Histogram Sort with Sampling

Vipul Harsh
University of Illinois at Urbana-Champaign
Urbana, IL, USA
vharsh2@illinois.edu

Laxmikant Kale
University of Illinois at Urbana-Champaign
Urbana, IL, USA
kale@illinois.edu

ABSTRACT

Standard parallel sorting algorithms like sample sort rely on data partitioning techniques to distribute keys across processors. The sampling cost in sample sort for good load balance is prohibitive for massive clusters. We present Histogram sort with sampling, a parallel sorting algorithm that provably ensures load balance bounded by a factor of $(1+\epsilon)$, where ϵ is provided as an input. Histogram sort with sampling combines sampling and histogramming to accomplish fast splitter determination. We show that the sample size in sample sort, with random sampling, can be reduced from $\mathcal{O}(p \log N/\epsilon^2)$ to $\mathcal{O}(p \log p/\epsilon)$* by performing one round of histogramming on the sample w.h.p.†. With k rounds of histogramming, the sample size can be further brought down to $\mathcal{O}(kp \sqrt[k]{\log p/\epsilon})$. Histogram sort with sampling 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 processors. In our practical implementation, we exploit shared memory within nodes to improve the performance of our algorithm on large modern clusters.

CCS Concepts

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

Keywords

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

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* [18], 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, or in other words the load imbalance‡ after sorting be bounded by $(1+\epsilon)$. 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. The i^{th} bucket is owned by the i^{th} processor. Two popular algorithms in this regard are Sample sort and Histogram sort. Our algorithm is partly inspired from both of these algorithms.

Sample sort [13] is a popular parallel sorting algorithm that relies on data partitioning. Sample sort with regular sampling [24, 28] 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. [8] requires $\Theta(p \log N/\epsilon^2)$ samples overall across all processors to achieve the same load balance. Both of these are impractical 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 [25]. We discuss Sample sort with both regular and random sampling in Section 4.1.

Histogram sort as proposed by Kale et al. [20, 29] 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

*N = total number of keys, p = number of processors

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

‡More precisely, we define load imbalance to be the ratio of maximum load and average load

DOI: 10.475/123_4

$\log \mathcal{N}$, where $\mathcal{N}$ is the range of the input (i.e. maximum key minus the minimum key), although in practice it takes fewer rounds [20, 29] for many distributions. Nevertheless, for skewed distributions the number of rounds could be large.

We present Histogram sort with sampling (HSS), a highly scalable and practical parallel sorting algorithm that relies on data partitioning. 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 $\mathcal{O}(p \log p/\epsilon)$ samples, it determines all the splitters with high probability. The splitters determined this way achieve load balance levels specified by ϵ. We describe Histogram with Sampling with one round of histgoramming in Section 3.1. The data movement step of Histogram sort with sampling after determining the splitters is identical to sample sort and histogram sort.

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 $\mathcal{O}(p \sqrt[k]{\log p/\epsilon})$ every round, accounting for an overall sample size of $\mathcal{O}(kp \sqrt[k]{\log p/\epsilon})$ across all rounds. The details for the general case of k rounds are described in Section 3.2. Lemmas 3.1, 3.2 and 3.3 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 Histogram sort with sampling with one round and two rounds, respectively.

In section 5, we compare the running time complexity of HSS and sample sort. Finally, we demonstrate the scalability of our algorithm on large supercomputing clusters. We leverage shared memory within nodes to make node level optimizations to our algorithm for better performance and scalability. The details of our implementation and experiments are described in Section 6.

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 4.3, we describe how to deal with the case when the input has too many duplicates. 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. In other words, the problem requires the load imbalance to be bounded by a factor of $(1+\epsilon)$. Cheng et al. [9] propose an algorithm that finds exact splitters that achieve perfect load balance with $\mathcal{O}(p \log N)$ 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 [28, 24], histogram sort [20, 29], sample sort by random sampling [13, 8] and parallel sorting by over partitioning [23] 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 will partition 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 tolerated in the problem statement, for our algorithm, we enforce the following conservative condition:

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

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 get affected from load imbalance in the input.

We evaluate our algorithm using the bulk synchronous parallel model (BSP) suggested by Valiant [31]. Comparison of execution times with sample sort suggest that Histogram sort with sampling is theoretically more efficient than parallel sample sort (see section 5).

Since our algorithm borrows some ideas from sample sort [13] and histogram sort [20], 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 [13, 7, 16, 24, 28] 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, often chosen to be merge sort.

Although sample sort requires a large number of samples to provably determine "good" 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 [20, 29] 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. After finalizing the splitters, the rest of the algorithm involving data movement is identical to 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 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* [18].

3. HISTOGRAM SORT WITH SAMPLING

The basic skeleton of our algorithm is similar to that of Histogram Sort. In addition, we use 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, Histogram sort with sampling 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.

SAMPLING METHOD 1. *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. With high probability, the splitters obtained after the first histogramming round achieve the specified level of load balance.

Recall that 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 $\mathcal{T}_i$ with high probability.

Every processor picks a sample from their local input as per Method 1. The overall sample collected from all processors is used for the histogramming round.

THEOREM 3.1. *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

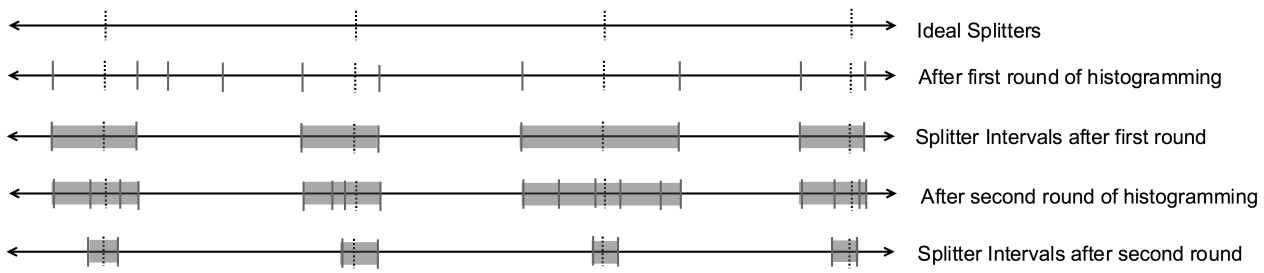

Figure 1: Figure illustrating Histogram sort with sampling (HSS) with multiple rounds. After first round, samples are picked only from the splitter intervals. Notice how the splitter intervals shrink as the algorithm progresses.

chosen from $\mathcal{T}_i$ in the overall sample is given by,

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

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 3.1. *With one round of histogramming and $\mathcal{O}\left(\frac{p\log p}{\epsilon}\right)$ sized sample, Histogram sort with sampling achieves $(1+\epsilon)$ load balance w.h.p.*

3.2 Multiple rounds of histogramming

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

The remainder of this section enumerates a sequence of theorems that sheds more light on the algorithm. Theorems 3.2 and 3.3 bound $\mathcal{G}_j$ in terms of the sampling ratios, Theorem 3.4 bounds the sample size in terms of the sampling ratios and Theorem 3.5 provides a way to bound the number of rounds required to finalize all splitters.

THEOREM 3.2. *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, we have,

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

We have, $\forall x : 0 \leqslant x \leqslant \Big(U_{j-1}(i) - \frac{Ni}{p}\Big)$,

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

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

$$= \sum_{x=1}^{U_{j-1}(i) - \frac{Ni}{p}} \big(1 - \frac{ps_j}{N}\big)^x$$

$$\leqslant \sum_{x=1}^{\infty} \big(1 - \frac{ps_j}{N}\big)^x = \frac{N}{ps_j}$$

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 3.2. □

Theorem 3.2 suggests that $\mathcal{G}_j$ will be small in expectation. The next theorem shows that it is also small with high probability.

THEOREM 3.3. *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 3.4. *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 3.3.

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

$$P(Z_j \geqslant (7ps_j/s_{j-1})) \leqslant P(Z_j \geqslant E[Z_j] + ps_j/s_{j-1})$$

$$\leqslant e^{-\frac{(ps_j/s_{j-1})^2}{3E[Z_j]}}$$

$$= e^{-\frac{(ps_j/s_{j-1})^2 N}{3\mathcal{G}_{j-1} ps_j}}$$

$$\leqslant e^{-\frac{(ps_j/s_{j-1})^2 N s_{j-1}}{36 N ps_j}}$$

$$\leqslant e^{-\frac{p}{36}}$$

This completes the proof of Theorem 3.4. □

The next theorem provides a stopping criterion for the algorithm.

THEOREM 3.5. *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 any 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, the probability that no sample is picked from $\mathcal{T}_l$ even after the k^{th} round is given by,

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

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 3.5. □

With Theorems 3.2, 3.3, 3.4 and 3.5 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 Histogram sort with sampling 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 3.5. 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 3.4. This gives us our main lemma.

LEMMA 3.2. *With k rounds of histogramming and a sample size of $\mathcal{O}\Big(p \sqrt[k]{\frac{\log p}{\epsilon}}\Big)$ per round , Histogram sort with sampling achieves $(1+\epsilon)$ load balance w.h.p. for large enough p*[§].

Observe from Lemma 3.2 that there is a trade off between the sample size per round ($=\mathcal{O}(p \sqrt[k]{\log p/\epsilon})$) 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 $\mathcal{O}(kp \sqrt[k]{\log p/\epsilon})$ 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 $\mathcal{O}\big(p(\log p/\epsilon)^{1/(c \log(\log p/\epsilon))}\big) = \mathcal{O}(p)$, where c is a constant. Across all rounds, the overall sample size from all processors across all rounds is $\mathcal{O}(p \log(\log p/\epsilon))$. This leads us to the following lemma.

[§]Specifically, when $s_k = \frac{2 \ln p}{\epsilon} \leq \frac{p}{3 \ln N}$ for Theorem 3.3

LEMMA 3.3. *With $k = \mathcal{O}(\log(\log p/\epsilon))$ rounds of histogramming and $\mathcal{O}(p)$ sized sample per round (i.e. constant number of samples from each processor), Histogram sort with sampling achieves $(1+\epsilon)$ load balance w.h.p. for large enough p.*

It's worthwhile to note at this point 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 rounds required for good load balance to be smaller than our bounds. Nevertheless, we believe that the asymptotic bounds proven in our analysis for our algorithm are tight.

4. DISCUSSION AND RELATED WORK

4.1 Sample sort: Sampling methods

In this section, we discuss two popular sampling methods for the sampling step (step 1) in sample sort, namely random sampling and regular sampling.

4.1.1 Random sampling

We consider sample sort with random sampling as proposed by Blelloch et al. [8]. If the oversampling ratio is s, each processor divides its local sorted input into s blocks of size (N/ps) and samples a random key in each block. 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 [8], 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 $\mathcal{O}(p\log N/\epsilon^2)$ 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

Regular sampling was proposed by Shi et al. [24, 28] as a deterministic sampling technique for sample sort. Every processor sorts its local input data and picks s evenly spaced keys from its local data. The central processor collects these samples and merges them to obtain a combined sorted sample $\{\lambda_0, \lambda_1 ..., \lambda_{ps-1}\}$ of size ps. $(p-1)$ splitters are selected by picking evenly spaced keys from this sample. More precisely, $\lambda_{si-\frac{p}{2}}$ is chosen for S_i, the i^{th} splitter.

Previous works [28] have shown that if the oversampling ratio s is p, then each processor will end up with no more than $\lceil 2N/p \rceil$ 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 $R(S_i) \in [Ni/p - N/2p,\ Ni/p + N/2p]$.

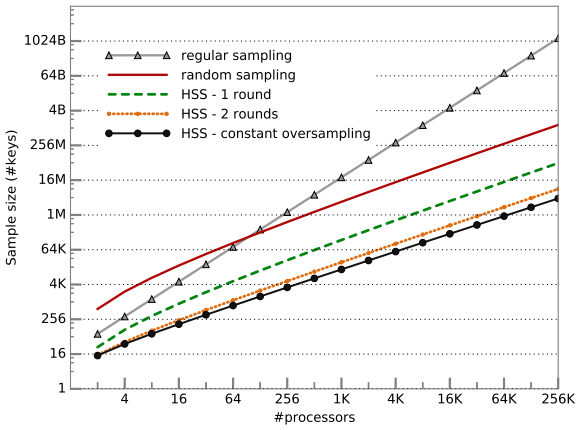

Figure 2: Sample size in practice: sample sort vs HSS for 5% load imbalance

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 largest element of each block being a sampled key. 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 processors 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 processors, which are less than S_i.

Using a similar argument, the number of blocks greater than S_i across all processors is at least $\big(s(p-i) + p/2 - p\big)$ since there are $\big(s(p-i) + p/2\big)$ samples greater than S_i. Hence, the number of keys across all processors which are greater than S_i is at least $\big(s(p-i) + p/2 - p\big) \times N/ps = N - (Ni/p + N/2s)$.

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 4.1. *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. For instance, if $p = 4K$ and $\epsilon = 0.05$, about 300 million keys need to be analyzed. For these reasons, regular sampling does not scale well to a large number of processors. One way to make regular sampling scalable is to sort the sample in parallel, which introduces significant complexity in the algorithm. Goodrich [15] 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 illustrates the comparison between sample size required for sample sort with random sampling and regular sampling versus the sample size required for HSS. Clearly for large processors, both regular sampling and random sampling suffer from large sample sizes.

4.2 Other 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 parallel 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) $\mathcal{O}(\log^2 N)$ with N processors. Ajtai et. al [2] gave the first sorting circuit of depth $\mathcal{O}(\log N)$, commonly referred to as the AKS network. The AKS network has large hidden constants because of the use of expander graphs in the circuit [10, 26]. Later, Cole [11] proposed a sorting algorithm that runs in $\mathcal{O}(\log N)$ time using N processors and has smaller constants than the AKS network. Another communication optimal $\mathcal{O}(\log N)$ algorithm for the BSP model was proposed by Goodrich [15]. However, *merge-based* parallel sorting algorithms do not scale very well when $N >> p$ because of their large data movement. We briefly describe some of these algorithms below.

4.2.1 Bitonic Sort

Batcher's Bitonic sort [4] is a parallel sorting algorithm that is based on merging *bitonic sequences*. A bitonic sequence is a sequence that first increases and then decreases. Its theoretical properties have been extensively studied on various parallel network topologies such as the hypercube [6, 7, 17]. It is not however used in large applications where $N >> p$ because of its large data movement. It moves every piece of data $\Theta(\log p)$ times. As Bitonic sort has been extensively studied, we'll not go into its analysis or experimentation. We refer the interested reader to [4] and [7] for a comparative study.

4.2.2 Radix Sort

Radix sort [7, 12] uses binary representation to group keys into buckets and then recursively sorts each bucket. A k-bit radix looks at k bits of the binary representation every iteration starting from the most significant bits. In the first iteration keys are grouped in 2^k buckets according to their k most significant bits. Each bucket is then recursively grouped using the next k bits. The parallelism in Radix sort comes from the fact that the recursive sorting of buckets can be assigned to multiple processors. One significant issue with parallel radix sort, that makes it impractical for large applications, is that it has large data movement. In every step, there is an all-to-all data exchange. Moreover, since it uses bit representations instead of comparisons, it is not suitable for sorting non-integer type keys like floating points or strings, for instance.

4.2.3 Parallel sorting by over partitioning

Parallel sorting by over partitioning was proposed by Li et al. [23] for shared memory multiprocessors. Every processor picks a sample of size pks from its local input randomly and sends it to a central processor. The overall collected sample is sorted by the central processor and $pk - 1$ splitters are chosen from the sample 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 ordered from the largest sublists size to the smallest. Each processor picks one task at a time and processes it by copying the data to the appropriate position in the memory. The memory positions are determined using the splitters. Li et al. [23] demonstrated that good load balance is achieved when the over partitioning ratio is set to $\log p$. However, it is not immediately clear how to extend the idea of task queues for a distributed cluster.

4.3 Dealing with duplicates

Previous works [28] have shown that with sample sort, load balance deteriorates *linearly* with the number of duplicates, no matter how the samples are chosen. Helman et al. [16] propose a variant of sample sort that automatically handles duplicates. Their key idea is to transpose the input data by redistributing it across processors. We note however that this requires two steps of the expensive all-to-all communication instead of one.

We propose a simple mechanism to deal with a large number of duplicates in the input without blowing up the input size. Any algorithm which uses sampling techniques depends only on the relative ordering of the keys. For a large number of 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) key, where PE denotes the processor that k resides on and ind denotes the index in the local data structure where k is stored. Implicit tagging does not blow up the size of input data, but increases the size of the histogram by a constant factor since probe keys for the histogram have to be explicitly tagged. A similar scheme was used to deal with duplicates in [3].

5. RUNNING TIMES

To analyse the running times, we'll assume a simple BSP model [31]. Such a model is 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 histogram sort with sampling.

Both algorithms have the same cost for initial local sorting, broadcasting splitters and all-to-all data exchange for the final data movement. The computational cost of local sorting is $\mathcal{O}(N \log \frac{N}{p}/p)$. No communication is involved in local sorting. The cost of broadcasting splitters once they are finalized is $\mathcal{O}(p \log p)$ assuming a binary spanning tree of processors and $\mathcal{O}(p + \log p)$ using pipelined broadcasts [27].

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

Table 1: Running time complexity of Histogram sort with sampling (HSS) and Sample sort. The computation cost of histogramming is $(\log N)$ times the overall sample size. Both algorithms have the same cost of local sorting and data movement. Pipelined reductions and broadcasts is assumed for large messages.

The final data movement requires all data to be sent to their destination processors, hence the communication cost involved is $\mathcal{O}(N/p)$. Once a processor receives all messages from the data movement step, it needs to merge all data pieces, which takes $\mathcal{O}(N \log p/p)$ computation time.

5.1 Cost of Sampling

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

5.2 Cost of Histogramming

Computing a local histogram is equivalent to answering multiple rank queries. Since, the local input is sorted, a local histogram can be computed in $\mathcal{O}(S \log \frac{N}{p})$ 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 [27, 30] for reduction, the computational time involved is $\mathcal{O}(S \log p)$. Thus, the total computation cost of histogramming is $\mathcal{O}(S \log \frac{N}{p} + S \log p) = \mathcal{O}(S \log N)$ using a binomial algorithm for reduction. Alternatively, one can divide the histogram into fragments and pipeline the reductions [27]. 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 $\mathcal{O}(S + \log p)$. In conclusion, the computational cost of histogramming with pipelined reductions is $\mathcal{O}(S \log \frac{N}{p} + S) = \mathcal{O}(S \log \frac{N}{p})$.

The histogram probes and the splitter intervals are broadcast to every processor for histogramming. The communication cost of broadcasting a length S message is $\mathcal{O}(S \log p)$ using a binomial algorithm [27]. However, one can do better by using a pipelined broadcast. The communication cost of a pipelined broadcast message of size S is $\mathcal{O}(S + \log p)$. The reduction of all local histograms to obtain a global histogram also costs $\mathcal{O}(S + \log p)$ in terms of communication using pipelined reductions. Thus, the overall communication cost of histogramming is $\mathcal{O}(S + \log p)$. 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 histogram sort with sampling is asymptotically superior than sample sort with random and regular sampling that achieve the same level of load balance.

6. IMPLEMENTATION AND EXPERIMENTS

6.1 Implementation details

In this section, we discuss some of the details of our implementation. We implemented HSS in C++11 using the Charm++ [1, 19] framework. The programing model in Charm++ employs virtual processors allowing the flexibility and elegance of object based decomposition for parallel programming. It provides a debugging tool and a useful visualization software [5] for analyzing performance, timelines, cpu usage, memory footprint etc. and getting useful insights into the algorithm. Charm++ allows an application to create any number of virtual processors, called chares. In addition, node level chares can be created and concurrently scheduled on any processor core on a single node. Node level chares or *nodegroups* in Charm++ terminology provide a very powerful abstraction for convenient shared memory programming.

6.1.1 Using shared memory

In this section, we describe shared memory optimizations to our algorithm. The all-to-all data exchange step of the sorting algorithm results in $p(p-1)$ messages in the network, which is extremely large for large values of p. We leverage

shared memory within a physical node to reduce the number of messages. In most supercomputing clusters, multiple processor cores lie on a single physical node and memory can be shared between the cores. Hence, its superfluous to send multiple fine grained data messages between a pair of nodes. Instead, all messages going to the same node can be combined into a larger message. For instance, if the number of cores on one node of a machine is 50, then combining node level messages results in $\sim 2500\times$ fewer messages in the network.

This opens door for other optimizations too. With node level destinations, the data partitioning needs to be only across physical nodes and not across individual processor cores. Data partitioning across nodes reduces the size of the histogram considerably, since p now reflects the number of physical nodes instead of the number of individual processor cores. For instance BlueGene/L system [14] has 16 cores in a single physical node. Consider sorting on $8K$ nodes or equivalently $8K \times 16 = 128K$ cores. For data partitioning across processor cores, the required size of the sample is approximately 250 MB for HSS with one round, whereas for data partitioning across nodes, the required sample size is only 12 MB. Clearly, node level partitioning makes the histogramming step much more scalable, in addition to reducing the number of messages in the network in the data movement step. 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. Since the number of splitters required for splitting data within node is significantly smaller, we use sample sort with regular sampling for this purpose.

6.1.2 Other implementation details

We let p denote the number of processor cores and n denote the number of physical nodes. If there were 16 cores on a single node, then p would be equal to $16n$.

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. The local sorting phase is embarrassingly parallel and requires no communication.

- **Histogramming Phase**
  The histogramming phase determines $n-1$ splitters for node level splitting. For the sampling phase before every histogramming round, each processor picks a small sample from its input which lie in the union of splitter intervals. If δ denotes the fraction of input covered by the splitter intervals, then every processor picks $5/\delta$ 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 from every processor is $5p$. The overall sample is assembled at the central processor and broadcast for histogramming. Every processor 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 processor. The reduction and broadcast infrastructure is provided by Charm++. The load balance threshold is set to 2% for node level partitioning and 5% for within node partitioning.

- **Data movement**
  Once every processor receives the splitters, input data from all processors 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 processor cores. Our implementation uses regular sampling to determine splitters for processors within a node. This step executes completely within a multicore node and does not inject any traffic on the network.

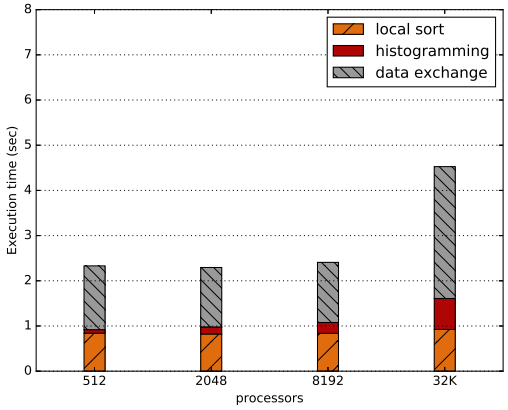

Figure 3: HSS on Mira (ALCF) with node level partitioning. Each processor core had 1 million 8 byte long integer keys with a 4 byte payload.

6.2 Experimental results

In this section, we present our experiments on large supercomputing clusters - Mira at Argonne Leadership Computing Facility (ALCF). We present results for weak scaling experiments.

Mira is an IBM Blue Gene/Q supercomputer with 16 cores per node. 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 gigabytes of DDR3 memory. The nodes are connected by IBM's 5D torus interconnect. We ran weak scaling experiments on Mira with 1 million 8 bytes integer keys per core with 4 byte payload with each key. A summary of our experiments are reported in Figure 3. We used 16 threads per node, one for each core.

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 3.2. As discussed earlier, the constants involved in our analysis are conservative. In practice, we observe that HSS required fewer histogramming rounds. The number of rounds with constant oversampling per round; $\mathcal{O}(\log(\log p/\epsilon))$ increases very slowly with p and hence, HSS is extremely scalable and practical. We observe that even for large number of processors, the histogramming phase takes very little fraction of the running

$p(\times 10^3)$	sample size/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. The load balance threshold ϵ was set to 0.02. The above runs were executed without the shared memory optimization.

time. We believe this is both because of efficient histogramming and node level partitioning.

With the histogramming cost so reduced, the all-to-all data exchange step is the most expensive step of the algorithm. Note that this step represents inherent cost in any sorting algorithm, since all keys need to be moved to their correct destination. All-to-all communication does not scale very well on torus networks, because communication load per link increases with number of processors, causing increased contention. It is possible that the data exchange step can be optimized by using specialized collective implementations [21, 30, 22] for all-to-all communication. We leave this for future work.

7. CONCLUSIONS

In this paper, we presented Histogram sort with sampling, 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 $\mathcal{O}(p\sqrt[k]{\log p/\epsilon})$ per round as compared to sample sort with random sampling which requires $\mathcal{O}(p \log p/\epsilon^2)$ samples. We argued that the cost of histogramming is dominated by the sampling cost in the running time and hence Histogram sort with sampling is theoretically more efficient than sample sort with random and regular sampling. We also note that most of the running time is spent in local sorting and data exchange, both of which are inherent to any sorting algorithm. The reduced sample size makes Histogram sort with sampling extremely practical for massively parallel applications, scaling to tens of thousands of processors.

8. ACKNOWLEDGMENTS

The authors are grateful to Edgar Solomonik for providing useful feedback and Omkar Thakoor for proofreading the analysis. 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/.

9. REFERENCES

[1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, et al. Parallel programming with migratable objects: charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 647–658. IEEE, 2014.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *Proceedings of the 15th annual ACM Symposium on Theory of computing*, pages 1–9. ACM, 1983.

[3] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz. Practical massively parallel sorting. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 13–23. ACM, 2015.

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

[5] A. Bhatele. 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*, page 33, 2014.

[6] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.

[7] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM Symposium on Parallel algorithms and architectures*, pages 3–16. ACM, 1991.

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[9] D. R. Cheng, A. Edelman, J. R. Gilbert, and V. Shah. A novel parallel sorting algorithm for contemporary architectures. 2006.

[10] R. Cole. Note on the AKS sorting network. *Computer Science Department Technical Report*, 243. New York University, New York,, 1988.

[11] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. mcgraw-hill. , *MIT Press, 1990.*, 16(2):91–99, 1998.

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

[14] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, et al. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2):195–212, 2005.

[15] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.

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

[17] Z. Hong and R. Sedgewick. Notes on merging networks (prelimiary version). In *Proceedings of the 14th Annual*

ACM Symposium on Theory of Computing, STOC '82, pages 296–302, New York, NY, USA, 1982. ACM.

[18] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn. Massively parallel cosmological simulations with changa. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12. IEEE, 2008.

[19] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[20] L. V. Kale and S. Krishnan. A comparison based parallel sorting algorithm. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume 3, pages 196–200. IEEE, 1993.

[21] L. V. Kale, S. Kumar, and K. Varadarajan. A framework for collective personalized communication. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2003.

[22] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj. Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer. *The International Journal of High Performance Computing Applications*, 28(4):450–464, 2014.

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

[24] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, Oct. 1993.

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

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

[27] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[28] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, Apr. 1992.

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

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

[31] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

APPENDIX

A. PROOF OF THEOREM 3.3

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,

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

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.

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

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.

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

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,

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

Note that 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 with high probability.
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,

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

This completes the proof of Theorem 3.3. $\square$