# Histogram Sort with Sampling

Vipul Harsh
University of Illinois at Urbana Champaign
vharsh2@illinois.edu

Laxmikant Kale
University of Illinois at Urbana Champaign
kale@illinois.edu

Edgar Solomonik
University of Illinois at Urbana Champaign
solomon2@illinois.edu

## ABSTRACT

To minimize data movement, state-of-the-art parallel sorting algorithms use techniques based on sampling and histogramming to partition keys prior to redistribution. Sampling enables partitioning to be done using a representative subset of the keys, while histogramming enables evaluation and iterative improvement of a given partition. We introduce Histogram sort with sampling (HSS), which combines sampling and iterative histogramming to find high-quality partitions with minimal data movement and high practical performance. Compared to the best known (recently introduced) algorithm for finding these partitions, our algorithm requires a factor of $\Theta(\log(p)/\log\log(p))$ less communication, and substantially less when compared to standard variants of Sample sort and Histogram sort. We provide a distributed-memory implementation of the proposed algorithm, compare its performance to two existing implementations, and provide a brief application study showing benefit of the new algorithm.

## CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**;

## KEYWORDS

parallel sorting; data partitioning; histogramming; sampling;

**ACM Reference Format:**
Vipul Harsh, Laxmikant Kale, and Edgar Solomonik. 2019. Histogram Sort with Sampling. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3323165.3323184

## 1 INTRODUCTION

Finding a global partition of the data is the key challenge that separates parallel sorting from sequential sorting. Partition-based sorting algorithms, that partition the data prior to redistributing it (in contrast to merge-based sorting algorithms), are advantageous on modern architectures due to their low communication cost. Sampling data either uniformly or selectively and histogramming the split produced by the partition are the two most common techniques for determining a good partition. By quantifying the parallel execution cost in terms of computation and communication, we demonstrate that a simple but careful combination of these two techniques leads to an algorithm that provides both theoretical and practical improvements over the best previously known algorithm.

A parallel sorting algorithm needs to redistribute $N$ keys across $p$ processors such that they are in a globally sorted order. In such an order, keys on processor $k$ are no greater than keys on processor $k+1$ and keys are sorted within each processor. An exact splitting (we use the terms partitioning and splitting interchangeably) is achieved if all processors own the same number of keys, while an approximate splitting guarantees that every processor owns no more than $N(1+\epsilon)/p$ keys for some $\epsilon$; we call this an $\epsilon-$balanced partition. Given sorted keys with an approximate splitting for $\epsilon = O(1)$, an exact splitting can be achieved at no cost in asymptotic running time. However, it increases the running time in practice and is often not required by applications. Algorithms that guarantee a balanced partition for a given $\epsilon$ are favorable since a large $\epsilon$ increases the memory footprint and can hurt application performance.

The most natural way to cheaply determine a global partition is to collect a sample of keys, and infer a global partition from the ideal partition of the sorted sample. Sample sort [15] and its variants are basic realizations of this approach, which are widely used in practice [25], and also serve as building blocks for our algorithm. Selecting a random sample of the data and partitioning the input key space using the random sample suffices to achieve the desired load balance w.h.p.[1] so long as $\Theta(p\log N/\epsilon^2)$ keys are collected in the sample [19]. A deterministic balanced splitting is also possible via sampling, for example, using sample sort with regular sampling [24, 28]. With regular sampling, the algorithm collects $p/\epsilon$ keys from each processor, that partition the local input data on each processor evenly, requiring a total sample size of $\Theta(p^2/\epsilon)$ from all processors for a balanced split. However, these classical results leave substantial room for improvement. We show that it suffices to collect a number of samples that scales near-linearly with $p$ and logarithmically with $1/\epsilon$.

Histogram sort [22, 29], which embodies the histogramming technique, iteratively refines a partition (set of splitters), by repeatedly collecting histograms of the total number of input keys in each interval induced by the latest set of splitters. The number of histogramming rounds required to determine all splitters within the allowed threshold is bounded by $O(\log \mathcal{Z})$, where $\mathcal{Z}$ is the size of the input domain. For skewed distributions, the number of rounds could be large and the use of the input domain implies that Histogram sort is not a pure comparison-based sorting algorithm.

Recently, Axtmann et al. [5] proposed a scheme based on the histogram of the partition induced by a random sample. They show that using $\Theta(p(\log(p) + 1/\epsilon))$ samples results in an $\epsilon$-balanced

[1]with high probability. In our context, $\geq (1 - O(p^{-c}))$ for some fixed $c > 0$

*SPAA '19, June 22–24, 2019, Phoenix, AZ, USA*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323184

partition w.h.p.. Our main contribution is demonstrating that by using $\log(\log(p)/\epsilon)$ steps of refinement with histogramming, $\Theta(p\log(\log(p)/\epsilon))$ samples in total suffice for an $\epsilon$-balanced partition. Our algorithm improves the communication cost for the partitioning step (proportionally to the reduction in sample size), at the cost of a small increase in the number of parallel steps (BSP supersteps / synchronizations). This factor of improvement also holds if the partitioning schemes are used in a multi-stage fashion, for example by first splitting the data into $\sqrt{p}$ parts, then sorting each part recursively with $\sqrt{p}$ processors.

The improvement in cost warrants the introduction of an algorithm that combines sampling and histogramming, which we call Histogram sort with sampling (HSS). HSS carefully weaves together standard techniques in such a way that the resulting algorithm is provably better than the state of the art. The analysis of the algorithm is nontrivial; the main challenge resolved in this paper is in identifying and proving an invariant that shows global quadratic convergence of the partitioning algorithm. The main intuition behind the algorithm and proof comes from consideration of *splitter intervals*, which are subranges around ideal splitter keys in the globally sorted order of input keys. In each round, HSS uniformly samples keys in the union of all splitter intervals, then tightens each splitter interval using a histogram of the collected sample. Our main analytical result is that the size of the union of the splitter intervals decreases geometrically with the number of rounds.

By characterizing HSS, we establish the theoretical soundness of iterative histogramming as a technique, that is known to be effective [21, 29] in practice. Our algorithm is simple, provably robust to arbitrary distributions with repeated keys, and effective in practical scenarios where the input is already partially sorted. We provide a parallel Charm++ implementation of the HSS algorithm and demonstrate improvements over one of the fastest publicly-available distributed-memory parallel sorting algorithms, HykSort [30] in both single-staged and multi-staged settings. Additionally, we show that our algorithm improves performance with respect to Histogram sort within the ChaNGa $N$-body code [20], which uses sorting every time-step to distribute moving cosmological bodies along a space-filling curve. Our theoretical analysis of parallel execution cost, comparative performance evaluation, and application case study unanimously identify HSS as the preferred parallel sorting algorithm. We have made our code available online [2].

## 2 PROBLEM STATEMENT

Let $A(0), \ldots, A(N-1)$ be an input sequence distributed across $p$ processing elements, such that each processor has $N/p$ keys. We assume that there are no duplicates in the input. In Section 6.1 we discuss how to reduce a sorting problem with duplicate keys to a sorting problem with no duplicates, with very little overhead. Our proofs and algorithm also translate to scenarios where input keys are not evenly distributed across each processor. Parallel sorting corresponds to redistributing and reordering the elements so that the $i$th processor owns the $i$th subsequence of keys in the sequence $I(0), \ldots, I(N-1)$, where $\{I(0), \ldots, I(N-1)\} = \{A(0), ..., A(N-1)\}$ and $I(i) \leq I(i+1)$. We say that key $A(j) = I(r)$ has rank $r$. In practice, keys are typically associated with values, but in the context of the algorithms we study, handling values of a given size is straightforward.

It is common to additionally require that the resulting distribution of sorted keys is load balanced among processors. We compare algorithms with the standard assumption that the distribution is *locally balanced*, i.e. each processor owns no more than $(1+\epsilon)N/p$ keys. However, our algorithm achieves a stronger guarantee, namely that the distribution is *globally balanced*, i.e. processor $i$ owns all keys greater than or equal to $S(i)$ and less than $S(i+1)$, where each $S(i)$ is a *splitter* that satisfies $S(i) = I(\chi(i))$, with

$$\chi(i) \in \mathcal{T}_i,$$
$$\text{where } \textit{target range: } \mathcal{T}_i = \left[\frac{Ni}{p} - \frac{N\epsilon}{2p}, \frac{Ni}{p} + \frac{N\epsilon}{2p}\right]$$

One practical advantage of a globally balanced distribution in the context of iterative applications, is that if the initial distribution is nearly sorted and globally balanced, the data exchange step is guaranteed to require little data movement.

We note that, given either type of load-balanced splitting, post-processing may be done to obtain an *exact* splitting [12]. For a locally balanced distribution, this might require some processors to potentially communicate all of their data to one or two other processors. However, given a globally balanced distribution, achieving an exact splitting would require communicating only at most $N\epsilon/p$ keys per processor. Therefore, a more fundamental distinction is in whether a parallel sorting algorithm *maintains* load balance at all times, i.e. no processor is ever assigned more than $(1+\epsilon)N/p$ keys. Satisfying this condition permits bounded memory footprint, which is desirable for a parallel sorting library implementation.

The focus of this paper is on the data-partition step of partition-based sorting algorithms. Sample sort by regular sampling [24, 28], histogram sort [22, 29], sample sort by random sampling [11, 15], parallel sorting by over partitioning [23], AMS sort [5], HyKSort [30] fall into this category. Partition-based sorting algorithms determine a set of splitters that achieve either a locally or globally balanced splitting, then redistribute keys. The algorithm can run in multiple *stages* by splitting up data among subsets of processors and sorting recursively within each subset. In section 5, we evaluate the time complexity of HSS and a multi stage variant of HSS using the standard Bulk Synchronous Parallel (BSP) model by Valiant [32].

## 3 RELATED WORK

Sample sort [15] and histogram sort [22] are closely related to our algorithm, we review these and other sorting algorithms before proceeding to our main result.

### 3.1 Sample sort

Sample sort [10, 15, 18, 24, 28] is a standard well studied parallel sorting algorithm. Sample sort samples $s$ keys from each processor, and sends them to a central processor to form an overall sample of size $M = ps$ keys. Let $\Lambda = \{\lambda_0, \lambda_1 ..., \lambda_{ps-1}\}$ denote the combined sorted sample. Sample sorting algorithms choose $p-1$ keys from $\Lambda$ as the final splitters. Generally, sample sort algorithms consist of the following three-phase skeletal structure.

(1) **Sampling Phase**: Every processor samples $s$ keys and sends it to a central processor. $s$ is often referred to as the oversampling ratio. See Section 3.2 for sampling methods.
(2) **Splitter Determination**: The central processor receives samples of size $s$ (from Step 1) from every processor resulting in a combined sample $\Lambda$ of size $(ps)$. The central processor then selects splitter keys: $\mathcal{S} = \{S(1), S(2)..., S(p-1)\}$ from $\Lambda$ by picking evenly spaced keys from $\Lambda$. The splitters partition the key range into $p$ ranges, each range assigned to one processor. Once chosen, the splitters are broadcast to all processors.
(3) **Data Exchange**: Once a processor receives the splitter keys, it sends each of its keys to their destination processor. As discussed earlier, a key in range $[S(i), S(i+1))$ goes to processor $i$. This step is akin to one round of all-to-all communication and places all input data onto their assigned destination processors. Once a processor receives all data that is assigned to it, it merges them using a sequential algorithm, like merge sort.

## 3.2 Sample sort: Sampling methods

We discuss two sampling methods- random sampling and regular sampling, for the sampling phase (step 1) of sample sort.

*3.2.1 Random sampling.* With random sampling as described by Blelloch et al. [11], each processor divides its local sorted input into $s$ blocks of size $(N/ps)$ and samples a random key in each block, where $s$ is the oversampling ratio. The splitters are chosen by picking evenly spaced keys from the overall sample of size $ps$, collected from all processors. Of particular reference to our work is the following theorem, (Lemma $B$.4 in [11]).

THEOREM 3.1. *With $O\left(\frac{p \log N}{\epsilon^2}\right)$ samples overall, sample sort with random sampling achieves $(1+\epsilon)$ load balance w.h.p..*

*3.2.2 Regular sampling.* With regular sampling [24, 28], every processor deterministically picks $s$ evenly spaced keys from its local sorted data. The central processor collects these samples and selects splitters from this sample, just like random sampling. We reproduce the following theorem from [24, 28].

THEOREM 3.2. *If $s = \frac{p}{\epsilon}$ is the oversampling ratio, then sample sort with regular sampling achieves $(1+\epsilon)$ load balance.*

Because of the large number of samples required, the sampling phase is unscalable for regular sampling. Sample sort with random sampling is more efficient, but scalability is still hindered in practice because of the large sample size required to achieve a load-balanced splitting.

## 3.3 Histogram Sort

Histogram sort [22, 29] addresses load imbalance by determining the splitters more reliably. Instead of determining all splitters using one large sample, it maintains a set of candidate splitter keys and performs multiple rounds of histogramming, refining the candidates in every round. Computing the histogram of a set of candidate keys gives the global rank of each candidate key. This information is used by the algorithm to finalize splitters or to refine the candidate splitter keys. Once all the splitters are within the given threshold, it finalizes the splitter keys from the set of candidate keys. The data exchange phase of Histogram sort is identical to the third phase of sample sort. We give an overview of the splitter determination step in histogram sort.

(1) The central processor broadcasts a probe consisting of $M$ sorted keys to all processors. Usually, the initial probe is spread out evenly across the key range (unless additional distribution information is available).
(2) Every processor counts the number of keys in each range defined by the probe keys, thus, computing a local histogram of size $M$.
(3) Local histograms are summed to obtain a global histogram at a central processor using an $M$-item reduction.
(4) The central processor finalizes and broadcasts the splitters if a probe key within the desired range has been found for each of the $p-1$ unknown splitters. Otherwise, it refines its probes using the histogram obtained and broadcasts a new set of probes for the next round of histogramming, in which case the algorithm loops back to Step 2.

Candidate keys are refined by splitting the input key range between successive candidate keys according to their ranks [29]. Histogram sort is guaranteed to achieve any arbitrary specified level of load balance. It is also scalable in practice for many input distributions, since the size of the histogram every round is typically kept small - of the order $O(p)$. The number of histogramming rounds required to determine all splitters within the allowed threshold is at most $\log_2(\mathcal{Z})$, where $\mathcal{Z}$ is the range of the input i.e. maximum key minus the minimum key (treating $\epsilon$ as a constant here). The number of rounds can be large, especially for skewed input distributions. Histogram sort has been successfully employed in real world, highly parallel scientific applications, for instance *ChaNGa* [20].

## 3.4 Other Sorting Algorithms

In parallel sorting by over partitioning [23], proposed for shared memory multiprocessors, every processor picks a random sample of size $pks$ from its local input and sends it to a central processor. The central processor sorts the overall collected sample and choses $pk-1$ splitters by selecting the $s^{th}, 2s^{th}, ..., (pk-1)s^{th}$ keys. These splitters partition the input into $pk$ buckets, more than required. The splitters are made available to all processors and the local input is partitioned into sublists based on the splitters. These sublists form a task queue and each processor picks one task at a time and processes it by copying the data to the appropriate position in the memory, determined using the splitters. The idea of over partitioning is closely related to histogramming. Recent work on sorting algorithms for asymmetric read and write costs [8] and low cache complexity [9] are complimentary to our work and can be used in combination with HSS.

*3.4.1 Merge based parallel sorting algorithms.* In this paper, we primarily focus on partition-based algorithms. *Merge-based* algorithms are another class of sorting algorithms that merge data in parallel using sorting networks. An early result was due to Batcher [6] which uses time (or equivalently depth in a sorting network) $O(\log^2 N)$ with $N$ processors. The AKS network [4] was the first sorting circuit of depth $O(\log N)$, but had large constants because of the use of expander graphs [13, 26]. Later, Cole [14] proposed a circuit that also ran in $O(\log N)$ time using $N$ processors, but had smaller constants. A communication optimal algorithm in the BSP model was proposed by Goodrich [16]. Cole's merge sort and its adaptation to BSP by Goodrich follow a merge-tree, but employ sampling to determine a partition that accelerates merging. Overall, unless data-partitioning schemes are also employed, merge-based algorithms tend to be less performant due to their need for more BSP supersteps for the expensive data-exchange step and in some cases more communication than partition-based alternatives.

## 3.5 Large scale parallel sorting algorithms

Several recent works have focused on large scale sorting. HykSort [30], a state of the art practical algorithm, employs multi-staged splitting and communication to achieve better scalability. HykSort is a hybrid of sample sort and hypercube quick sort. Even though Hyksort's algorithm for splitter selection also uses sampling and histogramming, there is a key difference in the sampling method between HSS and HykSort (see Section 4.2.2). As we show in Appendix A, this is critical for the running time as HykSort requires at least $\Omega\Big(\log(p)\Big/\log^2\log(p)\Big)$ times more samples than HSS in the worst case. Our experiments confirm faster convergence in HSS and benefits of HSS over Hyksort in both single-staged and multi-staged settings (see section 6).

AMS-sort [5] employs overpartitioning for splitting. AMS-sort's scanning algorithm (Lemma 2 of [5]), used to select splitters, is better than HSS with one round of histogramming by a factor of $\Theta(\min(\log p, 1/\epsilon))$. However, HSS with multiple rounds of histogramming is more efficient than AMS-sort. The scanning algorithm does not easily generalize to multiple rounds of histogramming. Further, HSS achieves a globally balanced partition, while AMS-sort achieves only a locally-balanced splitting, providing less robustness in preservation of existing distributions. AMS-sort can be performed in a multi-stage fashion, with successive steps of splitting and data exchange across a decreasing set of processors. HSS can run in the same multi-stage fashion, but with each data partitioning step done with multiple rounds of histogramming. We compare the asymptotic running times of AMS-sort and HSS with multiple stages in Table 3.

## 3.6 Single stage AMS sort

The single stage AMS-sort [5] collects a single set of samples, performs one round of histogramming, then picks a locally balanced splitting based on the histogram. The splitters obtained after the first histogramming round achieve the specified level of load balance w.h.p. with an oversampling parameter that is much less than sample sort with random sampling. In Section 5, we show that the cost of histogramming is asymptotically same as the cost of sampling an equal number of keys, so AMS sort achieves a clear theoretical improvement over sample sort. We review the AMS algorithm in some detail, due to its close relation to our approach.

*3.6.1 Scanning algorithm.* AMS sort uses a scanning technique to decide the splitters once the histogram is obtained. The algorithm scans through the histogram and assigns a maximal number of consecutive buckets (all keys between two consecutive keys in the total sorted sample) to each processor. Specifically, after assigning $i$ buckets to the first $j$ processors, it assigns buckets $i+1, \ldots, i+k$ to processor $j+1$, where $k \geq 0$ is picked maximally so that the total load on processor $j+1$ does not exceed $N(1+\epsilon)/p$. The last processor gets all the remainder elements. If the sample size is sufficiently large, the average load on the first $p-1$ processors is greater than $N/p$ w.h.p..

In particular, a sample of size $\Theta(p(\log p + 1/\epsilon))$ is necessary to achieve a locally balanced partitioning w.h.p.. Demonstrating this formally is difficult due to the conditional dependence of loads assigned to consecutive processors. We formalize the proof of a key lemma in the analysis of scanning algorithm in [5] (Appendix A in [17]). In Table 2 we compare the cost of AMS sort to versions of sample sort and HSS. AMS sort achieves a lower asymptotic complexity than HSS with a single round of histogramming. However, HSS can achieve an asymptotically lower complexity in $O(1)$ BSP supersteps and even lower complexity with $O(\log\log p/\epsilon)$ supersteps while at the same time providing a globally balanced distribution.

## 4 HISTOGRAM SORT WITH SAMPLING

The basic skeleton of HSS is similar to that of Histogram Sort. In addition, HSS employs sampling to determine the candidate probes for histogramming. Every histogramming round is preceded by a sampling phase where each processor samples local keys and the overall sample collected from all processors is used for the histogramming round. By histogramming on the sample, HSS requires significantly fewer samples compared to sample sort.

## 4.1 HSS with one histogram round

We first describe HSS with one round of histogramming, whose data-partitioning step is slightly less efficient than AMS sort, by a factor of $\Theta(\min(\log p, 1/\epsilon))$. However, HSS achieves a globally-balanced splitting because of which HSS with one round is easily generalizable to multiple rounds of histogramming, which we discuss in subsequent sections and is the main contribution of this paper. Extending the single round scanning algorithm of AMS sort to multiple rounds for improved complexity is non-trivial. With multiple rounds of histogramming, HSS is more efficient than the scanning algorithm of AMS-sort, in fact, $O(1)$ rounds of histogramming suffice for an asymptotic improvement.

Recall that in HSS, a satisfactory $i$th splitter (in terms of global load balance) is found when a candidate key is found that is known to have rank in target range $\mathcal{T}_i = [Ni/p - N\epsilon/2p, Ni/p + N\epsilon/2p]$. If for each target range $\mathcal{T}_i$, the sample contains at least one key with rank in $\mathcal{T}_i$, then after histogramming on the sample, all such splitters will be found. Intuitively, the algorithm should sample adequate number of keys so that at least one key is picked from each $\mathcal{T}_i$ w.h.p..

Lemma 4.1. *If every key is independently picked in the sample with probability, $\frac{ps}{N} = \frac{2p\ln p}{\epsilon N}$, where $s$, the oversampling ratio is chosen to be $\frac{2\ln p}{\epsilon}$, then at least one key is chosen with rank in $\mathcal{T}_i$ for each $i$, w.h.p..*

Proof. Recall that the input set is denoted by $A$. The size of $\mathcal{T}_i = N\epsilon/p$. The probability that no key is chosen with rank in $\mathcal{T}_i$

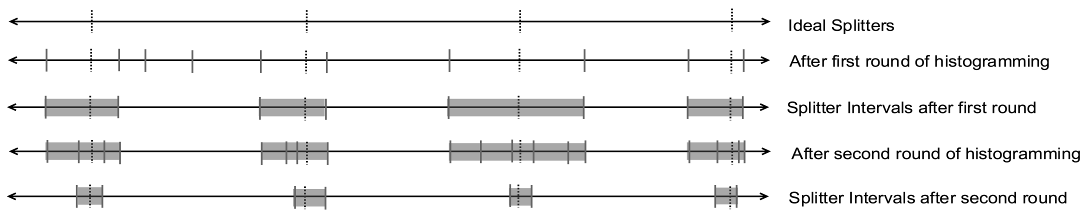
Ideal Splitters
After first round of histogramming
Splitter Intervals after first round
After second round of histogramming
Splitter Intervals after second round

**Figure 1: Figure illustrating HSS with multiple rounds. After first round, samples are picked only from the splitter intervals, in proportional to the interval length. Notice how the splitter intervals shrink as the algorithm progresses.**

| | | |
|---|---|---|
| problem | $N$ | number of keys to sort in total |
| | $p$ | number of processors sorting keys |
| | $A(i)$ | the $i$th input key |
| | $I(r)$ | key with rank $r$ in the overall global order |
| | $\mathcal{T}_i$ | $[Ni/p - N\epsilon/p,\ Ni/p + N\epsilon/p]$ is the target range for the $i$th splitter |
| algorithm | $s_j$ | the sampling ratio for the $j$th round, in particular each key in $\gamma_j$ is in the $j$th sample with probability $s_j p/N$ |
| | $L_j(i)$ | rank of largest sample key below rank $Ni/p$ after $j$ rounds |
| | $U_j(i)$ | rank of smallest sample key above rank $Ni/p$ after $j$ rounds |
| | $\mathcal{I}_j(i) =$ | $[I(L_j(i)), I(U_j(i))]$ is the $i$th splitter interval after $j$ rounds |
| | $\gamma_j$ | the union of all splitter intervals after $j$ rounds |

**Table 1: Notation used in paper, index $j$ refers to Histogramming round, while $i$ is the processor index.**

in the overall sample for a given $i$ is given by,

$$\left(1 - \frac{ps}{N}\right)^{|\mathcal{T}_i|} = \left(1 - \frac{2p\ln p}{\epsilon N}\right)^{\frac{N\epsilon}{p}} \leqslant e^{-2\ln p} = \frac{1}{p^2}$$

Since there are $p - 1$ splitters, the probability that no key is chosen from some $\mathcal{T}_i$, is at most $(p-1) \times p^{-2} < 1/p$. □

Lemma 4.1 leads us to the following theorem, showing global load balance of HSS with one round. The theorem will also be useful in the analysis of multiple rounds of histogramming, each round of which effectively increases the oversampling ratio by collecting the same number of samples from a smaller subset of the complete set of keys.

Theorem 4.2. *With one round of histogramming and sample size $O(p\log(p)/\epsilon)$, HSS achieves $(1+\epsilon)$ load balance w.h.p..*

## 4.2 HSS with multiple rounds

We show that HSS can be made more efficient by repeated rounds of sampling followed by histogramming. We build upon the key observation that after the first round of histogramming, samples for subsequent histogramming rounds can be intelligently chosen using results from previous rounds.

*4.2.1 Sampling method.* For the sampling phases, our algorithm chooses a sample from a subset $\gamma$ of the input. Initially, $\gamma$ represents the entire input. As the algorithm progresses, $\gamma$ gets smaller. HSS uses the following sampling method.

**Sampling Method**: Every key in $\gamma$ is independently chosen to be a part of the sample with probability $ps/N$, where we refer $s$ as the ***sampling ratio***. The above sampling method simplifies the analysis, since sampling from disjoint intervals are independent. Note that the notion of sampling ratio is different from the oversampling ratio of sample sort and one round HSS since the size of the overall sample collected from all processors with the above method is $(ps|\gamma|/N)$ in expectation.

*4.2.2 HSS with k histogramming rounds: Algorithm.*

(1) In the sampling phase before the first histogramming round, each input key is picked in the sample with probability $(ps_1/N)$, where $s_1$ is the sampling ratio for the first round. Samples are collected at a central processor and broadcast as probes for the first histogramming round.
(2) Every processor counts the number of keys in each range defined by the probe keys (the overall sample for the current round), thus, computing a local histogram. All local histograms are summed up using a global reduction and sent to the central processor.
(3) For each splitter $i$, the central processor maintains $L_j(i)$: the lower bound for the $i^{th}$ splitter rank after $j$ histogramming rounds, i.e. rank of largest key seen so far, which is ranked less than $Ni/p$. Likewise it maintains $U_j(i)$, rank of smallest key ranked greater than $Ni/p$. Once the histogram reduction results of the $j^{th}$ round are received, the central processor updates $L_j(i)$ and $U_j(i)$ and broadcasts the sample keys $I(L_j(i)), I(U_j(i))$ bounding each splitter by the *splitter interval* $\mathcal{I}_j(i) = [I(L_j(i)), I(R_j(i))]$.
(4) Once every processor is aware of the new splitter intervals, it begins its sampling phase for the $(j+1)^{th}$ round. Every key which falls in one of the splitter intervals is picked in the sample with probability $(ps_{j+1}/N)$, where $s_j$ denotes the sampling ratio for the $j^{th}$ round. If $j < k$, samples from all processors are collected at a central processor and broadcast for the next round of histogramming, in which case the algorithm loops back to step 2. If $j = k$, the histogramming phase is complete and the algorithm continues to step 5. Step 2, 3 and 4 can be executed efficiently if the local data is already sorted.
(5) Once the histogramming phase finishes, the key ranked closest to $Ni/p$ among the keys seen so far is set as the $i^{th}$ splitter. Later, we discuss how to choose $k$ and the sampling ratios $s_j$'s so that the splitters determined this way result in a globally balanced partition.

The critical difference between HykSort and HSS is in the sampling method. HykSort samples equally from all splitter intervals whereas HSS samples in proportion to the interval length. By sampling more from larger intervals, HSS is able to narrow down the intervals quicker. We show that HykSort requires at least $\Omega\Big(\log(p)\Big/\log^2\log(p)\Big)$ times more samples than HSS in the worst case (see Appendix A).

A crucial observation is that the splitter intervals shrink as the algorithm progresses and hence the sampling step is executed with a subset of the input that gets smaller every round. Let $\gamma_j$ denote the set of keys in the input that belong to one of the splitter intervals after $j$ rounds. $|\gamma_j|$ represents the size of the input that the algorithm samples from, for the $j^{th}$ round. We have, $|\gamma_j| \leqslant \sum_i U_j(i) - L_j(i)$, where $U_j(i) - L_j(i)$ is the number of keys in the $i$th splitter interval. Some splitter intervals can overlap, hence the inequality . In fact, it is easy to see that there is no partial overlap between two splitter intervals, that is, either two splitter intervals: $\mathcal{I}_j(i_1)$ and $\mathcal{I}_j(i_2)$ are disjoint or they are identical.

Our proof outline is as follows. First we show in Lemma 4.3 that the algorithm will achieve a good splitting w.h.p. if the sampling ratio for the final round (the $k^{th}$ round) is chosen to be large enough. Having shown that the algorithm terminates after $k$ rounds, achieving a globally load balanced partition, we bound the sample sizes in each round by first bounding $|\gamma_j|$ in terms of the sampling ratio $s_j$ necessary to obtain all splitters w.h.p.. Finally, we appropriately set the sampling ratios such that the size of the union of splitter intervals, that is, $|\gamma_j|$ decreases by a constant factor. Intuitively, sampling ratio $s_j$ in round $j$ (where samples are chosen only from the splitter intervals in round $j$) can be thought of as choosing samples from the entire input range with an oversampling ratio of $s_j$ and discarding unnecessary samples.

Lemma 4.3. *If $s_k = \frac{2\ln p}{\epsilon}$ be the sampling ratio for the $k^{th}$ round, then at least one key is chosen from each $\mathcal{T}_i$ after $k$ rounds w.h.p..*

Given a sampling ratio of $s_j = \frac{2p\ln p}{\epsilon N}$, all splitters are found w.h.p, by Lemma 4.1. We next bound the expectation of the size of the union of all splitter intervals.

Lemma 4.4. *Let $s_j$ be the sampling ratio for the $j^{th}$ round, $\mathcal{I}_j(i)$ be the splitter interval for the $i^{th}$ splitter after $j$ rounds and $\gamma_j$ denote the set of input keys that lie in one of the $\mathcal{I}_j$'s, then, $E(|\gamma_j|) \leqslant \frac{2N}{s_j}$.*

Proof. Since $L_j(i)$ and $U_j(i)$ are only improved every round,

$$L_{j-1}(i) \leqslant L_j(i) \leqslant \frac{Ni}{p} \leqslant U_j(i) \leqslant U_{j-1}(i)$$

Further $\forall x : 0 \leqslant x \leqslant \left(U_{j-1}(i) - \frac{Ni}{p}\right)$,

$$P\Big[U_j(i) - \frac{Ni}{p} \geqslant x\Big] = \left(1 - \frac{ps_j}{N}\right)^x$$

As a result, we can bound the size of the $i$th splitter interval,

$$\begin{aligned} E\Big[U_j(i) - \frac{Ni}{p}\Big] &= \sum_{x=1}^{U_{j-1}(i)-\frac{Ni}{p}} P\Big[U_j(i) - \frac{Ni}{p} \geqslant x\Big] \\ &= \sum_{x=1}^{U_{j-1}(i)-\frac{Ni}{p}} \left(1 - \frac{ps_j}{N}\right)^x \\ &\leqslant \sum_{x=0}^{\infty} \left(1 - \frac{ps_j}{N}\right)^x = \frac{N}{ps_j} \end{aligned}$$

By a similar argument we have that, $E\Big[\frac{Ni}{p} - L_j(i)\Big] \leqslant \frac{N}{ps_j}$,

$$\begin{aligned} \text{Thus,}\quad E[|\gamma_j|] \leqslant E\Big[\sum_{i=1}^{p-1} |\mathcal{I}_j(i) \cap A|\Big] &= \sum_i E\Big[U_j(i) - L_j(i)\Big] \\ &= \sum_i E\Big[\frac{Ni}{p} - L_j(i)\Big] + E\Big[U_j(i) - \frac{Ni}{p}\Big] \\ &\leqslant \sum_i \frac{2N}{ps_j} = \frac{2N}{s_j} \end{aligned}$$

□

Lemma 4.4 suggests that $\gamma_j$ will be small in expectation. The next lemma shows that it is also small w.h.p..

Lemma 4.5. *If $s_j < \sqrt{\frac{2p}{\ln p}}$, then, $|\gamma_j| \leqslant \frac{4N}{s_j}$ w.h.p.*

Proof. The main challenge in proving the above theorem is in handling the dependency in splitter intervals, for e.g. when they overlap. We first modify the definition of splitter intervals in the following way, so that the union of the splitter intervals remains unchanged.

$$U_j'(i) = \min\left(\frac{N(i+1)}{p}, U_j(i)\right),\ L_j'(i) = \max\left(\frac{N(i-1)}{p}, L_j(i)\right)$$

The above definition effectively strips the splitter interval $[I(L_j(i)), I(U_j(i))]$ to $[I(L_j'(i)), I(U_j'(i))]$. To see that stripping doesn't change the union of all splitter intervals, consider a $U_j(i)$ which is greater than $N(i+1)/p$. Then by definition, we have $U_j(i) = U_j(i+1)$. Thus, the portion of $\mathcal{I}_j(i)$ that extends beyond $N(i+1)/p$ is included in $\mathcal{I}_j(i+1)$. Hence, restricting $U_j(i)$ to $Ni/p + N/p$ doesn't change $\gamma_j$ - the union of $\mathcal{I}_j$'s. An inductive argument (by considering splitter intervals from left to right) shows that restricting all $U_j$'s doesn't change $\gamma_j$. A similar argument can be used for $L_j$'s.

Observe that, $U_j'(i)$'s are independent random variables. This is because the possible values of $U_j'(i_1)$ and $U_j'(i_2)$ for $i_1 \neq i_2$ are completely disjoint. The value of $U_j'(i)$ is determined completely by sampling in the interval $[Ni/p, N(i+1)/p)$ and since sampling in disjoint intervals are independent, $U_j'(i)$'s are independent.
We have, $E[U_j'(i) - Ni/p] \leq E[U_j(i) - Ni/p] \leq N/ps_j$.

$$\begin{aligned} \text{Thus, } P\Big[\sum_i U_j'(i) - \frac{Ni}{p} > \frac{2N}{s_j}\Big] \\ &= P\Big[\sum_i U_j'(i) - \frac{Ni}{p} - \frac{N}{ps_j} > \frac{N}{s_j}\Big] \\ &\leq P\Big[\sum_i U_j'(i) - \frac{Ni}{p} - E\Big[U_j'(i) - \frac{Ni}{p}\Big] > \frac{N}{s_j}\Big] \\ &\leq e^{-\frac{2N^2}{s_j^2} \Big/ \sum_i (N/p)^2} = e^{-\frac{2p}{s_i^2}} \leq e^{-2\ln p} \\ &\leq \frac{1}{p^2} \text{ (using Hoeffding's inequality)} \end{aligned}$$

Note that $U_j'(i) - Ni/p$ lies strictly in the interval $[0, N/p]$, this fact is used in the application of the Hoeffding's inequality. On similar lines we have, $\sum_i (Ni/p - L_j'(i)) \leqslant \frac{2N}{s_j}$, *w.h.p.*. We then conclude,

$$|\gamma_j| \leqslant \sum_i \left(U'_j(i) - \frac{Ni}{p}\right) + \left(\frac{Ni}{p} - L'_j(i)\right) \leqslant \frac{4N}{s_j} \quad w.h.p. \qquad \square$$

The next lemma bounds the sample size for each round in terms of the sampling ratios.

Lemma 4.6. *Let $Z_j$ be the sample size for the $j^{th}$ round and $s_j \geqslant s_{j-1}$, then $Z_j \leqslant (5ps_j/s_{j-1})$ w.h.p.*

Proof. We have, $E[Z_j] = |\gamma_{j-1}|ps_j/N$. We also have, $|\gamma_{j-1}| \leqslant 4N/s_{j-1}$ *w.h.p.*, using Lemma 4.5.

Given that $|\gamma_{j-1}| \leqslant 4N/s_{j-1}$, using Chernoff bounds,

$$\begin{aligned}P[Z_j \geqslant (5ps_j/s_{j-1})] &\leqslant P[Z_j \geqslant E[Z_j] + ps_j/s_{j-1}] \\ &\leqslant e^{-\frac{(ps_j/s_{j-1})^2}{3E[Z_j]}} = e^{-\frac{(ps_j/s_{j-1})^2 N}{3|\gamma_{j-1}|ps_j}} \\ &\leqslant e^{-\frac{(ps_j/s_{j-1})^2 Ns_{j-1}}{12Nps_j}} \leqslant e^{-\frac{p}{12}} \qquad \square\end{aligned}$$

With Lemmas 4.6 and 4.3 in hand, we are now prepared to discuss how to appropriately choose the sampling ratios so that our algorithm achieves the desired load balance.

For HSS with $k$ rounds, if we set the sampling ratio for the $j^{th}$ round as $s_j = (2\ln p/\epsilon)^{j/k}$, then after $k$ rounds all splitters are found w.h.p., using Lemma 4.3. The size of the union of splitter intervals, that is, $|\gamma_j|$ is less than $4N/s_j = 4N(\epsilon/2\ln p)^{1/k}$ using Lemma 4.5. The sample size for the $j^{th}$ histogramming round is at most $5ps_j/s_{j-1} = 5p(2\ln p/\epsilon)^{1/k}$ using Lemma 4.6. This gives us our main theorem.

Theorem 4.7. *With k rounds of histogramming and a sample size of $O\left(p\sqrt[k]{\frac{\log p}{\epsilon}}\right)$ per round, HSS achieves $(1+\epsilon)$ load balance w.h.p. for large enough $p$*[2].

Observe from theorem 4.7 that there is a trade off between the sample size per round ($O(p\sqrt[k]{\log p/\epsilon})$) of histogramming and the number of histogramming rounds. To minimize the number of samples across all rounds, we take derivative of ($kp\sqrt[k]{\log p/\epsilon}$) w.r.t. $k$ and set it to 0,

$$\begin{aligned}\frac{d(kp\sqrt[k]{\log p/\epsilon})}{dk} &= p\sqrt[k]{\log p/\epsilon}\left(1 - \frac{\log\frac{\log p}{\epsilon}}{k}\right) = 0 \\ \Rightarrow \quad k &= \log\frac{\log p}{\epsilon}\end{aligned}$$

The overall sample size $O(kp\sqrt[k]{\log p/\epsilon})$ attains global minimum for $k = \log(\log p/\epsilon)$ histogramming rounds and $|\gamma_j| \leq 4N/(e)^j$ at the minima using Lemma 4.5. Across all rounds, the overall sample size from all processors is $O(p\log(\log p/\epsilon))$. This leads us to the following main theorem.

Theorem 4.8. *With $k = O(\log(\log p/\epsilon))$ rounds of histogramming and $O(p)$ samples per round ($O(1)$ from each processor), HSS achieves $(1+\epsilon)$ load balance w.h.p., for large enough $p$.*

Setting $\epsilon = p/N$ results in exact splitting and hence we get the following result for exact splitting.

[2]Specifically, so long as $s_j = \frac{2\ln p}{\epsilon} \leq \sqrt{\frac{2p}{\ln p}}$ for Lemma 4.5

Theorem 4.9. *HSS with $O(p)$ samples per round overall can achieve exact splitting in $O(\log N/p + \log\log p)$ rounds.*

## 5 RUNNING TIMES

We model an algorithm's parallel execution as a sequence of BSP supersteps, during each of which, processors perform computation locally, then send and receive messages. An algorithm's BSP complexity consists of three components:

(1) the number of supersteps (synchronization cost),
(2) the sum over all supersteps of the maximum amount of computation done by any processor during the superstep (computation cost),
(3) the sum over all supersteps of the maximum amount of data sent or received by any processor during the superstep (communication cost).

We permit processors to send and receive $p$ messages every superstep, which simplifies the analysis of the all-to-all data exchanges. The model captures the performance trade-offs for our purpose, more histogramming rounds increase number of supersteps but lower communication cost.

We analyse the computation and communication cost for sample sort and HSS. Both algorithms have the same cost for initial local sorting, broadcasting splitters and data exchange. The computation cost of local sorting is $O((N/p)\log\frac{N}{p})$. No communication is involved in local sorting. The cost of broadcasting splitters once they are determined is $O(p)$. The final data movement requires all data to be sent to their destination processors, hence the communication cost involved is $O(N/p)$. Once a processor receives all data pieces, it merges them, which takes $O((N/p)\log p)$ computation time.

### 5.1 Cost of Sampling

Collecting a sample of overall size $S$ onto one processor, requires a single BSP superstep with a communication cost of $O(S)$. In practice, random sampling is usually performed with each processor selecting $S/p$ elements, and a *gather* collective communication protocol, which collects all samples onto one processor. Sorting the overall sample on a central processor costs $O(S\log p)$ work locally if each processor provides a sorted contribution to the sample.

### 5.2 Cost of Histogramming

A local histogram can be computed in $O(S\log(N/p))$ time using $S$ binary searches on the local sorted input, where $S$ denotes the size of the histogram. A global histogram is computed by reducing all local histograms. An $S$-item reduction requires 2 BSP supersteps (one for a reduce-scatter and one for an all-gather) with $O(S)$ communication and computation [27, 31]. The histogram probes and the splitter intervals are broadcast to every processor for histogramming. The communication cost of broadcasting a length $S$ message is $O(S)$. Thus, both the computation and communication costs of histogramming are proportional to the overall sample size.

The BSP complexity of the data partitioning step of sample sort, AMS sort, and HSS are shown in Table 2. AMS sort and HSS require significantly fewer samples due to histogramming. We observe that the best HSS configuration has strictly superior complexity to all other considered algorithms.

| Algorithm | Overall sample size | Computation complexity | Communication complexity | Supersteps |
|---|---|---|---|---|
| Regular sampling | $O(\frac{p^2}{\epsilon})$ | $O\left(\frac{p^2}{\epsilon}\log p\log p\right)$ | $O\left(\frac{p^2}{\epsilon}\right)$ | $O(1)$ |
| Random sampling | $O(\frac{p\log N}{\epsilon^2})$ | $O\left(\frac{p\log N\log p}{\epsilon^2}\right)$ | $O\left(\frac{p\log N}{\epsilon^2}\right)$ | $O(1)$ |
| Single stage AMS sort | $O(p(\log p+\frac{1}{\epsilon}))$ | $O\left(p(\log p+\frac{1}{\epsilon})\log N\right)$ | $O\left(p(\log p+\frac{1}{\epsilon})\right)$ | $O(1)$ |
| HSS with one round | $O(\frac{p\log p}{\epsilon})$ | $O\left(\frac{p\log p}{\epsilon}\log N\right)$ | $O\left(\frac{p\log p}{\epsilon}\right)$ | $O(1)$ |
| HSS with two rounds | $O(p\sqrt{\frac{\log p}{\epsilon}})$ | $O\left(p\sqrt{\frac{\log p}{\epsilon}}\log N\right)$ | $O\left(p\sqrt{\frac{\log p}{\epsilon}}\right)$ | $O(1)$ |
| HSS with $k$ rounds | $O(kp\sqrt[k]{\frac{\log p}{\epsilon}})$ | $O\left(kp\sqrt[k]{\frac{\log p}{\epsilon}}\log N\right)$ | $O\left(kp\sqrt[k]{\frac{\log p}{\epsilon}}\right)$ | $O(k)$ |
| HSS with $O(1)$ samples per processor per round | $O(p\log\frac{\log p}{\epsilon})$ | $O\left(p\log\frac{\log p}{\epsilon}\log N\right)$ | $O\left(p\log\frac{\log p}{\epsilon}\right)$ | $O(\log\frac{\log p}{\epsilon})$ |

**Table 2: Cost complexity of data partitioning step of Sample sort, AMS sort, and HSS. Data exchange costs are excluded.**

| Algorithm | Sample size per stage | Computation complexity | Communication complexity | Supersteps |
|---|---|---|---|---|
| AMS sort, $l$ stages | $O(r(\log r+\frac{1}{\epsilon}))$ | $O\left(\frac{N}{p}\log N+lr(\log r+\frac{1}{\epsilon})\log N\right)$ | $O\left(lr(\log r+\frac{1}{\epsilon})+\frac{lN}{p}\right)$ | $O(l)$ |
| HSS, $l$ stages $O(\log\frac{\log r}{\epsilon})$ rounds per stage | $O(r\log\frac{\log r}{\epsilon})$ | $O\left(\frac{N}{p}\log N+lr\log(\frac{\log r}{\epsilon})\log N\right)$ | $O\left(lr\log\frac{\log(r)}{\epsilon}+\frac{lN}{p}\right)$ | $O(l\log\frac{\log p}{\epsilon})$ |

**Table 3: Cost complexity of $l$-stage HSS and AMS sort; the size of processor group decreases by a factor of $r = p^{1/l}$ each round.**

## 5.3 HSS with Multiple Stages

Like AMS-sort, HSS can be generalized to a multi-stage algorithm. We refer readers to [5] for details on multi-stage AMS sort. We simply consider the benefit of replacing the data-partitioning step in multi-stage AMS-sort with multiple histogramming rounds of HSS. The rest of the algorithm involving the data exchange steps is unchanged. The running time complexities of $l$ stage AMS-sort and $l$ stage HSS in the BSP model are shown in Table 3. In each step, a processor group gets divided into $r = p^{1/l}$ processor groups.

The first local sorting takes $O(N\log(N/p)/p)$ time. At the end of each stage, every processor receives $O(r)$ data pieces that it needs to merge. So, the total computation cost of local sorting after every stage excluding the first local sorting is $O((lN/p)\log r) = O((N/p)\log p)$. This gives an overall computation cost of local sorting as $O((N\log(N/p))/p + (N\log p)/p) = O((N\log N)/p)$. The computation cost of sampling and histogramming for HSS is $O(r\log((\log r)/\epsilon))\log N)$ per stage. Each sampling, histogramming and data exchange step takes $O(1)$ BSP supersteps. The number and cost of all of these steps is uniform throughout stages, so all of these costs are multiplied by a factor of $l$, the number of stages. Consequently, we observe a trade-off between the cost of data partitioning and the cost of data exchanges that depends on $l$.

## 6 IMPLEMENTATION

We implemented HSS in C++11 in the Charm++ [3, 7, 21] framework. Charm++ allows an application to create any number of virtual processors, called chares which are scheduled by the runtime system. Additionally, chares can be tied to a specific core or a node (called *group* and *nodegroup* chares in Charm++ terminoology).

Our implementation comprises of three phases; local sorting of input data, splitter determination using histogramming, and final data exchange. We use C++ `sort` for local sorting for the first phase. Let $t$ denote the total number of threads and $p$ denote the number of processors (processes or ranks).

- **Histogramming Phase**: The histogramming phase determines $p-1$ splitters for process level splitting. For the sampling phase before every histogramming round, each thread samples probe keys, from its input, which lie in the union of splitter intervals. If $\delta$ denotes the fraction of input covered by the splitter intervals, then every thread picks $s/\delta$ samples from its entire input and discards samples that don't lie in any of the splitter intervals. This way the expected size of the overall sample is $s\times t$, where $s$ can be thought of as the oversampling ratio. The overall sample is assembled at the central processor and broadcast for histogramming. Every thread computes a local histogram using binary searches on it's locally sorted input. The local histograms are summed up using a reduction and sent to the central processor.
- **Data Exchange**: Once every processor receives the splitters, input data from all threads within a processor are merged and partitioned into $p$ messages, one for each processor.
- **Local Merging**: Once a process receives all data that falls in its bucket, it merges and redistributes data among its threads using HSS with one round.

## 6.1 Handling duplicates via implicit tagging

We use the following standard technique to deal with duplicates in the inputs. We enforce a strict ordering on keys by implicitly replacing each key $k$ with a triplet $(k, \mathit{processor}, \mathit{ind})$, where *processor* denotes the processor that $k$ resides on and *ind* denotes the index in the local data structure, where the key is stored.

## 6.2 Multi-staged sorting and comparisons with other algorithms

To compare HSS to Hyksort in both single-staged and multi-staged settings, we implemented the splitting algorithm of HSS in the HykSort code [1], written in the MPI framework. This allows a fair comparison without the side-effects of two different parallel programming frameworks, namely Charm++ and MPI.

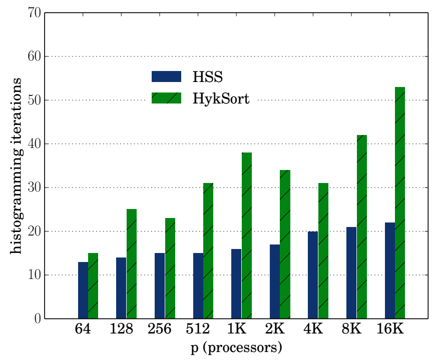

**Figure 2: Number of histogramming iterations for HSS vs HykSort. We used 1 sample per processor per round for this experiment. The worst case number of iterations increases more gradually in HSS ($O(\log(\log p/\epsilon))$) than Hyksort ($\Omega(\log(p/\epsilon)/\log\log p)$). Experiments were run on the Stampede 2 supercomputer.**

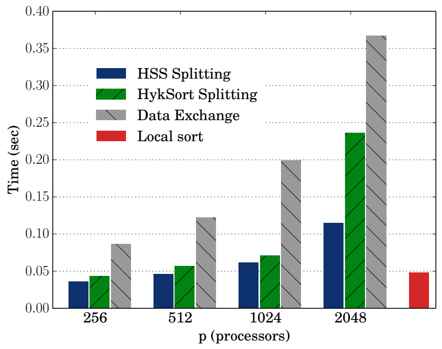

**Figure 3: Single staged runs with 16 threads per processor, 1M 8-byte long keys per processor. Experiments were run on the Stampede 2 supercomputer.**

## 7 EXPERIMENTAL RESULTS

In this section, we describe our experimental results. The goal of our experiments is to demonstrate the fast splitter determination of HSS compared to other state of the art algorithms and to demonstrate its benefits in both single-staged and multi-staged settings. We also include a brief application study to supplement our results.

### 7.1 Fast convergence of splitters

HSS determines all splitters in $O(\log(\log p)/\epsilon)$ rounds using $O(1)$ samples per round per processor. This results in faster convergence compared to HykSort, which requires $\Omega(\log(p/\epsilon)/\log\log p)$ rounds with the same number of keys. We ran the splitting algorithm of HSS and HykSort with 1 sample per processor per round and $\epsilon = 2\%$ to verify the same. As illustrated in Figure 2, the number of iterations in HSS increases gradually compared to HykSort. Note that the execution time of the splitting phase is directly proportional to the number of iterations.

### 7.2 Weak scaling and comparison to HykSort

In this section we describe single-staged experiments and comparisons to HykSort. We implemented HSS's splitting algorithm in HykSort's code for the fairest comparison. For this set of experiments we used 1 million keys per processor and 16 threads per processor. We used a probe size equal to 5p per histogramming round for both HSS and Hyksort which we found to be a reasonable sample size. We also found the default sample size of Hyksort, set as per [30] to be suboptimal for this set of experiments. Figure 3 illustrates weak scaling experiments. Besides the splitting time for the splitter determination step, the local sorting time and the data exchange times are also shown (which are common to both Hyksort and HSS). As can be observed from the figure, the difference between HSS and Hyksort's splitting phase becomes more apparent with increasing number of processors. The improved splitting of HSS results in a modest improvement of 10-15% in the overall running time for higher number of processors.

Single-staged AMS sort requires about $2p\ max(1/\epsilon, \log p)) \approx 100p$ for $p = 2048$ samples to achieve the desired splitting. In contrast, HSS took 6 iterations to converge with $5p$ samples per iteration resulting in about $30p$ samples overall. The execution time of the splitting phase is directly proportional to the number of samples, hence one can expect single-staged AMS to take approximately $3x$ time for the splitting phase.

### 7.3 Multi-staged experiments

In this section, we present multi-staged experiments (specifically with 2 stages) where data is first distributed among $k$ processor groups consisting of $p/k$ processors each for $k-way$ sorting. Multi-staged sorting is helpful when the number of message startups ($= p$ messages) per processor becomes a bottleneck. This happens for a large number of processors or when the number of keys per processor is small enough that very fine grained messages have to be sent to other processes in the data exchange step which slows down the sorting operation. Note that there is an overhead associated with multi-staged sorting as data needs to be moved multiple times in comparison to single-staged sorting where data is exchanged just once between the source and the destination processors.

For this set of experiments, we used $10^5$ keys per processor and 1 thread per processor. We used $k = 128$ as we found it to be a reasonable threshold for using 2-staged sorting. It also happens to be the recommended setting as per [30]. Note that $\lceil\sqrt{p}\,\rceil = 128$ for $p = 16384$. We used a tolerance threshold $\epsilon = 1\%$ per stage so that the overall imbalance is at most 2%. A single thread per process was used to maximize the number of processors. Accordingly, we also scaled down the number of keys/process when compared to section 7.2. This also kept the number of keys/thread comparable to section 7.2 (it is slightly higher in this case).

Figure 4 illustrates 2-staged runs for $p = 8192$ and $p = 16384$ processors. As can be seen from the figure, multiple stage sorting alleviates the data exchange bottleneck and the splitter determination step becomes the major bottleneck. Figure 4 demonstrates the benefit of using HSS in a multi-staged setting. HSS improves the overall running time by $15-20\%$ for both $p = 8192$ and $p = 16384$. We verified that this improvement comes from improving the number of iterations for convergence in each stage: HSS converged in 6 iterations while Hyksort took 9 iterations.

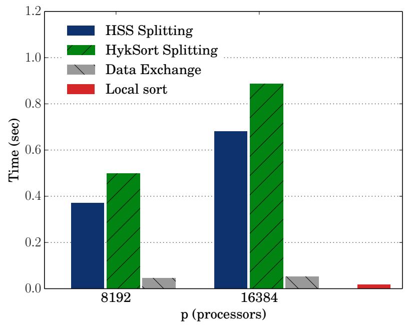

**Figure 4: 2-staged runs with 1 thread per processor, $10^5$ 8-byte long keys per processor. Experiments were run on the Stampede 2 supercomputer.**

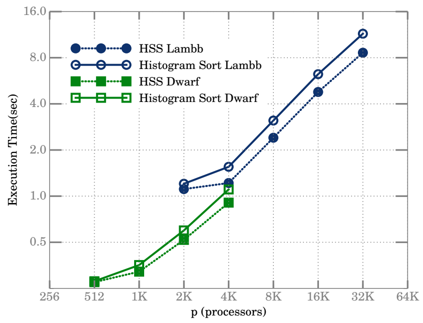

**Figure 5: Experiments showing performance of sorting routine of Changa. Datasets used were Lambb and Dwarf. Experiments were run on the Bluegene Mira supercomputer.**

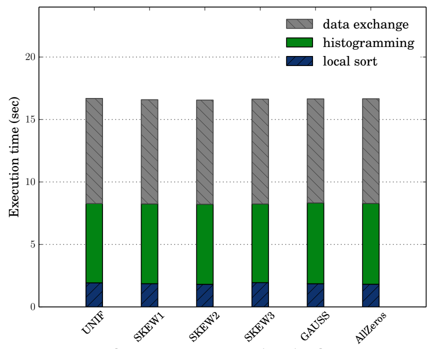

**Figure 6: HSS performance with different input distributions for 2M keys/processor and 32K processors. We used 1 thread per processor to accentuate the splitter selection (histogramming) time. Experiments were run on the Bluegene Mira supercomputer.**

## 7.4 Strong scaling in ChaNGa

We implemented HSS in ChaNGa [20], a popular astronomical application that often runs on several thousands of processors. Sorting in ChaNGa poses unique challenges for two reasons- (i) it employs virtual processors and hence the number of buckets (virtual processors) are far more than the actual number of processors. In our experiments, the number of virtual buckets were typically **10x** the number of cores and (ii) the virtual processors can be arbitrarily placed across physical nodes and buckets on a single node need not be contiguous. Hence, most of our shared memory optimizations are not useful. The reason ChaNGa uses more virtual processors than cores is to accelerate other stages of computation, made possible by efficient parallel data overpartitioning.

Figure 5 compares sorting performance of ChaNGa with HSS and the existing Histogram sort implementation for two datasets: Dwarf and Lambb (see [20] for details). The datasets have a constant number of keys, so Figure 5 represents strong scaling results. HSS results in up to 25% improvement over Histogram sort. Note that Histogram sort is much more sensitive to the input distribution than HSS as it does not employ sampling (see section 7.5). The parallel sorting execution increases for the same dataset as we increase the number of processors. This may appear odd at first. The majority of sorting time is spent in data splitting, and since the number of buckets increase multiplicatively with the number of processors, we see an increase in the execution time. The performance results suggest it would be possible to improve strong scaling of the splitting algorithm within ChaNGa by using a multi-staged version of HSS. We leave this for future work.

## 7.5 Effect of input distribution

We ran HSS with the following input distributions to verify that its running time is independent of the distribution:

(1) UNIF: Uniformly at random from the entire range
(2) SKEW1: Half of the keys are picked uniformly at random from the entire range, the other half, uniformly at random from a small range of size 1000
(3) SKEW2: Uniformly at random from the range [0, 100]
(4) SKEW3: Each key is bitwise and of two uniformly at random chosen keys
(5) GAUSS: Gaussian distributed
(6) AllZeros: All keys are set to 0

As figure 6 illustrates, HSS is impervious to the input distribution as expected from the analysis. To underscore the histogramming cost, we used 1 thread per process since the number of splitters for process level splitting is equal to the number of processes.

## 8 CONCLUSION

We presented Histogram sort with sampling (HSS), which combines sampling and histogramming to accomplish fast splitter determination. We showed that for approximate-splitting ($\epsilon = O(1)$), our algorithm requires $\Theta(\log\log p)$ histogramming rounds and an overall sample of size $\Theta(p \log\log p)$, improving the communication cost by a factor of $\Theta(\log p / \log\log p)$ with respect to the best known partitioning algorithm. HSS is theoretically more efficient for both approximate and exact (memory-efficient) splitting, while minimizing the number of data exchanges for both small and large degrees of parallelism. Our work provides theoretical groundwork for the benefits of iterative histogramming in splitter selection, a technique that is known to work well in practice. The reduced sample size makes HSS extremely practical for massively parallel applications, scaling to tens of thousands of processors. We demonstrated speed-ups with HSS over two other state-of-the-art parallel sorting implementations for both single-staged and multi-staged settings. The robustness of our results makes a compelling case for HSS as the algorithm of choice for large scale parallel sorting.

## ACKNOWLEDGMENTS

The authors are thankful to Omkar Thakoor, Umang Mathur and Harshita Prabha for helpful discussions and Nitin Bhat for helping with experiments. The authors acknowledge the Argonne Leadership Computing Facility (ALCF) and the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. URL: https://www.alcf.anl.gov/ and http://www.tacc.utexas.edu

## REFERENCES

[1] 2013. Hyksort code. https://github.com/hsundar/usort. (2013).

[2] 2019. HSS code. https://github.com/vipulharsh/hss. (2019).

[3] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, et al. 2014. Parallel programming with migratable objects: charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 647–658.

[4] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An O(n log n) sorting network. In *Proceedings of the 15th annual ACM Symposium on Theory of computing*. ACM, 1–9.

[5] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. 2015. Practical massively parallel sorting. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 13–23.

[6] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 307–314.

[7] Abhinav Bhatele. 2014. Chapter 5.2 Projections: Scalable Performance Analysis and Visualization. *Technical Report: Connecting Performance Analysis and Visualization to Advance Extreme Scale Computing, Lawrence Livermore National Laboratory* (2014), 33.

[8] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. 2015. Sorting with Asymmetric Read and Write Costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, New York, NY, USA, 1–12.

[9] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low Depth Cache-oblivious Algorithms. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 189–199.

[10] Guy E Blelloch, Charles E Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith, and Marco Zagha. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM Symposium on Parallel algorithms and architectures*. ACM, 3–16.

[11] Guy E. Blelloch, Charles E. Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith, and Marco Zagha. 1998. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems* 31, 2 (1998), 135–167.

[12] David R Cheng, Alan Edelman, John R Gilbert, and Viral Shah. 2006. A novel parallel sorting algorithm for contemporary architectures. (2006).

[13] Richard Cole. 1988. Note on the AKS sorting network. *Computer Science Department Technical Report* 243. New York University, New York, (1988).

[14] Richard Cole. 1988. Parallel merge sort. *SIAM J. Comput.* 17, 4 (1988), 770–785.

[15] W Donald Frazer and AC McKellar. 1970. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)* 17, 3 (1970), 496–507.

[16] Michael T Goodrich. 1999. Communication-efficient parallel sorting. *SIAM J. Comput.* 29, 2 (1999), 416–432.

[17] Vipul Harsh, Laxmikant V. Kalé, and Edgar Solomonik. 2018. Histogram Sort with Sampling. *CoRR* abs/1803.01237 (2018). arXiv:1803.01237

[18] David R Helman, Joseph JáJá, and David A Bader. 1998. A new deterministic parallel sorting algorithm with an experimental evaluation. *Journal of Experimental Algorithmics (JEA)* 3 (1998), 4.

[19] JS Huang and YC Chow. 1983. Parallel sorting and data partitioning by sampling. (1983).

[20] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V Kale, and Thomas Quinn. 2008. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1–12.

[21] Laxmikant V Kale and Sanjeev Krishnan. 1993. *CHARM++: a portable concurrent object oriented system based on C++*. Vol. 28. ACM.

[22] Laxmikant V Kale and Sanjeev Krishnan. 1993. A comparison based parallel sorting algorithm. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Vol. 3. IEEE, 196–200.

[23] Hui Li and Kenneth C Sevcik. 1994. Parallel sorting by over partitioning. In *Proceedings of the sixth annual Symposium on Parallel algorithms and architectures (SPAA)*. ACM, 46–56.

[24] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. 1993. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Comput.* 19, 10 (Oct. 1993), 1079–1103.

[25] Owen O'Malley. 2008. Terabyte sort on apache hadoop. *Yahoo, http://sortbenchmark.org/YahooHadoop.pdf* (2008), 1–3.

[26] M. S. Paterson. 1990. Improved sorting networks with O(logN) depth. *Algorithmica* 5, 1 (1990), 75–92.

[27] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. 2007. Performance analysis of MPI collective operations. *Cluster Computing* 10, 2 (2007), 127–143.

[28] Hanmao Shi and Jonathan Schaeffer. 1992. Parallel Sorting by Regular Sampling. *J. Parallel and Distrib. Comput.* 14, 4 (April 1992), 361–372.

[29] E. Solomonik and L. V. Kale. 2010. Highly scalable parallel sorting. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 1–12.

[30] Hari Sundar, Dhairya Malhotra, and George Biros. 2013. HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 293–302.

[31] Rajeev Thakur and William D Gropp. 2003. Improving the performance of collective operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 257–267.

[32] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111.

## A HYKSORT SAMPLING ALGORITHM: ANALYSIS

HykSort [30] selects $O(\beta)$ samples from every splitter interval in every round, thus resulting in an overall sample size $O(\beta p)$. In contrast, HSS picks samples uniformly from the union of all splitter intervals, also resulting in an overall sample size $O(\beta p)$. Effectively, sampling in HSS from a splitter interval is proportional to the size of the interval. We prove that HykSort requires at least $\Omega\left(\frac{\log(p)}{\log\log(p)}\right)$ rounds, so that all splitters are within a distance of $N\epsilon/p$ from the ideal splitters.

Our proof strategy is the following. First of all we reduce the problem by using $\beta = 1$. Sampling $\beta$ samples per round can bring down the number of rounds by at most a factor of $\beta$. Since we're only interested in the dependence of $p$ on the number of rounds, it suffices to show that HykSort requires at least $\Omega\left(\frac{\log(p)}{\log\log(p)}\right)$ rounds with $\beta = 1$.

Secondly, we assume a better starting point for the splitter intervals. More specifically, we assume that the initial $i^{th}$ splitter interval is given to be $[Ni/p - N/2p, Ni/p + N/2p]$, instead of the entire range $[0, N]$. Starting with a narrowed splitter interval will only decrease the number of rounds. This eases the analysis since effectively each splitter interval is being independently sampled and the number of rounds should be enough so that for all $i$, at least one key is sampled that is within the target range $\mathcal{T}_i = [Ni/p - N\epsilon/p, Ni/p + N\epsilon/p]$.

From here on, we can work with just one interval and determine the number of rounds required so that at least one key is chosen in the target range $\mathcal{T}_i = [Ni/p - N\epsilon/p, Ni/p + N\epsilon/p]$ with probability $\geq 1 - 1/p$. Note that probability $\geq 1 - 1/p$ is required to use the union bound to bound the probability of not finding a sample in the target range for any of the splitter intervals (there are $p - 1$ splitters to be determined).

HykSort's sampling algorithm is as follows. In round $r$, it samples one key $k$ (recall that we assumed $\beta = 1$) in the splitter interval $[L_r(i), U_r(i)]$ and then updating the splitter interval as

$$U_{r+1}(i) = min(U_r(i), k)$$

$$L_{r+1}(i) = max(L_r(i), k)$$

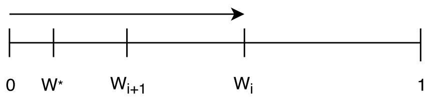

**Figure 7: Line Algorithm : $w_{i+1}$ is picked uniformly at random from** $[0, w_i)$

As discussed earlier, the initial interval is

$$[L_0(i), U_0(i)] = \left[\frac{Ni}{p} - \frac{N}{2p}, \frac{Ni}{p} + \frac{N}{2p}\right]$$

In the following section we prove that it takes $r = \Omega\left(\frac{\log(p)}{\log\log(p)}\right)$ rounds such that $P[U_r(i) \leq Ni/p + N\epsilon/p]$ with probability $\geq 1 - 1/p$.

By basically the same argument it can be shown that it takes, $r = \Omega\left(\frac{\log(p)}{\log\log(p)}\right)$ rounds such that $P[L_r(i) \geq Ni/p - N\epsilon/p]$ with probability $\geq 1 - 1/p$.

## A.1 The line algorithm

Consider the following algorithm.

Line Algorithm: Pick a point $w_0$ uniformly at random in the real interval $[0, 1)$. In the next round pick a point $w_1$ uniformly at random in the real interval $[0, w_0)$. Similarily, in the i$^{\text{th}}$ round pick a random point $w_i$ in the interval $[0, w_{i-1})$ and so on. The line algorithm captures the sampling algorithm of HykSort.

Given a point $w^* \in [0, 1)$ and probability bound $p^*$, we wish to bound the number of rounds $r$ so that $P[w_r > w^*] < p^*$. For the analysis of HykSort, we'll set $w^* = \frac{N\epsilon/p}{N/p} = \epsilon$ and $p^* = 1/p$. We prove the following lemma.

Lemma A.1. *For a given $w^* \in [0, 1)$, the number of rounds $r$ after which $P[w_r > w^*] < p^*$ is $\Omega\left(\frac{\log(1/p^*)}{\log\log(1/p^*)}\right)$.*

Proof. By definition of the line algorithm,

$$0 \leq w_{i+1} \leq w_i$$

Let $f^i(x)$ be the probability density function of $w_i$. We have the following recurrence for $f^i(x)$,

$$f^i(x) = \int_x^1 \frac{f^{i-1}(y)}{y} dy \quad \forall i \geq 1$$

The expression inside the integral represents the probability $P\Big[w_i \in [x, x + dx] \,\Big|\, w_{i-1} \in [y, y + dy]\Big] P\Big[w_{i-1} \in [y, y + dy]\Big]$. We have $f^0(x) = 1$. Using induction on $i$, it can be easily seen that

$$f^i(x) = \frac{\log^i(\frac{1}{x})}{i!}$$

We can also obtain the corresponding cumulative density functions $F^i(x)$,

$$\begin{aligned} F^i(x) &= P[w_i \leq x] \\ &= \int_0^x f^i(y) dy \\ &= x \sum_{k=0}^{i} \frac{\log^k\left(\frac{1}{x}\right)}{k!} \end{aligned}$$

It can be verified that $\lim_{i\to\infty} F^i(x) = xe^{\log(\frac{1}{x})} = 1$, using Taylor's expansion for the exponential function. We have

$$\begin{aligned} P[w_r > w^*] &= 1 - F^r(w^*) \\ &= 1 - w^* \sum_{k=0}^{r} \frac{\log^k\left(\frac{1}{w^*}\right)}{k!} \\ &= \left(1 - e^t \sum_{k=0}^{r} \frac{t^k}{k!}\right), \text{ where } t = \log\frac{1}{w^*} \\ &= e^{-t}\left(e^t - \sum_{k=0}^{r} \frac{t^k}{k!}\right) \\ &= e^{-t}\left(\frac{e^{\xi} t^{r+1}}{(r+1)!}\right) \text{ for some } \xi \in [0, t] \end{aligned}$$

The last deduction is based on the error term in the Taylor expansion. We want the error term to be smaller than $p^*$. Hence we have,

$$\begin{aligned} & e^{-t}\left(\frac{e^{\xi} t^{r+1}}{(r+1)!}\right) \leq p^* \\ \Rightarrow\quad & \frac{(r+1)!}{t^{r+1}} \geq \frac{1}{p^* e^{t-\xi}} \\ \Rightarrow\quad & \log(r+1)! - (r+1)\log t \geq \log\frac{1}{p^*} - (t - \xi) \\ \Rightarrow\quad & (r+1)\log(r+1) - (r+1) + O(r) \geq \log\frac{1}{p^*} - (t - \xi) \end{aligned}$$

The last deduction is using Stirling's formula:

$$\log n! = n \log n - n + O(\log n)$$

Note that the dominating term in LHS is $(r + 1)\log(r + 1)$ as $t$ is a constant. Thus we obtain

$$r = \Omega\left(\frac{\log(1/p^*)}{\log\log(1/p^*)}\right)$$

□