arXiv:2302.03317v1 [cs.DS] 7 Feb 2023

通过共享内存并行随机重排，工程化实现原地生成随机排列

Manuel Penschuck

歌德大学法兰克福分校

摘要

随机重排是将元素序列重新排列为随机顺序的过程，使任何排列出现的概率都相等。它是几乎所有科学领域中大量技术的重要构建模块。因此，人们投入了大量工作来设计和实现洗牌算法。

据我们所知，我们首次工程化实现了一种实际运行速度快的并行随机重排算法，其并行深度为 $\mathcal{O}\left(\sqrt{n}\log\ n\right)$，且仅需多对数级辅助内存。我们用 Rust 编写的参考实现可自由使用、易于集成到其他项目中，并能够处理接近系统内存容量的大型数据集。在实证评估中，我们在多种计算机体系结构上将这些实现与若干现有解决方案进行了比较。我们的算法在所有机器上都持续实现了最高吞吐量。此外，我们还证明，并行算法的运行时间与其他算法为获取操作系统内存并复制输入所需的时间相当。

2012 年 ACM 学科分类 计算理论 → 共享内存算法

关键词与短语 随机重排、随机排列、并行性、原地、算法工程、实际实现

补充材料 源代码、相关原始数据和分析脚本可在此处找到 `https://algorithm.engineering/rip_shuffle.tar`（将上传至 Zenodo）。若要在其他 Rust 项目中纳入这项工作，我们建议 `https://crates.io/crates/rip_shuffle`。

资金支持 *Manuel Penschuck*：获德国科学基金会（DFG）资助，项目编号 ME 2088/5-1（FOR 2975——网络中的算法、动力学与信息流）

1 引言

随机排列在许多科学领域中受到广泛研究，并有众多应用。它们通常被视为一种“简单而公平”的安排，因此影响着日常生活的许多方面，从友谊赛中洗一副扑克牌，到决定士兵被征召参战的命运顺序（例如，[23]）。

在计算机科学中，应用包括数值模拟、复杂对象（例如随机图）的采样、机器学习或统计检验（例如，[3, 25, 15, 19]）。尤其是，如果结合拒绝采样，随机重排可能会成为占主导地位的子程序（例如，[1]，其中使用了本工作的一个原型，但未讨论任何算法细节）。此外，假设输入数据按随机顺序（而非对抗性顺序）提供，可以设计出几乎总是高效的实用算法。这一概念还促成了在线算法的随机顺序模型 [10]。出于同样的原因，离线算法的实现可能会先对其输入数据进行随机重排；例如，按照业界惯例，在使用简单的 快速排序 实现进行排序之前，先对输入数据进行随机重排。

从算法的角度来看，随机重排 与排序 的任务紧密相关，因为二者都需要能够生成任意排列的算法。不过，排序需要应对对抗性输入，而随机重排则可以针对表现良好的均匀分布进行优化。通过为每个输入元素附加一个均匀随机变量，再按其排序，可以借助整数排序在线性时间内实现随机重排 [6]；我们将这种方法称为

SortShuffle。广为人知但不切实际的$BogoSort$ 展示了另一种方向，即通过随机重排来排序，但其期望运行时间 $\Omega(n \cdot n!)$ [13]"略微"不理想。

对原地算法的探索，源于内存的各种成本。最显而易见的一点是，如果通过复制来生成输出，机器能够处理的最大数据集大小大致会减半。此外，在现代计算机系统上分配主存需要相当长的时间；在第 6 节中，我们会证明，我们的洗牌算法的运行时间与获取输入大小的额外内存所需的时间相当。另一个隐藏成本是，为处理动态内存分配失败（例如系统内存耗尽）而增加的代码复杂度。最后，有些编程语言提供了不可复制数据的概念；例如，在 C++ 中可以删除复制构造函数，而在 Rust 中，数据类型需要明确声明为可复制。

1.1 我们的贡献

我们设计并实现了实用的共享内存并行算法并行原地 ***ScatterShuffle*** （$PIpScShuf$）。$PIpScShuf$的并行深度为$\mathcal{O}(\log(n)\sqrt{nk}/\log\ k)$，工作量为$\mathcal{O}(n\log_k(n))$（以高概率成立），其中 k 是一个较小的调优参数。该算法是对***ScatterShuffle*** [28] 的原地改造，并融合了 ***MergeShuffle*** [2] 的思想，进一步降低了隐藏常数。我们在一个免费且经过充分测试、即插即用的 Rust 库中提供了快速洗牌实现。我们的$PIpScShuf$ 不使用堆分配；如果提供可设定种子的伪随机数生成器，还能生成可复现的排列。

在第 2 节和第 3 节讨论符号约定及相关工作后，我们将在第 3 节推导顺序的 原地 ***ScatterShuffle*** ($IpScShuf$)，并在第 4 节将其并行化。在第5节，我们将讨论实现细节，并在第 6 节对实现进行评估。

2 预备知识与符号约定

表达式 $(x_i)_{i=a}^{b}$ 表示序列 $x_a, \ldots, x_b$；如果上下文已明确其界限，也可简写为 $(x_i)_i$。我们将包含 n 个元素的数组记为 $X[1..n]$，并将子范围 $X[i], \ldots, X[j]$ 记为 $X[i..j]$。此外，$[n]$ 表示集合 $\{1, \ldots, n\}$。于是，排列是一个双射 $\pi\colon [n] \to [n]$，其中 $\pi(i)$表示第 i个输入元素在输出中的位置。如果某个概率陈述的误差概率至多为 $1/n$，其中参数 n由上下文隐含给出，我们就称该陈述以高概率(whp)成立。

2.1 并行计算模型

对于并行算法，我们采用广为认可的二叉 Fork-Join 模型 [7]。这一选择与我们实现中使用的 `rayon`[1] 基础设施十分契合。执行从单位成本随机访问机上的单个任务开始。此外，任何任务 t_0 都可以递归地分叉 为两个任务 t_1 和 t_2。此时，t_0 会等待 t_1 和 t_2 完成计算并汇合，然后继续执行 t_0。在实践中，Fork-Join 框架（例如 oneTBB[2]、Cilk[3] [18], 或 rayon）会将工作线程池与工作窃取调度器结合起来，把任务映射到处理器核心上。算法性能的衡量指标包括工作量（即指令总数）以及并行深度 （跨度），后者定义为关键路径的长度，相当于假设工作线程数量不受限制时的执行时间。

[1] `https://crates.io/crates/rayon`[2] 此前称为英特尔线程构建模块，
`https://github.com/oneapi-src/oneTBB`[3] 另请参阅 `https://www.opencilk.org`, `http://cilkplus.org`

2.2 随机洗牌

顺序*Fisher-Yates* 洗牌 （FY，又称*Knuth* 洗牌） [17] 可在$\mathcal{O}(n)$时间内获得数组 $A[1..n]$ 的随机排列。概念上，它将所有元素放入一个瓮中，依次不放回地抽取，并按照抽取顺序返回这些元素。该算法原地执行，并在第 $i \in [1...n-1]$次迭代中通过交换 $A[i]$ 和 $A[j]$ 来确定 $A[i]$ 的值，其中 j 是从尚未确定的位置 $[i..n]$中均匀随机选出的。换言之，在第 i次迭代中， A 的$(i-1)$前缀存储了迄今为止得到的结果，而$(n-i)$后缀则代表瓮。

Shun 等人证明，这种看似本质上是顺序执行的算法具有足够的独立性，因此可通过对数级并行深度进行处理（高概率） [29]。随后，Gu 等人提出了一种原地变体，该变体基于并行FY [14]的所谓分解性质。然而，这两种算法所要解决的问题都与我们的略有不同。它们以明确规定的方式对输入进行排列。因此，排列本身是输入的一部分，而实现[4] 需 [14]为每个元素使用两个额外的指针（也就是说，在 64 位机器上洗牌 32 位数值会使内存占用增加到原来的五倍）。

通过将每个元素均匀随机地分配到 $\mathfrak{P}$ 个桶中的一个， $\mathfrak{P}$ 个处理器可以并行计算随机排列，然后对每个桶应用顺序算法 [28]。我们将此算法称为*ScatterShuffle* ，并将在第 3 节以此为基础。类似的技术也可得到一种 I/O 高效随机排列算法 [28]。

反向进行同样可以得到一种高效算法。*MergeShuffle* (MS) 首先为每个处理器分配输入数组中的一个连续区段，以高度并行的方式随机重排子问题，最后递归地将它们合并 [2]。合并可以解释为 *ScatterShuffle* 使用两个桶进行分散的逆过程。在一项前期研究中，我们发现，使用为降低递归深度所需的 k 路合并来推广 *MergeShuffle* 的速度过慢。不过，我们在第 3.3 节中的$RoughScatter$ 例程建立在这样一个洞见之上：对于大多数元素，可以仅使用一个随机比特来实现二元随机合并。

Cong 和 Bader [6] 通过实证研究了其他技术，例如使用排序算法进行随机重排（*SortShuffle*）或随机掷飞镖（*DartThrowingShuffle*）。然而，我们并不了解如何以原地方式实现这些方法。[5]

2.3 从离散分布中采样

下面，我们从几种离散概率分布中进行采样（可以说，随机重排本身就是其中一种）。具体做法是，首先获取由相互独立且无偏的随机比特组成的流，然后将其重新塑形以获得所需的分布。实现第一步的默认方式是使用伪随机数生成器，例如 Pcg64Mcg [24]。

从随机字中采样一个来自 $[0, s)$ 的整数，使用 $s = 2^k$ for 某个 $k \in \mathbb{N}$，成本极低，只涉及移位和掩码操作。我们采用拒绝采样算法，以期望常数时间从 $[0, s]$ for 一般的 s 中采样均匀随机变量（见 [19]）和二项随机变量（见 [8]）。通过依次进行参数设置适当的二项采样，以期望时间 $\mathcal{O}(k)$ 实现对 k维多项随机变量的采样。

3 顺序原地洗牌

[4] `https://github.com/ucrparlay/PIP-algorithms` 写入时的主导因素（`6af1df9`）[5] 我们的 $IpScShuf$ 算法可以解释为一种优化的原地基数排序，其中的桶是随机抽取的。因此，它在概念上与*SortShuffle*存在相似之处。

```
pub fn fisher_yates<R: Rng, T>(rng: &mut R, data: &mut [T]) {


                                                                    }
```

图 1 Rust 中的 Fisher-Yates 算法实现

本节提出 原地 *ScatterShuffle* （$IpScShuf$），这是 Sanders 的并行 *ScatterShuffle* [28] 的一种顺序原地变体。图 2 概述了该算法。基于所得的性能结果，我们将在第 4 节重新引入并行性。

3.1 研究现状

实践中最常用的洗牌算法似乎是简单而快速的费希尔–耶茨洗牌 （FY） [17]。事实上，图 1 展示的正是第 6 节采用的“朴素”实现，与 `rand` crate 中包含的版本几乎完全相同；该 crate 是 Rust 生态系统中事实上的标准随机化库。由于简单易用，该算法在输入数据较小时优于更复杂的方案。然而，FY 对主存的无规律访问会导致处理较大输入数据时性能严重下降。这对于并行算法尤其重要，因为多个处理器核心共享内存子系统（见第 6 节）。

ScatterShuffle 旨在成为一种并行算法，同时也能在外存模型 [28]中表现良好。给定一个输入 $(x_i)_{i=1}^n$，该算法会将每个输入元素 x_i移入从 $B_1, \ldots, B_k$中独立且均匀抽取的一个桶中。随后，每个桶都构成一个独立的子问题，其元素期望数量为 $\Theta(n/k)$，算法将对这些子问题进行递归处理。对于较小的子问题，我们切换到 FY 作为基本情况算法。

在 *ScatterShuffle*最初的并行实现中，桶数量 k 等于处理单元数量 $\mathfrak{P}$，以展现最大的并行度。然而，Sanders 已经指出，在存在存储层次结构的情况下，应将参数 k 设置得足够小，使各个处理器至少能够缓存每个桶的尾部。在他的实现中[6], ，对于 [28]中规模最大的运行任务，该参数为 $k = 32$。截至本文撰写之时——二十多年过去了，我们使用完全不同的硬件，对规模大三个数量级以上的数据集进行实验——通过经验发现，对于我们的 原地 *ScatterShuffle*， $k \leq 256$ 是涵盖广泛输入大小时的最佳选择。因此，应直观地将 k视为一个较小的常数，它主要控制递归的分支因子。在第 4 节中，我们将加入独立于 k的并行性。

我们进行的主要改进是$IpScShuf$，其核心是原地散射 ($IpSc$)，它将输入散射到 k 个桶中。形式化地，设 $X = (x_i)_{i=1}^n$ 为输入， $A = (a_i)_{i=1}^n$ 为从 $[1, k]$ 中独立抽取的均匀变量，用于索引上述桶。然后，$IpSc$ 通过使用某个排列 π 对 X 中的元素进行重排，使其按 A排序，从而按 A 对 X进行分组。

整数排序领域也研究过类似问题。$k = 2$ （即二进制划分）和 $k = 3$ （称为荷兰国旗问题）这两种特殊情况可以高效地原地解决 [9, 22]。对于 $k > 3$，可以采用两遍方法（例如美国国旗排序 [21]），但需要反复访问每个 a_i。 `libstdc++` 中包含的*ScatterShuffle* 并行实现 [30] 使用了这一技术，并显式存储 A，因此需要 $\Theta(n \log k)$ 位。另一种思路是遵循 [12], ，要求使用伪随机数生成器

[6] `https://web.archive.org/web/20050827081959/http://www.mpi-sb.mpg.de/~sanders/programs/randperm/` [sic]

```
    pub fn shuffle(&self, rng: &mut R, data: &mut [T]) {
```

```
kets
```

```
2
```

Rust

通过复制并恢复生成器的内部状态，可以多次重放。

3.2 IpScShuf——ScatterShuffle 的原地实现

下面我们介绍支持真正随机比特的原地散射（*IpSc*）。该方法以高概率在线性时间内运行，并且仅使用$\mathcal{O}(k\log k)$比特的额外存储。由于每个 n元素都被分配到均匀随机选择的桶中，因此分配到各个桶的元素数量（n_i）$_{i=1}^k$ 服从多项分布，并且高度集中在 n/k附近。

下文中，我们假设 $n \gg k(\log k)^3$ 且 $k^3 \log\ k = \mathcal{O}(n)$，因为否则问题规模太小，更适合使用费希尔–耶茨洗牌。这些假设仅用于界定*IpScShuf*的复杂度，并不影响其正确性。在实践中，这意味着建议的最小规模约为 10^5 个元素。

一种直接的解决方案是将所有桶的大小抽取为一个多项分布随机变量。然后，我们按照目标大小递减的权重对桶进行无放回采样。借助适当的动态加权抽样数据结构（例如，[20]），可以在线性期望时间内实现这一过程。如第5节进一步讨论的那样，这类方法的性能不如下面的方案。受 [2] 框架的启发（但在具体细节上有很大不同），我们的分配任务由两个阶段组成。首先，在*RoughScatter*阶段，我们非常高效地分配绝大多数元素——但几乎可以确定不会分配完全部元素。其次，在*FineScatter* 阶段，我们处理剩余的少量元素。

3.3 RoughScatter——机会主义的主力

RoughScatter 利用最终桶大小集中在其均值 n/k 附近这一现象，以机会主义的方式放置元素，直到达到上述 n/k 屏障。令$X[1..n]$ 表示输入数组。如图3所示，我们将 X 划分为 k 个大小相等的连续桶 n/k （允许舍入误差）。每个桶 B_i 都存储为一个由索引组成的三元组（b_i，s_i，e_i），其中 b_i 指向 B_i 的起始位置，而 e_i 指向桶末尾之后的位置。每个桶还会进一步划分为：（i）一个初始为空、用于存放所谓的已放置 项的区段 $X[b_i..s_i)$；以及（ii）用于存放所谓的暂存 项的区段 $X[s_i..e_i)$。当桶 B_i 中的所有项都已放置时，我们称该桶为满，即 $s_i = e_i$。在第 3.4 节的最后一步之前，暂存 项可以自由移动，

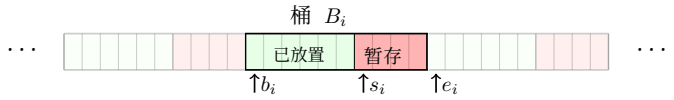


图3 *IpScShuf* 将输入划分为 k 个桶，每个桶大约包含 n/k 个元素。最初，所有项都处于暂存状态（$b_i = s_i$），此时称桶为空。最终，会有越来越多的项（从左侧开始）被放置。如果 $s_i = e_i$，则称桶为满。

而已放置 项的位置则具有特定含义。

每次迭代中，算法都会确定元素 x 的桶分配，具体来说，就是确定此时指向 s_1的第一个暂存元素 B_1 。为此，我们从 $[1..k]$ 中均匀随机抽取一个伙伴桶 j，交换元素 $X[s_1] \leftrightarrow X[s_j]$ （如果 $j = 1$，则跳过），并递增 s_j。因此，元素 x 会被移入伙伴桶 B_j 的已放置区域。如果 B_j 此时已满，算法便停止；否则继续重复。

▶ **引理1**。 设 P 为 *RoughScatter* 已放置元素的集合， $x \in P$ 为任意一个已放置项。那么 x 以概率 $1/k$ 被分配到桶 B_i 。

证明。 不失一般性地，我们假设最初所有项目都处于暂存状态。于是，每个项目都对应唯一的一次迭代 $x \in P$，在该迭代中它会被放置。为此，当时仍处于暂存状态的元素 x 会与一个暂存元素 y 交换，其中 $y \in B_i$ 的概率为 $1/k$。随后它递增 s_i ，从而将 x 定义为已放置。由于 *RoughScatter* 只交换暂存元素，因此 x 的此次放置是最终的。元素 y 的位置可能发生变化，但这无关紧要，因为每次分配都是独立进行的。 ◀

下面，我们界定在 *RoughScatter* 之后仍处于暂存状态的元素数量。

▶ **引理 2**。 一次 *RoughScatter* 运行后，令 $r_i = e_i - s_i$ 表示桶 b_i 中仍处于暂存状态的项目数量，并令 $R = \sum_i r_i$ 表示它们的总和。对于 $n \gg k$(对数 k)[3],, 我们有 $R \le \sqrt{2nk}$ 对数 k （以高概率成立）。

证明。 我们将 *RoughScatter* 的输入解释为 n 个球，这些球被独立地投入 k 个均匀箱中。如果我们将球入箱实验运行至结束，则任意桶的最大负载至多为 $M(n,\ k) = \frac{n}{k} + \sqrt{2\frac{n}{k}}$对数 k（以高概率成立）[26]。[7]

令 n' 表示该游戏中最大负载首次达到n/k时分配的球数。从算法角度看，这对应于 *RoughScatter*的终止。通过确定$M(n', k) = n/k$并求解 n'，我们发现，以高概率有 $n' \ge n - \sqrt{2nk}$ log $k := n - R$。 ◀

▶ 备注 3。未处理元素所占的比例 R/n 在 $n \to \infty$时趋于零。即使对于 $n = 2^{22}$ 较小且 $k = 64$符合实际情况的输入，以高概率仍有不到 1% 的输入未被分配。

3.4 FineScatter——处理小规模剩余部分

执行完 *RoughScatter* 后，在 *FineScatter* 阶段只需分配 ~~$R = \mathcal{O}(\sqrt{nk}\log k)$~~ 个元素 whp。若我们对 $R \gg k$(log k)[3], 的初始假设仍然成立，就可以将暂存元素压缩到一个连续内存区域中，应用 *RoughScatter* 并递归处理。然而，对于较小的输入，该假设很可能不成立；而对于较大的输入，该比例 R/n 对总运行时间的影响可以忽略不计。因此，我们认为不值得为这种情况设计合并过程，而是直接使用专门的

[7] 在分析*ScatterShuffle* [28]时已经使用过类似的论证。

基本情况算法，该算法基于以下引理。它指明了如何高效地获得最终桶大小，并独立分配剩余元素。

▶ **引理 4.** 设 $X = (x_i)_{i=1}^n$ 是一个序列，且 $N = (n_i)_{i=1}^k$ 是从权重相等的多项分布中抽样得到的，其中 $p = 1/k$ 满足 $\sum_i n_i = n$。设 $f_N \colon [n] \to [k]$ 是 X 的任意划分，其类别大小为 N。最后，设 $\pi \colon [n] \to [n]$ 是一个随机排列。那么，对于固定的 i 和 j，元素 x_i 经由 $f_N(\pi(i))$ 映射到类别 j 的概率为 $1/k$。

证明。 根据对称性，只需考虑第一个划分类别 $j = 1$。其大小 n_1按照二项分布计算，试验次数为 n，成功概率为 $p = 1/k$，这是由多项分布的定义可知的。此外，存在一个排列 γ，使得复合映射 $f_N \circ \gamma$将索引 $1, \dots, n_1$ 映射到第一个划分类别。由于 π是均匀随机选取的，$\pi' = \pi \circ \gamma$ 同样是等可能的。因此，只需计算在所有 π' 中，x_i 将一个固定的 n_1 映射到前 $0 \le n_1 \le n$个秩的总概率：

$$\sum_{j=0}^{n} P[\pi'(i) \le n_1 \mid n_1 = j] \cdot P[n_1 = j] = \sum_{j=0}^{n} \underbrace{\frac{j}{n}}_{=1-(1-\frac{j}{n})} \cdot \binom{n}{j} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{n-j} \tag{1}$$

$$= \underbrace{\sum_{j=0}^{n} 1 \cdot \binom{n}{j} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{n-j}}_{=1} - \sum_{j=0}^{n} \underbrace{(1 - \frac{j}{n}) \cdot \binom{n}{j}}_{= \begin{cases} \binom{n-1}{j} & \text{if } j < n \\ 0 & \text{if } j = n \end{cases}} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{n-j} \tag{2}$$

$$= 1 - \underbrace{\sum_{j=0}^{n-1} \binom{n-1}{j} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{(n-1)-j}}_{=1} \left(1 - \frac{1}{k}\right)^1 = 1/k \qquad \blacktriangleleft$$

3.4.1 确定最终桶大小

令 $N = (n_i)_i$ 表示由 *RoughScatter* 分配给桶 B_i 的元素数量。根据引理 4，基本情况算法首先抽取一个多项分布变体 $N' = (n'_i)_i$，其中 n'_i 对应于将由*FineScatter*放入桶 B_i 的元素数量。然后，最终大小 $N^f = (n^f_i)_i$ 为 $n^f_i = n_i + n'_i$。由于 N 和 N' 服从具有 k 个等权类别的多项分布，它们的和 N^f 也服从多项分布。

根据构造，期望桶大小为 n/k。令 $d_i = n^f_i - n/k$ 表示桶 B_i 大小的偏差，即相对于*RoughScatter*初始估计，该桶还需要增加的元素数量。与引理 2 的证明类似，我们以高概率有 $\max_i \{|d_i|\} = \mathcal{O}(\sqrt{n/k \log k})$ [26, 28]。因此，桶边界很可能只会略微移动。

幸运的是，***ScatterShuffle*** 对递归前桶内元素的顺序并不敏感。因此，我们只需使用我们的*TwoSweep* 算法，将少量元素适当地移至桶的边界附近即可。首先，我们按索引升序遍历各个桶。同时，我们维护一个计数器 $C_i = \sum_{j=1}^{i-1} d_j$，用于指示当前桶 B_i左侧还需要多少额外元素。如果桶 B_i 超出大小限制的元素多于 C_i，我们就将多出的暂存元素交换到桶 B_{i+1}的暂存区。在从 B_k 到 B_1的第二次扫描中，我们将剩余的多余元素移向索引更小的桶。

▶ **引理 5。** 设 $N^f = (n^f_i)_i$为最终桶大小，其与均值 n/k的偏差记为 $d_i = n^f_i - n/k$，偏差的包含当前项的前缀和记为 $D_i = \sum_{j=1}^{i} d_j$。那么，***TwoSweep*** 总共执行 $M(N^f) = \sum_i |D_i|$次交换，耗时为 $O\big(k + M(N^f)\big)$。

证明。 *TwoSweep* 可以通过一次交换并采用两个相关桶的指针，在 B_i 时间内将桶中的一个暂存元素与其直接邻居 $B_{i\pm1}$ 进行$\mathcal{O}(1)$ 交换；以这种方式在桶 B_i 和 B_j 之间交换元素，需要一串 $|i-j|$ 次交换，产生 $\mathcal{O}(|i-j|)$ 的工作量。基于此，*TwoSweep* 会进行两次推雪犁式移动，很可能沿着链推动中间元素。其余内容见 **??**。

请注意，正值 d_i 表示桶 B_i 需要从其他桶接收 d_i 个额外的暂存元素。相反，负值 d_i 意味着桶 B_i 必须移出 $-d_i$ 个元素。前缀和 D_i 具有类似的含义，但其累加范围是前 i 个桶。这会导出以下情况：

1. 如果 D_i 为正，则桶 B_1、.....、B_i 中有 D_i 个元素需要在B_{i+1}、.....、B_k中的某处使用。这些 D_i 元素将在第一次扫描期间被推向右侧。**2.** 如果 D_i 为负，则桶 B_1、.....、B_i 有 $|D_i|$ 个元素的需求，这些需求由桶 B_{i+1}、.....、B_k中的某处的过量元素满足。因此， B_i 在第二次扫描中接收 $|D_i|$ 个元素。

总而言之，桶 B_i 与其直接邻居进行了 $|D_i|$ 次交换，总计进行了$M(N^f)$次交换，并产生了 $\mathcal{O}\big(M(N^f)$ 的 $+k\big)$ 工作量，其中 k 计入了每个桶的开销。◀

▶ **推论 6.** *TwoSweep* 的运行时间为 $O\big(k\sqrt{nk}\log k\big)$（以高概率）。

证明。 我们基于引理 5 证明该结论，方法是建立 $\sum_i |D_i| = \mathcal{O}\big(k\sqrt{nk}\log k\big)$（以高概率），其中 $D_i = \sum_{j=1}^{i} d_i$ 是桶大小相对于均值的偏差的前缀和。注意，根据构造，只有在执行 *RoughScatter* 后仍处于暂存状态的元素才会产生影响，因此 $\sum_i |d_i| \le 2R$ 其中 $R \le \sqrt{2nk}\log k$（以高概率），由引理 2 可得。此外，由于所有桶的偏差总和为零，我们有 $\sum_i d_i = 0$。因此，显然有 $\max_i |D_i| \le R$。

我们假设最坏情况下的偏差：第一个桶需要从最后一个桶（或反之）获得全部 R 个元素。虽然这相当悲观（回想一下 $\max_i\{|d_i|\} = \mathcal{O}\Big(\sqrt{n/k}\log k\Big)$（以高概率）），但只需证明该界即可。在这种情况下，对于所有 i，我们有 $|D_i| \le R$，且 $\sum_{i=1}^{k} |D_i| \le \sum_{i=1}^{k} R \le kR = \mathcal{O}\big(k\sqrt{nk}\log k\big)$（以高概率）。◀

3.4.2 分配剩余的暂存元素

执行到此时，所有桶都已达到最终大小，但每个桶 B_i 仍有 n'_i 个暂存元素，其中 $\sum_i n'_i = R$。我们不按 $N' = (n'_i)_i$进行加权采样，而是应用引理 4，随机重排所有暂存元素。具体做法是将暂存元素压缩到 $X[1..R]$ 中，并随机重排 X。为此，我们将暂存元素与最初存储在 X中的元素交换；随机重排后，再按相反顺序执行相同的交换序列，以还原原始元素，并使暂存元素形成一个随机排列。

3.5 综合起来

原地散射(*IpSc*) 是*IpScShuf*每个递归层上执行的算法。它依次运行*RoughScatter* 和*FineScatter* ，将 n 个元素随机分配到 k 个桶中。输入会重新排列，使每个桶对应一个连续内存区域。

▶ **引理 7**。 对于 $n \gg k \log^3 k$ 和 $k^3 \log k = O(n)$，*IpSc* 可在 $O(n)$ 时间内分配 n 个项目，且以高概率成立。

证明。 我们总结一下执行的四项任务：
1. RoughScatter 首先用 O(k) 的时间将输入划分到各个桶中，然后以高概率随机分配 $n - R =$ O(n) 个元素。假设获取一个随机数值需要常数时间，则其时间复杂度为 O($k + n$) = O(n)。2. 以高概率，采样一个 k 维多项分布随机变量需要 O(k) 时间。3. 运行 TwoSweep 来调整边界大小，以高概率需要 O$\left(k\sqrt{nk}\log k\right)$ = O(n) 时间。4. 使用费希尔–耶茨洗牌对暂存元素进行随机重排，需要 O(R) =O(n) 时间。[8] ◀

原地 *ScatterShuffle* 由递归应用*IpSc*构成。当子问题达到 $N_0 = \mathcal{O}(1)$ 的基本情况规模时，我们便停止递归，并使用费希尔–耶茨洗牌完成处理。

▶ **定理 8**。 高概率下，原地散射 *ScatterShuffle* 的运行时间为 $O(n\log_k(n/N_0))$，并需要 $O(k\log_k(n/N_0))$ 个额外存储字，其中 $N_0 = \Omega(k^3)$ 为基本情况的规模。

证明。IpScShuf 将长度为 n 的输入划分为 k 个大小为 $\Theta(n/k)$ 的独立子问题（高概率保证）。随后，它递归调用自身，直到达到 N_0 的基本情况规模。高概率下，这需要 O($\log_k(n/N_0)$) 层递归，每层耗时 O(n)，深度优先遍历需要 O(k) 个字的内存。基本情况 FY 对每个大小为 n' 的子问题使用 O(1) 个字的内存，耗时 O(n')，所有子问题合计耗时 O(n)。◀

4 并行算法

本节介绍并行原地 *ScatterShuffle* (***P****IpScShuf*)，即*IpScShuf*的并行变体。显然，执行*IpSc*（即*IpScShuf*的单个递归层）后，我们可以轻松地并行处理 k 个独立子问题——这是原始 *ScatterShuffle* [28]的核心见解之一。遗憾的是，在我们的场景中，仅并行化子问题会导致线性并行深度，因为首次*IpSc* 执行需要 $\Omega(n)$ 的顺序工作量。因此，我们还必须并行化 *IpSc* 本身。我们重点讨论*RoughScatter* 的并行化；在实践中，它占据了绝大部分工作量。

4.1 并行化 RoughScatter

从本质上说，并行*ParRoughScatter* 会在独立子问题上并发运行顺序*RoughScatter* 。为此，我们利用了*RoughScatter* 允许桶之间 存在任意间隙这一特点。其次，每次赋值后，我们都可以随时暂停并恢复执行，而无需额外开销，因为算法的状态已由桶的指针三元组以及已放置元素的划分完整表示。

与第 3.3 节类似，我们首先将输入划分为 k 个大小大致相等的桶。为了进行分叉，我们进一步将每个桶一分为二，并将其中一半分配给一个子任务。随后，每个子任务要么递归地继续划分，要么在子问题足够小时，运行顺序*RoughScatter*。两个子任务汇合后，我们合并每个桶的两半。这只涉及对桶的指针进行操作，并将第一半中的暂存元素交换到第二半。此外请注意，如果存在一个已填满的桶 $B_i^{(1)}$，第一个子任务就会结束；第二个子任务的情况也类似 $B_j^{(2)}$。只有在概率为 $1/k$的情况下，才会有 $i = j$，因此合并后的桶 B_j 已满。否则，所有合并后的桶都至少包含一个暂存元素，我们会继续执行*RoughScatter*。

[8] 根据定理 8，我们也可以自由地使用*IpScShuf* 进行递归，而不必使用费希尔–耶茨洗牌。

▶**观察 9.** 由于 *ParRoughScatter* 在每次汇合后都会应用 *RoughScatter*，根据引理 *2*，剩余暂存元素的数量 R同样适用于 *ParRoughScatter*。

▶ **引理 10.** 对于 $n \gg k\log^3 k$ 和 $k^2 = O(n)$，以高概率，*ParRoughScatter* 的并行深度为 $O\big(\sqrt{nk}\log k\big)$，所需工作量为 $O(n)$ 。

证明。 将 k 个桶划分为 $2k$ 需要$\mathcal{O}(k)$ 时间。根据观察 9，引理 2 给出了从两个子任务接收的暂存元素数量上界，为 $\mathcal{O}\big(\sqrt{nk}\log k\big)$。这也给出了合并期间交换元素所需时间的上界，以及合并后对剩余暂存元素运行$RoughScatter$ 所需时间的上界。为满足引理 2 的前提，我们将基本情况的规模设为 $N_0 = k^2$ ，并在 $\mathcal{O}(N_0)$时间内顺序处理规模更小的子问题。这得出并行深度的如下界 $D(n)$:

$$D(n) = \begin{cases} D(n/2) + \mathcal{O}\big(k + \sqrt{nk\log k}\big) & \text{if } n \geq N_0 \\ \mathcal{O}(N_0) & \text{if } n < N_0 \end{cases} \tag{3}$$

$$= \mathcal{O}\Big(N_0 + \log(n/N_0)k + \sqrt{nk\log k}\Big) = \mathcal{O}\Big(\sqrt{nk\log k}\Big) \tag{4}$$

类似地，我们利用主定理 [4]对以下递归求工作量上界 :

$$W(n) = \begin{cases} 2W(n/2) + \mathcal{O}\big(k + \sqrt{nk\log k}\big) & \text{if } n \geq N_0 \\ \mathcal{O}(N_0) & \text{if } n < N_0 \end{cases} = \mathcal{O}(n)$$ ◀

4.2 并行化 FineScatter

正如我们在第 6.4 节中所讨论的，实际上，由于 FineScatter 对总运行时间的影响微乎其微，因此没有必要将其并行化。因此，下面我们仅概述必要的改动，将 FineScatter 的并行深度降低到 ParRoughScatter 的水平。根据观察 9，在执行 ParRoughScatter 后，第 3.4 节中对 FineScatter 的分析仍然成立。与引理 10 对比后，我们发现，ParRoughScatter 的并行深度支配了除 TwoSweep 之外的所有顺序操作。

回想一下，TwoSweep 会在 $\mathrm{O}\big(k\sqrt{nk}\log k\big)$ whp 时间内移动桶的边界，使其与最终桶大小相匹配。因此，我们只需将开销缩减一个 $\Theta(k)$ 因子；利用基于以下观察的标准并行化技术即可轻松实现：引理 5 中定义的前缀和 D_i 可解释为桶 B_i 末端所需移动的元素数量。因此，计算完 $(D_i)_i$ 并为每个桶分配一个工作线程后，每个工作线程都可以沿适当方向移动元素。为在相距较远的桶之间移动元素，我们执行 $\mathrm{O}(k)$ 轮。每轮的时间由跨越任意桶边界的最大元素交换量决定，而该值又不超过第一轮中的最大偏差 $\mathrm{O}\Big(\sqrt{n/k}\log k\Big)$ whp。因此，TwoSweep 可以通过朴素方式并行化，其并行深度为 $\mathrm{O}\big(\sqrt{nk}\log k\big)$ ，与 ParRoughScatter 相匹配。由此得到的工作量平凡上界为 $\mathrm{O}\big(k\sqrt{nk}\log k\big)$ ，与推论 6 中对顺序 TwoSweep 所作的高估相匹配。

▶ **定理 11.** *PIpScShuf* 的并行深度为 $O\Big(\sqrt{nk}/\log k\log(n)\Big)$（高概率），工作量为 $O(n\log k(n))$，并使用 $O(k[\log k(n) + \mathfrak{P}])$ 个字的内存，其中 $\mathfrak{P}$ 是并行子任务的数量。

证明。 证明过程与定理 8 的证明类似，只需将 $IpSc$ 的复杂度度量替换为引理 10，再进行符号化简。内存上界还需计入每个子任务的 k 个桶指针三元组。◀

5 实现

我们的实现使用 `Rust`，这是一种具有强内存安全性和并行性保证的编程语言。虽然代码仓库包含许多原型，但我们认为已公开提供的算法（例如 *IpScShuf* (`seq_shuffle`) 和 *PIpScShuf*(`par_shuffle`)）已可用于其他项目。为监控代码质量，我们依赖 Rust 生态系统中强大的静态分析工具和动态检查。我们还使用了 80 多项测试，其中包括对生成排列均匀性的统计检验（例如，输出秩的 1-独立性和 2-独立性）。

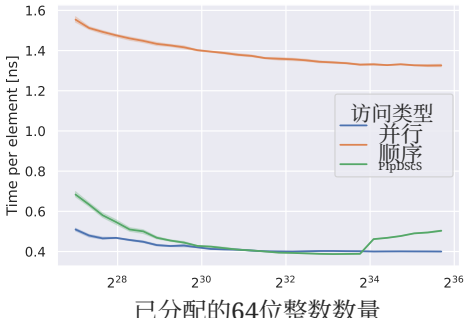
PIpScShuf 使用 `rayon`[9] crate 中包含的工作窃取调度器。我们仅通过 `rayon::join` 函数使用二叉 Fork-Join 并行，该函数在工作池初始化后无需进行堆分配。鉴于`rayon`的广泛使用，调用应用程序很可能已经设置了这个工作池。这样，我们的算法就不会导致任何堆内存分配。如果两个任务由同一个工作线程执行，那么一次 `rayon::join` 的开销非常小。因此，我们通常会定义超过 2^{11} 个并行子任务，以实现细粒度的工作负载均衡。若使用兼容的伪随机数生成器（需要 `rand::SeedableRNG` trait），我们会利用确定性序列，根据提供的生成器为各子任务派生生成器。这样，即使调度具有非确定性，只要初始状态相同，两次运行也会得到相同的排列；这种可选的可复现性可能至关重要（例如，在调试嵌入代码时）。

对于中等规模的输入数据（小于 128 MiB），我们通过经验将桶数量 k 优化为 $k = 64$；对于更大的实例，则优化为 $k = 256$。这一阈值是在各种计算机体系结构之间取得的折中方案，影响较小。绝大多数代码都使用 Rust 的 安全子集实现。在第 6 节中，我们使用了 *RoughScatter* 的一种可选高度优化实现，该实现要求在没有显式边界检查的情况下进行指针运算和内存访问，而这在 Rust 中被视为不安全。在 `x86` 平台上，代码执行 $2\lfloor 2/\lceil \log_2(k) \rceil \rfloor$ 次随机赋值，且无需任何分支指令（即，对于 $k = 64$，执行 20 次；对于 $k = 256$，执行 16 次）。这使 CPU 流水线能够得到充分利用，而通过显式预取内存位置，利用率还能进一步提高。此外，我们没有使用带有三次移动操作的标准 `swap`(x, y)（即 $t \leftarrow x$, $x \leftarrow y$, $y \leftarrow t$，其中 t 为临时存储），而是使用两个临时变量，使每次赋值只需进行 $2 + \epsilon$ 次数据移动。最终得到的赋值过程至少比我们尝试过的任何加权抽样策略快五倍（包括 [20] 的快速实现，以及各种秉承 [5] 思路的拒绝方案）。

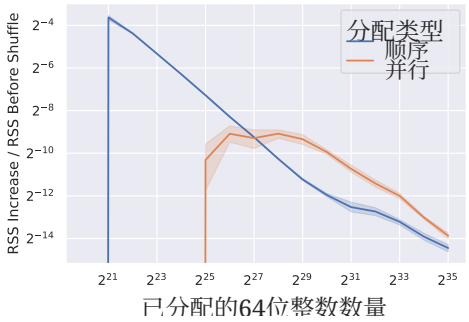
IpScShuf 和*PIpScShuf* 在规模小于 2^{18} 个元素的情况下使用FY 实现，r改用 32 位算术，并且通常从一个随机 64 位字中生成两个索引。

代码仓库包含高度优化的顺序和并行重实现的***MergeShuffle***，其中采用了与上述类似的技术。它们的性能无法与原始实现直接比较[10]：原始实现一方面使用手工编写的汇编代码进行合并，另一方面使用 `rdrand` 指令 [16] 获取随机比特；根据具体的处理器，这种方式比 `Pcg64Mcg`慢一到两个数量级。 [11, 27] 此外，这两种选择的可移植性都很差。总体而言，我们使用 `Pcg64Mcg` 的可移植实现比原始实现更快。请注意，***MergeShuffle*** 具有线性并行深度，因为只有独立子问题会并行执行，而第一递归层的合并完全按顺序进行。

[9] `https://github.com/rayon-rs/rayon`
[10] `https://github.com/axel-bacher/mergeshuffle`



(a) 内存分配的运行时间开销



(b) *PIpScShuf* 实现的内存使用量

图4 按照第6.1节所述测得的内存运行开销和内存使用量。

6 实证评估

本节中，我们将考察多种洗牌算法在不同设置下的性能。除非另有说明，我们使用以下标准参数：

- 测量是在一台配备*AMD EPYC 7702P* 处理器的机器上进行的，该机器拥有64个处理器核心和128个硬件线程、512 GB RAM，运行*Ubuntu 20.04.5*, `rustc1.68.0-nightly (92c1937a9)` and `gcc-10`。我们使用标准发布版本（`cargo-release`, `g++ -O3`），不进行针对特定机器的优化。
- 实验侧重于快速伪随机数生成器 `Pcg64Mcg`，因为这种选择能凸显洗牌算法本身而非生成器的开销。在图6中，我们报告了不同生成器的性能。对于不同的随机源，算法的相对性能总体上保持相似；不过，与*FY* 变体相比，*IpScShuf* 和*PIpScShuf* 受生成器选择的影响较小。
- 实验侧重于64位整数，这似乎是数百 GiB 数据集中常见的索引大小。如图 7所示，吞吐量（以字节/秒为单位）会随着元素增大而提高，因为每个元素的开销随之降低。同样，与*FY* 变体相比，*IpScShuf*和*PIpScShuf* 的差异范围较小。
- 报告的所有性能测量结果均为至少五次运行的均值。为减少系统误差，单次运行取 N 次重复的平均值，其中 N 的选择方式是确保测得的时间超过 100 毫秒。连续重复时会使用较大内存区域中的不同位置，以模拟冷启动，此时输入尚未缓存。

6.1 内存使用量和分配成本

这项工作的一个重要动机，是分配大量内存所产生的运行时开销，我们在图4a中对其进行了量化。每次测量时，我们使用低级 `libc::malloc`指令获取一定量的数据，初始化数据，然后通过 `libc::free`返回数据。对于大量数据， `malloc` 会请求操作系统将特定大小的内存映射到虚拟内存中。关键在于，只有实际访问内存时，才会为其分配物理内存。因此，我们将数据初始化两次，并从第一次的测量结果中减去第二次的结果。两次运行的差值就是系统提供物理内存所需的时间（不包括初始化时间）。我们的测量结果还表明，并行写入数值并不能很好地扩展——尽管投入了128个线程，初始化的加速比也只有4.9；由于 `malloc` 和 `free` 是顺序执行的，整个过程的加速比更是降至3.1。作为参考，我们还纳入了

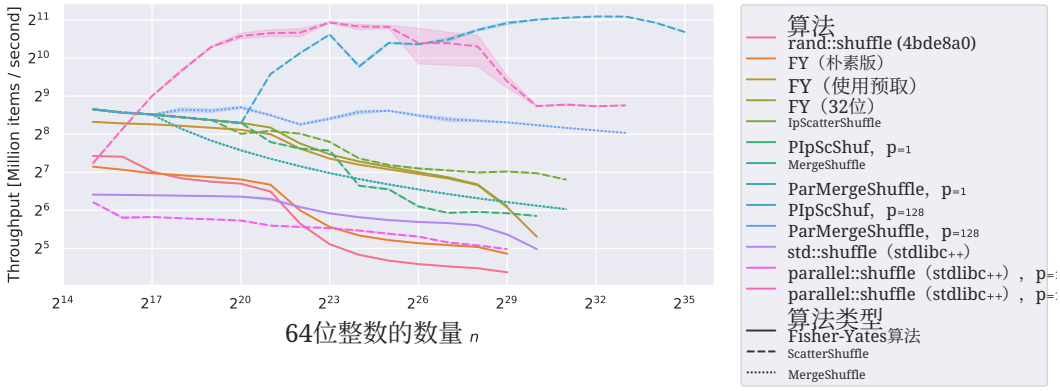


图5 每次运行时间预算为 30 秒时，几种洗牌算法的性能。*FY*、*IpScShuf*和*PIpScShuf* 是我们自行实现的。`std::shuffle` 和 `parallel:shuffle` 以 C++ 实现。对于并行算法，我们将可用处理器核心数标示为 p。

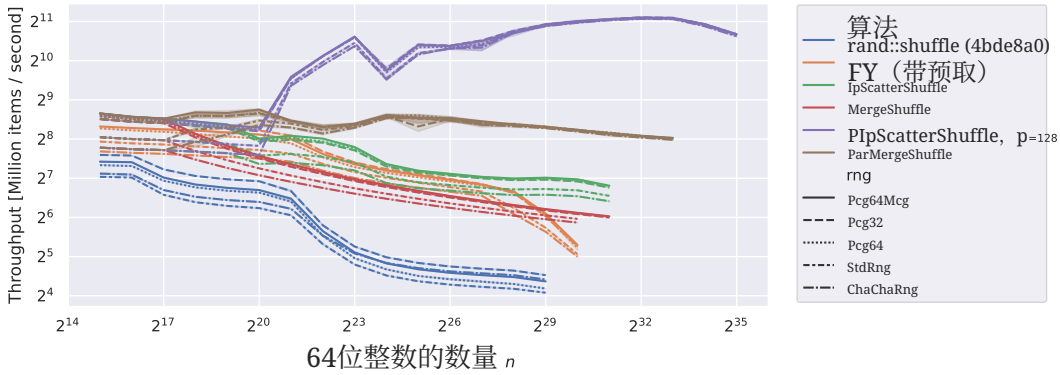


图6 使用不同伪随机数生成器时，所选算法的性能。

PIpScShuf 的运行时间，并发现，并行随机重排数据所花的时间，大致相当于获取存储一份副本所需的内存所花的时间——但并未实际复制数据。

在图4b中，我们报告了*PIpScShuf*的实际内存使用量。我们为每次运行启动一个独立进程，并测量该进程的最大驻留集大小（RSS）（即执行期间任何时刻获得物理内存支持的最大内存量）。我们在调用*PIpScShuf* 前后测量 RSS，并报告其相对增长。如第5节所述，`rayon`的工作池只需初始化一次。如果我们按顺序分配数据，*PIpScShuf* 会隐式建立一个工作池，导致执行 1631 次堆操作，以预留总计 923 KiB 的空间。另一方面，并行分配后不会执行任何堆操作。即便如此，对于较大的输入数据，我们仍观察到 RSS 略有增加——这似乎是由 128 个活动线程不断增长的栈内存造成的。在这种情况下，观察到的最大增幅为 0.2%，且输入数据非常大时增幅会减小。

6.2 性能概览

在图5中，我们报告了几种洗牌实现的性能。每次运行时，我们都将超时时间设为 30 秒，并在图中的算法达到该时限后停止运行。唯一的例外是我们的 *PIpScShuf* 实现，它在最大数据点上的运行时间为 20.8 秒。

速度最快的两种算法分别是 `parallel::shuffle`（以 C++ 实现的*ScatterShuffle*

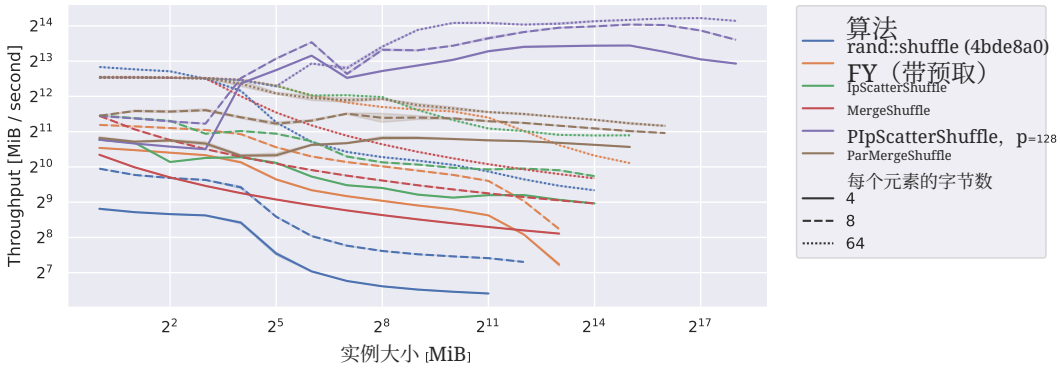


图 7 所选算法在不同元素大小下的性能。

包含在 `stdlibc++` 中）以及我们的 *PIpScShuf*。对于相对较小的输入数据，*ScatterShuffle* 速度更快；当输入数据大于 256MiB 时，*PIpScShuf* 则更胜一筹。

所有算法在输入数据规模较大时，吞吐量都会下降。对于*FY* 衍生算法而言，这一点尤为明显，因为其预测运行时间为线性。其速度变慢可归因于缓存未命中以及内存层次结构的相关影响。通过比较 `FY (naive)` 和`FY (with prefetch)`，我们证明，提前显式预取内存位置可以隐藏内存延迟。[11] 不过，预取只能模拟稍大的缓存；在约 2 GiB 处，我们仍能观察到显著的性能下降。Rust 中事实标准的洗牌算法（`rand::shuffle`）速度明显变慢，这是由于近期针对小范围数据的优化所致。该优化会尽可能从一个随机 32 位字中获取更多随机索引，因此在输入规模较大时效率较低。

正如预测的那样，所有基于 *ScatterShuffle* 的实现，其运行时间都与 $\log_k(n)$ 成正比；在单个处理器核心上执行 *PIpScShuf* 时，这一点尤为明显。与专门的顺序版本相比，并行实现受到的影响更大，因为每次递归都会产生额外开销，用于将较大的子问题拆分为相互独立的任务。

由于软件库不兼容，我们无法在测试中纳入 [14] 的性能测量。不过，图10显示，在四核笔记本电脑（i7-8550U）上，*PIpScShuf* 的吞吐量高于 [14, Table 6] 在配备72个处理器核心的四路服务器（E7-8867 v4）上的吞吐量——而 *PIpScShuf* 使用的内存更少，且其计算时间包含了随机比特的生成时间。

6.3 顺序算法的并行执行

为 *PIpScShuf*选择合适的基本情况算法时，图5可能会产生误导，因为它报告的是单独执行顺序算法时的性能。在这种情况下，与多个实例在独立数据上并行执行时相比，所研究的算法可用资源要多得多。这在其他场景中也可能很重要，例如，某个在“云端”运行的顺序数值模拟，可能会与其他内存密集型任务同时运行，而用户对此并不知情。

图8的记录方式与图5类似，不同之处在于我们并行执行128个相互独立的任务，并将这些任务各自运行时间的均值作为一次运行的结果。为避免调度伪影，我们会舍弃并重复任何一次实验，只要该次实验的运行时间（即从第一个线程启动到最后一个线程终止）比各个任务的均值运行时间长20%。

[11]我们使用环形缓冲区生成并预取随机索引，在实际交换之前提前 16 次交换。

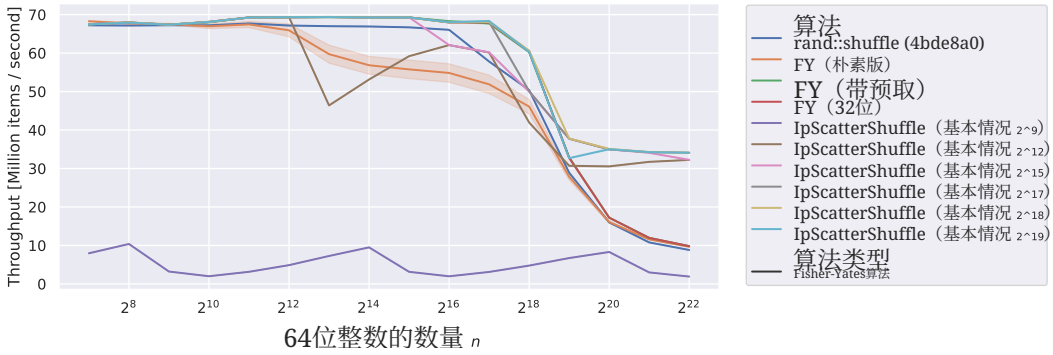
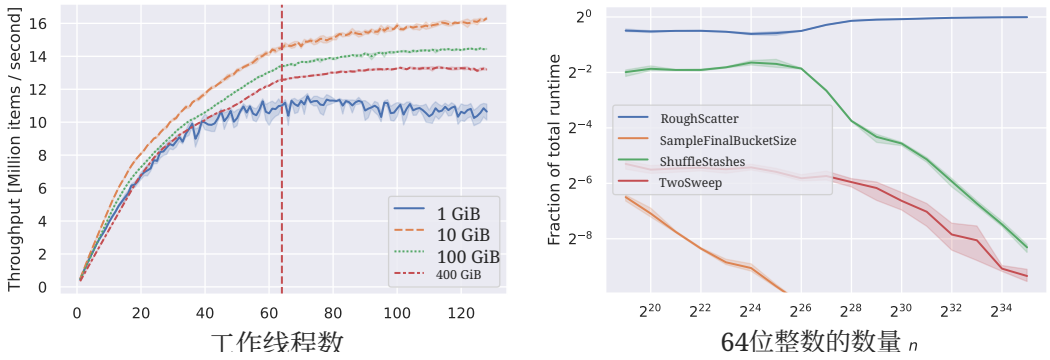


图8 在不同数据上并行运行多种顺序洗牌算法的性能。



(a) *PIpScShuf*的强扩展。竖线表示物理核心数。**(b)** 使用128个处理器核心时，*PIpScShuf* 第一层递归的运行时间占比。

图9 并行原地 ScatterShuffle的并行性能。

实验（即从第一个线程启动到最后一个线程终止）比各个任务的均值运行时间大 20%。

在这种情况下，我们观察到内存成为主要瓶颈；除少数例外（例如基本情形大小过小）外，对于小于1 MiB的数据，所有算法的性能大致相同（CPU拥有256 MiB的三级缓存，现由128个线程共享）。对于更大的数据，*IpScShuf* 变体的速度是*FY* 变体的3倍以上。还可以看到，与内存局部性的重要性相比，预取或基本情形大小等实现细节的影响微不足道。

6.4 并行扩展

为了量化*PIpScShuf*的并行加速比，我们进行如下强扩展实验。对于固定的输入大小，我们以 *IpScShuf* 和最快的*FY* 实现作为基准，然后测量*PIpScShuf* 在工作线程数不断增加时的表现。在图 9a 中，我们报告了相对于快速顺序实现的并行加速比（所有情况下均为*IpScShuf* ）。

对于 10 GiB 及以上的数据集大小，相对于最快顺序方案的加速比最高接近 16。自身加速比更高（例如，100 GiB 时为 32.2），表明其可扩展性良好，但并行实现带来的额外开销在一定程度上抵消了这一优势。对于同一实例，*PIpScShuf* 的速度比费希尔–耶茨洗牌快 142 倍。

随着工作线程数增加，速度显著提升，直到工作线程数达到物理核心数

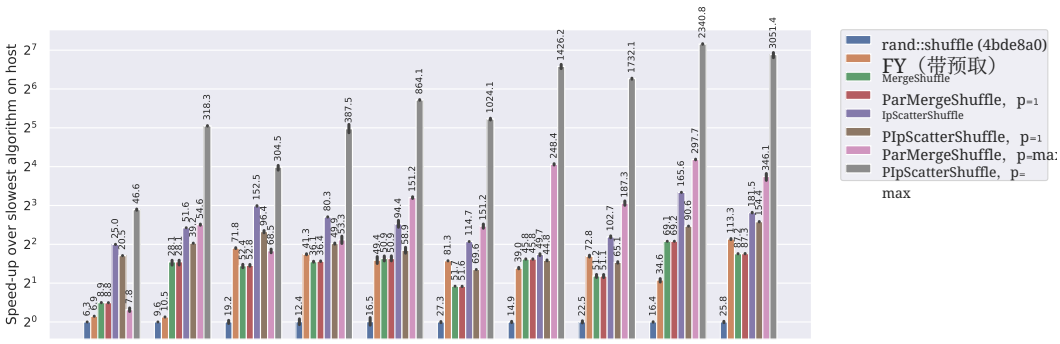


图10 在不同计算机上运行所选算法对64位整数进行随机重排的性能。条形长度表示相对于该系统上最慢算法的加速比。条形上方的数字表示绝对吞吐量，单位为每秒百万个元素。

；使用虚拟核心（同时多线程）几乎没有影响。这是意料之中的，因为随机重排受内存限制，而虚拟核心主要有助于充分利用超标量处理器的算术单元，但不会影响内存性能。

6.5 不同机器上的性能

为验证我们的实证发现是否适用于现代计算机，我们在图10中量化了不同机器上的随机重排性能。这些机器种类多样，从单板计算机、笔记本电脑到双路服务器不等（购买价格相差超过两个数量级）。它们采用不同的指令集、微架构，来自不同的处理器制造商，核心数量也各不相同。

为适应大多数系统，我们选择了 10 GiB 的实例规格，Raspberry PI 4B 除外，它仅配备 4 GiB 主存。测量结果包括顺序算法的运行、并行算法的顺序运行（以 $p = 1$ 表示），以及每个硬件线程配备一个工作线程的并行运行（以 $p =$max 表示）。最快系统的最大吞吐量是最慢系统的 50 倍。在所有情况下， `rand::shuffle` 都是表现最差的方案，*IpScShuf* 是最快的顺序实现，而 *PIpScShuf* 则是整体上最快的解决方案。

6.6 子问题的相对性能

在设计、实现和分析 *IpScShuf* 与 *PIpScShuf* 时，我们重点关注快速的机会式 *RoughScatter* ，随后需要通过额外的*FineScatter* 后处理来处理少数剩余项。尽管我们对 *RoughScatter*进行了大量优化，但在*FineScatter*中，我们选择了简单且易于实现的方案。

为从实证角度支持这一设计决策，我们测量了*PIpScShuf* 在各种输入大小下第一递归层的运行时间。随后，我们在图9b中报告了构成该层的四个子算法的相对墙钟时间。在我们的实现中，只有*RoughScatter* 采用并行方式执行，并提供128个工作线程，而其余部分

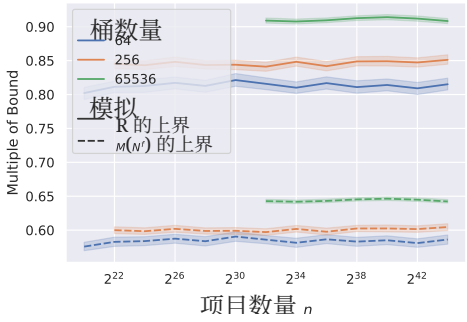


图 11 引理 2 和推论 6 的模拟。

是顺序算法。尽管可用工作线程数量存在不对称性，我们观察到，对于$n \geq 2^{27}$ *ParRoughScatter* 占据了超过90%的运行时间（对于 $n \geq 2^{33}$则为99%）。这支持了我们的设计决策，因为优化*FineScatter* 只能带来递减的收益。

7 隐藏常数的量化

在引理 2 中，我们证明经过*RoughScatter* 后，以高概率仍处于暂存状态的项数为 $R \leq \sqrt{2nk}\log k$。此外，在推论 6 中，我们证明了*TwoSweep* 的复杂度为$\mathcal{O}\big(k\sqrt{nk}\log k\big)$。为了提供实证依据并研究隐藏常数，我们对这两个过程进行了模拟。在图 11 中，我们报告了~~1000 次独立运行的均值~~，分别除以$\sqrt{2nk}\log k$和 $k\sqrt{nk}\log k$。请注意，我们的实现使用了 $k = 64$和$k = 256$；此外，我们还模拟了 $k = 2^{16}$，将其作为面向可预见未来的宽松上界。图 11 中可见的 k 小幅增长，是由于 k 值较小。对于最大为 2^{28} 的 k 进行的模拟结果与引理 2 一致，并趋近于推论 6 中的 0.66 倍。

参考文献

1 Daniel Allendorf、Ulrich Meyer、Manuel Penschuck、Hung Tran 和 Nick Wormald。针对给定幂律度序列的图进行均匀采样的工程实现。载于 *ALENEX*，第 27–40 页。SIAM，2022 年。2 Axel Bacher、Olivier Bodini、Alexandros Hollender 和 Jérémie O. Lumbroso。MergeShuffle：一种极快的并行随机排列算法。载于 *GASCom*，收入第 2113 卷 *CEUR Workshop Proceedings*，第 43–52 页。CEUR-WS.org，2018 年。3 Edward A. Bender 和 E. Rodney Canfield。给定度序列的标记图渐近数量。载于《组合论杂志 *A*辑》，24(3)：296–307，1978 年。4 Jon Louis Bentley、Dorothea Haken 和 James B. Saxe。求解分治递推式的一般方法。载于 *SIGACT News*，12(3)：36–44，1980 年。5 Petra Berenbrink、David Hammer、Dominik Kaaser、Ulrich Meyer、Manuel Penschuck 和 Hung Tran。以每次交互亚常数时间模拟群体协议。载于 *ESA*，收入第 173 卷 *LIPIcs*，第 16:1–16:22 页。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2020 年。6 Guojing Cong 和 David A. Bader。对对称多处理器上的并行随机排列算法进行实证分析。载于 *PDCS*，第 27–34 页。ISCA，2005 年。7 Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein。《算法导论》，第 *3* 版。麻省理工学院出版社，2009 年。8 Luc Devroye。《非均匀随机变量生成》。施普林格，1986 年。9 Edsger W. Dijkstra。《程序设计的学科》。普伦蒂斯霍尔，1976 年。

10 Thomas S. Ferguson。谁解决了秘书问题？《统计科学》，4(3)：282–289，1989 年。11 Agner Fog。指令表。网址： https://www.agner.org/optimize/instruction_tables.pdf。12 Daniel Funke、Sebastian Lamm、Ulrich Meyer、Manuel Penschuck、Peter Sanders、Christian Schulz、Darren Strash 和 Moritz von Looz。无通信大规模分布式图生成。《并行与分布式计算杂志》，131：200–217，2019 年。13 Hermann Gruber、Markus Holzer 和 Oliver Ruepp。慢速排序：对糟糕透顶的随机排序算法的分析。载于 FUN，收入《计算机科学讲义》第 4475 卷，第 183–197 页。施普林格，2007 年。14 Yan Gu、Omar Obeya 和 Julian Shun。并行原地算法：理论与实践。载于 APOCS，第 114–128 页。SIAM，2021 年。15 Chris Hinrichs、Vamsi K Ithapu、Qinyuan Sun、Sterling C Johnson 和 Vikas Singh。加速神经影像学中的置换检验。载于 C. J. C. Burges 等编，《神经信息处理系统进展》，第 26 卷，第 890–898 页。Curran Associates, Inc.，2013 年。16 英特尔公司。Intel 64 和 IA-32 架构软件开发人员手册，2022 年。17 Donald E. Knuth。《计算机程序设计艺术》第二卷：半数值算法，第 2 版。Addison-Wesley，1981 年。18 Charles E. Leiserson。使用 Cilk 编写不规则并行应用程序。载于 IRREGULAR，收入《计算机科学讲义》第 1253 卷，第 61–71 页。施普林格，1997 年。19 Daniel Lemire。区间内随机整数的快速生成。《ACM 建模与计算机仿真汇刊》，29(1)：3:1–3:12，2019 年。20 Yossi Matias、Jeffrey Scott Vitter 和 Wen-Chun Ni。离散随机变量的动态生成。《理论计算机系统》，36(4)：329–358，2003 年。21 Peter M. McIlroy、Keith Bostic 和 M. Douglas McIlroy。基数排序的工程实现。《计算机系统》，6(1)：5–27，1993 年。22 SJ Meyer。结构化程序设计的失败，Zilog 公司。技术报告，软件部技术报告，1979 年。23 Richard Nixon。第11497号行政命令——修订征兵条例以规定随机选拔，1969 年。24 Melissa E. O’Neill。PCG：一族简单、快速、节省空间且统计性能良好的随机数生成算法。技术报告 HMC-CS-2014-0905，哈维穆德学院，美国加利福尼亚州克莱蒙特，2014 年 9 月。25 Manuel Penschuck、Ulrik Brandes、Michael Hamann、Sebastian Lamm、Ulrich Meyer、Ilya Safro、Peter Sanders 和 Christian Schulz。可扩展网络生成的最新进展。CoRR，abs/2003.00736，2020 年。26 Martin Raab 和 Angelika Steger。“球入箱”——一种简单而精确的分析。载于 RANDOM，收入《计算机科学讲义》第 1518 卷，第 159–170 页。施普林格，1998 年。27 Matthew Route。射电耀发超冷矮星种群合成。《天体物理学杂志》，845(1)：66，2017 年 8 月。 doi:10.3847/1538-4357/aa7ede。28 Peter Sanders。分布式、外部及层次化存储器上的随机排列。《信息处理快报》，67(6)：305–309，1998 年。29 Julian Shun、Yan Gu、Guy E. Blelloch、Jeremy T. Fineman 和 Phillip B. Gibbons。顺序随机排列、链表收缩和树收缩都具有高度并行性。载于 SODA，第 431–448 页。SIAM，2015 年。30 Johannes Singler 和 Benjamin Konsik。GNU libstdc++并行模式：软件工程方面的考量。载于 IWMSE@ICSE，第 15–22 页。ACM，2008 年。