Algorithms for High-Throughput Disk-to-Disk Sorting

Hari Sundar
The University of Texas at Austin, Austin, TX 78712
hsundar@gmail.com

Dhairya Malhotra
The University of Texas at Austin, Austin, TX 78712
dhairya.malhotra@gmail.com

Karl W. Schulz
Texas Advanced Computing Center, Austin, TX 78712
karl@tacc.utexas.edu

ABSTRACT

In this paper, we present a new out-of-core sort algorithm, designed for problems that are too large to fit into the aggregate RAM available on modern supercomputers. We analyze the performance including the cost of IO and demonstrate the fastest (to the best of our knowledge) reported throughput using the canonical sortBenchmark on a general-purpose, production HPC resource running Lustre. By clever use of available storage and a formulation of asynchronous data transfer mechanisms, we are able to almost completely hide the computation (sorting) behind the IO latency. This latency hiding enables us to achieve comparable execution times, including the additional temporary IO required, between a large sort problem (5TB) run as a single, in-RAM sort and our out-of-core approach using 1/10th the amount of RAM. In our largest run, sorting 100TB of records using 1792 hosts, we achieved an end-to-end throughput of 1.24TB/min using our general-purpose sorter, improving on the current Daytona record holder by 65%.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and Searching*

Keywords

Sorting; Out-of-Core Algorithms; Parallel Algorithms; shared-memory parallelism; distributed-memory parallelism; hypercube; quicksort; samplesort; asynchronous methods.

1. INTRODUCTION

The design and implementation of parallel algorithms for comparison-based sort is a well studied subject. Here we are concerned with the practical issue of developing efficient sorting algorithms that can handle petabyte-sized datasets efficiently on existing and near-future architectures. Comparison sort is used in many data-intensive scientific computing and data analytics codes (e.g., [12, 20, 22]). There has been tremendous growth in the amount of data available in a variety of application domains such as meteorology, astrophysics, genomics, and biological and environmental research [1]. For example the Square Kilometer Array, once completed, will produce an estimated 1 exabyte of data a day [4]. Storing such large amounts of information in volatile-memory (RAM) for processing is not cost effective. Additionally, the rate at which new data is being generated clearly outpaces the rate at which total available RAM grows. Consequently, the imbalance between what we can fit in memory, and the amount of data we wish to process is only going to increase for the foreseeable future.

To address such large-scale data-processing needs, several software frameworks have been developed in the past few years, including MapReduce [5], the Google File System [6], and Hadoop [24]. The main attraction of these frameworks has been their ability to work with heterogeneous clusters, fault tolerance, and linear scalability with the number of nodes in the cluster. The key limitation has been the low per-node performance, arising due to the fact that improvements in disk-latency and bandwidth have not kept up with increase in disk capacity as well as computational capabilities of processors. Recent work has attempted to address this with two opposing approaches. The first approach [16] relies on specialized hardware and software architectures, that by maintaining a high disk-to-node ratio, as well as by allowing application-level control of in-memory buffers, is able to achieve extremely high read and write throughput. This strategy is highly effective, and they currently hold the GraySort sorting benchmark [8] with a very impressive rate of 0.938 TB/min[1]. The key problem with this approach is that such specialized systems do not scale well to other algorithms and applications; beyond sorting records in this case. The second approach [11] advocates operating from RAM instead of moving data back-and-forth to the disk. This is the so called RAMcloud approach [14]. The fact that data such as Google search indices, and non-image Facebook data is primarily stored in RAM [15], supports this argument. As mentioned earlier, due to the rate of increase of the amount of data, RAMclouds are unlikely to be sufficient for several big-data applications. As most real-world big-data applications are a combination of data-intensive and compute intensive algorithms, it is important that we develop algorithms that can sustain high-throughput on general-purpose archi-

[1]For the Indy benchmark. As a general purpose sorter (Daytona benchmark), their throughput is lower at 0.725TB/min

tectures and clusters that leverage commodity components. Indeed, the challenge of big-data must be met algorithmically, not circumvented by extra RAM or extra disks.

In recent work [21], we presented a new in-RAM sorting algorithm, HykSort, where we sustained a in-RAM sort throughput of 54TB/min on 262,144 cores of the CRAY XK7 "Titan" platform at the Oak Ridge National Laboratory. Such a high throughput was achieved by avoiding $\mathcal{O}(p)$-collective communication primitives, ensuring better load balancing and using a staged-communication pattern to avoid network congestion. In this work, we extend HykSort and develop an asynchronous data-delivery mechanism which, when combined, allow for efficient out-of-core sorting while maintaining high throughput on general-purpose, production HPC hardware. This is made possible by designing algorithms that interleave stages of the sort while continuing to stream input data from a global file system and utilize local node storage for temporary out-of-core data. By sorting datasets with a single read and write to the global filesystem, we are able to achieve performance comparable to in-RAM sorting routines. In addition to this sort application, we believe there are lessons to be learned for other problems involving data that is too large to fit into main memory. As a highlight, we sustained a disk-to-disk throughput of 1.24TB/min, while sorting 100TB using 1792 nodes of Stampede. Compared against the annual Daytona GraySort sorting benchmark, our performance is 65% better performance over the previous record holder and improves on the previous Indy record as well[2].

Contributions.

Our main contribution is the development of algorithms and techniques that allow us to efficiently sort extremely large records on disk, by overlapping the computation (sorting) with the reading in and writing back of the records to the global distributed file system. Our methods are scalable, load-balanced, and capable of handling extremely skewed data distributions. We conduct an experimental study where we compare our throughput with non-overlapped approaches (using the same in-RAM sort and IO techniques), optimized for maximum IO performance and for maximum sort throughput, and demonstrate that our approach outperforms both approaches.

Limitations.

Our implementation does not support heterogeneous architectures. Efficient sorting implementations exist and can be coupled to our code, for example [13, 17]. Although we have tried to optimize our local sort, its performance is subpar to the implementation in [11]. In addition, since we estimate the distribution based on the first chunk of data, our approach might not be efficient on certain nearly sorted sequences. We partially handle this case by reading the input files in a random order, but pathological cases exist where our approach can fail. We are currently working on detecting such situations to trigger a re-sampling of the dataset.

Organization of the paper.

In §2, we discuss the most relevant distributed sorting algorithms. In §3, we describe the system architecture and the distributed filesystem. In §4, we present the overall methodology. Finally, in §5, we present experimental validation of the proposed approach.

[2]These are the 2012 records held by TritonSort [16].

2. DISTRIBUTED SORTING

In our discussion of prior work, we focus our attention to distributed memory algorithms that have been experimentally shown to scale to large core counts and large datasets.

Given an array A with N keys and an order (comparison) relation, we will like to sort the elements of A in ascending order. In a distributed memory machine with p tasks, every task is assigned an N/p-sized block of A. Upon completion, task i will have the i_{th} block of the sorted array. The same definition extends to keys stored in files across multiple disks on a distributed file systems. Designing and implementing an efficient sorting algorithm that can scale to thousands of cores is difficult since it requires irregular data access, communication, and load-balance.

The most popular algorithm used in actual parallel sort implementations is SampleSort [3]: Given p tasks, we reshuffle the elements of A in p buckets so that keys in the i_{th} bucket are smaller or equal than keys in the $(i+1)_{th}$ bucket. Bucket i is assigned to task i and once we have reshuffled A, each task i can sort its keys locally. The challenge is to obtain good load-balancing so that every task i has roughly the same number of keys after reshuffling while minimizing communication costs.

SampleSort reshuffles A by estimating the boundaries for each bucket using $p-1$ keys, which we call the "*splitters*". This can be done for example, by sampling a subset of keys in A, sorting them (say, by gathering these to the task 0 and using a sequential sort) and selecting the splitters from that set. Once these $p-1$ splitters have been selected, a global all-to-all data exchange redistributes the keys of every task to their correct bucket. An additional local sort is invoked to finalize the output array. SampleSort is well understood. However, its performance is quite sensitive to the selection of splitters, which can result in load imbalance. Most important, the all-to-all key redistribution scales linearly with the number of tasks and can congest the network. As a result SampleSort may scale suboptimally, especially when the communication volume approaches the available hardware limits [21].

Parallel HistogramSort [10, 20] is another variant of SampleSort that estimates the splitters more effectively than the original method. The authors presented one of the largest distributed comparison sort runs (32K cores on BG/P) with 46% efficiency. The algorithm overlaps communication and computation in all stages. During the splitter estimation stage, the iterative estimation of the ranks of the splitters is combined with partial local sorting of the data, by using the splitter candidates as pivots for quicksort. Once the splitters are estimated, the communication of data is staged and overlapped with local merging. In [20] the best throughput was obtained on 16,384 cores of Jaguar XT4 at ORNL for 8M (64-bit) keys per core; the sort took 4.3 seconds achieving a in-RAM throughput of 14.4TB/min.

CloudRAMSort [11] is a recent paper demonstrating good scalability on 256 nodes with shared memory parallelism using pThreads and SIMD vectorization. The best results are for sorting 1TB of data (10byte key + 90byte record) in 4.6 secs achieving an in-RAM throughput of 12.6 TB/min. They use a variant of the Histogram Sort [10], where the samples are iteratively computed (in parallel) in order to ensure a minimum quality of load-balance across processes. They split the communication of the records into two parts by first communicating the keys followed by the values, during which the keys are merged.

TritonSort [16] is the current record holder at the Sort-Benchmark [8]. It uses a hybrid TCP/IP and `pthreads` implementation of histogram sort using efficient data structures (trie). Unlike the other works, the key contributions of this work are in improving the overall IO throughput. As with other approaches, they do not sort the local samples initially, relying instead on random samples. The main challenge for scalability is the use of a single coordinator node that computes the global histogram and assigns segments to other nodes. This is the reason that the largest reported run is for 52 nodes. This group manages to sort 100TB input data (100-byte tuple, 10-byte key, 10 trillion keys) on 52 nodes achieving a disk-to-disk throughput of 0.9TB/min.

We focused on experimentally verified distributed memory sort algorithms with an emphasis in high performance computing. For more details on the theory of distributed sort see [2, 7].

3. SYSTEM ARCHITECTURE

The hardware employed for the runtime experiments carried out herein is the **Stampede** system at the Texas Advanced Computing Center (TACC). Stampede entered production in January 2013 and is a high-performance Linux cluster consisting of 6400 computes nodes, each with dual, eight-core processors for a total of 102,400 available cpu-cores. The dual-cpus in each host are Intel Xeon E5 (Sandy Bridge) processors running at 2.7GHz with 2GB/core of memory and a 3 level cache. The nodes also feature the new Intel Xeon Phi coprocessors. Stampede has a 56GB/s FDR Mellanox InfiniBand network connected in a fat tree configuration which carries all high-speed traffic (including both MPI and parallel file-system data).

As a comparison, we also tested on **Titan**, a Cray XK7 supercomputer at Oak Ridge National Laboratory (ORNL). Titan has a total of 18,688 nodes consisting of a single 16-core AMD Opteron 6200 series processor, for a total of 299,008 cores. Each node has 32GB of memory (but does not contain any local storage). It is also equipped with a Gemini interconnect and 600 terabytes of memory across all nodes. The IO subsystem configuration on Titan runs the Lustre [18] file system software, but the architecture is different from Stampede in that the IO system is a shared resource across multiple platforms. ORNL currently employs a site-wide global file system termed *Spider* to support temporary, large-scale parallel IO. On Titan, there are three equally sized scratch file systems available to users (widow[1-3]) that are 1.4 PB in size; each of these file systems is driven by 336 individual Object Storage Targets (OSTs). The peak speed of the Spider file system is reported at 240 GB/sec [19] but the amount available to a single file system on Titan is considerably less.

Even for modest dataset sizes that can be sorted completely in RAM, a primary performance consideration is the speed at which the input data can be read in from disk and then subsequently be rewritten after the sort is complete. While other researchers have devised specialized, dedicated IO subystems to achieve record sort speeds, a motivating challenge in the current work is the desire to leverage commonly available file systems on high-end supercomputers.

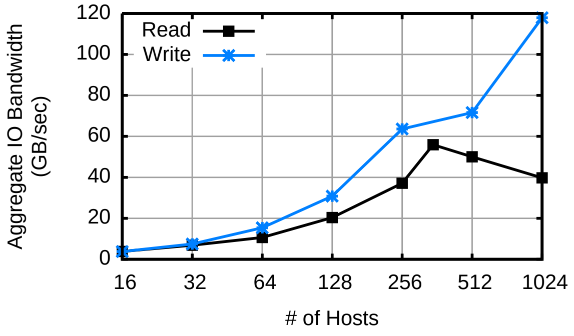

Figure 1: Aggregate read/write performance characteristics of a large, parallel Lustre file system. Results measured using one IO task per host on TACC's *Stampede* SCRATCH file system which contains 348 object-storage targets (rates measured using a fixed payload size per host: *read*: 40 GB/host, *write*: 2 GB/host).

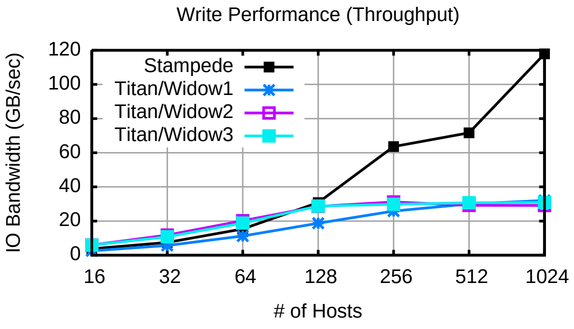

Figure 2: Comparison of aggregate write performance characteristics of *Stampede* (SCRATCH) and *Titan* (3 widow file systems). Results measured using one IO task per host with a fixed payload size of 2GB per host.

On Stampede, the file system components are built from 76 individual Dell DCS8200 commodity storage servers, each with 64 3TB drives configured across six individual raid devices. The raid sets are setup in an (8+2) RAID6 configuration using standard Linux software raid tools. To aggregate the storage hardware, the Lustre file system is also used on Stampede to provide three parallel file systems available system wide (HOME, WORK, and SCRATCH). The largest and most capable of these three file systems is SCRATCH, which is supported by 58 of the 76 servers. With 6 raid devices defined per server, this equates to a total of 348 OSTs defined within Lustre for SCRATCH, resulting in a total of 7.4 PB of user-addressable storage.

An interesting file-system characteristic that is common to Lustre, and is particularly relevant for the IO portion of the sort workload considered, is that aggregate *read* performance is typically maximized using a smaller number of participating clients than *write* performance. To illustrate this point, Figure 1 presents aggregate read/write measurements on Stampede's SCRATCH filesystem as a function of the number of hosts issuing the IO requests. In these measurements, only 1 MPI task/host is doing IO to individual files (using a Lustre file stripe count equal to 1 for all files). The results in Figure 1 presents weak-scaling results using a fixed payload size per host. To minimize any local caching effects for the read measurements, we purposefully choose a read size that is larger than the amount of memory available locally per host (32GB of RAM per compute host on Stampede). Based on this configuration, we chose a *read* payload of 40 GB and a *write* payload of 2 GB for the aggregate performance measurements.

Note that the measured write performance observed is generally higher than the read (also typical with Lustre) and that the write performance continues to improve, even up to 1K hosts writing simultaneously. In fact, this trend continues with modest improvements measured up to 4K participating write hosts yielding peak write speeds of over 150 GB/s on Stampede. However, the read performance peaks at a much smaller number of hosts, and not coincidentally, occurs when the number of read hosts is approximately equal to the maximum number of OSTs, (348 in the case of SCRATCH). In contrast, the throughput measured on Titan to the Spider file system was significantly lower with example write speed measurements presented in Figure 2. Recall that there are three high-speed file systems available for use on Titan, and peak write rates plateaued to ∼30 GB/sec for 128 hosts and beyond. Due to these lower observed rates, we focused our large-scale algorithmic testing on Stampede, but do include results at smaller scale on Titan to demonstrate the method's applicability. Note, howver, that the upcoming *Spider II* filesystem which is to be deployed on Titan should improve performance significantly.

In addition to the Lustre file system which houses the input datasets for our sort procedure, we also utilize other system resources when available to stage binned, temporary data to accommodate out of core sorts via our algorithm. On Stampede, each of the 6400 compute nodes has a single, 250GB commodity SATA hard drive that is used for OS provisioning, local HPC package installation support, and temporary per-job use by user applications via `/tmp`. The amount of `tmp` space available for application use is 69 GB/node; the IO rate to these commodity drives is measured to be 75 MB/sec for large block IO patterns. On Titan, due to the lack of local hard drives, we used one of the three available *widow* filesystems as temporary storage.

3.1 Experimental Environment

The algorithms and software developed to support the end-to-end disk sorting procedure were all written in `C++`. On-node parallelism was achieved via threading and off-node data transfers were accommodated with MPI. The specific library requirements and runtime software revisions used on Stampede to carry out the experiments are highlighted below:

- MPI → MVAPICH2 (v. 1.9b)
- C++/OpenMP → Intel XE Compiler (v. 13.1.0.146)
- Interprocess communications → Boost (v. 1.51.0)
- File system → Lustre (v 2.1.3)
- Interconnect fabric → OFED (v. 1.5.4.1)

The software revisions used for the companion Titan experiments are similarly highlighted below:

- MPI → Cray MPICH2 (v. 5.6.3)
- C++/OpenMP → GNU Compiler (v. 4.7.2)
- Interprocess communications → Boost (v. 1.53.0)
- File system → Lustre (v 1.8.6)
- Interconnect fabric → Cray Gemini

Note that all of the data presented herein was measured on the system running in normal, production operation via batch job submission. Consequently, since the global file systems are a shared resource, available to all users for general purpose I/O in their applications, the I/O bandwidth delivered to our sort procedure is not guaranteed to be constant. Indeed, we expect dynamic behavior as other users also utilize the resource simultaneously and our developed method does not assume a fixed IO rate.

3.2 Input Data

The input data used to test our methodology was generated using the `gensort` program written in C, which creates random records for use with the sortBenchmark [8]. Each record is a `100-byte` structure consisting of a `10-byte` key and a `90-byte` payload. To leverage the maximum amount of read bandwidth afforded by the Lustre file system, we used `gensort` to create N_f individual files that were each 100MB in size so that multiple read clients can be accessing differing storage targets simultaneously. As discussed in the previous section, the maximum read bandwidth corresponds to the number of Lustre OSTs configured in SCRATCH and it is clearly advantageous to have the N_f files distributed equally over all 348 OSTs. To facilitate this, we made one small addition to `gensort` to allow the resulting input files to be created on pre-specified targets. This relies on Lustre API extensions for POSIX which provide a mechanism to set desired file OST locations during the file creation process via `ioctl()` calls. In particular, an `oflag` argument to `open()` of `O_LOV_DELAY_CREATE` is used, combined with an `ioctl()` request for `LL_IOC_LOV_SETSTRIPE` to set the desired file stripe index.

4. ALGORITHMS & DESIGN

In this section, we describe the overall algorithms. We introduce the notation used in the rest of the section in Table 1. The overall algorithm can be broken down into two main stages; the **read** and **write** stages as shown in Figure 3. Our goal during the process is to overlap these stages with computation (sorting the data), such that these stages are completely dominated by read/write bandwidth. We also aim to maximize the disk-throughput. We divide the active processes into two distinct work groups, the *read_group* and the *sort_group*. The *read_group* is a set of processes dedicated to reading input data from the global parallel file system and delivering this data in a streaming fashion to the processes belonging to *sort_group*. As we are interested in sorting datasets that are larger than the collective main memories of both groups, we divide the original problem, of N records, into q chunks, each of M records. During the **read** stage, the *read_group* processes sequentially read the data from the global filesystem and transfer this data to the *sort_group* processes (§4.2). The *sort_group* processes, on receiving M records, bins these into q buckets (§4.3). These q buckets are stored on the local filesystem on each node. Once all the q chunks have been read in, we will have q files on each local disk. During the **write** stage, the flow of information is mostly reversed, with the *sort_group* processes

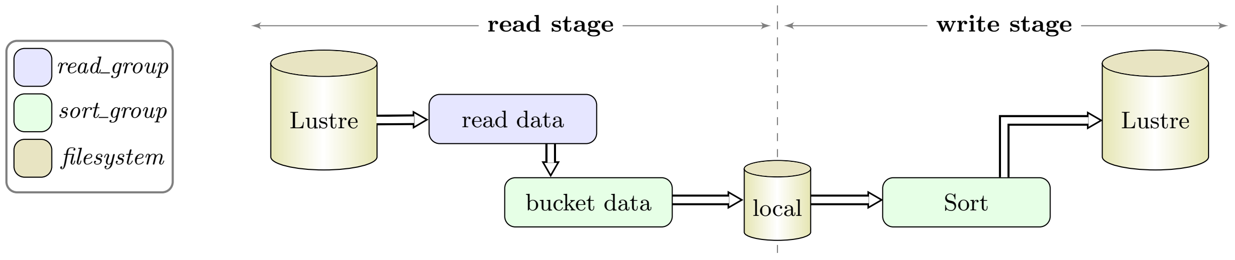

Figure 3: Overview of the end-to-end sort process. The arrows depict the flow of information. The use of the local disks as a large buffer, allows us to effectively 'stream' the records in and 'stream' the sorted records out. Only a single read and a single write are performed per record to/from the global parallel file system (Lustre).

A	input array (distributed)
B	output array (ascending order, local block)
$comm$	MPI communicator
p	number of MPI tasks in $comm$
N	global number of keys in A
M	number of records that can fit in RAM
$q = N/M$	number of in-RAM sorts needed
$n = N/p$	local number of keys in A
k	number of splitters ($k \neq p$)
s	the splitters
p_r	the task id (its MPI Rank)
A_r	array to be sorted (local block)
Functions	
`Length`(B)	number elements in array B
`Rank`(s, B)	`Length`$(\{B_i : B_i < s\})$
`Sort`	local, non-MPI, sort
log	base-2 logarithm

Table 1: ***Here we summarize the notation.*** `Sort` ***can be any sequential sort (in our experiments, when we have one thread for each MPI task, we use*** `std::sort`***). Note the*** `Rank`(s, B)***, is equal to the number of keys in*** B ***that are strictly smaller than*** s***.***

reading the q local files, one at a time, synchronized across all processes, sorting them globally(§4.4), and then writing the final sorted data back to the global filesystem. Note that the size of *sort_group* is generally larger than *read_group* and consequently, we perform the final writes using *sort_group* directly due to the scaling properties of Lustre identified previously in Figure 1. In the rest of this section, we first describe the set of MPI communicators needed to overlap these steps, which is essential to obtain high throughput.

4.1 Communicators

To support a high-degree of asynchronous operations and implement the *read_group* and *sort_group* process groups, we first divide all available processes into two MPI communicators, `READ_COMM` and `SORT_COMM`, respectively. In addition to these two distinct communicators, a third high-level communicator, `XFER_COMM`, is defined which serves as the vehicle to provide a data transfer path between the N_{read} hosts responsible for reading data to the N_{sort} hosts in charge of bucketing/storing temporary data and performing a final sort over N_{bins}. Because the workloads on these two groups are quite different, special care is taken to define how many processing cores per host are placed in these three groups. To help illustrate the communicators and processor layout, Figure 4 presents an overview of the streaming read portion of the overall sort process. On the right hand side of the Figure, we see N_{sort} hosts, each assumed to have at least two processing cores each. One core per sort host (shown in grey) is reserved for receiving data and is placed in `XFER_COMM`. The remaining cores (shown graphically in orange) available on each sort host are placed in `SORT_COMM`. The `READ_COMM` is simply the collection of N_{read} hosts utilizing one processing core per host. The `XFER_COMM` is shown on the figure highlighting the groups of processes which perform the underlying asynchronous MPI data transfers.

In addition to the three primary communicators defined so far, we also create a family of binning communicators as a subset of `SORT_COMM`. These N_{bin} communicators, defined as `BIN_COMM`$_i$ are built using one process from each sort host. These `BIN_COMM`$_i$ groups are used to overlap the process of bucketing local data and saving to to temporary storage with the receipt of new input data. An example of binning communicators is illustrated in Figure 5. Note that the maximum number of possible binning communicators is bound by the total number of cores per host minus one (since one core is reserved for `XFER_COMM`).

4.2 Streaming Operations

Paramount to maintaining sustained performance throughout the process is the need to mitigate stalling any portion of the raw data reads and to allow for out-of-order data transfer since the read performance will, in general, vary on a per storage-target basis on a large, shared file system. To keep the data transfer pipeline moving as quickly as possible there are two primary spin loops depicted in Figure 4. The first loop is engineered to keep N_{read} tasks busy performing raw reads from input files residing on the global file system. Recall from the previous section that only one processing core per sort host is defined in `READ_COMM`. However, this process is threaded using OpenMP work-sharing constructs on the reader hosts such that one thread per host is dedicated solely to reading new input files and storing the streaming data in a `fifo` queue. The companion transfer tasks on the same IO host in `READ_COMM` are in a constant spin loop checking for new data to become available. The processes across all `READ_COMM` hosts work in tandem checking for new data and when detected, each host pops a record from the queue and issues asynchronous `Isends()` to receiving ranks in `XFER_COMM`. Note that since the data exchange between threads is shared-memory based, the `fifo` queue updates (both `push` and `pop`) are mutexed via OpenMP `critical` sections and this is the only potential synchronization be-

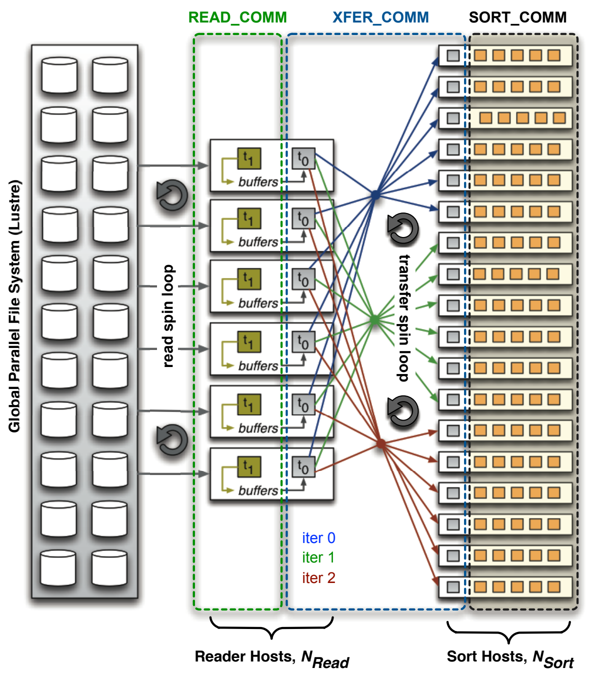

Figure 4: Overview of the read stage identifying principal asynchronous work division between IO and sort hosts. The reader hosts use OpenMP threading to cache data streamed from disk into a `fifo` queue for subsequent transfer to sort hosts using the `XFER_COMM` MPI communicator. As data is received by a dedicated core on each sort host, it is subsequently transferred to tasks in `SORT_COMM` via mapped shared memory segments.

tween read-tasks and transfer-tasks on the IO hosts.

To ensure that matching receives can be posted in advance, the receiving tasks in `XFER_COMM` simply cycle over available ranks, each taking a turn posting a blocking `Recv()` against any IO host. The data transfer mechanism between a receiving task in `XFER_COMM` and a sort task in `SORT_COMM` cannot be done using the same OpenMP shared-memory constructs employed on the IO hosts as they are defined to be in unique MPI threads. Instead, the approach adopted here is to perform the interprocess communication using mapped shared memory segments via the `boost` library. The receiving tasks in `XFER_COMM` write into this shared segment directly as data arrives. In tandem, matching spin loops in the currently active `BIN_COMM` check for the availability of new data and copy into local buffers so that the next round of new data can be processed. The whole process is repeated until the desired amount of input data has been delivered to the sort hosts and processed by one or more `BIN_COMM` groups and written to temporary local storage. After all the data has been processed, the bucketed data is reread from temporary storage, sorted, and written back to the global file system. Note that this final step is also performed cyclicly across one or more `BIN_COMM` groups in order to overlap the read/sort/write process as much as possible across multiple sort bucket boundaries. This process is illustrated in Figure 5.

4.3 Bucket Records

As we get records from the *read_group* we wish to bin them into q buckets and write them to the local disk. For this we need $q-1$ "splitters" that equally partition the N input records into q equal chunks. We use a parallel selection algorithm that is also part of out sorting routine. The splitters for the local disk buckets are determined using samples from the first M records for matters of efficiency. For sufficiently large datasets, with well shuffled data, this should not cause any performance problems. As we load-balance across all processes before writing to local disk, all processes should have the same file size on local disk. Errors in splitter selection, will result in different sized buckets, but each bucket is still guaranteed to be load-balanced across all processes. We now describe our parallel selection algorithm followed by the procedure for binning the incoming data based on these splitters and overlapped write to local disks.

4.3.1 *Selecting Splitters using Parallel Select*

HyperQuickSort, HistogramSort, SampleSort and other splitter based sorting algorithms all rely on efficient determination of accurate splitters [21]. For HyperQuickSort, in each stage, a single splitter is determined, usually by choosing the median on any one task. However, this is not a reliable method and an error of ϵN in the rank of the splitter can lead to the final load-imbalance on a task to be as large as $\mathcal{O}((1+\epsilon)^{\log p}n)$. In SampleSort, a set of $p-1$ splitters is generated by taking p evenly spaced samples in each locally sorted array, sorting the p^2 samples (using bitonic sort) and taking the last element on each processor as a splitter. The maximum load on any processor is guaranteed to be less than $2n$.

Algorithm 4.1 ParallelSelect

```
Input: $A_r$ (locally sorted), $n$, $N$, $R[0,...,k-1]$ (expected global ranks), $N_\epsilon$ global rank tolerance, $\beta \in [20,40]$.
Output: global splitters $S \subset A$ with approximate global ranks $R[0,...,k-1]$
 1: $R^{start} \leftarrow [0,...,0]$   ▷ Start of range for sampling splitters
 2: $R^{end} \leftarrow [n,...,n]$   ▷ End of range for sampling splitters
 3: $n_s \leftarrow [\beta/p,...,\beta/p]$   ▷ #of local samples (each splitter)
 4: $N_{err} \leftarrow N_\epsilon + 1$
 5: while $N_{err} > N_\epsilon$ do
 6:     $Q' \leftarrow A_r\left[\mathtt{rand}\left(n_s, \left(R^{start}, R^{end}\right)\right)\right]$
 7:     $Q \leftarrow \mathtt{Sort}(\mathtt{All\_Gather}(\hat{Q}'))$   ▷ $\mathcal{O}(k\log p + k\log k)$
 8:     $R^{loc} \leftarrow \mathtt{Rank}(Q, A_r)$   ▷ $\mathcal{O}(k\log n)$
 9:     $R^{glb} \leftarrow \mathtt{All\_Reduce}\left(R^{loc}\right)$   ▷ $\mathcal{O}(k\log p)$
10:     $I[i] \leftarrow \mathrm{argmin}_j |R^{glb}[j] - R[i]| \quad \forall i$
11:     $N_{err} \leftarrow \max\left|R^{glb} - R[I]\right|$
12:     $R^{start} \leftarrow R^{loc}[I-1]$
13:     $R^{end} \leftarrow R^{loc}[I+1]$
14:     $n_s \leftarrow \beta\frac{R^{end}-R^{start}}{R^{glb}[I+1]-R^{glb}[I-1]}$
15: end while
16: return $S \leftarrow Q[I]$
```

In Algorithm 4.1, we use a technique also used in HistogramSort. However, instead of computing $p-1$ splitters, which becomes costly for large p, we only compute $k < p$ splitters: *a*) Select βk samples from the entire data (across all tasks) and collect these on all tasks. *b*) Sort samples and determine ranks locally using binary search. *c*) Perform an

MPI_AllReduce to determine the global ranks of the samples. *d*) Select pairs of samples with ranks just greater and just smaller than the expected global rank for each of the k splitters. *e*) Repeat steps $a - d$ by taking β samples in the narrowed range for each splitter and iterating till the range is within a tolerance value. The number of samples β must be such that the number of iterations needed is not very high and also the cost of each iteration is small. In our experiments, $\beta \in [20, 40]$ worked well.

4.3.2 Handling Skewed Data

Although the parallel select algorithm (Algorithm 4.1) can handle certain non-uniform distributions, it fails for extremely skewed distributions such as the Zipf distribution [9]. As the Zipf distribution is known to model several big-data problems, it is important that we be able to handle such distributions. The parallel select algorithm used in [21] fails when there are sufficiently large duplicate keys, especially when the number of duplicate keys is $\mathcal{O}(n)$. To remedy this solution, we rank our splitters first based on *key* and resolve equal keys based on their rank in the original array. This only needs to be done for the ranking of the splitters and increases the communication during the splitter selection by a long int[3]. This does not affect the communication of the actual records. It is important to note that unlike [11], the number of samples does not have to double every iteration to achieve load-balance.

4.3.3 Binning and Local Writes

The relatively low global read allows us to perform overlapping computations to ensure load-balance, for the subsequent write stage. Every node in the *sort_group* locally sorts its records using a shared memory parallel implementation of mergesort [21]. We use a binary search within the records using the splitters as the keys (since, in general, $n >> q$), thereby dividing it into the desired buckets. The records in each of these buckets are equally distributed across all processes in the *sort_group* to ensure load balance. The load-balanced records in each bucket are written to separate files on the local filesystem. Since there are multiple tasks sharing a single local disks (tasks on the same host), we overlap the writing of bucket i with the re-distribution of records from other buckets. The overlapped read/sort/write processes are performed cyclically across multiple BIN_COMM groups. This process is illustrated in Figure 5. In §5.1, we present experimental evidence for the efficacy of this approach in improving the read efficiency.

4.4 HykSort - Scalable RAM Sort

Our recent in-RAM sort, HykSort [21], is used for distributed sorting. HykSort improves upon the original Hypercube QuickSort [23], by using Algorithm 4.1 to select splitters and by generalizing the 2-way splitting to a k-way splitting. We presort the local array and following data-exchanges, we locally merge the k segments to maintain locally sorted arrays. Following [20], we overlap communication to receive the new keys with the merging of the keys with the local array. The overall algorithm is described in Algorithm 4.2. For complexity estimates and additional details of the in-RAM sort and the splitter selection, please refer to [21]

[3]In general, an integer datatype large enough to store N.

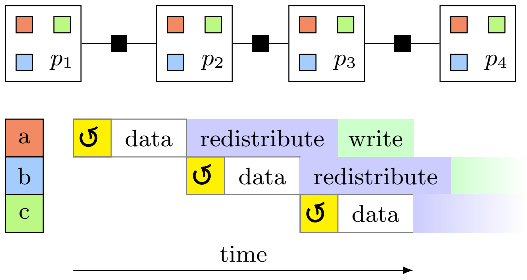

Figure 5: Illustration of overlap process for performing binning and local writes within the *sort_group* using multiple BIN_COMM communicators. An example timeline is shown for three sub-communicators; the initial active communicator (a) spins until sufficient data is available; it then activates the next communicator (b) which starts the spin process waiting for more data to become available. While (b) is gathering new data, (a) performs a local binning redistribution and writes the data to temporary storage. This process is repeated cycling from $(a) \to (b) \to (c) \to (a)$ until all input data has been received and been placed into q bins.

5. RESULTS

In this section, we present results evaluating the performance of our sorting algorithm measured on Stampede and Titan. We first present results demonstrating the overlap efficiency obtained via the use of multiple BIN_COMM communicators on each host belonging to the *sort_group*. Note that earlier designs of our data delivery process relied on only the single communicator SORT_COMM. However, based on early testing we determined a need for additional asynchronous processing in order to sufficiently hide the latency for the binning process. The overlap efficiency results are followed by overall throughput results and its comparison to the current records for the Indy and Daytona sort benchmark. We also provide results comparing the performance of our code on Stampede and Titan. This is followed by results demonstrating the ability of the algorithm to handle skewed distributions. Finally, we present a direct comparison using our best in-RAM sorting algorithm (HykSort) including IO compared against the proposed out-of-core sorting implementation.

Because of the low IO throughput on "Titan" (Figure 2), most experiments were carried out on "Stampede". Extensive scalability experiments have been performed on "Titan" for the in-RAM sort in previous work [21]. We compare the throughput on both systems for problem sizes up to 10TB. On "Stampede" this was scaled up to 100TB.

5.1 Overlap Efficiency for Read

As explained in §4.2, the *read_group* hosts read in the records and transfer the data to the *sort_group* hosts. The *sort_group* hosts, on receiving a sufficient number of records (M), bucket these records and write them out to local storage. We use multiple MPI communicators during this stage (§4.3.3) to ensure that these steps are completely hidden behind the global read latency. We define the overlap efficiency as the ratio of the overall time taken for reading in the records, without any overlapping work, i.e., without

Algorithm 4.2 HykSort

```
Input: $A_r$, $comm$, $p$ (w.l.g., assume $p = mk$), $p_r$ of current
    task in $comm$,
Output: globally sorted array $B$
 1: $B \leftarrow$ LocalSort($A$)
 2: while $p > 1$ do                          ▷ Iters: $\mathcal{O}(\log p / \log k)$
 3:     $N \leftarrow$ MPI_AllReduce($|B|, comm$)
 4:     $s \leftarrow$ ParallelSelect ($B, \{i\,N/k \mid i = 1, ..., k-1\}$)
 5:     $d_{i+1} \leftarrow$ Rank($s_i, B$),   $\forall i$
 6:     $[d_0,\ d_k] \leftarrow [0,\ n]$
 7:     $color \leftarrow \lfloor k\,p_r/p \rfloor$
 8:     parallel for $i \in 0, ..., k-1$ do
 9:         $p_{recv} \leftarrow m\,((color - i) \bmod k) + (p_r \bmod m)$
10:         $R_i \leftarrow$ MPI_Irecv($p_{recv}, comm$)
11:     end for
12:     for $i \in 0, ..., k-1$ do
13:         $p_{recv} \leftarrow m\,((color - i) \bmod k) + (p_r \bmod m)$
14:         $p_{send} \leftarrow m\,((color + i) \bmod k) + (p_r \bmod m)$
15:         MPI_Issend($B[d_i, ..., d_{i+1} - 1], p_{send}, comm$)
16:         $j \leftarrow 2$
17:         while $i > 0$ and $i \bmod j = 0$ do
18:             $R_{i-j} \leftarrow$ merge($R_{i-j}, R_{i-j/2}$)
19:             $j \leftarrow 2j$
20:         end while
21:         MPI_WaitRecv($p_{recv}$)
22:     end for
23:     MPI_WaitAll()
24:     $B \leftarrow$ merge($R_0, R_{k/2}$)
25:     $comm \leftarrow$ MPI_Comm_split($color, comm$)
26:     $p_r \leftarrow$ MPI_Comm_rank($comm$)
27: end while
28: return $B$
```

binning the records and writing to local storage against the time taken for reading in the records with overlapping work. Since the efficiency of the overlap depends on the number of binning groups, we computed the overlap efficiency using two configurations,

- 64 *read_group* hosts with 256 *sort_group* hosts, and
- 128 *read_group* hosts with 512 *sort_group* hosts,

while varying the number of binning groups from 1 to 12. These results are plotted in Figure 6. As can be seen, for the smaller configuration, we get 100% efficiency for $N_{bins} \geq 2$, while for the larger configuration, we obtained 95% efficiency for $N_{bins} \geq 4$. In general, we saw more variability in the overlap efficiency with $N_{bin} = 2$ and this is owed to the sensitivity of the underlying file system performance when N_{bin} is small. Note also that the results in Figure 6 illustrate the need to have more than a single communicator coordinate the binning and local storage as the efficiencies were both under 70%. Based on these results, we selected a value of $N_{bins} = 8$ for all other experiments to ensure good overlap efficiency.

5.2 Sort Throughput

We tested the overall sort throughput of "Stampede" using 348 *read_group* hosts (chosen to match the peak read rate configuration for Stampede's SCRATCH file system) and 1444 *sort_group* hosts for problem sizes up to 100TB. The throughput achieved is plotted in Figure 7. The throughput

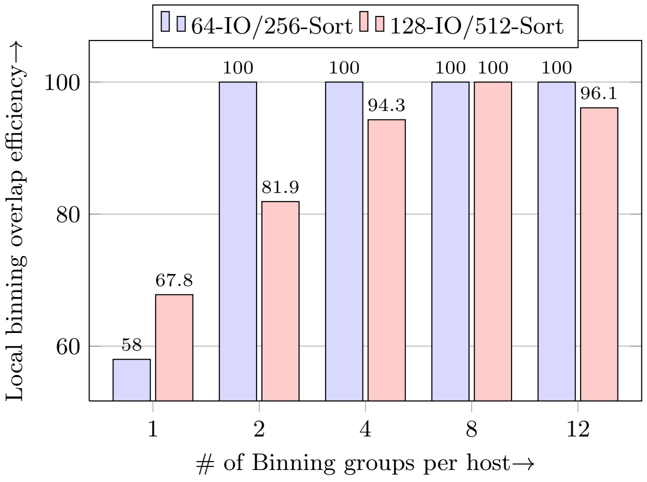

Figure 6: Overlap efficiency achieved via the use of N_{bin} binning groups (MPI communicators) per sort host. The ratio between sort hosts and IO hosts is $4X$ with every IO host reading 40GB of data. Values of $N_{bin} >= 4$ are shown to provide excellent overlap efficiencies indicating the latency of the local binning and temporary storage output procedure is sufficiently hidden via the method.

achieved by the current sortBenchmark record holders in both the Indy and Daytona GraySort categories is also plotted. On "Titan", we used 168 *read_group* hosts and a maximum of 344 *sort_group* hosts for problem sizes up to 100TB. The throughput achieved is plotted in Figure 8. These results are on a system with other users accessing the global filesystems during an extremely busy period. However, even without dedicated resources the throughput achieved is impressive when compared with the current Indy and Daytona benchmark records holders. These are even more impressive given the fact that our sort routine is generic and agnostic to the `100-byte` records used for the benchmark—hence is a Daytona run.

5.3 Skewed distributions

We performed runs with uniform and skewed datasets using the same setups for problem sizes up to 10TB on "Stampede". The throughput dropped from 17GB/s for the uniform to 12GB/sec for the skewed datasets. The drop in performance was due to increased load-imbalance. We are currently working on improving the performance for heavily skewed datasets.

5.4 Comparison with in-RAM sort

As a final example of the efficacy of our out-of-core sorting algorithm, we compared it against itself while operating as an in-RAM sort. In both cases, we sorted 5TB of records on disk, using 1408 hosts for the in-RAM version, and 1372 hosts for the out-of-core version (348 for IO and 1024 for sorting). The mismatch in the number of cores is to be able to choose the best parameter (k) for HykSort. Additional details on this can be found in [21]. For the in-RAM version, all records are read into memory, and sorted using a single call to HykSort. The sorted array is then written back to the global filesystem. For the out-of-core version, we con-

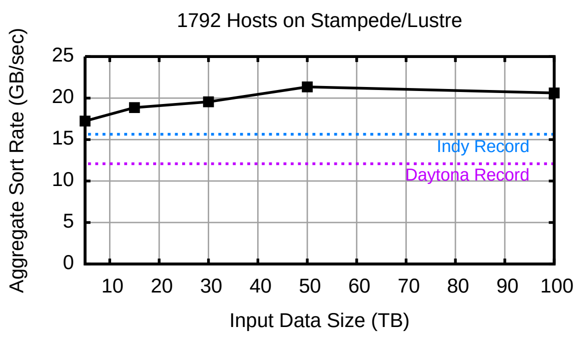

Figure 7: Sustained *aggregate* sort rates via the developed method as measured on Stampede's SCRATCH file system using 348 IO hosts and 1024 sort hosts.

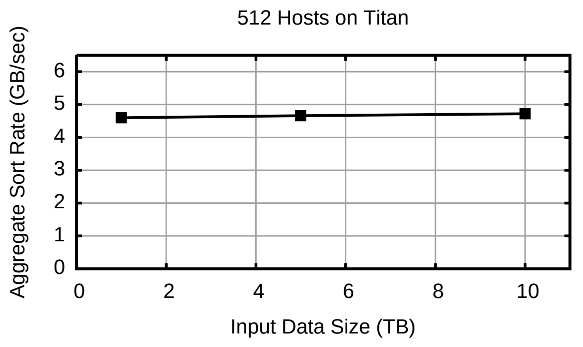

Figure 8: Sustained *aggregate* sort rates via the developed method as measured on Titan's widow1 file system using 168 IO hosts and 344 sort hosts.

sidered a scenario where only 1/10 RAM is available and sorted the arrays using a value of $q = 10$. The total disk-to-disk sorting time for the in-RAM case was 253.41s and for the out-of-core case was slightly higher at 272.6s. This was with fewer hosts and an even fewer number of hosts participating in the actual sort again indicating significant overlap efficiencies and the merits of using a smaller number of read clients to maximize the raw read-bandwidth on a Lustre file system.

6. CONCLUSIONS

We have presented a new, asynchronous out-of-core sorting algorithm designed for problems that are too large to fit into the aggregate RAM available on modern supercomputers. Our sort algorithm is datatype agnostic and can be used with any datatype for which an ordering and equality can be defined. We also presented a modification to our in-RAM sort algorithm, HykSort, making it stable, and more importantly, able to handle highly skewed distributions, such as Zipf. We evaluated our out-of-core sorting mechanism using `100-byte` records used for the sorting benchmark, and compared against the current record holders. Our results compare favorably with the current record holders, both Indy and Daytona metrics. We obtained a 65% improvement in throughput compared with the current Daytona record, in spite of being executed on a general-purpose, production HPC resource with IO resource contention amongst all system users. Our overlapped out-of-core algorithm also compared favorably with its in-RAM variant, sorting `5TB` in comparable total runtime, while using 1/10th the amount of RAM. In our largest run, sorting 100TB of records using 1792 hosts on Stampede, we achieved an end-to-end throughput of 1.24TB/min using our general-purpose sorter (Daytona). As the developed out-of-core method tests and stresses nearly all components of modern supercomputing architectures (global IO, local IO, interconnect, local compute performance, etc) we also plan to package the entire process (*data delivery* plus *sort*) for use as a standalone, system-level benchmark. The code for the sort is available at `http://padas.ices.utexas.edu/research/sorting`.

Moving forward, the main improvements in our implementation will be to use the *read_group* hosts during the write stage, as they are currently idle during this stage.

Acknowledgment

This research was partially supported by AFOSR grant FA9550-12-10484; DOE grants DE-FG02-08ER2585 and DE-SC0009286; and NSF grants OCI-1047980, OCI-1029022, OCI-0923710, and OCI-1134872. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources on Stampede that have contributed to the research results reported within this paper. Computing time on ORNL's Titan system was provided by an allocation from the Oak Ridge Leadership Computing Facility. We would also like to thank Dr. Judith Hill of ORNL for her assistance with porting our codes to Titan.

7. REFERENCES

[1] Community cleverness required. *Nature*, 455(7209):1–1, Sept. 2008.

[2] P. Berthomé, A. Ferreira, B. Maggs, S. Perennes, and C. Plaxton. Sorting-based selection algorithms for hypercubic networks. *Algorithmica*, 26(2):237–254, 2000.

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

[4] P. C. Broekema, R. V. van Nieuwpoort, and H. E. Bal. Exascale high performance computing in the square kilometer array. In *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*, Astro-HPC '12, pages 9–16, New York, NY, USA, 2012. ACM.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[7] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *An Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison Wesley, second edition, 2003.

[8] J. Gray. Sort benchmark home page. http://sortbenchmark.org/, Apr. 2013.

[9] R. Günther, L. Levitin, B. Schapiro, and P. Wagner. Zipf 's law and the effect of ranking on probability distributions. *International Journal of Theoretical Physics*, 35:395–417, Feb. 1996.

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

[11] 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 *Proceedings of the 2012 international conference on Management of Data*, pages 841–850. ACM, 2012.

[12] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Communications of the ACM*, 55(5):101–109, May 2012.

[13] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[14] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al. The case for ramcloud. *Communications of the ACM*, 54(7):121–130, 2011.

[15] P. Ranganathan. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, pages 39–48, 2011.

[16] A. Rasmussen, G. Porter, M. Conley, H. Madhyastha, R. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *USENIX NSDI*, volume 11, 2011.

[17] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast sort on CPUs, GPUs and intel MIC architectures. Technical report, Technical report, Intel, 2010.

[18] P. Schwan. Lustre: Building a file system for 1,000-node clusters. In *PROCEEDINGS OF THE LINUX SYMPOSIUM*, page 9, 2003.

[19] G. Shipman, D. Dillow, S. Oral, F. Wang, D. Fuller, J. Hill, and Z. Zhang. Lessons learned in deploying the world's largest scale lustre file system. In *Proceedings of the 52nd Cray User Group Conference (CUG)*, May 2010.

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

[21] H. Sundar, D. Malhotra, and G. Biros. Hyksort: A new variant of hypercube quicksort on distributed memory machines. In *Proceedings of the 27th ACM international conference on Supercomputing*, ICS '13, New York, NY, USA, 2013. ACM.

[22] H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

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

[24] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.