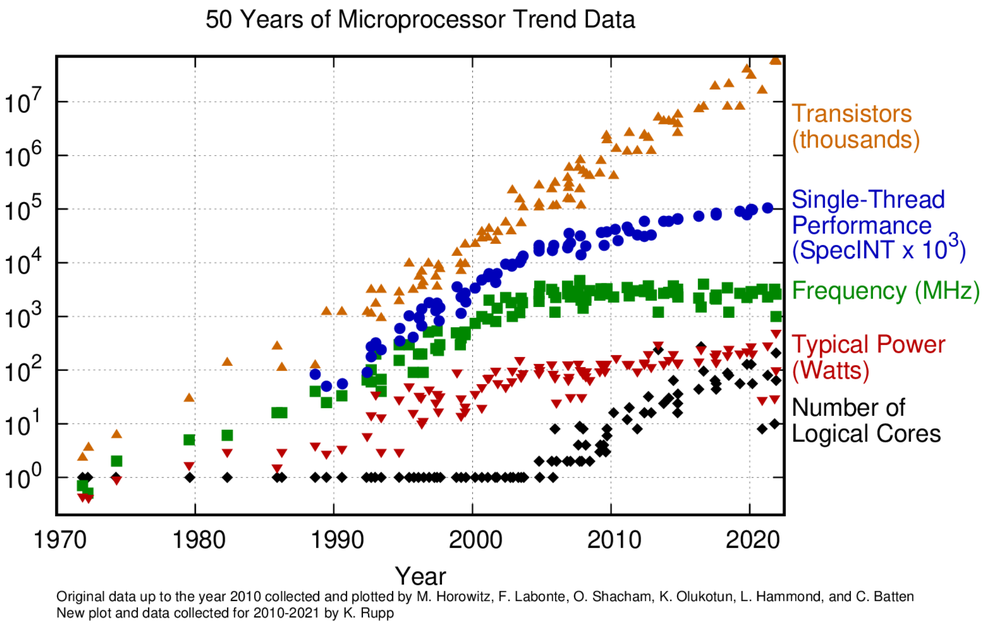
助教：张宇博 楼家宁

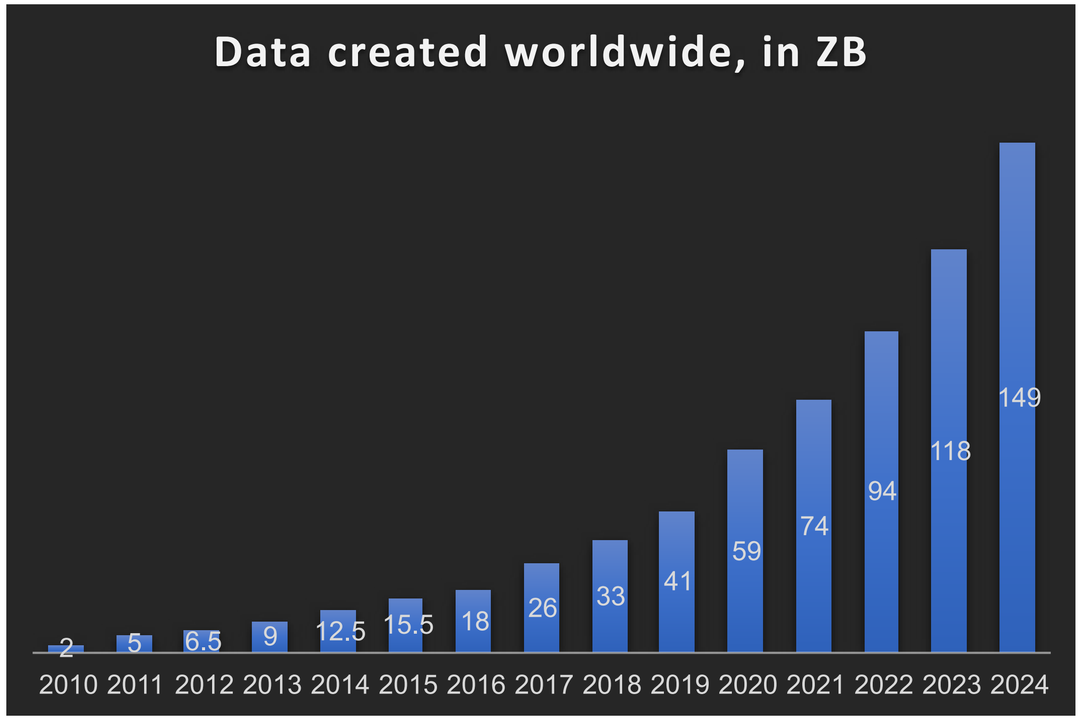
Email: shaofeng.jiang@pku.edu.cn

# 大数据时代

甚至线性时间都不够高效

有效算法 = 多项式时间算法?



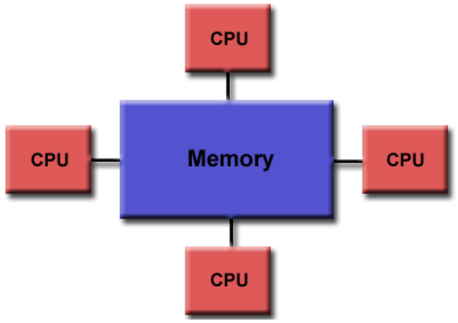




单个CPU：每秒$10^{10}$浮点运算；超级计算机：每秒$10^{18}$浮点运算

即使仅扫描一遍数据也不切实际；单CPU性能增速跟不上数据

# “亚线性”计算模型

- 数据流算法：扫一遍数据流，用较小的空间计算
  - 路由器/物联网设备上直接做数据分析
- 并行算法






MapReduce框架是分布式计算
最流行的软件实现

# MapReduce

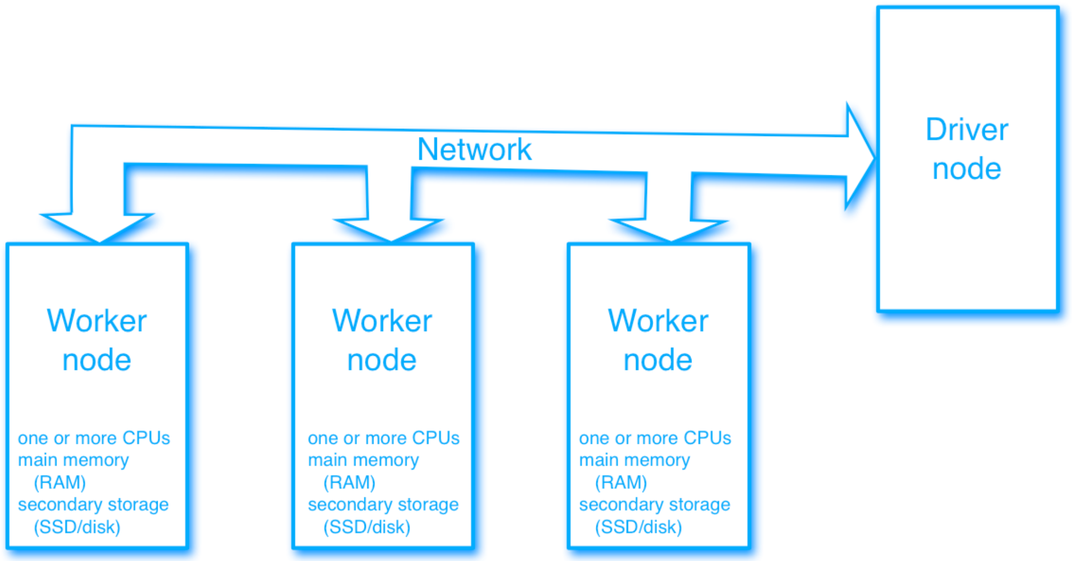
# MapReduce：基本设定

- 设有规模$n$的输入数据，这些数据分散存储在若干机器上
  - 机器空间和机器个数：$n^{1-\epsilon}$

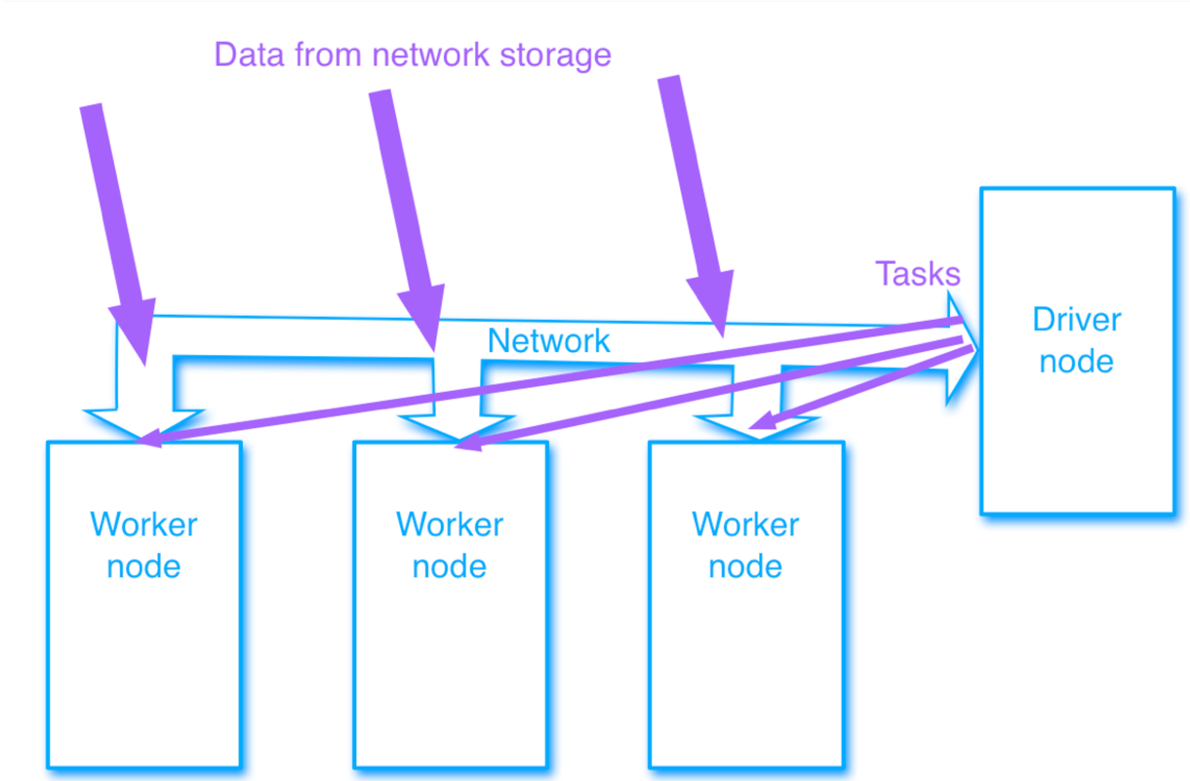
亚线性与输入规模$n$，否则trivial/不是大数据

- 在数据集上的计算通过某些抽象的基本操作进行

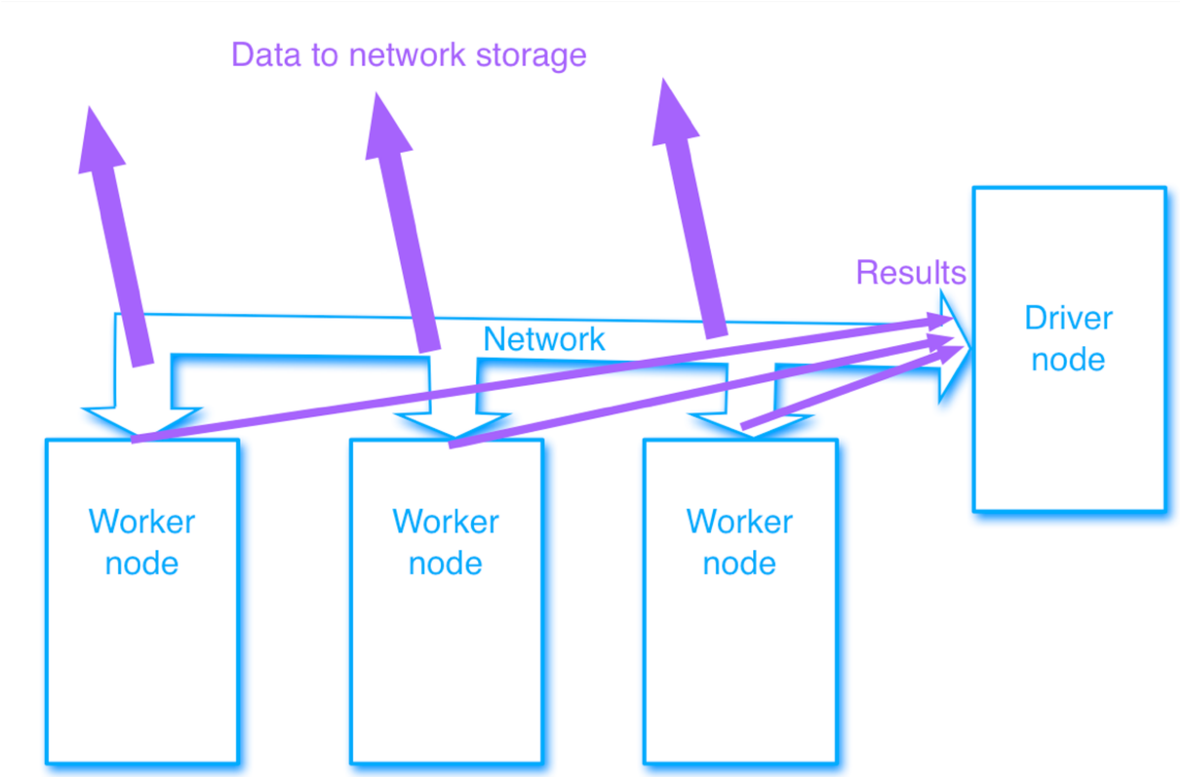
# 大体架构： Driver + Worker



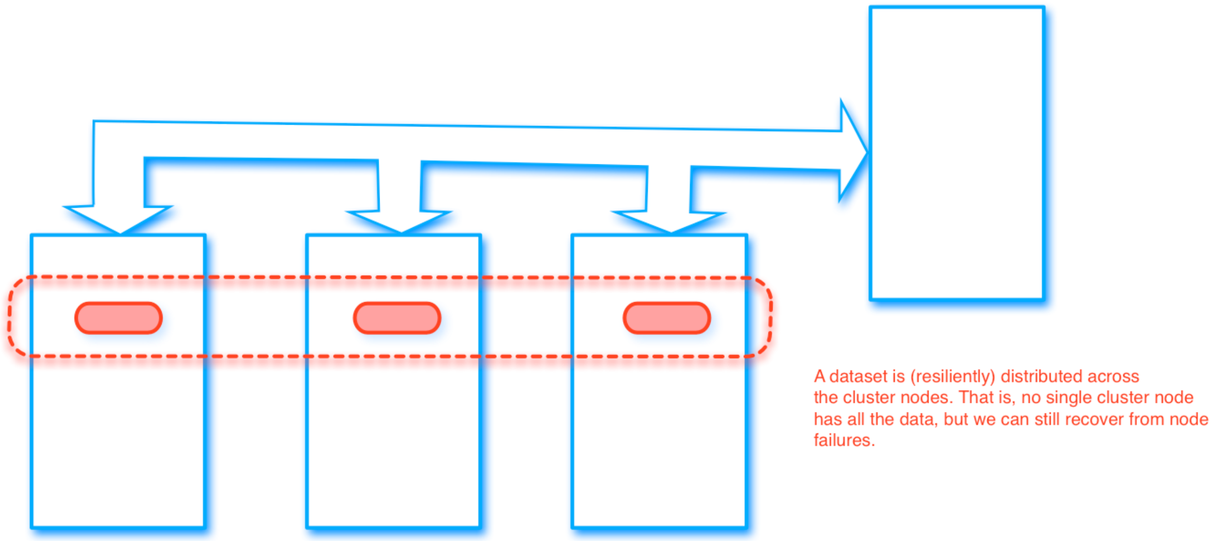
# Driver将数据与任务分配给Worker



# 计算完成后Driver从Worker处收集结果



# 存储： Resilient Distributed Dataset (RDD)



A dataset is (resiliently) distributed across the cluster nodes. That is, no single cluster node has all the data, but we can still recover from node failures.

# MapReduce

- MapReduce框架将RDD抽象出来，定义了若干种RDD上的基本操作
  - 这些基本操作的细节由MapReduce实现，使用者不需要关心
  - 换言之，MapReduce定义了几个基本操作/编程语言，想要操作MapReduce大数据的程序必须要用这几个基本操作写成

# 一般形式

## [Karloff, Suri, Vassilvitskii, 2010]

第一次将MapReduce框架的基本操作描述成数学语言

- 输入： 一组(K, V)对

可以理解成把输入的每个(K, V)对映射成若干其他(K, V)对

- Mapper: 以数据中的每一对(K, V)为输入，输出一个或多个(K, V)对
- Reducer: 输入$K = K^*$的若干(K, V)，返回另一组同样以$K^*$为key的若干(K, V)
  - 即都是$(K^*, (V_1, \ldots, V_t))$形式
- 一轮 = Mapper + Reducer在整个数据集进行一次操作

# 例子：统计频数向量的$\ell_0$-范数

## 即不同元素的个数

- 输入$n$个int分布式存储在一些机器上，想要计算这些数据的频数向量的$\ell_0$-范数
- 如何用Mapper和Reducer的语言在O(1)轮完成?
  - 输入$\{(K,V)=(i,a_i)\}$
  - Mapper: 将每个$(i,a_i)$ 映射成$(a_i,1)$
  - Reducer: 输入$K=a_i$的所有tuple $(K=a_i,(V_1=1,\ldots,V_t=1))$输出$(a_i,1)$
  - Mapper: $(a_i,t)\mapsto(0,t)$, Reducer: $(0,(t_1,\ldots,t_m))\mapsto(0,\sum_i t_i)$

切换关键字

由于关心$\ell_0$，此处将$a_i$所有的出现汇总成1次
若要求$\ell_p$范数应该输出$(a_i,t^p)$

对t求和这一简单操作的MapReduce语言描述

# Apache Spark

- Apache Spark是一种非常流行的支持MapReduce的框架

Spark运行模式很灵活，可以尝试在自己单机配置运行，会利用多个CPU核心进行并行计算

| | | | |
|---|---|---|---|
| **Transformations** | *map*(*f* : T ⇒ U) | : | RDD[T] ⇒ RDD[U] |
| | *filter*(*f* : T ⇒ Bool) | : | RDD[T] ⇒ RDD[T] |
| | *flatMap*(*f* : T ⇒ Seq[U]) | : | RDD[T] ⇒ RDD[U] |
| | *sample*(*fraction* : Float) | : | RDD[T] ⇒ RDD[T] (Deterministic sampling) |
| | *groupByKey*() | : | RDD[(K, V)] ⇒ RDD[(K, Seq[V])] |
| | *reduceByKey*(*f* : (V, V) ⇒ V) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *union*() | : | (RDD[T], RDD[T]) ⇒ RDD[T] |
| | *join*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))] |
| | *cogroup*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))] |
| | *crossProduct*() | : | (RDD[T], RDD[U]) ⇒ RDD[(T, U)] |
| | *mapValues*(*f* : V ⇒ W) | : | RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning) |
| | *sort*(*c* : Comparator[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *partitionBy*(*p* : Partitioner[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| **Actions** | *count*() | : | RDD[T] ⇒ Long |
| | *collect*() | : | RDD[T] ⇒ Seq[T] |
| | *reduce*(*f* : (T, T) ⇒ T) | : | RDD[T] ⇒ T |
| | *lookup*(*k* : K) | : | RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs) |
| | *save*(*path* : String) | : | Outputs RDD to a storage system, *e.g.*, HDFS |

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

# MPC：
# 简化的MapReduce计算模型

# MPC (Massively Parallel Computing)

这里是指一般意义上抽象的数据点，可以是一条图上的边，可以是一个$\mathbb{R}^d$的点等等
总体要求是：假设一个点占据一个存储单位

- 输入数据：$n$个“点”，任意分布在若干机器上
- 每台机器的存储至多是$s = o(n)$，典型设定是$s = n^{\alpha}$

$0 < \alpha < 1$是一个常数

- 定义机器个数为$m = \Omega(n/s)$

$m$不是独立参数，而是取决于$n$和$s$

- 在一轮中：每台机器可以发信息到任何机器、从任何机器接受信息，但每台机器的总收发量必须是$O(s)$的
- 一般我们忽略$\text{poly}\log n$的factor

例如上述的机器数$m$允许$(n/s \cdot \text{poly}\log n)$，
每轮每台机器总通信可以是$O(s \cdot \text{poly}\log n)$

# 关于分布式算法的输出

- 输出一般也是分布式的，因为输出很大且无法存储在一台机器上
- 具体设定取决于问题，例如
  - 排序: 每个元素存储自己在排序后的rank
  - 图的匹配: 每个点存储自己匹配到的点的ID
  - 对于图上的MST：每条输入边存一个Yes/No表示自己是否被选进MST

# MPC算法

- 一段相同的算法/程序需要同时运行在所有机器上
- 目标：一般追求$\mathrm{poly}\log n$轮，最好是$O(1)$轮
- 理论上，主要追求轮数，本地计算可以先忽略不计

但一般设计的算法的本地计算也是高效的

# MPC基本算法

# Broadcast广播

- 设机器1存储了一个长度为$t$的信息A
- 任务：将信息A传送/广播给所有机器
  - 直接发给大家是不行的，因为会爆内存
- 设：机器内存$s \geq t^{1+\epsilon}$

一开始，为不爆通信限制O(s)，机器1将长度t的信息A用1轮发送给$t^{\epsilon}$个其他机器
收到信息的机器再用1轮每个发给其他$t^{\epsilon}$个机器，以此类推

- Claim: 可在$O(\log_s n)$轮完成 (考虑一个$t^{\epsilon}$-叉树进行广播)

思考：要是$s = O(t)$呢?

# 广播的“逆”过程：Converge-cast

- Converge-cast：从每个机器上汇总长度为$t$的信息，到一个leader（如机器1）
- 例如：
  - 每个机器有一个t-维int向量，想要所有机器的t-维向量的求和
- Claim: 可达到与broadcast相同的轮数

将converge-cast看作是“反向”的广播

# 简单应用：单点查询

- 例如，考虑$n$个分布存储在机器上的$\mathbb{R}^d$上的点
- 给一个query point $p \in \mathbb{R}^d$，要求返回$p$在$n$个点中的$\ell_2$最近邻
- $O(\log_s n)$轮可以做到：

比离线算法要容易：离线设定并不能有效支持这种暴力算法

  - 把$p$ broadcast给所有机器，所有机器求本地点到$p$最近邻
  - converge-cast找这些最近邻中的最近邻，汇总到机器1

事实上不止可以回答一个query point，可以回答一个batch，只要batch size $\leq s^{1-\epsilon}$

# 重要工具：排序

- 分布式输入$\{a_i\}_{i=1}^n$
- 要求将$a_i$升序排列使得位次相同的元素占据连续编号的一段机器
- 结论：可以在$O(\text{poly}\log_s n)$轮完成

在$s = n^\alpha$的典型设定下，这个轮数就是$O(1)$的

# MPC实现思路：基于Multi-pivot快速排序

- 基于multi-pivot排序方法

回忆：如果输入数组是$(2,3,7,5,4,10,8)$且pivot是下标$\{2,4\}$，那么就需要根据$a_2 = 3, a_4 = 5$来划分输入，得到值$< 3, [3,5], > 5$三个部分，即$(2), (3,4,5), (7,8,10)$三个新序列；之后要在三个新序列递归排序

- 如何在MPC实现?
  - 找1台机器选好pivot，告知所有机器，每个机器根据pivot划分数据
  - 为实现递归：将每个pivot区间汇总起来发送到连续编号的机器段上

然后在每段机器上进行递归排序

# MPC实现

这里选任何$s^t$ (t < 1)都可以，是为了让broadcast在$O(\log_s n)$轮可做

- 在1号机器选$\sqrt{s}$个$\{1,\ldots,n\}$上的随机pivot，然后broadcast给所有机器
- 每台机器统计每个pivot区间有多少元素，并用converge-cast给1号机

这里pivot区间指的是例子中的三个新序列
注意到每个区间可以很大、横跨多台机器，因此需要converge-cast

- 1号机器再把得到的每个pivot区间的元素个数broadcast给所有机器

是一个长度是$\sqrt{s}+1$的数组/向量

- 此时每台机器同时知道：所有pivot，和每个pivot区间长度

# MPC实现

- 回忆：此时每台机器同时知道：所有pivot，和每个pivot区间长度
- 我们想让每个pivot区间都对应一段连续编号的机器
  - 机器的编号可以用区间长度来计算
- 因此每台机器：
  - 根据pivot划分本机的数据，将每个区间的数据发送到对应编号段的机器

这样所有机器都知道每个pivot区间的数据
应该放到哪些个连续机器上

小细节：比如一个编号段有100台机器，应该如何决定发给哪台?
难在每台机器都有一部分数据要发到这100台上，而机器间很难协调负载均衡
因此应该用随机负载均衡技术：发给这100台的随机的一台

# 排序轮数分析

- pivot个数是$\sqrt{s}$，因此递归产生的子问题大概规模在$n/\sqrt{s}$
  - 总递归层数$O(\log_s n)$
- 每轮递归，所有连续的机器段内进行若干次broadcast和converge-cast
  - 这部分是$O(\log_s n)$
- 总体上：$O(\log_s^2 n)$

# 直接应用：Linear Regression的MPC算法

- linear regression：输入是$X \in \mathbb{R}^{n \times d}$
  - 以分布式稀疏方式给出：$(i, j, X_{ij})$
- 如果假定$d$很小而$n$很大，具体来说$d \ll s \ll n$
- 我们学过的方法：先降维，把$n$降成大概$O(d)$
  - 然后在$O(d) \times d$矩阵$\hat{X}$上用公式$w = (\hat{X}^\top \hat{X})^{-1} \hat{X}^\top \hat{y}$

# 回忆：降维矩阵S

设随机$S \in \mathbb{R}^{m \times n}$每列只有一个均匀随机位置非0，且该元素取均匀随机$\{-1,1\}$

$$S = \left(\begin{array}{c|c|c|c|c|c} 0 & 0 & \cdots & 1 & 0 & -1 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 1 & \vdots & \cdots & \vdots & \vdots & \vdots \\ \vdots & 1 & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & -1 & 0 \end{array}\right)$$

[Clarkson-Woodruff, STOC 13]

定理：用$A' = S$替代JL矩阵A，$m = O(\epsilon^{-2}d \cdot \mathrm{poly}\log(\epsilon^{-1}d))$有类似保证

$$\forall v \in \mathcal{U}, \quad \Pr[\|A'v\|^2 \in (1 \pm \epsilon) \cdot \|v\|^2] \geq 0.9$$

回忆$\mathcal{U}$是$\mathbb{R}^n$的$d$维子空间

# 如何在MPC计算$SX$?

- 对$i \in [n]$，设$S$的第$i$列选取的随机行为$\pi(i)$
- 性质：设$M = SX$，则$X_{ij}$只会影响$M_{\pi(i),j}$
- MPC算法：将$(i, j, X_{ij})$按照$i$为关键字排序，计算$\pi(i)$并broadcast给$(i, \cdot, \cdot)$
  - 再按$\pi(i)$为第一关键字、$j$为第二关键字排序$(i, j, X_{ij})$
- 最后再将$M$直接发送到机器1，进行本地求解$w = (\hat{X}^\top \hat{X})^{-1} \hat{X}^\top \hat{y}$

$\pi$需要$\Omega(n)$空间存储，无法预先在一台机器生成，只能在每个$i$的那一组分布式生成

轮数?

此时排序后的一段可汇总出$M_{\pi(i),j}$的值

若$\hat{X}$无法存放在一台机器，即$d^2 > s$，则这步不能使用

# 思考：其他基本task

- 是否需要排序？不排序能不能做得更好？轮数？
  - 查找某个元素是否出现，返回rank区间有多少元素，去重，众数
- $\ell_0$-采样

# Mapper和Reducer都如何在MPC实现?

本机对每个元素直接处理

- Mapper: 以单独的一对(K, V)为输入，输出一系列(K, V)对
- Reducer: 以$K^*$的相同若干(K, V)为输入，返回另一组同样以$K^*$为key的(K, V)

此处可以使用排序来将同一个K下的V整理到一起

MPC是一种“更灵活”的“MapReduce”模型

# MPC算法研究概述

- MPC算法是新兴算法领域，与大数据处理的关系密切
  - 也受到了以Google Research为代表的工业界实验室的大力推动
- 一个研究主线是为各种不同数据设计MPC算法：
  - 图数据 目前研究最广泛
  - $\mathbb{R}^d$数据 很重要，但是缺乏研究
  - 字符串数据等

# 图数据的MPC算法：细分Regime

- 图数据： 设有$n$个顶点，输入是图的边集

数据至多可以是$O(n^2)$大小

- 注意到输入规模其实是边数$m$，而不是$n$
- 根据机器空间$s$与$n$的关系分为三个regime：
  - super-linear：$s = n^{1+\epsilon}$，near-linear：$s = \tilde{O}(n)$，sub-linear：$s = n^{1-\epsilon}$

连点集和问题的解都存不下

能存下点集和典型问题，例如MST/matching的解
还能有$n^{\epsilon}$空余做别的

能存下点集和很多典型问题的解
但是没有很多额外空间做其他事情

# Super-linear Regime下的MST

## Filtering技术

- 空间$s = \tilde{O}(n^{1+\epsilon})$，且设$|E| = m = n^{1+c}$

边数太少没意义：基本上可以发到一台机器上运行经典算法解决掉

- 结论: 存在MPC算法能在$O(c/\epsilon)$轮计算最小生成树的精确解

**Algorithm:** MST(V,E)

1: **if** $|E| < \eta$ **then**
2: Compute $T^* = MST(E)$
3: **return** $T^*$
4: **end if**
5: $\ell \leftarrow \Theta(|E|/\eta)$
6: Partition $E$ into $E_1, E_2, \ldots, E_\ell$ where $|E_i| < \eta$ using a universal hash function $h : E \rightarrow \{1, 2, \ldots, \ell\}$.
7: In parallel: Compute $T_i$, the minimum spanning tree on $G(V, E_i)$.
8: **return** MST$(V, \cup_i T_i)$

$\eta$就是我们的$s$

如果当前边数少于$s$了，就可以发到一台机器解决

否则，把边集均匀分配到每台机器上，在每台机器上收到的边集$E_i$用传统算法计算MST

细节：一轮converge-cast之后，会收集到$n^\epsilon$个MST边集的并。此时，需要把这个并集的MST求出来，再继续converge cast

最后需要一个converge-cast来汇总每台机器上的MST

# Near-linear Regime下的MST

## Linear Sketching技术

- 这里介绍一个一般方法：数据流算法经常可以转成MPC算法！
- 回忆：对于图数据流，我们得到了一个$\tilde{O}(n/\epsilon)$空间的$(1+\epsilon)$近似MST的算法
- 具体来说：我们对每个顶点$v$维护了总大小$\tilde{O}(1)$的若干$\ell_0$-采样结构
  - 重点：这些$\ell_0$-采样结构都是linear-sketch

回忆linear sketch重要性质：可合并

# 具体实现

- 将边按照较小顶点号排序

回忆：排序操作可以在$O(\log_s n)$轮完成

  - 这样与每个顶点邻接的边都放在了某同一台机器上
- 对于每个顶点$v$，将邻接边插入streaming算法维护的$\tilde{O}(1)$大小的一众$\ell_0$结构里
- 把每个顶点的数据结构直接发到1号机器

注意这里不需要converge-cast来汇总，并且不会爆空间（为什么？）

  - 此时1号机器就有了整个图的streaming数据结构，可直接运行查询得到MST

# 类似的：欧氏空间近似直径

- 回忆：数据流sparse recovery算法可在$O(\epsilon^{-d}\text{poly}\log n)$空间$(1+\epsilon)$近似直径
- MPC算法：
  - 机器将本机点集喂给该数据流算法，得到一个$O(\epsilon^{-d}\text{poly}\log n)$大小的结构
  - 使用converge-cast把所有的结构“求和”，汇总到机器1
  - 机器1继续运行求直径的算法，得出直径

因为是linear sketch，所以可以相加

# 一些思考：什么方法不适合MPC?

- 高度串行的算法很难有效在MPC实现，例如：
  - local search、贪心等迭代式方法
  - 动态规划

一般MPC轮数与迭代方法的轮数相关

[Im-Moseley-Sun, STOC 2017]给出了LIS等问题的有效MPC算法，改动态规划

- 因此如何改造这些算法从而让其可以使用，或者为具体问题设计新的MPC算法，是研究的重点

# 研究方向：MPC上的几何问题

- 很多非常基本的计算问题都没有得到研究，即使对于低维/2维
  - 举例：最近点对，全局最小权匹配，凸包……
- 高维：
  - 什么都不知道，包括直径、最小生成树这些问题能做到什么程度都是open的
- 我正在考虑的问题：高维的直径、MST，聚类

# 课程评估

- 请同学们花几分钟时间参与一下课程评估
- PC端可访问 kcpg.pku.edu.cn
- 手机可扫码关注“本科课程评估”
  - 首页－输入“学号”及“密码”登录－任务评价
  - 非本校同学：点击个人设置－校外绑定－输入“学号”，“密码”默认为学号
  - 根据“我的任务”中的课程，填写问卷并点击“提交”。