A Load-Balanced Parallel and Distributed Sorting Algorithm Implemented with PGX.D

*Zahra Khatami[1,2], Sungpack Hong[1], Jinsoo Lee[1], Siegfried Depner[1],
Hassan Chafi[1], J. Ramanujam[2], and Hartmut Kaiser[2]
[1]Oracle Labs, USA
[2]Center for Computation and Technology, Louisiana State University

***Abstract*—Sorting has been one of the most challenging studied problems in different scientific researches. Although many techniques and algorithms have been proposed on the theory of having efficient parallel sorting implementation, however achieving desired performance on different types of the architectures with large number of processors is still a challenging issue. Maximizing parallelism level in applications can be achieved by minimizing overheads due to load imbalance and waiting time due to memory latencies. In this paper, we present a distributed sorting algorithm implemented in PGX.D, a fast distributed graph processing system, which outperforms the Spark's distributed sorting implementation by around 2x-3x by hiding communication latencies and minimizing unnecessary overheads. Furthermore, it shows that the proposed PGX.D sorting method handles dataset containing many duplicated data entries efficiently and always results in keeping balanced workloads for different input data distribution types.**

***Index Terms*—Distributed sorting method, PGX.D distributed graph framework, Graph.**

I. Introduction

Having efficient and scalable distributed sorting method has been one of the most studied problems in computer science, since it is of importance for many algorithms and also part of widely adopted benchmarks for parallel supercomputers [1], [2]. There have been many different parallel sorting techniques proposed, however, improving the scalability of a distributed sorting algorithm with conventional techniques is still one of the major challenges. Overall, the scalability of the parallel distributed sorting technique mostly depends on how well overheads, synchronizations and latencies are scheduled in both algorithm and the framework that the algorithm is implemented in. It is usually hard to achieve the optimum performance with existing frameworks since they often impose large overheads for scheduling, introduce unnecessary synchronization steps and are not able to handle large amount of memory consumption [3]–[6]. In addition to these problems the system needs to partition the data entries between the machines equally, otherwise, it results in having workload imbalance, which in fact would prevent the system from scaling when dealing with large data. The communication overhead is another big challenge in these systems that needs to be taken in account when designing an efficient sorting technique.

In this research, we propose a new distributed sorting method, which overcomes these challenges by keeping balanced load and minimizes the overheads by fetching data efficiently in the partitioning and merging steps. The new handler is proposed that results in having a balanced merging while parallelizing merging steps, which improves the parallel performance. Moreover, the new investigator is proposed that results in keeping a balanced workloads among the distributed processors while dealing with dataset containing many duplicated data entries. This method is implemented in PGX.D, which is a scalable framework for various distributed implementations. PGX.D [7], [8] is a fast, parallel and distributed graph analytic framework that is able to process large graphs in distributed environments while keeping workloads well balanced among distributed machines. It improves the performance of the proposed sorting technique by exposing programming model that intrinsically reduces poor utilization of the resources by maintaining balanced workloads, minimizes latencies by managing parallel tasks efficiently and provides asynchronous task execution for sending/receiving data to/from the remote processors. The results presented in [7] show that PGX.D has low overhead and a bandwidth efficient communication framework, which easily supports remote data pulling patterns and is about 3x-90x faster than the other distributed graph systems such as GraphLab. Moreover, PGX.D decreases communication overheads by delaying unnecessary computations until the end of the current step, which allows the other processes to be continued without waiting for the completion of all the previous computations. Also it allows having asynchronous local and remote requests that avoids unnecessary synchronization barriers that helps in increasing scalability of the distributed sorting method [9].

In this paper, we show the performance improvements using distributed sorting method implemented in PGX.D to hide the communication latencies and to achieve better scalability. The performance of the proposed sorting framework shows around 2x-3x faster performance compared to the Spark's distributed sorting implementation. Spark is a fast engine for processing large-scale data and it is widely used for processing large amount of data in parallel and distributed. Moreover, the results of the proposed technique confirm maintaining load balance regardless of the data distribution type.

The remainder of this paper is structured as follows: in section II, the related researches and the methods for dis-

*This work was done during the author's internship at Oracle Labs.

tributed sorting are briefly explained; background information about PGX.D is provided in section III; the proposed PGX.D distributed sorting method and its implementation are explained in more details in section IV, and the experimental results and the comparison between the proposed distributed sorting method with Spark's distributed sorting technique implemented for different data distributions are presented in section V. Conclusions can be found in section VI.

II. Related Works

In this section, we briefly discuss about the implementation of some of the existing sorting techniques and their challenges. Different parallel algorithms have been studied for many years to achieve an efficient distributed sorting method implemented on distributed machines. Keeping balanced load is usually difficult while implementing most of these algorithms when facing dataset containing many duplicated data entries, which usually results in having some machines holding larger amount of data than the others. Most of these techniques consist of two main steps: *(a)* partitioning, and *(b)* merging. The big challenge in the partitioning step is to maintain the load balancing on different distributed machines in addition to distribute data among them in a sorting order [10], [11]. Also, the big challenge in the merging step is decreasing latencies and keeping the desired scalability during all merging steps till the end. The performance of the merging step is usually scalable when having a small number of machines. However, while utilizing larger number of machines, the synchronizations reduce its parallel scalability. Moreover, most of these existing techniques require a significant communication bandwidth, which increases overheads and degrades the parallel performance [12].

Batcher's bitonic sorting [13] is basically a parallel mergesort and was considered as one of the most practical sorting algorithms for many years. In this algorithm, data on each local is sorted, then for each processors pair, the data sequence is merged into ascending order while on the other pair, the data sequence is merged into descending order. This algorithm is popular because of its simple communication pattern [14], [15]. However, it usually suffers from high communication overhead as its merging step highly depends on the data characteristics and it often needs to exchange the entire data assigned to each processor [16].

Radix sort [17] is also used for implementing parallel and distributed sorting algorithms in many different applications due to the simplicity of its implementation [18]. This algorithm is not based on data comparison for sorting purposes. It is implemented by considering bit representation of the data. The algorithm starts by examining the least significant bit in each data entry. Then, in each time step, data entries are being sorted by considering their r-bit values. One of the big challenges in implementing this sorting technique is having unequal number of input keys [2], [19]. It usually suffers in irregularity in communication and computation. Since it also highly depends on the data characteristics, the computation on each processor and the communication between different processors are not known in the very first steps, which can negatively affect its parallel performance.

In order to increase the parallel performance of the sorting algorithm, its structure and design should be independent of the data characteristics to decrease communication overheads. Therefore, sample sort is often chosen for implementing distributed sorting. It is a fast sorting algorithm that keeps the load balancing better than the quick sort and doesn't have the mentioned communication problem faced in Batcher's bitonic sort and radix sort [20], [21]. The runtime in this algorithm is almost independent of the input data distribution types. It works by choosing some samples regularly from the locally sorted data of each processor and selecting final splitters from these samples to split data into different number of processors for the sorting purpose. However, in addition to having scalability challenge in its merging step, its performance highly depends on choosing sufficient amount of samples from each processor and picking the efficient final splitters from them. If this step is not efficiently designed, the algorithm may result in having load imbalance when facing dataset containing many duplicated data entries.

Nowadays, *Spark* is widely used for processing large data as it provides the scalable framework for analyzing large input data on distributed machines. Spark provides a resilient distributed dataset (RDD), which is a collection of tasks that are partitioned across distributed machines and they can be processed in parallel. The distributed sorting algorithm implemented in Spark uses three main stages: sample, map and reduce [22], [23]. In the sample stage, samples from the data is chosen for having balanced data partitioned among the distributed machines. In the map stage, data is partitioned on distributed machines and is locally sorted and being merged in the reduce stage. TimSort [24] is chosen as a sorting technique in Spark and the experimental results show that it performs better when the data is partially sorted. This algorithm starts by finding subsequences of the elements in descending or ascending order and performs balanced merges on them in each merging step. For this purpose, it proceeds on the chosen minimum run sizes that are bulked up by using insertion sort and partially merge them in place, which helps its implementation to fetch data efficiently.

In this paper, the sample sorting technique is chosen as the base method for implementing a distributed sorting technique in PGX.D, which solves the mentioned challenges by implementing improvement in choosing splitters and managing communications between different processors efficiently. In addition, the performance optimizations considered in TimSort are also applied in the proposed sorting technique. A sorting library is generated in PGX.D that performs efficiently by:

✓ maintaining load balance on different number of processors, even when having a dataset containing many duplicated data entries,

✓ reducing overheads in the merging step by keeping balanced merging and fetching data efficiently,

✓ reading/writing data from/to the remote processors asynchronously, which avoids the unnecessary synchro-

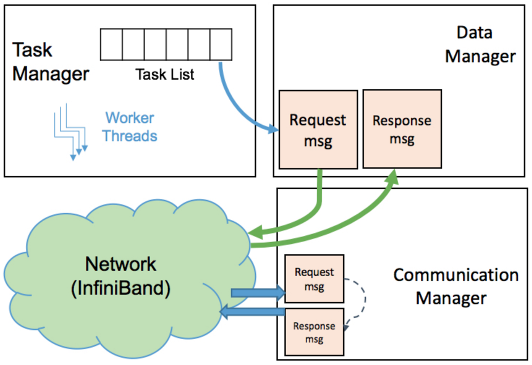


Figure 1: Three important mangers in PGX.D

nizations in each step, and

✓ keeping bandwidth-efficient communication implemented in PGX.D.

Our sorting technique is modeled by considering problem size, number of processors and type of the data distribution for improving load balancing. Section III discusses how PGX.D framework helps distributed sorting to reduce the synchronizations and the latencies during the sorting implementation. The programming model of the distributed sorting method is explained in more details in section IV.

III. Background Information

In this section the overview of PGX.D is presented to show its capability in providing scalable framework for implementing efficient distributed sorting. PGX.D is a fast, parallel and distributed graph analytic framework and it is able to process large graphs in distributed environments while keeping workloads well balanced among multiple distributed machines [7], [8]. It uses a fast context switching mechanism that results in having low overhead in task scheduling as well as a bandwidth efficient communication process. It guarantees low communication overhead by applying ghost nodes selection that results in decreasing number of the crossing edges as well as decreasing communication between different processors. Also, a new edge chunking strategy is implemented that improves task scheduling and results in having balanced workload between the processors in each machine. Moreover, it relaxes the unnecessary synchronizations while applying local/remote write requests. Figure 1 shows the architecture of PGX.D. The layers of the system are explained as follows:

- Task Manager: A list of tasks is created within a task manager at the beginning of each parallel step. The task manager initializes a set of worker threads and each of these threads grab a task from the list and executes it. If the task is requested from the remote machine, the execution of the process is continued until the data becomes available, which relaxes the synchronization barriers. Also, it determines the schedules for sending messages to the remote machines by considering size of the messages stored in the request buffer and sending them whenever that buffer reaches to its maximum size or the worker thread has completed all its scheduled tasks.
- Data Manger: Graph data across different machines is maintained within the data manager and they are stored in the Compressed Sparse Row (CSR) data structure on each machine. Whenever a message has to be sent to the remote machine, the data manager buffers it up in the request buffers for the task manager while keeping corresponding data structure. Also, the partitioning step, choosing ghost nodes and edge chunking on the graph data is applied within this manager during the graph loading. These implementations reduce communication overheads significantly and result in improving load balancing [7]. The location of each node is identified with this manager as well, which allows to handle a data transformation and to determine the destined machine for the messages within the request buffers.
- Communication Manager: The response messages to the remote machines are managed with the communication manager. The main goal of the communication manager is to provide communication primitives which allow for low overhead communication. It provides a fast and a low overhead mechanism while processing incoming request messages and applies the atomic instructions while processing writing requests. Also, it is able to handle large amount of small sized communication between different machines efficiently.

PGX.D takes care of the synchronizations between each step and provides both write and read requests within the same parallel region. As a result, PGX.D framework is more relaxed compared to the bulk-synchronization model used in the MapReduce models. The previous analysis and experiments show a good performance on the certain types of the graph algorithms. For more details about PGX.D, we recommend the reader to refer to [7]. In this research, we choose PGX.D to implement the distributed sorting techniques, which helps the algorithm to perform efficiently and scales desirably. Moreover, by adding this distributed sorting method in PGX.D, user can also easily sort data of their multiple graphs with different types and implement more analysis on them, such as retrieving top values from their graph data or implementing binary search on the sorted data. The programming model of the PGX.D distributed sorting technique is explained in more details in the next section.

IV. Programming Model

The PGX.D distributed sorting technique is designed based on sample sort algorithm, which is one of the most popular sorting algorithms and is usually faster than than other techniques by better maintaining load balance and minimizing communication overheads [20], [21]. As discussed in section II, this algorithm works by choosing samples from the locally sorted data of each processor and selecting final splitters from these samples to split data into different number of processors for the sorting purpose. However, one of the big challenges that can easily impact the performance of this sorting algorithm is having non-efficient splitters. In other words, if the splitters do not spilt data into the equal pieces, applying this algorithm may result in having larger amount of data on some processors compared to the others, which can degrade the performance significantly. Therefore this sampling step can be considered as the critical step for this algorithm that has received a wide attention for the development to obtain the expected workload on distributed processors. The performance of this step depends on the number of samples received on Master from the other processors, which arises two different scenarios that should be considered:

A) If a large amount of samples is sent to Master from each processor, Master will get a good idea about a data distribution on each processor, which helps it to choose the efficient splitters from the received samples that results in having better load balance. However, sending large amounts of data from each processor to Master, increases communication overheads as well as increasing the time needed for merging received data on Master.

B) If a small amount of samples is chosen from each processor, the communication overheads will be decreased. However, Master may be unable to choose splitters which cause an equal distribution which results in having load imbalance issues.

PGX.D distributed sorting method manages this issue by designing scalable framework that efficiently uses the bandwidth efficient as well as providing large enough number of samples for keeping balanced workloads. In addition, handling balanced merging and providing asynchronous task executions in sending/receiving data to/from the other processors are another features that are provided with the proposed sorting technique in PGX.D to achieve a desired parallelism level. These features are considered while implementing proposed sorting technique that mainly consists of six steps, which will be briefly outlined here and explained in more detail later.

1) Each processor sorts its data locally by using parallel quick sort.
2) The regular samples are chosen from the sorted data and they are sent to Master.
3) Master selects the final splitters from the received samples and it broadcasts them to all processors.
4) Each processor determines the range of the data to be sent to the remote processor by implementing binary search of these splitters on their local sorted data.
5) After setting up the senders and the receivers, each processor starts sending and receiving data simultaneously.
6) After receiving data from different remote processors, all data is merged together while keeping information regards to their previous processors and locations.

A. Handling Balanced Merging

In step **(1)**, data is sorted locally on each processor by implementing parallel quick sort in such a way that data is divided equally among a number of the worker threads on each processor. Then, each worker thread sorts its data locally. Sorted data from each thread is merged together by keeping balanced merging. For this purpose, the new handler is proposed in this section that manages the merging implementation to keep the balanced merging in each step, which avoids the cache misses. For example in figure 2 with 8 threads, in the first step, the data in thread 1 is merged with the data in thread 0, the data in thread 3 is merged with the data in thread 2, the data in thread 5 is merged with the data in thread 4, and the data in thread 7 is merged with the data in thread 6. So, at the end of the first step, data on the threads with odd ID are merged with data on the threads with even ID and they have almost the equal sizes. In the next step, the data in thread 2 is merged with the data in thread 0 and the data in thread 6 is merged with the data in thread 4. In the last step, the data in thread 4 is merged with the data in thread 0. Also, this handler implements each of these merging steps in parallel which improves the parallel performance of this sorting technique.

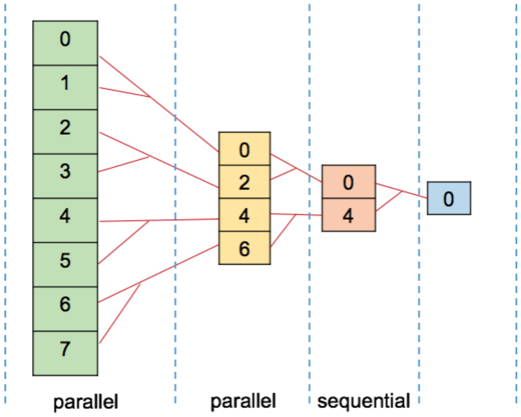
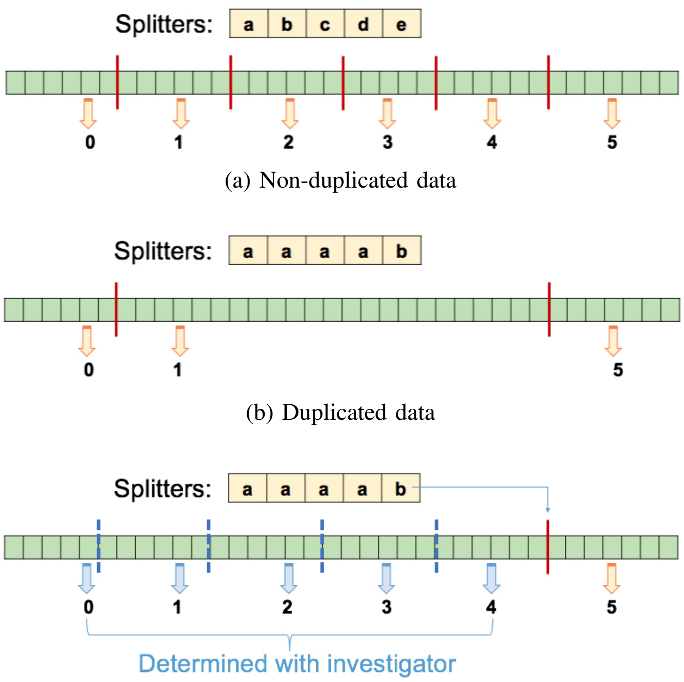


Figure 2: Keeping balanced merging in PGX.D distributed sorting method.

B. Providing Large Enough Number of samples

PGX.D distributed sorting method is designed in such a away that it keeps the communication overheads as less as possible while letting the largest possible amount of samples to be sent from each processor destined to the Master in step **(2)** to achieve a better load balance. The amount of data is chosen by considering number of processors and the size of a read buffer defined in the PGX.D data manager. The size of this buffer is assigned 256 Kbyte in PGX.D based on measuring different performances and choosing the best one [7]. So if there are p number of processors in the system, each processor has to send only $256/p$ Kbyte data to Master, which results in sending only one buffer data entries from each processor to Master and gathering all received data entries in one receiving buffer on Master. The experimental results of this algorithm on different types of data confirm that the overheads are significantly reduced and the amount of data provided to the Master is large enough to choose the efficient splitters.

After receiving $p - 1$ splitters from step **(3)**, in step **(4)**, the ranges of data for sending to the other processors are determined by implementing binary search on data for each splitter. Then data from the splitter$[j - 1]$ to the splitter$[j]$ is sent to the processor $j - 1$, as it is illustrated in figure 3a.



(a) Non-duplicated data

(b) Duplicated data

(c) Dividing range between duplicated splitters

Figure 3: Determining range of data to be sent to the remote processor by implementing binary search for the received splitters on locally sorted data.

However, if there exists dataset containing many duplicated data entries on the processors, this method results in having load imbalance by sending large data range to one processor and not sending any data to the others as shown in figure 3b. To solve this problem, we propose the investigator to be implemented during searching for each splitter on local data entries, which makes a binary search to be executed for only non-duplicated splitters, then it divides the determined range between the duplicated splitters equally. For example in figure 3c, the range is divided equally between 4 duplicated splitters with the same *a* value.

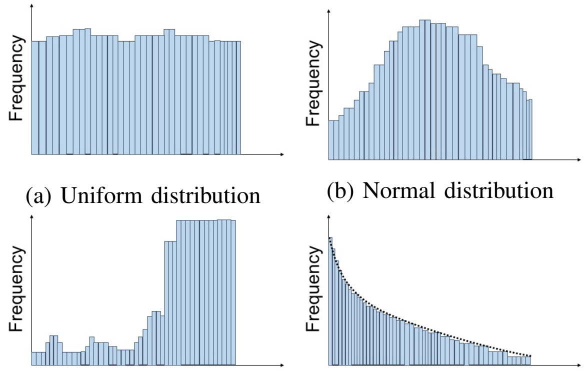
C. Asynchronous Task Execution

After determining ranges for each destination in step **(4)**, these information are broadcasted to all processors. So each processor knows how much data it will receive from the other processors. This strategy enables the sorting technique to receive data from multiple processors at the same time by applying offsets for each received data entry from the neighbor lists, which allows writing in the local data list simultaneously. Also each processor is able to send data while receiving data, which avoids the unnecessary synchronizations between these steps as well. In the last step, the new arriving data is merged together in parallel by applying same method described in figure 2.

At the end, data is sorted in such a way that smaller data entries is stored and gathered in the processor with the smaller ID and the bigger data is stored and gathered in the processor with the bigger ID. This sorting library also provides an API for the users to implement a binary search on data as well as finding information regards to the previous processors and the previous indexes of the new received data entry of each processor. Also it is able to sort multiple different data simultaneously and keeping the balanced load while having a dataset containing many duplicated data entries. The experimental results of PGX.D's distributed sorting technique is studied in more details in section V.

Category	Item	Detail
SW	OS	Linux 2.6.32
	compiler	gcc 4.8.2
CPU	Type	Intel Xeon E5-2660
	Frequency	2.40 GHz
	Parallelism	2 socket with 8 cores
DRAM	Type	DDR3-1600
	Size	256 GB
Size	# Machines	32
Network	Card	Mellanox Connect-IB
	Switch	Mellanox SX6512
	Raw BW	56Gb/s (per port)

Table I: Experimental Environment



(a) Uniform distribution

(b) Normal distribution

(c) Right-skewed distribution

(d) Exponential distribution

Figure 4: Different types of data distribution

V. Experimental Results

The proposed PGX.D distributed sorting mechanism has been implemented for different types of data on our cluster, which is summarized in Table I. Its performance is compared with the Spark's distributed sorting implementation with version 1.6.1 that is publicly available. For our evaluation, first we study the proposed sorting technique performance on four different input datasets shown in figure 4 with the uniform, normal, right-skewed, and exponential distribution type. The implementation of our proposed sorting technique on data with the right-skewed and the exponential distribution type are specially intended to confirm its ability to maintain load balancing in a case of having a dataset containing many duplicated data entries. All of these input datasets have one billion entries and the sorting is performed on 8 up to 52 processors with using 32 threads per each for in-node parallelization purposes. Distributed sorting method in Spark

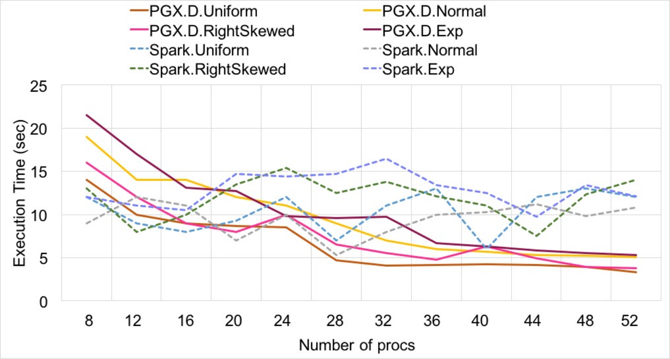


Figure 5: PGX.D distributed sorting method total execution times for data from figure 4.

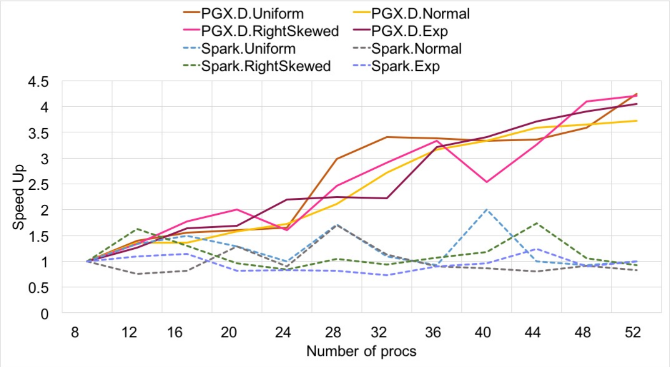


Figure 6: Comparison results of strong scaling of the distributed sorting method implemented in PGX.D and Spark for data from figure 4.

is executed by implementing *sortByKey()* in the MapReduce operation.

Figure 5 shows the execution time of the distributed sorting methods on data from figure 4. It illustrates that PGX.D sorts data efficiently regardless of the input data distribution type. Since the communication latencies between different processors increase with an increasing number of processors, it is usually hard to achieve a good scaling behavior. However, figure 6 shows a better speedup of PGX.D distributed sorting technique compared to the sorting technique in Spark.

Figure 7 shows the execution time of each steps for the experiments on the normal and right skewed distribution types studied in figure 5. It can be seen that sending/receiving data costs less time than the other steps, which validates the efficient-bandwidth communication and the asynchronous execution provided in PGX.D that remove the unnecessary barriers and allow to have a low communication overhead to achieve a better parallelization performance.

Table II shows the size of data on each processor after PGX.D distributed sorting implementation having 10 processors. It illustrates data is distributed equally on the processors, in the case of having a dataset containing many duplicated data entries in both right-skewed and exponential distribution types.

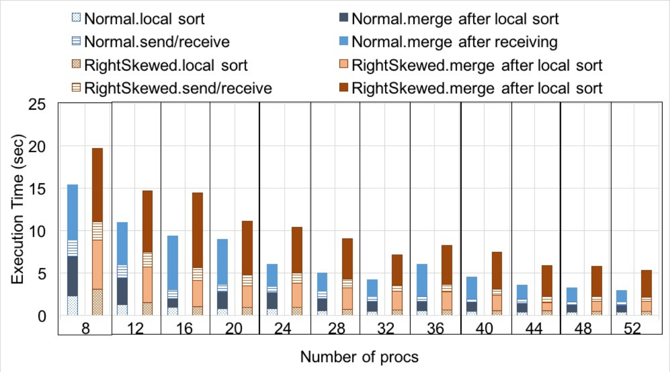


Figure 7: PGX.D distributed sorting method execution times of each different steps for the normal and right skewed distribution types from figure 4.

Also, these information generally shows how the proposed investigator works during a binary search implementation on the duplicated splitters. For example, the results according to the sizes of data in the right-skewed distribution show having the exact equal sized 9.998% for each data on the processors $2 - 9$.

We also evaluate the performance of our sorting technique on the Twitter graph dataset, which has 41.6 million vertices and 1.5 million edges in a 25GB dataset. Figure 8 shows the execution time compared to Spark's distributed sorting implementation, which illustrates that it is faster than Spark by around 2.6x on 52 processors. The ranges of data on each processor after sorting with 8, 12 and 16 processors are included in Table III, which confirms the accuracy of the proposed technique that data with the smaller value are located on the processor with the smaller ID and data with the bigger value are located on the processor with the bigger ID.

In figure 9, we study about the impact of sample sizes on the communication overheads and the total execution time. Seven different sample sizes are used: $0.004X$, $0.04X$, $0.4X$, X, $1.004X$, $1.04X$, and $1.4X$, where X = *256KB/number of processors* (*256KB* is equal to the size of the read buffer in PGX.D). It shows that the small number of samples not only results in having load imbalance, but it also increases communication overheads, which is due to having non-efficient splitters that result in sending more data to some number of processors. Also, the total execution time for the cases of having very small amount of samples and large amount of samples are both greater than the execution time of having X samples.

In figure 10, we study about the impact of sample sizes on the load balance from figure 9 for three different samples sizes as $0.004X$, X, and $1.4X$. The minimum and the maximum size of data is illustrated in using different number of processors. It shows that $0.004X$ number of samples is not large enough to keep balanced workloads between the processors, as it results having the load difference by an average 133192278 while using 52 processors. However, both

Distribution type	proc0	proc1	proc2	proc3	proc4	proc5	proc6	proc7	proc8	proc9
Uniform	9.984%	9.989%	9.998%	10.003%	10.022%	10.000%	9.993%	9.994%	10.018%	9.993%
Normal	10.011%	10.003%	10.000%	10.000%	9.981%	9.999%	9.992%	10.008%	9.999%	10.002%
Right-skewed	10.020%	9.990%	9.998%	9.998%	9.998%	9.998%	9.998%	9.998%	9.998%	9.998%
Exponential	9.998%	9.998%	9.998%	9.998%	9.997%	9.997%	9.997%	9.996%	9.996%	10.020%

Table II: The ratio of data sizes on each processor after implementing balanced PGX.D distributed sorting method using 10 processors on 1 billion data from figure 4.

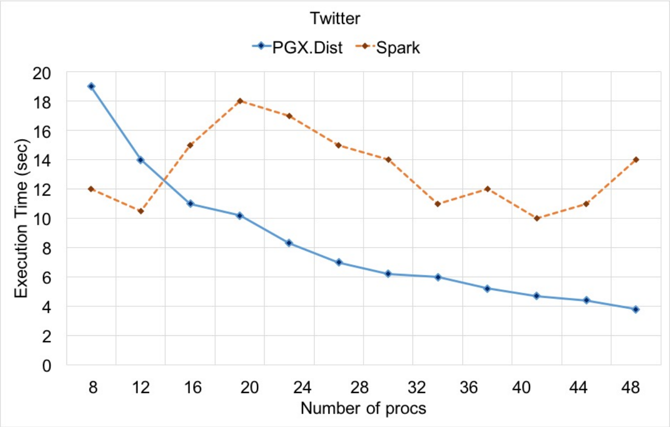


Figure 8: Execution time of distributed sorting method implemented in PGX.D compared to Spark's distributed sorting implementation on Twitter graph dataset.

X and $1.4X$ result in having balanced loads in all experiments. By considering results from the figures 9 and 10, X is chosen as the sample size in the distributed sorting method implemented in PGX.D, which results in having load balance while keeping low communication overhead.

Since the main purpose of this sorting technique is to distribute and to sort large amount of data efficiently, it is very important to avoid using extra memory during the implementation. The average memory consumptions using different number of processors for Twitter graph dataset are shown in figure 11. Resident Set Size (RSS) is the RAM memory that is allocated for the process and it is shown in dark blue. Light blue illustrates the total temporary memory usage during the process except RSS usage, which is allocated during the process and becomes free at the end. It is shown that the maximum memory needed for sorting on 20 processors is less than 300MB, which is used for keeping previous information of each data's previous processor and location.

# procs	8	12	16
proc0	0 - 12.58	0 - 8.39	0 - 6.29
proc1	12.58 - 25.07	8.39 - 16.68	6.29 - 12.58
proc2	25.07 - 37.55	16.68 - 25.07	12.58 - 18.78
proc3	37.55 - 50.04	25.07 - 33.36	18.78 - 25.06
proc4	50.04 - 62.53	33.36 - 41.66	25.06 - 31.26
proc5	62.53 - 75.02	41.66 - 50.04	31.26 - 37.55
proc6	75.02 - 87.50	50.04 - 58.33	37.55 - 43.75
proc7	87.50 - 95	58.33 - 66.63	43.75 - 50.04
proc8		66.63 - 75.02	50.04 - 56.24
proc9		75.02 - 83.31	56.24 - 62.52
proc10		83.31 - 91.61	62.52 - 68.73
proc11		91.61 - 95	68.73 - 75.01
proc12			75.01 - 81.21
proc13			81.21 - 87.50
proc14			87.50 - 93.70
proc15			93.70 - 95

Table III: Range of data on each processors after implementing balanced PGX.D distributed sorting method on Twitter graph dataset.

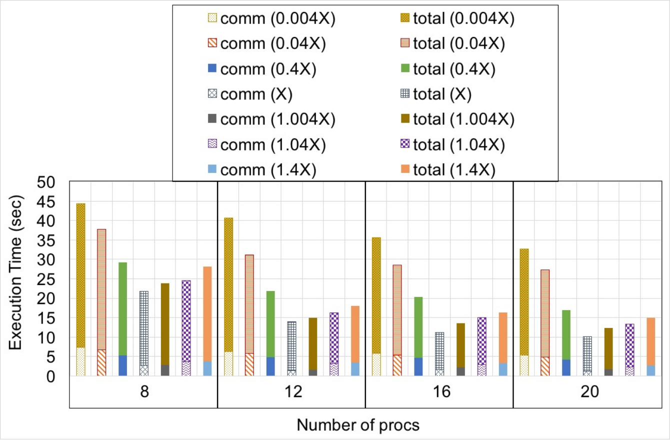


Figure 9: Communication overhead and total execution time comparison of distributed sorting method implemented for Twitter graph dataset using different sample sizes, where X = *256KB/number of processors*.

VI. Conclusions

Implementing sample sort is straightforward and simple and can be faster than the other techniques if it is designed efficiently. In another words, despite its simplicity, it causes a load imbalance and increases communication overheads if it is not able to handle having a dataset containing many duplicated data entries and if it is performed within a framework without the capability of exploiting bandwidth efficiently. In this paper, we have developed a distributed sorting technique with PGX.D that eliminates the load balancing problem when sorting dataset containing many duplicated data entries on distributed machines. This technique is implemented by considering number of processors and the maximum buffer size provided with PGX.D, which helps gathering enough information from different machines without increasing communication overheads. Also, the new handler is proposed that results in having a balanced merging while parallelizing merging steps, which improves the parallel performance. Moreover, this technique is able to keep the workload balanced among the distributed processors by applying proposed investigator during a binary search. One of the other advantages of this sorting technique

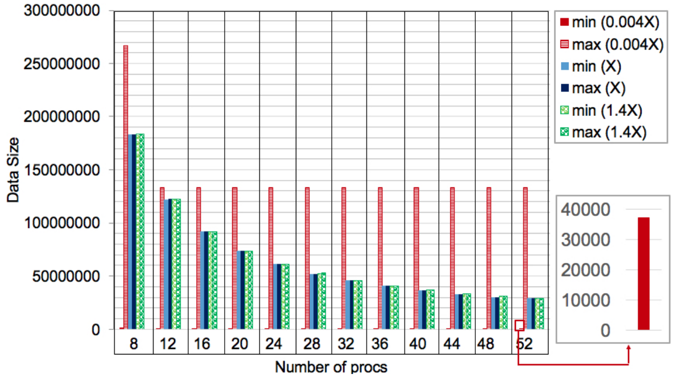


Figure 10: Work load comparison on each processor using different sample sizes on Twitter graph dataset, where X = *256KB/number of processors* .

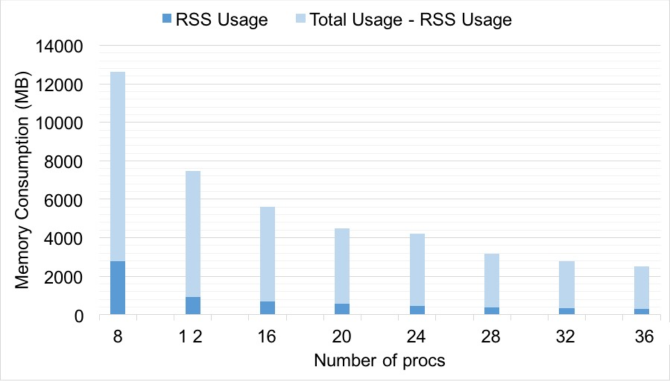


Figure 11: Average memory consumptions of PGX.D distributed sorting method on Twitter graph dataset using different number of processors.

is the high-level API exposed to the user, which is a generic and works with any data type and is able to sort different data simultaneously.

To evaluate the performance of the proposed distributed sorting technique, we have measured it with different input datasets having various distribution types: uniform, normal, right-skewed and exponential distributions. We were able to sort one billion data on up to 52 machines by maintaining load balance on all processors. In addition, a balanced PGX.D distributed sorting method has been compared with Spark's distributed sorting implementation on the same input data and the experimental results illustrated that our technique is by around $2x - 3x$ faster than Spark's distributed sorting method. Also, it was shown that it uses as less memory as possible to avoid increasing memory usage while facing big data.

References

[1] Michael C Loui. The complexity of sorting on distributed systems. *Information and Control*, 60(1):70–85, 1984.

[2] Zhang Qian, Ge Yufei, Liang Hong, and Shi Jin. A load balancing task scheduling algorithm based on feedback mechanism for cloud computing. *International Journal of Grid and Distributed Computing*, 9(4):41–52, 2016.

[3] Rolf Rabenseifner, Georg Hager, Gabriele Jost, and Rainer Keller. Hybrid MPI and OpenMP parallel programming. In *PVM/MPI*, page 11, 2006.

[4] Ashay Rane and Dan Stanzione. Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In *Proc. of 10th LCI Int'l Conference on High-Performance Clustered Computing*, 2009.

[5] Zahra Khatami, Hartmut Kaiser, and J Ramanujam. Using hpx and op2 for improving parallel scaling performance of unstructured grid applications. In *Parallel Processing Workshops (ICPPW), 2016 45th International Conference on*, pages 190–199. IEEE, 2016.

[6] Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. A massively parallel distributed n-body application implemented with hpx. In *Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 57–64. IEEE Press, 2016.

[7] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. Pgx. d: a fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. ACM, 2015.

[8] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[9] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2016.

[10] Minsoo Jeon and Dongseung Kim. Parallelizing merge sort onto distributed memory parallel computers. In *International Symposium on High Performance Computing*, pages 25–34. Springer, 2002.

[11] Shmuel Zaks. Optimal distributed algorithms for sorting and ranking. *IEEE Transactions on computers*, 100(4):376–379, 1985.

[12] KVS Ramarao. Distributed sorting on local area networks. *IEEE Transactions on Computers*, 37(2):239–243, 1988.

[13] Yong Cheol Kim, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Communication-efficient bitonic sort on a distributed memory parallel computer. In *Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on*, pages 165–170. IEEE, 2001.

[14] Mihai F Ionescu and Klaus E Schauser. Optimizing parallel bitonic sort. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 303–309. IEEE, 1997.

[15] David Nassimi and Sartaj Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, 100(1):2–7, 1979.

[16] Michael Hofmann and Gudula Runger. A partitioning algorithm for parallel sorting on distributed memory systems. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 402–411. IEEE, 2011.

[17] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Partitioned parallel radix sort. *Journal of Parallel and Distributed Computing*, 62(4):656–668, 2002.

[18] Andrew Sohn and Yuetsu Kodama. Load balanced parallel radix sort. In *Proceedings of the 12th international conference on Supercomputing*, pages 305–312. ACM, 1998.

[19] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

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

[21] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *European Symposium on Algorithms*, pages 784–796. Springer, 2004.

[22] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. Graysort on apache spark by databricks. *GraySort Competition*, 2014.

[23] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[24] Tim Peters. Timsort, 2002.