Histogram Sort with Sampling: A review

Megha Agarwal
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
meghaagarwal@cmail.carleton.ca

December 19, 2021

Abstract

Sorting poses as one of the most important areas of study in computer science owing to its application in a wide variety of applications. This paper demonstrates a state-of-the-art sorting algorithm which is efficient, accurate, and does not involve a lot of data movement. Histogram Sort with Sampling[11] is an amalgamation of two sorting algorithms by picking the best of both and balancing out any drawbacks. This has been compared with two algorithms: HykSort[22] and AMS-sort[3] to demonstrate the margin by which it excels.

1 Introduction

For any computer application today, most actions boil down to mere database access that can help classify the data into a category. For example, with data analysis, a clear picture of the data helps reach a conclusion, this becomes easy when a data is already sorted. There has been great research on sorting algorithms which lead to algorithms like quick sort, merge sort, sample sort[10], etc. This was until the focus was majorly on sequential processing, but eventually, parallel processing was introduced along with its many benefits especially the drop in the amount of time a computation would take. And sorting was not far from this either. With the advent of parallelism in all domains, research included parallelism of sorting algorithms, and this produced great results.

The primary goal of a parallel sorting algorithm is to partition the keys such that all N keys are partitioned over the p processors in a globally sorted order, that is, no key on processor k is greater than any key on processor k+1. This implies that the load on each processor must be efficiently balanced. Splitting in parallel sort is used to determine the number of elements on each processor that need to be sorted. Now, for an input sequence of N elements to be processed on p processors, an exact splitting would be if each processor houses N/p elements and an approximate splitting is observed when each processor contains no more than $N(1+\epsilon)/p$ elements, for any value of ϵ. The combination of histogramming and sampling prior to redistribution of keys minimizes data movement when sorting. Sampling facilitates the partitioning using a representative set of keys while histogramming helps by iterating over and evaluating the given partition numerous times. A natural way to determine global partition is by deriving approximate splitters from targeted ideal splitters. Sample sort, which works by selecting a random sample of data and partitioning the data using these samples, can be used to achieve a deterministic balanced splitting. Histogram

sort[14] is a comparison based sorting algorithm that works on refining the splitters by repeatedly creating histograms of the number of keys included in each set of the latest splitter intervals. It is used to manage the load imbalance that may be incurred by the determination of splitters and data exchange The chief idea of this algorithm is derived from splitter intervals which are subranges about the ideal splitter keys that will produce a globally sorted data. Each round of HSS consists of uniformly sampling the keys in each splitter interval and then shrinking the interval to converge to the ideal splitters.

With the use of repeated histogramming combined with sampling, this algorithm is proved to be robust to distributions with repeated keys and also effective in situations where the data may be partially sorted. The implementation for this algorithm is performed using Charm++, which is C++ based parallel-programming framework. The theoretical analysis of this algorithm comprising of the execution cost in parallel, and performance evaluation when compared to the algorithms that were just as good, prove HSS to be an algorithm that can be chosen over the rest when working in a parallel environment. To give an overview of the structure of this paper, Section 2 describes the previous work that helps derive this algorithm, Section 3 highlights the problem that this algorithm address followed by the description of the actual algorithm in Section 4. The evaluated result of the performance of the algorithm under various scenarios is mentioned in the Section 5. Finally, Section 6 provides the conclusion and wraps up the paper.

2 Literature Review

The primary goal for any parallel sorting algorithm is to find a global partition. In such partition-based algorithms, like quick sort, it is highly beneficial to partition the data before redistributing it, in order to lower the communication cost. Two very efficient parallel sorting algorithms, Parallel Quick Sorting Algorithm (PQSA) and Merging Subarrays from Quick Sorting Algorithm (MSQSA), are proposed by Lingxiao Zeng in [25] but they perform exceptionally with a complexity of $\frac{N}{p}O\left(log\frac{N}{p}\right)$ with shared memory. However, the focus of this project is over distributed memory parallel computers. The end result achieved in a partition-based sort algorithm is a set of splitters that achieve a globally or locally balanced splitting, after which they redistribute the keys. This generally occurs in a number of stages where the data is split into number of processors and then sorted recursively.

2.1 Sample Sort

The Sample Sort algorithm[10] along with its variants[8], works as the building blocks of this algorithm. It is a comparison based algorithm that has been worked on for a long time now. The algorithm works by partitioning the data into multiple sets that can easily be represented by a subset of keys. Each processor samples s keys, that are then sent to the central processor to form an overall of ps keys. These keys when received at central processor, are split into p ranges, determined by *p-1* keys. The sampling method can either be random[4] or regular[18] when determining the sample at each processor. Regular sampling[18] is when every processor selects evenly spaced s keys to be the samples and then selects splitters from these samples at the central processor. Random sampling[4] on the other hand, is when the processor divides the data into blocks of size N/ps and then randomly picks a key from each block. Here, s is called the oversampling ratio. When in practical use, random sampling has proven to be more efficient but both methods do not

perform ideally with very large samples and there is a slight hinderance in the scalability. So, how Sample Sort works is by undergoing the following steps:

1. **Sampling Phase**: Each processor samples s keys and sends it over to the central processor resulting in a total of ps keys at the central processor, where s is called the oversampling ratio.

2. **Splitter determination**: The central processor then sorts these samples and identifies *p-1* splitters resulting in p partitions. These p-1 keys are then broadcast to all the processor in order to exchange the data such that all keys on processor i are lesser than all keys on processor *i+1* and the range of values that are stored on each processor is determined the splitter keys.

3. **Exchanging data**: Once splitter keys are received, all processors exchange their data in order to send them to their appropriate processors. This step requires an all-to-all communication between processors. Once all the data is transferred to their respective destinations, they are then locally sorted using any shared memory sorting algorithm[25] such that the data on each processor is sorted.

These steps do result in a globally sorted data. However, the chances of observing a load imbalance are high. So, to overcome that issue, ideas from Histogram Sort[14, 16] are used. Further advancement on Sample Sort has resulted in probabilistic partitioning which does not pick splitters after just one round of sampling, rather, it maintains a vector of splitter candidates and iteratively refines it until the load balance achieved on all processors is satisfactory[16].

2.2 Histogram Sort

This is a comparison based algorithm that is responsible for managing the load balancing on each processor by maintaining a set of potential splitter keys and refining the splitter interval by observing an histogram. This histogram shows the number of keys that fall within the ranges defined by each of the candidate splitters over multiple iterations. Once a satisfactory histogram is met, such that all processors have a nearly equal number of keys, the splitters that lead to this partition are then selected as the final splitter keys. A distributed histogram sort takes place in 4 basic steps:

1. **Local Sorting**: Each processor sorts its local data using any shared memory sorting algorithm which has an expected time complexity of *O(n log n)*.

2. **Splitting**: Generalizing the distributed selection algorithms, like median of partition strategy with weighted medians, to distributed multi-selection, the local array is partitioned into P subsequences [16, 20]. Rather than determining one pivot, multiple pivots are selected in each iteration, for each active range. When a pivot matching a specific rank is found, that range is discarded and pivots for all other ranges is examined from each of the two subranges. Here, each splitter is represented as a tuple containing the splitter upper and lower bounds and the splitter value. So, any splitter S_i is successfully determined if the lower bound L_i and upper bound U_i satisfy the condition $L_i(S_i) < K_{i+1} \leq U_i(S_i)$, where K denotes the rank of splitter and i ϵ {1,2,...,P}. It is here that histogramming is performed, as mentioned in [16]. All splitters are initialized with a minimum and maximum for the global range. Then,

the splitters are iteratively determined by converging the minimum and maximum for each local histogram which is combined to form a global histogram. The complexity for this is $O(p(logp) + p(logn_i))$ per iteration.

3. **Data exchange**: All processors exchange their locally sorted sequences with all other processors after determining all splitters. A permutation matrix is created where that helps identify the send displacements for each processor i [16].

4. **Local Merge**: Finally, each processor locally merges the received sorted sequence. There are two options possible here, either to sort the full array with a complexity of $O\left(\frac{N}{p}log\frac{N}{p}\right)$ using a fast shared memory algorithm or merge it out of place in $O\left(\frac{N}{p}logp\right)$ time using a binary merge algorithm [16].

2.3 Other Partition-based sorting algorithms

AMS-sort[3] is another algorithm that works on the principal of histogramming and sampling. The splitter determination done with one round of histogramming in AMS-sort is observed to perform better than HSS with one round of histogramming by a significant amount, $\Theta(min(log\ p, 1/\epsilon))$. But, this scanning algorithm is not easily generalizable to multiple stages of histogramming whereas HSS can be extended to multiple stages without much overhead. Also, the result obtained by HSS is seen to be globally balanced partitions as compared to AMS-sort which produces a locally balanced splitting. When AMS-sort is performed in a multistage manner, it can be done by a repetitive process of splitting and data exchange over process that are decreasing in number. The scanning process in AMS-sort works by deciding the splitters once the histogram is obtained. The system is to scan through the global histogram and assign a maximum possible consecutive buckets to one processor while considering the limit on the number of elements it can store. So, if buckets up to x are assigned to the first y processors, then buckets *x+1*,..., *x+z* are assigned to processor *y+1*. Quicksort algorithm adapted for hypercube by Wagar et al.[23] works by identifying a pivot which divides the cube into two sub-cubes. Both these partitions exchange data with each other based on the pivot and then merge the received data into local data portion. This results in a globally sorted sequence after a recursion depth of log(p). HykSort[22] as proposed by Sundar *et al.* is also a state-of-the-art parallel sorting algorithm, derived from this idea, which works on the idea of splitting and data exchange at multiple levels to attain a scalable algorithm. It is a derivation of quick sort for hypercube[23] and sample sort which also uses histogramming and sampling principals to sort the data in parallel. However, there is a significant difference between the sampling method for HSS and HykSort which results in HykSort requiring at least $\Omega(log\,p/log^2\,log\,p)$ samples more as compared to HSS, which showcases a slower convergence to final splitter values.

3 Problem Statement

A parallel sorting algorithm chiefly aims at distributing all the keys on the available processors in an ideal manner such that once the processing is done, all keys are in a globally sorted manner, and the processors are appropriately load balanced. So, given a sequence of N input elements, these must be partitioned into p ranges which is contained by p processors, such that all keys from processor i are less than all keys on processor *i+1*. The

elements of the input sequence are aimed to be rearranged to a globally sorted array R such that if A(i)=R(j), then the key has rank j. In order to achieve a global sorting, the splitters that partition the data must be ideal, such that each splitter $S(i) = R(\chi_i)$, where $R(\chi_i)\ \epsilon\ \tau_i$ called the target range. Besides identifying ideal splitters, it is also essential for an efficient parallel sorting algorithm to maintain the load-balance at all times, that is, that no processor owns more than the approximate partition amount. From the results obtained in [6], it can be understood that for any type of load balancing, an exact spliting can be attained by some post-processing, which under a globally balanced distribution would take only $N\epsilon/p$ steps. Ideally, the final histogram obtained must show an about equal number of elements of each processor partitioned by each splitter. As can be seen from figure 1, each interval contains a near equal number of elements.

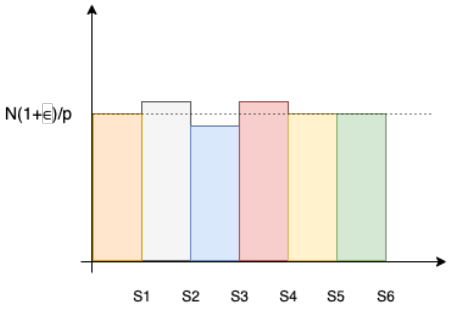

Figure 1: Ideal global approximate sort

4 Proposed Solution

There has been a good amount of study on the parallel sorting algorithms in the PRAM model, which shows best possible solution that can achieved takes O(log n/log^2 n)[7, 9]. However, this does not apply for distributed memory machines. So, for a distributed memory machine, the median of median algorithm[5] and its variants, median of partition[2], weighted median[21] are used. Here, each processor figures out the median of its local partition and broadcasts it. Each median is weighted by its partition's size. After each processor performs a three-way partition around the weighted median to determine the subrange to recurse in, the subset shrinks with each successive step. Taking into account the computation complexity of the select and the partition operations in this algorithm, further study suggests to include additional optimizations which are that the selection of pivot is better when using sampling[19] and another one is to split the group of p processors into subgroups of O($\sqrt{p(i)}$) for the ith phase to result in a better complexity[17].
From the results obtained in all previous studies, Histogram Sort with Sampling[11] works on the basic framework of the histogram sort, while also implementing a sampling phase which will help identify the candidate probes for splitter determination. The central idea of the algorithm is to sample elements at each processor, histogram over them, and adjust the selected splitters.
For an input sequence A, with N elements to be processed over P processors such that all elements are in a globally sorted order, is the main aim of this algorithm. Although HSS with one round of histogramming is not as efficient as AMS-sort with one round of histogramming, it can be easily generalized to multiple rounds resulting in a lower compu-

tation and communication overhead, as a constant number of rounds of histogramming are enough to get an asymptotic improvement. Recall that a satisfactory i^{th} splitter is the one that falls in the target range $\tau_i = [Ni/p - N\epsilon/2p, Ni/p + N\epsilon/2p]$. In a situation where for all target ranges there is at least one key in each range, our final splitters would have been found. From Theorem 4.2 [11], it can be seen that with only one round of histogramming and using a sample size of $O(p\,log(p)/\epsilon)$, a load balance of $(1+\epsilon)$ can be achieved with high probability. The algorithm takes place in 4 major steps:

Local Sort: This is the initial step where the data on each processor is sorted using any shared memory sorting algorithm that efficiently costs O(n log n), like the quicksort algorithm [24].

Splitter Determination: To split the data into P ranges to then be transferred to respective processors. The histogramming and sampling methods are used to help reach the ideal splitters.

Data Exchange: Once splitters are identified for a round, the data in each range is then moved to each processor respectively. Doing this iteratively will place the elements in a globally sorted order such that a processor i has all elements that are smaller than all elements at processor *i+1*

Local Merge: Finally, when data arrives on each processor from all other processors, they are merged together in a format that is locally sorted.

HSS can be made more efficient compared to other state-of-the-art algorithms by using repeated rounds of histogramming and sampling. This branches from the observation that the samples for each round of histogramming can be obtained from the previous rounds, except for the first round. So, the detailed explanation of each step that make up the procedure is as follows:

Splitter Determination: It is during this step that the partitions are defined making it the most important step to achieve global sorting. Both histogramming and sampling take place in this step, but out of both of them, sampling is performed first. For sampling, initially the entire input is considered. However, with subsequent steps, the data to sample over keeps shrinking. All keys in the sample phase are picked with an equal probability of ps_1/N, where s_1 is called the sampling ratio for the first round. Here, regular sampling [18] has been used, where the processor picks evenly spaced keys because it is practically more efficient owing to the ability to attain global balancing [12]. Rather than determining one pivot, as in a sequential sort, we determine multiple for each range that has not found its ideal splitter key from the input sample. If a pivot matching its specific rank is found, that range is not further considered, otherwise the data within that range is sampled to locate the splitter that best partitions it. Once samples have been determined they are collected at the central processor and broadcast to all other processors as probes for the first round of histogramming. When these samples are received at each processor, a local histogram is constructed by counting the number of keys that fall between each splitter range. These local histograms are then summed up to result in a global histogram that helps narrow the interval converging to the ideal splitters. Now, for a splitter in each active range, a lower bound($L_j(i)$) and upper bound($U_j(i)$) is maintained for the j_{th} round of histogramming where $L_j(i)$ denotes the largest key smaller than the ideal splitter i and $U_j(i)$ denotes the smallest key greater than the i^{th} ideal splitter. Here, an ideal i^{th} splitter is calculated using

Ni/p. Each splitter is stored as a tuple of three values namely, the the splitter itself, its lower bound and its upper bound. It is not complicated to determine the values of the lower and bounds for every splitter as the data on each processor is locally sorted. So just using a binary search on processor i, the bounds can be identified. t

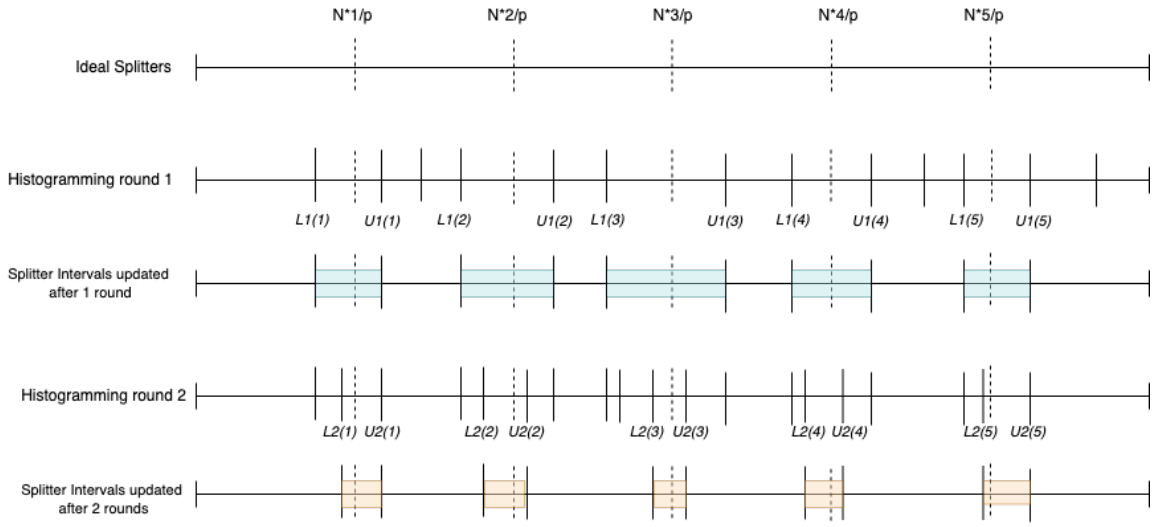

Figure 2: HSS with multiple rounds of histogramming[11]

Once the lower and upper bounds are identified, the splitter interval is updated by the keys that define the lower and upper bound values, restricting the data to be sampled in the next round by $Range_{j+1}(i) = I(L_j(i)), I(U_j(i))$. These updated intervals are then used in the next round as the input to sample and further converge the splitter interval towards ideal. After all processors have received the updated intervals, they begin the next round, $j + 1^{th}$ round, of sampling. For k rounds of histogramming, if the sample size per round is $O(p\sqrt[k]{log\, p/\epsilon})$, then HSS achieves a load balance of $(1 + \epsilon)$ for a large enough p^2 with high probability [11]. Further, in order to minimize the value of the number of rounds, k, we derivate the sample size and equate it with zero, to get a minimized k as $log\frac{log\, p}{\epsilon}$. Moving forward with the algorithm, if the value of $j < k$, then the next round of histogramming begins and process continues iteratively. However, if j=k, then the algorithm moves forward by determining the final splitters as the keys closest to each Ni/p among all the keys seen so far.

Data Exchange: After the splitters have been determined, each processor performs a MPI All-to-All communication where they exchange the data according to the splitters identified such that each processor has data bounded by the splitter intervals. As a total of p-1 splitters are defined the data can be partitioned into p sections. Each i^{th} splitter is assigned to the i^{th} processor and keys are then transferred to the target processor to satisfy the condition of globally sorted data.

Local Merge: Once data is received at each processor, it also must be merged in a locally sorted order. Now this can be done either of the two ways, sort the full array after it has been received or sort place the elements as received in a sorted order. So, this can be done either using a tournament tree [15] or using a binary merge algorithm that takes O((N/p) log p) time. Here, for a binary merge, it can start the merging as soon as data starts arriving but for the tournament tree, all data needs to be present before any processing can start.

These steps help attain a state of data on all processors that is globally sorted. The ideal situation would be if the data is near sorted. Then the splitters could be easily determined. On the contrary, the worst case would arise when there are multiple keys that have the same value. This would lead to the size of one particular bucket being way bigger than others, that is, one processor would contain a lot more elements than proposed by approximate splitting. In order to tackle this issue, each such key is labelled with 3 values, the value of the element, the processor on which the key is located, and the index value which denotes the index of the value in its local data structure. The evaluation of how it performs follows in the next section.

5 Experimental Evaluation

The paper which proposed the HSS algorithm [11] describes the computation and communication costs incurred at multiple points during the algorithm including the local sort, data exchange, etc. which are reviewed in this section followed by the experimental analysis for the worst case situation where a huge fraction of keys have the same value.

Local Sort: The cost at which the local sorting takes place is O($(N/p)log\,(N/p)$). Once this done, the splitters are broadcast to the central processor, which takes O(p) time. As there is no communication taking place in the local sorting, there is no cost for that. After the final splitters are determined, all data needs to be shuffled to reach their target processor, which incurs a cost of O(N/p). And finally, the local merge step takes O($(N/p)log\,p$) time for computation.

Sampling: Determining a sample on one processor having size S takes O(S) time. And sorting the collected samples on the central processor takes O($S\ log\ p$) time if random sampling is used which collects S/p samples from each processor.

Histogramming: To create a local histogram of size S, the time taken is O($S\,log(N/p)$). Computing the global histogram from the local takes O(S) time which is an aggregate of 2 major steps, all-gather and reduce-scatter. And broadcasting the probes and splitter intervals to every processor takes O(S) steps.All these computation and communication cots indicate that the time complexity of HSS is highly dependent on the sample size.
For multiple stage HSS, the histogramming, sampling and data exchange takes O(1) BSP super steps, so the total time is calculated by just multiplying this by the number of stages, l. And the overall computation time of the local sort is calculated to be O($(N\,log\,N)/p$).

To analyze the performance of this algorithm in the worst-case situation where a huge fraction of keys have the same value, Charm++ [1, 13] framework supported by C++ was used. This enables the program to create numerous virtual processors called *chares* which can be assigned to a node or core. The program runs the algorithm in three steps: local sorting and merging, splitter determination using histogramming and sampling, and finally the data exchange. The results from [11] display why the splitter determination in HSS is better than AMS-sort [3] and HykSort [22] by depicting faster convergence to ideal splitters in HSS. It is also observed that HSS samples according to the interval, that is, it samples more keys for a larger interval and lesser keys for a smaller interval. However, that is not the case with HykSort. So, in order to analyze how the algorithm performs for when there are many keys with the same value, the algorithm was tested for the kind of dataset by giving such numerical data values as input, which are a combination of random values and a number of keys with the same value. These values were then randomly shuffled and stored to pass as input to the main algorithm. To evaluate the effect of number of processors on

this dataset, the experiment was conducted on 3 clusters with 4,8, and 16 cores each. To handle the duplicate values, the algorithm works by tagging them with 3 variables, that is, the key, the processor that the key resides on and the index in the local array. So now when such data causes load imbalance, the processor and index values help determine the number of keys that are supposed to be moved to another processors. On a much smaller scale, if 30 out of 100 keys have the value 7 and these are to be processed using 10 processors then in an ideal situation each processor must contain 10 keys to sort. but there will be one splitter that places all the 30 keys onto one processor. However, this gives rise to a load imbalance. To help overcome this, the processor and index values will be used. First the processor value is checked to identify how many of the keys are on one processor, and which processor they are on. Then using the index key, it is determined whether the initial few keys can be moved from the i^{th} processor to $i - 1^{th}$ processor or the later keys are to be moved to the $i + 1^{th}$ processor, and in case of this example, moved to both, the previous and the next processors. Evaluating the results obtained, as shown in Figure 3, it can be seen that the

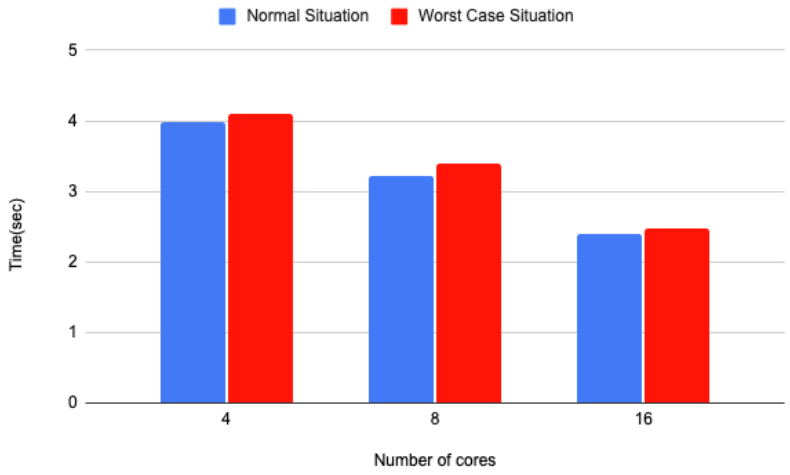

Figure 3: Worst case situation evaluation for HSS

time required to sort the data shows a decrease with the increase in number of processors, which was expected as the overall time complexity is inversely proportional to the number of processor cores. And it also demonstrates the difference between the time taken for the worst case situation and the dataset with all random values. Here, the value of ϵ is taken to be 0.1 denoting a 10% imbalance. Also, the size of dataset is 1000 values, and for the worst case situation, 350 keys had the same value. A few more experiments were conducted with varying values, but the resulting time obtained was similar to the results shown in the chart. There does not seem to be much difference in the performance for both situations. The small amount of additional time may be arising due to the additional steps to balance the load and movement of the respective data.

6 Conclusions

Histogram Sort with Sampling works as an aggregation of the histogram sort and the sample sort providing the best of both algorithms by sampling and managing the load imbalance. From the initial work on the algorithm and the work mentioned in this paper, HSS seems to

be a robust partition-based sorting algorithm that works efficiently in theory and in practice equally providing approximate and exact splitting.The reiteration mechanism to shrink the splitter interval in order to reach the keys closest to ideal splitters is what helps make this algorithm better than others.

References

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, et al. Parallel programming with migratable objects: Charm++ in practice. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658. IEEE, 2014.

[2] Ibraheem Al-Furiah, Srinivas Aluru, Sanjay Goil, and Sanjay Ranka. Practical algorithms for selection on coarse-grained parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):813–824, 1997.

[3] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, 2015.

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

[5] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, Robert Endre Tarjan, et al. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[6] David R Cheng, Alan Edelman, John R Gilbert, and Viral Shah. A novel parallel sorting algorithm for contemporary architectures. *Submitted to ALENEX*, 2006.

[7] Richard John Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26:295–299, 01 1988.

[8] Frank Dehne and Hamidreza Zaboli. Deterministic sample sort for gpus. *Parallel Processing Letters*, 22(03):1250008, 2012.

[9] Paul F Dietz and Rajeev Raman. Small-rank selection in parallel, with applications to heap construction. *Journal of Algorithms*, 30:33–51, 01 1999.

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

[11] Vipul Harsh, Laxmikant Kale, and Edgar Solomonik. Histogram Sort with Sampling. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 201–212, Phoenix AZ USA, June 2019. ACM.

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

[13] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[14] Laxmikant V. Kale and Sanjeev Krishnan. A comparison based parallel sorting algorithm. *1993 International Conference on Parallel Processing - ICPP93 Vol3*, pages 196–200, 1993.

[15] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[16] Roger Kowalewski, Pascal Jungblut, and Karl Furlinger. Engineering a distributed histogram sort. *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

[17] Fabian Kuhn, Thomas Locher, and Rogert Wattenhofer. Tight bounds for distributed selection. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 145–153, 2007.

[18] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, 1993.

[19] Conrado Martínez and Salvador Roura. Optimal sampling strategies in quicksort and quickselect. *SIAM Journal on Computing*, 31(3):683–705, 2001.

[20] E. L. G. Saukas and S. W. Song. A note on parallel selection on coarse-grained multicomputers. *Algorithmica*, 24(3-4), 1999.

[21] ELG Saukas and SW Song. A note on parallel selection on coarse-grained multicomputers. *Algorithmica*, 24(3):371–380, 1999.

[22] Hari Sundar, Dhairya Malhotra, and George Biros. Hyksort. *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS 13*, 2013.

[23] B. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. *Hypercube Multiprocessors*, 1987:292–299, 1987.

[24] Andrew W Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th annual International symposium on Computer architecture*, pages 244–252, 1987.

[25] Lingxiao Zeng. Two parallel sorting algorithms for massive data. *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, 2021.