MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds

Yilong Li, *Stanford University;* Seo Jin Park, *MIT CSAIL;*
John Ousterhout, *Stanford University*

https://www.usenix.org/conference/nsdi21/presentation/li-yilong

This paper is included in the
Proceedings of the 18th USENIX Symposium on
Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the
18th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds

Yilong Li* Stanford University
Seo Jin Park* MIT CSAIL
John Ousterhout Stanford University

Abstract

Today's datacenter applications couple scale and time: applications that harness large numbers of servers also execute for long periods of time (seconds or more). This paper explores the possibility of *flash bursts*: applications that use a large number of servers but for very short time intervals (as little as one millisecond). In order to learn more about the feasibility of flash bursts, we developed two new benchmarks, MilliSort and MilliQuery. MilliSort is a sorting application and MilliQuery implements three SQL queries. The goal for both applications was to process as many records as possible in one millisecond, given unlimited resources in a datacenter. The short time scale required a new distributed sorting algorithm for MilliSort that uses a hierarchical form of partitioning. Both applications depended on fast group communication primitives such as shuffle and all-gather. Our implementation of MilliSort can sort 0.84 million items in one millisecond using 120 servers on an HPC cluster; MilliQuery can process .03–48 million items in one millisecond using 60-280 servers, depending on the query. The number of items that each application can process grows quadratically with the time budget. The primary obstacle to scalability is per-message costs, which appear in the form of inefficient shuffles and coordination overhead.

1 Introduction

One of the benefits of datacenter computing is the ability to run large-scale applications that harness hundreds or thousands of machines working together on a common task. Using frameworks such as MapReduce [11] and Spark [67], developers can easily create applications in a variety of areas such as large-scale data analytics.

Until recently, large-scale applications have executed for relatively long periods of time: seconds or minutes. This was necessary in order to amortize the high cost of allocating and coordinating a collection of servers. Similarly, frameworks such as MapReduce and Spark have traditionally operated on very large blocks of data, in order to amortize high network latencies. As a result, they cannot be applied to real-time tasks. Instead, real-time queries must return precomputed results, such as those produced in overnight batch runs. This means that the queries must be carefully planned in advance; ad-hoc queries cannot easily be supported.

Streaming frameworks such as Flink [15] or Spark Streaming [66] operate on incoming data in real time, but they do this by incorporating new data into queries or transformations that are planned long in advance. To support real-time queries, the scale of computation triggered by each event is quite limited.

In recent years, new serverless platforms such as AWS Lambda [5], Azure Cloud Functions [45], and Google Cloud Functions [20] have made it possible to run short-lived tasks (as small as a few hundred milliseconds) in datacenters. However, the unit of execution in these environments is an individual function call. It is difficult to harness multiple machines in a single serverless computation; for example, direct communication between lambdas is not officially supported, and existing workarounds [17, 3] have high latency and low bandwidth.

This paper explores the possibility of extending serverless computing in two ways: first, by further reducing the timescale; and second, by reintroducing scale, so that large numbers of servers can work together. We use the term *flash burst* to describe a computation that has a short lifetime yet harnesses large numbers of servers. Flash bursts offer the potential of analyzing large amounts of data in real time. This could enable the creation of new applications that execute customized queries on large datasets in real time, without the need to predict queries hours or days ahead of time.

Rather than making incremental improvements on existing systems, our goal is to push the notion of flash bursts to the extreme, in order to understand the limits of this style of computation. In particular, we set out to answer the following questions:

- What is the smallest possible timescale at which meaningful flash bursts can operate?
- What is the largest number of servers that can be harnessed at such a timescale?
- What aspects of current systems limit the duration and scale of flash bursts?

We hypothesized that timescales as small as 1 ms might be possible, so we set that as our initial goal.

Making flash bursts practical will require advances in many different areas. This paper focuses on one aspect of the problem: whether the core algorithms likely to be used in flash bursts can operate efficiently at very small timescales. We do not address issues such as the time required to load applications and data, or how to achieve high resource utilization in the face of short-lived computations; we leave these for future work.

We implemented two focused applications that capture patterns of computation and communication we expect to be common in flash bursts. The first application is MilliSort: given unlimited resources in a datacenter, what is

*Co-first authors, listed alphabetically.

- Sort 40,000 10-byte keys using 8 cores [7].
- Copy 5 Mbytes of data from memory to memory sequentially.
- Send or receive 5 Mbytes of data with a 40 Gbps NIC.
- Invoke 300 back-to-back remote procedure calls on one core, using kernel bypass [50, 52, 32].
- Send or receive 2–5k small messages on one core [50, 32, 48].
- Take 10,000 back-to-back L3 cache misses on one core.

Figure 1: Examples of tasks that can be completed in one millisecond on modern hardware.

the largest number of small records that can be sorted in one millisecond? The second application is MilliQuery, which consists of three representative SQL queries from the tutorials for Google BigQuery. The queries range from a simple scan-filter-aggregate query to a distributed join requiring multiple shuffles. As with MilliSort, our goal was to understand how much data can be analyzed, and how many servers can be harnessed, in timescales around 1 ms.

The process of developing and measuring these applications has yielded interesting results in three categories:

- Measurements: MilliSort and MilliQuery demonstrate that large-scale data analytics can operate efficiently even at timescales of 1–10 ms. MilliSort can sort 0.84 million small records in one millisecond using 120 servers running on 30 machines. The MilliQuery benchmarks process .03–48 million records in one millisecond using 60–280 servers, depending on the query.
- Observations: the development of MilliSort and MilliQuery yielded several interesting insights about flash bursts, which are summarized in Section 7. For example, we found that the amount of data that a flash burst can handle grows quadratically with the time budget (both the amount of data per server and the number of servers grow at least linearly with the time budget). Some of the most important and common limits to scalability are shuffle cost and coordination overhead (both can be attributed to per-message overheads).
- Algorithms: while implementing MilliSort we developed a new low-latency algorithm for partitioning the keys, which uses a hierarchical series of distributed sorts. We also developed a novel splitter selection algorithm that improves the balance among data partitions. Overall, MilliSort runs with efficiency comparable to other systems that operate at much larger timescales.

2 Background

One millisecond is not very long. Figure 1 lists a few things that can be done in one millisecond on today's machines; these create fundamental limitations for flash bursts.

2.1 Limited data per server

One of the most important limitations evident from Figure 1 is that each server can only manipulate a small amount of data (on the order of a few Mbytes). For example, a single server core can only access a few megabytes of data in one millisecond, and network bandwidth allows only a few megabytes to be transmitted in one millisecond.

Given the large number of servers and small data per server, data must stay evenly distributed throughout a flash burst. If even a small fraction of data accumulates on a single server, the network link into that server will become a bottleneck.

2.2 Coordination cost

Given that each server can only access a small amount of data, the overall scale of a flash burst will be limited by the number of servers that can be harnessed. But, the small time scale makes it difficult to coordinate very many servers; at some scale one millisecond isn't even enough time to notify all the servers to start working. Thus, coordination overheads play a fundamental role in flash bursts, since they limit the scale. "Coordination" includes such activities as engaging all of the servers, determining work assignments for each server, and sequencing the phases of the algorithm. Existing large-scale applications such as Spark store much larger amounts of data per server and also run for longer time periods; this combination makes coordination overheads less important.

2.3 Multiple communication costs

For many existing large-scale applications, the only communication cost that matters is network bandwidth. Systems such as MapReduce [11] and Spark [67] are explicitly designed to exploit bandwidth and hide communication latency. However, for flash bursts three different costs may become important. In addition to *bandwidth*, which matters when sending large blocks of data, and *latency*, which matters when sending small chunks of data, a third cost plays an important role in flash bursts: *per-message overhead* (the CPU time required to send and receive short messages). Per-message overhead comes into play when a server has a collection of small requests that can be sent to other servers concurrently; it limits how quickly a series of messages can be issued. Per-message overheads are particularly important in flash bursts because they dominate the cost of group communication primitives (discussed below), which in turn dominate the cost of coordination.

2.4 Group communication

If a collection of servers is to cooperate closely, the servers will probably need to exchange data frequently and in small chunks. However, in a flash burst, where there are hundreds or thousands of servers operating on a very small time scale, it is not practical for each server to communicate directly with all of the other servers. For example, if a server broadcasts data to 1000 other servers by sending a separate small message to each of them, the broadcast will consume a substantial fraction of a millisecond, due to per-message overheads.

Thus, *group communication* plays an important role in flash bursts. In group communication, many or all of the servers in a cluster transmit data concurrently to carry out a cluster-wide goal. The HPC community has identified and implemented a

variety of useful group communication mechanisms [60], of which four play a role in MilliSort and MilliQuery:

Broadcast: data that is initially present on a single server must be distributed to every server in the group.

Gather: the reverse of broadcast. A single server must collect distinct data from each of the servers in the group.

All-gather: initially each server in the group stores a distinct data item; the all-gather operation must arrange for every server in the group to receive a copy of all the items.

Shuffle: each server initially stores a separate data item for every other server in the group; the shuffle must transmit all the items to their intended targets.

Group communication provides two benefits. First, it harnesses multiple servers operating concurrently to speed up the communication; for example, several servers can be used to complete a broadcast more quickly by distributing messages via a tree structure. Second, it can sometimes replace many small messages with a few larger messages; this reduces the impact of per-message overheads. For example, a hypercube implementation [62] of all-gather requires only $MlogM$ messages for M servers, vs. M^2 messages if each server communicates independently with every other server.

3 Applications

One of the challenges in exploring flash bursts is that there are no flash burst applications available today (unsurprising, given that there is no infrastructure capable of supporting them). Fortunately, there appears to be a small set of patterns of computation and communication that are used commonly across a variety of large-scale applications and account for much of their performance [4, 19, 33]. These are referred to as *dwarfs* by Asanovic et al. [4] and others. For example, matrix operations, sorting, and statistics computations are dwarfs for big-data and AI workloads. Rather than guessing at full applications, we have implemented two small applications that capture several dwarfs. Although the behavior of these dwarfs will not be a perfect predictor of any real application, this approach has two advantages. First, lessons learned from the dwarfs are likely to apply to many real applications. Second, it is easier to understand the properties of the dwarfs if we study them in isolation, rather than as part of a full application with many interacting parts.

Our first application is a sorting benchmark called MilliSort. Sorting has been used to evaluate system performance for many decades, originating with a challenge proposed by Jim Gray and others in 1985 [2]. Sorting plays an important role in many distributed computations. For example, sorting can be used as a data preprocessing step to support efficient range queries, to improve data locality for graph partitioning [44], and to perform load balancing [44, 11, 23]. Sorting is also very challenging because it requires intensive and unpredictable communication (any record can potentially end up on any server). We expected this to create challenges both for the algorithm and the underlying infrastructure.

```
/* MilliQuery Q1: count article views on Wikipedia by language */
SELECT language, SUM(views)
FROM `bigquery-samples.wikipedia_benchmark.Wiki1B`
GROUP BY language

/* MilliQuery Q2: top 10 IPs by the number of edits to Wikipedia */
SELECT contributor_ip AS ip, COUNT(*) AS count
FROM `publicdata.samples.wikipedia`
GROUP BY ip ORDER BY count DESC LIMIT 10

/* MilliQuery Q3: complex data analytics on GitHub data */
WITH
 repo_authors AS (   -- Build the intermediate author table
  SELECT repo_name, author.name AS author
  FROM `bigquery-public-data.github_repos.commits`,
   UNNEST(repo_name) AS repo_name
  GROUP BY repo_name, author),
 repo_languages AS (  -- Build the intermediate language table
  SELECT lang.name AS lang, lang.bytes AS lang_bytes, repo_name
  FROM `bigquery-public-data.github_repos.languages`,
   UNNEST(language) AS lang)
SELECT lang, author, SUM(lang_bytes) AS total_bytes
FROM (repo_languages JOIN repo_authors USING repo_name)
GROUP BY lang, author ORDER BY total_bytes DESC LIMIT 100
```

Listing 1: The three SQL queries used in the MilliQuery benchmark

For MilliSort, the goal is to sort as many small records as possible in intervals around one millisecond, using any number of servers in a datacenter. Each record contains 100 bytes, consisting of a 10-byte key and a 90-byte value. Before starting the benchmark, the MilliSort application is preloaded and the unsorted records are distributed evenly among the available machines in DRAM. The data on each server is structured with all of the keys in a single block of memory and all the values in another block, in corresponding order. Upon completion, the data must be redistributed across the same servers, sorted such that one server contains the records with the smallest keys, and so on. At the end of the sort, the data on each server is structured in two blocks of memory, one containing the keys in sorted order and another containing the values in the same order as their keys. The challenge does not require that the result data be distributed evenly across the servers, but this turns out to be essential for good performance.

Our second "application" is a collection of three SQL queries from Google's BigQuery documentation[63, 38]; we refer to these queries collectively as MilliQuery. We expect that many flash burst applications will perform data analytics to provide real-time results from ad hoc queries; the goal for MilliQuery is to capture dwarfs that will be common in these applications.

We chose three queries with different levels of complexity, in order to span a range of SQL behaviors (see Listing 1).

Q1: counts the number of article views on Wikipedia by language (there are at most a few hundred different languages).

Q2: finds the top 10 IP addresses by the number of edits they made to Wikipedia articles.

	Coordination	Shuffle(s)	Dwarf(s)
MilliSort	Heavy	≥2	Sort
MilliQuery Q1	None	0	Aggregate
MilliQuery Q2	Light	1	Repartition,Aggregate
MilliQuery Q3	Light	3	Repartition,Join

Table 1: A comparison of the applications used for studying flash bursts.

Q3: for each combination of author and programming language, sum all of the bytes in that language in any repository for which the author was a contributor; returns the top 100 author-language pairs.

In each case, the goal is to process as much data as possible within 1 ms using unlimited datacenter resources, assuming that the application is pre-loaded and data is initially distributed uniformly across the nodes.

Table 1 compares these applications in terms of the complexity of coordination (how difficult it is to divide the work among the participating nodes and coordinate their behaviors), the number of shuffle steps required, and the dwarfs captured. Together, MilliSort and MilliQuery cover a significant range of interesting behaviors.

Our goal for MilliSort and MilliQuery was to push the idea of flash bursts to the extreme. We don't know what burst sizes will prove useful to applications, so we set out to characterize the range that is practical: what is the smallest time interval and what is the largest number of servers that can be harnessed efficiently? We also hoped to learn what are the technical factors that limit flash bursts. We chose a one millisecond time limit because it seemed like an extreme goal: at the outset, we were unsure whether it would be possible to do anything useful in such a short interval.

4 The MilliSort algorithm

Although distributed sorting is not new, designing a sorting algorithm to operate at a timescale of one millisecond introduces new challenges due to the high cost of coordination. This section describes MilliSort's algorithm in detail and presents a novel hierarchical approach to data partitioning that enables efficient coordination even at very small timescales.

Most distributed sorting algorithms use a partitioning approach, and MilliSort follows this tradition. First, the data is partitioned by deciding which key ranges should end up on each server; then the records are shuffled between servers to implement the chosen partitions. This approach optimizes the use of network bandwidth by transmitting each record only once, during the shuffle phase. The partitioning approach makes sense because it optimizes the use of network bandwidth, which has traditionally been the scarcest resource in distributed sorting. Alternative approaches, such as those that use multi-stage merge sorts, require data to be transmitted over the network multiple times, so they have proven slower than the partitioning approach.

More precisely, MilliSort implements a variant of distributed bucket sort, with one bucket for each server. It consists of four phases:

1. Local sort: each server sorts its initial data.
2. Partition bucket boundaries: the servers collectively determine the key ranges that will end up on each server after sorting.
3. Shuffle: each server transmits its keys and values to the appropriate targets.
4. Local rearrangement: the data arriving on each server during the shuffle phase must be rearranged into two totally sorted arrays, one for keys and one for values.

The sections below discuss each of these phases in more detail. We start with the partitioning phase, since it is the most complex phase and also the most interesting phase from an algorithmic standpoint.

4.1 Histogram sort

One of the most widely used partition-based sorting algorithms is histogram sort, which computes the final key ranges by iteratively refining an existing partition until the keys are evenly distributed on the servers. A typical workflow of histogram sort is as follows. In the beginning, a central server picks $M-1$ splitters, which divide the key space into M buckets, and broadcasts them to the other servers. Then, each server computes a local histogram of its keys in the buckets and sends it back to the central server. Finally, the central server computes a global histogram by summing up the local histograms and adjusts the splitters to reduce the imbalance. The process of histogramming and refinement is repeated until an even partition is achieved. In addition to the one mentioned above, there are other variations of histogram sort which use more splitters for histogramming or increase the number of splitters as they progress.

However, histogram sort is undesirable for MilliSort since it requires many iterations to converge, and each iteration incurs significant message delays. To avoid overloading the central server, histogram sort often uses group communication to broadcast splitters and reduce local histograms in a tree structure. As a result, each iteration incurs $2log(M)$ back-to-back message delays. In our environment, the combined cost of broadcast and gather is at least 50 µs for 100 servers. With just 10 iterations, message delays alone will take away half of our 1 ms time budget. Finally, the actual cost of partitioning will be even higher due to other overheads; the reported partitioning times of some recent histogram sort implementations easily exceed 50 ms for 512 HPC nodes [35, 21].

4.2 Sample sort partitioning

MilliSort takes a different approach to partitioning, selecting a larger number of initial keys so that it can estimate the distribution in a single iteration. MilliSort's partitioning algorithm is based on sample sort with regular sampling [58, 40]; the basic idea is to select many keys from the starting data, use these to estimate the distribution of keys, and pick partition boundaries based on the estimated distribution. Figure 2 shows the basic idea. After sorting its local data, each server samples its keys at equally-spaced intervals within the sorted

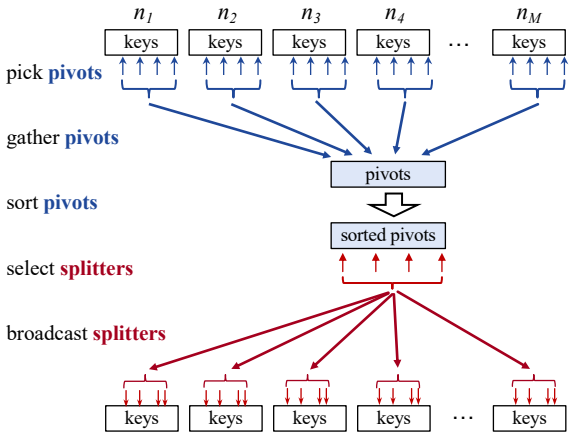

Figure 2: A basic sample sort partition mechanism; $n_1 - n_M$ are the MilliSort nodes.

records; we refer to these samples as *pivots*. The pivots of all servers are collected and sorted (more details on this below). Finally *splitter* values are chosen from the sorted pivots. If there are M machines participating in the sort, $M-1$ splitters are chosen, which divide the sorted pivots into M equal-size groups. The splitters determine how records are divided between servers during the shuffle phase: server i will eventually hold all records with keys greater than or equal to the ith splitter and less than the i+1th splitter.

Because of the sampling used by this approach, there is no guarantee that each server will end up with exactly the same number of records at the end of the sort. If there are N total records divided among M machines and each machine chooses sM pivots, then, in the worst case, a server may end up with as many as $(1+1/s)(N/M)$ records (N/M is the ideal number in a perfect partition) [58, 40]. For MilliSort, we use $s = 1$ (each machine chooses M pivots), so the final partition sizes are guaranteed to be within a factor of 2 of the ideal size. In practice the distribution of records is considerably more uniform than suggested by the worst-case formula above.

With this approach, the total number of pivots to sort is sM^2. This means that as the number of machines increases, partitioning will take more and more time, even if all of the machines share the work. Given a limited amount of time for the sort, partitioning cost will limit the number of machines that can be harnessed.

4.3 Recursive partitioning

One way to perform the partitioning is to gather all of the pivots on a single coordinator server, sort them locally on that server, then broadcast the splitters back to all of the servers. However, this approach is too inefficient for MilliSort. If 300 machines participate in the sort, there will be 90,000 pivots; as shown in Figure 1, a single server can only sort about 40,000 keys in one millisecond, so the sorting alone would take more than 2 ms. The overhead of receiving all the pivots on a single server is also problematic. Thus, millisecond-scale sorting requires partitioning to be performed in a distributed fashion.

MilliSort uses a recursive approach to partitioning: the pivots are sorted in a distributed fashion using a smaller instance of MilliSort, as shown in Figure 3. A subset of the machines, called *pivot sorters*, sort the pivots and select splitters; each of the other servers is assigned to one of the pivot sorters. To begin the sort, each pivot sorter gathers the pivots from its assignees. The pivots arriving from each source machine are already sorted, so the pivot sorter can use merge sort on the arriving data to produce a sorted list of all the pivots for which it has responsibility. Then each pivot sorter samples its pivots to choose a smaller number of *level 2 pivots*; the level 2 pivots are passed to a coordinator, which sorts them and produces a set of *level 2 splitters*. The coordinator broadcasts the level 2 splitters back to the pivot sorters, which then perform a shuffle to redistribute the pivots among the pivot sorters in sorted order.

At this point the pivots have been sorted and splitters must be chosen (i.e., we must select every sM^{th} pivot across all of the pivot sorters). We would like for each pivot sorter to independently select splitters from its pivots, but in order to do this, the pivot sorter must know its rank (i.e., how many pivots stored on other servers are smaller than the pivots that it stores). The rank is not immediately obvious because pivots are not distributed uniformly across the pivot sorters. The solution is to distribute rank information during the shuffle phase of the pivot sort. When a pivot sorter sends a group of pivots to another pivot sorter during the shuffle, it includes the *local rank* of that group (i.e., the number of pre-shuffle pivots from that pivot sorter that are smaller than those in the group). Each pivot sorter can determine its rank by summing the local ranks in all of the shuffle messages it receives. Once a pivot sorter knows its rank, it can identify the splitters that it stores.

Finally, once the splitters have been determined, they must be disseminated to all of the machines participating in the sort. One approach would be for each of the pivot sorters to broadcast its splitters to all of the M machines; however, this would have a high cost because of the large number of messages that would result. Instead, MilliSort uses a two-step approach to distribute the splitters. In the first step, an all-gather operation is used to exchange the splitters among the pivot sorters, so that each pivot sorter has all $M-1$ splitters. Then each pivot sorter broadcasts the complete set of pivots to all of the machines assigned to it.

If the number of servers is very large, the 2-level approach described above will still take too long. If that is the case, additional levels may be used in the partition. For example, in a 3-level approach the level 2 pivots will not be sorted on a single coordinator; instead, they will be collected by a smaller number of second-level pivot sorters, which will then select a set of level 3 pivots. The level 3 pivots will be collected and sorted on a single coordinator, resulting in level 3 splitters, which are used to shuffle the level 2 pivots. The approach can be extended to an arbitrary number of levels, though our experiments suggest that 2 or 3 levels is appropriate for MilliSort.

We use r to refer to the reduction factor at each level of recursive sorting. For M total machines, there will be M/r

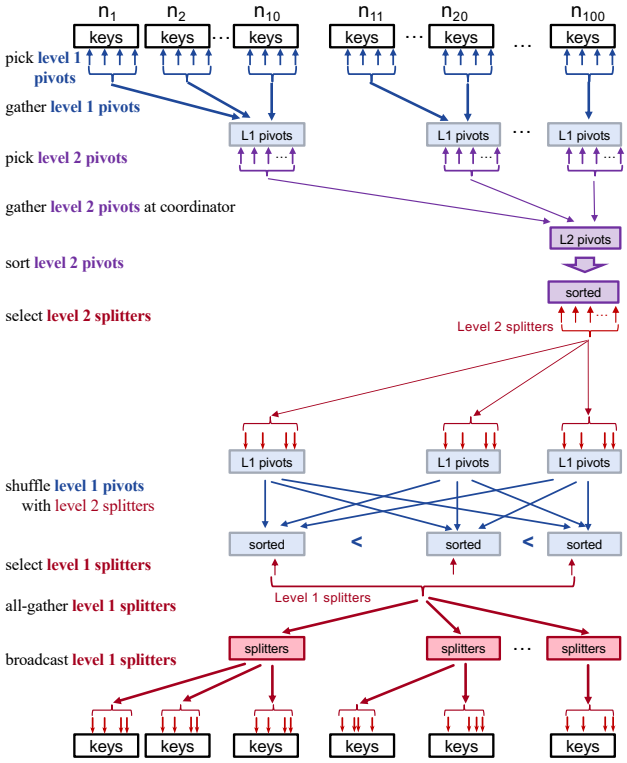

Figure 3: A two-level partitioning example with $M = 100$, $s = 1, r = 10$.

pivot sorters, M/r^2 second-level pivot sorters, and so on. For the MilliSort implementation we used a reduction factor of 10.

4.4 Improved splitter selection

The algorithm described above for choosing splitters from the sorted pivots results in uneven key distribution, where the first server is likely to be assigned 50% more keys than the average, and the last server is likely to be assigned 50% fewer keys than the average. We developed an improved splitter selection algorithm that eliminates this imbalance; because of space limitations, we describe that algorithm in Appendix A.1.

4.5 Local sort

Now that the partitioning mechanism has been described, the next few subsections discuss the remaining phases of MilliSort. These phases are more straightforward than the partitioning phase, though their performance is still important.

MilliSort relies on prior work for the local sort. The problem of sorting on a single compute server is well-studied, with many solutions that can take advantage of multiple cores. In our experiments we use the In-place Parallel Super Scalar Samplesort (IPS^{4}o) algorithm [7] as a reasonable representation of a multi-core comparison-based sorting algorithm.

Once the keys have been sorted, the values must be rearranged to match the order of the keys. This can be overlapped with the partitioning phase and the key shuffle; the rearranged values are not needed until the value shuffle.

4.6 Shuffle

Once the splitters have been selected and distributed to all of the machines, MilliSort uses an all-to-all shuffle to transmit each key and value to the appropriate server. The keys and values on each server were already sorted during the local sort phase. Each server uses the splitters to determine the range of keys to send to each other server. It then sends a message to that server containing the appropriate range of keys, followed by the data associated with those keys.

4.7 Local rearrangement

Within each message that a server receives during the shuffle phase, the keys and values are in sorted order. The server must then combine these chunks into two sorted arrays, one containing all the keys and other containing all the values. To do this, each server first performs a merge sort on the arrays of keys; then it rearranges the values to match the order of the keys. Each key contains its index in the incoming shuffle message (which is the same as the index of the value in the message). Once the keys have been sorted, a sequential scan is made over the key array; the index stored with each key is used to find the corresponding value and the value is then stored at the next sequential location in the result array. This two-step approach is faster than a one-step approach that merges both keys and values together, because it copies the (larger) values only once.

5 Implementation

5.1 Group communication

We created a C++ library that implements the group communication primitives described in Section 2.4; both MilliSort and MilliQuery make extensive use of this library. The group communication primitives are implemented using the network transport infrastructure from RAMCloud [50]. RAMCloud's transports use kernel bypass with either DPDK [10] or the Infiniband verbs interface to provide low latency (5 µs round-trips) and high throughput (up to 25 Gbps). However, RAMCloud requires all network communication to pass through a single dispatcher thread, which limits message throughput to about 1.6 million messages per second. The group communication primitives contain 1500 lines of C++ code, not counting code in RAMCloud.

Broadcast, gather, and all-gather were implemented using well-known approaches [62, 65]. For broadcast, MilliSort uses a k-nomial tree, with the topology optimized based on precise knowledge of message latency and per-message cost. For gather, MilliSort uses a k-nomial tree with $k = 6$, in order to utilize as much network bandwidth as possible at each step. For all-gather, MilliSort uses a hypercube approach, extended to handle group sizes that are not even powers of two [56].

Shuffle is the most important of the group communication primitives: it accounts for half or more of the end-to-end time in the highest performing MilliSort configurations and it is also used in two of the three MilliQuery queries. However, achieving high-performance for shuffle is challenging. Ideally, shuffles should utilize the full bandwidth of the network, with each host simultaneously sending and receiving at the speed of its uplink. However, achieving this

goal is difficult. One problem is small messages, which are more likely to occur in flash bursts than other applications (see below). Another problem is that shuffle requires full bisection bandwidth in the underlying network. Fortunately, our test cluster has full bisection bandwidth. Unfortunately, we were unable to harness all of the available bandwidth.

Achieving full bisection bandwidth requires a near-perfect bipartite matching, where at any given time each source transmits at full bandwidth to a different target. This is difficult to achieve, for two reasons. First, two servers might attempt to transmit simultaneously to the same target. When this occurs, bandwidth is wasted on the senders, since the target can only receive from one of them at a time; in addition, some other server will not be receiving anything at all, which wastes its incoming bandwidth. Second, achieving full bisection bandwidth requires perfect load-balancing across the network fabric. Unfortunately, our test cluster does not support packet-level load-balancing. Instead, it uses flow-consistent hashing, where all packets from a particular source to a particular destination are routed over a single (randomly chosen) path through the fabric. Even if two sources send to different targets, their routes might traverse a common intermediate link, resulting in bandwidth underutilization. With large clusters, routing conflicts are virtually guaranteed.

After implementing shuffles in the most obvious way and observing poor performance, we attempted a lock-step approach to ensure a bipartite matching. In the lock-step approach, in step i each server n transmits to server $(n+i)$ mod M, and the start of step $i+1$ is delayed until step i has completed. This mostly eliminated the problem where two senders transmit to the same target, but it works best when all messages are large and the same size. If messages are small, or if messages have different sizes, the act of maintaining lock-step wastes most of the network bandwidth (e.g. each step must wait for the longest message in the preceding step to complete). Furthermore, lock-step is still vulnerable to conflicts in network routes. We were unable to achieve a satisfactory level of performance with this approach.

We then switched to a nearly-opposite approach, implementing shuffles in a "high-entropy" fashion that is granular, concurrent, and random. The first step is to ensure that messages sent during shuffles are relatively short (at most 40 KB in the current implementation). In many cases, the messages are inherently short; if the data from one server to another exceeds a threshold size, it is divided into multiple short chunks, which are sent as separate messages. Each host sends multiple messages concurrently; the targets are chosen randomly, but a given source will have at most one message outstanding to a given target at a time. With this approach there will still be conflicts but they will not stall senders; conflicts simply result in packet queueing in the network. Since each sender has multiple outstanding messages, it is likely that there will be incoming data for each server at all times. Since messages are short, buffer overflows in the network are unlikely and a sender doesn't waste much time on a busy receiver before directing its bandwidth elsewhere. Stalls will occur only at the end of the shuffle, when a sender is waiting for its last few messages to complete. The high-entropy approach resulted in much better performance than the alternatives we tried before it.

Shuffles are more challenging in a flash burst than in more traditional environments that operate at large timescales. A flash burst scales by increasing the number of servers, with less data on each server, in order to complete more quickly. Less data on each server means the size of each shuffle message will drop; more servers means that each server's data is split among more shuffle messages, which also results in less data per message. The result is a rapid drop in shuffle message size as a flash burst scales. This leads to low network bandwidth utilization and high per-message overheads.

One possible solution is to use a two-level shuffle to reduce the number of messages sent and received by each server from $M-1$ to $2\cdot(\sqrt{M}-1)$. Two-level shuffle arranges all servers into a virtual mesh and proceeds in two rounds: each server first exchanges data with other servers in the same row, and then in the same column. However, this approach doubles the network bandwidth consumed, since most data must be transmitted twice. For our experiments the increased bandwidth usage of a two-level shuffle was more problematic then per-message overheads for a single-layer shuffle; we did not find any situations where multi-level shuffles are advantageous.

5.2 MilliSort

The MilliSort implementation is fully decentralized: each server operates independently in an event-driven style, with no central coordinator (central coordination is intolerable for flash bursts, both because of the latency it adds and also because central coordination often involves lock-step operation at the end of each stage, which suffers from stragglers). While the order of stages executed in a server is well defined, the timing is affected by remote procedure calls (RPCs) from other servers. At various points, the progress of the server will stall until certain pieces of data have arrived. As one example, a pivot sorter cannot select level 2 pivots until it has received pivots from all of the servers in its group.

Different servers must have different behaviors during the sort. For example, only a subset of the servers will act as pivot sorters. Each server has an identifier ranging from 0 to $M-1$, which determines the various roles it will serve during the sort. For example, servers with identifiers that are 0 mod r serve as pivot sorters and they receive pivots from servers with the following $r-1$ identifiers. At the start of the sort each server knows its identifier, the value of M, and the addresses of the other servers.

Achieving the highest overall performance for MilliSort requires careful optimization of both communication and computation. Section 5.1 has already discussed the challenges associated with shuffles. Using cores efficiently is another challenge, because the number of threads changes rapidly

over the life of the sort. Thus, MilliSort uses Arachne [54] for efficient user-level thread and core management. For example, Arachne allowed us to quickly spawn a group of threads to parallelize a task, such as local sort, and place them precisely on all available cores. In addition, MilliSort creates a new thread for each incoming RPC and leaves it to Arachne to schedule those threads on cores that are less busy.

5.3 MilliQuery

We created a special-purpose implementation of each of the three MilliQuery queries, largely based on the query plans generated by BigQuery. These queries were much easier to implement than MilliSort, particularly given the availability of the group communication library. The total implementation time was only a few days, and the three queries contain 250, 300, and 800 lines of C++ code, respectively.

The implementation of Q1 follows a simple scan-aggregate pattern: each server scans its data independently to count the views by language, then the local results are gathered back to one server, using a k-nomial tree and combining the statistics at each node. Since the number of distinct languages is only a few hundreds, all messages in the gather phase are small.

Similar to Q1, Q2 can also be implemented as local scan followed by a gather. But, the number of distinct IP addresses is quite large, so the gather phase would consume too much network bandwidth with this naive approach. Thus, before the gather phase, a shuffle is used to collect all the counts for each address in one place, using a hash partition. Then each node in the gather tree collects local results from all its children, but only needs to send the top ten IP addresses to its parent.

Q3 is considerably more complex and requires a total of three shuffles. First, two shuffles are used to materialize two intermediate tables, distributing the records for each table using a hash partition with the `repo_name` field. Then, the two tables are joined locally using the same key, and a third shuffle redistributes the joined records by hash partitioning on `(lang, author)`. Finally, top-100 statistics are computed locally and aggregated as in Q2.

For simplicity, we didn't implement additional mechanisms to handle hot keys in hash partitioning for Q2 or Q3; however, it's possible to use MilliSort's recursive partitioning scheme to create a more balanced range partitioning, at the expense of higher partitioning cost.

5.4 Communication infrastructure

We chose to build our MilliSort and MilliQuery prototypes atop RAMCloud [50] mainly to reuse its flexible network infrastructure. However, this choice comes with the cost of limited message throughput for two reasons:

- The single dispatch thread in RAMCloud has relatively high per-message costs (all messages must pass through the dispatch thread, which results in expensive cross-core communication) and presents a central bottleneck which prevents us from achieving higher throughput by adding more cores.

CPU	Xeon Gold 6148 (2 sockets × 20 cores @ 2.40GHz)
RAM	384 GB DDR4-2666
Networking	100Gbps Intel Omni-Path Interconnect

Table 2: The hardware configuration used for benchmarks. All machines ran CentOS 7.3.1611 (Core) with hyperthreading disabled. The network fabric used a two-level fat-tree topology to provide full bisection bandwidth at 100 Gbps per machine.

	Total records processed (# servers used)			
Time budget	**MilliSort**	**Q1**	**Q2**	**Q3**
1 ms	0.84 M (120)	47.6 M (280)	6.72 M (140)	0.034 M (60)
10 ms	26 M (280)	980 M (280)	224 M (280)	2.24 M (280)
Scaling	31.0x (2.3x)	20.6x (1x)	33.3x (2x)	67.9x (4.7x)

Table 3: Overall performance of MilliSort and MilliQuery.

- The underlying Omni-Path PSM2 library [26] we use to send and receive packets is actually not a packet I/O driver but a reliable message-passing transport with its own congestion/flow control, which adds considerable overhead; in addition, the Omni-Path host fabric interface (HFI) does not have a lightweight mechanism for transfering small chunks of data via DMA, so we have to send/receive packet data using programmed I/O, which leads to higher CPU overhead and, even worse, cache pollution.

As a result, a single server cannot fully utilize its network bandwidth when the messages are relatively small. In our experiments, we decided to place multiple servers on each machine, each with its own dispatch thread, to increase the network utilization. This approach can scale the overall message throughput of each machine linearly, but it also incurs higher coordination overhead due to the increased number of servers. Future implementations that are based on a more efficient networking stack such as [32, 18] could eliminate the need to run multiple servers per machine.

6 Performance measurements

Our goal in evaluating MilliSort and MilliQuery was to answer the following questions:

- How much data can be processed in one millisecond (or ten milliseconds), and how many servers can be harnessed efficiently in each interval?
- How does application behavior change if the time budget is increased or decreased?
- What factors limit the applications' performance and scalability, and what stresses do these applications create for underlying infrastructure?
- How effective is MilliSort's new partitioning mechanism?
- How efficient is MilliSort compared to purely local sort or other distributed sorts?

We used the hardware configuration shown in Table 2 to evaluate MilliSort and MilliQuery. To better utilize the network bandwidth (discussed in Section 5.4), we ran four independent servers on each machine, two on each socket. We had access to 70 machines in the cluster, which allowed up to 280 servers; and there were no competing tasks running on the machines. All MilliSort experiments used 2-level partitioning.

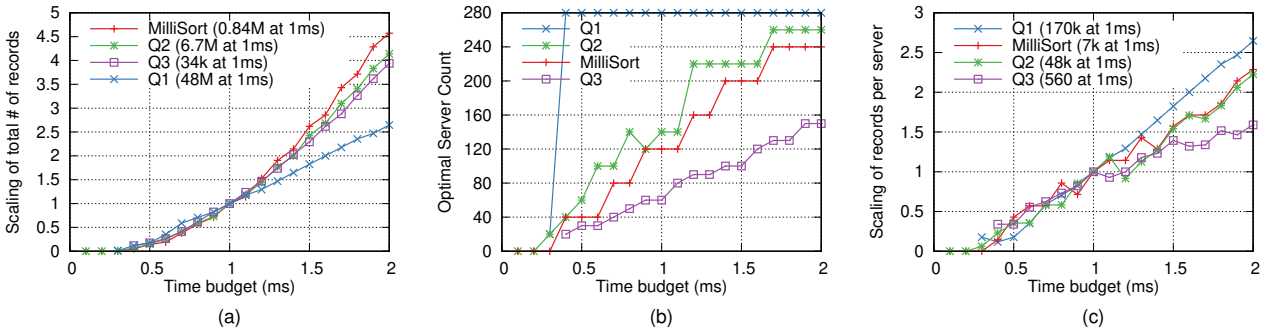

Figure 4: Scaling properties of MilliSort and MilliQuery as a function of time budget: (a) size of dataset processed (normalized to a value of 1.0 for a 1 ms time budget for each benchmark); (b) cluster size that yields the largest dataset processed; (c) records per sever at largest dataset processed (also normalized to a value of 1.0 for a 1 ms time budget).

	120 servers (0.84M records)				**240 servers** (1.68M records)			
Phase	Mean	Max	%	Max/Min	Mean	Max	%	Max/Min
Local Sort	147.0	216.3	22.2%	1.83	137.8	202.3	12.7%	1.77
Partitioning	200.5	214.4	22.0%	1.16	410.4	428.6	26.9%	1.11
Pivot Shuffle	83.2	87.9	9.0%	1.13	219.3	240.1	15.1%	1.27
Shuffle	377.2	402.8	41.3%	1.16	738.9	789.0	49.6%	1.14
Rearrangement	128.1	142.7	14.6%	1.18	146.9	173.3	10.9%	1.26
Total	942.3	976.0	100%	1.09	1523.8	1591.1	100%	1.08

Table 4: Time breakdown of each MilliSort phase with two different cluster sizes, where per-node dataset sizes are fixed at 7000 records. All times are in µs and reflect the median over 200 runs. "%" is the time spent by the slowest server for that phase, as a fraction of total time. "Max/Min" is the ratio of times for the slowest and fastest servers for that phase. "Partitioning" includes the time spent on "Pivot Shuffle".

6.1 Overall performance

We varied the amount of data per server and the size of the cluster to find the largest amount of data that can be processed by each of the applications in either 1 ms or 10 ms; Table 3 shows the results, along with the best configurations for each interval. With a 10 ms time budget all of the applications can harness all 280 available servers and they can process 2.2–224M records, depending on the application. A 1 ms time budget is more challenging for most of the applications. Only MilliQuery Q1 can use all of the servers; the other applications ranged from 60–140 servers. The amount of data processed in 1 ms varied dramatically among the applications, from a low of 34K records in MilliQuery Q3 to a high of 48M records in Q1.

6.2 Quadratic scaling

The total number of records that can be processed in an interval varies quadratically with the size of the interval. If the time budget increases by a factor of X, we observe that both the data handled by each server and the number of servers increase by at least X, for a total increase in throughput of at least X^2. This effect can be seen in Figure 4. Figure 4(a) shows that all of the benchmarks except Q1 exhibit quadratic or better scaling as the time budget increases from 1 ms to 2 ms. Q1 would scale exponentially if more servers were available, but it already used all of the available servers at 1 ms, so it can only scale linearly by increasing the amount of data per server. Figure 4(b) shows optimal cluster size as a function of time budget. All except Q1 exhibit almost linear scaling. Figure 4(c) shows the scaling of the per-server dataset size as the time budget increases. All except Q3 scale at least linearly, after a fixed initial overhead. Q3 scales per-server records slightly less than linearly.

The quadratic behavior can also be seen for Q3 in Table 3: its throughput scales by 68x as the time budget increases from 1–10 ms. Scaling for the other applications becomes limited by the available servers long before reaching the 10 ms time budget; Figure 4(b) shows that MilliSort, Q1, and Q2 have consumed almost all the available servers with a 2 ms time budget. Linear scaling of servers suggests that these applications could harness at least 1200 servers with a 10 ms budget (5x the number at 2 ms).

6.3 Scaling below 1 ms

Quadratic scaling also means that throughput drops rapidly with time budgets less than 1 ms. This is visible in Figure 4(a). With a time budget of 0.5 ms, throughput has dropped by more than 4x for all of the applications, and none of the applications has appreciable throughput for budgets less than 0.5 ms. For these applications, the lower bound on useful timescale is around 0.5–1.0 ms with our current per-message overheads.

6.4 Limiting factors for scalability

There are two primary factors that limit the ability of the applications to scale up in servers or down in time: coordination and shuffles. The costs of both activities increase with the cluster size; when the amount of data per server is zero, these are essentially the basic costs of harnessing servers. Table 4 illustrates the effect of these two factors by showing the cost of each MilliSort phase for two different cluster sizes with the records per server held constant. In the 120-server configuration, partitioning and shuffles take 63% of the total running time. If the cluster size is doubled, the time taken by these two activities is also doubled (1218 µs vs. 617 µs), even though each server still processes roughly the same amount of data. The fraction of total running time consumed by coordination and shuffles increases from 63% to 77%. At the same time, the combined cost of other phases remains almost constant in both configurations. In general, when the time budget is held constant, larger clusters result in larger basic costs, so less time is left for actual work such

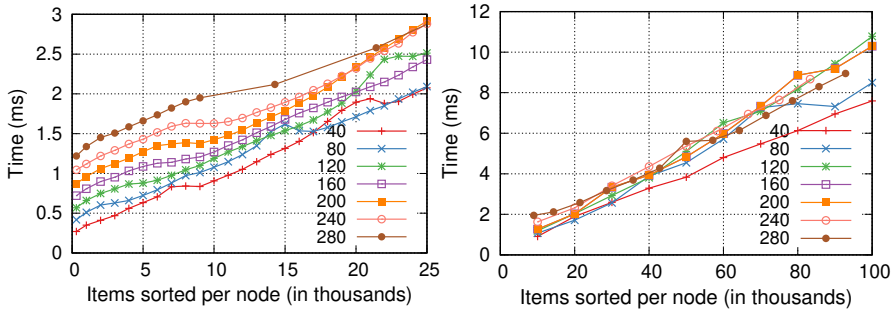

Figure 5: Total sorting time for MilliSort as a function of records per server, with different cluster sizes. The left figure assumes a 1 ms goal and the right figure assumes 10 ms. Each line represents a different cluster size (the number in the legend is the server count). Each data point is the median time from 200 runs.

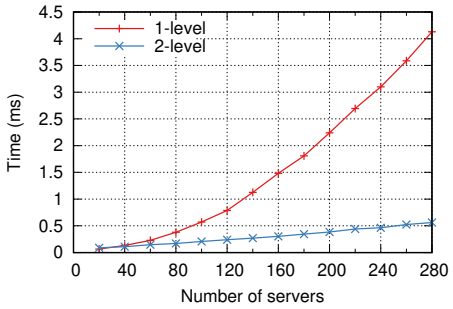

Figure 6: MilliSort's partitioning time as a function of cluster size with different partitioning schemes.

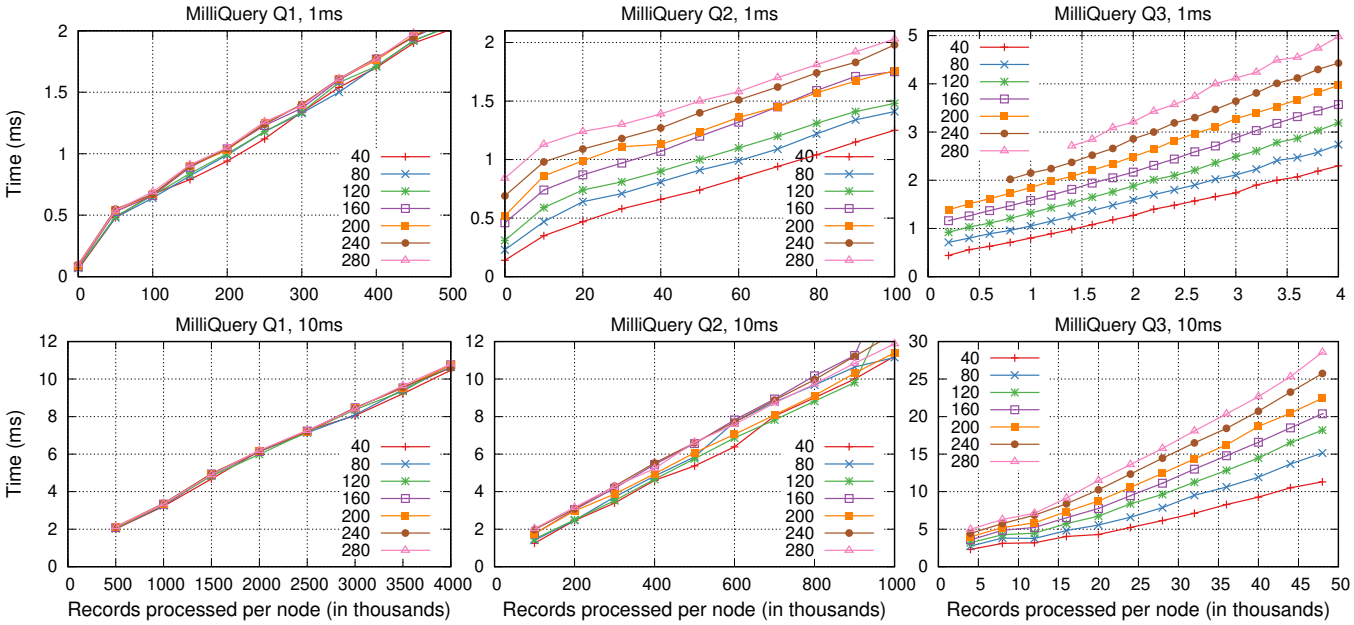

Figure 7: Total processing time for the three MilliQuery queries as a function of records per server, with different cluster sizes. From left to right, the figures represent Q1, Q2, and Q3 respectively. Each data point is the median time from 100 runs.

as local computation and data transfer. As a result, this limits our ability to harness more servers within the time budget or perform meaningful flash bursts in smaller timescales.

Figure 5 and 7 provide more details of the two limiting factors for MilliSort and MilliQuery by showing how the total processing time changes with the cluster size and amount of data per server. In most graphs of Figure 5 and 7, the lines for different cluster sizes are roughly parallel, indicating that the marginal cost of handling additional data is about the same for all sizes (the marginal cost starts higher when the amount of data per server is smaller, but plateaus out quickly once the benefit of batching diminishes). However, larger cluster sizes have larger fixed overheads (y-intercepts of the lines), which consist of the partitioning cost plus the per-message costs of the shuffles (a shuffle must send one message to each peer, even if it only contains a single record). These figures also show the diminishing returns at timescales less than 1 ms: even as the number of records per server approaches zero, running times remain 0.3–1.2 ms and 0.15–0.84 ms for MilliSort and MilliQuery Q2 respectively, depending on cluster size.

MilliQuery Q3 difers from the other applications in that its total processing time increases more than linearly with the input records per server, and the rate of increase is higher for larger clusters (the gaps between the lines in the right graphs of Figure 7 become wider for larger numbers of input records). This is because Q3 uses join operations where the number of output records tends to increase quadratically with the number of input records, so the number of input records doesn't reflect the actual number of records each server has to process.

Different applications have very different fixed overheads of harnessing servers. Q1 has the smallest fixed overheads and they are about the same for all cluster sizes. This is because Q1 requires very little coordination and doesn't use shuffle; the only communication primitives used by Q1 are broadcast and gather, whose overheads increase only in log scale with the cluster size. MilliSort and Q2 are very similar: their fixed overheads increase almost linearly with the cluster size since both applications require a big shuffle at the end. However,

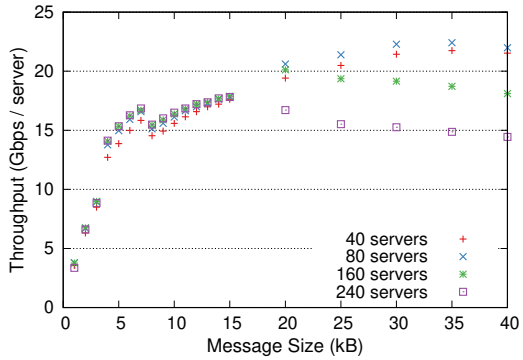

Figure 8: Stand-alone shuffle benchmark performance as a function of the cluster size and the message size (amount of data sent from each source to each destination).

for the same cluster size, the fixed overhead of MilliSort is higher due to the additional partitioning cost (with 2-level partitioning, this cost also increases almost linearly with the cluster size; this will be discussed in Section 6.6). For the same reason, the lines of Q2 in the graphs are closer than the lines of MilliSort. Finally, the fixed overheads of Q3 are about 3x higher than Q2 for all cluster sizes (unfortunately, it's hard to see in the graphs), as Q3 requires a total of three shuffles.

Figure 5 and 7 also show that efficiency improves with increasing records per server. For MilliSort, once the number of records per server reaches about 25000, the coordination costs become small compared to other factors and the shuffle efficiency improves, so there is little difference in overhead between different cluster sizes (the 40-server cluster remains significantly more efficient than the other cluster sizes because it avoids contention in the network core; this will be discussed in Section 6.5). The closeness of the lines in these 10 ms graphs indicates that all applications except Q3 could easily harness many more than 280 servers efficiently with a 10 ms time budget.

The next subsections discuss shuffle and partitioning costs in more detail.

6.5 Shuffle efficiency

Shuffles present the most significant challenge to scalability in our experiments, especially at 1 ms time scales. This is reflected in Table 3: Q1, which has no shuffles, can harness far more servers and process far more data than the other benchmarks, while Q3, which has three shuffles, is the least scalable. MilliSort and Q2, which each use one shuffle, fall in between. In Table 4, when the number of servers doubles while fixing the number of records per server, 85% of the time increase comes from additional shuffle costs; shuffle costs affect not only the main shuffle phase, but also the partitioning phase.

To get a better understanding of shuffle costs, we ran a stand-alone shuffle benchmark in which each of a group of servers sends a fixed-size message to each other server. Figure 8 graphs shuffle performance in terms of throughput per server. We ran four servers per machine, so the ideal throughput per server would be 25 Gbps.

We observed two different potential reasons that contribute to the higher shuffle time with more servers. First, as the message size decreases, efficiency drops because of per-message overheads. With 120 servers and 0.96M total records, the average message size for shuffle is about 6.7KB, which still provides a good throughput. However, with 240 servers and 1.92M total records, the average shuffle message size drops to about 3.3KB, resulting in less than 10 Gbps throughput per server. This also justifies the quadratic scaling property discussed in Section 6.2; when the number of servers and number of records per server are scaled simultaneously, we have a better chance of maintaining shuffle efficiency since the average message size for shuffle is fixed.

Second, even with large messages, throughput per server drops as the cluster size increases. It drops from 22 Gbps per server (with 40 servers) to less than 15 Gbps per server with 240 or more servers. This is because the cluster network uses flow-consistent load balancing, rather than packet-level load balancing. With large numbers of active transmissions, paths conflict in their link usage, resulting in congestion on those links and under-usage of other links. As the cluster size increases, shuffles consume a larger fraction of the core bandwidth, which makes congestion more likely, and our high-entropy approach to shuffles cannot completely compensate. As a result, shuffles cannot harness the full bisection bandwidth offered by the network. We speculate that more granular in-network load-balancing techniques such as [47, 64] may help remove this bottleneck in the future.

6.6 Partitioning cost

Figure 6 shows the results of a stand-alone experiment that measures partitioning time as a function of the number of servers. The multi-level partitioning algorithm we developed for MilliSort provides a significant peformance benefit: by the time the number of servers reaches 200, the 2-level approach is more than 5x as fast as the 1-level approach. As the number of servers increases, the partitioning time increases superlinearly for both 1-level and 2-level partitioning schemes. However, for 2-level partitioning, the increase in time is almost linear up to 280 servers (each additional server adds about 2 µs in our current implementation). A 1-level approach is too slow to harness 100 or more servers in a 1 ms budget, but a 2-level approach can easily coordinate 120 servers within 1 ms and 280 servers within 10 ms. In the best configurations for 1 ms and 10 ms time budgets, the partitioning cost only accounts for about 20% and 5% of the total time, respectively. Unfortunately, even with 2-level partitioning, the coordination cost for 280 servers is already more than half a millisecond; this prevents us from harnessing hundreds of servers in 1 ms for MilliSort. We didn't implement 3-level partitioning and beyond, but we expect that increasing the number of levels will further reduce the coordination cost for large clusters.

6.7 MilliSort efficiency

It might seem that flash bursts must sacrifice throughput (or efficiency) in order to operate at millisecond timescales.

	Throughput (efficiency)	
	120 servers (0.84M records)	**280 servers** (26M records)
MilliSort	7172 (19.6%)	10378 (33.1%)
Ideal Distr. Sort	11272 (30.8%)	13600 (43.3%)
Local Sort	36656 (100%)	31398 (100%)

Table 5: MilliSort efficiency of the best configurations for 1 ms and 10 ms time budgets, compared to local sort and ideal distributed sort. Throughput numbers are in "records/ms/server". Efficiency is the relative throughput compared to local sort.

	CPU	SMT	BW/core	Throughput
MilliSort	Xeon Gold 6148 @ 2.4GHz	1	3.1	1297 (1.00x)
TencentSort	IBM POWER8 @ 2.9GHz	8	5.0	1977 (1.52x)
CloudRAMSort	Xeon X5680 @ 2.9GHz	2	2.7	707 (0.55x)

Table 6: Per-core throughput comparison of MilliSort, Tencent Sort [28], and CloudRAMSort [34]. "SMT" (i.e., simultaneous multithreading) is the number of hardware threads per core. "BW/core" is the average network bandwidth per core, in Gbps. "Throughput" numbers are in "records/ms/core". MilliSort's per-core throughput is computed from the 10 ms configuration in Table 5, while Tencent Sort and CloudRAMSort's numbers are computed from their published results [28, 34]. Tencent Sort's reported throughput is for sorting from disk to disk; we estimate that less than half of the time is spent on disk I/O, so we double its reported throughput for a fair comparison.

However, our results suggest that this need not be true. Table 5 shows the overall efficiency of MilliSort compared to purely local sort and ideal distributed sort. The ideal distributed sort represents an imaginary situation where the partitioning cost is zero and data are shuffled at full network bandwidth; this provides an upper bound on the throughput of any distributed sort. Despite operating at a 1 ms timescale, MilliSort is relatively efficient (e.g., the maximum possible throughput for distributed sorts is only about 50% higher); MilliSort is even more efficient at a 10 ms timescale.

MilliSort's throughput is also on par with state-of-the-art sorting systems [34, 28] that have much longer running times. Table 6 summarizes the hardware differences and presents the throughput numbers in "records/ms/core" for comparison. In particular, Tencent Sort [28] is the current record holder for the GraySort benchmark [61], and its total running time is about 100 seconds. MilliSort's throughput per core at 10 ms is only 33% lower than Tencent Sort, even though Tencent Sort's POWER8 cores have 8x the hardware threads and 1.6x the average network bandwidth of MilliSort cores.

Table 5 also provides insight into why distributing data at very fine granularity makes sense. Network communication is the largest source of overhead for distributed computation (the throughput of local sort is 3–5x higher than MilliSort in Table 5). However, most of this cost is paid immediately when scaling beyond a single machine. For example, performing a distributed sort with just two machines requires half of the data to be sent over the network. There is not much additional loss in efficiency when scaling out further. This suggests that if you can afford to distribute at all, you can afford to distribute a lot.

7 Observations

This section summarizes the key observations that emerged from our MilliSort and MilliQuery experiments:

Granular data distribution. Distributing data in fine granularity allows one to harness more CPU cores and aggregated network bandwidth for faster computation and communication. Our experiment also confirms that, at least for some dwarfs, it is possible to scale out quite efficiently even at millisecond timescale. Thus, we predict that future distributed data-parallel systems will need to be optimized for large scale-out architectures with smaller data per server. This will likely present interesting challenges to the design of future systems since they will be required to scale down gracefully (i.e., operate efficiently even on smaller data).

Efficient group communication is essential. Even simple dwarfs have complex communication patterns internally; all of the benchmarks except MilliQuery Q1 depend heavily on group communication. Even when carefully optimized, they account for 50%–60%, 35%–40%, and 50%–65% of the overall running time in the best configurations of MilliSort, MilliQuery Q2, and MilliQuery Q3, respectively.

Per-message overhead is critical. Network bandwidth and latency are well known to be important for the performance of large-scale systems, and they remain important for flash bursts. Flash bursts differ from traditional large-scale systems in that per-message costs are at least as important as the traditional metrics. This is because group communication primitives tend to generate many small messages when operating at small time scales with large cluster sizes.

Coordination must be structured hierarchically. It is well known in the HPC community that communication must be structured hierarchically to handle large cluster sizes. Thus, group communication primitives are commonly implemented in a tree or hypercube topology [62], so that each machine only communicates with a small number of its peers. This helps to mitigate per-message overheads and harness more aggregated network bandwidth. This principle also applies to coordination. As an example, MilliSort's partition phase is structured as a hierarchical series of distributed sorts, so that the computation doesn't become a central bottleneck.

Low-latency shuffle is challenging. Of all the group communication primitives, shuffle is the most challenging in flash bursts. There are several reasons for this. First, per-message overheads become critical since shuffles require each server to send many small messages. Unlike other primitives, a hierarchical approach to shuffles generally isn't practical because it doubles the network bandwidth utilization. Second, shuffles need to use the full network bandwidth of each server for best performance, but many networks do not provide full bisection bandwidth, especially at large scale. Even where full bisection bandwidth is available, transient congestion can occur due to imperfect bipartite matching, either in the network itself (because of routing) or in the application.

Finally, hash partitioning often results in non-trivial data imbalance (e.g., in MilliQuery Q2, the most unlucky server typically has to process 1.5x much data as the average).

Quadratic scaling. As the time budget increases, the number of machines that can be harnessed effectively increases at least linearly. The amount of data each server can process also grows at least linearly with the time budget, so the overall dataset size grows at least quadratically with time budget. For some workloads, such as MilliQuery Q1, where the only coordination overhead is a final aggregation, the number of machines can grow exponentially with the time budget, so the dataset size grows faster than quadratically. The flip side of this is that reducing the time budget results in superlinear reductions in the number of servers and overall dataset size, which creates a lower limit on the timescale at which the system can operate efficiently. In our experiments, 1 ms appears to be tractable for all of the workloads except MilliQuery Q3. Future systems that are built on top of better networking infrastructure will likely enable these workloads to run efficiently at an even smaller timescale.

Ultimate limits of strong scaling. If we increase the number of machines while fixing the dataset size, efficiency inevitably decreases; this eventually prevents us from harnessing more machines to speed up the job. While it is well known that the theoretical speedup of a parallel program is ultimately limited by its serial portion (Amdahl's Law), this is not the limiting factor in our experiments. Instead, two other factors result in the drop in efficiency. First and foremost, coordination cost rises. This is mostly due to per-message overheads (e.g., manifested as higher shuffle costs). Coordination algorithms also take longer to run (e.g., MilliSort's partitioning time increases linearly with the number of servers, even with the recursive scheme). Second, straggler effects are more significant when there are more servers and less data per server (e.g., even in MilliQuery Q1, the overall efficiency drops about 50% when scaling from 40 servers to 280 servers).

8 Applicability of results

We conducted our experiments with limitations and assumptions that may not apply to today's computing systems. In this section we discuss some of these factors and argue that our results will be relevant for future systems.

Is 1–10 ms the right target? 1–10 ms is not the timescale at which most people think of large-scale distributed computation today. For example, response times for users of a few hundred milliseconds are considered adequate, and it can take 100 ms just to communicate between browser and datacenter. However, applications such as AR/VR require response times of 10 ms or better, and new edge computing offerings such as AWS Local Zones and WaveLength [6] can already provide single-digit millisecond latency between the end-user and an edge cloud. Furthermore, there are increasing numbers of applications that make real-time decisions without humans in the loop. Examples include controllers for autonomous vehicles [42] and IoT devices [24], which make decisions on the order of 10 ms, and financial applications [22, 51], for which there appears to be no lower bound on desirable latency. Flash bursts can enable these applications to run data-intensive algorithms at millisecond timescale.

Idealized experiment setup. Our experiments are conducted on a bare-metal HPC cluster with no interference from competing workloads. A dedicated cluster is not economically feasible in practice; however, we believe that recent work on colocating latency-critical and batch jobs [48, 18] can be applied to achieve high CPU efficiency without hurting the performance of flash bursts. In addition, we assume input data are resident in memory when the experiments start. This may not be a realistic assumption today, where data is stored on flash or disk. However, future datacenter systems are likely to store data in nonvolatile memories (NVMs) with access times not far above today's DRAM [27]. Ideally, future applications will run directly on NVM-based storage servers; in the worst case an initial shuffle step will be needed to extract data from the storage servers to computational nodes.

Missing pieces. Given the tight time budget, a flash burst requires every component involved in its lifetime to support low latency. There are several elements that our work did not address, such as application loading, but there are many other projects attacking these pieces, such as storage systems [49, 12, 41, 36, 25], cluster schedulers [31], networking infrastructure [46, 32, 43, 37, 14], fast threading and dispatching [8, 52, 30, 48, 54], and lightweight virtualization [1]. Many interesting problems have yet to be solved [39] in order to create a unified flash burst infrastructure.

9 Related work

There are several efforts underway to support shorter task durations in a cloud setting. For example, major serverless platforms [5, 45, 20] can support tasks, also called serverless functions, as small as 100 ms. Although serverless functions were initially intended for simple microservices, many projects have successfully used them to parallelize jobs over thousands of cores in the cloud. To overcome the relatively high overheads of the serverless platform, these projects typically targeted compute-intensive workloads and large time budgets such as 1 min. ExCamera [17] harnessed 3600 cores for 2 mins for video encoding; gg [16] harnessed 384 cores for 1.5 mins for software compilation; PyWren [29] sorted 1TB of data in 204 seconds with 1000 workers; Sprocket [3] harnessed 1000 cores for less than 1 min for video processing. Finally, these projects have mainly focused on exploiting large parallelisms in applications and reducing monetary cost, as opposed to revealing or addressing the new problems that emerged by coordinating many nodes at small timescales.

In addition to serverless computing, HPC is another area that greatly inspires the development of flash bursts. Scalability is of utmost importance to HPC applications, and

highly-optimized HPC applications can scale beyond ten thousand compute nodes in a supercomputer. Furthermore, unlike many data analytic workloads, traditional HPC workloads such as scientific simulation typically require frequent communication between compute nodes (e.g., at the end of each simulation timestep). Flash bursts share the goal of running communication-intensive workloads in a scalable way, and we borrow techniques that are well explored in the HPC community (e.g., efficient group communication primitives) to achieve this goal. However, flash bursts differ from HPC in two important aspects. First, HPC workloads usually run for much longer time (hours or more), so they don't expose the interesting problems of operating at millisecond timescales. Second, HPC workloads usually run in a cleaner environment which has less interference from other jobs, so system-level challenges such as performance isolation and resource utilization are less of a concern to HPC.

Shuffle is known to be one of the most expensive operations in distributed data analytics. Many projects have focused on optimizing its performance. Riffle [68] and Magnet [57] are two recent shuffle services designed to optimize disk-to-disk shuffle performance in Spark. Locus [53] implemented shuffle in a serverless setting by mixing different cloud storage services to balance shuffle performance and storage cost. Dataflow Shuffle [9] is an in-memory shuffle tier used by Google BigQuery. These systems focused on shuffling large data and did not address the challenges in low-latency shuffle.

Distributed sorting has been studied extensively for many decades, with a variety of well-established benchmarks [61]. Most prior work has focused on sorting on-disk data at large scale (e.g., TritonSort [55]). One exception is CloudRAMSort [34], which is designed to sort in-memory data to speed up database operations. It can sort 1 TB data in 4.6 seconds using 256 servers. As a comparison, the per-core throughput of MilliSort at 10 ms is almost 2x higher than CloudRAMSort despite operating at 1000x smaller timescales.

Another approach to speed up data-intensive computation is to exploit shared-memory parallelism on more powerful servers (i.e., scale-up). The major benefit of scale-up systems is efficiency because they can avoid the expensive network communication and overhead induced by fault tolerance. As shown in [13, 59], scale-up systems can be orders of magnitude faster than traditional systems like Spark [67]. However, there is a practical limit on the number of CPU cores available on a single server (a few hundred). Also, scale-up systems are much less flexible when the application requires fast data movement: if the input data need to be loaded over the network, the bandwidth of a single server will become a significant bottleneck. As a result, we expect both scale-up systems and flash burst to be valuable depending on the application.

10 Conclusion

MilliSort and MilliQuery demonstrate that several core patterns in data analytics can run efficiently at millisecond timescales. With a budget of 1 ms, most of the patterns can harness at least 100 servers; with a budget of 10 ms, our results suggest that most of the patterns can harness at least 1000 servers. Furthermore, our results indicate that there is only a small efficiency penalty for running at millisecond timescales. We identified two related problems that currently limit scalability: per-message costs and shuffle overheads. If future systems can improve on our implementation in these areas, it should be possible to execute large-scale computations even more efficiently, and at timescales even less than one millisecond. We do not yet know whether applications can take advantage of these small timescales, but we hope our results will encourage application developers to explore the potential benefits of running large-scale computations at millisecond granularity.

Acknowledgements

We thank our shepherd, Haryadi Gunawi, and our anonymous NSDI and OSDI reviewers for their feedback. We also thank Collin Lee for the early discussions that helped develop many of the ideas. This work was supported by the industrial affiliates of the Stanford Platform Lab, a Samsung Scholarship, and NSF grant CNS-191067.

References

[1] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 419–434.

[2] ANON ET AL, BITTON, D., BROWN, M., CATELL, R., CERI, S., CHOU, T., DEWITT, D., GAWLICK, D., GARCIA-MOLINA, H., GOOD, B., GRAY, J., HOMAN, P., JOLLS, B., LUKES, T., LAZOWSKA, E., NAUMAN, J., PONG, M., SPECTOR, A., TRIEBER, K., SAMMER, H., SERLIN, O., STONEBRAKER, M., REUTER, A., AND WEINBERGER, P. A Measure of Transaction Processing Power. *Datamation 31*, 7 (Apr. 1985), 112–118.

[3] AO, L., IZHIKEVICH, L., VOELKER, G. M., AND PORTER, G. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC 18, Association for Computing Machinery, pp. 263–274.

[4] ASANOVIC, K., BODIK, R., DEMMEL, J., KEAVENY, T., KEUTZER, K., KUBIATOWICZ, J., MORGAN, N., PATTERSON, D., SEN, K., WAWRZYNEK, J., WESSEL, D., AND YELICK, K. A View of the Parallel Computing Landscape. *Commun. ACM 52*, 10 (Oct. 2009), 56–67.

[5] AWS Lambda. `https://aws.amazon.com/lambda`.

[6] AWS for the Edge: Bringing data processing and analysis closer to end-points. https://aws.amazon.com/edge/.

[7] AXTMANN, M., WITT, S., FERIZOVIC, D., AND SANDERS, P. In-place Parallel Super Scalar Samplesort (IPSSSSo). *CoRR abs/1705.02257* (2017).

[8] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 49–65.

[9] BLOG, G. C. How distributed shuffle improves scalability and performance in cloud dataflow pipelines. https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines, 2018.

[10] Data Plane Development Kit. http://dpdk.org/.

[11] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM 51* (January 2008), 107–113.

[12] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[13] ESSERTEL, G., TAHBOUB, R., DECKER, J., BROWN, K., OLUKOTUN, K., AND ROMPF, T. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 799–815.

[14] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., ET AL. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 51–66.

[15] Apache Flink - Stateful Computations over Data Streams, Jan 2021. https://flink.apache.org.

[16] FOULADI, S., ROMERO, F., ITER, D., LI, Q., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 475–488.

[17] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)* (Boston, Massachusetts, USA, 2017), USENIX.

[18] FRIED, J., RUAN, Z., OUSTERHOUT, A., AND BELAY, A. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association.

[19] GAO, W., ZHAN, J., WANG, L., LUO, C., ZHENG, D., TANG, F., XIE, B., ZHENG, C., WEN, X., HE, X., ET AL. Data Motifs: A Lens Towards Fully Understanding Big Data and AI Workloads. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (2018), pp. 1–14.

[20] Google Cloud Functions. https://cloud.google.com/functions/.

[21] HARSH, V., KALE, L., AND SOLOMONIK, E. Histogram sort with sampling. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2019), SPAA '19, Association for Computing Machinery, p. 201212.

[22] HASBROUCK, J., AND SAAR, G. Low-latency trading. *Journal of Financial Markets 16*, 4 (2013), 646–679.

[23] HOFMANN, M., RÜNGER, G., GIBBON, P., AND SPECK, R. Parallel sorting algorithms for optimizing particle simulations. In *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)* (2010), IEEE, pp. 1–8.

[24] HU, J., BRUNO, A., ZAGIEBOYLO, D., ZHAO, M., RITCHKEN, B., JACKSON, B., CHAE, J. Y., MERTIL, F., ESPINOSA, M., AND DELIMITROU, C. To centralize or not to centralize: A tale of swarm coordination. *arXiv preprint arXiv:1805.01786* (2018).

[25] Daos: Distributed asynchronous object storage, Aug. 2020. https://github.com/daos-stack/daos.

[26] Intel OPA-PSM2 Git Repository, Feb. 2020. https://github.com/intel/opa-psm2.git.

[27] IZRAELEVITZ, J., YANG, J., ZHANG, L., KIM, J., LIU, X., MEMARIPOUR, A., SOH, Y. J., WANG, Z., XU, Y., DULLOOR, S. R., ZHAO, J., AND SWANSON,

S. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[28] JIANG, J., ZHENG, L., PU, J., CHENG, X., ZHAO, C., NUTTER, M. R., AND SCHAUB, J. D. Tencent sort.

[29] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC 17, Association for Computing Machinery, pp. 445–451.

[30] KAFFES, K., CHONG, T., HUMPHRIES, J. T., BELAY, A., MAZIÈRES, D., AND KOZYRAKIS, C. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 345–360.

[31] KAFFES, K., YADWADKAR, N. J., AND KOZYRAKIS, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 158–164.

[32] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 1–16.

[33] KALTOFEN, E. L. The seven dwarfs of symbolic computation. In *Numerical and symbolic scientific computing*. Springer, 2012, pp. 95–104.

[34] KIM, C., PARK, J., SATISH, N., LEE, H., DUBEY, P., AND CHHUGANI, J. CloudRAMSort: Fast and Efficient Large-Scale Distributed RAM Sort on Shared-Nothing Cluster. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (2012), pp. 841–850.

[35] KOWALEWSKI, R., JUNGBLUT, P., AND FRLINGER, K. Engineering a distributed histogram sort. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)* (2019), pp. 1–11.

[36] KULKARNI, C., MOORE, S., NAQVI, M., ZHANG, T., RICCI, R., AND STUTSMAN, R. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 627–643.

[37] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., ET AL. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 514–528.

[38] LAKSHMANAN, V., AND TIGANI, J. *Google BigQuery: the Definitive Guide: Data Warehousing, Analytics, and Machine Learning at Scale*. O'Reilly Media, 2019.

[39] LEE, C., AND OUSTERHOUT, J. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2019), pp. 149–154.

[40] LI, X., LU, P., SCHAEFFER, J., SHILLINGTON, J., WONG, P. S., AND SHI, H. On the versatility of parallel sorting by regular sampling. *Parallel Computing 19*, 10 (1993), 1079 – 1103.

[41] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 429–444.

[42] LIN, S.-C., ZHANG, Y., HSU, C.-H., SKACH, M., HAQUE, M. E., TANG, L., AND MARS, J. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), pp. 751–766.

[43] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., ET AL. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 399–413.

[44] MCSHERRY, F. Graph analysis and hilbert space-filling curves. `https://bigdataatsvc.wordpress.com/2013/07/02/graph-analysis-and-hilbert-space-filling-curves/`, Jul 2013. Accessed: 2020-01-30.

[45] Microsoft Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`.

[46] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proc. 2018 ACM SIGCOMM Conference* (New York, NY, USA, 2018), SIGCOMM '18, ACM.

[47] Dispersive Routing - Intel Omni-Path Fabric Performance Tuning, Aug 2020. https://edc.intel.com/content/www/xl/es/design/products-and-solutions/networking-and-io/fabric-products/omni-path/intel-omni-path-performance-tuning-user-guide/dispersive-routing/.

[48] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 361–378.

[49] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Commun. ACM 54* (July 2011), 121–130.

[50] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[51] PRABHAKAR, B. Time perimeters and stock exchanges in the cloud. https://conferences.sigcomm.org/sigcomm/2020/tutorial-netfinance.html, Aug 2020. Accessed: 2020-08-30.

[52] PREKAS, G., KOGIAS, M., AND BUGNION, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 325–341.

[53] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 193–206.

[54] QIN, H., LI, Q., SPEISER, J., KRAFT, P., AND OUSTERHOUT, J. Arachne: Core-aware Thread Management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2018), OSDI'18, USENIX Association, pp. 145–160.

[55] RASMUSSEN, A., PORTER, G., CONLEY, M., MADHYASTHA, H. V., MYSORE, R. N., PUCHER, A., AND VAHDAT, A. TritonSort: A Balanced Large-scale Sorting System. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 29–42.

[56] RUEFENACHT, M., BULL, M., AND BOOTH, S. Generalisation of recursive doubling for allreduce: Now with simulation. *Parallel Computing 69* (2017), 24–44.

[57] SHEN, M., ZHOU, Y., AND SINGH, C. Magnet: Push-based shuffle service for large-scale data processing. *Proc. VLDB Endow. 13*, 12 (2020), 3382–3395.

[58] SHI, H., AND SCHAEFFER, J. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing 14*, 4 (1992), 361 – 372.

[59] SHUN, J. *Shared-memory parallelism can be simple, fast, and scalable*. PUB7255 Association for Computing Machinery and Morgan & Claypool, 2017.

[60] SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd. (revised) ed. MIT Press, Cambridge, MA, USA, 1998.

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

[62] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[63] TIGANI, J., AND NAIDU, S. *Google BigQuery Analytics*. John Wiley & Sons, 2014.

[64] Broadcom's Trident 3 enhances ECMP with Dynamic Load Balancing, Sept 2017. https://www.broadcom.com/blog/broadcom-s-trident-3-enhances-ecmp-with-dynamic-load-balancing.

[65] WICKRAMASINGHE, U., AND LUMSDAINE, A. A Survey of Methods for Collective Communication Optimization and Tuning. *CoRR abs/1611.06334* (2016).

[66] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 423–438.

[67] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

[68] ZHANG, H., CHO, B., SEYFE, E., CHING, A., AND FREEDMAN, M. J. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–15.

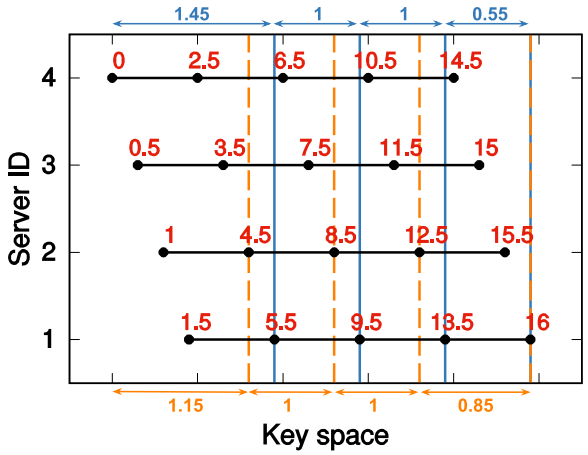

Figure 9: A comparison of the splitter selection algorithms on an example scenario. The keys of each server's data are uniformly distributed in a range indicated by a horizontal line, where dots are pivots (starting and ending pivots are not considered by the original algorithm). The splitters selected by the original algorithm are indicated with solid blue vertical lines. The splitters selected by the weighted algorithm are indicated with dashed vertical lines. The numbers on pivots are the cumulative pivot weights used by the weighted algorithm. The numbers on arrows above and below the diagram indicate the relative sizes of the buckets for the original and weighted algorithms, respectively.

A Appendix

A.1 Improved splitter selection

The splitter selection algorithm described in Section 4.2 tends to produce unbalanced partitions where the first server contains considerably more records than the last server. This imbalance can be explained by considering the groups of keys delimited by the pivots from each server; all of the groups contain about the same number of records. If we choose the Mth smallest pivot as the first splitter (assuming $s = 1$), then the first server will contain M full groups of records. In addition, it will contain some records from up to $M-1$ additional groups, whose contents are divided between the first two servers (see Figure 9). Thus, the first server is likely to contain about $1.5M$ groups worth of records. In contrast, the last server will contain records from at most M groups, of which all but one are partial (some of their records have keys smaller than the last splitter). Thus, the last server will probably hold only about $0.5M$ groups of data.

To mitigate the data imbalance, we developed a new weighted approach to selecting splitters. The original approach behaved as if all of the keys in each group had the same value as the pivot at the end of the group. The new approach changes the weighting, so that half of the keys in each group are attributed to the beginning of the group and half to the end. Specifically, in the new approach, each server also includes its smallest and largest keys as pivots, and each pivot is annotated with a weight. The first and last pivots have a weight of 0.5 (half a group), and middle keys have a weight of 1.0 (half of the preceding group and half of the following group). The annotated pivots from all servers are collected and sorted as usual. Then the pivots are scanned from smallest to largest, adding up the pivot weights; when a given pivot is reached, the cumulative weight is an estimate of how many groups worth of data have keys less than or equal to the pivot. The ith splitter will be the first pivot encountered where the cumulative weight is at least $i \times sM$.

	120 servers (7000 records per server)							
	Uniform				Gaussian			
	Naive		Improved		Naive		Improved	
Pivots	P50	P90	P50	P90	P50	P90	P50	P90
120	33.6%	45.5%	6.9%	8.6%	37.6%	47.7%	6.9%	8.6%
240	23.5%	25.4%	4.4%	5.4%	23.4%	25.4%	4.4%	5.5%
360	15.9%	16.9%	3.1%	3.8%	15.9%	17.0%	3.2%	3.9%
480	11.5%	12.3%	2.5%	3.0%	11.6%	12.4%	2.5%	3.0%
600	9.6%	10.3%	2.0%	2.4%	9.7%	10.4%	2.0%	2.5%

Table 7: Excess records in the largest partition, relative to the average partition size, when using the improved splitter selection algorithm, vs. the naive algorithm. Input keys were drawn from two random distributions, and rows corresopnd to different numbers of pivots per server. "P50" and "P90" represent the median and 90th percentile over 1000 runs, respectively.

We evaluated the improved mechanism for splitter selection by running stand-alone simulations with keys drawn from random distributions. Table 7 shows that the improved mechanism ensures that the largest partition is within 10% of the ideal size. Without the improvement, the largest partition will be 30-50% larger than ideal if the number of pivots per server equals the number of servers. Said another way, using the new splitter selection mechanism provides a greater benefit than increasing the pivots per server by 5x.