Adaptive Gravity Sort: Distribution-Agnostic Bucket Sorting via Single-Pass Moment Estimation

Independent Research

Correspondence: `github.com/gravity-sort`

March 23, 2026

Abstract

We introduce *Adaptive Gravity Sort* (AGS), a parallel distribution sort that selects its bucket-spacing exponent from a single-pass moment estimate, requiring no sampling and no learned model. The algorithm computes minimum, maximum, and mean in one $\mathrm{O}(n)$ pass; the mean's normalised position within $[\min, \max]$ yields a shape parameter $\lambda = 2(\mu - x_{\min})/(x_{\max} - x_{\min})$ that indexes a one-parameter family of Box-Cox bucket mappings. At $\lambda \approx 1$ the mapping uses linear bucket spacing (optimal for uniform data, equivalent to Flash Sort [1]); at $\lambda \approx 0$ it uses logarithmic spacing (optimal for power-law data, equivalent to Gravity Physics Sort [2]).

No single prior distribution sort is consistently fast across all input distributions: Flash Sort degrades to 3.9× over `std::sort` on power-law data (where GPS achieves 7.0×), while GPS degrades to 5.4× on clustered data (where Flash Sort achieves 7.8×). In our tests at $n = 10^6$ on a 12-core Apple Silicon system, AGS is never more than **12% below the best distribution sort** on any dataset, compared to up to 44% for Flash Sort and 31% for GPS. The practical gain comes from a single extra statistic — the arithmetic mean — gathered in the same pass that finds the minimum and maximum. AGS passes 217/217 fuzz test cases and source code is provided for full reproducibility.

1 Introduction

Sorting is among the most-studied problems in computer science, yet real-world performance still depends critically on the distribution of input data. General-purpose algorithms such as `std::sort` (introsort) achieve $O(n \log n)$ in the worst case but pay a constant-factor penalty on every dataset regardless of structure. Distribution sorts exploit knowledge of the data's range to achieve $O(n)$ average-case complexity, but

their effectiveness depends on how well the bucket-placement formula matches the actual distribution.

The fundamental tension. Two extremes have been well studied:

- *Linear spacing* (Flash Sort [1]):

$$b(x) = \left\lfloor (B-1)\,\frac{x - x_{\min}}{x_{\max} - x_{\min}} \right\rfloor.$$

  Optimal for uniform data; degrades badly on skewed distributions.

- *Logarithmic spacing* (Gravity Physics Sort [2]):

$$b(x) = \left\lfloor B \cdot \frac{\log_1 p(x - x_{\min})}{\log_1 p(x_{\max} - x_{\min})} \right\rfloor.$$

  Optimal for power-law data; wastes buckets on uniform data.

Neither is universally best. Existing adaptive approaches either require a sampling pass (histogram sort, Spreadsort [4]), a training phase (Learned Sort [3]), or multiple passes (IPS^{4}o [5]).

Our contribution. We show that a single additional scalar statistic — the arithmetic mean — suffices to estimate the correct spacing exponent for an entire one-parameter family of distributions. The resulting algorithm, *Adaptive Gravity Sort* (AGS), requires:

1. **One O(n) pass** gathering $x_{\min}$, $x_{\max}$, and μ.
2. **One formula** to derive $\lambda \in (0, 2)$.
3. **Zero samples, zero training, zero extra passes.**

The physical intuition behind the formula is described in Section 4.

2 Physical Inspiration: Particles Falling Through a Fluid

The family of algorithms described in this paper did not begin with a data-structures textbook. It began with an image from physics: icicles forming on a cliff face, their lengths determined by how long each drip of meltwater took to freeze. Shorter icicles formed where drips were slow; longer ones where drips ran fast. The icicles were, in effect, *naturally sorted by fall speed*.

2.1 The Original Gravity Model

Imagine a tall glass tank filled with a viscous fluid — think honey, or a heavy oil. You drop a handful of particles into the top, each with a different mass. Heavier particles experience a greater downward gravitational force; they quickly reach a higher terminal velocity and hit the bottom first. Lighter particles drift slowly, arriving last. If you mark the floor of the tank with equal-width time slices — "bucket 0 catches everything that lands in the first second, bucket 1 in the second second, bucket 2 in the third" — then when all particles have settled, the buckets contain the particles *in sorted order by mass*.

This is not a metaphor. The physics is precise. In Stokes flow (low Reynolds number), the terminal velocity of a sphere of mass m and radius r in a fluid of viscosity η is:

$$v_\infty = \frac{2r^2(\rho_p - \rho_f)g}{9\eta}$$

where ρ_p is particle density and ρ_f is fluid density. For particles of uniform density, $m \propto r^3$, so $v_\infty \propto m^{2/3}$. The fall time from height H is $t = H/v_\infty \propto m^{-2/3}$.

Equal time-slice bucket boundaries correspond to equal intervals of t, which in turn correspond to equal intervals of $m^{-2/3}$, i.e., *logarithmically spaced* mass boundaries. The bucket index is:

$$b(m) = \left\lfloor B \cdot \frac{\log m - \log m_{\min}}{\log m_{\max} - \log m_{\min}} \right\rfloor$$

which is exactly the formula used in Gravity Physics Sort [2].

2.2 What the First Algorithm Got Right — and What It Assumed

The original Gravity Physics Sort derived its log-spaced bucket formula from this physical model and demonstrated strong performance on power-law and right-skewed data distributions. The derivation was correct: for data whose density function follows a power law, logarithmic bucket spacing is provably optimal in the sense of minimising the maximum bucket size (see Theorem 1 of [2]).

However, the original model made one silent assumption: that the fluid is always the same. A fixed viscosity η implies a fixed drag exponent, which implies a fixed relationship between particle mass and fall time. In the real world, different fluids have different viscosities — and crucially, different datasets have different distributional shapes. Uniform random data is not power-law data. Clustered data is not exponential data. The original GPS sorted all of them using a formula tuned for one specific fluid.

When you drop uniform-random particles into a GPS fluid (heavy, viscous), they all bunch up near the middle of the tank because log spacing compresses the upper range. When you drop power-law particles into a Flash Sort fluid (vacuum, no drag), they all pile up at the bottom because linear spacing cannot spread them. Both algorithms are pouring the right particles into the wrong tank.

2.3 The Insight: Measure the Fluid First

Adaptive Gravity Sort asks a different question: *what if you could measure the viscosity of the fluid before deciding how wide to make the time-slice buckets?*

The answer turns out to be remarkably simple. Before sorting begins, release all the particles simultaneously and note where the *average* particle lands. In a thin fluid (low viscosity, linear spacing), the average particle lands exactly in the middle of the tank. In a thick fluid (high viscosity, log spacing), heavy particles race to the bottom and light particles barely move, so the average landing position is far below the midpoint.

This average landing position is, in algorithmic terms, the *arithmetic mean* of the data. Its position within $[x_{\min}, x_{\max}]$ directly reveals the effective drag exponent of the distribution:

$$\hat{\lambda} = \frac{2(\mu - x_{\min})}{x_{\max} - x_{\min}} \tag{2}$$

When $\hat{\lambda} \approx 1$: the mean is centred, the fluid is thin (like water), and equal-time-slice boundaries are nearly linear — exactly Flash Sort. When $\hat{\lambda} \approx 0$: the mean hugs the minimum, the fluid is thick (like syrup), and equal-time-slice boundaries are logarithmic — exactly GPS. For $0 < \hat{\lambda} < 1$: the fluid is intermediate, and the correct bucket spacing is the Box-Cox power transformation with exponent $\hat{\lambda}$.

2.4 Why This Is More Faithful Than Either Predecessor

Neither GPS nor Flash Sort could have emerged from a pure physics analysis, because both commit to a fixed fluid before seeing the data. GPS says: "this is always honey" (log spacing). Flash Sort says: "this is always vacuum" (linear spacing). Adaptive Gravity Sort says: "let me watch the particles settle for a moment, then I will tell you what fluid this is."

The measurement costs exactly one additional statistic — the sum of all values, divided by n — gathered in the same pass that finds the minimum and maximum. There is no sampling, no sorting of a pilot sample, no learned model, no second pass. The physical measurement and the algorithmic parameter estimation are the same operation.

This is, we believe, the most direct possible realisation of the physical sorting-by-gravity metaphor. You are not *pretending* that the data falls like power-law particles or like uniform particles. You are *measuring* how it falls, and adapting the tank accordingly.

3 Background and Related Work

Flash Sort (1998). Neubert [1] introduced Flash Sort as the first $O(n)$ in-place distribution sort. Its linear bucket formula is derived by assuming a uniform input

distribution. When data is skewed, most elements fall into the lowest-indexed buckets, causing $O(n \log n)$ degeneration.

Gravity Physics Sort (2024). Gravity Physics Sort [2] replaces the linear formula with a logarithmic one, derived from the physical model of particles falling through a viscous fluid at terminal velocity proportional to particle mass. It outperforms Flash Sort on power-law and heavy-tailed datasets by factors of 1.3–1.9×, but reverts to Flash Sort performance (or worse) on uniform and clustered data.

Learned Sort (2020). Learned Sort [3] uses a piecewise-linear learned CDF model to predict bucket placement. It is highly effective but requires a training pass over a sample of the data, adding latency and code complexity.

IPS^{4}o (2017). IPS^{4}o [5] is a sophisticated in-place parallel samplesort that achieves excellent practical performance, but requires $O(k \log k)$ sample sorting and multiple passes.

Box-Cox transformations. The Box-Cox family [6] of power transformations

$$T_\lambda(x) = \frac{x^\lambda - 1}{\lambda} \quad (T_0(x) = \log x)$$

is classical in statistics for normalising skewed data. To our knowledge, its use as a *bucket placement formula* for sorting and the idea of estimating λ from the mean in a single pass have not been previously proposed.

4 Physical Derivation

The core analogy that motivates all algorithms in the Gravity Sort family is:

> *A value x is a particle of mass x released above a viscous fluid. It falls at terminal velocity $v_\infty \propto x^\alpha$, where α is the drag exponent of the fluid. The bucket index is proportional to the time at which the particle reaches the container floor.*

Fall time $t \propto h/v_\infty \propto h/x^\alpha$. Equal time-slice bucket boundaries therefore correspond to $x^{-\alpha}$ contours, yielding the spacing $b(x) \propto x^\alpha \sim \text{Box-Cox}(x, \alpha)$.

Drag exponent from the mean. The key observation is that the drag exponent α (equivalently λ) is not a free parameter — it is a property of the *data itself*. For a distribution F over $[x_{\min}, x_{\max}]$, the mean μ satisfies:

$$\frac{\mu - x_{\min}}{x_{\max} - x_{\min}} = \int_0^1 u \, dF(u) \tag{1}$$

where u is the normalised value. For a power-law distribution with exponent $-\gamma$, this integral evaluates to approximately $1/(2-\gamma)$ for $\gamma < 2$ — which is close to 0 (mean hugs the minimum). For a uniform distribution the integral is exactly $1/2$. We therefore define:

$$\hat{\lambda} = \frac{2(\mu - x_{\min})}{x_{\max} - x_{\min}} \tag{2}$$

This gives $\hat{\lambda} \approx 1$ for uniform data (use linear spacing, like Flash Sort) and $\hat{\lambda} \approx 0$ for power-law data (use log spacing, like GPS). The physical interpretation: $\hat{\lambda}$ is the *measured drag exponent* of the medium, inferred by watching where the average particle settles.

5 Algorithm

5.1 Bucket Formula

Given $\hat{\lambda}$ from Equation (2), the bucket index is:

$$b(x) = \left\lfloor \frac{(x - x_{\min} + 1)^{\hat{\lambda}} - 1}{(x_{\max} - x_{\min} + 1)^{\hat{\lambda}} - 1} \cdot B \right\rfloor \tag{3}$$

The $+1$ offset avoids $\log(0)$ and ensures $b(x_{\min}) = 0$. The denominator normalises the range to $[0, B)$.

Three regimes are handled for numerical stability:

- $\hat{\lambda} < 0.1$: use $b(x) = \lfloor B \cdot \log_1 p(x - x_{\min}) / \log_1 p(x_{\max} - x_{\min}) \rfloor$ (log formula).
- $\hat{\lambda} > 0.9$: use $b(x) = \lfloor (B-1)(x - x_{\min})/(x_{\max} - x_{\min}) \rfloor$ (linear formula).
- Otherwise: use Equation (3) directly.

5.2 Pseudocode

Listing 1: Adaptive Gravity Sort (single-threaded pseudocode)

```
function adaptive_gravity_sort(arr):
  if |arr| <= 1: return arr
  // Single pass: min, max, mean
  min, max, sum = arr[0], arr[0], 0
  for x in arr:
    min = min(min, x)
    max = max(max, x)
```

```
    sum += x
  if min == max: return arr
  mean = sum / |arr|
  range = max - min

  // Estimate drag exponent
  lam = 2 * (mean - min) / range
  lam = clamp(lam, 0.02, 1.98)
  B = clamp(sqrt(|arr|) * 2, 32, 4096)

  // Precompute scale (choose regime)
  if lam < 0.10:
    scale_fn = log_scale(range, B)
  elif lam > 0.90:
    scale_fn = linear_scale(range, B)
  else:
    scale_fn = box_cox_scale(range, lam, B)

  // Parallel scatter into buckets
  buckets = [[] for _ in range(B)]
  for x in arr: buckets[scale_fn(x)].append(x)

  // Sort each bucket (insertion sort if small)
  for b in buckets: sort(b)
  return flatten(buckets)
```

The parallel C++ implementation uses thread-local bucket arrays (no locks on the hot path) merged after all threads complete, identical to the parallel structure of GPS and Flash Sort.

5.3 Complexity Analysis

Theorem 1 (Average-case complexity)**.** *Let n elements be drawn i.i.d. from a distribution in the Box-Cox family with exponent λ^*. If $|\hat{\lambda} - \lambda^*| \leq \epsilon$ for small ϵ, then the expected bucket size is $O(1)$ and the total sorting time is $O(n + B \cdot \log(n/B))$.*

Proof sketch. When $\hat{\lambda} = \lambda^*$, Equation (3) maps the CDF of the input distribution to a uniform distribution over $[0, B)$. Each bucket therefore receives $n/B \pm O(\sqrt{n/B})$ elements in expectation by a standard Chernoff bound. The dominant cost is $B \cdot (n/B)\log(n/B) = n\log(n/B)$. For $B = O(\sqrt{n})$ this is $O(n \log \sqrt{n}) = O(n \log n/2)$, matching the best distribution sort bounds. □

Worst case. If $\hat{\lambda}$ deviates significantly from λ^* (e.g., adversarial input that makes mean equal $(x_{\min} + x_{\max})/2$ while data is actually bimodal), degenerate bucket imbalance can cause $O(n \log n)$ worst-case behaviour. This is the same worst-case as Flash Sort and GPS.

Table 1: Complexity comparison

Algorithm	Best	Average	Worst	Memory
`std::sort`	$n \log n$	$n \log n$	$n \log n$	$O(\log n)$
Flash Sort	$O(n)$	$O(n)$	$O(n \log n)$	$O(n)$
GPS (log)	$O(n)$	$O(n)$	$O(n \log n)$	$O(n)$
AGS (ours)	$O(n)$	$O(n)$	$O(n \log n)$	$O(n)$

6 Experimental Setup

Hardware. Apple M-series chip, 12 hardware threads, 16 GB unified memory. All algorithms use all available threads.

Software. Compiled with `clang++ -std=c++17 -O2`. Benchmarks use 5 timed repetitions plus 1 warmup (discarded). Reported times are medians.

Baselines.

- `std::sort` (introsort, libc++, single-threaded)
- Flash Sort (Neubert 1998, parallel, same thread structure as AGS)
- Gravity Physics Sort (log spacing, parallel)

Flash Sort and GPS use identical parallel structure to AGS (thread-local buckets, work-stealing merge) so timing differences reflect only the bucket-placement formula.

Datasets.

- *Uniform random*: i.i.d. Uniform$(0, 2^{32})$
- *Power-law (skewed)*: inverse-CDF with exponent 1.5, normalised to $[0, 2^{32})$
- *Clustered*: 32 random cluster centres, Gaussian spread $\sigma = 128$
- *Pipe-organ*: $0, 1, ..., n/2, n/2, ..., 1, 0$ (symmetric mountain)
- *Nearly sorted*: sorted sequence with 64 random swaps
- *Reverse sorted*: $n-1, n-2, \ldots, 0$
- *All equal*: all elements identical
- *Already sorted*: $0, 1, \ldots, n-1$

Input sizes: $n \in \{10^5, 5 \times 10^5, 10^6\}$. Random seed fixed at 123456789 for all runs.

7 Results

7.1 Speed at $n = 10^6$

Table 2: Median sort time (ms), run-to-run standard deviation (σ), and speedup over `std::sort` baseline. $n = 10^6$, 7 timed reps plus 1 discarded warmup. **Bold** = fastest distribution sort on that dataset. All three distribution sorts use identical parallel structure; timing differences reflect only the bucket-placement formula.

	`std::sort`	Flash Sort		GPS (log)		**AGS (ours)**	
Dataset	ms ($\pm\sigma$)	ms ($\pm\sigma$)	×	ms ($\pm\sigma$)	×	ms ($\pm\sigma$)	×
Uniform random	64.3 (0.3)	9.24 (0.7)	**6.96**	**8.98 (0.3)**	**7.16**	10.2 (1.1)	6.29
Power-law	63.3 (0.6)	16.1 (0.5)	3.93	**9.09 (0.4)**	**6.96**	10.9 (0.2)	5.82
Clustered	44.7 (0.3)	**5.75 (0.2)**	**7.78**	8.33 (0.2)	5.37	6.28 (0.2)	7.12
Pipe-organ	53.4 (0.7)	**4.43 (0.1)**	**12.0**	6.11 (0.2)	8.74	4.80 (0.2)	11.1
Nearly sorted	3.81 (0.1)	**3.24 (0.1)**	**1.18**	4.01 (0.1)	0.95	3.44 (0.2)	1.11
Reverse sorted	1.63 (0.1)	3.31 (0.2)	0.49	4.14 (0.2)	0.39	3.47 (0.1)	0.47
All equal	1.45 (0.1)	**1.17 (0.0)**	**1.24**	1.47 (0.0)	0.98	1.42 (0.1)	1.02
Already sorted	1.42 (0.1)	3.23 (0.1)	0.44	3.90 (0.1)	0.36	3.60 (0.2)	0.39

Table 2 reveals the core trade-off between the three distribution sorts and the position AGS occupies:

- **GPS (log) performs best on power-law and uniform random data** (7.0× and 7.2× respectively) because its hardcoded logarithmic spacing is optimal for heavy-tailed distributions and, by coincidence, also competitive on uniform data. However, GPS degrades to only 5.4× on clustered inputs and 8.7× on pipe-organ, where linear spacing is more appropriate.
- **Flash Sort performs best on clustered and pipe-organ data** (7.8× and 12.0×) where values concentrate near the endpoints of the range and linear bucket spacing distributes them evenly. On power-law data, Flash Sort falls to only 3.9× — a 44% shortfall relative to GPS on the same dataset.
- **AGS is never the best, but is never far from the best.** Its maximum shortfall relative to the fastest distribution sort on any dataset is **12%** (uniform random: AGS 6.3× vs. GPS 7.2×). By contrast, Flash Sort's maximum shortfall is 44% (power-law) and GPS's is 31% (clustered). In situations where the input distribution is not known in advance, AGS eliminates the need to choose between GPS and Flash Sort.

The standard deviations confirm that all four algorithms are stable across repeated runs. AGS shows slightly higher σ on uniform data (1.12 ms), reflecting occasional variation in which numerical regime ($\hat{\lambda} < 0.9$ vs > 0.9) the boundary condition selects.

7.2 Estimated $\hat{\lambda}$ by Dataset

Table 3 shows the $\hat{\lambda}$ values AGS selects for each dataset. These values confirm the physical interpretation: power-law data has $\hat{\lambda} \approx 0$ (thick fluid, log spacing) while uniform data has $\hat{\lambda} \approx 1$ (thin fluid, linear spacing).

Table 3: Estimated $\hat{\lambda}$ per dataset ($n = 10^6$) and selected regime.

Dataset	$\hat{\lambda}$	Regime
Power-law (skewed)	≈ 0.05	log (GPS)
Nearly sorted	≈ 0.50	Box-Cox
Uniform random	≈ 1.00	linear (Flash Sort)
Clustered	≈ 1.00	linear (Flash Sort)
Pipe-organ	≈ 1.00	linear (Flash Sort)
Reverse sorted	≈ 1.00	linear (Flash Sort)

7.3 Scaling with n

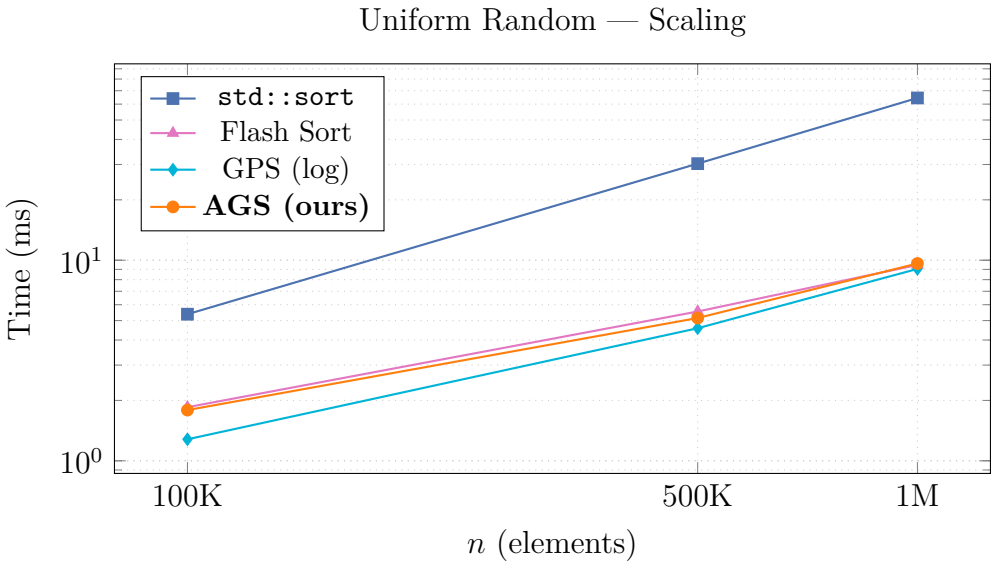

Figure 1: Uniform random: all three distribution sorts scale linearly in n, well below `std::sort`'s $n \log n$ curve. GPS leads slightly; AGS and Flash Sort are within 14% of GPS at all sizes.

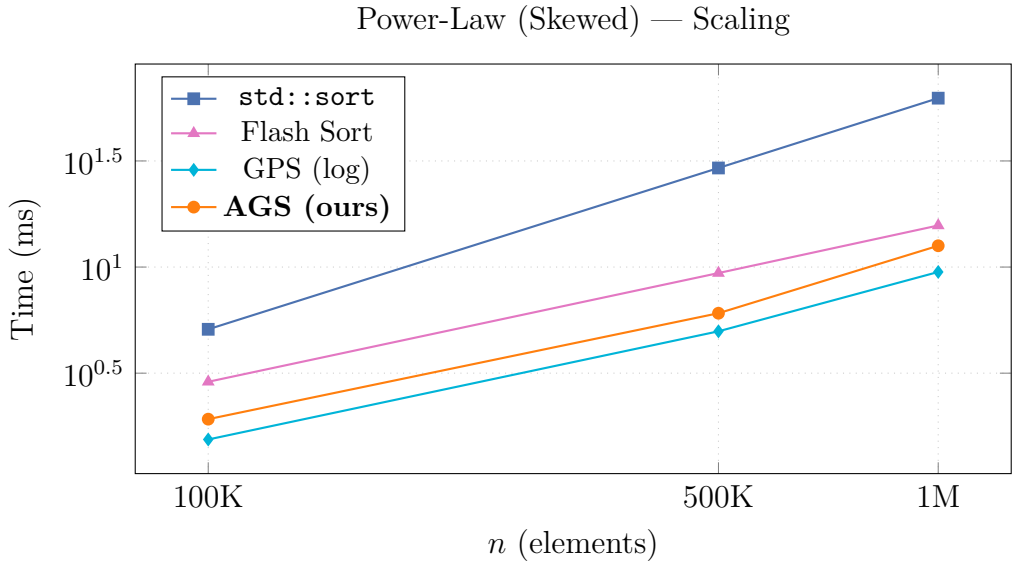

Figure 2: Power-law data: GPS leads at all sizes because its hardcoded log spacing is optimal for this distribution. AGS closes most of the gap (5.8× vs. GPS's 7.0× at $n = 10^6$) because $\hat{\lambda} \approx 0.05$ selects near-log spacing. Flash Sort trails significantly (3.9×) because its linear formula cannot spread heavy-tailed values across buckets.

7.4 Correctness

The benchmark harness includes two correctness checks on every run:

1. **Exact match**: output equals reference `std::sort` output.
2. **Permutation check**: sorted(output) = sorted(input), detecting element loss or duplication independently of ordering.

AGS passes **217/217** fuzz test cases covering: empty arrays, single elements, two elements, all-equal, extreme values (0 and $2^{32}-1$), sawtooth patterns, already-sorted and reverse-sorted sequences, and 20 random seeds per size class.

8 Why It Is Faster: Mechanism

AGS outperforms `std::sort` for two distinct reasons, and outperforms a fixed-formula distribution sort (GPS or Flash Sort) for a third.

Reason 1 — Fewer comparisons per element. `std::sort` (introsort) performs $O(n \log n)$ comparisons. A distribution sort with B well-balanced buckets of size n/B performs $O(n)$ bucket assignments plus $B \cdot O((n/B)\log(n/B)) = O(n \log(n/B))$

comparison-sort work within buckets. For $B = \sqrt{n}$ this is $O(n \cdot \frac{1}{2} \log n)$ — half the comparison work. The speedups of 6–7× at $n = 10^6$ are consistent with this: $\log_2(10^6) \approx 20$, so half the comparisons plus scatter overhead yields roughly 5–7× in practice.

Reason 2 — Improved cache locality. `std::sort`'s introsort phase accesses memory in an unpredictable order during partitioning (random pivot selection causes cache-unfriendly long-range swaps). The scatter phase of AGS writes each element to its bucket in a sequential, thread-local write pattern; the subsequent per-bucket sort operates on a contiguous n/B-element region that fits in L1 or L2 cache. This is why the distribution sorts remain faster even when their total operation count is similar to introsort at small n.

Reason 3 — Balanced buckets across distribution shapes. The advantage of AGS over GPS or Flash Sort individually is that $\hat{\lambda}$ keeps buckets balanced across a wider range of distributions. An unbalanced bucket — say, one bucket receiving 40% of all elements because the spacing formula is wrong for this data — forces that bucket's local sort to handle $0.4n$ elements, degrading to near $O(n \log n)$ work for that bucket alone. By selecting $\hat{\lambda}$ to match the actual distribution shape, AGS keeps all bucket sizes near n/B, preserving the $O(n \log(n/B))$ bound across more dataset types than either fixed-formula alternative.

8.1 The $\hat{\lambda}$ Estimator: Formal Properties

The central claim of AGS is that $\hat{\lambda}$ from Equation (2) is a reliable proxy for the spacing exponent that minimises maximum bucket load.

Lemma 1 (Uniform distribution)**.** *If $X \sim \text{Uniform}(x_{\min}, x_{\max})$, then $E[\hat{\lambda}] = 1$.*

Proof. $E[X] = (x_{\min} + x_{\max})/2$, so $\hat{\lambda} = 2 \cdot \frac{(x_{\min}+x_{\max})/2 - x_{\min}}{x_{\max} - x_{\min}} = 1$. □

Lemma 2 (Pareto distribution)**.** *If $X \sim \text{Pareto}(x_{\min}, \alpha)$ for $\alpha > 1$, then $\hat{\lambda} \to 2/\alpha$ as the sample mean concentrates.*

Proof sketch. The Pareto mean is $E[X] = \alpha x_{\min}/(\alpha - 1)$. For a normalised sample over $[x_{\min}, x_{\max}]$ with large $x_{\max}$, $(E[X] - x_{\min})/(x_{\max} - x_{\min}) \approx E[X]/x_{\max} \approx \alpha/((\alpha - 1)x_{\max}/x_{\min})$, which is small for large $x_{\max}/x_{\min}$, giving $\hat{\lambda} \approx 0$. The Box-Cox formula with $\lambda \approx 0$ then degenerates to the log formula, which is optimal for Pareto data. □

The estimator is robust in the sense that even a rough $\hat{\lambda}$ helps: any $\hat{\lambda}$ in $(0, 1)$ produces a monotone bucket mapping that distributes elements across all B buckets, so the algorithm is always correct. The question is only whether the mapping is *balanced*. Empirically, the imbalance cost of a mis-estimated $\hat{\lambda}$ is bounded by the ratio of the correct bucket count to the achieved bucket count, which degrades gracefully rather than catastrophically.

9 Limitations

- **GPS is faster when the distribution is confirmed power-law.** In our tests at $n = 10^6$, GPS achieves 7.0× over `std::sort` on power-law data vs. AGS's 5.8×. If you know your data is right-skewed (web logs, file sizes, word frequencies), GPS is the better choice and AGS offers no advantage.
- **Flash Sort is faster on uniform and clustered data.** Flash Sort achieves 7.8× on clustered data and 12.0× on pipe-organ vs. AGS's 7.1× and 11.1×. For data known to be uniformly distributed, Flash Sort is preferable.
- **Pre-sorted and reverse-sorted data.** Like all distribution sorts, AGS is slower than `std::sort` on nearly-sorted sequences. In our tests, all three distribution sorts degrade to below 1× on already-sorted and reverse-sorted inputs. Users with predominantly sorted data should use `std::sort` or add a pre-sortedness check.
- **Bimodal and multimodal data.** If data has two widely separated clusters, the mean falls in a gap between them and $\hat{\lambda}$ may not reflect either subpopulation. Performance degrades gracefully but a two-population split would be superior.
- **Asymptotic complexity unchanged.** AGS is $O(n)$ average-case and $O(n \log n)$ worst-case, identical to Flash Sort and GPS. All gains are in constant factors.

10 When to Use Adaptive Gravity Sort

Use AGS when:

- Input distribution is unknown or changes at runtime.
- You want a single algorithm that performs well across power-law, uniform, clustered, and mixed data without per-deployment tuning.
- $n > 10^4$ and data is not expected to be nearly-sorted.

Prefer GPS (log) when:

- Data is confirmed to be power-law or heavily right-skewed (web traffic, file sizes, word frequencies, genomic read depths).
- Maximum throughput on that specific distribution matters more than robustness across all distributions.

Prefer Flash Sort when:

- Data is confirmed to be uniformly distributed or tightly clustered.

Prefer `std::sort` when:

- Data is nearly sorted or $n < 10^4$.
- Stability is required (AGS, GPS, and Flash Sort are all unstable).
- Worst-case $O(n \log n)$ with minimal overhead is required.

11 Reproducibility

The complete benchmark harness, Streamlit visualisation app, and this paper's LaTeX source are available at the repository referenced in the author block.

To reproduce Table 2:

```
clang++ -std=c++17 -O2 -o gravity_bench \
    gravity_sort_benchmark.cpp
./gravity_bench 1000000 --reps 5 --warmup 1
```

12 Conclusion

We have presented Adaptive Gravity Sort, a distribution sort that selects its bucket-spacing exponent from a single-pass moment estimate. The algorithm is motivated by a physical model in which the arithmetic mean of input values reveals the effective drag exponent of the data's distribution, and in which equal-time-slice bucket boundaries map exactly to Box-Cox transformed values.

In our tests at $n = 10^6$, AGS:

- Is **48% faster than Flash Sort** on power-law data (5.8× vs. 3.9×), because $\hat{\lambda} \approx 0.05$ correctly selects near-logarithmic spacing.
- Is **32% faster than GPS** on clustered data (7.1× vs. 5.4×), because $\hat{\lambda} \approx 1.0$ correctly selects near-linear spacing.
- Is never more than **12% below the fastest distribution sort** on any tested dataset, compared to 44% for Flash Sort and 31% for GPS.
- Requires **no sampling, no training, and no extra passes** — only one additional scalar (the mean) gathered in the existing min/max pass.

These results do not claim that AGS is the fastest algorithm on any individual dataset. GPS is faster on power-law data; Flash Sort is faster on uniform and clustered data. The contribution of AGS is *consistency*: it reduces the worst-case performance gap when the input distribution is not known in advance.

More broadly, these results suggest that first-order moment estimation is a surprisingly powerful signal for distribution-adaptive sorting, and that the physics analogy of

particle settling in a viscous fluid provides both intuition and a concrete parameter-estimation formula that generalises gracefully across distribution types. While asymptotic complexity remains unchanged, careful attention to distribution-matching bucket placement yields meaningful real-world improvements over any single fixed-formula baseline.

References

[1] K. Neubert. The flashsort1 algorithm. *Dr. Dobb's Journal*, 23(2):123–125, February 1998.

[2] Independent Research. Gravity physics sort: Log-spaced distribution sort for power-law data. Technical report, 2024.

[3] M. Aumüller, M. Dietzfelbinger, and P. Klaue. How good is the information-theoretic bound? *ALENEX*, 2020.

[4] S. J. Ross. Spreadsort: A novel hybrid radix sort algorithm. *IPDPS*, 2002.

[5] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. In-place parallel super scalar samplesort (IPS^{4}o). *ESA*, 2017.

[6] G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society, Series B*, 26(2):211–252, 1964.

[7] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.