CloudRAMSort: Fast and Efficient Large-Scale Distributed RAM Sort on Shared-Nothing Cluster

Changkyu Kim†, Jongsoo Park†, Nadathur Satish†, Hongrae Lee⋆,
Pradeep Dubey† and Jatin Chhugani†

†Parallel Computing Lab, Intel Labs

⋆Structured Data Group, Google Research

ABSTRACT

Sorting is a fundamental kernel used in many database operations. The total memory available across cloud computers is now sufficient to store even hundreds of terabytes of data in-memory. Applications requiring high-speed data analysis typically use in-memory sorting. The two most important factors in designing a high-speed in-memory sorting system are the *single-node sorting performance* and *inter-node communication*.

In this paper, we present **CloudRAMSort**, a fast and efficient system for large-scale distributed sorting on shared-nothing clusters. CloudRAMSort performs multi-node optimizations by carefully overlapping computation with inter-node communication. The system uses a dynamic multi-stage random sampling approach for improved load-balancing between nodes. CloudRAMSort maximizes per-node efficiency by exploiting modern architectural features such as multiple cores and SIMD (Single-Instruction Multiple Data) units. This holistic combination results in the highest performing sorting performance on distributed shared-nothing platforms. CloudRAMSort sorts **1 Terabyte (TB) of data in 4.6 seconds** on a 256-node Xeon X5680 cluster called the Intel Endeavor system. CloudRAMSort also performs well on heavily skewed input distributions, sorting 1 TB of data generated using Zipf distribution in less than 5 seconds. We also provide a **detailed analytical model** that accurately projects (within avg. 7%) the performance of CloudRAMSort with varying tuple sizes and interconnect bandwidths. Our analytical model serves as a useful tool to *analyze performance bottlenecks* on current systems and *project performance* with future architectural advances.

With architectural trends of increasing number of cores, bandwidth, SIMD width, cache-sizes, and interconnect bandwidth, we believe CloudRAMSort would be the *system of choice* for distributed sorting of large-scale in-memory data of current and future systems.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Distributed databases, Parallel databases*

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

Keywords

Sorting, Distributed, Cloud Computing, RAM Sort, Multi-Core, Many-Core, SIMD

1. INTRODUCTION

With the rise of data-center computing, there has been a rapid increase in data-centric workloads that attempt to perform complex analysis of massive amounts of data. Disk-oriented approaches to store this data are unable to scale to meet the performance needs of large complex data analysis [46]. This is because disk latency and bandwidth improvements have lagged behind disk capacity, as well as the increase in computational resources of modern processors [16]. Consequently, the trend for performance-sensitive data-centric workload processing is operating from DRAM rather than disk. This includes large-scale industrial applications such as those used in Google and Yahoo!, where search has evolved to store indices entirely in DRAM [49], and Facebook, where over 75% of all non-image data is stored in DRAM [46]. In academia, RAMCloud has been proposed as a new approach to data-center storage, where disk is only used as a backup medium [46].

In order to process their massive data in a reasonable amount of time, companies such as Google use a Data-Intensive Super-Computing (DISC) model [15], where a cluster of connected commodity processors take the place of traditional supercomputers in the HPC domain. However, a typical DISC software architecture based on Hadoop [1], Dryad [31], or MapReduce [22] only focuses on node scalability, and not per-node efficiency [50]. In fact, it has been found that, on a typical data-intensive task such as sorting, only 5-10% of per-node efficiency (either computation or inter-node communication) of non-DISC systems has been achieved [11]. In the presence of such per-node inefficiency, a much larger number of nodes are required to perform a given task, with associated wasteful energy costs and increasing failure probability [48, 56]. In order to improve per-node efficiency, recent advances in two fields can be borrowed: first, there has been a large amount of recent work that optimizes widely-used kernels such as sort, search, join, and aggregation for recent architectural trends [34, 57, 17, 35, 19]; and second, work in the HPC community for overlapping computation with inter-node communication to improve the utilization of available communication bandwidth.

This paper presents CloudRAMSort, a fast and efficient distributed sort on shared-nothing clusters. While sorting on shared-nothing clusters has been studied extensively [29, 32, 9, 54, 23, 24, 12], we revisit this topic in the context of cloud computing, where the large number of machines and increasing interconnect bandwidth has helped virtualize the memory available on each node and treat it as a large pool of hundreds of terabytes of memory. This enables CloudRAMSort to perform in-memory sorting on an aggre-

gation of tens to hundreds of commodity server processors. We carefully overlap computation and inter-node communication. We maximize per-node efficiency by exploiting modern architectural features such as multiple cores and data parallelism from SIMD (Single-Instruction Multiple Data) units while optimizing for available memory bandwidth. Our optimizations result in the highest performing sort on a distributed shared-nothing platform using commodity processors. Our CloudRAMSort sorts 1TB in only 4.6 seconds with each record having 10 Byte key and 90 Byte payload (as described in sorting benchmark [7]). Our algorithm handles skewed data using a dynamic multi-stage random sampling approach that can significantly reduce the number of samples required (and communication costs) for load-balancing.

We also evaluate a Hadoop implementation of TeraSort [45] with disk and DRAM (using RAM disk). We show that Hadoop TeraSort is around 2.7× faster when run from DRAM than from disk. However, the Hadoop TeraSort using RAM disk is still up to an order of magnitude slower than our CloudRAMSort on the same platform—even after accounting for a slower RAM disk bandwidth as compared to the raw DRAM bandwidth. This is because the focus of Hadoop is on disk, not on per-node efficiency. This gap further widens when the inter-node communication bandwidth increases, since the Hadoop TeraSort does not efficiently utilize processor resources and is therefore bound by the single-node performance.

We present a performance model to compute the efficiency of our implementation and to identify which part of the system is the bottleneck. Our achieved results match to the model within 7% on average. The model also helps us predict the performance of our sort on future architectures, with increasing per-node computation and memory bandwidth; as well as increasing inter-node communication bandwidth.

2. DATA-CENTER COMPUTING TREND

This section overviews recent trends of computing systems with respect to large-scale data processing and motivates our in-memory distributed sorting study.

Storage.

The transfer rate and seek time of disks have lagged behind disk capacity. In the 1980s, according to the analysis of Gray and Putzolu [27], if a record was accessed more often than every 5 minutes, it was faster and cheaper to store it in memory than on disk. In 2009's technology, the crossover point has increased to 30 hours [46].

As a result, systems handling large-scale data increasingly rely on memories for performance-sensitive tasks. For instance, in Google, high-performance products such as search, maps, or Google Fusion Tables [26] process data from distributed memory because it is virtually impossible to meet the extremely high throughput and stringent latency requirements by serving data from disks. As another example, SAS recently announced SAS In-memory Analytics to manage 'Big Data' meeting expectations of near-realtime response [6].

Going further, storage system designs that store all of the information in main memories such as RAMCloud [46] have been proposed. Even though individual DRAMs are volatile, RAMCloud can provide durability and availability equivalent to disk-based systems by replication and backup techniques.

In addition, recent studies have started exploring the application of emerging byte-addressable non-volatile memory technologies [38] for high-performance persistent data storage. Emerging non-volatile memories such as phase change memory (PCRAM) [39] and resistive RAM (RRAM) [41] can offer byte-addressable interface and performance both similar to those of DRAMs. Since these memory technologies are yet to be widely available, researchers have used DRAMs to emulate the performance of these emerging non-volatile memories [21, 20], and our evaluation of sort using RAMs can be similarly viewed as a proxy to such future systems.

Interconnection.

Once the entire data is stored in memory, the interconnect network can become the bottleneck in large-scale data analysis as demonstrated by our evaluation of sorting 100-byte tuples. This is also indicated by TritonSort where the second largest bottleneck after disks is 10Gbps Ethernet [50]. 40Gbps Ethernet is beginning to be commercially available and 100Gbps will also be available soon [43]. Further, innovations that allow continued bandwidth improvement through adopting optical communications continue [30, 18]. Such advances in optical interconnect technology are already being explored for data-center networks [25, 43].

However, network latency has not been improved as fast as bandwidth [47]. In fact, communication latency hiding has been considered as one of the most important optimizations in the High-Performance Computing (HPC) community: even a small fraction of unhidden communication latency can be a significant problem in achieving supercomputing-level scalability. Since the size of clusters used for large-scale data processing increasingly resembles that of super-computers, HPC practices may need to be adopted. For example, in our distributed sorting, communication is almost completely hidden as illustrated in Figure 4 when tuples are as small as 10 bytes.

In addition, the bus topology used in Ethernet is not scalable. Kim et al. [37] show that the pin bandwidth of routers have been increasing at Moore's law rates, and it is more efficient to use this increase by making the radix of the routers higher than simply making channels wider [37]. They also propose a high-radix topology called Flattened Butterfly [36, 10], which is more efficient than fat trees (a.k.a. folded-Clos) [40] in terms of hardware cost and energy consumption. Currently, high performance interconnects such as InfiniBand have not been widely adopted in data centers since their cost is higher than commodity Ethernet components. However, as more data is stored in memory and more sophisticated communication-heavy data analysis are demanded, the interconnect will increasingly become the bottleneck. In addition, the utility computing model that consolidates computing resources may make adoption of high performance interconnections easier, which, in turn, may democratize the cost by expanding the market size.

Per-node Computation and Energy Efficiency.

While interconnection is the bottleneck in our in-memory distributed sort with 100-byte tuples, compute becomes the bottleneck for 20-byte tuples as shown in Sec. 6. Therefore, it is crucial to effectively utilize the abundant computation resources presented in modern many-core processors with wider SIMD units [52]. Effectively utilizing these computation resources leads to a good per-node efficiency, which has important consequences: first, by solving the same problem with fewer nodes and/or in a shorter time, we reduce the probability of failure during the computation [48]. This can allow for less aggressive and light-weight fault tolerance techniques, which, in turn, can further improve the per-node efficiency. Second, per-node efficiency greatly affects the energy consumption. For example, Anderson and Tucek [11] show that it is possible for a sorting implementation on a small 8-node cluster to match that of a Hadoop cluster while being 22× more energy efficient, given an optimized single-node implementation. Energy efficiency is a primary concern of contemporary data-centers since

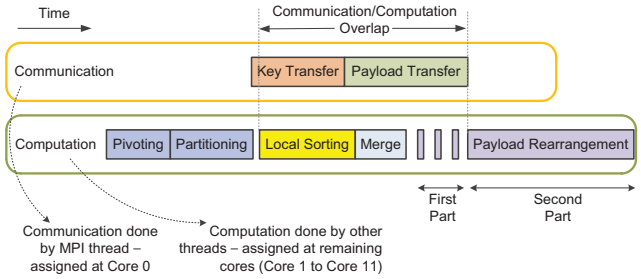

Figure 1: Overview of our distributed sorting algorithm.

their electric bills for three years are comparable to hardware costs, and the proportion of electric bills in the total cost of ownership is expected to increases significantly [42].

3. ALGORITHM

Input and Output:

The input consists of tuples, where each tuple is comprised of a key and payload, with the tuples evenly distributed among computing nodes (say $\mathcal{M}$ in total – Node_1, Node_2, .., $\text{Node}_\mathcal{M}$). The output is a globally sorted list of the tuples, with each node consisting of a sorted list of tuples, and (tuples of Node_1) $\leq$ (tuples of Node_2) $\leq$ .. $\leq$ (tuples of $\text{Node}_\mathcal{M}$). For ease of explanation, we assume that all the tuples consist of a fixed length key ($\mathcal{L}_\mathcal{K}$), and a fixed length payload ($\mathcal{L}_\mathcal{P}$). For load balancing, CloudRAMSort takes as input a skewness factor (s) [24], defined as the threshold for the maximum number of sorted records on any node divided by the average number of records on a node. In case the application requires equal number of output records on each node, we set s to a number slightly greater than 1 (e.g., 1.05) (details follow), and perform a post-process step to transfer the relevant sub-set of tuples to the destination nodes.

Motivation and Design Choices

CloudRAMSort uses a partitioning approach, where *each node* computes a mutually exclusive subset of its input tuples that need to be transferred to the remaining nodes in the system. Motivated by previous work, our system is based on a combination of probabilistic splitting [29, 24, 13] and histogram splitting [33, 55]. However, there are a few key differences: (1) we use a dynamic multi-stage approach that iteratively increases the number of samples until the desired load-balance (s) is achieved. (2) we do not sort the tuples before transferring them (contrary to other systems [55]). This helps in reducing the delay before transferring the tuples to their destination node, and also helps in overlapping compute with communication – since the receiver nodes can sort the incoming tuples upon receiving them from other nodes. (3) we employ the start-of-the-art cache-, SIMD-, and multi-core friendly sorting algorithms [17, 51], which efficiently utilizes computing resources of each node.

The overall structure of our algorithm is described in Figure 1. We now describe our algorithm step by step.

Pivoting and Partitioning: The *Pivoting phase* computes the ($\mathcal{M}$-1) pivot keys that divide the given set of input tuples between the $\mathcal{M}$ nodes within a skewness factor of s. This phase takes as input two parameters, p_{min}, the minimum number of pivots chosen by each node during the execution, and p_{max}, the maximum number of pivots chosen by each node. Consider the i^{th} iteration. p_i below is initialized to p_{min} for the first iteration. Each iteration consists of the following sub-steps.

1. Each node computes p_i pivot keys, and broadcasts them to the remaining nodes.

2. Each node sorts $p_i\mathcal{M}$ keys, and scans through its list of input tuples to compute histogram of number of tuples with keys falling between each pair of adjacent pivot keys. The histogram consists of $p_i\mathcal{M}$ values. This step is parallelized across multiple cores on the node. Note that since the key and payload are stored together, the scanning of tuples needs to read in both the key and payload for each tuple, and may lead to an increased bandwidth consumption. Hence, as an *optimization*, the first step writes out the keys into a separate array, which is subsequently read for following iterations. This helps greatly reduce the bandwidth consumption for medium-to-large size payloads.

3. Each node broadcasts the above histogram to the remaining nodes. The resultant $\mathcal{M}$ histograms (each with $p_i\mathcal{M}$ bins) is reduced to compute the number of tuples in each of the $p_i\mathcal{M}$ bins. This step is also parallelized on each node.

4. The histogram is scanned to compute ($\mathcal{M}$-1) pivots such that the load-balance criterion is satisfied. If a solution exists, the pivoting step terminates with the computed pivot elements. In case no solution exists, p_i is doubled, and if $p_i \leq p_{max}$, go back to step (1). Else go to step (5).

5. The algorithm fails to find the ($\mathcal{M}$-1) pivots in case there are large number of duplicate keys in the input (pathological example being all keys the same). In such a case, we consider the payload to be part of the key too, and go back to step (1) with $p_i = p_{max}$. For all realistic scenarios we tried on, this terminated with the appropriate ($\mathcal{M}$-1) pivots.

Having computed the ($\mathcal{M}$-1) pivots, the *Partitioning phase* traverses through the input tuples and scatters each tuple to the relevant bin (one for each node). This step is again parallelized across the cores of the node. In order to initiate the transfer of tuples as early as possible, during partitioning, we store the key and payload into separate arrays. Hence, for each destination node, all the keys are stored together and all the payloads are stored together in a separate location.

Per-Node Computation: As soon as a node receives keys from some other node, it appends a unique record id (or *Rid*) with each key (to keep track of the original position of the key). The Key_Rid's from each node are then sorted cooperatively by multiple cores on each node. We choose the appropriate sorting algorithm (radix v/s mergesort) based on the key-length [51]. The resultant implementation efficiently exploits the available caches, cores, and SIMD to improve the node-efficiency. We refer to this phase as *Local Sorting*. Once the Key_Rid pairs from each node are sorted, they are merged [17] to compute a sorted list of Key_Rid pairs (*Merging Phase*). As the payloads are received from other nodes, we scatter each payload to its appropriate final location (using the unique Rid assigned to each incoming tuple). This phase is referred to as the *Payload Rearrangement Phase*. We implemented this phase such

that there is overlap of the communication between nodes and the computation on each node to maximize the system efficiency.

Inter-Node Communication: In order to optimize our overlap of computation and communication, we transfer the tuple in two steps: first, we send only the keys to all the nodes, and then we send the respective payloads – the motivation being that local sort and merge can be done with Key_Rid pairs only (more architecture friendly to avoid moving around large payloads). Thus, it is critical to receive keys as soon as the Pivoting phase has computed the relevant pivots and the Partitioning phase has divided the tuples based on their destination node. The receipt of payloads can be delayed since the payload rearrangement can be done as the last step.

We now present the implementation details of the various steps described above.

4. IMPLEMENTATION

This section elaborates on a few details of our algorithm described in the previous section. These mainly pertain to our techniques to increase overlap of computation and communication in our implementation. We also briefly discuss our sorting and merging approach, as well as the payload rearrangement step. In the next section, we present a detailed analytical model for our implementation that can help compute the efficiency of our implementation and analyze performance bottlenecks.

Figure 1 shows the overall flow of our algorithm. The major steps, including pivoting and partitioning of the data among the nodes, the transfer of keys and payloads, local sorting and merging of keys at each node, and finally, payload rearrangement were described in Section 3. Many of our design choices were to enable overlap between computation and inter-node communication, and we start by describing the mechanics by which this overlap is achieved.

MPI + Pthreads:
The Message Passing Interface [8] (MPI) is a widely used standard in the HPC community to abstract communication details of high-bandwidth links like InfiniBand. Communication system stacks and standard groups like OpenFabrics [5] have allowed for commonly available Ethernet devices on commodity hardware to communicate using the MPI standard as well, allowing for portability of software between commodity clusters and high-performance systems.

In our implementation, we use a combination of MPI calls for inter-node communication and the pthread library for thread-level parallelism within each node. We dedicate one of the threads per node after the partitioning step (usually thread 0, called MPI thread) to perform the MPI calls on behalf of all threads in the node. This thread is responsible for all MPI calls for inter-node communication, as well as any copies to/from any internal message buffers required for the MPI communication. All other threads perform computation tasks - this is thus a "hybrid" threading approach that allows for overlap of computation and communication as described below.

Overlap of computation and communication:
Our general scheme is to use MPI asynchronous calls that allow for a thread to indicate that one or more messages are expected by this node, without explicitly blocking for them. For example, after the partitioning step, the keys to be communicated to every other node is known. The MPI thread first posts a set of receive requests using `MPI_Irecv` calls. These calls notify the MPI communication stack to be ready to receives a set of messages (containing keys) from other nodes, but do not block at this point. Once these non-blocking receives are issued, the MPI thread then posts a set of `MPI_Isend` calls (one for each other node) that initiates the sending of the keys.

The thread then follows up with `MPI_Waitany` which takes in a set of receive requests posted earlier, and blocks until one or more of these messages are received. The messages are allowed to arrive in arbitrary order depending on the network. Once any message (with keys) is received, the MPI thread puts the keys into an user-space array (visible and coherent across all threads), uses a set of shared variables (one per message per compute thread) to notify the other threads that a message has been received, and then goes back to waiting for any remaining messages.

On the computation side, the remaining threads were blocked waiting for the MPI thread to signal that at least one message with keys has arrived. Once even one message arrives, the threads cooperatively work to sort the keys contained in the message. Once messages have arrived from all other nodes, the threads go into a cooperative parallel merge phase where all the locally sorted keys (one per node) are merged. A similar mechanism is then carried out for payloads - we describe this in more detail later in this section.

Local Sorting and Merging: We use a local sort and merging algorithm that efficiently utilizes compute and bandwidth resources on a single-node (including thread, data, and memory level parallelism). We choose between state-of-the-art radix sort and merge sort algorithms depending on the key size. For more details, please refer to Satish et al. [51] and Chhugani et al. [17].

Payload Rearrangement: As soon as the key transfer completes, the MPI thread begins transferring the payloads. For medium to large sized payloads, the payload transfer time is much longer than the time taken by the compute threads to sort and merge the keys. Consequently, if we wait for all payloads to arrive before starting to rearrange the payloads according to the merged keys, then the compute threads are idle for a significant amount of time.

In order to avoid this, we split the payload rearrangement into two phases, one while some payload transfers are still ongoing, and the second after all payload transfers are complete. In the *first phase*, as each payload message is received (and signaled by the MPI thread), the payloads inside the message are scattered to their final location in the output sorted list. This is done using a scan of the Key_Rid tuples and only updating those records whose rids are in the range that is contained in the payload message. At some point while this per-message payload rearrangement is ongoing, the MPI thread receives all remaining payload messages. At that point, the MPI threads signal the compute threads to switch to the *second phase* of payload rearrangement. Here, the threads traverse through the sorted list of Key_Rid tuples one final time, and update it with the payload indexed by the Rid—this is now guaranteed to be valid since all payloads have arrived. To avoid processing the Rid's that have already been handled in the first phase, we use tags on each processed Key_Rid entry to indicate that processing is done.

In contrast to the Key sorting step where each message can be independently sorted with no loss of efficiency, the Payload rearrangement step does suffer loss of efficiency due to the first phase. The most efficient approach with fewest passes over the Key_Rid tuples would be to delay payload rearrangement until all payload messages arrive. However, since this excess computation in the first step overlaps with communication (by design), it is in essence

"free". Only the second phase occurs after payload communication, and is thus exposed in overall runtime.

5. PERFORMANCE MODEL

We now present an *analytical model*, aimed at computing the total run-time (in *cycles per tuple*) required by our CloudRAMSort algorithm. The model serves the following purposes: (1) Computing the efficiency of our implementation. (2) Analyzing performance bottlenecks. (3) Projecting performance for tuples with different key and payload sizes, and varying architectural features such as the number of cores and network bandwidth.

Notation:

$\mathcal{N}$: Number of input tuples (to be sorted).
$\mathcal{L}_\mathcal{K}$: Size of the Key (in bytes).
$\mathcal{L}_\mathcal{P}$: Size of the Payload (in bytes).
$\mathcal{M}$: Number of computational nodes.
$\mathcal{P}$: Number of cores per node.
$\mathcal{B}_{MPI}$: Average achievable MPI bandwidth between any pair of computational nodes (in bytes/cycle).
$\mathcal{B}_M$: Average achievable main memory bandwidth (in bytes/cycle).
Itr : Number of iterations performed during the pivoting step.
p_i : Number of pivots chosen per computational node (for the i^{th} iteration). Varies between p_{min} and p_{max}.
s : Skewness Factor [24], defined as the threshold for the maximum number of sorted records on any node divided by the average number of records per node.

Note that each node has $\mathcal{N}/\mathcal{M}$ tuples at the beginning. In our specific implementation, we set *s* to 1.05, while p_{min} and p_{max} are set to 32 and 1024 respectively. We now describe the computational time of individual steps:

Pivoting and Partitioning:

Consider the i^{th} iteration. Each node computes p_i pivot keys, and broadcasts them to the remaining ($\mathcal{M}$-1) nodes to obtain $p_i\mathcal{M}$ pivots. These $p_i\mathcal{M}$ pivots are sorted and the list of keys are traversed to obtain the histogram of number of tuples falling between each pair of pivots. Assuming ($p_i \ll \frac{\mathcal{N}}{\mathcal{M}^2}$) (the typical use case), computing, broadcasting, and sorting of the pivots contributes an insignificant amount to the total run-time. However, computing the histogram involves calculating the histogram bin index for each key. This involves performing a binary search for each key, and evaluates to around $10\lceil\log(p_i\mathcal{M})\rceil$ cycles per core [34]. Assuming perfect scaling across the processing cores per node and across the nodes, we obtain $\mathcal{T}^{Pivoting}$, the time taken to perform pivoting (in *cycles per tuple*) as:

$$\mathcal{T}^{Pivoting} = \sum_{i=1}^{Itr} \frac{10\lceil\log(p_i\mathcal{M})\rceil}{\mathcal{MP}} \tag{1}$$

For a specific example of say $\mathcal{N}$ = 5*10^9 tuples (500 GB of data, with $\mathcal{L}_\mathcal{K}$ = 10 bytes and $\mathcal{L}_\mathcal{P}$ = 90 bytes), $\mathcal{M}$ = 128 nodes, and $\mathcal{P}$ = 12 cores/node, with *Itr* = 2 (hence p_1 = 32 and p_2 = 64), we obtain $\mathcal{T}^{Pivoting}$ = (120 + 132)/12/128 = 21/128 = 0.17 *cycles per tuple*.

Once the appropriate ($\mathcal{M}$-1) pivots has been obtained, the input tuples are partitioned into separate Key and Payload packets for each of the other nodes. This *partitioning* involves reading the input tuples and writing them to the appropriate memory locations. On modern architectures, partitioning is expected to be bound by the available memory bandwidth, and since writing data also involves reading the destination cache-lines, we obtain $\mathcal{T}^{Partitioning}$, the time taken to perform partitioning (in *cycles per tuple*) as:

$$\mathcal{T}^{Partitioning} = \frac{3(\mathcal{L}_\mathcal{K} + \mathcal{L}_\mathcal{P})}{\mathcal{MB}_\mathcal{M}} \tag{2}$$

We obtain $\mathcal{B}_M$ of around 12 bytes/cycle on each node, and hence $\mathcal{T}^{Partitioning}$ = 300/12/128 = 25/128 = 0.20 *cycles per tuple*.

Summing up Eqn. 1 and Eqn. 2, we obtain:

$$\mathcal{T}^{Pivoting+Partitioning} = \frac{1}{\mathcal{M}}\left(\sum_{i=1}^{Itr}\frac{10\lceil\log(p_i\mathcal{M})\rceil}{\mathcal{P}} + \frac{3(\mathcal{L}_\mathcal{K} + \mathcal{L}_\mathcal{P})}{\mathcal{B}_\mathcal{M}}\right) \tag{3}$$

Key and Payload Transfer: As explained in Section 4, we first transfer the keys to all the other computing nodes. This is followed by transferring the respective payloads. Since there might be an imbalance of a factor up to *s*, the time taken to transfer keys ($\mathcal{T}^{Key}$) in *cycles per tuple* as:

$$\mathcal{T}^{Key} = \frac{s\mathcal{L}_\mathcal{K}}{\mathcal{MB}_{MPI}} \tag{4}$$

Continuing with the above example, and using $\mathcal{B}_{MPI}$= 0.55 bytes/cycle, we obtain $\mathcal{T}^{Key}$ = (1.05) (10)/0.55/128 = 19/128 = 0.15 *cycles per tuple*. Similarly, the time taken to transfer payloads ($\mathcal{T}^{Payload}$) in *cycles per tuple* equals:

$$\mathcal{T}^{Payload} = \frac{s\mathcal{L}_\mathcal{P}}{\mathcal{MB}_{MPI}} \tag{5}$$

Continuing with the above example, we obtain $\mathcal{T}^{Payload}$ = (1.05)(90)/0.55/128 = 171/128 = 1.34 *cycles per tuple*.

Summing up Eqn. 4 and Eqn. 5, we obtain:

$$\mathcal{T}^{Key+Payload} = \frac{s}{\mathcal{MB}_{MPI}}(\mathcal{L}_\mathcal{K} + \mathcal{L}_\mathcal{P}) \tag{6}$$

Local Sorting and Merging: As the nodes receive a packet with the keys, each key is appended with a unique *Rid* and the resultant Key_Rid pairs are sorted. For each packet, ($\mathcal{P}-1$) cores per node cooperatively sort each packet. Let $\mathcal{L}_\mathcal{K}^{'}$ denote the length of each Key_Rid pair. Depending on the key length, we either use radix-sort or merge-based sort to sort the pairs [51]. In the analysis below, we assume $\mathcal{L}_\mathcal{K}^{'}$ is *greater than* 8 bytes, and hence merge-based sort is used. Each node receives a maximum of $s\frac{\mathcal{N}}{\mathcal{M}}$ tuples summed up over $\mathcal{M}$ nodes (including itself). Although the distribution of tuples received from each node may vary significantly over the different nodes, the total number of tuples is still bounded. Consider the j^{th} node, and say $\mathcal{N}_{ij}$ denotes the number of tuples obtained from i^{th} node ($i \in [1\ ..\ \mathcal{M}]$). Sorting $\mathcal{N}_{ij}$ tuples requires $\lceil\log(\mathcal{N}_{ij})\rceil$ merging iterations, with each iteration requiring around 18 cycles on a single-core (for $\mathcal{L}_\mathcal{K}^{'}$ between 8 and 16 bytes) [17, 51]. we obtain $\mathcal{T}^{Sorting}$, the time taken to perform sorting (in *cycles per tuple*) as:

$$\mathcal{T}^{Sorting} = \max_{i=1}^{\mathcal{M}}\frac{18\lceil\log(\mathcal{N}_{ij})\rceil}{\mathcal{M}(\mathcal{P}-1)} \leq \frac{18\lceil\log(s\frac{\mathcal{N}}{\mathcal{M}})\rceil}{\mathcal{M}(\mathcal{P}-1)} \tag{7}$$

For a specific example of $\mathcal{N}$ = 5*10^9 tuples, $\mathcal{M}$ = 128 nodes and *s* = 1.05, $\mathcal{T}^{Sorting} \leq$ (18)(26)/11/128 *cycles per tuple* = 42.5/128 = 0.33 *cycles per tuple*. For the average case, $\mathcal{T}^{Sorting} \sim$ (18)(20)/11/128 = 33/128 = 0.26 *cycles per tuple*.

As far as merging is concerned, the threads cooperatively merge the $\mathcal{M}$ sorted chunks ($\lceil\log(\mathcal{M})\rceil$ iterations). Each iteration reads

and writes data from/to the main memory. For our specific implementation, we use streaming stores, and hence do not read the destination cache-line while writing. For modern architectures, each iteration is expected to be bound by memory bandwidth, and the resultant run-time, $\mathcal{T}^{Merging}$, the time taken to perform merging (in *cycles per tuple*) equals:

$$\mathcal{T}^{Merging} = \frac{2(\mathcal{L}_{\mathcal{K}}^{'})\lceil \log \mathcal{M} \rceil}{\mathcal{MB}_{\mathcal{M}}} \quad (8)$$

We obtain $\mathcal{B}_M$ of around 12 bytes/cycle on each node, and hence $\mathcal{T}^{Merging}$ = (28)(7)/12/128 = 16/128 = 0.13 *cycles per tuple*.

Payload Rearrangement: This phase consists of two phases. In the *first part*, for each received payload packet, payloads inside the packet are scattered to the final location in the sorted list of Key_Payload tuples. This part is carried out for all packets that are received before all the payload packets have been received. Once all payload packets are received, we switch to the *second part*, where we traverse through the sorted list of Key_Rid tuples, and scatter the payloads from all the non-processed packets. Since the first part overlaps with the packet transfer time, we *focus on the second part in this analysis*, which is the additional time spent once all the payload packets have been received.

Let $\alpha_{rem.}$ denote the fraction of tuples that have not been processed at the end of the first part. $0 \leq \alpha_{rem.} \leq 1$. The second part involves traversing through each Key_Rid tuple, and in case the payload for the corresponding tuple has not been scattered, we copy the payload from the appropriate memory location, and store it to the destination address. On modern architectures, this read/write is expected to be bound by the memory bandwidth. Hence, $\mathcal{T}^{Payload\_Rearr.}$, the time taken to perform rearrangement (in *cycles per tuple*) equals:

$$\mathcal{T}^{Payload\_Rearr.} = \frac{\mathcal{L}_{\mathcal{K}}^{'} + (\alpha_{rem.})(2\mathcal{L}_{\mathcal{K}} + 3\mathcal{L}_{\mathcal{P}})}{\mathcal{MB}_{\mathcal{M}}} \quad (9)$$

Note that $\alpha_{rem.}$ is a function of network bandwidth, number of computational cores, etc. As an example, for $\alpha_{rem.}$ = 0.8, $\mathcal{T}^{Payload\_Rearr.}$ evaluates to (14 + 232)/12/128 = 21/128 = 0.16 *cycles per tuple*.

Overall Execution Time: The total execution time is computed by adding up (1) pivoting and partitioning time (Eqn. 3), (2) maximum of sorting+merging (Eqn. 7 + Eqn. 8) and key+payload transfer times (Eqn. 6), and (3) payload rearrangement time (Eqn. 9). Hence, $\mathcal{T}^{Overall}$ equals:

$$\mathcal{T}^{Overall} = \mathcal{T}^{Pivoting+Partitioning} + \max\left(\mathcal{T}^{Sorting} + \mathcal{T}^{Merging}, \mathcal{T}^{Key+Payload}\right) + \mathcal{T}^{Payload\_Rearr.} \quad (10)$$

6. RESULTS

Platform Specification: We evaluate the performance of CloudRAMSort on a 256-node Intel Endeavor cluster. Each CPU node consists of 24GB DRAM and a 2.93 GHz dual-socket Intel Xeon X5680 CPU with 6 cores per socket (12 cores per node) and 2-way SMT per core. In addition to scalar units, each core has a 4-wide SIMD unit. Each socket has a 12MB last-level shared cache, hence a total of 24MB on each node. Each node has 300 GFLOPS peak computation throughput and 30GB/sec peak memory bandwidth between the DRAM and each socket. Each node is operated by Red Hat Enterprise Linux Server 6.1 with kernel version 2.6.32.

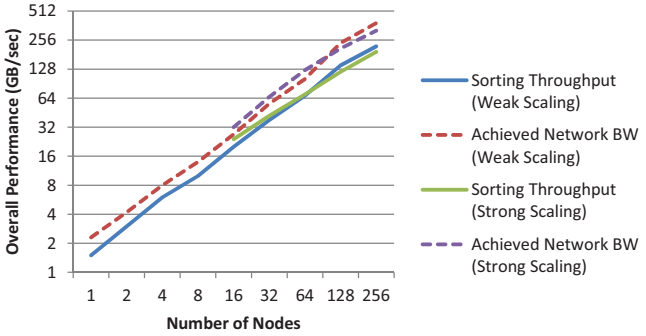

Figure 2: Sorting throughput and achieved interconnect bandwidth, measured in GB/sec, for both strong and weak scaling. Each tuple consists of 10 byte key and 90 byte payload. Strong scaling fixes the data set size (100GB) and varies the number of nodes from 16 to 256. Weak scaling varies the total input size proportional to the number of nodes as 4GB/node (1TB for 256 nodes).

Clusters are connected through two network interface cards: 1) QDR InfiniBand 4X with the peak bandwidth 4 GB/sec and a two-level 14-ary fat tree topology [40]. 2) 1Gbps Ethernet switch.

Input Datasets: In order to be compatible with standard benchmarks [7], we generated tuples with key length ($\mathcal{L}_{\mathcal{K}}$) equal to 10 bytes, and varied the payload lengths ($\mathcal{L}_{\mathcal{P}}$) between 10 – 90 bytes. 90 bytes payload represents the standard input dataset used for various sorting benchmarks [7]. We used the generator code (gensort [3]) for this purpose. In order to test the applicability of CloudRAMSort for more skewed input distributions, we also generated keys using Gaussian and Zipf [28] distributions. Zipf distributions are known to model a wide range of data usage scenarios found on web and others [14].

Throughput Measure: We measure the **sorting throughput** in GB/sec by dividing the overall dataset size by the total execution time of the algorithm (in seconds). In order to highlight the efficiency, we also plot the achieved interconnect bandwidth on our system (in GB/sec).

6.1 Scalability of CloudRAMSort – Weak and Strong Scaling

Figure 2 highlights the strong and weak scaling of CloudRAMSort. We also plot the achieved interconnect bandwidth. For strong-scaling, we fix the total input dataset size to 100GB and vary the number of nodes from 16 to 256. We achieve a sorting throughput of 18 GB/sec on 16 nodes and scale to around 193 GB/sec on 256-nodes. We achieve a sub-linear 10× scaling with 16× nodes due to the following reason: Since the total problem size is kept constant with increasing number of nodes, the time spent in pivoting and partitioning (Eqn. 3) increases with increasing number of nodes, and leads to the sub-linear scaling (e.g., with 256 nodes, around 20% of the run-time is spent before the key transfer starts—while the corresponding time is only 5% for 16 nodes). Even on 256 nodes (with around 400 MB/node), our achieved throughput is around 60% of the achieved interconnect bandwidth.

To test the weak-scaling of the system, we varied the input dataset size in proportion to the number of nodes, keeping the data per node constant to 4GB. We achieve a throughput of around 220 GB/sec on 256-nodes, and sort **1 Terabyte in 4.6 seconds**.

6.2 Varying Input Distributions

Figure 3 plots the achieved throughput as we vary the input dis-

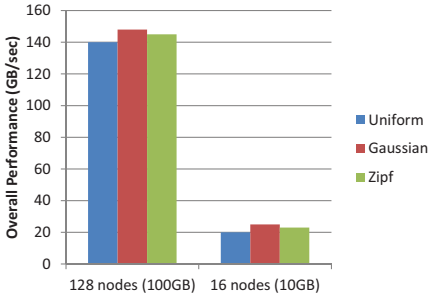

Figure 3: Sorting throughput on various distribution: Uniform, Gaussian, and Zipf, with 100 byte tuples.

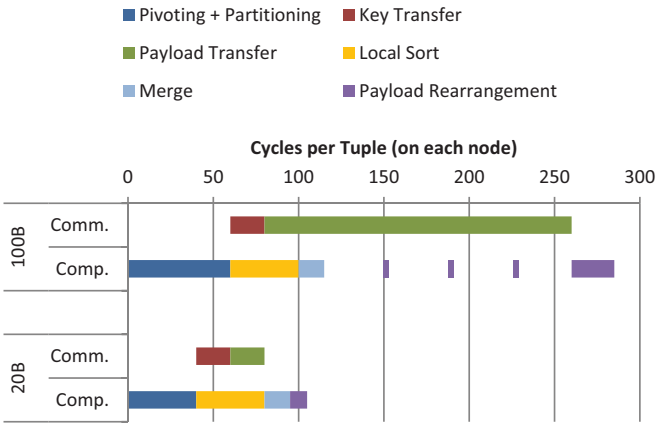

Figure 4: Breakdown of times spent on each phase of CloudRAMSort for a 128-node execution. We split the time into two categories: Communication and Computation. The execution times (cycles/tuple) are the average time spent on each node with a dataset of 5 billion tuples, with key size ($\mathcal{L}_\mathcal{K}$) of 10 bytes, and payload sizes ($\mathcal{L}_\mathcal{P}$) of 90 bytes (top: 500 GB), and 10 bytes (bottom: 100 GB).

tribution — uniform, Gaussian and Zipf. We achieve 140 – 150 GB/sec throughput with 100 GB on 128 nodes, and 20 – 25 GB/sec with 10 GB on 16 nodes. These two dataset sizes are chosen to show that we achieve high efficiency with varying input data per node and different input distributions. For both Gaussian and Zipf distributions, the pivoting phase takes 2 – 4 iterations to produce desired pivots. In fact, our throughput on Zipf and Gaussian is slightly larger than uniform distribution due to the reduced randomness while scattering the data during the partitioning step which leads to better cache behavior and slightly reduced run-time (around 5%). Scattering the data in the partitioning step generates random writes into different partitions and thus increasing TLB/cache misses. Skewed data actually helps reduce this latency impact.

6.3 Varying Payload Sizes

Figure 4 plots the timing breakdown for two different payload sizes—10 bytes and 90 bytes. These two data points reflect the two extremes—sorting with 10 bytes payload requires effective utilization of computational resources on each node to achieve a high sorting rate, while sorting with 90 bytes payload requires effective utilization of both computation and communication resources.

For 10-byte payloads, around 40 cycles is spent on Pivoting + Partitioning. At the end of this phase, the keys are distributed to destination nodes, followed by payloads transfers. As shown, the communication channel is now active, and time between 40 – 80 cycles/tuple is spent in transferring the keys and payloads. On the compute side, there is a small delay (less than 2 cycles/tuple) before which the keys being arriving at the nodes, and the local sorting is performed, which takes around 40 cycles/tuple, followed by the 7 iterations of merging (around 15 cycles). Note that for a 20-byte tuple, the interconnect throughput is fast enough for the payloads to arrive before the Key_Rid pairs are sorted. Hence, the payload rearrangement is initiated with all the payloads received by the nodes ($\alpha_{rem.}$ = 1 (Sec. 5)). Note that the overall run-time is same as the total computation time on the nodes – and hence would improve with increasing per-node compute by increasing the number of cores, memory bandwidth, etc. (more details later in Sec. 6.3.1).

The 90-byte payload represents the case where the communication of tuples across the network is the major performance bottleneck. Around 70% of the total execution time is spent in transferring the data across the nodes. The pivoting + partitioning phase takes longer than the 10-byte payload due to the increased time in partitioning (due to larger payload size). The execution time would benefit with increase in network bandwidth by reducing the communication time. (more details in Sec. 6.3.1).

6.3.1 *Comparison with Analytical Model and Performance Projection*

We now compare our achieved performance to the analytical model developed in Section 5. Consider Fig. 4, with total size of each tuple equal to 100 bytes. Hence, $\mathcal{L}_\mathcal{K}$ = 10 bytes, and $\mathcal{L}_\mathcal{P}$ = 90 bytes. $\mathcal{N} = 5 \times 10^9$ tuples, $\mathcal{M}$ = 128 nodes, and $\mathcal{P}$ = 12 cores/nodes. For this specific input dataset, *Itr* = 3. We abbreviate cycles/tuple as *cpt* for the discussion below.

Plugging these values into Eqn. 3, we obtain $\mathcal{T}^{Pivoting+Partitioning}$ = (33+25)/128 *cpt*. Hence, time on each node = 33+25 = 58 *cpt*. The actual measured run-time is around 60 *cpt*, which is within 3% of the analytical model. Transferring the 100 byte tuple would take around 190 *cpt* (Eqn. 6), while the obtained performance is around 200 *cpt* (5% difference). The local sorting time ($\mathcal{T}^{Sorting}$)(Eqn. 7) evaluates to around 33 *cpt* on each node, while the actual measured performance is around 37 *cpt* (within 9% of the projected performance). The time for merging in Eqn 8 evaluates to 16 *cpt*, while the actual obtained performance is around 15 *cpt* (7% difference). Finally, the payload rearrangement time from Eqn. 9 (with $\alpha_{rem.}$ = 0.9) equals 23 *cpt*, while the achieved performance is around 25 *cpt* (8% difference). Thus, the overall analytical performance evaluates to (Eqn. 5) (58 + 190 + 23) = 271 *cpt*. The obtained performance is around 285 *cpt*, which is **within 5%** of the analytical performance.

For the payload with 10 bytes (total tuple length of 10+10 = 20 bytes), $\mathcal{T}^{Partitioning}$ reduces to around 5 *cpt* (Eqn. 2), which reduces $\mathcal{T}^{Pivoting+Partitioning}$ to around 38 *cpt*. The obtained performance is around 40 *cpt*, and accounts correctly for the reduced time for the partitioning phase. Note that in this case the interconnect bandwidth is not a performance bottleneck, and hence $\alpha_{rem.}$ = 1.0 results in 5.4 *cpt*, while the actual measured time is around 7 *cpt*. The total analytically computed time is around (38 + 33 + 16 + 5.4) $\sim$ 93 *cpt*, while the measured performance is around 103 *cpt* (within 10%).

When measured for the complete range of run-times presented in this paper, our model was within 7% of the actual performance on average, and is therefore used for the performance projection below with varying architectural features such as increasing number of cores, main memory bandwidth, and interconnect bandwidth.

Performance Projection: We use the analytical model developed in Section 5 to project performance for various tuple sizes with varying architectural features in Table 1. For example, consider tu-

100-byte tuples	Our Model	2× Network BW	2× (Cores + Memory BW)
Pivot + Partitioning	58	58	**29**
Key/Payload Transfer	190	**95**	190
Sorting	33	33	**17**
Merging	16	16	**8**
Rearrangement	23	23	**11**
Total	271	**176**	**231**

20-byte tuples	Our Model	2× Network BW	2× (Cores + Memory BW)
Pivot + Partitioning	38	38	**19**
Key/Payload Transfer	38	**19**	38
Sorting	33	33	**17**
Merging	16	16	**8**
Rearrangement	5.4	5.4	**2.7**
Total	93	93	**46**

Table 1: Performance Projection(s) (model described in Section 5). All numbers represent cycles/tuple on each node. The second column shows the individual breakdown of various steps using the current system parameters. Our actual run-time is within 7% of the projected performance on average. Column 3 lists the projection by increasing the interconnect bandwidth by 2×, which increases the throughput by 1.53×(100-byte tuples). In contrast, doubling the number of cores and memory bandwidth on each node (Col. 4) increases the throughput by 1.17× (100-byte tuples) and 2×(20-byte tuples) (details in Section 6.3.1). The numbers in bold are those affected by architectural parameters.

ples with key-size of 10 bytes, and payload size of 90 bytes. The projected run-time (on 128 nodes) is around 271 cpt. If the interconnect bandwidth is say doubled (to 1.1 bytes/cycle from the current 0.55 bytes/cycle), the performance would reduce to around 176 *cpt*, an increase in sorting throughput of 1.53×. For smaller payload sizes (say 10 bytes), this change would have no effect on the run-time.

Similarly, increasing the number of cores by 2× (without changing the bandwidth) would reduce the sorting time for 20-byte tuples to around 60 *cpt* (from 93 *cpt*) – an increase in throughput of 1.55×. Increasing the memory bandwidth by 2× would only increase the throughput by 1.16× (80 *cpt* as compared to 93 *cpt*). In contrast, for larger sized tuples (100 bytes), 2× increase in cores would increase the throughput by only 1.1×, while a 2× increase in the memory bandwidth would only result in 1.04× increase in throughput. By doubling both the per-node computational power and per-node memory bandwidth, the throughput for 20-byte tuples would increase by 2×, while the corresponding increase in throughput for 100-byte tuples would be around 1.17×.

To summarize, the benefit of various architectural advances depends on the usage scenario of the sorting algorithm, i.e., the size of keys, payloads, etc. CloudRAMSort is optimized for each of these features and would be a system of choice for sorting large-scale in-memory data for current and future systems.

6.4 Comparison with Other Systems

We compare CloudRAMSort with a Hadoop implementation of TeraSort [45] with varying inter-node communication bandwidths. Note that TeraSort is focused on disk-based sorting (referred to as Hadoop$_{Disk}$) and hence is not necessarily optimized for performing per-node compute optimizations. For fair comparison, we ran TeraSort using RAM disk (referred to as Hadoop$_{RamDisk}$). Our disk and RAM disk benchmark tests (using `bonnie++` [2]) and other benchmark results [44] indicate that read/write bandwidth of RAM disk is about 4× slower than that of raw main memory accesses. Hence, for comparison purposes, we optimistically project the TeraSort execution times using RAM by scaling down the run-times by 4× (and refer to them as Hadoop$_{RamSort_Projected}$) to infer performance differences between the two systems (TeraSort and CloudRAMSort). We use 0.21.0 version of Hadoop run-time, the TeraSort implementation included in it, and Oracle Java 64-bit server SDK 1.6.0_27 version. We tune parameters such as HDFS block size, the buffer capacity of in-memory merging, log-level, the number of mappers/reducers, task setup/cleanup policy, and Java virtual machine heap size following [45, 4].

On an 1Gbps Ethernet, using a total dataset size of 32 GB (320M tuples, each of length 100 bytes) on 16 nodes, the total sorting time of Hadoop$_{Disk}$ is 506 seconds, while Hadoop$_{RamDisk}$ takes 233 seconds. In contrast, CloudRAMSort on the same 1Gbps Ethernet took 34.1 seconds, signifying a speedup of 6.8× over Hadoop$_{RamDisk}$ and a projected speedup of 1.7× over Hadoop$_{RamSort_Projected}$.

In order to test the scalability of Hadoop with increasing interconnect bandwidth, we ran another set of experiments on a higher bandwidth InfiniBand interconnect. The Hadoop TeraSort was appropriately modified to use InfiniBand. The resultant run-times indicate a **projected speedup of 18.5X** of CloudRAMSort as compared to Hadoop$_{RamSort_Projected}$. The increase in speedup is due to the relative increase in the proportion of time spent on sorting/merging, and indicative of better utilization of computational resources (multi-cores, caches, SIMD execution) and interconnect bandwidth by CloudRAMSort as compared to TeraSort. This relatively low per-node efficiency of Hadoop TeraSort agrees with the observation of Anderson and Tucek [11], who reported only 6% and 10% utilization of disk and network I/O bandwidth, respectively. As mentioned earlier in Section 2, per-node computational capability and interconnect bandwidth are expected to increase on upcoming and future processor generations, and it is imperative to design a system that exploits these architectural trends.

7. RELATED WORK

Jim Gray believed that sort is at the heart of large scale data processing and created competitions for disk-to-disk sorting [7]. In 2009, O'Malley and Murthy [45] set a record for the Daytona Gray sort (0.578TB/min with 3,452 nodes) using an open source implementation of MapReduce [22] in Hadoop [1]. However, Anderson and Tucek [11] point out that the impressive results obtained by operating at high scale mask a typically low individual per-node efficiency, requiring a larger-than-needed scale. For example, the Hadoop sort results indicate that only 6% and 10% of the peak disk and network I/O bandwidth were utilized [11]. As a result, the Hadoop cluster used for the 2009 Gray sort achieves only 3MB/s bandwidth per node on machines theoretically capable of supporting a factor of 100 more throughput [50].

TritonSort [50] significantly improves the per-node efficiency, setting a new record for the Indy Gray sort in 2010 and again in 2011 (0.938TB/min with 52 nodes). The key optimization of TritonSort is *balanced* system design: since disk I/O is the bottleneck of typical disk-to-disk sorting pipelines, they focused on increasing disk bandwidth by using a large number of disks—a total of 832 disks. We however focus on in-memory sorting where the design choices are motivated by the need for 1) load balancing computation across nodes in the presence of skewed data, and 2) overlapping communication and computation.

While TritonSort does not consider skewed data, typical distributed sorting algorithms use *splitter-based* sorting to ensure load balance [29, 32, 9, 54, 55]. It is important to choose the right splits (correspond to pivots described in Sec. 3) to ensure load-balance, and researchers have proposed primarily four techniques: probabilistic splitting (a.k.a. sample sort) [29], exact splitting [32, 9], splitting by regular sampling [54], and histogram splitting [33].

Probabilistic splitting samples a subset of data and use theoretical results such as [13, 53] to control load-imbalance. Note

that these theoretical results assume no duplicated keys. DeWitt et al. [24] suggest 100 samples per processor based on the theoretical results and their evaluations. Exact splitting [32, 9] ensures perfect load balance, which is theoretically interesting but takes too long to compute the splits, resulting in poor scaling [24]. Both regular sampling [54] and histogram splitting [33] require each node to first sort its local data. Regular sampling then proceeds by picking splits that are equally distanced over the sorted results. Histogram splitting improves on this by iteratively refining these splits based on histograms until desired load balance is achieved. Solomonik and Kalé [55] show that histogram splitting is more scalable than regular sampling.

Our splitting scheme described in Sec. 3 can be viewed as a combination of probabilistic and histogram splitting. Rather than refining an initial set of splits (whose size is fixed as the number of nodes) as in histogram splitting, we first start with a small sample set and iteratively double the number of samples until a user-specified load balance is achieved. As per the theoretical bounds described in [13, 24], the probability of exceeding a certain level of load imbalance exponentially decreases with increasing number of splits. This scheme works well in practice, and we converge in 2–4 iterations even for heavily skewed distributions such as Zipf. Interestingly, the number of final samples we use is close to what DeWitt et al. [24] found to be optimal in their experiments. While previous work on histogram splitting [55] do not consider payloads, we find that different design choices are required for medium to large payloads. We avoid the initial sorting of local data required by histogram splitting, and instead sort at the receiver nodes. This can significantly reduce the delay introduced by local sorting before transferring tuples to the destination, and increase overlap of computation with communication—since the receiver nodes can sort incoming keys while payloads are being transferred.

Recent trends in computer architecture involve a dramatic increase in per-node computation due to increasing core count as well as widening of SIMD units. There have been recent efforts in exploiting the increased computational resources to speed up single-node sorting [17, 51]. While sorting on distributed shared-nothing architectures have been widely studied previously [29, 32, 9, 54, 23, 24, 12], we revisit this by incorporating state-of-the-art single-node sorting algorithms to improve per-node efficiency.

8. CONCLUSION

In this paper, we presented CloudRAMSort, a fast and efficient system for large-scale in-memory distributed sorting on shared-nothing clusters. CloudRAMSort performs multi-node optimizations by carefully overlapping computation with inter-node communication. CloudRAMSort maximizes per-node efficiency by exploiting modern architectural features such as multiple cores and SIMD units. The system sorts 1 TB of data in 4.6 seconds on a 256-node Intel Xeon X5680 Endeavor system. CloudRAMSort also performs well on heavily skewed input distributions. We also provided a detailed analytical model that accurately projects the performance of CloudRAMSort with varying tuple sizes and interconnect bandwidth.

With architectural trends of increasing number of cores, bandwidth, SIMD width, cache-sizes, and interconnect bandwidth, we believe CloudRAMSort would be the system of choice for distributed sorting of large-scale in-memory data of current and future systems.

9. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org.

[2] bonnie++: a program to test hard drive performance. http://www.coker.com.au/bonnie++.

[3] Data Generator For Sorting Benchmarks. www.ordinal.com/gensort.html.

[4] Hadoop Cluster Setup. http://hadoop.apache.org/common/docs/current/cluster_setup.html.

[5] OpenFabrics Alliance. https://www.openfabrics.org/index.php.

[6] SAS In-Memory Analytics. http://www.sas.com/software/high-performance-computing/in-memory-analytics/.

[7] Sort Benchmark Home Page. http://sortbenchmark.org.

[8] The Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/.

[9] B. Abali, F. Özgüner, and A. Bataineh. Balanced Parallel Sort on Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):572–581, 1993.

[10] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy Proportional Datacenter Networks. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[11] E. Anderson and J. Tucek. Efficiency Matters! *ACM SIGOPS Operating Systems Review*, 44(1), 2010.

[12] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD*, pages 243–254.

[13] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 3–16, 1991.

[14] L. Breslau, P. Cue, P. Cao, L. Fan, et al. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[15] R. E. Bryant. Data-Intensive Supercomputing: The case for DISC. Technical Report CMU-CS-07-128, 2007.

[16] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO*, pages 385–395, 2010.

[17] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[18] H. Cho, P. Kapur, and K. C. Saraswat. Power Comparison Between High-Speed Electrical and Optical Interconnects for Interchip Communication. *Journal of Lightwave Technology*, 22(9), 2004.

[19] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[20] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011.

[21] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[23] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[24] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting. In *International Conference on Parallel and Distributed Information Systems*, pages 280–291.

[25] M. Glick. Optical interconnects in next generation data centers: An end to end view. In *Hot Interconnects*, pages 178–181, 2008.

[26] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *SIGMOD Conference*, pages 1061–1066, 2010.

[27] J. Gray and F. Putzolu. The 5 minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time. In *SIGMOD*, pages 395–398, 1987.

[28] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.

[29] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *International Computer Software and Applications Conference*, 1983.

[30] Intel Research. Light Peak: Overview. Intel White Paper, 2010.

[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[32] B. R. Iyer, G. R. Ricard, and P. J. Varman. Percentile Finding Algorithm for Multiple Sorted Runs. In *VLDB*, pages 135–144, 1989.

[33] L. V. Kalé and S. Krishnan. A Comparison Based Parallel Sorting Algorithm. In *International Conference on Paralllel Processing (ICPP)*, pages 196–200, 1993.

[34] C. Kim, J. Chhugani, N. Satish, et al. FAST: Fast Architecture Sensitive Tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[35] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[36] J. Kim, W. J. Dally, and D. Abts. Flattened Butterfly : A Cost-Efficient Topology for High-Radix Networks. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[37] J. Kim, W. J. Dally, B. Towles, and A. Gupta. Microarchitecture of a High-Radix Router. In *International Symposium on Computer Architecture (ISCA)*, 2005.

[38] M. H. Kryder and C. S. Kim. After Hard Drives—What Comes Next? *IEEE Transactions on Magnetics*, 45(10), 2009.

[39] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Phase change memory architecture and the quest for scalability. *Commun. ACM*, 53(7):99–106, 2010.

[40] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10), 1985.

[41] D. L. Lewis and H.-H. S. Lee. Architectural evaluation of 3d stacked rram caches. In *3DIC*, pages 1–4, 2009.

[42] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *International Symposium on Computer Architecture (ISCA)*, pages 315–326, 2008.

[43] H. Liu, C. F. Lam, and C. Johnson. Scaling optical interconnects in datacenter networks: Opportunities and challenges for wdm. In *Hot Interconnects*, 2010.

[44] M. LLC. In-Memory Database Systems: Myths and Facts. 2010.

[45] O. O'Malley and A. C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. `http://sortbenchmark.org/Yahoo2009.pdf`, 2009.

[46] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanana, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4), 2010.

[47] D. A. Patterson. Latency Lags Bandwidth. *Communications of the ACM*, 47(10), 2004.

[48] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, pages 165–178, 2009.

[49] P. Ranganathan. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *IEEE Computer*, 44(1):39–48, 2011.

[50] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *USENIX conference on Networked System Design and Implementation*, 2011.

[51] N. Satish, C. Kim, J. Chhugani, et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362. ACM, 2010.

[52] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *SIGGRAPH*, 27(3), 2008.

[53] S. Seshadri and J. F. Naughton. Sampling Issues in Parallel Database Systems. In *Advanced in Database Technology - EDBT*.

[54] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[55] E. Solomonik and L. V. Kalé. Highly scalable parallel sorting. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12.

[56] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[57] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.