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 and Histogram sort rely on data partitioning techniques to distribute keys across processors. One major challenge of data partitioning is to achieve good load balance We present a parallel sorting algorithm that provably ensures arbitrary good load balance bounded by a factor of $(1+\epsilon)$, where ϵ is provided as an input parameter. We show that the sample size in sample sort can be reduced w.h.p. from $O(p \log N/\epsilon^2)$ to $O(p \log N/\epsilon)$ by doing one round of histogramming and can be further brought down to $O(p \log N + p/\epsilon)$ with two rounds of histogramming (N = total number of keys, p = number of processors). Thus, Histogram sort with sampling is more efficient than Sample sort algorithms that achieve the same level of load balance, both in theory and in practice, especially for massively parallel applications, scaling to tens of thousands of processors.

We also show that a fairly accurate histogram can be obtained using a relatively small sample size of $\mathcal{O}(\sqrt{p \log p})$ from 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 experiments, we exploit shared memory within nodes to make our algorithm faster and more scalable on large modern clusters.

CCS Concepts

•**Computing methodologies** → **Parallel algorithms;** *Massively parallel 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 even slight load imbalance in one processor will slow down that processor for subsequent computations. For instance, *Changa* [11], an N-body application to perform collisionless N-body cosmological simulations uses parallel sorting at the beginning of every iteration in its simulation and hence has a strict need for good load balance.

A parallel sorting algorithm needs to distribute keys across processors such that they are in a globally sorted order. A globally sorted order implies that 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 gurantees about the load balance after sorting. For our discussions, we let the application specify a parameter ϵ as a part of the input to indicate that every processor should end up with no more than $N(1+\epsilon)/p$ keys. 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 data ranges for every processor by picking $p-1$ keys, also called as splitters (since they split the data range into p parts: one for each processor). These splitters are broadcast to all processors and every key is sent to the appropriate destination. The two most popular algorithms in this regard are Sample sort and Histogram sort.

Sample sort [9] is one of the most widely deployed parallel sorting algorithms in parallel processing systems [16]. Sample sort with regular sampling has to sample $\Theta(p^2/\epsilon)$ keys overall across all processors to achieve the specified level of load balance. A central processor then decides the splitters in a way that gurantees 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. [6] 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, as the number of samples to be analysed at a central processor becomes the bottleneck.

Histogram sort [13] tries to address this by conducting multiple rounds of histogramming to determine "good" splitter keys, refining candidate splitter keys every round. To the best of our knowledge, there are no gurantees on the number of rounds of histogramming needed to determine good splitter keys, assuming nothing about the initial distribution of keys.

We present a highly scalable parallel sorting algorithm that relies on data partitioning but determines all the split-

© 2016 ACM. ISBN 123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

ters after doing just one round of histogramming, with high probability. The splitters determined this way achieve the specified level of load balance specified by ϵ. Our algorithm in its presented form works for any initial distribution of keys unless there are too many duplicates in the input. We discuss how to deal with the case when the input has a lot of duplicates in Section 5.4, without blowing up the size of the input.

We show that the total number of samples for histogramming can be brought down further to $\mathcal{O}(p\log p + p/\epsilon)$ with two rounds of histogramming. We argue that this has a drastic impact on scalability. For instance, for $p = 64K$, $\epsilon = 0.05$, $N/p = 1000000$ and 64 bit keys, the sample size is 5 GB for Sample sort with random sampling, 250 MB for Histogram sort with one round of histogramming and 22 MB for Histogram sort with two rounds of histogramming.

We demonstrate the scalability of our algorithm on large supercomputing clusters. We leverage the shared memory (SMP) paradigm to make node level optimisations to make our algorithm more scalable. The details of our experiments are described in Section 8.

In summary, the key contributions of this paper are

- We present a data partitioning algorithm which, with high probability, finishes after one round of histogramming, determining all the splitter keys to the specified level of load balance. The size of the histogram is $\mathcal{O}(p\log p/\epsilon)$. We describe this algorithm in Section 3.

- We show that our overall sorting algorithm is asymptotically superior to state of the art Sample sort algorithms that achieve the same level of load balance. Details of running time analysis is in Section 5.

- We exploit shared memory to do node level optimisations in our algorithm. Our final algorithm is a two level parallel sorting algorithm which first partitions the input data across nodes and then performs a local sort within node. This optimisation significantly reduces the size of the histogram and number of messages in the network. We describe this optimisation and its benefits in Section 7.

- We present the results of our experiments on large clusters and comparison with other algorithms. Details and conclusions of our experiments are described in Section 8.

2. PRELIMINARIES

2.1 Problem Statement

The input has N keys across p processing elements, distributed evenly such that each processor has $n = N/p$ keys. The input needs to be sorted to obtain a global sorted order. A global sorted order implies that all output keys on i^{th} processor p_i are greater than keys on processor p_j for all $j < i$. Furthermore, keys within each processor are locally sorted. Additionally for good load balance, each processor should end up with no more than $N(1+\epsilon)/p$ keys, where ϵ is specified as a part of the input. In other words, the load imbalance is bounded by a factor of $(1+\epsilon)$. Since different parallel systems tolerate varying amount of load imbalance, this model captures many practical scenarios. Cheng et. al. [7] propose an algorithm that finds exact splitters that achieve perfect load balance.

The bulk of this paper focuses on data partitioning algorithms. Sample sort by regular sampling [17, 15], histogram sort [13, 18], sample sort by random sampling [9, 6] and parallel sorting by over partitioning [14] fall into this category. A data partitioning algorithm determines $p-1$ splitter keys, that split the input into p ranges. Let the splitter keys determined by an algorithm be $\{S_1, S_2..., S_{p-1}\}$. 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. For ideal load balance, every processor should end up with N/p keys. If $R(k)$ denotes rank of key k in the input, then in such an ideal case, $R(S_i) = Ni/p$. Since some load imabalance is tolerated in the problem statement, in our algorithm, we enforce the following:

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

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 achieved if a key k has been found such that $R(k) \in \mathcal{T}_i$, i.e., if we have found a suitable candidate key k for S_i.

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 analysis does not rely on input keys to be evenly distributed across each processor.

Our algorithm naturally fits into the bulk synchronous parallel model (BSP) [19]. We analyse the cost of our algorithms using the BSP model in section 5.

Our algorithm borrows some ideas from sample sort and histogram sort. Hence, we briefly describe them before proceeding to our main result. A more detailed survey of various parallel sorting algorithms is in section 4.

2.2 Sample sort

Sample sort [9] is a widely studied and analysed parallel sorting algorithm. In Sample sort, s keys are sampled from each processor and sent to a central processor to form an overall sample of size $M = ps$ keys. Let $\{\lambda_0, \lambda_1..., \lambda_{ps-1}\}$ denote the combined sorted sample. From this combined sample, $p-1$ keys are chosen as the final splitters. Any algorithm for Sample sort has the following structure.

2.2.1 Sampling phase

Every processor samples some s keys and sends it to a central processor. s is often referred to as the oversampling ratio. We discuss and analyse different ways to obtain this sample in more detail in Section 4.1.

2.2.2 Splitter determination

The central processor receives samples of size s (obtained in Step 1) from every processor resulting in a combined sample of size $M = sp$. The central processor then selects $p-1$ splitter keys: $\{S_1, S_2..., S_{p-1}\}$ from this combined sample that splits the key range into p ranges, each range assigned to one processor. Once the splitters have been finalized, it broadcasts the $p-1$ splitters: $\{S_1, S_2..., S_{p-1}\}$ to all other processors.

2.2.3 *Data movement*

Once a processor receives the splitters from Step 2, 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 processors. Once a processor receives all the keys that it owns from other processors, it merges all the receieved data chunks. It can use any sequential sorting algorithm for this step.

Although Sample sort requires a large number of samples to provably achieve the desired amount 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 smaller sample sizes for well behaved distributions. We discuss various tradeoffs and factors to consider for Sample sort in more detail in Section 4.1.

2.3 Histogram Sort

Histogram sort [13] addresses load imbalance in Sample sort by determining the splitters more accurately. Instead of determining all splitters using one big sample, it maintains a set of candidate splitters and performs multiple rounds of histogramming, refining the candidates every round. Once all the splitters are within the given threshold range, it broadcasts its candidates as the final splitter keys. The skeletal structure of Histogram sort is as follows

- Step 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.
- Step 2: Every processor counts the number of keys in each range defined by the probe keys, thus, computing a local histogram of size M.
- Step 3: All local histograms are summed up using a reduction to obtain the global histogram at the central processor.
- Step 4: The central processor finalizes and broadcasts the splitters if all splitters have been achieved. Otherwise, it refines its probes using the histogram and broadcasts a new set of probes for next round of histogramming. Section/Algorithm*** describes an efficient technique to refine probes using the histogram.
- Step 5: Once all splitters are finalized in Step 4 and broadcasted, every processor sends its keys to the appropriate destination in exactly the same fashion as sample sort (i.e. keys in range $[S_i, S_{i+1})$ go to processor i).

Histogram sort achieves any arbitrary specified level of load balance. It is also more scalable than Sample Sort since the size of the histogram every round is usually small - of the order $O(p)$. However, it might take several rounds of histogramming before all splitters are determined within the allowed threshold , especially for uneven distributions. Nevertheless, its reliability and scalability make it a popular choice especially for large scientific computing applications.

3. HISTOGRAM SORT WITH SAMPLING

The basic skeleton of our algorithm is Histogram Sort. But instead of evenly spreading the first set of probes throughout the key range, we use sampling to determine the initial set of probes.

3.1 One round of histogramming

We perform one round of sampling for the initial choice of probes for histogramming. We show that if $\mathcal{O}(p \log p/\epsilon)$ samples are chosen randomly and independently, then with high probability the histogramming phase ends after just one round.

A splitter S_i is achieved where we have found a key k such that $R(k) \in \mathcal{T}_i$. Intuitively we should sample enough keys that the probability of no key lying in this range is really small. Thus, we motivate the following theorem.

THEOREM 1. *If $(Kp \log_e p)$ keys are sampled independently and uniformly at random across the entire input, then for any $1 \leq i < p$, the probability that $\forall k \in$ sample, $|R(k) - \frac{Ni}{p}| > \frac{N\epsilon}{p}$ is atmost $p^{-2\epsilon K}$.*

Consider a key k chosen uniformly at random across all processors. The probability Q_i that $|R(k) - \frac{Ni}{p}| \leq \frac{N\epsilon}{p}$ is given by

$$Q_i = \frac{2\frac{N}{p}\epsilon}{N} = \frac{2\epsilon}{p}$$

If M keys are chosen uniformly at random and independently across all processors, then the probability P_i that $\forall k \in$ sample, $|R(k) - \frac{Ni}{p}| > \frac{N\epsilon}{p}$ is given by,

$$P_i = (1 - Q_i)^M = \left(1 - \frac{2\epsilon}{p}\right)^M \leq e^{-\frac{2\epsilon M}{p}}$$

If we substitute $M = Kp \log_e p$, we obtain

$$P_i \;\leq\; e^{-\frac{2\epsilon K p \log_e p}{p}} = e^{-2\epsilon K \log_e p} \;=\; p^{-2\epsilon K}$$

Finally the probability P that there is some splitter i for which no key from the chosen sample qualifies for S_i is given by

$$\begin{aligned} P &= \bigcup_i P_i \\ &\leq \sum_i P_i \\ &= p \times p^{-2\epsilon K} \\ &= \frac{1}{p^{(2\epsilon K - 1)}} \end{aligned}$$

Observe that P really represents the probability that all splitters are determined after the first round of histogramming.

COROLLARY 1. *For $K = 10$ and $\epsilon = 0.05$, $P \leq 1/p$. This really means that if we allow the splitters to deviate 5% from their ideal positions and sample $(10p \log p)$ keys, then with probability $\geq 1 - 1/p$, sample would have candidate splitter keys for every splitter. Table *** represents the value of K required for various values of ϵ.*

We now describe detailed steps of our algorithm

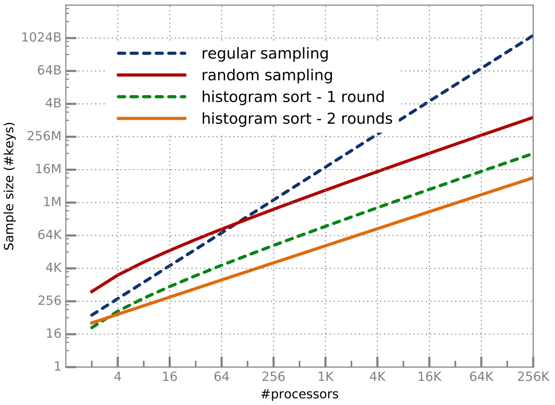

Figure 1: Samplesize: Regular sampling vs random sampling vs histogramming sample for $\epsilon = 0.05$

3.1.1 Sampling step

Every processor samples $K \log p$ samples and sends it to the central processor. The central processor receives samples of size $K \log p$ from other processors to form a combined sample of size $Kp \log p$. It then broadcasts this combined sample as the initial probe for the first round of histogramming.

3.1.2 Histogramming / Splitter determination

Every processor counts the number of keys in each range defined by the probe keys, thus, computing a local histogram of size $Kp \log p$. All local histograms are summed up using a reduction to obtain the global histogram at the central processor. With high probability, all the splitters would have been achieved and the central processor would finalize and broadcast the splitters. In the unlikely event when all splitters were not achieved, the central processor would have to refine its probes using the histogram and broadcast a new set of probes for next round of histogramming.

3.1.3 Final Data movement

Once all splitters are finalized after histogramming and broadcasted, every processor sends its keys to the appropriate destination in exactly the same way as Sample sort and Histogram sort (i.e. keys in range $[S_i, S_{i+1})$ go to processor i).

Note that our analysis assumes that all samples are taken independently and uniformly from the input data across all processors. Since all processors have the same number of keys to begin with, it's okay to sample equal number of keys from every processor. If all processors don't have equal number of keys, then a central processor will have to decide how much each processor should sample, based on the size of their local input data.

3.2 Two rounds of histogramming

We earlier showed that Histogram sort with sampling achieves the specified level of load balance, using a sample size of $(p \log N/\epsilon)$ keys with one round of histogramming. In this section, we show that doing two rounds of histogramming further reduces the size of the histogram by a factor of $1/\epsilon$. Intuitively the first round of histogramming, even with a small histogram sample gives a lot of information about the input. The second histogramming round can be conducted on an intelligently sampled set of keys that uses the information from the first histogram.

Suppose for the first round of histogramming, p/ϵ keys are randomly sampled overall across all processors. We claim that the first round will provide enough information to discard much of the $(p \log N/\epsilon)$ samples required for histogramming and keep only about ϵ fraction of the samples. Thus, the size of the second histogram is roughly, $\epsilon \times (p \log N/\epsilon) = \mathcal{O}(p \log N)$ samples.

To prove our claim, for splitter i, let $\mathcal{I}(i) = [l_i, u_i]$ where l_i and u_i denote the lower bound and upper bound for the i^{th} splitter rank respectively, obtained after first round of histogramming. Clearly if $l_i \in \mathcal{T}_i$ or $u_i \in \mathcal{T}_i$, we have already found the i^{th} splitter. For splitters that have not been achieved, we restrict our sample for the second histogram to $\mathcal{I}(i)$'s. For this purpose, we define $\mathcal{C}_i$, the cost for each $\mathcal{I}_i$ for the second round of histogramming.

$$\mathcal{C}_i = \begin{cases} 0 & (l_i \in \mathcal{T}_i) \vee (u_i \in \mathcal{T}_i) \\ u_i - l_i & \text{o/w} \end{cases}$$

$\mathcal{C}_i$ represents the length of the input for which we need to sample for the i^{th} splitter. Note that $\mathcal{C}_i = 0$ if i^{th} splitter has been already achieved since we don't need to sample for the i^{th} splitter.

Next we show that the expected length of $\sum_i \mathcal{C}_i$ is small. We have the following theorem

THEOREM 2. $E(\sum_i \mathcal{C}_i) = \mathcal{O}(N\epsilon)$

The full proof of the theorem is in Appendix A.

3.3 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 a fairly accurate histogram can be computed using an overall sample of size $\mathcal{O}(p\sqrt{p \log p})$.

Assume that for approximate histogramming, every processor samples s keys. We use a sampling technique similar to random sampling as suggested by Blelloch. et. al. [6]. 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 the sample. We use this sample to answer rank queries of the following type, given a key k find rank of k in the overall input. Denote the set of sorted sampled keys by $S = \{\lambda_1, \lambda_2 ..., \lambda_{ps}\}$. For answering rank queries, if r denotes the number of sample keys $\leq k$, we return Nr/ps.

THEOREM 3. *For* $s = \sqrt{cp \log p}/\epsilon$, *the rank returned by the above algorithm is within a distance of* $N\epsilon/p$ *from true rank of* k *w.h.p.*

Proof: Consider a processor i. Let the samples contributed by processor i be $S_i = \{\lambda_1^i, \lambda_2^i ..., \lambda_s^i\}$. Let the number of blocks in processor i that are completely less than k be b_i. Clearly, atleast b_i samples and atmost $(b_i + 1)$ samples in S_i are less than k. Let the fraction of keys in the $(b_i + 1) - th$ block that are $\leq k$ 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: $\lambda_{b_i+1}^i \leq k$ is l_i. Let

X_i be the bernoulli random variable denoting if $\lambda^i_{b_i+1} \le k$. Thus, $P(X_i = 1) = l_i$.

True rank of 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 $E(R) = E(R_k)$.

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

$$
\begin{aligned}
&P\Big(|R - R_k| > \frac{N\epsilon}{p}\Big) \\
&= P\Big(|\sum_{i=1}^{p}(b_i + X_i)\frac{N}{ps} - \sum_{i=1}^{p}(b_i + l_i)\frac{N}{ps}| > \frac{N\epsilon}{p}\Big) \\
&= P\Big(|\sum_{i}(X_i - l_i)| > s\epsilon\Big) \\
&\le 2e^{-\frac{2(s\epsilon)^2}{p}} = 2e^{-2c\log p} = 2p^{-2c}
\end{aligned}
$$

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

The above algorithm 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^{2c}. Even though, the approximation is good enough for getting splitters that achieve the specified load balance, we note that this might not be as useful for our algorithm since histogramming is done only once, with high probability. Nevertheless, it might be useful in systems where histograms need to be computed repeatedly.

4. DISCUSSION AND RELATED WORK

4.1 Sample sort: Sampling methods

In this section, we discuss various methods of doing the sampling step (Step 1) in Sample sort, described in Section 2.2.

4.1.1 Random sampling

Random Sampling leads to good load balance with a relatively reasonable number of samples, espeically for small values of p [8, 6]. For large values of p, it still might be unscalable as we show later in this paper.

Next, we consider random sampling as proposed by Blelloch et. al. [6]. If the oversampling ratio is s, each processor divides its keys into s blocks of size (N/ps) and samples a random key in each block. The splitters chosen by the central processor are $S_i = \lambda_{si}$. We reproduce Theorem $B.4$ from [6] below:

THEOREM 4. *If s be the oversampling ratio, then, for any $\alpha \ge 1 + 1/s$, 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 $s = (c\log N/\epsilon^2), \alpha = (1+\epsilon)$, where c is a constant to be determined later, the probability that no processor contains more than $\alpha = (1+\epsilon)$ keys after sorting is atmost $Ne^{-\frac{c\log N}{2(1+\epsilon)}} = e^{-\log N\left(\frac{c}{2(1+\epsilon)}-1\right)} = \left(\frac{1}{N}\right)^{\left(\frac{c}{2(1+\epsilon)}-1\right)}$.

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 $\Theta(p\log N/\epsilon^2)$ keys.

Despite good theoretical gurantees about the quality of splitters with random sampling, we found it unscalable in practice to large processors because of the large sample size, which has ϵ^2 in the denominator. Figure 3.1 illustrates the comparison between sample size of random sampling, regular sampling and the sample required for histogramming. Clearly for large processors, both regular sampling and random sampling are impractical.

4.1.2 Regular sampling

To mitigate some of the load imbalance with random sampling, regular sampling was proposed [17, 15] owing to its better analytical gurantees about load balancing. Every processor sorts its local input data and samples s evenly spaced keys. The central processor collects these samples and merges them to obtain a combined sorted sample $\{\lambda_0, \lambda_1 ..., \lambda_{ps-1}\}$ of size $M = ps$. $p-1$ splitters are then selected from this sample by selecting evenly spaced keys, $\lambda_{si-\frac{p}{2}}$ is chosen as S_i.

Previous works [?] have shown that if the oversampling ratio s is p, then each processor will end up with no more than $\lceil \frac{2N}{p} \rceil$ keys. In our model we show that $\frac{Ni}{p} - \frac{N}{2p} \le R(S_i) \le \frac{Ni}{p} + \frac{N}{2p}$. This is consistent with previous results since the number of keys on each processor is upper bounded by $\frac{2N}{p}$.

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

We now prove bounds for the quality of splitters obtained using regular sampling for a general oversampling ratio. We motivate the following theorem.

THEOREM 5. *If the oversampling ratio for Regular Sampling is s and $S_i = \lambda_{si-\frac{p}{2}}$, then $\forall i : 1 \le i < p$, $|R(S_i) - \frac{Ni}{p}| < \frac{N}{2s}$.*

Proof: The number of samples $\le S_i$ is $\left(si - \frac{p}{2}\right)$. The number of keys between two consecutive samples, on the same processor, is $\frac{N/p}{s} = \frac{N}{ps}$. Hence, the total number of keys across all processors which are $\le S_i$ is atleast $\left(si - \frac{p}{2}\right) \times \frac{N}{ps} = \frac{Ni}{p} - \frac{N}{2s}$.

Also, the number of samples $> S_i$ is $\left(s(p-i) + \frac{p}{2}\right)$. Hence the number of keys $> S_i$ across all processors is atleast $\left(s(p-i) + \frac{p}{2} - p\right) \times \frac{N}{ps} = \frac{N(p-i)}{p} + \frac{N}{2s} - \frac{N}{s} = \frac{N(p-i)}{p} - \frac{N}{2s}$. Thus, $R(S_i)$ the global rank of S_i is atmost $N - \left(\frac{N(p-i)}{p} - \frac{N}{2s}\right) = \frac{Ni}{p} + \frac{N}{2s}$.

Combining the above two bounds, for an oversampling ratio s, we obtain that $R(S_i) \in \left[\frac{Ni}{p} - \frac{N}{2s}, \frac{Ni}{p} + \frac{N}{2s}\right]$. This completes the proof of the theorem.

We note that Sample sort by regular sampling with oversampling ratio $s = p$ can still lead to poor load balance. In the above bound for $s = p$, we have that $R(S_i) \in \left[\frac{Ni}{p} - \frac{N}{2p}, \frac{Ni}{p} + \frac{N}{2p}\right]$. Thus, potentially some processor can end up with $\frac{2N}{p}$ keys in contrast to the ideal desired case of $\frac{N}{p}$ keys. This bound is indeed tight as was shown by [***], that there exists inputs for which some processor will end up with $\left(\frac{2N}{p} - \frac{N}{p^2} - p + 1\right)$ keys [10] with oversampling ratio $s = p$.

We observe that if the oversampling ratio $s = \frac{p}{\epsilon}$, then the splitters determined using regular sampling are within the required threshold specified by ϵ. Indeed, substituting $s = \frac{p}{\epsilon}$ in Theorem 5, we obtain $|R(S_i) - \frac{Ni}{p}| < \frac{N\epsilon}{2p}$. Thus we conclude that to achieve good levels of load balance specified by the input parameter ϵ with Regular Sampling, every processor should sample $\frac{p}{\epsilon}$ keys, and an overall sample of size $\frac{p^2}{\epsilon}$. Recall that all samples have to be assembled, sorted and analysed at one central processor. If p is large, $\frac{p^2}{\epsilon}$ is a rather large sample to be analysed at one central processor. For instance, if $p = 4K$ and $\epsilon = 0.05$, the central processor has to analyse about 300 million keys. For these reasons, Regular Sampling does not scale very well to a large number of processors while allowing good load balance.

4.2 Approximate median using regular sampling

Finding a median is equivalent to finding a splitter that will seperate the entire data into two equal halves. Often applications require only an approximate median. Our model of approximation of a median is similar to what we did for splitters. We require that the output median Λ of any algorithm that seeks to find an approximate median respects the following condition: $R(\Lambda) \in (\frac{N}{2} - \frac{N\epsilon}{2}, \frac{N}{2} + \frac{N\epsilon}{2})$, where ϵ, as before, is fed as an input to the algorithm. We show that with a constant oversampling ratio $s = 1/\epsilon$, regular sampling will find such a median.

THEOREM 6. *If the oversampling ratio for Regular Sampling: $s = (\frac{1}{\epsilon})$ and $\Lambda = \lambda_{\frac{p}{2\epsilon} - \frac{p}{2}}$, then $|R(\Lambda) - \frac{N}{2}| < \frac{N\epsilon}{2}$.*

Proof: The number of samples $\leq \Lambda$ is $\left(\frac{p}{2\epsilon} - \frac{p}{2}\right)$. The number of keys between two consecutive samples on the same processor is $\frac{N/p}{s} = \frac{N\epsilon}{p^{\prime\prime}}$. Hence, the total number of keys across all processors which are $\leq \Lambda$ is atleast $\left(\frac{p}{2\epsilon} - \frac{p}{2}\right) \times \frac{N\epsilon}{p} = \frac{N}{2} - \frac{N\epsilon}{2}$.

Also, the number of samples $> \Lambda$ is $\left(\frac{p}{2\epsilon} + \frac{p}{2}\right)$. Hence the number of keys $> \Lambda$ across all processors is atleast $\left(\frac{p}{2\epsilon} + \frac{p}{2} - p\right) \times \frac{N\epsilon}{p} = \frac{N}{2} - \frac{N\epsilon}{2}$. Thus $R(\Lambda)$, the global rank of Λ is atmost $N - \left(\frac{N}{2} - \frac{N\epsilon}{2}\right) = \frac{N}{2} + \frac{N\epsilon}{2}$.

4.3 Histogram sort: Refining probes

4.4 Other Sorting Algorithms

4.4.1 Bitonic Sort

Bitonic sort [3] is a merge based parallel sorting algorithm. Its theoretical properties have been extensively studied. It is not however used in large applications where $N >> p$ because of its large data movement. It moves each piece of data $\Theta(\log p)$ times. Nevertheless, it laid good foundations for research on parallel sorting algorithms, especially from a theoretical perspective. As Bitonic sort has been extensively studied, we'll not go into its analysis or experimentation. A comparative study can be found in [5].

4.4.2 Radix Sort

Radix sort [5] 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 be 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 practical issue with parallel Radix sort is that it has large data movement since in every step, there is an All-to-all exchange. Morever, since it uses bit representations, it is not suitable for sorting non-integer type keys like floating points or strings, for instance.

5. RUNNING TIMES

5.1 Sample sort with regular sampling

For sample sort with regular sampling, this means a sample size of $\mathcal{O}(p^2/\epsilon)$. The time for initial local sorting by every processor is $\mathcal{O}\left(\frac{N}{p} \log \frac{N}{p}\right)$. The time for the sampling step where all samples are assembled at one central processor and merged is $\mathcal{O}(p^2 \log p/\epsilon)$. Finally, the time for the final step where all keys are sent to appropriate destinations is $\mathcal{O}(N \log p/p)$. The destination is decided by doing a binary search over all the splitters to determine the right range for the key. Thus, the overall running time of sample sort with regular sampling is

$$\mathcal{O}\Big(\frac{N}{p} \log \frac{N}{p} + \frac{p^2}{\epsilon} \log p + \frac{N}{p} \log p\Big)$$

5.2 Sample sort with random sampling

Sample sort with random sampling requires a sample of size $\mathcal{O}(p \log p/\epsilon^2)$. Note that the algorithm still requires a local sorting before sampling. As with regular sampling, the time for initial local sorting by every processor is $\mathcal{O}\left(\frac{N}{p} \log \frac{N}{p}\right)$. The time for the sampling step is $\mathcal{O}(p \log^2 p/\epsilon^2)$. Finally, the time for the final step where all keys are sent to appropriate destinations is $\mathcal{O}(N \log p/p)$. Thus, the overall running time of sample sort with random sampling is

$$\mathcal{O}\Big(\frac{N}{p} \log \frac{N}{p} + \frac{p \log^2 p}{\epsilon^2} + \frac{N}{p} \log p\Big)$$

5.3 Histogram sort with sampling

5.3.1 One round of histogramming

With one round of histogramming, the sample size required is $\mathcal{O}(p \log p/\epsilon)$. The samples have to be broadcast to all the processors for histogramming and a histogramming reduction needs to take place. The extra histogramming step, which is absent in Sample sort results in an execution time overhead of $\mathcal{O}\big((N \log \frac{p}{\epsilon})/p\big)$. This overhead is however, hidden because of other dominant factors in the running time. Note that initial local sorting is not required initially because of random sampling. The time for the sampling step is $\mathcal{O}(p \log^2 p/\epsilon)$. Finally, the time for the final data movemement step is $\mathcal{O}(N \log p/p)$. Overall running time, thus, is

$$\mathcal{O}\Big(\frac{p\log^2 p}{\epsilon} + \frac{N}{p}\log\frac{p}{\epsilon} + \frac{N}{p}\log p\Big)$$

5.3.2 *Two rounds of histogramming*

The sample size required with two rounds of histogramming is $\mathcal{O}(p/\epsilon)$ for the first round and $\mathcal{O}(p\log p)$ for the second round. The overall running time, thus , is

$$\mathcal{O}\Big(\frac{p\log p}{\epsilon} + p\log^2 p + \frac{N}{p}\log\frac{p}{\epsilon} + \frac{N}{p}\log p\Big)$$

The sampling step in Sample sort with random sampling takes $\mathcal{O}(p\log^2 p/\epsilon^2)$ time, in contrast to $\mathcal{O}(p\log^2 p/\epsilon)$ time with one round of histogramming and $\mathcal{O}(p\log p/\epsilon + p\log^2 p)$ with two rounds of histogramming. For large p, the sampling term dominates the running time of the algorithm. In our experiments we found out that the extra ϵ factor for the sampling step is what makes Sample sort impractical for large values of p. We thus conclude that running time of our algorithm is asymptotically superior than Sample sort with random and regular sampling for the same level of load balance.

5.4 Dealing with duplicates

Previous works [17] 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. [10] 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.

Our key idea to deal with a large number of duplicates in the input stems from the following observation. Any algorithm which uses sampling techniques depends only on the relative ordering of the keys. For a large number of duplicates we enforce an implicit 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. This does not blow the input data, but it does increase the size of the histogram by a constant factor since probe keys in the histogram will have to explicitly tagged. Figure 2 shows how the input data can be implicitly tagged to enforce strict ordering. A similar scheme was used to deal with duplicates in [2].

6. DATA MOVEMENT

7. HIERARCHICAL SORTING

The All-to-all communication phase in the last step of the algorithm results in $p(p-1)$ messages in the network. This is particularly large for greater values of p. To mitigate this, we leverage Shared Memory (SMP) Paradigm. In a supercomputing cluster like ***, multiple processors lie in a single physical node. Hence, its superfluous to send multiple messages between a pair of nodes. Instead, we can easily combine all messages going to the same node into one larger message. This opens door for other optimisations too. The data partitioning now really needs to be across nodes and not across individual processors. Data partitioning across nodes reduces the size of the initial histogram considerably, since p now is the number of physical nodes instead of the number of individual processors. For instance BlueGene/Q system [1] has 16 cores in a single physical node. Thus, if we were to do data partitioning across nodes on $8K$ nodes, the size of the initial histogram would approximately be 12 MB. The number of processor cores on $8K$ nodes are $8K \times 16 = 128K$ and thus, to do data partitioning across processors, the size of the initial histogram would be 250 MB. Clearly, this optimisation makes the histogramming step much more scalable, in addition to reducing the number of messages in the network. Once a node receives all the data after the partitioning step, it can locally sort the data within node without injecting traffic on the network.

8. EXPERIMENTS

In this section, we present our experiments on large supercomputing clusters - Stampede and ***. We show good scalable performance for weak scaling and compare our algorithm to the histogram sort algorithm in [18]. As described earlier, we take advantage of shared memory programming to decrease the number of messages in the network. We believe this is another critical reasons for better performace (other than efficient histogramming). Both of these algorithms were implemented in Charm++ [12]. The programming 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 nice visualization software [4] for analysing performance, timelines, cpu usage, memory footprint etc. and getting useful insights into the algorithm.

We benchmarked our algorithm for weak scaling with 8 million keys per processor. We set the load balance threshold ϵ to 0.05. We generated random input by peforming bitwise and operations between random numbers to generate skewed distributions. We observe that the histogramming phase in Histogram sort gets affected as the initial distribution becomes more skewed.

9. CONCLUSIONS

In this paper we made the case that for large number of processors, Sample sort results in a large number of samples, so much so, that it renders the algorithm impractical for massively parallel applications. We showed that by doing one round of histogramming, the sample size can be reduced significantly. More specifically if the application requires that every processor should have $N(1+\epsilon)/p$ keys after sorting, the sample size with histogramming is smaller by a factor of $\mathcal{O}(1/\epsilon)$. Even for moderate values of ϵ like 0.05, this comes out to be 20×. We showed that Histogram sort with sampling is both theoretically efficient and extremely practical for massively parallel applications, scaling to tens of thousands of processors.

10. ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu

11. REFERENCES

[1] 2016.

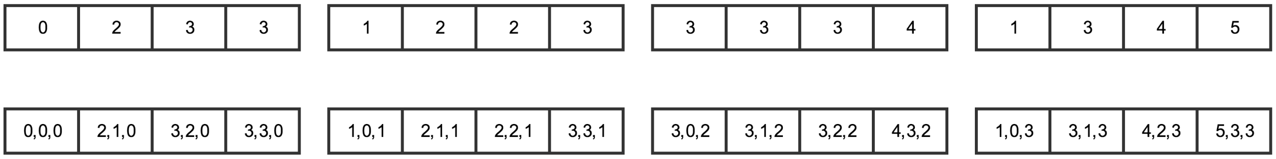

Figure 2: Impicit tagging to deal with large number of duplicates, $k - - > (k, index, PE)$

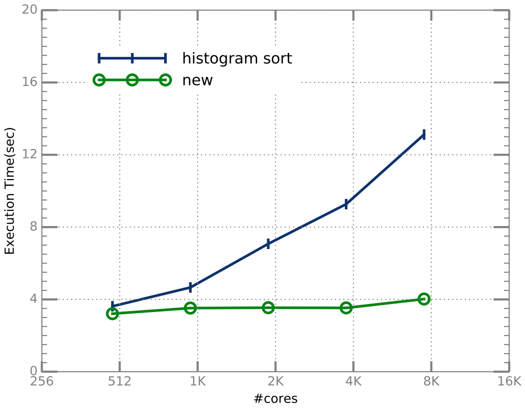

Figure 3: Stampede: Histogram sort vs new algorithm with fast histogramming and SMP optimisations. Each processor had 8 **million** 64**-bit long integer type keys. Probably make it the first figure**

[2] 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.

[3] 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.

[4] A. Bhatele. 5.2 projections: Scalable performance analysis and visualization. *Connecting Performance Analysis and Visualization to Advance Extreme Scale Computing*, page 33, 2014.

[5] 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.

[6] 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.

[7] D. R. Cheng, A. Edelman, J. R. Gilbert, and V. Shah. A novel parallel sorting algorithm for contemporary architectures. *Submitted to ALENEX*, 2006.

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Parallel and distributed information systems, 1991., proceedings of the first international conference on*, pages 280–291. IEEE, 1991.

[9] 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.

[10] 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.

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

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

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

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

[15] 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, 1993.

[16] O. OâĂŹMalley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pages 1–3, 2008.

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

[18] E. Solomonik and L. V. Kale. Highly scalable parallel sorting. In *IPDPS*, pages 1–12, 2010.

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

APPENDIX

A. PROOF OF THEOREM 2

To prove the theorem, we first bound $E(\mathcal{C}_i)$. We write $\mathcal{C}_i$ as a sum of N indicator random variables X_i^j, i.e. we have $\mathcal{C}_i = \sum_j X_i^j$, where

$$X_i^j = \begin{cases} 1 & j \in \mathcal{I}_i \wedge (\text{Splitter } i \text{ is not achieved}) \\ 0 & otherwise \end{cases}$$

$$P(X_i^j = 1) = \begin{cases} \big(1 - \frac{\epsilon}{p}\big)^{\frac{p}{\epsilon}} & j \in \big(\frac{Ni}{p} - \frac{N\epsilon}{p}, \frac{Ni}{p} - \frac{N\epsilon}{p}\big) \\ \big(1 - \frac{(\frac{N\epsilon}{2p} + |j - \frac{Ni}{p}|)}{N}\big)^{\frac{p}{\epsilon}} & otherwise \end{cases}$$

Using the identity $(1-x) < e^{-x}$, $\big(1 - \frac{\epsilon}{p}\big)^{\frac{p}{\epsilon}} \leq e^{-\frac{\epsilon}{p}\frac{p}{\epsilon}} \leq \frac{1}{e}$.

Also,

$$\begin{aligned} \Big(1 - \frac{(\frac{N\epsilon}{2p} + |j - \frac{Ni}{p}|)}{N}\Big)^{\frac{p}{\epsilon}} &\leq e^{-\frac{\left(\frac{N\epsilon}{2p} + |j - \frac{Ni}{p}|\right)p}{N\epsilon}} \\ &= e^{-\frac{1}{2}} e^{-\frac{|j - \frac{Ni}{p}|}{N\epsilon/p}} \end{aligned}$$

$$P(X_i^j = 1) \leq \begin{cases} 1/e & j \in \big(\frac{Ni}{p} - \frac{N\epsilon}{p}, \frac{Ni}{p} - \frac{N\epsilon}{p}\big) \\ e^{-\frac{1}{2}} e^{-\frac{|j - \frac{Ni}{p}|}{N\epsilon/p}} & \text{otherwise} \end{cases}$$

$$\begin{aligned} E(\mathcal{C}_i) &= \sum_{j=0}^{N} E(X_i^j) \\ &= \sum_{j=0}^{\frac{Ni}{p} - \frac{N\epsilon}{p}} E(X_i^j) + \sum_{j=\frac{Ni}{p} - \frac{N\epsilon}{p}}^{\frac{Ni}{p} + \frac{N\epsilon}{p}} E(X_i^j) + \sum_{j=\frac{Ni}{p} + \frac{N\epsilon}{p}}^{N} E(X_i^j) \\ &\leq \sum_{j=\frac{N\epsilon}{p}}^{\frac{Ni}{p}} E(X_i^{\frac{Ni}{p} - j}) + \frac{2N\epsilon}{pe} + \sum_{j=\frac{N\epsilon}{p}}^{N - \frac{Ni}{p}} E(X_i^{\frac{Ni}{p} + j}) \\ &\leq \sum_{j=\frac{N\epsilon}{p}}^{\frac{Ni}{p}} e^{-\frac{1}{2}} e^{-\frac{jp}{N\epsilon}} + \frac{2N\epsilon}{pe} + \sum_{j=\frac{N\epsilon}{p}}^{N - \frac{Ni}{p}} e^{-\frac{1}{2}} e^{-\frac{jp}{N\epsilon}} \\ &\leq \frac{2N\epsilon}{pe} + 2 \sum_{j=\frac{N\epsilon}{p}}^{\infty} e^{-\frac{1}{2}} e^{-\frac{jp}{N\epsilon}} \\ &\leq \frac{2N\epsilon}{pe} + 2e^{-\frac{3}{2}} \frac{1}{(1 - e^{-\frac{p}{N\epsilon}})} \\ &\leq \frac{2N\epsilon}{pe} + 2e^{-\frac{3}{2}} \frac{e^{\frac{p}{N\epsilon}}}{(e^{\frac{p}{N\epsilon}} - 1)} \\ &\leq \frac{2N\epsilon}{pe} + 2e^{-\frac{3}{2}} \frac{e}{p/N\epsilon} \\ &= \mathcal{O}(\frac{N\epsilon}{p}) \end{aligned}$$

The last deduction uses the identity $(e^x - 1) \geq x$ and $e^{\frac{p}{N\epsilon}} \leq e$ if $\frac{N\epsilon}{p} \geq 1$, which is true for our case.

Using the above result

$$E(\sum_i \mathcal{C}_i) = \sum_i E(\mathcal{C}_i) \leq p\ \mathcal{O}(\frac{N\epsilon}{p}) = \mathcal{O}(N\epsilon)$$