SDS-Sort: Scalable Dynamic Skew-aware Parallel Sorting

Bin Dong   Surendra Byna   Kesheng Wu

Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720

{DBin, SByna, KWu}@lbl.gov

ABSTRACT

Parallel sorting is an essential algorithm in large-scale data analytics using distributed memory systems. As the number of processes increases, existing parallel sorting algorithms could become inefficient because of the unbalanced workload. A common cause of load imbalance is the skewness of data, which is common in application data sets from physics, biology, earth and planetary sciences. In this work, we introduce a new scalable dynamic skew-aware parallel sorting algorithm, named *SDS-Sort*. It uses a skew-aware partition method to guarantee a tighter upper bound on the workload of each process. To improve load balance among parallel processes, existing algorithms usually add extra variables to the sorting key, which increase the time needed to complete the sorting operation. SDS-Sort allows a user to select any sorting key without sacrificing performance. SDS-Sort also provides optimizations, including adaptive local merging, overlapping of data exchange and data processing, and dynamic selection of data processing algorithms for different hardware configurations and for partially ordered data. SDS-Sort uses local-sampling based partitioning to further reduce its overhead. We tested SDS-Sort extensively on Edison, a Cray XC30 supercomputer. Timing measurements show that SDS-Sort can scale to 130K CPU cores and deliver a sorting throughput of 117TB/min. In tests with real application data from large science projects, SDS-Sort outperforms HykSort, a state-of-art parallel sorting algorithm, by 3.4X.

1. INTRODUCTION

Parallel sorting is a commonly used function by scalable data management systems and scientific applications. For example, data management systems, such as SciDB [7] and Scientific Data Services (SDS) framework [12], sort large-scale data records in parallel to improve the locality of data accesses. Data clustering applications, such as BD-CATS[21], use parallel sorting to order cosmological particles. Well-known parallel sorting algorithms include bitonic sort [4], radix sort [30], and parallel sort by sampling (PSS) [24]. Among them, PSS minimizes the interprocess data movement and communication, both of which are expensive operations in distributed memory systems [2]. Because of the reduced communication, PSS algorithm has been used by multiple data analytics methods [12, 21] and has been extensively optimized [24, 28, 19]. Implementations of PSS, however, generally target at a certain dataset or a specific hardware. Hence, existing PSS algorithms could be ineffective on some computing hardware and exhibit significant load-imbalance on real-world datasets that are highly skewed. In this research, we aim to explore a new PSS algorithm, which can efficiently sort various datasets, including skewed data and partially ordered data, and also can take into account the features of the hardware of current supercomputers, i.e., high throughput network interconnect and multicore CPUs.

On a distributed memory computing system with p parallel processes and enough memory to hold data in core, classical PSS algorithm [19] has three steps. 1) *Pivot selection*: using a random sample of the data to choose $p-1$ ordered global pivots. These $p-1$ global pivots partition the value space of the data into p ranges and each range is assigned to a single process. 2) *Exchange*: perform all-to-all data exchange to gather the data records belonging to a value range onto a single process. 3) *Local ordering*: order the data records within each process. This is the final ordering step that produces a globally ordered output.[1]

It is widely accepted that partitioning the data (i.e., load or workload) evenly among processes will significantly benefit the overall sorting performance by reducing the time of exchange and local ordering step [28]. Applying existing PSS algorithms in sorting uniformly distributed data with N records, the upper bound on the workload for a process is $O(2N/p)$ [24, 28, 19]. But, when applying these methods to sort the skewed data, which widely appears in physics, biology, earth, and planetary sciences [20, 21, 12], this upper bound increases linearly as the skewness of data increases [19]. A dataset is said to be skewed when its value distribution is not uniform, for example, having a cluster of very popular values toward one end of its distribution function [14]. Often, a skewed dataset contains many duplicated values and the amount of duplicates represents the skewness of data. These duplicated values might produce duplicated global pivots, which eventually cause existing PSS al-

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

© 2016 ACM. ISBN 978-1-4503-4314-5/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907300

[1]The pivot selection step might also employ the local ordering operation. When it is necessary distinguish these two ordering steps, we will call the one in pivot selection the initial ordering step and step 3 the final ordering step.

gorithms to assign more data to a single process than others leading to significant load imbalance. In some applications, users might also require stable sorting algorithm to maintain the relative order of duplicated values[19]. Using secondary sorting keys, such as payload [18] or the rank of data record [28], could alleviate these issues. However, the presence of these secondary sorting keys increases the amount of work in all three steps of PSS, increasing both CPU time and communication time. Moreover, requiring secondary sorting might also let users have limited sorting keys to choose or spend extra efforts to pick up secondary sorting keys. Thus, it is highly desirable to develop parallel stable sorting algorithms that do not require such secondary sorting keys.

In order for a PSS to execute efficiently, it must be able to take advantage of the new hardware. For example, current supercomputers typically have multicore CPUs and high-throughput interconnect network. However, existing PSS algorithms [28] are generally optimized for slow network through merging the data from multiple CPU cores on the same node to avoid slow data exchange step. New research indicates that using multiple CPU cores to saturate high-throughput network is equally important in reducing the time of the data exchange [8, 17]. At the same time, because the high-throughput interconnects could reduce the data transfer time so much that the benefit of overlapping data exchange and local ordering, widely used by existing PSS algorithms [24, 28], might also be significantly reduced.

In the local ordering step, if we use a straightforward sorting algorithm, its time complexity would be $O(Nlog(N))$. If we view the data as p ordered chunks received from p process, we might achieve a lower time complexity of $O(Nlog(p))$ using merging [28]. However, if the data records are partially ordered, the complexity of sorting might reduce from $O(Nlog(N))$ to $O(N)$ [9]. In some cases the input data might be partially ordered [9], or the data produced by the data exchange step might be nearly sorted [28]. Therefore, it would be useful to recognize the partially ordered data and use efficient approaches for the local ordering step.

To achieve the design goals above, we propose a novel scalable dynamic skew-aware parallel sorting algorithm, called SDS-Sort. SDS-Sort only needs to exchange data once among processes in the whole sorting procedure. More importantly, based on the characteristics of the data and the computing hardware, SDS-Sort employs adaptive idea to dynamically select the most appropriate subroutine for each step. Specifically, SDS-Sort has following unique features:

- SDS-Sort achieves tight upper bound of $O(4\frac{N}{p})$ on the workload of each process and achieves balanced load among processes in sorting skewed datasets without rely on secondary sorting keys.
- SDS-Sort performs various optimizations dynamically to make use of multicore processing and high-speed interconnects. SDS-Sort has dynamic optimizations in local ordering and in overlapping data exchange and local ordering.
- SDS-Sort is capable of performing parallel stable sorting algorithm to maintain the relative order of same values. To the best of our knowledge, this is the first sampling-based stable parallel sorting algorithm for distributed memory systems.
- SDS-Sort uses the local pivots-based parallel partition to further reduce the overhead of data partition.

We provided theoretical analysis for the workload on each process. On Edison, a Cray XC30 supercomputer, we evaluated SDS-Sort with both synthetic and real datasets. We show that SDS-Sort outperforms HykSort, a state-of-art parallel sorting algorithm, by 3.4× in sorting highly skewed Palomar Transient Factory data, which contains around 28% duplicated values. We have scaled SDS-Sort up to 128K CPU cores and where it delivers $\sim$ 117TB/min throughput in sorting highly skewed data.

The rest of the paper is organized as follows. In Section 2, we present the details of SDS-Sort and its design considerations. Section 3 describes our experimental setups. In Section 4, we evaluate the performance of SDS-Sort algorithm. In Section 5, we provide related research efforts of parallel sorting algorithm. We conclude the paper with a discussion of future work in Section 6.

2. SDS-SORT ALGORITHM DESIGN

2.1 SDS-Sort algorithm overview

With the goal of developing a parallel sorting algorithm that works on skewed and partially ordered datasets, uses multicore processors efficiently, and preserves the order of duplicate values, we propose **SDS-Sort**, a scalable dynamic skew-aware parallel sorting algorithm. We introduce a new skew-aware partition strategy to ensure the balanced load of all parallel processes in sorting highly skewed data. Through the same skew-aware partition method, SDS-Sort can also preserve the order of duplicated values scattered across different compute nodes. By supporting dynamic node level merging and overlapping of data exchange and local ordering phases, we devise SDS-Sort to take full advantage of hardware features on HPC systems such as high-throughput network interconnect and multicore CPUs. Our algorithm also incorporates strategies to work with partially ordered data, which can be ordered more quickly than random data. We allow using different local ordering strategies based on the characteristics of the data.

The overall SDS-Sort algorithm is shown in Fig. 1. We follow the single instruction and multiple data (SIMD) pattern used by MPI [16] to describe the algorithm. All the functions shown in Fig. 1 run in parallel on multiple cores or on multiple computing nodes or both. The standard functions of MPI (prefixed with *"MPI"*) are used to clearly express the operations. The functions prefixed with *"Sdss"* are defined in this paper and will be discussed in the following sections.

Let A be a vector of data records to be sorted and each record has a key for sorting and an arbitrary number of non-key values (also called payload). B is a vector to hold the sorted output data. Assume that there are N records in the input array to be sorted using p MPI processes. Each MPI process works on n data records, where $n = N/p$. The size of B on each MPI process, referred as m, is dynamically determined based on the data partition of global pivots. The MPI communicator *comm* and the number of cores per node c are passed into SDS-Sort for communication and adaptive merging, respectively. Theoretically, c might be any number from 1 to p. In our implementation, we set c to be the number of CPU cores per node, which can avoid merging data cross nodes. We provide a flag s_f to specify whether SDS-Sort needs to perform stable sorting. If s_f = **TRUE**, SDS-Sort preserves the order of duplicate keys, and if s_f = **FALSE**, SDS-Sort ignores the stability in the output. Ad-

```
function SDS-Sort($A[1,\ldots,n]$, $B[1,\ldots,m]$, $com$, $c$, $s_f$, $\tau_m$, $\tau_o$, $\tau_s$)
        $A$ : data array with size $n$ to sort                  $s_f$ : flag of stable sorting (TRUE or FALSE)
        $B$ : sorted data array with size $m$                   $\tau_m$ : merging parameter
        $com$ : MPI communicator for all process                $\tau_o$ : overlapping parameter
        $c$ : the number of CPU cores per node                  $\tau_s$ : local ordering parameter
 0.  $p = MPI\_Comm\_size$(com)      ▷ get the number of processes used to sort data ◁
 1.  $r = MPI\_Comm\_rank$(com)      ▷ get the rank of current process, ( 0, $p-1$) ◁
 2.  $A = SdssLocalSort(A,\ 1,\ s_f)$      ▷ SdssLocalSort is described in § 2.2 ◁
 3.  if ($\frac{n}{p} \le \tau_m$) then      ▷ merge data when the average size of all-to-all exchange is small ◁
 4.      ($c_g$, $c_l$ )= $SdssRefineComm(com)$      ▷ SdssRefineComm is described in § 2.3 ◁
 5.      $A = SdssNodeMerge$(A, $c_l$)      ▷ SdssNodeMerge is described in § 2.3 ◁
 6.      $n = n \times c$, $p = \frac{p}{c}$, $com = c_g$, $c = 1$      ▷ update parameters, n, p, com, and c ◁
 7.  end if
 8.  $P_l[0, \ldots, p-1] = A[1+\lfloor \frac{n}{p} \rfloor, \ldots, 1+(p-1)\lfloor \frac{n}{p} \rfloor]$      ▷ local sampling ◁
 9.  $P_g[0, \ldots, p-1] = SdssSelectPivots(p_l)$      ▷ SdssSelectPivots is described in § 2.4 ◁
10.  $(sdisp[0,\ldots,p], scount[0,\ldots,p]) = SdssPartition(A,\ P_l,\ P_g,\ p,\ r,\ s_f,\ com)$      ▷ SdssPartition is described in § 2.5 ◁
11.  $MPI\_Alltoall(scont,\ p,\ rcont,\ p,\ com)$      ▷ exchange the size of data to receive ◁
12.  $rdisp[0,\ldots,p] = Accumulate(rcont)$      ▷ compute the displacement of received data ◁
13.  $m = rdisp[p-1] + rcount[p-1]$
14.  $B = Memory\_Alloc(m)$
15.  if( $s_f$ == TRUE or $p > \tau_o$) then      ▷ no overlapping for stable sorting and larger number of processes ◁
16.      MPI_Alltoallv($A$, B, sdisp, scont, rdisp, rcont, comm)      ▷ all-to-all data exchange ◁
17.      if( $p < \tau_s$) then
18.          B = $SdssMergeAll(B,\ rdisp,\ rcont,\ p,\ c)$      ▷ SdssMergeAll is described in § 2.6 ◁
19.      else
20.          B = $SdssLocalSort(B,\ s_f,\ c)$      ▷ SdssLocalSort is described in § 2.6 ◁
21.      endif
22.  else      ▷ overlap all-to-all exchange and local sorting ◁
23.      C = $Memory\_Alloc(m)$
24.      $aid[0,\ldots,p]$=$SdssAlltoallvAsync(A$, C, $sdisp$, $scont$, $rdisp$, $rcont$, $comm$) ▷ SdssAlltoallvAsync is described in § 2.6 ◁
25.      while( ($id_1$, $id_2$) = $SdssFinished(aid)$ != (NULL, NULL) ) do      ▷ SdssFinished is described in § 2.6 ◁
26.          B = $SdssMergeTwo(C,\ rdisp,\ rcont,\ (id_1, id_2),\ c)$      ▷ SdssMergeTwo is described in § 2.6 ◁
27.      end while
28.  end if
```

Figure 1: SDS-Sort algorithm overview

ditionally, SDS-Sort accepts three parameters, τ_m, τ_o, and τ_s, to determine the merging for data exchange, overlapping data exchange with local ordering, and choosing sorting or merging to perform local ordering. Even though determining optimal values for these parameters is out the scope for this paper, we will provide their empirical optimal values in our test results section § 4.1.1.

To ensure the quality of global pivots selected, SDS-Sort starts a local pivots selection process on each MPI process by first sorting the local content of A with the function *SdssLocalSort*. This function is described in § 2.2. Then, SDS-Sort detours to merge the data from all processes located at the same node when the average message size ($\frac{n}{p}$) is smaller than τ_m. Merging the data before actual local pivot selection can determine the number of local pivots to select. More importantly, local node based merging provides SDS-Sort adaptive capability for the network with different throughput and the computing node with different numbers of CPU cores. To support the merging at each node, SDS-Sort refines its communication through *SdssRefineComm* (discussed in § 2.3) and updates the number of processes (p), that would be processing data from there onwards. The function *SdssNodeMerge* (see § 2.3) is used to merge data at each node. In step 8 of Fig. 1, SDS-Sort selects local pivots (stored in p_l) using equal striping length $\lfloor \frac{n}{p} \rfloor$, also called regular sampling [19]. Based on local pivots, the global pivots (stored in vector p_g) are chosen using the function *SdssSelectPivots* which is described in § 2.4.

After the global pivots are determined, the data is partitioned with function *SdssPartition*(described in § 2.5). Especially, SDS-Sort uses local pivots (p_l) to speedup the partition function *SdssPartition*. Meanwhile, *SdssPartition* is skew-aware and therefore it can partition the skewed data evenly among the processes. From steps 10 to 13, the displacements (i.e., *sdisp* and *rdisp*) and amounts (i.e., *scount* and *rcount*) required for the data exchange are computed. In step 14, the memory space for storing the sorted data in B is allocated. When one requires stable sorting or the number of processors is larger than τ_o, SDS-Sort does not overlap data exchange and local ordering steps. In this case, SDS-Sort uses *MPI_alltoallv* to exchange data in step 16. After the data exchange is finished, the received data is local ordered. The local ordering step uses *SdssMergeAll* (described in § 2.6) when the number of processes is less than τ_s. Otherwise, it calls *SdssLocalSort* to obtain globally ordered data. Choosing different methods for local ordering enables SDS-Sort to efficiently work on partially ordered data. When the number of processes is smaller than τ_o and the stable sorted is not required, SDS-Sort calls *SdssAlltoallvAsync* and *SdssFinished* (§ 2.6) to overlap data exchange and local ordering. In this case, SDS-Sort uses *SdssMergeTwo* (§ 2.6) to merge the received data. We elaborate the steps involved in SDS-Sort in the following subsections.

2.2 Skew-aware merging based local sorting

In line 2 and line 20 of SDS-Sort (Fig. 1), a shared memory parallel sorting algorithm (*SdssLocalSort*) is required. Sorting local data at the beginning of SDS-Sort (line 2) helps the sampling step (line 8) to thoroughly measure the value distribution and also the final local ordering step (line

20) to quickly process the partially ordered data. A popular strategy to sort an array on a shared memory machine with c CPU cores is to divide the array into c chunks; sort each chunk in parallel; and then merge these chunks in parallel [11]. As sorting on each core is a straight forward method, it is important to design an efficient parallel merging method, especially for skewed data. The designed *SdssLocalSort* function can quickly merge sorted chunks from multiple cores via its skew-aware partition.

Our implementation of *SdssLocalSort* accepts three parameters: a vector of data to sort (A), the number of CPU cores (c) , and the stable sorting flag (s_f). In line 2 of SDS-Sort with $c = 1$, the sequential version of *SdssLocalSort* is used. Internally, *SdssLocalSort* partitions its input A into c chunks. Then, depending on stable flag s_f, *SdssLocalSort* calls *std::sort* (when s_f=**FALSE**) or *std::stable_sort* (i.e., s_f=**TRUE**) of C++ library [27] to sort each chunk. Finally, these sorted chunks are merged to be a single vector in parallel using OpenMP. A sampling based parallel merging for shared memory local sorting was proposed in previous research [28]. This method could suffer from load imbalance issue in merging skewed data as it might cause one CPU core to be assigned with more data to merge than the others. In *SdssLocalSort*, we use the same skew-aware partition method as described in the following sub-section (§ 2.5) to partition each sorted chunk into subchunks and then merge these subchunks in parallel. Hence, basically, *SdssLocalSort* is a shared memory version of SDS-Sort without network connection.

2.3 Node-level merging

When sorting data on low-throughput network, merging sorted data from all CPU cores on a single node can reduce the number of messages and therefore reduce the network initialization overhead for the data exchange step. However, the same approach might not fully utilize the high bandwidth network available on the current-generation supercomputer systems because a single process running on a CPU core does not have the processing power to saturate the network. To address this issue, SDS-Sort adaptively decides whether or not merging the data at each node before the data exchange process (from line 3 to line 7 of Fig. 1). Specifically, we assume the average exchange size for data exchange is $\frac{n}{p}$. When $\frac{n}{p}$ is smaller than τ_m, SDS-Sort merges the data package at each node. This approach is suitable for low-throughput network. When the average data volume is larger than τ_m, each process sends its own data records to their respective destinations. This approach allows SDS-Sort to quickly feed all data into high throughput network.

To implement node level merging, we use function *SdssRefineComm* to create two MPI communicators: c_g and c_l. *SdssRefineComm* uses *MPI_Comm_split_type*[2] of MPI with the *MPI_COMM_TYPE_SHARED* parameter. The local communicator c_l is used by *SdssNodeMerge* to merge the data from all CPU cores into a single core. *SdssNodeMerge* shares the same idea as the skew-aware merging used in local sorting in previous section. One difference is that *SdssNodeMerge* uses network communication instead of memory copying. At line 8, the global communicator *comm* that is used for the following all-to-all data exchange phase that occurs later is replaced with global communicator c_g.

[2]http://www.open-mpi.org/doc/v1.8/man3/MPI_Comm_split_type.3.php

2.4 Regular sampling and pivots selection

Sampling method is used to choose local pivots on each process and global pivots at the global scale. The local pivots are chosen from original data and global pivots from local pivots. SDS-Sort uses equal-striped sampling method (also called regular sampling [19]) to choose both local and global pivots. In line 8 of Fig. 1, $p - 1$ local pivots are selected at regular striping size $\lfloor \frac{n}{p} \rfloor$. Since each node sorts the data at the beginning, these $p - 1$ local pivots can represent its local value distribution very well. In this case, each local pivot represents at most $2\frac{N}{p^2}$ values. To choose $p - 1$ global pivots from local pivots, a popular method is to gather all local pivots onto a single process, sort all local pivots, and choose the global pivots at equal-striped distance p. In this case, each global pivot represent at most $2\frac{N}{p^2} \times p = 2\frac{N}{p}$ values. SDS-Sort uses this regular sampling to select pivots and incorporates new optimizations from existing work in implementations, as discussed in next paragraph.

Such sampling method is simple and efficient but when p is large, these $p(p-1)$ local pivots might overflow the memory of single process. There are two solutions to address this issue. The first one is histogram sorting [24], where each node builds a histogram for a common global pivot vector and uses a single node to gather the histograms to choose global pivots. In sorting the non-skewed data that has a small number of replicated values, histogram sorting is useful as it can choose distinctive global pivots easily. But for the skewed data with highly replicated values, histogram sorting might need secondary sorting keys to distinguish the same values. The second method is to use the parallel sorting algorithms, which do not require gathering local pivots onto single node [28]. In SDS-Sort, we use the second approach and choose bitonic sort [4] to select pivots (*SdssSelectPivots*). Although bitonic sort needs a few data communications, the performance of botonic sort is acceptable in sorting $p(p-1)$ local pivots with p processes.

2.5 Fast and skew-aware partition

Equally partitioning data among all processes is key to ensure the load balance in the final ordering step. In sorting skewed data with highly replicated values, the selected global pivots might be replicated too. Using replicated global pivots to partition the data will cause serious load imbalance. Specifically, among all the processes that share the same global pivot, one of the processes could be assigned with all the data belonging to these same global pivots, while the other processes are assigned with no data. The process that is assigned all data ends up being the bottleneck while other processes will be idle in the final ordering step. Even worse, this might cause out-of-memory (OOM) errors and crash the sorting program. Adding the original rank or the non-key value (named secondary sorting key) of each data record to distinguish the replicated global pivots can avoid this issue, but it increases extra overhead of comparing and communicating one or more secondary sorting keys. Such limitation might let users have limited sorting keys to choose or spend extra efforts to pick up secondary sorting keys. Our SDS-Sort does not rely on secondary sorting keys to ensure the load balancing of local ordering. We also propose to use local pivots to partition data and reduce the overhead of data partition. Our partition method named *SdssPartition* is summarized in Fig. 2 and its details are discussed in below subsections.

```
function SdssPartition($A$, $P_l$, $P_g$, $p$, $r$, $s_f$, $com$)
    $A$ : sorted data with size $n$      $p$ : number of processes
    $P_l$ : local pivots                  $r$ : rank of processes
    $P_g$ : global pivots                 $s_f$ : flag of stable sorting
    $com$ : MPI communicator for all process
 0.   $rdisp[0] = 0$      ▷ Initialization ◁
 1.   for $i = 0, \ldots, p-2$ do
 2.     $p_i$ = std::upper_bound($P_l$[0], $P_l$[p-1], $P_g[i]$)
 3.     $p_d$ = std::upper_bound(A[$p_i \lfloor \frac{n}{p} \rfloor$], A[$(p_i+1)\lfloor \frac{n}{p} \rfloor$], $P_g[i]$)
 4.     $(f_r, r_s, r_r, pp_v)$ = SdssReplicated($P_g$, $p$, $i$)
 5.     if $f_r$ == TRUE then ▷ Replicated pivots detected ◁
 6.         $pp_i$ = std::upper_bound($P_l$[0], $P_l$[p-1], $pp_v$)
 7.         $pp_d$ = std::upper_bound(A[$pp_i \lfloor \frac{n}{p} \rfloor$],
                A[$(pp_i+1)\lfloor \frac{n}{p} \rfloor$], $pp_v$)
 8.         if $s_f$ != TRUE then ▷ Fast version ◁
 9.             $rdisp[i+1] = pp_d + \frac{p_d - pp_d}{r_s}(r_r+1)$
10.         else     ▷ Stable version ◁
11.             $c_r$= $pp_d$ - $p_d$ + 1
12.             MPI_Allgather($c_r$, 1, $c_v$, $p$, $com$)
13.             $s_b$ = sum($c_v[0, r-1]$)
14.             $s_a$ = sum($c_v[0, p-1]$)/$r_s$
15.             $r_t = \frac{s_b}{s_a}$       ▷ skip $r_t$ process ◁
16.             for $k$ = 0, ... , ($r_t$ -1) do
17.                 $rdisp[i+k+1] = pp_d$
18.             for
19.             for k = $r_t$, ... , $r_s$ do
20.                 if $s_b \% s_a + c_r \le s_a$ then
21.                     $rdisp[i+k+1] = pp_d + c_r$
22.                 else     ▷ Split replicated on a node ◁
23.                     $rdisp[i+k+1] = pp_d + s_a$
24.                     $s_b = s_b + s_a$, $c_r = c_r - s_a$
25.                 end if
26.             for
27.             $i = i + r_s$;
28.         end if
29.     else     ▷ No replicated pivots ◁
30.         $rdisp[i+1] = p_d$
31.     endif
32.   endfor
33.   $scount[0, \ldots, p-1]$ = difference($rdisp$)
34.   return ($rdisp$, $scount$)
```

Figure 2: SdssPartition Algorithm.

2.5.1 Local pivots based partition

The data partitioning step uses the $p-1$ global pivots (line 11 of Fig. 1) to partition the whole data space into p chunks. Specifically, the goal of data partition is to find the pairs of *sdisp* and *scount* for all-to-all data exchange. The *sdisp* variable is the starting displacement of data in A and the *scount* is the amount of the data after *offset*. A widely used data partitioning method is to shift through the total $O(n)$ local data once. When n is large, shifting all data may have significant overhead. To reduce this overhead, SDS-Sort uses local pivots-based partition to reduce its shift space from $O(n)$ to $O(\frac{n}{p})$. It is because local pivots and their associated displacements provide good representation and partition of the sorted local data space. The algorithm to perform this partition is shown in Fig. 2 (from line 1 to line 4). For a global pivot $P_g[i]$, SDS-Sort firstly ranks it among $p-1$ local pivots via *std::upper_bound* function of C++, which is based on binary search [27]. Then, SDS-Sort uses *std::upper_bound* again to find the actual displacement of pivot $P_g[i]$ between $P_{ld}[p_i]$ and $P_{ld}[p_i+1]$ of A.

2.5.2 Skew-aware partitioning

Next, we describe the skew-aware data partition method. A key factor affecting the partition sizes in a parallel sorting procedure is the presence of replicated global pivots. To enable *SdssPartition* to detect replicated global pivots dynamically (line 5 of Fig. 2), we designed an algorithm named *SdssReplicated* (in Fig. 3). For a specific global pivot $P_g[i]$, *SdssReplicated* scans all $p-1$ global pivots once and identifies that "is $P_g[i]$ duplicated with its neighborhood pivots?" If $P_g[i]$ is not a replicated pivot (f_r=**FALSE**), SDS-Sort partitions the data as traditional method [19] (line 30 of Fig. 2). Once $P_g[i]$ is found replicated (f_r=**TRUE**), *SdssReplicated* continues to find "what is the number (i.e., r_s) of pivots that are equal to $P_g[i]$?" and "what is the rank (i.e., r_r) of $P_g[i]$ in its replicated pivots?" When *SdssReplicated* detects replicated pivots, it also finds the pivot (pp_v) right before all replicated $P_g[i]$. The r_r, r_s, and pp_v are used to evenly partition the skewed data among processes, as discussed in rest of this subsection.

```
function SdssReplicated($P_g$, $p$, $i$)
    $P_g$ : global pivots vector          $i$ : target process index
    $p$ : number of processes
 0.   $f_r$ = FALSE, $r_s$ = 1, $r_r$ = 0, $pp_v$ = $P_g[0]$, $j = i - 1$
 1.   while ($j >= 0$) and $P_g$[j] == $P_g$[i] do
 2.       $j--$, $r_s++$, $f_r$ = TRUE
 3.   end while
 4.   $pp_v$ = $P_g$[j]
 5.   $r_r$ = $r_s$ - 1, j = i + 1
 6.   while ($j < p-1$ and $P_g$[j] == $P_g$[i] ) do
 7.       $j++$; $r_s++$; $f_r$ = TRUE
 8.   end while
 9.   return ($f_r$, $r_s$, $r_r$, $pp_v$)
```

Figure 3: SdssReplicated Algorithm

Since the requirement to maintain stability requires us to handle the same key value differently, next we discuss the two cases of partitioning with and without stability requirement separately. We will refer to the version without stability requirement as the **"fast version"** and the version with stability requirement as the **"stable version"**.

Fast version of skew-aware partitioning. Without duplicated keys, the theoretical upper bound for the number of data records for each process in local ordering step is $O(2\frac{n}{p})$ [19]. When sorting skewed data, this upper bound of existing methods will increase proportionally as the number of replicates increases [19]. Using secondary sorting key can alleviate this issue, but, as mentioned above, requires extra overhead in comparing and communicating. Without rely on secondary keys in sorting skewed data, we device a more efficient partitioning method in SDS-Sort. More importantly, this new partition method has fixed upper bound of $O(4\frac{n}{p})$ for the number of data records for each process in local ordering step (see proof in §2.8). The partitioning method is presented from line 6 to line 9 of Fig. 2. First, SDS-Sort finds the index (pp_i) and the displacement (pp_d) of pp_v in P_l and A, respectively. Combined with the displacement p_d found in line 3, SDS-Sort knows that all replicated values fall between pp_d and p_d. As the fast version of SDS-Sort does not require to maintain the relative order of replicated data, SDS-Sort equally partitions the replicated values between pp_d and p_d among all r_s process (at line 9 of Fig. 2). In other words, the partitioning method is equal to implicitly adding the rank of replicated pivots (i.e, r_r) to distinct the replicated values between pp_d and p_d. A simple example of this partition method is presented in Fig 4.

Stable version of skew-aware partitioning. In the stable version of SDS-Sort, the replicated values in output must have the same order as they have in the input. Especially,

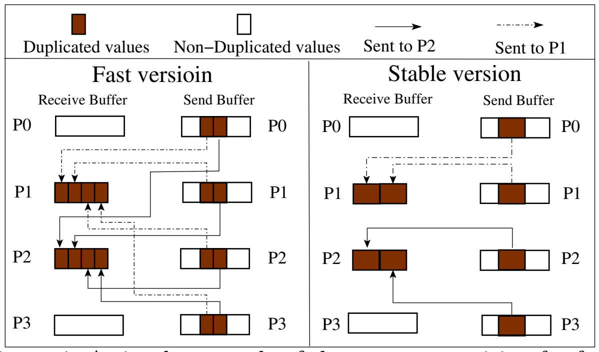


Figure 4: A simple example of skew-aware partition for four processes: P0, P1, P2, and P3; where two global pivots (out of three) are the same. In the fast version, each process partitions its own replicated values evenly into two chunks, denoted with brown boxes. The first brown box of all four processes is gathered onto P1 and the second brown box of all four processes onto P2. In the stable version, all replicated values are partitioned into two groups. P0 and P2 belongs to the first group and P2 and P3 the second group. All brown boxes on P0 and P1 are sent to P1 and all brown boxes on P2 and P3 are sent to P2.

the duplicated values located on different processes must be ordered by their MPI process rank in final output. The partition method used by the fast version of SDS-Sort can not guarantee this stability request. For instance, in the fast version example of Fig 4, process P3 sends its replicated values to both P1 and P2. In the final output, the replicated value sent from P0 to P2 could be placed between these replicated values sent by P3, which breaks the requirement of stable parallel sorting.

To address this issue, SDS-Sort uses a new partitioning method for the stable version. The main idea of our method is to consider all duplicated values from the first MPI process to the last MPI process as a contiguous 'replicated value space', to evenly partition this space into groups, and then to designate an MPI process in each group for gathering and storing all the replicated values in its respective group. When there are r_s replicated pivots, the number of groups is set equal to r_s. These designated processes are the processes assigned with duplicated global pivots. As each designated process only gathers equally sized replicated values inside its group, the load on these processes will be very small and balanced. The upper bound on the load of each process of this partition method is $O(4\frac{n}{p})$, the same as the fast version (see proof in Section 2.8). The implementation details are presented from line 11 to 25 of Figure 2. SDS-Sort first lets each process gather duplicated value count (c_r) of all p processes into a vector (sv), which has the size equal to p. Since we have r_s groups and also r_s designated processes, the average size of replicated values for each designated process is calculated as $s_a = \text{sum}(c_v[0, p-1])/r_s$. The first designated process will gather s_a duplicated values from the first process p_0 until the process p_i, where $i \geq 0$ and $\text{sum}(c_v[0, i-1]) \leq s_a \leq \text{sum}(c_v[0, i])$. Note that when a single process has more replicated values than s_a, these replicated values will then be divided into different groups (line 23 and line 24). By applying the same idea, each designated process can find its s_a contiguous duplicated values within its group. In the gathering process, we preserve the order of value as their MPI process rank by taking advantage of the blocking (non-asynchronous) MP_alltoallv function (line 16 of Fig. 1). A very simple example of the partition method is also shown in Fig 4, where the number of replicated values on P0 and P2 is equal to that on P2 and P3.

2.6 Adaptive all-to-all data exchange

When performing the data exchange, SDS-Sort can dynamically choose between two options: synchronous and asynchronous. The asynchronous data exchange is used to overlap data exchange and local ordering phases. This option is effective when CPU is relatively fast compared to the network, where the CPU could process the data to be sent without waiting for the receiver to acknowledge the message, and then wait to perform the local ordering operations as soon as the data arrives. On HPC systems with relatively weak CPUs, the CPUs might have to devote a large portion of its computing power to feed the network with no opportunity to conduct other operations. Furthermore, asynchronous communication requires the senders nad receivers to dedicate a certain amount of system resources to monitor the progress of the messages. In a large scale all-to-all data exchange, the competition for these system resources could introduce unexpected delays and reduce the overall performance. In such a case, using synchronous communication might be faster. In the current implementation, we use a threshold τ_s, certain number of processes, to choose synchronous and asynchronous data exchange adaptively.

As the asynchronous might break the order of data exchange but the stable sorting requires to maintain relative order, SDS-Sort uses synchronous to perform all-to-all data exchange for stable sorting. SDS-Sort uses *MPI_Alltoallv* to perform synchronous data exchange (line 16 of Fig. 1) and *MPIAlltoallvAsync* (line 24 of Fig. 1) to perform asynchronous data exchange. Note that *MPIAlltoallvAsync* is not a standard MPI function, but a function we implemented with *MPI_Isend*, *MPI_Irecv*, and *MPI_Test* functions. *MPIAlltoallvAsync* returns two received data chunk ids: id_1 and id_2, corresponding to the ranks of two processes. Function *SdssMergeTwo*, a special case of *SdssMergeall* reported in following subsection 2.6, is used to merge two received data chunks.

2.7 Adaptive local ordering

After the all-to-all data exchange, SDS-Sort performs the final local ordering within each process to place the data records in their output order. In this process, SDS-Sort could dynamically decide to use a number of different procedures as shown from line 17 to line 21 of Fig. 1. By design, the input to this final ordering step on each MPI rank is a list of p ordered chunks. Currently, we only consider two options named merging and sorting, where the first option merges the p ordered chunks into a single order array and the second option simply invoke a standard sorting algorithm on the incoming data. The reason is that the complexity of merging increases as the number of processes p increases, but the complexity of sorting decreases as p increases. Specifically, the time complexity of merging p sorted chunks (received from p processes) is $O(nlog(p))$, which highly depends on the number of sorted chunks. On the other hand, p sorted chunks form a partially ordered data. The best complexity of sorting partially ordered reduces from $O(nlog(n))$ to $O(n)$ [9]. It is obviously that given certain number of processes p, choosing sorting or merging wisely can reduce the time spent on local ordering. In implementation, we uses two functions: *SdssMergeAll* and *SdssLocalSort*. Both can run in parallel on shared memory if its input parameter c is larger than one. *SdssMergeAll* takes advantage of sorted order of the chunks from other processes and uses *std::merge* of C++ to obtain globally sorted data. As described in previous Section 2.2, *SdssLocalSort* is a shared memory sorting algorithm which is based on *std::sort* or *std::stable_sort* of C++.

2.8 Analysis of workload on each process

Following the analysis by Li et al. [19], we use m_i to denote the number of data records on the i^{th} process in the final local ordering step and

$$U = \max_{1 \leq i \leq p} m_i = 2\frac{N}{p} - \frac{N}{p^2} - p + 1 + d$$

where N is the number of data records, p is the number of processes used, and d is the number of the records whose key value is duplicated most. In the analysis done by Li et al., the big O notation is also used to denote the dominant term (also called the 'upper bound') in the expression. Assuming both p and d are much smaller than N, Li et al. express the upper bound of U as $O(2\frac{N}{p})$. Using these conventions, we have the following theorem:

THEOREM 1. *The upper bound of workload on each process (U) with SDS-Sort in sorting N data records using p processes is $O(4\frac{N}{p})$.*

PROOF. We divide the proof into two parts based on whether or not there are any duplicated global pivots.

When there are no duplicated global pivots, the worst case bound on U is same as given by the analysis of Li et al. [19]. In this case, the worst case value for d can be computed as follows. Since there is no duplicated global pivot, there are at most $p-1$ replicated local pivots. Since SDS-Sort orders original data before choosing local pivots, each local pivot represents at most $2\frac{N}{p^2}$ replicated values. Hence, we have $d < 2\frac{N}{p^2} \times (p-1) < 2\frac{N}{p^2} \times p = 2\frac{N}{p}$. Thus, $U < 2\frac{N}{p} - \frac{N}{p^2} - p + 1 + 2\frac{N}{p} = 4\frac{N}{p} - \frac{N}{p^2} - p + 1 + 2$, which gives that the upper bound of U as $O(4\frac{N}{p})$.

When there are duplicated global pivots, SDS-Sort denotes the number of duplicated global pivots with r_s and the rank of each replicated global pivot with r_r, where $1 \leq r_s \leq p-1$ and $0 \leq r_r \leq r_s - 1$. As the fast version and the stable version of SDS-Sort partition the data with different methods, we prove the upper bound for them separately as below:

- The fast version SDS-Sort uses r_s and r_k to partition the replicated values equally among r_s processes. In this partition method, a replicated global pivot implicitly assigns its rank value (i.e., r_r) to all duplicated values which are represented by it. This will make the duplicated values represented by different replicated global pivots are distinct from each other. Meanwhile, we have that each global pivot represents at most $2\frac{N}{p^2} \times p = 2\frac{N}{p}$ duplicated data records. Hence, the maximum number of the duplicated values reduce from d to $2\frac{N}{p}$. As indicating in above steps, the upper bound of sorting data with at most $2\frac{N}{p}$ values is $O(4\frac{N}{p})$. Therefore, we have that the upper bound of U is equal to $O(4\frac{N}{p})$ in this case.
- The stable version of SDS-Sort partitions the number of processes into r_s groups and uses a designated process within a group to gather the duplicated values. This partition method implicitly attach a rank value r_k to all the duplicated values within a single group. Thus, it can distinct the values belonging to different groups. As the number of duplicated values of a single group is at most $2\frac{N}{p^2} \times (p) = 2\frac{N}{p}$, which is also equal to the number of data records represented by a single pivot, the maximum number of the duplicated values reduce to be $2\frac{N}{p}$. As indicating in above steps, we have that upper bound of U is equal to $O(4\frac{N}{p})$ in this case. This completes the proof.

□

3. SYSTEM CONFIGURATION

We have conducted all the experiments reported in this paper on Edison [3], a Cray XC30 supercomputer at the National Energy Research Scientific Computing Center (NERSC). Edison is equipped with 133,824 compute cores and 357 terabytes of aggregate memory. Each compute node of Edison is configured with two 12-core Intel "Ivy Bridge" processors at 2.4 GHz and 64 GB DDR3 1600 MHz memory. Edison uses Cray Aries [6] high-speed interconnect, which has $0.25\mu s$ to $3.7\mu s$ MPI latency and 8GB/sec MPI bandwidth. The high-speed interconnect of Edison uses Dragonfly topology, which is able to deliver 23.7TB/s global bandwidth. Our SDS-Sort implementation code is written in C++ and is compiled with Intel Compiler version 16.0 and Cray's implementation of MPI. We have scaled our sorting experiments up to 131,072 CPU cores. The measured performance in this paper does not include the time to read the data from the parallel file system, i.e., the measured time includes sorting time after the data is loaded into memory.

4. EXPERIMENTAL EVALUATION

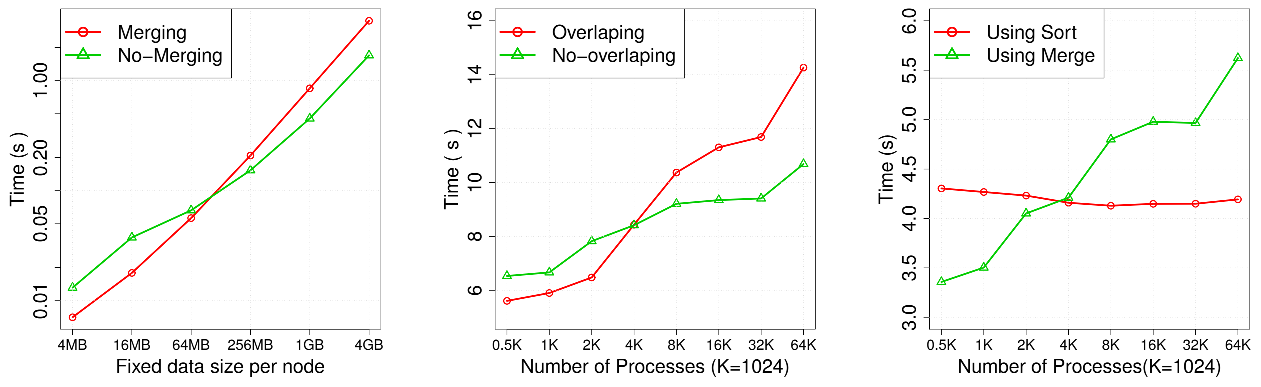
In this section, we conduct an in-depth evaluation of SDS-Sort, and compare it with the most efficient in-memory sorting algorithm known as Hyksort [28]. The evaluation uses two synthetic data sets and two real scientific data sets from cosmology and astronomy.

4.1 Performance evaluation of SDS-Sort

Two synthetic data sets, a Uniform data set and a Skewed data set, are used to evaluate SDS-Sort. Uniform data set is generated by standard Uniform distribution, which is widely used in previous research to test various parallel sorting algorithms [28]. The Skewed data set is generated by Zipf distribution: $p(i) = \frac{C}{i^{\alpha}}$, where $i = 1$ to N, α is the Zipf exponent, and C is the normalization constant [26]. Since the number of duplicates is a critical parameter to the performance of parallel sample sorting algorithms, we next introduce a parameter δ as the maximum replication ratio. For a data set with d denoting the number of the records whose key value is duplicated most, δ is defined as $\frac{d}{N} \times 100\%$, where N is the number of data records to sort. All tests were repeated three times and the best performing values are reported.

Using these synthetic workloads, we first explore the optimal values for the parameters τ_m, τ_o, τ_s that are used by SDS-Sort. Next, we explore the performance characteristics

[3]http://www.nersc.gov/users/computational-systems/edison/



(a) All-to-all data exchange with or without merging.

(b) Overlapping and not-overlapping data exchange with local-sorting.

(c) Final local ordering using sorting vs. using merging.

Figure 5: Performance test results of exploring optimal value for parameters τ_m, τ_o, and τ_s.

of *std::sort* and *std::stable_sort* from C++ Standard Template Library as SDS-Sort uses them to perform sequential sorting on a CPU core. Finally, we compare SDS-Sort with a state-of-art parallel sorting algorithm, namely HykSort [28]. HykSort has a parameter, k, representing k-way communication. Previous study [28] shown that 128 is the optimal for k, which is used in our evaluation. Specifically, we compare SDS-Sort and HykSort in terms of different replication ratio values (i.e., δ) and varying number of MPI processes.

4.1.1 *Evaluation with varying parameters*

The SDS-Sort algorithm has a number of parameters, such as the threshold τ_m for merging data in the exchange phase, the threshold τ_o for overlapping communication with computation, and the threshold τ_s in the local ordering phase. Next, we use the synthetic data sets to find the optimal values of these parameters.

Performance with merging data before the exchange phase (τ_m). The merging parameter (τ_m) of SDS-Sort decides whether to merge data of each CPU node for the all-to-all data exchange phase. Fig. 5a reports the execution time for all-to-all data exchange with varying message size. It is clear that when the message size is small (i.e., less than 160MB), merging data at each node is beneficial. On the other hand, when the message size is larger than 160MB, merging data at each node has high overhead. The main reason is that merging small messages on each node can avoid the overhead of establishing all-to-all communication. But, when the message size is large, using all the CPU cores to feed data individually into the network without merging can take advantage of high bandwidth of the network. Our test results indicate that setting τ_m to be 160MB is reasonable on our test bed, Edison.

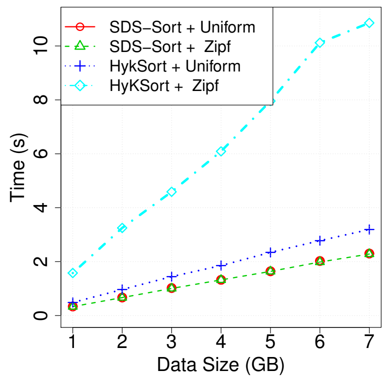
Overlapping of the exchange and the local ordering phases (τ_o). The threshold τ_o decides whether overlapping the all-to-all exchange with the local sorting phases of SDS-Sort. To explore the optimal values of τ_o, we tested the time for overlapping and not overlapping using different numbers of MPI processes. The results reported in Fig. 5b show that overlapping all-to-all data exchange with local ordering is faster than not overlapping when the number of processes is smaller than 4096. A reason for this behavior is that when the number of processes is small, the network bandwidth that the sorting phase obtains is small. As a result, when data is transferred on network, the CPU might be idle and overlapping data exchange and local ordering can reduce the overall time. As the number of processes increases, the workload that local ordering phase requires to do increases too. Hence, overlapping the all-to-all data exchange and local ordering phases can delay the rate of feeding the data into network. As a result, the performance with overlapping degrades. Through our tests on Edison, we decide the optimal τ_o to be 4096.

Table 1: Time (s) of using *std::sort* and *std::stable_sort* of C++ to sort $1GB$ data.

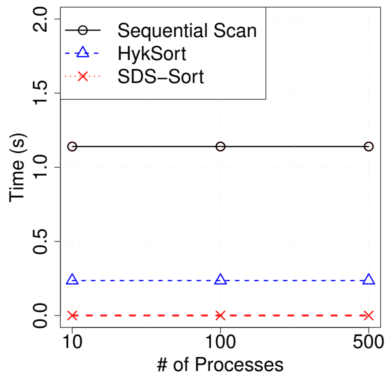
	Uniform	Zipf (a (δ%))		
		0.7 (2)	1.4 (32)	2.1 (63)
std::sort	26.1	14.6	8.9	6.6
std::stable_sort	35.2	24.3	16.5	12.5

Merging vs. sorting (τ_s). In SDS-Sort, τ_s decides whether to use merging or sorting to perform the local ordering phase. In our analysis in Section 2.6, we argue that the time for using merging to perform local ordering phase will increase as the number of processes increases. On the other hand, the time to sort partially sorted data will reduce. We report the time used by sorting and merging with different number of processes in Fig. 5c. As expected, the time for using merging to perform local ordering phase increases sharply from 512 processes to $64K$ processes. On the other hand, the time for using sorting on CPU cores to perform local ordering is much more stable and decreases gradually. Hence, the test results are consistent with theoretical analysis. 4000 MPI processes is the turning point where using merging becomes more expensive than sorting. Hence, in following tests on Edison, τ_s is set to be 4000.

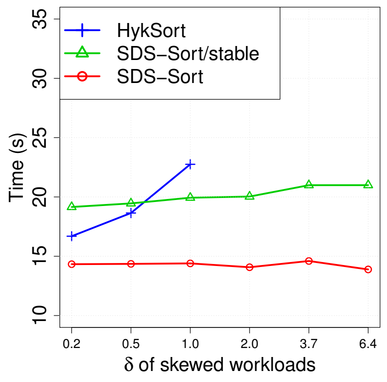
Evaluation of *std::sort* and *std::stable_sort*. The standard C++ functions *std::sort* and *std::stable_sort* are used as the sequential sort algorithm in SDS-Sort. In Table 1, we show the performance of sorting 1 GB data (268 millions float values) from both uniform and skewed data sets. In the skewed data set tests, we used skewed data sets with different Zipf distribution settings. As expected, *std::sort* is faster than *std::stable_sort*. The time to sort highly skewed data is smaller than sorting the uniform distributed data set. Moreover, the time to sort skewed data gradually decreases as the replication ratio (δ) increases.



(a) Time to merge data in parallel for different workloads on a single node.



(b) Time to partition data using different methods



(c) Time to sort skew data with varying replication ratios (δ, %).

Figure 6: Micro performance comparison of different optimizations: skew aware merging, local pivots based parallel partition, and skew aware sorting.

Table 2: Relatioship between δ and α.

α	0.4	0.5	0.6	0.7	0.8	0.9
δ (%)	0.2	0.5	1.0	2.0	3.7	6.4

4.1.2 *Large-scale comparison with HykSort*

In this section, we compare SDS-Sort and HykSort[24] using the synthetic data sets. We first compare their two components: shared memory parallel merging and data partition methods. Then we compare SDS-Sort with HykSort to sort the skewed data set with different replication ratios and to run using on different numbers of CPU cores. We label *SDS-Sort* to denote the fast version and *SDS-Sort/stable* to denote the stable stable version of SDS-Sort.

Skew-aware parallel merging. Parallel merging is an important step in functions *SdssLocalSort*, *SdssMergeSort* and *SdssNodeMerge*. Compared with the parallel merging used by HykSort, SDS-Sort uses a skew aware partition method to support parallel merging. In Fig. 6a, we compare the time to merge different data workloads and different data sizes. The parallel merging used in HykSort uses more time when merging skewed data represented by Zipf distribution. This is consistent with the analysis by Li et al [19], which indicates that a high number of duplicated values would increase the load imbalance. On the other hand, the skew-aware parallel merging used in SDS-Sort delivers stable performance in both Uniform and Zipf workloads because the skew-aware parallel merging is better at maintaining load balance.

Local pivots-based data partition. In Section 2.5, we propose the function *SdssPartition* that uses local pivots to partition data. Here we evaluate this idea experimentally. In this test, we fix the data size for each process at 2GB and test different number of processes. The test results are reported in Fig. 6b. In this test, we compared the performance of full scan partition, parallel partition used by HykSort, and local pivot based partition used in SDS-Sort. As the test results indicate that using local pivots can reduce the time for data partition to almost zero. Hence, the proposed local pivots based data partition is an efficient method to reduce the data partition overhead.

Replication ratios δ scaling tests. Next, we compare SDS-Sort with HykSort using the skewed workloads of different replication ratios. The performance comparison is shown in Fig. 6c. The α and δ values in the test data are reported in Table 2. The timing results in Fig. 6c indicate that both SDS-Sort and SDS-Sort/Stable deliver scalable performance with different replication ratios. On the other hand, HykSort can only work when the replication value is less than 1.0% (δ=0.6). The reason is that skewed data causes load imbalance in HykSort. When the replication ratio is too high, certain nodes will be assigned so much data that the processes running on those nodes run out of memory.

Scalability of SDS-Sort. We show weak-scaling performance of SDS-Sort on uniform and skewed data sets in Fig. 7 and Fig. 8, respectively. In these tests, we fix the data size per process at 400MB (i.e., 100 millions records) and increase the number of CPU cores from 512 (0.5K) to 131,072 (128K). The data size for sorting with 128K cores is 52.4TB (i.e., 10^{13} data records). For uniform workload, HykSort takes 42.6 seconds to sort the 52.4TB. Using sorting throughput, a popular metric [29], to express this performance number, HykSort archives at most 73.8TB/min sorting throughput using 128K cores. SDS-Sort takes 28.25 seconds to sort the same data, resulting in 111TB/min sorting throughput. SDS-Sort is 51% faster than HykSort. For the SDS-Sort/stable, it delivers 54TB/min in sorting the same data. The reason for SDS-Sort/Stable is slower than both HykSort and SDS-Sort is that it takes more time to select pivots and to perform local ordering as discussed in the previous sections. For the skewed workload, HykSort fails to execute due to the Out-of-Memory (OOM) error because of the load imbalance issue after the all-to-all data exchange phase. Both SDS-Sort and SDS-Sort/stable deliver performance similar to that of sorting the uniform data set. Using 128K processes, SDS-Sort delivers 117TB/min sorting throughput, and SDS-Sort/stable delivers 55.8TB/min sorting throughput.

We now evaluate the impact of load balancing of different sorting algorithms in the above scaling tests. Popular metric used to compare the load balancing of sorting algorithms is $RDFA$, which is the Relative Deviation of the size of the largest partition From the Average size of the p processes, and is defined as: $RDFA = \frac{\max_{i=1}^{p}(m_i)}{ave_{i=1}^{p}(m_i)}$ [19], where m_i de-

Table 3: RDFA of different parallel sorting algorithms

		Number of Cores								
		512	1024	2048	4096	8192	16384	32768	65536	131072
Uniform	HykSort	1.0692	1.0433	1.0232	1.0145	1.0126	1.0096	1.0085	1.0073	1.2051
	SDS-Sort	1.0025	1.0044	1.0049	1.0076	1.011	1.0177	1.0264	1.0353	1.0546
	SDS-Sort/stable	1.0025	1.0044	1.0049	1.0076	1.011	1.0177	1.0264	1.0353	1.0546
Zipf(0.7-2.0)	HykSort	∞	∞	∞	∞	∞	∞	∞	∞	∞
	SDS-Sort	1.6816	1.8172	1.8411	1.9222	1.9552	1.9556	1.9732	1.4889	2.6753
	SDS-Sort/stable	1.6816	1.8172	1.8411	1.9222	1.9552	1.9556	1.9732	1.4888	2.6753

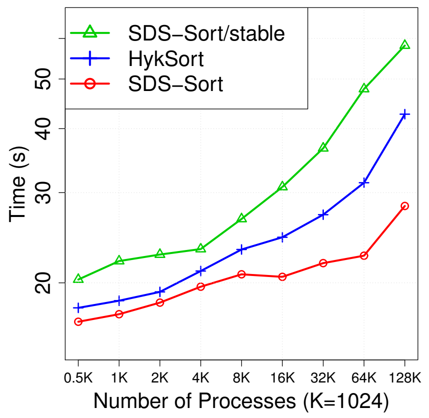


Figure 7: Scaling test of Uniform distribution

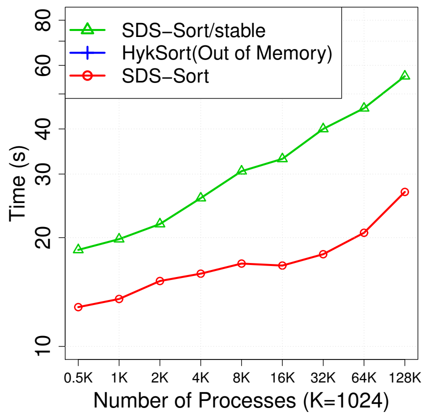


Figure 8: Scaling test of Zipf distribution

note the load on the ith processes. The $RDFA$ of all scaling tests are reported in Table 3. HykSort uses histogram sampling to choose pivots. SDS-Sort uses striped-equal sampling (also called regular sampling) to choose pivots. For the Uniform distribution workload, we can see that HykSort, HykSort/stable and SDS-Sort have almost equal RDFA values. The difference of RDFA values between these sorting methods is negligible. For the skewed data set generated using Zipf distribution, both SDS-Sort and SDS-Sort/stable deliver almost similar RDFA values. But, histogram sampling used in HykSort assigns a lot of replicated values to single node, which causes out-of-memory errors. Hence, we denote the RDFA values for HykSort as ∞ in these tests. Using external values or rank of replicated values to distinct the replicated one can turn HykSort to allocate replicated values among processes [29]. But, it requires extra overhead to store, exchange, and process external values. Also, user's objective selection for secondary sorting keys can impact the tests results. Hence, we only compare the method without using secondary sorting keys here. In summary, we can see that SDS-Sort works on different workloads and also show good scalability to different number of processes.

4.2 Evaluation of SDS-Sort with Real Application Data Sets

We next compare SDS-Sort with HykSort using two real scientific data sets from Palomar Transient Factory (PTF) observations and a Cosmology simulation of billion particles.

Palomar Transient Factory (PTF) data. The PTF is an automated survey system of the sky for identifying supernova and other transient events in the universe [5]. An important component of the survey is the automated transient detection pipeline to enable the early detection of these events. One task among this pipeline is to make automated real/bogus decision about each detected objects based on image and context features using real-bogus (RB) classifier [5]. In the PTF data sets, the RB classifier is represented by real-bogus score as a real number. Hence, sorting the objects by real-bogus score is one typical method used by the RB classifier. In our sorting tests, we used a 27GB data set (with 1 billion records) for measuring the performance of SDS-Sort and HykSort. Before SDS-Sort and HykSort starts to work, the time to read the data into memory is 19.6 seconds.

The PTF data shows high skewness as its replication ratio (δ) of real-bogus score is 28.02%. We show the performance of sorting PTF data using HykSort and SDS-Sort in Fig. 9[4]. The RDFA values of different sorting methods are reported in Table 4, where load imbalance with HykSort for PTF data is significantly larger than that of SDS-Sort. As each node of Edison has 64GB memory, the whole data can be stored on a single node. Hence even though HykSort has serious load imbalance (RDFA=32.68), it can still finish the sorting without out-of-memory (OOM) issues. Overall, SDS-Sort is 3.4× faster than HykSort in sorting PTF data and SDS-Sort/stable is 2.2× faster than HykSort.

Cosmology simulation data. Large-scale cosmological simulations such as GADGET-2 [25] and NyX [1] play critical roles in exploring the unknown structure formation process of the universe. Analyzing the data generated by cosmological simulations is essential step to extract its insights of cosmological discoveries. Recently, researchers proposed BD-CATS, a KD-tree based clustering method, to analyze the particle data from GADGET-2 [21]. An important step in BD-CATS is to sort the particles based its clustering ID.

[4]The Exchange time for HykSort also contains the time for local ordering as it uses overlapping inside

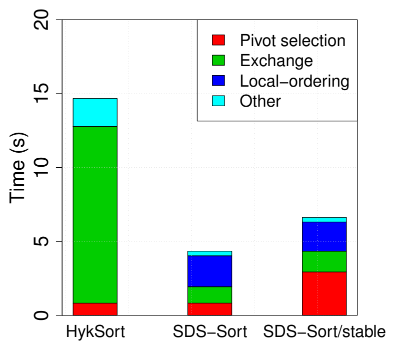


Figure 9: Sorting 27GB PTF data with 192 cores.

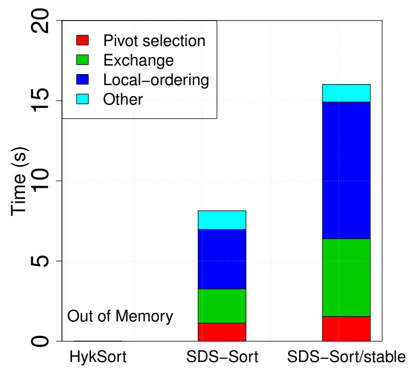


Figure 10: Sorting 2.1TB Cosmology data with $16K$ cores.

In this test, we apply both SDS-Sort and HykSort to sort a 2.1TB particle data by using its clustering ID as the sorting key. Before SDS-Sort and HykSort starts to work, the time to read the data into memory is 438.1 seconds. Specifically, the 2.1TB data contains 68 billion particles and we sort these particles by their clustering IDs. Meanwhile, each particle data record also contains spatial location (x, y, and z), and particle velocities (vx, vy, and vz). In sorting tests, we trade these extra attributes as payload. In the data, the replication ratio (δ) for clustering ID is 0.73. Hence, the cosmology data used in this test is a typical example of skewed data. The performance of sorting is reported in Fig. 10. The RDFA values of sorting this 2.1TB data are reported in Table 4, which are small for SDS-Sort. The RDFA value for HykSort is ∞ because that HykSort fails to sort the cosmology data set due to OOM errors. Both SDS-Sort and SDS-Sort/stable can sort the Cosmology data quickly, giving 15.63TB/min and 7.87TB/min sorting throughput, respectively.

5. RELATED WORK

Non-sampling based parallel sorting algorithms include bitonic sort[4], radix sort [30], bubble sort[3], merging sort[11], and so on. Bitonic sort [4] focuses on converting a random sequence of numbers into a bitonic sequence which monotonically increases and then decreases. Radix sort [30] is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits

Table 4: RDFA values of sorting Cosmology and PTF data

	HykSort	SDS-Sort	SDS-Sort/stable
PTF	32.6759	1.9908	1.6908
Cosmology	∞	1.3962	1.3962

that share the same significant position and value. Bubble sort [3] repeatedly steps through the data to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. Merge sort [11] divides data list into the small unit, then compare each element with the adjacent list to sort and merge the two adjacent lists. Generally, these non-sampling based parallel sorting algorithms need a significant amount of communication and data exchange, which are expensive operations on parallel systems [28].

Sampling based sorting algorithm was invented in 1970s [15] and it is based on the divide and conquer idea. Parallel sampling sorting [24] was then devised to sort large-scale data on distributed memory systems. A theoretical study on the load balancing of parallel sampling sorting algorithms was conducted in [19], where authors proved $O(2\frac{n}{p})$ upper bound for load balancing without highly duplicated keys. Parallel sorting by sampling was compared with other algorithms [2], e.g radix sort and bitonic sort. Authors found that sampling sorting is good at the distributed memory machine where interprocess communication and data movement are expensive.

To reduce the sampling size and improve scalability of parallel sampling sorting, Solomoik et al [24] use histogram-based method to choose pivots and propose to overlap data exchange and local ordering to reduce its overhead. In out-of-core parallel sorting, similar idea have been employed to overlap computation and disk I/O operations [28, 10]. CloudRAMSort [18] performs multi-node optimizations by carefully overlapping computation with inter-node communication. CloudRAMSort uses payload to be part of the key to deal with skewed data.

HykSort was devised to reduce all-to-all communication overhead via avoiding network transmission contentions [28]. As HykSort divides all process into groups to reduce transmission contention, the global pivots used in HykSort are reduced to k, a user controlled parameter. Based on the number of pivots, HykSort is general sampling sorting algorithm of quick sort [23] and standard parallel sampling sort [19]. To the best of our konwledage, HykSort is the fastest and the most scalable parallel sorting algorithm so far.

Other sorting algorithms, e.g., TritonSort [22] and NTOSort [13], also exist. Being different from the above sorting algorithms and from our SDS-Sort, these sorting algorithms are generally disk-based sorting algorithms, or usually called out-of-core sorting. These out-of-core sorting algorithms mainly focus on optimizing the I/O performance and cache efficiency in sorting.

6. CONCLUSIONS

The SDS-Sort algorithm, we introduced in this paper, addresses the issues inherited in parallel sorting algorithms on common application data sets, such as skewed data and partially ordered data. Furthermore, existing parallel sorting algorithms are not designed for heterogeneous architecture of current generation of supercomputers. SDS-Sort uses skew-aware partitioning method to address the load imbalance issue in working with skewed data. To run efficiently on new computing and networking hardware and to sort partially ordered data, SDS-Sort is capable of decide many aspects of its execution dynamically, such as merging data at each node, overlapping communication and computation, and selecting different approaches for local ordering. Through extensive experiments with both uniformly distributed and skewed synthetic workloads, SDS-Sort scaled to 131, 072 cores and was shown to be at 50% faster than a state-of-the-art parallel sorting algorithm, HykSort. On two sets of scientific data from astronomy and cosmology, SDS-Sort outperformed HykSort by 3.4X. In the future, we plan to systematically study the configuration parameters τ_m, τ_o, and τ_s. We plan to perform more comparisons against various parallel sorting methods and carry out more tests with well-known sorting benchmarks and scientific data sets. We are also interested in exploring the new hardware such as general graphics processing unit (GGPU).

Acknowledgment

This work was supported by the DOE Office of Science, Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231 (Program manager: Lucy Nowell). This research used resources of the National Energy Research Scientific Computing Center (NERSC).

7. REFERENCES

[1] A. Almgren, J. Bell, M. Lijewski, Z. Lukic, and E. Van Andel. Nyx: A Massively Parallel AMR Code for Computational Cosmology. *Astrophys. J.*, 765:39, 2013.

[2] N. Amato, R. Iyer, S. Sundaresan, and Y. Wu. A comparison of parallel sorting algorithms on different architectures. Technical report, TX, USA, 1998.

[3] O. Astrachan. Bubble sort: An archaeological algorithmic analysis. *SIGCSE Bull.*, 35(1):1–5, Jan. 2003.

[4] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical report, Ithaca, NY, USA, 1986.

[5] J. S. Bloom, J. W. Richards, P. E. Nugent, R. M. Quimby, M. M. Kasliwal, D. L. Starr, D. Poznanski, E. O. Ofek, S. B. Cenko, N. R. Butler, S. R. Kulkarni, A. Gal-Yam, and N. Law. Automating discovery and classification of transients and variable stars in the synoptic survey era. *Publications of the Astronomical Society of the Pacific*, 124(921):pp. 1175–1196, 2012.

[6] L. K. Bob Alverson, Edwin Froese and D. Roweth. Cray XC (R) Series Network. Technical report, Cray Inc., 2012.

[7] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *ACM SIGMOD*, pages 963–968, 2010.

[8] A. Chan, P. Balaji, W. Gropp, and R. Thakur. Communication analysis of parallel 3d fft for flat cartesian meshes on large blue gene systems. In *Proceedings of the 15th International Conference on High Performance Computing*, HiPC'08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] B. Chandramouli and J. Goldstein. Patience is a virtue: Revisiting merge and sort on modern processors. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 731–742, New York, NY, USA, 2014. ACM.

[10] M. Clement and M. Quinn. Overlapping computations, communications and i/o in parallel sorting. *Journal of Parallel and Distributed Computing*, 28(2):162 – 172, 1995.

[11] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, Aug. 1988.

[12] B. Dong, S. Byna, and K. Wu. Expediting scientific data analysis with reorganization of data. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, Sept. 2013.

[13] A. Ebert. Ntosort. Technical report, April, 2013.

[14] C. Faloutsos, Y. Matias, and A. Silberschatz. Modeling skewed distribution using multifractals and the 80-20 law. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 307–317, 1996.

[15] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970.

[16] W. Gropp, R. Thakur, and E. Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface.* MIT Press, Cambridge, MA, USA, 2nd edition, 1999.

[17] M. Howison, E. W. Bethel, and H. Childs. Mpi-hybrid parallelism for volume rendering on large, multi-core systems. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, pages 1–10, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[18] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. Cloudramsort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster. In *SIGMOD '12*, pages 841–850, New York, NY, USA, 2012. ACM.

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

[20] M. E. J. Newman. Power laws, pareto distributions and zipfâĂŹs law. *Contemporary Physics*, 2005.

[21] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey. Bd-cats: Big data clustering at trillion particle scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 6:1–6:12, New York, NY, USA, 2015. ACM.

[22] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced and energy-efficient large-scale sorting system. *ACM Trans. Comput. Syst.*, 31(1):3:1–3:28, Feb. 2013.

[23] R. Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, Oct. 1978.

[24] E. Solomonik and L. Kale. Highly scalable parallel sorting. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[25] V. Springel. The cosmological simulation code GADGET-2. Technical report, 2005.

[26] M. H. Stanley, S. V. Buldyrev, S. Havlin, R. N. Mantegna, M. A. Salinger, and H. E. Stanley. Zipf plots and the size distribution of firms. *Economics Letters*, 49(4):453 – 457, 1995.

[27] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[28] H. Sundar, D. Malhotra, and G. Biros. Hyksort: A new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 293–302, New York, NY, USA, 2013. ACM.

[29] H. Sundar, D. Malhotra, and K. W. Schulz. Algorithms for high-throughput disk-to-disk sorting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 93:1–93:10, New York, NY, USA, 2013. ACM.

[30] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Supercomputing '92*, pages 14–19, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.