Histogram Sort with Sampling

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

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

Abstract

Standard parallel sorting algorithms rely on data partitioning techniques like sampling and histogramming to distribute keys across processors. The sampling cost in sample sort for good load balance is prohibitive for massive clusters. We describe Histogram sort with sampling (HSS), an adaptation of the popular Histogram sort algorithm. We present the first comprehensive analysis of histogramming as a technique and show that HSS has sound theoretical guarantees. HSS reduces the sample size requirements from $O(p \log N/\epsilon^2)$ to $O(kp \sqrt[k]{\log p/\epsilon})$ with k rounds of histogramming w.h.p.[1]. HSS is more efficient than Sample sort algorithms that achieve the same level of load balance, both in theory and practice, especially for massively parallel applications, scaling to tens of thousands of PEs[2]. We also show that an approximate but fairly accurate histogram can be obtained using a $O(\sqrt{p \log N}/\epsilon)$ sample on every processor. This can be used to speed up the histogramming step and can be of independent interest for answering general queries in large parallel processing systems. In our practical implementation, we exploit shared memory within nodes to improve the performance of HSS on large modern clusters.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords parallel sorting; data partitioning; histogramming; histogram sort; sample sort;

ACM Reference Format:
Vipul Harsh and Laxmikant Kale. 2018. Histogram Sort with Sampling. In *Proceedings of , , ,* 14 pages.
https://doi.org/

1 Introduction

Scalability, load balance and performance are major challenges of parallel sorting. Load balance is crucial for many parallel applications, because an overloaded processor slows down the entire application. For this reason, *ChaNGa* [15], an N-body application to perform collisionless N-body cosmological simulations, which uses parallel sorting at the beginning of every iteration in its simulation, has a strict requirement for good load balance.

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 greater than keys on processor $k-1$ and keys are sorted within each processor. Different parallel sorting algorithms have different guarantees on the load balance after sorting. We assume that the application specifies the load balancing parameter ϵ to indicate that every processor should end up with no more than $N(1+\epsilon)/p$ keys,[3] This model closely captures many real world systems as different applications have varying tolerance for load imbalance.

Several parallel sorting algorithms rely on data partitioning techniques to determine $p-1$ splitter keys that partition the data range into p buckets, one for each processor. The splitters are broadcast to all processors and keys are sent to their destination processors, determined by the bucket it falls in. Two popular algorithms in this regard are Sample sort and Histogram sort. HSS is partly inspired from both of these algorithms.

Sample sort [11] is a popular parallel sorting algorithm that relies on data partitioning. Sample sort with regular sampling [21, 25] collects $\Theta(p^2/\epsilon)$ sampled keys from all processors at a central processor, which then decides the splitters in a way that guarantees that every processor will have no more than $N(1+\epsilon)/p$ keys at the end of the algorithm. Sample sort with random sampling as proposed by Blelloch et al. [7] requires $\Theta(p \log N/\epsilon^2)$ samples overall across all processors to achieve the same load balance. Both of these are unscalable for large p and a reasonable value of ϵ, because of the high cost of sampling. Nevertheless, because of its simplicity, sample sort is widely deployed in modestly large parallel processing systems [22]. We discuss Sample sort with both regular and random sampling in Section 4.1.

Histogram sort as proposed by Kale et al. [17, 26] conducts multiple rounds of histogramming to determine "good" splitter keys, refining candidate splitter keys every round. The number of histogramming rounds required to determine all splitters within the allowed threshold is loosely bounded by $\log \mathcal{N}$, where $\mathcal{N}$ is the range of the input (i.e. maximum key minus the minimum key), although in practice it takes

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

[2] processing elements, typically threads

[3] More precisely, we define load imbalance to be the ratio of maximum load and average load

, ,
2018. ACM ISBN ...$15.00
https://doi.org/

fewer rounds [17, 26] for many distributions. Nevertheless, for skewed distributions the number of rounds could be large.

In this paper, we propose Histogram sort with sampling (HSS), an adaptation of Histogram sort and a highly scalable and practical parallel sorting algorithm that relies on data partitioning via histogramming. We provide a comprehensive and thorough study of HSS. By characterizing HSS, we establish the theoretical soundness of histogramming as a technique, that has been known to be effective [3, 16, 26, 27] in practice. To the best of our knowledge, we are the first to provide such a comprehensive analysis. HSS accomplishes fast splitter determination by performing histogramming rounds on a significantly smaller sample as compared to sample sort. After one round of sampling followed by histogramming on $O(p \log p/\epsilon)$ samples, it determines all the splitters w.h.p.. The splitters determined this way achieve load balance levels specified by ϵ. We describe HSS with one round of histogramming in Section 3.1. We show that the sample size can be further brought down by repeating the above idea for multiple rounds, that is, by repeated rounds of sampling followed by histogramming. In general, with k rounds, the required sample size is $O(p\sqrt[k]{\log p/\epsilon})$ every round, for an overall sample size of $O(kp\sqrt[k]{\log p/\epsilon})$ across all rounds. The details for the general case of k rounds are described in Section 3.3. Lemmas 3.3, 3.8 and 3.9 outline the main findings of this paper.

We focus on reducing the sample size because, as we show in section 5, the cost of determining splitters, including the cost of histogramming, is proportional to the sample size. To see the impact of reduced sample size on scalability, consider $p = 64 \times 10^3$, $\epsilon = 0.05$, $N/p = 10^6$ and 64 bit keys. The required sample size is 655 GB for sample sort with regular sampling and 5 GB for Sample sort with random sampling. In contrast, it is 250 MB and 22 MB for HSS with one round and two rounds, respectively.

In section 3.4, we propose an enhancement to speed up the histogramming step. Every processor maintains a $O(\sqrt{p \log N}/\epsilon)$ representative sample of its own local input and uses this sample, instead of the entire input, to execute the histogramming step. We show that the histogram, thus obtained, is accurate enough for the purpose of our algorithm. This method can also be of independent interest in answering repeated rank queries in general parallel processing systems.

In section 5, we compare the running time complexity of HSS and sample sort. Finally, in Section 6, we demonstrate the scalability of HSS on large supercomputing clusters. We leverage shared memory within nodes for better performance and scalability.

2 Preliminaries

2.1 Problem Statement

We assume the input A to consist of N keys across p processing elements, distributed evenly such that each processor has N/p keys. For simplicity, we assume that there are no duplicates in the input. In Section 6.2, we describe how to deal with duplicates in the input. A parallel sorting algorithm needs to redistribute the input to obtain a global sorted order. Additionally for good load balance, each processor should end up with no more than $N(1 + \epsilon)/p$ keys after sorting, where ϵ is specified by the application. Cheng et al. [8] propose an algorithm that finds exact splitters that achieve perfect load balance with $O(p \log N)$ rounds of communication. Such an algorithm is largely of theoretical interest as we are not aware of any practical applications that have such a stringent requirement of load balance.

The bulk of this paper focuses on data partitioning algorithms. Sample sort by regular sampling [21, 25], histogram sort [17, 26], sample sort by random sampling [7, 11] and parallel sorting by over partitioning [20] fall into this category. A data partitioning algorithm determines $p - 1$ splitter keys, that split the input into p ranges, one for each processor. Let the sorted sequence of splitter keys determined by a data partitioning algorithm be $\mathcal{S} = \{S_1, S_2 ..., S_{p-1}\}$. Once the algorithm finishes, p_i has all keys in range $[S_i, S_{i+1})$, where we define $S_0 = Min\_Key$ and $S_p = Max\_Key$ - the minimum and maximum key value. For numeric keys, we can define them to be $-\infty$ and ∞, respectively.

For good load balance, any algorithm should find "good" splitters that partitions the data evenly. With ideal load balance, every processor ends up with exactly N/p keys. If $R(k)$ denotes the rank of key k in the overall input, then in such an ideal case, $R(S_i) = Ni/p$, or equivalently if $I(r)$ denotes the key with rank r in the overall input, then, $S_i = I(Ni/p)$. Since some load imbalance is usually tolerable, we enforce the following conservative condition for splitters:

$$S_i \in \mathcal{T}_i, \text{ where, } \mathcal{T}_i = \left[I\Big(\frac{Ni}{p} - \frac{N\epsilon}{2p}\Big), I\Big(\frac{Ni}{p} + \frac{N\epsilon}{2p}\Big) \right]$$

Consequently, every processor will end up with no more than $N(1 + \epsilon)/p$ keys, as required by the problem. We say that splitter i is **finalized** if a key k has been found that is known to be from $\in \mathcal{T}_i$, i.e., if we have found a suitable candidate key k for S_i.

Our proofs do not rely on input keys to be evenly distributed across each processor. Our algorithm can be easily adapted for uneven divisions of input by adjusting the sample size from every processor, based on the length of the local input. The running time, however, will suffer because of load imbalance in the input. We note that any parallel sorting algorithm will be affected by load imbalance in the input. We evaluate our algorithm using the bulk synchronous parallel model (BSP) suggested by Valiant [29]. Comparison of execution times with sample sort suggest that HSS is theoretically more efficient than parallel sample sort (see section 5).

Since our algorithm borrows some ideas from sample sort [11] and histogram sort [17], we give their high level

description before proceeding to our main result. We review these and other parallel sorting algorithms in more detail in section 4.

2.2 Sample sort

Sample sort [6, 11, 14, 21, 25] is a widely studied and analyzed parallel sorting algorithm. Sample sort samples s keys from each processor in some fashion, 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. $p - 1$ keys are chosen from Λ as the final splitters. Any algorithm for sample sort has the following skeletal structure, consisting of three phases.

1. **Sampling Phase**: Every processor samples s keys and sends it to a central processor. s is often referred to as the oversampling ratio. Much research has taken place on how to select samples for better load balance. We discuss different sampling methods in section 4.1.
2. **Splitter determination**: The central processor receives samples of size s (obtained in Step 1) from every processor resulting in a combined sample Λ of size (ps). The central processor then selects $p - 1$ splitter keys: $\mathcal{S} = \{S_1, S_2 ..., S_{p-1}\}$ from Λ that partitions the key range into p ranges, each range assigned to one processor. Usually, the splitters are chosen by picking evenly spaced keys from Λ. Once the splitters have been finalized, they are broadcast to all processors.
3. **Data movement**: Once a processor receives the splitter keys, it sends each of its key to the appropriate 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 the input data onto their assigned destination processors. Once a processor receives all data that falls in its bucket, it merges them using a sequential algorithm, like merge sort.

Although sample sort requires a large number of samples to provably determine splitter keys that achieve the desired level of load balance, and may not be scalable in practice for a large number of processors, it is popular for its simplicity. In practice, it achieves good load balance with fewer samples for well behaved input distributions.

2.3 Histogram Sort

Histogram sort [17, 26] addresses load imbalance by determining the splitters more accurately. 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 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 keys. Once all the splitters are within the given threshold, it finalizes the splitter keys from the set of candidate keys. The rest of the algorithm involving data movement is identical to histogram 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 since no other 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. All local histograms are summed up using a reduction to obtain the global histogram at the central processor.
4. The central processor finalizes and broadcasts the splitters if all splitters have been finalized. Otherwise, it refines its probes using the histogram obtained and broadcasts a new set of probes for next round of histogramming, in which case the algorithm loops back to step 2.

Histogram sort is guaranteed to achieve any arbitrary specified level of load balance. It is also scalable 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 loosely bounded by $\log \mathcal{N}$, where $\mathcal{N}$ is the range of the input (i.e. maximum key minus the minimum key). The number of rounds can be large, especially for skewed input distributions.

Nevertheless, Histogram sort is reliable for good load balance and more scalable than sample sort for many practical scenarios. It has been successfully deployed in real world, highly parallel scientific applications, for instance *ChaNGa* [15].

3 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 every processor samples some s 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, to achieve the desired load balance. We argue later in this paper that the cost of histogramming isn't a big overhead, both theoretically and in practice (see section 5 and section 6).

For the sampling phases, our algorithm chooses a sample from a subset $\mathcal{G}$ of the input. Initially, $\mathcal{G}$ represents the entire input. As the algorithm progresses, $\mathcal{G}$ gets smaller. We use the following method for sampling throughout our discussions, unless stated otherwise.

Every key in $\mathcal{G}$ is independently chosen to be a part of the sample with probability ps/N, where we refer s as the **sampling ratio**.

Note that the size of the overall sample collected from all processors with the above method is $(ps|\mathcal{G}|/N)$ in expectation.

3.1 One round of histogramming

We show that the sample size in sample sort can be reduced by an order of magnitude by performing just one round of histogramming on the collected sample. The splitters obtained after the first histogramming round achieve the specified level of load balance w.h.p..

3.2 Scanning algorithm

We describe the scanning algorithm proposed by Axtmann et. al. [3] to decide the splitters, once the histogram is obtained. The algorithm scans through the histogram and assigns bucket between two keys to a PE. It skips to the next PE when the total load on that PE exceeds $N(1+\epsilon)/p$. The last PE gets all the remainder elements. We should sample enough so that the remainder elements is less than $N/p(1+\epsilon)$. We reproduce the following result from [3]. and also provide a full proof for the same

Theorem 3.1. *If every key is independently picked in the sample with probability, $\frac{ps}{N} = \frac{2p}{\epsilon N}$, where s, the sampling ratio is chosen to be $2/\epsilon$, then the number of elements assigned to the last processor by the scanning algorithm is $\leq \frac{N(1+\epsilon)}{p}$ w.h.p.*

Proof: Let the number of elements assigned to the i^{th} processor by the scanning algorithm be n_i. Clearly, $n_i \leq \frac{N(1+\epsilon)}{p}$ for all $i < p-1$. The last processor however, may have more elements. Ideally, with perfect load balance, the load on every processor should be N/p. Define $r_i = \left(N/p + N\epsilon/p - n_i\right)$. By design of the scanning algorithm, $r_i \geq 0 \ \forall \ i \in [0, p-1)$.

We have,

$$\begin{aligned} n_{p-1} &= N - \sum_{i=0}^{p-2} n_i \\ &= N - \sum_{i=0}^{p-2} \left(\frac{N}{p} + \frac{N\epsilon}{p} - r_i\right) \\ &= \frac{N}{p} - \frac{N\epsilon(p-1)}{p} + \sum_{i=0}^{p-2} r_i \end{aligned}$$

Since all samples are picked independently using binomial trials, r_i is exponentially distributed. More specifically, $P\left[r_i \geq k\right] = (1 - ps/N)^k$. Thus, we have,

$$E\left[r_i\right] \leq \sum_{k=1}^{\infty} \left(1 - \frac{ps}{N}\right)^k \leq \frac{N}{ps}$$

Since all samples are picked independently, the random variables r_i are also mutually independent. With sampling ratio $s = \frac{2}{\epsilon}$ and using expression for n_{p-1}, we have,

$$\begin{aligned} P\left[n_{p-1} \geq \frac{N}{p} + \frac{N\epsilon}{p}\right] &= P\left[\sum_{i=0}^{p-2} r_i \geq N\epsilon\right] \\ &\leq P\left[\sum_{i=0}^{p-2} r_i \geq \frac{N\epsilon}{2} + \frac{N\epsilon(p-1)}{2p}\right] \\ &\leq P\left[\left(\sum_{i=0}^{p-2} r_i - E[\sum_{i=0}^{p-2} r_i]\right) \geq \frac{N\epsilon}{2}\right] \\ &\leq e^{\frac{-p\epsilon^2}{2(1+\epsilon)^2}} \quad \square \end{aligned}$$

The last inequality is obtained using an application of Hoeff ding's inequality since all r_i's are independent.

Next, we describe HSS with one round of histogramming, a slightly worse algorithm than scanning algorithm in terms of the data partitioning step. We emphasize that if one were to use just one round of histogramming, the scanning algorithm does better and should be used over HSS with one round. However, extending the scanning algorithm for multiple rounds is non-trivial and unclear. HSS with one round is easily generalizable to multiple rounds of histogramming, as we discuss in subsequent sections. With multiple rounds of histogramming, HSS is better than the scanning algorithm.

Recall that in HSS, splitter i is finalized when the algorithm finds a candidate key that is known to be in $\mathcal{T}_i$. If the sample contains at least one key from $\mathcal{T}_i$, then after histogramming on the sample, all splitters will be finalized. 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..

Theorem 3.2. *If every key is independently picked in the sample with probability, $\frac{ps}{N} = \frac{2p \ln p}{\epsilon N}$, where s, the sampling ratio is chosen to be $\frac{2 \ln p}{\epsilon}$, then at least one key is chosen from each $\mathcal{T}_i$ w.h.p.*

Proof: Recall that the input set is denoted by A. The size of $\mathcal{T}_i \cap A$, $|\mathcal{T}_i \cap A| = N\epsilon/p$. The probability that no key is chosen from $\mathcal{T}_i$ in the overall sample is given by,

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

Finally, since there are $p-1$ splitters, the probability that no key is chosen from some $\mathcal{T}_i$, is at most $(p-1) \times p^{-2} < 1/p$. $\square$ This leads us to the following lemma for HSS for one round.

Lemma 3.3. *With one round of histogramming and $O\left(\frac{p \log p}{\epsilon}\right)$ sized sample, HSS achieves $(1+\epsilon)$ load balance w.h.p.*

3.3 Multiple rounds of histogramming

We show that the sample size can be further reduced by repeated rounds of sampling followed by histogramming.

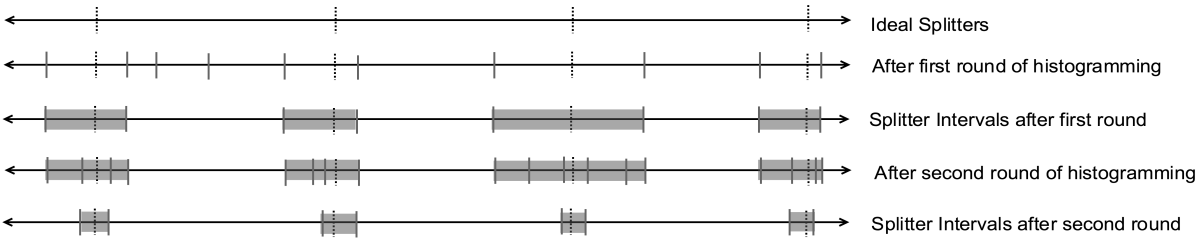

Figure 1. Figure illustrating HSS with multiple rounds. After first round, samples are picked only from the splitter intervals. Notice how the splitter intervals shrink as the algorithm progresses.

Our algorithm builds 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. We first describe the steps of our histogramming algorithm in terms of the sampling ratios s_j's for each round. Later, in our analysis, we describe how to appropriately set these parameters to achieve the desired load balance after k rounds.

1. In the sampling phase before the first round of histogramming, each key in the input is picked in the sample with probability (ps_1/N), where s_1 is the sampling ratio for the first round. Samples from all processors are collected at a central processor and broadcast as probes for the first round of histogramming.
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 intervals $\mathcal{I}_j(i) = [I(L_j(i)), I(U_j(i))]$ for the next round. We refer to these intervals as **splitter intervals**.
4. Once every processor receives the splitter intervals $\mathcal{I}_j$'s, 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_t denotes the sampling ratio for the t^{th} round. Keys which don't fall in any of the $\mathcal{I}_j$'s are not picked. 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 and 3 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 finalized for the i^{th} splitter. In further discussions, we discuss how to choose k and the sampling ratios s_j's so that the splitters determined this way achieve the desired load balance.

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 $\mathcal{G}_j$ denote the number of keys in the input that belong to one of the splitter intervals after j rounds. $\mathcal{G}_j$ represents the size of the input that the algorithm samples from, for the j^{th} round. We have, $\mathcal{G}_j \leqslant \sum_i |\mathcal{I}_j(i) \cap A|$, where $|\mathcal{I}_j(i) \cap A|$ denotes the number of input keys that fall in $\mathcal{I}_j(i)$. Some splitter intervals can overlap, hence the inequality . In fact, it is easy to reason out 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.

For ease of understanding, we break down the analysis into a sequence of theorems that shed more light on the algorithm. Theorems 3.4 and 3.5 bound $\mathcal{G}_j$ in terms of the sampling ratios, Theorem 3.6 bounds the sample size in terms of the sampling ratios and Theorem 3.7 provides a way to bound the number of rounds required to finalize all splitters.

Theorem 3.4. *Let s_j be the sampling ratio for the j^{th} round, $\mathcal{I}_j(i)$ be the splitter interval for i^{th} splitter after j rounds and $\mathcal{G}_j$ denote the number of input keys that lie in one of the $\mathcal{I}_j$'s, then, $E(\mathcal{G}_j) \leqslant \frac{2N}{s_j}$.*

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

$$\text{We have,} \quad L_{j-1}(i) \leqslant L_j(i) \leqslant \frac{Ni}{p} \leqslant U_j(i) \leqslant U_{j-1}(i)$$

$$\text{We have, } \forall x : 0 \leqslant x \leqslant \left(U_{j-1}(i) - \frac{Ni}{p}\right),$$
$$P\left[U_j(i) - \frac{Ni}{p} \geqslant x\right] = \left(1 - \frac{ps_j}{N}\right)^x$$

$$\therefore E\Big[U_j(i) - \frac{Ni}{p}\Big] = \sum_{x=1}^{U_{j-1}(i)-\frac{Ni}{p}} P\Big[U_j(i) - \frac{Ni}{p} \geqslant x\Big]$$
$$= \sum_{x=1}^{U_{j-1}(i)-\frac{Ni}{p}} \Big(1 - \frac{ps_j}{N}\Big)^x \leqslant \sum_{x=1}^{\infty} \Big(1 - \frac{ps_j}{N}\Big)^x = \frac{N}{ps_j}$$

On similar lines, we have $E\Big[\frac{Ni}{p} - L_j(i)\Big] \leqslant \frac{N}{ps_j}$

$$E[\mathcal{G}_j] \leqslant E\Big[\sum_{i=1}^{p-1} |\mathcal{I}_j(i) \cap A|\Big] = \sum_i E\Big[U_j(i) - L_j(i)\Big]$$
$$= \sum_i E\Big[\frac{Ni}{p} - L_j(i)\Big] + E\Big[U_j(i) - \frac{Ni}{p}\Big] \leqslant \sum_i \frac{2N}{ps_j} = \frac{2N}{s_j}$$

This completes the proof of Theorem 3.4. Theorem 3.4 suggests that $\mathcal{G}_j$ will be small in expectation. The next theorem shows that it is also small w.h.p..

Theorem 3.5. *If $s_j < \frac{p}{3\ln N}$, then, $\mathcal{G}_j \leqslant \frac{6N}{s_j}$ w.h.p.*

Proof: In Appendix A.

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

Theorem 3.6. *Let Z_j be the sample size for the j^{th} round and $s_j \geqslant s_{j-1}$, then $Z_j \leqslant (7ps_j/s_{j-1})$ w.h.p.*

Proof: We have, $E[Z_j] = \mathcal{G}_{j-1}ps_j/N$. We also have, $\mathcal{G}_{j-1} \leqslant 6N/s_{j-1}$ *w.h.p.*, using Theorem 3.5.

Given that $\mathcal{G}_{j-1} \leqslant 6N/s_{j-1}$, using Chernoff bounds,

$$P[Z_j \geqslant (7ps_j/s_{j-1})] \leqslant P[Z_j \geqslant E[Z_j] + ps_j/s_{j-1}]$$
$$\leqslant e^{-\frac{(ps_j/s_{j-1})^2}{3E[Z_j]}} = e^{-\frac{(ps_j/s_{j-1})^2 N}{3\mathcal{G}_{j-1}ps_j}} \leqslant e^{-\frac{(ps_j/s_{j-1})^2 Ns_{j-1}}{36Nps_j}} \leqslant e^{-\frac{p}{36}}$$

This completes the proof of Theorem 3.6. The next theorem provides a stopping criterion for HSS.

Theorem 3.7. *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..*

Proof: If after round j, $L_j(i) \in \mathcal{T}_i$ or $U_j(i) \in \mathcal{T}_i$, then clearly splitter i is finalized in that round. For l, such that splitter l is not finalized after the $(k-1)^{th}$ round, probability that no sample is picked from $\mathcal{T}_l$ even after the k^{th} round is given by

$$\Big(1 - \frac{2p\ln p}{\epsilon N}\Big)^{|\mathcal{T}_l \cap A|} \leqslant e^{-\frac{2p\ln pN\epsilon}{\epsilon Np}} \leqslant \frac{1}{p^2}$$

Note that the above probability is 0 for splitters that have already been finalized, so the inequality holds for all l.

Finally, since there are at most $p-1$ splitters, the probability that no key is chosen from $\mathcal{T}_i$ for some splitter i, after k rounds, is at most $(p-1) \times p^{-2} < 1/p$. This completes the proof of Theorem 3.7. With Theorems 3.4, 3.5, 3.6 and 3.7 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 finalized w.h.p., using Theorem 3.7. The sample size for the j^{th} histogramming round is at most $7ps_j/s_{j-1} = 7p(2\ln p/\epsilon)^{1/k}$ w.h.p., using Theorem 3.6. This gives us our main lemma.

Lemma 3.8. *With k rounds of histogramming and a sample size of $O\Big(p\sqrt[k]{\frac{\log p}{\epsilon}}\Big)$ per round, HSS achieves $(1+\epsilon)$ load balance w.h.p. for large enough p[4].*

Observe from Lemma 3.8 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,

$$\frac{d(kp\sqrt[k]{\log p/\epsilon})}{dk} = p\sqrt[k]{\log p/\epsilon}\Big(1 - \frac{\log\frac{\log p}{\epsilon}}{k}\Big) = 0$$
$$\Rightarrow \quad k = \log\frac{\log p}{\epsilon}$$

The overall sample size $O(kp\sqrt[k]{\log p/\epsilon})$ attains global minimum for $k = \log(\log p/\epsilon)$ rounds of histogramming. With $k = c\log(\log p/\epsilon)$ rounds of histogramming, the sample size for each round is $O\Big(p(\log p/\epsilon)^{1/(c\log(\log p/\epsilon))}\Big) = O(p)$, where c is a constant. Across all rounds, the overall sample size from all processors across all rounds is $O(p\log(\log p/\epsilon))$. This leads us to the following lemma.

Lemma 3.9. *With $k = O(\log(\log p/\epsilon))$ rounds of histogramming and $O(p)$ sized sample per round (i.e. constant number of samples from each processor), HSS achieves $(1+\epsilon)$ load balance w.h.p. for large enough p.*

We note that our analysis is slightly conservative. Specifically, the algorithm need not keep track of splitter intervals for splitters that get finalized early. In practice, we found the number of histogramming rounds required to be smaller than our bounds. Nevertheless, we believe that the asymptotic bounds proven in our analysis for HSS are tight.

3.4 Approximate Histogramming using random sampling

Often parallel data processing systems have humongous amounts of data and computing histograms repeatedly might be expensive. In this section, we show that an approximate but fairly accurate histogram can be computed using a sample representative of the input data at every processor. The representative sample size at every processor is $O(\sqrt{p\log p}/\epsilon)$.

[4]Specifically, when $s_k = \frac{2\ln p}{\epsilon} \leq \frac{p}{3\ln N}$ for Theorem 3.5

Assume that for approximate histogramming, every processor maintains a representative sample of s keys. We use a sampling technique similar to random sampling as suggested by Blelloch. et al. [7]. Every processor divides its sorted input into s blocks of size N/ps. From every block, a random key is selected as a part of its representative sample. To answer rank queries of the following type: given a key k find rank of k in the overall input, a reduction is performed on local ranks obtained using the representative sample at every processor, rather than the entire input. If r denotes the number of representative sample keys $\leqslant k$ across all processors, the algorithm returns Nr/ps.

Theorem 3.10. *For $s = \sqrt{2p \ln p}/\epsilon$, the rank returned by the above algorithm is within a distance of $N\epsilon/p$ from true rank of k w.h.p.*

Proof: Denote the set of sorted sampled representative keys by $V = \{\lambda_1, \lambda_2 ..., \lambda_{ps}\}$. Consider a processor i. Let the representative sample at processor i be $V_i = \{\lambda_1^i, \lambda_2^i ..., \lambda_s^i\}$. Let the number of blocks (of size N/ps) in processor i that are completely less than k be b_i. Clearly, atleast b_i samples and at most $(b_i + 1)$ samples in V_i are less than k. Let the fraction of keys in the $(b_i + 1) - th$ block that are $\leqslant k$ be l_i, $l_i < 1$. The total number of keys in processor i that are less than k is $(b_i + l_i)N/ps$. Also, the probability that $(b_i + 1) - th$ sample: $\lambda^i_{b_i+1} \leqslant k$ is l_i. Let X_i be the bernoulli random variable denoting if $\lambda^i_{b_i+1} \leqslant k$. Thus, $P(X_i = 1) = l_i$.

True rank of k:R_k is given by $R_k = \sum_i (b_i + l_i)N/ps$. Rank of k as returned by the algorithm is $R = \sum_i (b_i + X_i)N/ps$. Clearly, $E[R] = R_k$.

Since, all random samples are chosen independently, all X_i's are independent. For $s = \sqrt{2p \ln p}/\epsilon$, we have

$$
\begin{aligned}
&P\left(|R - R_k| > \frac{N\epsilon}{p}\right) \\
&= P\left(\left|\sum_{i=1}^{p}(b_i + X_i)\frac{N}{ps} - \sum_{i=1}^{p}(b_i + l_i)\frac{N}{ps}\right| > \frac{N\epsilon}{p}\right) \\
&= P\left(\left|\sum_i (X_i - l_i)\right| > s\epsilon\right) \\
&\leqslant 2e^{-\frac{2(s\epsilon)^2}{p}} = 2e^{-4\ln p} = 2p^{-4} \quad \square
\end{aligned}
$$

The last inequality is obtained using Hoeffding's inequality which holds for non-identical, independent indicator random variables.

If the above algorithm reports the rank of a key k to be in $(Ni/p - N\epsilon/p,\ Ni/p + N\epsilon/p)$, then using Theorem 3.10 its true rank lies in $(Ni/p - 2N\epsilon/p,\ Ni/p + 2N\epsilon/p)$ w.h.p.. The above enhancement can now be used as an oracle to compute the histogram, since histogram is a bunch of rank queries, as long as the size of the histogram is smaller than p^4. It is useful if histograms need to be computed repeatedly.

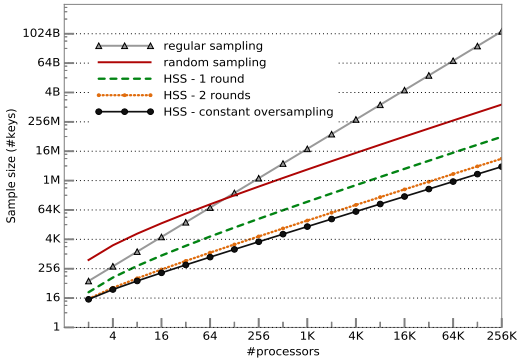

Figure 2. Sample size in practice: sample sort vs HSS for 5% load imbalance

4 Discussion and Related Work

4.1 Sample sort: Sampling methods

We discuss two sampling methods: random sampling and regular sampling, for the sampling step (step 1) in sample sort.

4.1.1 Random sampling

With random sampling as proposed by Blelloch et al. [7], 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. We reproduce Theorem $B.4$ from [7], below:

Theorem 4.1. *Let s be the oversampling ratio, then, for any $\alpha \geqslant 1 + 1/s$, the probability that random sampling causes any processor to contain more than $\frac{\alpha N}{p}$ keys, after sorting is at most $Ne^{-(1-1/\alpha)^2 \frac{\alpha s}{2}}$.*

For $s = (c \ln N/\epsilon^2), \alpha = (1 + \epsilon)$, the probability that no processor contains more than $\alpha = (1 + \epsilon)$ keys after sorting is at most $Ne^{-\frac{c \ln N}{2(1+\epsilon)}} = e^{-\ln N\left(\frac{c}{2(1+\epsilon)}-1\right)} = N^{-\left(\frac{c}{2(1+\epsilon)}-1\right)}$. With $c = 4(1 + \epsilon)$, this comes out to be $1/N$. We conclude thus that the number of samples required with random sampling to achieve desired levels of load balance, specified by ϵ, is $O(p \log N/\epsilon^2)$ keys.

Despite good theoretical guarantees about the quality of splitters with random sampling, its scalability is hindered in practice because of the large sample size it requires for "good" splitting.

4.1.2 Regular sampling

With Regular sampling as proposed by Shi et. al. [21, 25], every processor deterministically picks s evenly spaced keys from its local sorted data. The central processor collects these

samples and merges them to obtain a combined sorted sample $\{\lambda_0, ..., \lambda_{ps-1}\}$ of size ps. $(p-1)$ splitters are selected by picking evenly spaced keys from this sample. Specifically, $\lambda_{si-\frac{p}{2}}$ is chosen for S_i, the i^{th} splitter.

Shi et. al. [25] showed that if the oversampling ratio s is p, then each processor ends up with no more than $\left\lceil 2N/p \right\rceil$ keys. We show that if $s = p$, $R(S_i) \in [Ni/p - N/2p,\ Ni/p + N/2p]$ (recall that $R(k)$ denotes the rank of key k in the overall input). This is consistent with previous results since the number of keys on each processor is upper bounded by $2N/p$, if $R(S_i) \in [Ni/p - N/2p,\ Ni/p + N/2p]$.

For simplicity, we assume that the total number of elements on a processor, N/p, is a multiple of the oversampling ratio s. Denote the local input data on a processor i after local sorting by $\{I^i_1, I^i_2 ..., I^i_{\frac{N}{p}}\}$. Processor i selects keys $\{I^i_{\frac{N}{ps}}, I^i_{\frac{2N}{ps}} ..., I^i_{\frac{N}{p}}\}$ as its local sample.

Theorem 4.2. *If the oversampling ratio for sample sort with regular sampling is s and $S_i = \lambda_{si-\frac{p}{2}}$, then $\forall i : 1 \leqslant i < p,\ |R(S_i) - \frac{Ni}{p}| < \frac{N}{2s}$.*

Proof: Assume that the local sorted input on every processor is partitioned into blocks of size (N/ps), the sample is chosen to be the last element in the block. Since, the number of samples less than or equal to S_i is $(si - p/2)$, there are at least $(si - p/2)$ blocks across all PEs which are less than or equal to S_i. We say that a block is less than a key k if all keys in that block are less than k. Thus, there are at least $(si - p/2) \times N/ps = (Ni/p - N/2s)$ keys across all PEs, which are less than S_i.

Using a similar argument, the number of blocks greater than S_i across all PEs is at least $\left(s(p-i) + p/2 - p\right)$ since there are $\left(s(p-i) + p/2\right)$ samples greater than S_i. Hence, the number of keys across all PEs which are greater than S_i is at least $\left(s(p-i) + p/2 - p\right) \times N/ps = N - (Ni/p + N/2s)$. Using the above two bounds, we conclude that,

$$R(S_i) \in \left[\frac{Ni}{p} - \frac{N}{2s}, \frac{Ni}{p} + \frac{N}{2s}\right] \quad \square$$

If the oversampling ratio s is set to p/ϵ, then the splitters determined using regular sampling are within the required threshold specified by ϵ. Indeed, substituting $s = p/\epsilon$ in Theorem 4.2, we obtain $|R(S_i) - Ni/p| < N\epsilon/2p$.

Lemma 4.3. *If $s = \frac{p}{\epsilon}$ be 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 costly for regular sampling and makes it unscalable for large problem sizes. One way to make regular sampling scalable is to sort the sample in parallel, which introduces significant complexity in the algorithm. Goodrich [13] proposed a communication optimal algorithm that combines ideas from sample sort and merge sort and sorts samples in parallel in a pipelined fashion.

Figure 2 compares the sample size required for HSS and sample sort with random sampling and regular sampling. Clearly for large PEs, both regular and random sampling suffer from large sample sizes.

4.2 Other Sorting Algorithms

HykSort [27] is a state of the art parallel sorting algorithm. It employs multi-staged splitting and communication to achieve better scalability. It uses histogramming for splitter selection and was shown to be quite effective in practice. We believe that our work serves as a justification for the efficacy of HyKSort for large problem sizes. We emphasize that our performance evaluation is meant to be a demonstration of practicality of using histogramming for massive scale parallel sorting rather than building a fully optimised sorting library, which would warrant further considerations of low-level performance optimizations and a separate study. Indeed, the splitting guarantees of HSS are equally valid for a multi-staged sorting algorithm. Nevertheless, we compare our implementation to HykSort (see section 6.3.1).

Parallel sorting by over partitioning was proposed by Li et al. [20] 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 overall collected sample is sorted by the central processor and $pk-1$ splitters are chosen by selecting $s^{th}, 2s^{th}, ..., (pk-1)s^{th}$ keys. These splitters partition the input into pk buckets, more than required and hence k is referred to as the *over partitioning* ratio. 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. Axtmann et. al. [3] proposed a multi-staged algorithm based on the scanning algorithm discussed in 3.1.

4.2.1 Merge based sorting algorithms

In this paper, we focused on data partitioning parallel sorting algorithms, which we refer to as *splitter-based* algorithms. *Merge-based* algorithms are another class of sorting algorithms that employ merging data using sorting networks. An early such result was due to Batcher [4] which uses time (or equivalently depth in a sorting network) $O(\log^2 N)$ with N processors. Ajtai et. al [2] gave the first sorting circuit of depth $O(\log N)$, commonly referred to as the AKS network. The AKS network has large hidden constants because of the use of expander graphs in the circuit [9, 23]. Later, Cole [10] proposed a sorting algorithm that runs in $O(\log N)$ time using N processors and has smaller constants than the AKS network. Another communication optimal $O(\log N)$ algorithm for the BSP model was proposed by Goodrich [13].

However, *merge-based* parallel sorting algorithms do not scale very well when $N >> p$ because of their large data movement and are typically not used for large problem sizes.

5 Running times

We assume a simple BSP model [29], characterized by two parameters T_I, unit computational time and T_c, time for communicating one unit of data. We analyse the computational and communication cost seperately for sample sort and HSS.

Both algorithms have the same cost for initial local sorting, broadcasting splitters and all-to-all data exchange. The computational cost of local sorting is $O(N \log \frac{N}{p}/p)$. No communication is involved in local sorting. The cost of broadcasting splitters once they are finalized is $O(p \log p)$ assuming a binary spanning tree of processors and $O(p + \log p)$ using pipelined broadcasts [24]. The final data movement requires all data to be sent to their destination PEs, hence the communication cost involved is $O(N/p)$. Once a PE receives all data pieces, it needs to merge them, which takes $O(N \log p/p)$ computation time.

5.1 Cost of Sampling

If the overall sample collected at the central PE from all PEs is S, then the communication cost involved is $O(S)$. This step is akin to a *gather* collective operation. The overall sample is sorted by the central PE. Since there are p pieces of samples, one from each PE, the computational cost involved in sorting the overall sample is $O(S \log p)$, assuming merge sort.

5.2 Cost of Histogramming

As the local input is sorted, a local histogram can be computed in $O(S \log \frac{N}{p})$ time using S binary searches, where S denotes the size of the histogram. A global histogram is computed by reducing all local histograms. Using a binomial algorithm [24, 28] for reduction, the computational time involved is $O(S \log p)$. Thus, the total computation cost of histogramming is $O(S \log \frac{N}{p} + S \log p) = O(S \log N)$ using a binomial algorithm for reduction. Alternatively, one can divide the histogram into fragments and pipeline the reductions [24]. Pipelined reductions is suitable for large messages and large p, which is what we need. The computational cost of a pipelined reduction for a message of size S is $O(S+\log p)$. In conclusion, the computational cost of histogramming with pipelined reductions is $O(S \log \frac{N}{p} + S) = O(S \log \frac{N}{p})$.

The histogram probes and the splitter intervals are broadcast to every PE for histogramming. The communication cost of broadcasting a length S message is $O(S \log p)$ using a binomial algorithm [24]. However, one can do better by using a pipelined broadcast. The communication cost of a pipelined broadcast message of size S is $O(S + \log p)$. The reduction of all local histograms to obtain a global histogram also costs $O(S + \log p)$ in terms of communication using pipelined reductions. Thus, the overall communication cost of histogramming is $O(S + \log p)$. Note that both the computation and communication cost of histogramming is proportional to the overall sample size.

The total cost of computation and communication of sample sort and histogram sort are tabulated in Table 1. For large p, the sampling cost dominates the running time of sample sort. Computing histograms on a significantly smaller sample improves the running time of the algorithm. We conclude thus, that running time of HSS is asymptotically superior than sample sort with random and regular sampling.

6 Implementation and Experiments

6.1 Implementation details

We implemented HSS in C++11 using the Charm++ [1, 16] framework. Charm++ employs virtual processors allowing the flexibility and elegance of object based decomposition for parallel programming. It provides a debugging tool and a visualization software [5] for analyzing performance, timelines, cpu usage, memory footprint etc.

6.1.1 Using shared memory

The all-to-all data exchange step results in $p(p-1)$ messages in the network, which doesn't scale very well. We leverage shared memory within a physical node to reduce the number of messages. We modified HSS to be a simple two phase algorithm. In the first phase, the algorithm performs a node level splitting placing all data onto their destination nodes. In the second phase, every node sorts its local data, for instance, using OpenMP or charm++ features.

Data partitioning across nodes reduces the size of the histogram considerably, since p now reflects the number of nodes rather than PEs. For instance BlueGene/L system [12] has 16 cores in a single physical node. Consider sorting on $8K$ nodes or equivalently $8K \times 16 = 128K$ PEs (assuming one PE per core). For data partitioning across all PEs, the required size of the sample for HSS with one round is approximately 250 MB , but only 12 MB for data partitioning across nodes. Once a node receives all the data after the all to all data exchange, it can locally sort the data within node without injecting traffic on the network. We use HSS with one round for final within node sorting.

6.1.2 Other implementation details

Let p denote the total number of PEs (threads) and n denote the number of nodes (ranks). HSS can be thought to comprise of three distinct phases; local sorting of input data, splitter determination using histogramming and final data exchange and assembly. We use STL's `std::sort` for local sorting for the first phase.

- **Histogramming Phase**: The histogramming phase determines $n-1$ splitters for node level splitting. For the sampling phase before every histogramming round, each PE picks a small sample from its input which lie in the union of

Algorithm	Overall sample size	Overall sample size for $p = 10^5, \epsilon = 5\%$	Computation complexity	Communication complexity
Sample sort with regular sampling	$O(\frac{p^2}{\epsilon})$	1600 GB	$O\left(\frac{N}{p}\log\frac{N}{p} + \frac{p^2}{\epsilon}\log p + \frac{N}{p}\log p\right)$	$O\left(\frac{p^2}{\epsilon} + p + \frac{N}{p}\right)$
Sample sort with random sampling	$O(\frac{p\log N}{\epsilon^2})$	8.1 GB	$O\left(\frac{N}{p}\log\frac{N}{p} + \frac{p\log N\log p}{\epsilon^2} + \frac{N}{p}\log p\right)$	$O\left(\frac{p\log N}{\epsilon^2} + p + \frac{N}{p}\right)$
HSS with one round	$O(\frac{p\log p}{\epsilon})$	184 MB	$O\left(\frac{N}{p}\log\frac{N}{p} + \frac{p\log p}{\epsilon}\log N + \frac{N}{p}\log p\right)$	$O\left(\frac{p\log p}{\epsilon} + p + \frac{N}{p}\right)$
HSS with two rounds	$O(p\sqrt{\frac{\log p}{\epsilon}})$	24 MB	$O\left(\frac{N}{p}\log\frac{N}{p} + p\sqrt{\frac{\log p}{\epsilon}}\log N + \frac{N}{p}\log p\right)$	$O\left(p\sqrt{\frac{\log p}{\epsilon}} + p + \frac{N}{p}\right)$
HSS with k rounds	$O(kp\sqrt[k]{\frac{\log p}{\epsilon}})$	-	$O\left(\frac{N}{p}\log\frac{N}{p} + kp\sqrt[k]{\frac{\log p}{\epsilon}}\log N + \frac{N}{p}\log p\right)$	$O\left(kp\sqrt[k]{\frac{\log p}{\epsilon}} + p + \frac{N}{p}\right)$
HSS with $O(\log\frac{\log p}{\epsilon})$ rounds	$O(p\log\frac{\log p}{\epsilon})$	10 MB	$O\left(\frac{N}{p}\log\frac{N}{p} + p\log\frac{\log p}{\epsilon}\log N + \frac{N}{p}\log p\right)$	$O\left(p\log\frac{\log p}{\epsilon} + p + \frac{N}{p}\right)$

Table 1. Running times complexity of HSS and Sample sort. The computation cost of histogramming is $(\log N)$ times the overall sample size. Pipelined reductions and broadcasts is assumed for large messages.

splitter intervals. If δ denotes the fraction of input covered by the splitter intervals, then every PE picks s/δ 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 from every PE is $s \times p$, where s can be thought of as the sampling ratio. The overall sample is assembled at the central PE and broadcast for histogramming. Every PE computes a local histogram using binary searches, since the input data is already sorted. The local histograms are summed up using a reduction and sent to the central PE. We set the load balance threshold to 5% for node level partitioning and 5% for within node partitioning.

- **Data movement**: Once every PE receives the splitters, input data from all PEs within a node are combined and partitioned into n messages, one for each node. We use Charm++'s *nodegroup* chares to implement the combining of messages within a node.
- **Final within node sorting**: Once a node receives all data that falls in its bucket, it needs to merge and redistribute data among its PEs. Our implementation uses regular sampling to determine splitters for PEs within a node.

6.2 Handling duplicates via Implicit Tagging

We propose a simple mechanism to deal with duplicates. Any algorithm that uses sampling depends only on the relative ordering of the keys. For inputs with duplicates, we enforce a strict ordering by implicitly tagging each element with the PE it resides on and the local index of the element in the local data structure. Thus, every input key k can be thought of as a triplet (k, PE, ind), where PE denotes the PE that k resides on and ind denotes the index in the local data structure, where k is stored. Note that implicit tagging does not blow up the size of the input, but increases the size of the histogram by a constant factor since probe keys for the histogram have to be explicitly tagged. Figure 3 illustrates the overhead incurred for handling duplicates with implicit tagging. For 32K PEs, implicit tagging resulted in only about 4% increase in running time.

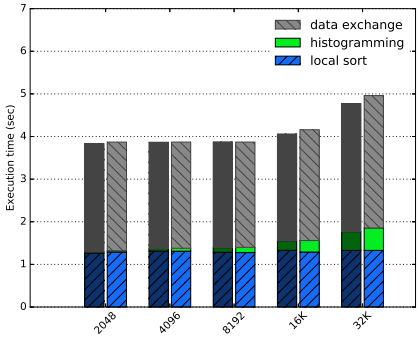

Figure 3. Overhead of handling duplicates via implicit tagging for UNIF distribution with 2M keys/PE and 16 threads/rank. The histograms with lighter shades are the runs with implicit tagging.

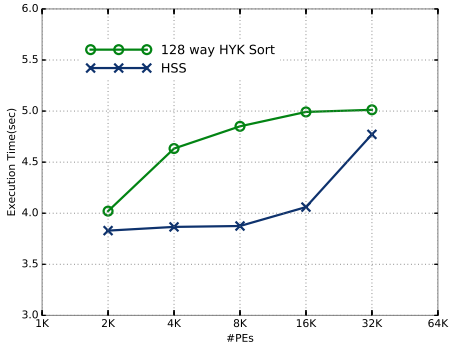

Figure 4. HSS vs 128 way HykSort, with uniformly distributed 2M keys/PE and 16 threads per rank.

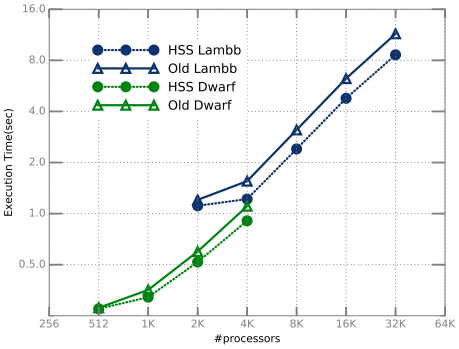

Figure 5. Experiments showing performance of sorting routine of Changa. Datasets used were Lambb and Dwarf

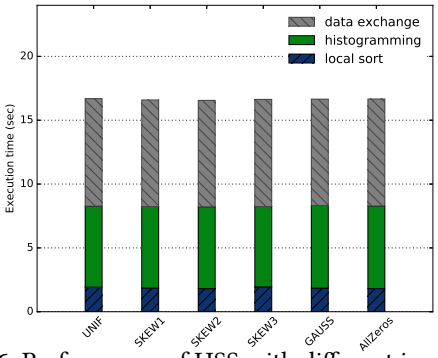

Figure 6. Performance of HSS with different input distributions for 2M keys/PE. We used 1 thread per rank to accentuate the splitter selection (histogramming) time.

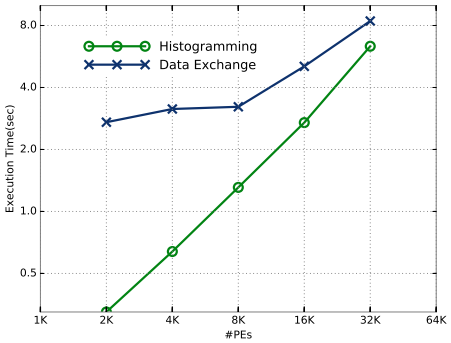

Figure 7. Histogramming time with increasing number of PEs for 2M keys per PE and 1 thread per rank.

6.3 Experimental results

All our experiments were performed on the Vesta IBM Blue Gene/Q supercomputer at Argonne Leadership Computing Facility (ALCF). Each node is equipped with a PowerPC A2 1600 MHz processor containing 16 cores, each with 4 hardware threads, running at 1.6 GHz, and 16 GB of DDR3 memory. The nodes are connected by IBM's 5D torus interconnect. We use 8 byte long integer type keys for all our experiments. Unless stated otherwise, we use oversampling ratio of 1 per PE, i.e. 1 key on average is sampled from every PE for the splitting round. We found this to be a reasonable value of the oversampling ratio.

6.3.1 Weak Scaling

We ran weak scaling experiments with 2 million keys per PE. We compare our results to HykSort [27]. We used $k = 128$ as the parameter for HykSort as per [27]. We used 16 threads per rank for both HSS and HykSort. A summary of our experiments are reported in Figure 4. HykSort suffers from multiple stages of data movement step. As the number of PEs increase however, the multiple stages of data movement is likely to pay off. From Figure 4, we conclude that won't happen till a large problem size. Since, the results presented in this paper are for the splitting phase, they are equally applicable to an algorithm that uses multiple stages of data movement.

6.3.2 Effect of Input Distribution

The theoretical guarantees for HSS are valid for all distributions. To verify the same experimentally, we ran HSS with the following input distributions:

1. UNIF: Keys are picked 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 picked uniformly at random from a small range of size 1000
3. SKEW2: Keys are chosen randomly from the range [0, 100]
4. SKEW3: Each key is bitwise and of two uniformly at random chosen keys
5. GAUSS: Keys are gaussian distributed
6. AllZeros: All keys are set to 0.

As figure 6 illustrates, HSS is impervious to the input distribution as also expected from the theory. To underscore the histogramming cost, we used 1 thread per rank. Recall that the number of splitters is equal to the number of ranks.

6.3.3 Histogramming time with varying PEs

Figure 7 illustrates the histogramming cost with varying number of PEs. We used one thread per rank to underscore the cost of histogramming (the number of splitters to be chosen thus, equals the total number of threads). The histogramming cost roughly increases by a factor of 2 with a 2x increase in the number of PEs as also expected from the theoretical $O(p \log \frac{\log p}{\epsilon})$ bound (Table 1).

With 16 threads per rank, the histogramming cost is much less because of fewer number of splitters to determine. With 16 threads per rank, we observe that even for large number of PEs, the histogramming phase takes very little fraction of the running time (Figure 3). We believe this is both because of efficient histogramming and node level partitioning.

The number of histogramming rounds taken by the algorithm is tabulated in Table 2. If the sample size is (pf) every round, the number of histogramming rounds required to determine $(p-1)$ splitters is $\lceil \ln(2\ln p/\epsilon)/\ln(f/2) \rceil$. We obtain this expression using expected value of sample sizes and length of splitter intervals and Lemma 3.8. As discussed earlier, the constants involved in our analysis are conservative. In practice, HSS required fewer histogramming rounds. The number of rounds with constant oversampling per round; $O(\log(\log p/\epsilon))$ increases very slowly with p and hence, HSS is extremely scalable and practical.

6.4 Parallel sorting in ChaNGa

In this section, we present our experimental results for ChaNGa [15], a real world astronomical application that often runs on several thousands of processors in typical use cases. ChaNGa poses unique challenges for sorting. First, it employs virtual processors and hence the number of buckets (equal to the number of virtual processors) are far more than the actual number of processors. In such a case, efficient splitting is even more crucial, as the number of buckets is substantially higher. In our examples the number of buckets were typically the number of cores. Second, the virtual processors can be arbitrarily placed on any physical processor. Hence, the processor ordering is not contiguous. More specifically, processor i and processor $i+1$ might be placed arbitrarily far away from each other. Because of this reason, the optimisation to execute splitting across nodes, exploiting shared memory across nodes is not relevant to ChaNGa.

Figure 5 illustrates performance of ChaNGa with HSS and just Histogram sort (old) for two datasets namely Dwarf and Lambb. We refer the interested readers to [15] for more details about the datasets. As can be observed from Figure 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. At this point, it is natural to ask, why would anyone use more number of processors if the parallel sorting algorithm performs better with fewer processors. The answer lies in the fact that parallel sorting only constitutes one part of the application. The rest of the computation can be parallelised with more processors.

With the histogramming cost so reduced, the all-to-all data exchange step is the most expensive step of the algorithm. The data exchange step represents inherent cost in any sorting algorithm, since all keys need to be moved to their correct destination. It is possible that the data exchange step can be optimized by using specialized collective implementations [18, 19, 28] for all-to-all communication. We leave this for future work.

$p(\times 10^3)$	sample size per round ($\times p$)	Number of rounds	Bound on number of rounds
4	5	4	8
8	5	4	8
16	5	4	8
32	5	4	8

Table 2. Number of histogramming rounds observed with $\epsilon = 0.02$, 1M keys per PE and 1 thread per rank.

7 Conclusions

We presented Histogram sort with sampling (HSS), that combines sampling and histogramming to accomplish fast splitter determination. Specifically, we showed that with k rounds of histogramming, the algorithm requires a sample of size only $O(p\sqrt[k]{\log p/\epsilon})$ per round as compared to sample sort with random sampling which requires $O(p\log p/\epsilon^2)$ samples. We argued that the cost of histogramming is dominated by the sampling cost in the running time and hence HSS is theoretically more efficient than sample sort. Our study establishes the theoretical soundness of histogramming for splitter selection, that has been known to work well in practice. We also note that most of the running time of HSS is spent in local sorting and data exchange, both of which are inherent to any sorting algorithm. The reduced sample size makes HSS extremely practical for massively parallel applications, scaling to tens of thousands of processors.

Acknowledgments

The authors are grateful to Edgar Solomonik for providing useful feedback, Omkar Thakoor for proofreading the analysis and Nitin Bhat in helping with the experiments. The authors acknowledge the Argonne Leadership Computing Facility (ALCF) for providing HPC resources that have contributed to the research results reported within this paper. URL: https://www.alcf.anl.gov/.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] Alan Gara, Matthias A Blumrich, Dong Chen, GL-T Chiu, Paul Coteus, Mark E Giampapa, Ruud A Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V Kopcsay, et al. 2005. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development* 49, 2 (2005), 195–212.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] Laxmikant V Kale, Sameer Kumar, and Krishnan Varadarajan. 2003. A framework for collective personalized communication. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[19] Sameer Kumar, Amith Mamidala, Philip Heidelberger, Dong Chen, and Daniel Faraj. 2014. Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer. *The International Journal of High Performance Computing Applications* 28, 4 (2014), 450–464.

[20] 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.

[21] 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. https://doi.org/10.1016/0167-8191(93)90019-H

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

[23] M. S. Paterson. 1990. Improved sorting networks with O(logN) depth. *Algorithmica* 5, 1 (1990), 75–92. https://doi.org/10.1007/BF01840378

[24] 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. https://doi.org/10.1007/s10586-007-0012-0

[25] Hanmao Shi and Jonathan Schaeffer. 1992. Parallel Sorting by Regular Sampling. *J. Parallel and Distrib. Comput.* 14, 4 (April 1992), 361–372. https://doi.org/10.1016/0743-7315(92)90075-X

[26] E. Solomonik and L. V. Kale. 2010. Highly scalable parallel sorting. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470406

[27] 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. https://doi.org/10.1145/2464996.2465442

[28] 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.

[29] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

A Appendix: Proof of Theorem 3.5

Theorem A.1.

$$\mathcal{G}_j \leq \sum_i min\Big(\frac{N}{p}, U_j(i) - \frac{Ni}{p}\Big) + min\Big(\frac{N}{p}, \frac{Ni}{p} - L_j(i)\Big)$$

Proof: By definition, $\mathcal{G}_j$ is the size of the union of the intervals $(\mathcal{I}_j(i) \cap A)$,

$$\mathcal{G}_j = \Big|\bigcup_i (\mathcal{I}_j(i) \cap A)\Big|, \text{ where } \mathcal{I}_j(i) = [I(L_j(i)), I(U_j(i))]$$

The above theorem effectively strips the splitter interval $[I(L_j(i)), I(U_j(i))]$ to $[I(max(Ni/p - N/p, L_j(i))), I(min(Ni/p + N/p, U_j(i)))]$.

To prove that stripping doesn't change the union of all splitter intervals, consider a $U_j(i)$ which is greater than $(Ni/p + N/p)$. Then by definition, we have $U_j(i) = U_j(i+1)$. Thus, the portion of $\mathcal{I}_j(i)$ that extends beyond $(Ni/p + N/p)$ is included in $\mathcal{I}_j(i+1)$. Hence, restricting $U_j(i)$ to $Ni/p + N/p$ does not change the union of $\mathcal{I}_j$'s, i.e. $\mathcal{G}_j$. One can use an inductive argument to see that restricting all U_j's doesn't change $\mathcal{G}_j$, by considering splitter intervals from left to right. A similar argument can be used for L_j's. □

Let $N_j(x) = \sum_i [U_j(i) - Ni/p > x]$, where square brackets signify an indicator variable. $N_j(x)$ denotes the number of splitters i for which $U_j(i) - Ni/p > x$. We have,

$$\sum_i min\Big(\frac{N}{p}, U_j(i) - \frac{Ni}{p}\Big) = \sum_i \sum_{x=0}^{N/p} [U_j(i) - \frac{Ni}{p} > x]$$
$$= \sum_{x=0}^{N/p} N_j(x)$$

Let x_0 be the smallest x such that $E[N_j(x)] \leqslant \frac{p}{s_j}$.

Theorem A.2. *For* $0 \leqslant x < min(x_0, \frac{N}{p})$, $N_j(x) < 2E[N_j(x)]$ *w.h.p.*

Proof: This can be easily seen using multiplicative chernoff's bound.

$$P\Big[N_j(x) \geqslant 2E[N_j(x)]\Big] \leqslant \left(\frac{e^1}{(2)^2}\right)^{E[N_j(x)]} \leqslant (0.53)^{\frac{p}{s_j}} \leqslant e^{-\frac{p}{3s_j}} \leqslant e^{-\ln N} = \frac{1}{N}$$

Theorem A.3. *For $x_0 \leqslant x \leqslant N/p$, $N_j(x) < E[N_j(x)] + \frac{p}{s_j}$ w.h.p.*

Proof: This can be easily seen using additive chernoff's bound.

$$P\Big[N_j(x) \geqslant E[N_j(x)] + \frac{p}{s_j}\Big] \leqslant e^{-\frac{p}{3s_j}} \leqslant e^{-\ln N} = \frac{1}{N}$$

In proving the above theorems, we have used the fact that $[U_j(i_1) - Ni/p > x]$ and $[U_j(i_2) - Ni/p > x]$ are independent indicator random variables for $0 < x \leqslant N/p$ and $i_1 \neq i_2$. Combining Theorem A.2 and Theorem A.3, w.h.p. we have,

$$\begin{aligned}
\sum_i min\Big(\frac{N}{p}, U_j(i) - \frac{Ni}{p}\Big) &= \sum_{x=0}^{N/p} N_j(x) \\
&= \sum_{x=0}^{x_0} N_j(x) + \sum_{x=x_0}^{N/p} N_j(x) \\
&\leqslant \sum_{x=0}^{x_0} 2E[N_j(x)] + \sum_{x=x_0}^{N/p} \Big(E[N_j(x)] + \frac{p}{s_j}\Big), \quad w.h.p. \\
&\leqslant 2\sum_i E\Big[U_j(i) - \frac{Ni}{p}\Big] + \frac{N}{p}\frac{p}{s_j} \\
&\leqslant \frac{2N}{s_j} + \frac{N}{s_j} = \frac{3N}{s_j}
\end{aligned}$$

The limits of x in $N_j(x)$ go from 0 to N/p. Hence, the above is false with probability at most $N/p \times 1/N = 1/p$. Hence it holds w.h.p..
On similar lines we have, $\sum_i min(N/p, Ni/p - L_j(i)) \leqslant \frac{3N}{s_j}$, *w.h.p.*.
And using Theorem A.1,

$$\begin{aligned}
\mathcal{G}_j &\leqslant \sum_i min\Big(\frac{N}{p}, U_j(i) - \frac{Ni}{p}\Big) + min\Big(\frac{N}{p}, \frac{Ni}{p} - L_j(i)\Big) \\
&\leqslant \frac{6N}{s_j}
\end{aligned}$$

This completes the proof of Theorem 3.5. □