Engineering Shared-Memory Parallel Shuffling to Generate Random Permutations In-Place

Manuel Penschuck
Goethe Universität Frankfurt, Germany

Abstract

Shuffling is the process of placing elements into a random order such that any permutation occurs with equal probability. It is an important building block in virtually all scientific areas. We engineer, – to the best of our knowledge – for the first time, a practically fast, parallel shuffling algorithm with $\mathcal{O}\left(\sqrt{n}\log n\right)$ parallel depth that requires only poly-logarithmic auxiliary memory (with high probability). In an empirical evaluation, we compare our implementations with a number of existing solutions on various computer architectures. Our algorithms consistently achieve the highest through-put on all machines. Further, we demonstrate that the runtime of our parallel algorithm is comparable to the time that other algorithms may take to acquire the memory from the operating system to copy the input.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Shuffling, random permutation, parallelism, in-place, algorithm engineering, practical implementation

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.5

Supplementary Material *Software (Source Code)*: `https://crates.io/crates/rip_shuffle`
Software (Source Code and Raw Data): `https://zenodo.org/record/7876820`

Funding *Manuel Penschuck*: Supported by the Deutsche Forschungsgemeinschaft (DFG) under grant ME 2088/5-1 (FOR 2975 – Algorithms, Dynamics, and Information Flow in Networks).

1 Introduction

Random permutations are heavily studied in many fields of science with numerous applications. They are commonly considered an "easy and fair" arrangement and thus influence many aspects of everyday life ranging from shuffling a deck of cards in a friendly game to determining the fateful order in which soldiers are drafted for war (e.g., [24]).

In computer science, applications include numerical simulations, sampling of complex objects, such as random graphs, machine learning, or statistical tests (e.g., [4, 16, 20, 26]). Especially, if coupled with rejection sampling, shuffling can become a dominating subroutine (e.g., [1] which triggered this work). Further, the assumption that an input is provided in random order (instead of adversarially) allows for practical algorithms that are almost always efficient. Among others, this notion motivates the random-order-model for online algorithms [11]. For the same reason, implementations of offline algorithms may start by shuffling their inputs; for instances, folklore suggests to shuffle the input before sorting it with a simple *Quicksort* implementation.

From an algorithmic point of view, the tasks of *shuffling* and *sorting* are tightly connected since both require an algorithm capable of emitting any permutation. Though, while sorting needs to handle adversarial inputs, shuffling can be optimized for the well-behaved uniform distribution. Shuffling can be implemented in linear-time via integer sorting by augmenting each input element with a uniform variate and sorting by it [7]; we refer to this approach as *SortShuffle*. The famously impractical *BogoSort* demonstrates the other direction, namely sorting by shuffling, but suffers from a "slightly" suboptimal expected runtime of $\Omega(n \cdot n!)$ [14].

The quest for *in-place* algorithms is driven by the various costs of memory. The most obvious aspect is that the maximal data set size that can be handled by a machine roughly halves if the output is produced in a copy. Further, it takes a considerable time to allocate main memory on modern computer systems; in Section 6 we demonstrate that the runtime of our shuffling algorithm is comparable to the time it takes to acquire additional memory of input size. Another hidden cost is the increased code complexity to handle failed allocations of dynamic memory (e.g., because the system ran out of memory). Finally, some programming languages have a concept of non-copyable data; e.g., in C++ the copy-constructor can be deleted, and in Rust data types need to explicitly declare that they can be cloned.

1.1 Our contributions

We design and implement the practical shared-memory parallel algorithm *Parallel In-Place ScatterShuffle* (*PIpScShuf*). Our contributions include:

- The algorithm is an in-place modification of *ScatterShuffle* [29]. Instead of directly putting elements to random positions, *ScatterShuffle* instead assigns the elements to random buckets and recurses on them until eventually a random permutation is achieved.
- We show that our *PIpScShuf* has (whp) a parallel depth of $\mathcal{O}(\log(n)\sqrt{nk/\log k})$ and uses $\mathcal{O}(n \log_k(n))$ work (see Section 2.1 for definitions) where k is small tuning parameter.
- While it is straightforward to implement *ScatterShuffle* in-place using standard techniques (e.g., by sampling the buckets sizes as a multinomial followed by weighted sampling to distribute the elements [26]), we design a multi-staged assignment process for practical performance inspired by *MergeShuffle* [3].
- We first assume that all buckets have the same sizes and randomly assign most elements very efficiently. We then show that an asymptotically negligible and practically cheap repair step can produce the required original random distribution.
- We provide fast shuffle implementations in a free and well-tested plug-and-play Rust library. Our *PIpScShuf* does not use heap allocations and emits reproducible permutations if a seedable pseudo-random number generator is provided.

After a discussion of notation and related work in Sections 2 and 3, we derive the sequential *In-Place ScatterShuffle* (*IpScShuf*) in Section 3 and parallelize it in Section 4. In Section 5, we discuss details of our implementations which we then evaluate in Section 6.

2 Preliminaries and notation

The expression $(x_i)_{i=a}^{b}$ denotes the sequence $x_a, \ldots, x_b$ and may be shortened to $(x_i)_i$ if the limits are implied by context. We indicate an array of n elements as $X[1..n]$ and reference the subrange $X[i], \ldots, X[j]$ as $X[i..j]$. Further, $[n]$ denotes the set $\{1, \ldots, n\}$. Then, a permutation is a bijection $\pi\colon [n] \to [n]$ where $\pi(i)$ encodes the position of the i-th input element in the output. We say that a probabilistic statement holds *with high probability* (whp) if the error probability is at most $1/n$ for some implied parameter n.

2.1 Parallel model of computation

For parallel algorithms, we assume the commonly accepted binary Fork-Join model [8]. This choice fits the `rayon`[1] infrastructure used in our implementation well. An execution starts with a single task on a unit-cost random access machine. Additionally, any task t_0 can

[1] `https://crates.io/crates/rayon`

Algorithm 1 Fisher Yates Shuffle on input $A[1..n]$. The array eventually holds the output.

```
for i in 1 to n−1 do
    j ← uniform sample from [i..n]      // A[1..i − 1] already have final values
    swap elements A[i] and A[j]
```

recursively *fork* into two tasks t_1 and t_2. In this case t_0 waits until t_1 and t_2 complete their computation and *join* to resume t_0. In practice, Fork-Join frameworks, such as oneTBB[2], Cilk[3] [19], or rayon, use a worker-pool in combination with a work-stealing scheduler to map tasks to cores. Algorithmic performance measures are the *work*, i.e. the total number of instructions, and the *parallel depth* (*span*), defined as the length of the critical path which corresponds to the execution time assuming an unbounded number of workers.

2.2 Random shuffling

The sequential *Fisher-Yates-Shuffle* (*FY*, also know as *Knuth-Shuffle*) [18] obtains a random permutation of an array $A[1..n]$ in time $\mathcal{O}(n)$. As summarized in Algorithm 1, conceptually, it places all items into an urn, draws them sequentially without replacement, and returns the items in the order they were drawn. The algorithm works in-place and fixes the value of $A[i]$ in iteration $i \in [1...n-1]$ by swapping $A[i]$ with $A[j]$ where j is chosen uniformly at random from the not yet fixed positions $[i..n]$. In other words, in the i-th iteration, the $(i-1)$-prefix of A stores the result obtained so far, while the $(n-i)$-suffix represents the urn.

Shun et al. show that this seemingly inherently sequential algorithm exposes sufficient independence to be processed with logarithmic parallel depth (whp) [30]. Later, Gu et al. propose an in-place variant based on the so-called decomposition property of the parallel *FY* [15]. However, both algorithms are designed to solve a subtly different problem. They *permute* the input in an explicitly prescribed manner. As such, the permutation is part of the input and the implementation[4] of [15] uses two additional pointers per element (i.e. shuffling 32 bit values on a 64 bit machine leads to a five-fold increase of memory).

A random permutation can be computed in parallel by $\mathfrak{P}$ processors by assigning each element to one of $\mathfrak{P}$ buckets uniformly at random and then applying the sequential algorithm to each bucket [29]. We refer to this algorithm as *ScatterShuffle* and discuss it in detail in Section 3. A similar technique yields an I/O-efficient random permutation algorithm [29].

Going the opposite direction also yields an efficient algorithm. *MergeShuffle* first assigns each processor a contiguous section of the input array, shuffles the subproblems pleasingly parallel and finally recursively merges them to obtain a larger random permutation [3]. Here, merging of two input sequences A and B exploits that A and B were previously shuffled. Hence, the relative order of elements from A (and B respectively) can be kept in the output. In other word, the merging phase conceptually produces a $|A|+|B|$ bit vector with exactly $|B|$ ones. If the i-th zero is at position j, we place $A[i]$ to the j-th output position (analogously for ones and B).

This merging can be interpreted as the inverse of *ScatterShuffle*'s scatter with two buckets. In a precursor study, we found it too slow to generalize *MergeShuffle* using k-way merging which is needed to reduce the recursion depth. *MergeShuffle* further uses a sequential merge

[2] previously known as Intel Threaded Building Blocks, `https://github.com/oneapi-src/oneTBB`
[3] see also `https://www.opencilk.org`, `http://cilkplus.org`
[4] `https://github.com/ucrparlay/PIP-algorithms` master at time of writing (`6af1df9`)

procedure and we are unaware of a parallelization that is as efficient as our *PIpScShuf* based on *ScatterShuffle*. However, the authors show that if $|A| \approx |B|$, we can assign the positions of all but expected $\mathcal{O}\left(\sqrt{|A| + |B|}\right)$ elements using a single random bit. A generalization of this insight is a crucial building block for our *RoughScatter* routine in Section 3.3.

Cong and Bader [7] empirically study additional techniques such as shuffling using sorting algorithms (*SortShuffle*) or random dart-throwing (*DartThrowingShuffle*). We are, however, unaware of how to implement these approaches in-place. Yet, it is worth pointing out that our *IpScShuf* algorithm can be interpreted as an optimized in-place RadixSort in which buckets are randomly drawn. As such, there are conceptual similarities to *SortShuffle*. In a precursor study, we found that even highly optimized parallel and in-place sorting algorithms, such as IPS2RA [2], are outperformed by our *PIpScShuf* implementation. This can be attributed to the fact that sorting is a much more constraint problem, while shuffling can algorithmically exploit the features of uniform permutations.

2.3 Sampling from discrete distributions

In the following, we sample from several discrete probability distributions (arguably, shuffling is just that). This is achieved by first obtaining a stream of independent and unbiased random bits that are subsequently reshaped to attain the required distribution. The default way of implementing the first step is using a pseudo-random generator, such as Pcg64Mcg [25].

Sampling an integer from $[0, s)$ with $s = 2^k$ for some $k \in \mathbb{N}$ from random words is very cheap and involves only shifting and masking. We adopt rejection-based algorithms with expected constant time to sample uniform variates from $[0, s]$ for general s (see [20]) and binomial variates (see [9]). Sampling of k-dimensional multinomial variates is implemented by chaining appropriately parametrized binomial samples in expected time $\mathcal{O}(k)$.

3 Sequential in-place shuffling

In this section, we propose *In-Place ScatterShuffle* (*IpScShuf*), a sequential in-place variant of Sanders' parallel *ScatterShuffle* [29]. Building on the performance results obtained, we reintroduce parallelism in Section 4.

3.1 State of the art

It seems that the simple and fast *Fisher-Yates Shuffle* (*FY*, see Section 2.2) is the shuffle algorithm most commonly used in practice. Due to its simplicity, the algorithm typically outperforms more advanced schemes for small inputs. However, *FY*'s unstructured accesses to main memory cause a severe slowdown for larger inputs. This is especially relevant for parallel algorithms where the memory subsystem is shared between cores (see Section 6).

ScatterShuffle (Algorithm 2) is designed to be a parallel algorithm that also fares well in the external memory model [29]. Given an input $(x_i)_{i=1}^n$, the algorithm moves each input element x_i into a bucket drawn independently and uniformly from $B_1, \ldots, B_k$. Afterwards, each bucket constitutes an independent subproblem of expected $\Theta(n/k)$ elements on which we recurse. For small subproblems, we switch to *FY* as the base case algorithm.

While we refer to [29] for a formal correctness proof, the following intuition should suffice to follow this article. Consider that we augment each input element x_i with a random integer r_i chosen uniformly from $[0; 2^\ell)$ where ℓ is sufficiently large such that all r_i are unique. Then, we use RadixSort to order the elements according to these random keys (starting with

Algorithm 2 Sequential variant of *ScatterShuffle*. The buckets' total size is $\Theta(n)$.

```
1  Function ScatterShuffle(X = [x_1, ..., x_n], k)     // X is modified in-place
2      if n is small then                               // Base case for small inputs
3          FisherYates (X) and return
4      Initialize empty buckets B_1, ..., B_k
5      for x ∈ X do                                     // Assign elements to buckets
6          copy x into B_j where B_j is uniformly chosen from B_1, ..., B_k
7      s ← 1
8      for B_j ∈ {B_1, ..., B_k} do                     // Recurse and overwrite input X
9          ScatterShuffle(B_j, k)
10         X[s..(s+|B_j|)] ← B_j
11         s ← s + |B_j|
```

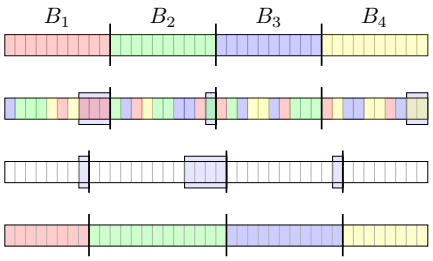
the most significant k-ary digit); this will yield a uniform permutation by construction. Now observe that the buckets in *ScatterShuffle* and RadixSort are treated analogously with the difference that *ScatterShuffle* samples the digits on-demand.

In the original parallel formulation of *ScatterShuffle*, the number of buckets k equals the number of processing units $\mathfrak{P}$ to expose the maximal degree of parallelism. Sanders, however, already discusses that in the presence of memory hierarchies, the parameter k should be chosen sufficiently small such that the individual processors can cache at least the tail of each bucket. In his implementation[5], the parameter is $k = 32$ for the largest runs in [29]. At time of writing – more than two decades later, using very different hardware to run experiments with more than three orders of magnitude larger data sets – we empirically find $k \leq 64$ to be the best choice for our *In-Place ScatterShuffle* over a wide range of input sizes. Hence, the parameter k should be intuitively treated as a small constant that governs primarily the branching factor of the recursion. In Section 4, we will add parallelism independent of k.

Our main modification to *ScatterShuffle* is *In-place Scatter* (*IpSc*) which scatters the input into k buckets. It effectively replaces lines 4–6 in Algorithm 2. Instead of copying the input into new arrays representing the buckets, the buckets become disjoint memory regions of the input (e.g., represented by two pointers). Then, the recursion (line 9) can directly modify each bucket's memory without copying the elements. Formally, let $X = (x_i)_{i=1}^n$ be the input of *IpSc*. Further, let $A = (a_i)_{i=1}^n$ be independent uniform variables from $[1, k]$ indexing into the aforementioned buckets. Then, *IpSc* groups X by A by rearranging the elements in X with some permutation π that sorts A. We exploit that the order of elements *within* a bucket can be arbitrary as the recursion will shuffle them randomly later on.

Similar problems have been studied in the context of integer sorting. The special case of $k = 2$ (i.e. binary partition) and $k = 3$ (known as the Dutch national flag problem) can be efficiently solved in-place [10, 23]. For $k > 3$, two-pass approaches can be used (e.g., *American flag sort* [22]) but require repeated access each a_i. The parallel implementation [31] of *ScatterShuffle* included in `libstdc++` uses this technique and stores A explicitly requiring $\Theta(n \log k)$ bits. Another way, in the spirit of [13], is to require a pseudo-random generator that can be replayed multiple times by copying and retrieving the generator's internal state.

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



Input X is a view (start/end pointers) to contiguous memory. Buckets $B_1, \ldots, B_k$ are views with an initial size n/k.

RoughScatter randomly assigns items to buckets until one is full (here B_3). Marked elements are *staged* (i.e. yet unassigned).

Sample the number of staged items per bucket. Then **TwoSweep** fixes boundaries, s.t. *placed* items remain in their assigned buckets.

After recursively shuffling the staged items, we recurse on each bucket independently.

Figure 1 *In-Place ScatterShuffle* (*IpScShuf*). The first three steps constitute *In-place Scatter* (*IpSc*). All operations are either pointer arithmetic or swapping of items. No input element is copied.

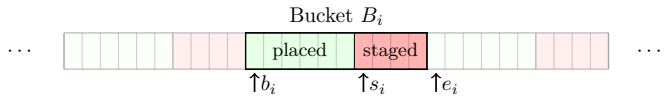


Figure 2 *IpSc* partitions the input into k buckets, each roughly containing n/k elements. Initially, all items are *staged* ($b_i = s_i$) and the bucket is said to be *empty*. Eventually, more and more items are placed (from the left). If $s_i = e_i$ the bucket is said to be *full*.

3.2 IpScShuf – an in-place implementation of ScatterShuffle

In the following, we describe *In-place Scatter* (*IpSc*) that supports true random bits, and, whp, runs in linear time using only $\mathcal{O}(k \log k)$ bits additional storage. Since each of the n items is assigned a uniformly selected bucket, the numbers $(n_i)_{i=1}^k$ of elements assigned to each bucket follow a multinomial distribution and are tightly concentrated around n/k.

For the remainder, we assume that $n \gg k(\log k)^3$ and $k^3 \log k = \mathcal{O}(n)$, since otherwise, the problem is so small that *Fisher-Yates Shuffle* is more appropriate. These assumptions are only needed to bound *IpScShuf*'s complexity and do not affect its correctness. In practice, they translate to a minimal recommended size of roughly 10^5 elements.

A straightforward solution is to draw the sizes of all buckets as a multinomial random variate. We then sample the buckets without replacement weighted by their decreasing target size. This can be implemented in expected linear time using suitable dynamic weighted sampling data structures (e.g., [21]). As discussed further in Section 5, such approaches are outperformed by the following scheme. Inspired by the framework of [3] (but quite different in its details), our assignment task consists of two phases that are illustrated in Figure 1. Firstly, during the *RoughScatter* phase, we very efficiently assign the vast majority of items – but almost certainly not all of them. Secondly, during the *FineScatter* phase, we process the remaining few elements.

3.3 RoughScatter – the opportunistic work horse

RoughScatter exploits the aforementioned concentration of the final bucket sizes around their mean of n/k to assign elements in an opportunistic fashion until we hit said n/k barrier. Let $X[1..n]$ denote the input array. As illustrated in Figure 2, we partition X into k contiguous buckets of equal sizes n/k (up to rounding). Each bucket B_i is stored as a triple of indices (b_i, s_i, e_i) where b_i points to the beginning of B_i and e_i beyond the bucket's end. A bucket is further subdivided into (i) an initially empty segment $X[b_i..s_i)$ of so-called *placed* items and (ii) $X[s_i..e_i)$ of so-called *staged* items. We say that bucket B_i is *full* iff all items are placed, i.e. $s_i = e_i$. Up to the last step in Section 3.4, *staged* items can be freely moved around, whereas the position of *placed* items carries meaning.

In each iteration, the algorithm finalizes the bucket assignment of the element x that s_1 points to, i.e. the first staged element in B_1 at that point in time. To this end, we randomly draw a partner bucket j uniformly from $[1..k]$, swap the elements $X[s_1] \leftrightarrow X[s_j]$ (skipped if $j = 1$), and increment s_j. As a result, element x is moved into the *placed* region of the partner bucket B_j. If B_j is now full, the algorithm stops, otherwise it repeats.

▶ **Lemma 1.** *Let P be the set of elements placed by RoughScatter and let $x \in P$ be an arbitrary placed item. Then x is assigned to bucket B_i with probability $1/k$.*

Proof. Without loss of generality, we assume that initially all items are staged. Then, there is a unique iteration for each item $x \in P$ in which it gets placed. To this end, the then still staged element x is swapped with a staged item y where $y \in B_i$ with probability $1/k$. It then increases s_i and thereby defines x as placed. Since *RoughScatter* only swaps staged items, this placement of x is final. The possible change of position of element y is inconsequential, since each assignment is carried out independently. ◀

In the following, we bound the number elements that remain staged after *RoughScatter*.

▶ **Lemma 2.** *After a RoughScatter run, let $r_i = e_i - s_i$ be the number of items still staged in bucket b_i and $R = \sum_i r_i$ their sum. For $n \gg k(\log k)^3$, we have $R \leq \sqrt{2nk \log k}$ whp.*

Proof. We interpret the input to *RoughScatter* as n *balls* that are independently thrown into k uniform bins. If we run the balls-into-bins experiment to completion, the maximal load of any bucket is at most $M(n,k) = \frac{n}{k} + \sqrt{2\frac{n}{k}\log k}$ whp [27].[6]

Let n' be the number of balls assigned in said game when the maximal load first reached n/k. Algorithmically, this corresponds to the termination of *RoughScatter*. By identifying $M(n',k) = n/k$ and solving for n', we find that whp $n' \geq n - \sqrt{2nk \log k} := n - R$. ◀

▶ Remark 3. The fraction of unprocessed elements R/n vanishes for $n \to \infty$. Even for small inputs with $n = 2^{22}$ and practical $k = 64$, less than 1% of the input remains unassigned whp.

3.4 FineScatter – fixing the small remainder

After the execution of *RoughScatter* only $R = \mathcal{O}\big(\sqrt{nk \log k}\big)$ items need to be assigned during the *FineScatter* phase whp. If our initial assumption still holds for $R \gg k(\log k)^3$, we can compact the staged items into a contiguous memory area, apply *RoughScatter* and recurse. However, for small inputs the assumption is likely violated, while for large inputs the fraction R/n contributes only negligibly to the total runtime. Thus, we do not consider it worthwhile to devise a merging procedure for this case and instead directly use a dedicated base case algorithm based on the following Lemma. It lays out the route to efficiently obtain the final bucket sizes and independently assign the remaining elements.

▶ **Lemma 4.** *Let $X = (x_i)_{i=1}^n$ be a sequence and $N = (n_i)_{i=1}^k$ be sampled from a multinomial distribution with equal weights $p = 1/k$ such that $\sum_i n_i = n$. Let $f_N \colon [n] \to [k]$ be an arbitrary partition of X with class sizes N. Finally, let $\pi \colon [n] \to [n]$ be a random permutation. Then, for fixed i and j, the probability that element x_i is mapped by $f_N(\pi(i))$ to class j is $1/k$.*

Proof. Due to symmetry, it suffices to consider the first partition class $j = 1$. Its size n_1 follows a binomial distribution over n attempts with success probability $p = 1/k$ by definition of the multinomial distribution. Further, let γ be a permutation such that the composition

[6] A similar argument was already used in the analysis of *ScatterShuffle* [29].

$f_N \circ \gamma$ maps the indices $1, \dots, n_1$ to the first partition class. Due to uniformity, π itself and $\pi' = \pi \circ \gamma$ are equally likely. Thus, it suffices to compute the total probability that π' puts a fixed x_i into the first n_1 ranks over all $0 \le n_1 \le n$:

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

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

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

3.4.1 Finalizing the bucket sizes

Let $N = (n_i)_i$ be the numbers of elements assigned to bucket B_i by *RoughScatter*. Guided by Lemma 4, the base case algorithm first draws a multinomial variant $N' = (n'_i)_i$ where n'_i corresponds to the number of elements that will be placed into bucket B_i by *FineScatter*. Then, the final sizes $N^f = (n^f_i)_i$ are $n^f_i = n_i + n'_i$. Since N and N' follow a multinomial distribution with k equally weighted classes, their sum N^f does too.

By construction, the expected bucket size is n/k. Let $d_i = n^f_i - n/k$ denote the deviation of the size of bucket B_i, i.e. the number of elements it needs to gain over the initial estimation of *RoughScatter*. Analogously to the proof of Lemma 2, we bound $\max_i \{|d_i|\} = \mathcal{O}(\sqrt{n/k \log k})$ whp [27, 29]. Thus, in all likelihood, the bucket boundaries only move slightly.

Luckily, *ScatterShuffle* is oblivious to the order of elements within a bucket prior to recursion. Thus, it suffices to appropriately move a few items near the boundaries of the buckets using our *TwoSweep* algorithm. First, we iterate over the buckets in ascending index order. Meanwhile, we keep a counter $C_i = \sum_{j=1}^{i-1} d_j$ that indicates how many additional items are needed left of the current bucket B_i. If bucket B_i is too large by more than C_i items, we swap the excess staged items into the staging area of bucket B_{i+1}. In a second sweep from B_k to B_1, we move the remaining excess items towards smaller bucket indices.

▶ **Lemma 5.** *Let $N^f = (n^f_i)_i$ be the final bucket sizes, denote their deviation from the mean n/k as $d_i = n^f_i - n/k$, and let $D_i = \sum_{j=1}^{i} d_j$ be the inclusive prefix sum of deviations. Then, TwoSweep executes a total of $M(N^f) = \sum_i |D_i|$ swaps and takes time $\mathcal{O}\big(k + M(N^f)\big)$.*

Proof. *TwoSweep* can exchange a staged item of bucket B_i with its direct neighbors $B_{i\pm1}$ in $\mathcal{O}(1)$ time by executing a single swap and adopting the pointers of the two involved buckets; exchanging an item between buckets B_i and B_j this way implies a chain of $|i-j|$ swaps causing $\mathcal{O}(|i-j|)$ work. Based on this, *TwoSweep* carries out two snow-plow-like motions likely pushing intermediate items along the chain. For the remainder see Appendix C. ◀

▶ **Corollary 6.** *TwoSweep takes time $\mathcal{O}\big(k\sqrt{nk \log k}\big)$ whp.*

Proof. see Appendix C. ◀

3.4.2 Assigning the remaining staged elements

At this point in the execution, all buckets have reached their final sizes, but each bucket B_i still has n'_i staged items with $\sum_i n'_i = R$. Rather than sampling weighted by $N' = (n'_i)_i$, we apply Lemma 4 and instead randomly shuffle all staged items. This can be done by compacting the staged item into $X[1..R]$ and shuffling X. To this end, we swap the staged items with the items originally stored in X; after shuffling, we apply the same swap sequence in reverse to restore the original items and put the staged items into a now random permutation.

3.5 Putting it all together

In-place Scatter (*IpSc*) is the algorithm executed on each recursion layer of *IpScShuf*. It runs *RoughScatter* and *FineScatter* in sequence to randomly assign n elements to k buckets. The input is rearranged such that each bucket corresponds to a contiguous memory region.

▶ **Lemma 7.** *For $n \gg k \log^3 k$ and $k^3 \log k = \mathcal{O}(n)$, IpSc assigns n items in time $\mathcal{O}(n)$ whp.*

Proof. We sum up the four tasks carried out:

1. *RoughScatter* first partitions the input into buckets in time $\mathcal{O}(k)$ and then randomly assigns $n - R = \mathcal{O}(n)$ elements whp. Assuming that obtaining a word of randomness takes constant time, this translates into a time complexity of $\mathcal{O}(k + n) = \mathcal{O}(n)$.
2. Sampling a k-dimensional multinomial random variate takes time $\mathcal{O}(k)$ whp.
3. Running *TwoSweep* to adjust the boundary size takes time $\mathcal{O}\big(k\sqrt{nk \log k}\big) = \mathcal{O}(n)$ whp.
4. Shuffling the staged items with *Fisher-Yates Shuffle* takes time $\mathcal{O}(R) = \mathcal{O}(n)$ time.[7] ◀

In-Place ScatterShuffle consists of recursive applications of *IpSc*. We stop the recursion on a subproblem as soon as it reaches the base case size of $N_0 = \mathcal{O}(1)$ at which point it is finalized using *Fisher-Yates Shuffle*.

▶ **Theorem 8.** *With high probability In-Place ScatterShuffle takes time $\mathcal{O}(n \log_k(n/N_0))$ and $\mathcal{O}(k \log_k(n/N_0))$ additional words of storage where $N_0 = \Omega(k^3)$ is the base case size.*

Proof. *IpScShuf* splits an input of length n into k independent subproblems of size $\Theta(n/k)$ whp. It then calls itself recursively until the base case size of N_0 is reached. Whp, this involves $\mathcal{O}(\log_k(n/N_0))$ recursion layers, each taking time $\mathcal{O}(n)$ and requiring $\mathcal{O}(k)$ words of memory for a depth-first traversal. The base case *FY* uses $\mathcal{O}(1)$ words of memory and takes time $\mathcal{O}(n')$ for a subproblem of size n' and in total $\mathcal{O}(n)$ for all subproblems. ◀

4 Parallel algorithms

In this section, we introduce *Parallel In-Place ScatterShuffle* (***P****IpScShuf*), a parallel variant of *IpScShuf*. It is obvious that after running *IpSc* (i.e. a single recursion level of *IpScShuf*) we can process the k independent subproblems pleasingly in parallel – this is one of the core insights of the original *ScatterShuffle* [29]. Unfortunately, in our case, parallelizing the subproblems alone leads to a linear parallel depth, since the first *IpSc* execution requires $\Omega(n)$ sequential work. Therefore, we also have to parallelize *IpSc* itself. We focus on the parallelization of *RoughScatter* which, in practice, accounts for the vast majority of work.

[7] Based on Theorem 8, we are also free to recurse with *IpScShuf* instead of using *Fisher-Yates Shuffle*.

4.1 Parallelizing RoughScatter

At heart, the parallel *ParRoughScatter* runs the sequential *RoughScatter* concurrently on independent subproblems. To this end, we exploit that *RoughScatter* allows arbitrary gaps *between* buckets. Secondly, we can freely pause and resume after each assignment without additional overhead since the algorithm's state is fully captured by the buckets' pointer triples and the partition of the placed elements.

Analogously to Section 3.3, we first split the input into k buckets of roughly equal size. In order to fork, we further split each bucket into two, and assign either half to one subtask. Then each subtask either recursively continues splitting, or, if the subproblem is sufficiently small, runs the sequential *RoughScatter*. After both subtasks join, we merge the two halves of each bucket. This involves only operations on the buckets' pointers and swapping the staged items of the first half to the second half. Additionally observe that the first subtask ends if there exists a filled bucket $B_i^{(1)}$, and analogously $B_j^{(2)}$ for the second subtask. Only with probability $1/k$, we have $i = j$, and thus, the merged bucket B_j is full. Otherwise, all merged buckets contain at least one staged item and we continue executing *RoughScatter*.

▶ **Observation 9.** *Since ParRoughScatter applies RoughScatter after each join, the number R of remaining staged items according to Lemma 2 also holds for ParRoughScatter.*

▶ **Lemma 10.** *For $n \gg k \log^3 k$ and $k^2 = \mathcal{O}(n)$, whp ParRoughScatter has $\mathcal{O}\big(\sqrt{nk \log k}\big)$ parallel depth and needs $\mathcal{O}(n)$ work.*

Proof. Splitting k buckets into $2k$ takes $\mathcal{O}(k)$ time. By Observation 9, Lemma 2 bounds the number of staged items received from both subtasks to $\mathcal{O}\big(\sqrt{nk \log k}\big)$. This bounds from above the time required to swap elements during merging, as well as the time to run *RoughScatter* on the remaining staged elements after merging. To meet the prerequisites of Lemma 2, we choose a base case size of $N_0 = k^2$ and process smaller subproblem sequentially in time $\mathcal{O}(N_0)$. This leads to the following bound on the parallel depth $D(n)$:

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

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

Analogously, we bound the work using the Master Theorem [5] for the following recursion:

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

4.2 Parallelizing FineScatter

As we discuss in Section 6.4, in practice, it is not necessary to parallelize *FineScatter* due to its negligible impact on the total runtime. Thus, in the following, we sketch just enough adoptions to reduce the parallel depth of *FineScatter* to that of *ParRoughScatter*. By Observation 9, the analysis of *FineScatter* in Section 3.4 remains valid after the execution of *ParRoughScatter*. By comparing with Lemma 10, we find that the parallel depth *ParRoughScatter* dominates all sequential operations but *TwoSweep*.

Recall that *TwoSweep* shifts the boundaries of buckets to match the final bucket sizes in time $\mathcal{O}\big(k\sqrt{nk \log k}\big)$ whp. Thus, we need to shave off only a factor of $\Theta(k)$ which is straightforward using standard parallelization techniques based on the following observation:

The prefix sum D_i defined in Lemma 5 can be interpreted as the number of elements that the end of bucket B_i needs to be shifted. Thus, after computing $(D_i)_i$ and placing one worker per bucket, each worker can shift elements in the appropriate direction. To shift items between distant buckets, we run $\mathcal{O}(k)$ rounds. The time per round is dominated by the largest swap of items over any bucket boundary which, in turn, is upper bounded by the maximal deviation $\mathcal{O}\left(\sqrt{n/k \log k}\right)$ whp in the first round. Thus, *TwoSweep* can be naïvely parallelized with a parallel depth of $\mathcal{O}\left(\sqrt{nk \log k}\right)$ matching that of *ParRoughScatter*. This results in a trivial upper bound of work of $\mathcal{O}\left(k\sqrt{nk \log k}\right)$ matching the overestimation of Corollary 6 used for the sequential *TwoSweep*.

▶ **Theorem 11.** *PIpScShuf has (whp) parallel depth* $\mathcal{O}\left(\sqrt{nk/\log k}\log(n)\right)$*, uses* $\mathcal{O}(n \log_k(n))$ *work and* $\mathcal{O}(k[\log_k(n) + \mathfrak{P}])$ *words of memory where* $\mathfrak{P}$ *is the number of parallel subtasks.*

Proof. The proof is analogous to the proof of Theorem 8 by replacing the complexity measures of *IpSc* with Lemma 10 followed by symbolic simplifications. The memory bound additionally accounts for k bucket pointer triples per subtask. ◀

5 Implementation

Our implementations use `Rust`, a programming language with strong memory safety and parallelism guarantees. While the code repository contains a number of prototypes, we consider the publicly exposed algorithms, such as *IpScShuf* (`seq_shuffle`) and *PIpScShuf* (`par_shuffle`), ready to be used in other projects. To monitor the code quality, we rely on the strong static analysis tools and dynamic checks available in the Rust ecosystem. We also use more than 80 tests that include statistical tests of the uniformity of the produced permutations (e.g., the 1 and 2-independence of the output ranks).

PIpScShuf uses the work-stealing scheduler included in the `rayon`[8] crate. We exclusively use binary Fork-Join parallelism by means of the `rayon::join` function which requires no heap allocations after the worker pool was once initialized. Given the widespread usage of `rayon`, it is very likely that the calling application already set up this pool. Then, none of our algorithms cause any heap memory allocation (thereby avoiding potential error sources or hidden synchronizations). A `rayon::join` incurs very little cost if both tasks are executed on the same worker. Hence, we regularly define more than 2^{11} parallel subtasks – allowing fine-grained work-balancing. In case of a compatible pseudo-random number generator (requires `rand::SeedableRNG` trait), we use a deterministic sequence to derive the subtasks' generators from the provided generator. Then, two runs from the same state yield the same permutation despite non-deterministic scheduling; this optional reproducibility can be crucial (e.g., for debugging of the embedding code).

Based on Figure 11 (Appendix), we empirically optimized the number of buckets k as $k = 64$. The vast majority of code is implemented in the *safe* subset of Rust. In Section 6, we use a highly optimized implementation of *RoughScatter* that requires pointer arithmetic and memory accesses without explicit boundary checks which is considered *unsafe* in Rust. While the memory safety guarantees of these sections are "only" comparable with C/C++, as an implementor it is easier to reason about these small code segments (as opposed to the whole program) and to check assumptions during runtime.

[8] `https://github.com/rayon-rs/rayon`

On the `x86` platform, the code executes $2\lfloor 64/\lceil \log_2(k) \rceil \rfloor$ (i.e. 32 for $k = 16$ and 20 for $k = 64$) random assignments without any branching instructions. This allows a high utilization of the CPU's pipeline which is further increased by explicitly prefetching the memory locations. Further, instead of using a standard `swap`(x, y) with three move operations (namely $t \leftarrow x,\ x \leftarrow y,\ y \leftarrow t$ where t is a temporary storage), we use two temporary variables resulting in $2 + \epsilon$ data movements per assignment. The resulting assignment process is at least five times faster than any weighted sampling strategy we experimented with (including fast implementations of [21] and various rejection schemes in spirit of [6]).

IpScShuf and *PIpScShuf* use a *FY* implementation for instances below 2^{18} items that resorts to 32 bit arithmetic and often produces two indices from one random 64 bit word.

The repository includes highly optimized sequential and parallel reimplementations of *MergeShuffle* which include similar techniques as above. Their performance is incomparable with the original implementation[9] which, on the one hand, includes handcrafted assembly code for merging, but, on the other hand, uses the `rdrand` instruction [17] to acquire random bits; depending on the specific processor, `rdrand` one to two orders of magnitude slower than `Pcg64Mcg`. [12, 28] Further, both choices are highly non-portable. Overall our portable implementation using `Pcg64Mcg` is faster than the original. Observe that *MergeShuffle* has linear parallel depth since only independent subproblems are executed in parallel while the merging of the first recursion layer is purely sequential.

6 Empirical evaluation

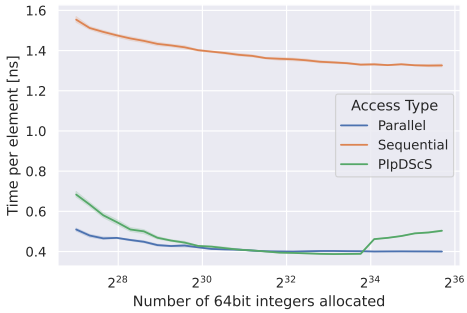
In this section, we investigate the performance of multiple shuffling algorithms for diverse settings. If not stated differently, we use the following standard parameters:

- Measurements are collected on a machine with an *AMD EPYC 7702P* CPU (64 cores/128 hardware threads), 512 GB RAM, running *Ubuntu 20.04*, `rustc 1.71` and `gcc-10`, using release builds (`cargo -release`, `g++ -O3`) without machine-specific optimizations. Further, we consider nine more machines with quite different configurations in Section 6.6.
- Experiments focus on the fast pseudo-random number generator `Pcg64Mcg`, as this choice exposes overheads in the shuffling algorithms rather in the generators itself. In Figure 8 (Appendix), we report the performance for different generators. The relative performance of algorithms remains qualitatively similar for different randomness sources, though *IpScShuf* and *PIpScShuf* are less affected by the generator choice as *FY* variants.
- Experiments focus on 64 bit integers, which seems to be a typical index size in data sets of several 100 GiB. As indicated in Figure 9 (Appendix), the throughput (measured in bytes per second) increases for larger elements since the per-element overhead shrinks. Again, *IpScShuf* and *PIpScShuf* exhibit a smaller spread than *FY* variants.
- All performance measurements reported are the mean of at least five runs. To reduce systematic errors, an individual run is the average of N repetitions where N is chosen such that the measured time exceeds 100 ms. Consecutive repetitions use different locations in a larger memory region to simulate a *cold start* where the input is not already cached.

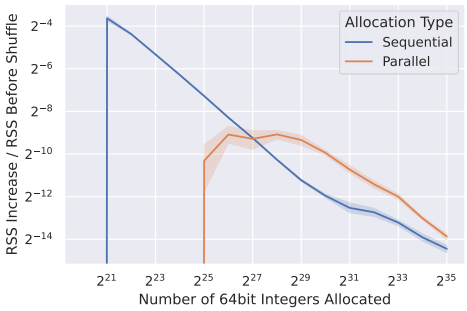
6.1 Memory usage and allocation costs

One important motivation for this work is the runtime cost of allocating large amounts of memory which we quantify in Figure 3a. For each measurement, we obtain a certain amount of data using the low level `libc::malloc` instruction, initialize it, and then return the data

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



(a) Runtime cost of memory allocation.



(b) Memory usage of *PIpScShuf* implementation.

Figure 3 Measurements of memory runtime costs and memory usage as described in Section 6.1.

using `libc::free`. For large volumes, `malloc` requests the operating system to map a certain memory size into virtual memory. Critically, the memory will only be backed by physical memory if it is actually accessed. For this reason, we initialize the data twice, and subtract the second round from the first one. The difference between both runs is the time it takes the system to provide the physical memory (without initializing it). Our measurements also suggest that writing the values in parallel does not scale well – despite an investment of 128 threads, we observe only a speedup of 4.9 for the initialization which is reduced to 3.1 for the whole process since `malloc` and `free` are sequential. For reference, we included the runtime of *PIpScShuf* and find that shuffling the data in parallel takes roughly as long as to acquire the memory needed to store a copy – without copying it.

In Figure 3b, we report the effective memory usage of *PIpScShuf*. We start a dedicated process for each run and measure the *maximal resident set size* (RSS) of the process (i.e. the maximal amount of memory that was physically backed at any time during the execution). We measure the RSS before and after the invocation of *PIpScShuf* and report the relative growth. As already discussed in Section 5, `rayon`'s worker pool needs to be initialized once. If we allocated the data sequentially, *PIpScShuf* implicitly sets up a pool resulting in 1631 heap operations to reserve a total of 923 KiB. After a parallel allocation, on the other hand, no heap operations are carried out. Even then, we observe a small increase of the RSS for large inputs – this seems to be caused by growing stack memory of the 128 active threads. In this case, the largest observed growth is 0.2 % and diminishes for very large inputs.

6.2 Performance overview

In Figure 4, we report the performance of several shuffling implementations. For each run, we set a timeout of 30 s and stop a graph after its algorithm hit said budget. The only exception is our *PIpScShuf* implementation with a runtime of 20.8 s for the largest data point. From a pool of various *Fisher-Yates Shuffle* implementations (see Figure 10, Appendix), we only include our own variant which strictly outperforms all competitors in the relevant regime.

The two fastest algorithms are `parallel::shuffle` (a C++ implementation of *ScatterShuffle* included in `stdlibc++`) and our *PIpScShuf*. For relatively small inputs *ScatterShuffle* is faster than *PIpScShuf* which takes the lead for inputs larger than 256 MiB.

All algorithms exhibit deteriorating throughput for larger inputs. This is remarkable for *FY* derivatives which have a predicted linear runtime. Their slowdown can be attributed to cache misses and related effects of the memory hierarchy. In Figure 10 (Appendix), we

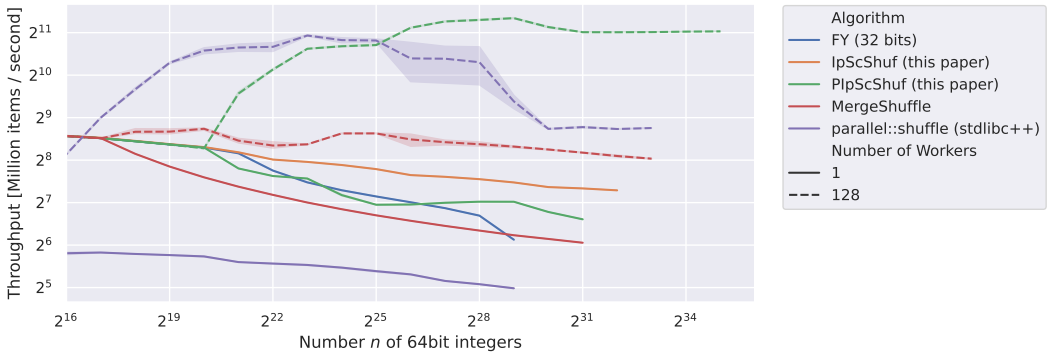


Figure 4 Performance of several shuffling algorithms with a time budget of 30 s per run. *FY*, *IpScShuf*, and *PIpScShuf* are our own implementations. `std::shuffle` and `parallel:shuffle` are implemented in C++. For parallel algorithms, we indicate the number of cores available as p.

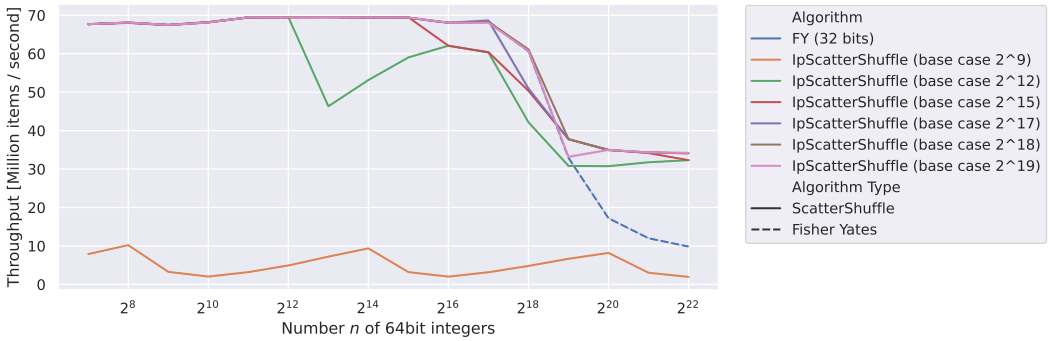


Figure 5 Performance of several sequential shuffling algorithms run in parallel on different data.

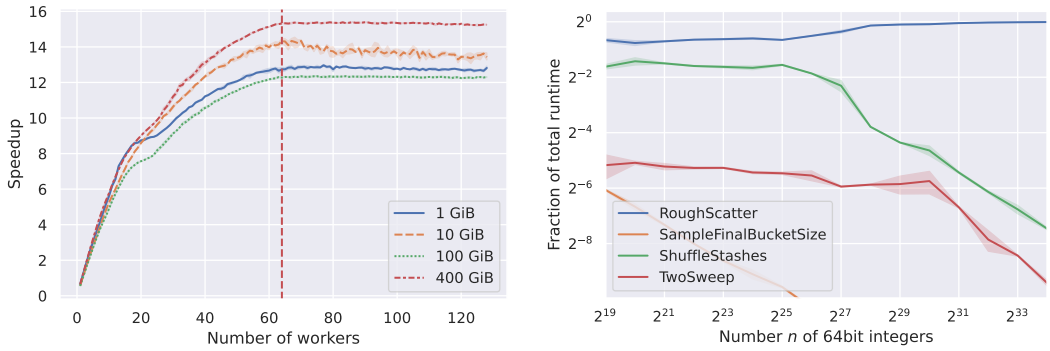
demonstrate that memory latency can be hidden by explicitly prefetching memory locations ahead of time.[10] However, prefetching only helps to simulate a slightly larger cache and we still observe a significant performance drop around 2 GiB. As predicted, all implementations based on *ScatterShuffle* exhibit a $\log_k(n)$ dependency in their runtime; this is especially visible for *PIpScShuf* when executed on a single core.

Due to incompatible libraries, we were unable to include fair performance measurements of [15] in our campaign. However, Figure 7 reports a higher throughput for *PIpScShuf* on a quad-core laptop (i7-8550U) than [15, Table 6] for a quad-socket server (E7-8867 v4) with 72 cores (while *PIpScShuf* uses less memory and includes the computation of random bits).

6.3 Parallel execution of sequential algorithms

When selecting an appropriate base case algorithm for *PIpScShuf*, Figures 4 and 10 can be misleading as they report the performance of sequential algorithms executed in isolation. In this setting, the studied algorithm has more resources at its disposal compared to the case where several instances are executed in parallel on independent data. This might also be relevant in different scenarios, e.g., if computational resources are shared by different users.

[10] We use a ring buffer to generate and prefetch random indices 16 swaps prior to the actual swap.



(a) Strong scaling of *PIpScShuf*. The vertical line indicates the number of physical cores.

(b) Fraction of runtime of the first recursion layer of *PIpScShuf* with 128 cores.

Figure 6 Parallel performance of *Parallel In-Place ScatterShuffle*.

Figure 5 is recorded similarly to Figure 4 with the difference that we execute 128 independent tasks in parallel and report the mean of their individual runtime as a single run. To avoid scheduling artifacts, we discard and repeat any run in which the wall-time of the experiment (i.e. from the start of the first thread to the termination of the last) is 20 % larger than the mean runtime of the individual tasks.

In this setting, we observe that memory becomes the dominating bottleneck; with minor exceptions (such as too small base case sizes), all algorithms exhibit roughly the same performance for instances below 1 MiB (the CPU has 256 MiB L3 cache that is now shared among 128 threads). For larger instances, *IpScShuf* variants are more than 3 times faster than the *FY* variants. Also observe that the contributions of implementation details, such as prefetching or base case size, pale in comparison the importance of memory locality.

6.4 Parallel scaling

To quantify the parallel speedup of *PIpScShuf*, we carry out a strong scaling experiment as follows. For fixed input sizes, we execute *IpScShuf* and the fastest *FY* implementation as base lines and then profile *PIpScShuf* for an increasing number of workers. In Figure 6a, we report the parallel speedup over the fast sequential implementation (*IpScShuf* in all cases).

For data set sizes of 10 GiB and larger, the speedup over the fastest sequential solution approaches up to 16. The self speedup is larger (e.g., 22.9 for 400 GiB), indicating a good scalability that is somewhat offset by the additional overhead of the parallel implementation. For the same instance, *PIpScShuf* is 141 times faster than *Fisher-Yates Shuffle*.

We see a substantial increase in speed until the number of workers matches the number of physical cores; using virtual cores (simultaneous multi-threading) has little impact. This is to be expected since shuffling is memory-bound and virtual cores primarily help to saturate arithmetic units of super-scalar processors, but do not affect memory performance.

6.5 Relative performance of subproblems

When designing, implementing, and analyzing *IpScShuf* and *PIpScShuf* we focused on the fast opportunistic *RoughScatter* which then requires the additional *FineScatter* post-processing to deal with the few remaining items. While we heavily optimized *RoughScatter*, we opted for simple and easy to implement solutions in *FineScatter*.

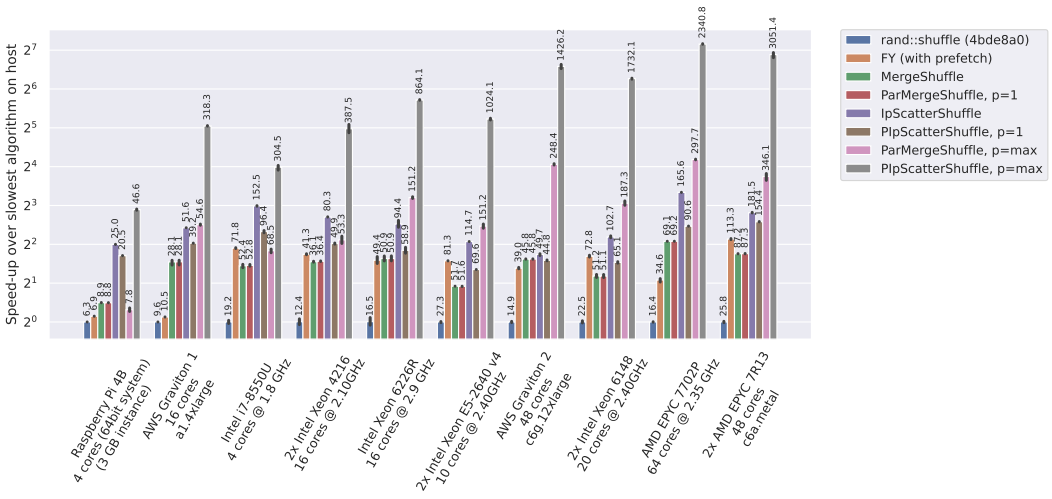


Figure 7 Performance of selected algorithms on different computers shuffling 64 bit integers. The length of a bar corresponds to the speedup compared to the slowest algorithm on that system. The numbers above a bar indicate the absolute through-put in million elements per second.

To empirically support this design decision, we measure the runtime of the first recursion layer of *PIpScShuf* for a wide range of input sizes. In Figure 6b, we then report the relative wall time of the four sub-algorithms that constitute said layer. In our implementation only *RoughScatter* is executed in parallel with 128 workers available while the remaining parts are sequential algorithms. Despite the asymmetry in available workers, we observe that for $n \geq 2^{27}$ *ParRoughScatter* accounts for more than 90 % (99 % for $n \geq 2^{33}$) of the runtime. This supports our design decisions since optimizing *FineScatter* leads to diminishing results.

6.6 Performance on different machines

To verify that our empirical findings are representative for modern computers, we quantify the performance of shuffling on different machines in Figure 7. The machines range from a single-board computer, over a laptop, to dual-socket servers (covering more than two orders in magnitude in purchase price). They use different instruction sets, micro-architectures, processor manufactures, and core counts. Their configurations are specified in the figure. We reiterate that no machine-specific optimizations are used; in fact, exactly two binaries were used (for ARM and x86, respectively).

To accommodate most systems, we selected an instance size of 10 GiB with the exception of the Raspberry PI 4B which features only 4 GiB of main memory. The measurements consist of runs of sequential algorithms, sequential runs of parallel algorithms (indicated by $p = 1$), and parallel runs with one worker per hardware thread (indicated by $p = \max$). The maximal throughput of the fastest system is 50 times higher than that of the slowest system. In all cases `rand::shuffle` is the slowest contender, *IpScShuf* the fastest sequential implementation, and *PIpScShuf* the overall fastest solution.

References

1 Daniel Allendorf, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Nick Wormald. Engineering uniform sampling of graphs with a prescribed power-law degree sequence. In *ALENEX*, pages 27–40. SIAM, 2022.

2 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.*, 9(1):2:1–2:62, 2022.

3 Axel Bacher, Olivier Bodini, Alexandros Hollender, and Jérémie O. Lumbroso. Mergeshuffle: a very fast, parallel random permutation algorithm. In *GASCom*, volume 2113 of *CEUR Workshop Proceedings*, pages 43–52. CEUR-WS.org, 2018.

4 Edward A. Bender and E. Rodney Canfield. The asymptotic number of labeled graphs with given degree sequences. *J. Comb. Theory, Ser. A*, 24(3):296–307, 1978.

5 Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

6 Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Simulating population protocols in sub-constant time per interaction. In *ESA*, volume 173 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

7 Guojing Cong and David A. Bader. An empirical analysis of parallel random permutation algorithms ON smps. In *PDCS*, pages 27–34. ISCA, 2005.

8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

9 Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986.

10 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

11 Thomas S. Ferguson. Who solved the secretary problem? *Stat. Sci.*, 4(3):282–289, 1989.

12 Agner Fog. Instruction tables. URL: `https://www.agner.org/optimize/instruction_tables.pdf`.

13 Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019.

14 Hermann Gruber, Markus Holzer, and Oliver Ruepp. Sorting the slow way: An analysis of perversely awful randomized sorting algorithms. In *FUN*, volume 4475 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2007.

15 Yan Gu, Omar Obeya, and Julian Shun. Parallel in-place algorithms: Theory and practice. In *APOCS*, pages 114–128. SIAM, 2021.

16 Chris Hinrichs, Vamsi K Ithapu, Qinyuan Sun, Sterling C Johnson, and Vikas Singh. Speeding up permutation testing in neuroimaging. In C. J. C. Burges et al., editor, *Advances in Neural Information Processing Systems*, volume 26, pages 890–898. Curran Associates, Inc., 2013.

17 Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, 2022.

18 Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

19 Charles E. Leiserson. Programming irregular parallel applications in cilk. In *IRREGULAR*, volume 1253 of *Lecture Notes in Computer Science*, pages 61–71. Springer, 1997.

20 Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1):3:1–3:12, 2019.

21 Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. *Theory Comput. Syst.*, 36(4):329–358, 2003.

22 Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *Comput. Syst.*, 6(1):5–27, 1993.

23 SJ Meyer. A failure of structured programming, Zilog Corp. Technical report, Software Dept. Technical Report, 1979.

24 Richard Nixon. Executive order 11497 — amending the selective service regulations to prescribe random selection, 1969.

25 Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

26 Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation. *CoRR*, abs/2003.00736, 2020.

27 Martin Raab and Angelika Steger. "Balls into bins" – A simple and tight analysis. In *RANDOM*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 1998.

28 Matthew Route. Radio-flaring ultracool dwarf population synthesis. *The Astrophysical Journal*, 845(1):66, August 2017. doi:10.3847/1538-4357/aa7ede.

29 Peter Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 67(6):305–309, 1998.

30 Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *SODA*, pages 431–448. SIAM, 2015.

31 Johannes Singler and Benjamin Konsik. The GNU libstdc++ parallel mode: software engineering considerations. In *IWMSE@ICSE*, pages 15–22. ACM, 2008.

A Additional measurements

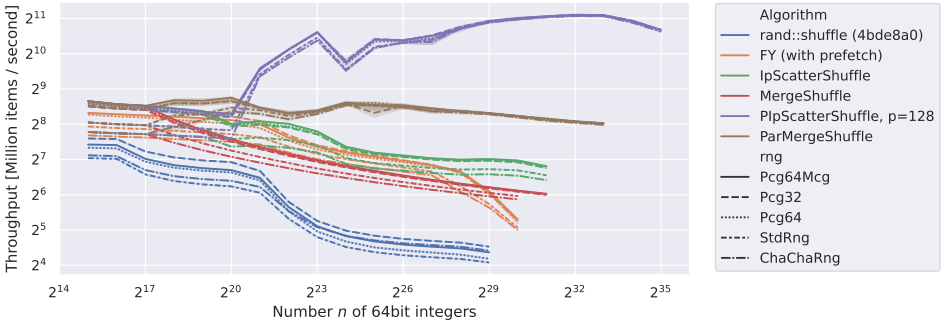


Figure 8 Performance of selected algorithms with different pseudo-random number generators.

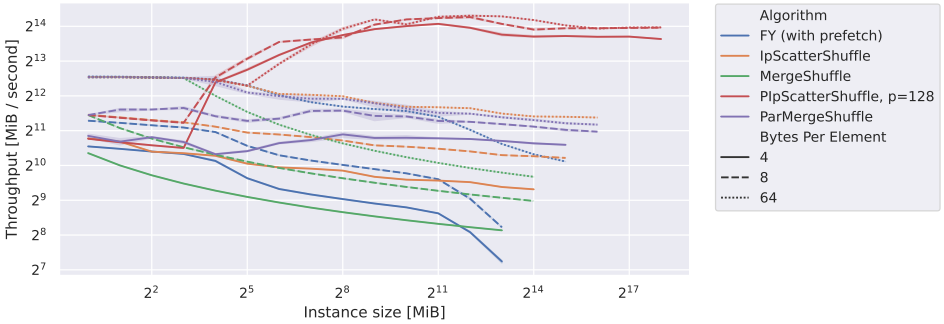


Figure 9 Performance of selected algorithms with different element sizes.

B Quantifying the hidden constants

In Lemma 2, we bound the number $R \leq \sqrt{2nk \log k}$ of items that remain staged after *RoughScatter* whp. Additionally, in Corollary 6, we bounded the complexity of *TwoSweep* by $\mathcal{O}\big(k\sqrt{nk \log k}\big)$. To provide empirical evidence and study the hidden constants, we simulate both processes. In Figure 12, we report the mean over 1000 independent runs divided by $\sqrt{2nk \log k}$ and $k\sqrt{nk \log k}$ respectively. Recall that our implementations use $k = 16$ and $k = 64$; we additionally simulated $k = 2^{16}$ as an accommodating upper bound for the foreseeable future. The small growth in k visible in Figure 12 is due to the small k values. Simulations with k up to 2^{28} agree with Lemma 2 and approach a factor of 0.66 in Corollary 6.

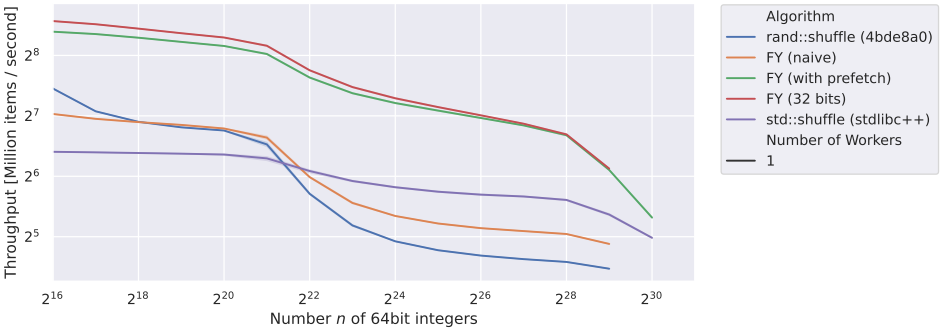


Figure 10 Performance of several Fisher-Yates implementations with a time budget of 30 s.

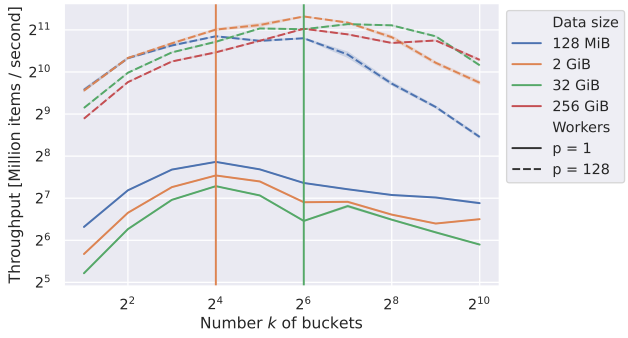


Figure 11 Performance of *In-Place ScatterShuffle* ($p = 1$) and *Parallel In-Place ScatterShuffle* ($p = 128$) for various data sizes as function of the number of buckets k. The two vertical lines correspond to default values of $k = 16$ for *IpScShuf* and $k = 64$ for *PIpScShuf*, respectively.

C Omitted proofs

▶ **Lemma 5.** *Let $N^f = (n_i^f)_i$ be the final bucket sizes, denote their deviation from the mean n/k as $d_i = n_i^f - n/k$, and let $D_i = \sum_{j=1}^{i} d_j$ be the inclusive prefix sum of deviations. Then, TwoSweep executes a total of $M(N^f) = \sum_i |D_i|$ swaps and takes time $\mathcal{O}\big(k + M(N^f)\big)$.*

Proof. Observe that a positive value d_i indicates that bucket B_i needs to receive d_i additional staged items from other buckets. Contrary, a negative value d_i means that bucket B_i has to give away $-d_i$ elements. The prefix sum D_i has an analogous meaning but accumulated over the first i buckets. This leads to the following cases:

1. If D_i is positive, buckets $B_1, \ldots, B_i$ have an excess of D_i items required somewhere in $B_{i+1}, \ldots, B_k$. These D_i items will be pushed to the right during the first sweep.
2. If D_i is negative, buckets $B_1, \ldots, B_i$ have a demand of $|D_i|$ items met by an excess somewhere in the buckets $B_{i+1}, \ldots, B_k$. Thus, B_i receives $|D_i|$ items in the second sweep.

In sum, bucket B_i is involved in $|D_i|$ swaps with its direct neighbors, leading to a total of $M(N^f)$ swaps and $\mathcal{O}\big(M(N^f) + k\big)$ work where the k accounts for per-bucket overheads. ◀

▶ **Corollary 6.** *TwoSweep takes time $\mathcal{O}\big(k\sqrt{nk \log k}\big)$ whp.*

Proof. We prove the claim based on Lemma 5 by establishing $\sum_i |D_i| = \mathcal{O}\big(k\sqrt{nk \log k}\big)$ (whp) where $D_i = \sum_{j=1}^{i} d_i$ is the prefix sum over the bucket size deviations from the mean. Observe that by construction, only elements that remain staged after the execution of

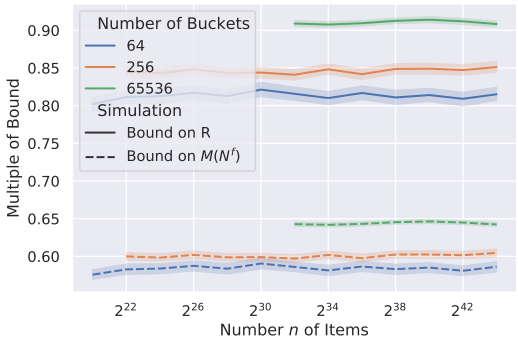


Figure 12 Simulation of Lemma 2 and Corollary 6.

RoughScatter can contribute and therefore $\sum_i |d_i| \leq 2R$ where $R \leq \sqrt{2nk \log k}$ (whp) by Lemma 2. Additionally, since the deviations balance over all buckets we have $\sum_i d_i = 0$. Thus, we trivially have that $\max_i |D_i| \leq R$.

We assume a worst case deviation where the first bucket needs to gain all R elements from the last bucket (or vice versa). While this is rather pessimistic (recall $\max_i \{|d_i|\} = \mathcal{O}\left(\sqrt{n/k \log k}\right)$ whp), it suffices to show the bound. In this instance, we have $|D_i| \leq R$ for all i and $\sum_{i=1}^{k} |D_i| \leq \sum_{i=1}^{k} R \leq kR = \mathcal{O}\left(k\sqrt{nk \log k}\right)$ (whp). ◀