Sorting 100 TB on Google Compute Engine

Michael Conley
University of California, San Diego
mconley@cs.ucsd.edu

Amin Vahdat
Google Inc.
University of California, San Diego
vahdat@cs.ucsd.edu

George Porter
University of California, San Diego
gmporter@cs.ucsd.edu

Abstract

Google Compute Engine offers a high-performance, cost-effective means for running I/O-intensive applications. This report details our experience running large-scale, high-performance sorting jobs on GCE. We run sort applications up to 100 TB in size on clusters of up to 299 VMs, and find that we are able to sort data at or near the hardware capabilities of the locally attached SSDs.

In particular, we sort 100 TB on 296 VMs in 915 seconds at a cost of $154.78. We compare this result to our previous sorting experience on Amazon Elastic Compute Cloud and find that Google Compute Engine can deliver similar levels of performance. Although individual EC2 VMs have higher levels of performance than GCE VMs, permitting significantly smaller cluster sizes on EC2, we find that the total dollar cost that the user pays on GCE is 48% less than the cost of running on EC2.

1. Introduction

Recent years have seen a rise in data-driven innovation. This trend, typically termed “Big Data” is pervasive both in industries, such as a health, retail, and technology, and also in sciences such as biology and astronomy. Business people, engineers, and researchers alike have shown interest in processing and analyzing large amounts of data.

The sheer volume of data to be processed presents its own technical challenge. Large workloads require increasingly more powerful clusters of machines to carry out data analysis in a reasonable period of time. Rather than building out large data centers of machines, companies and researchers often deploy applications to cloud computing services, which rent out large clusters at a fraction of the price of buying dedicated infrastructure.

Historically, cloud computing services, such as Amazon Web Services [1], Microsoft Azure [2], and Google Cloud Platform [6], have catered to web applications requiring the ability to elastically scale to meet user demands. A typical web retail application might increase the size of its cloud deployment around holidays to better handle peak loads. These applications are often latency-sensitive and are engineered to handle a target number of user queries per second.

In this work, we consider the cost and performance benefits of running a different class of applications, termed *I/O-intensive applications*, in the public cloud. These applications require processing enormous data sets and are geared more towards backend processing than user-facing frontend applications. By definition, the throughput of I/O-intensive applications is often limited by the speed of I/O components, including local storage, remote network-attached storage, network devices, fabrics, and topologies.

In particular, we run a sorting application on Google Compute Engine (GCE) on data sizes of up to 100 TB and clusters consisting of up to 299 virtual machines. We examine the performance and cost of these workloads and compare to our earlier experience running on Amazon Elastic Compute Cloud (EC2) [4, 5].

We find that although the per-VM performance is lower, requiring larger cluster sizes, GCE is capable of sorting data at nearly the speeds of EC2, but at much lower cost. In particular, we sort 100 TB on 296 VMs in 915 seconds at a cost of $154.78, which is 48% cheaper than on EC2.

2. Application Workload

We now describe our application workload. We first discuss the data set and problem statement, and then we briefly describe our sorting application framework.

2.1 Sort Benchmark

We choose to measure the performance and cost of the Indy 100 TB GraySort benchmark taken from the suite of Sort Benchmark applications [13]. These benchmarks are designed to stress test the I/O capabilities of large batch-processing systems by providing a workload that is larger than the aggregate memory of most clusters. Sorting is a well-understood problem, so the benchmark also represents a canonical workload for I/O-intensive applications. We have extensive experience with this workload [4, 10, 11], so it also provides a good way to compare performance and cost across cloud services and clusters.

The 100 TB data set consists of one trillion records, where each record is a 100-byte pair consisting of a 10-byte key and 90-byte value. Keys are uniformly distributed across the space of 256^{10} possible byte sequences. A generator

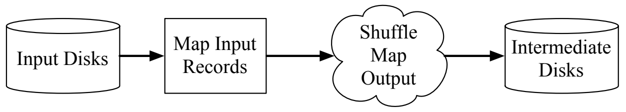

Figure 1: The map-and-shuffle phase. Figure reproduced from [5].

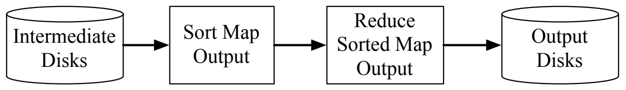

Figure 2: The sort-and-reduce phase. Figure reproduced from [5].

program distributed on the benchmark website guarantees the same deterministically "random" data set across runs. A similarly provided validation program checks that the output records are in sorted order and checksum to the appropriate value for any given data size.

The GraySort benchmark measures the absolute performance of sorting this 100 TB data set, and takes the form of an annual contest. Contest entries are permitted to use any commercially available hardware they can find. Winning entries range from companies running large data centers [7, 8] to academic research groups running small clusters or cloud deployments [4, 10, 11].

The Indy variant of the GraySort is a specialized benchmark with the fewest requirements on the sorting system. Systems that are custom built for sorting, or highly tuned for the particular uniformly distributed data set are permitted. Data replication is not required, and application failures are permitted. The benchmark is designed to measure the absolute fastest sorting system possible.

2.2 Themis MapReduce

We use Themis MapReduce [9] as our data processing framework to carry out the sort benchmark. Themis is a highly efficient MapReduce framework that we built for running I/O-intensive applications and is a good fit for the sort benchmark workload. Themis is derived from our earlier experience in high-performance sorting with TritonSort [12], and together these systems have created a total of ten world records in the sort benchmark contest [4, 10, 11].

Themis implements MapReduce as a two-pass pipelined application. The first pass, the *map-and-shuffle* phase, is illustrated in Figure 1. In this pass, each cluster node reads input records from disk, applies a map function, and then transfers map output records to other nodes in the cluster based on a partitioning function. These output records are stored in partition files on the receiving nodes' disks.

After all nodes complete map-and-shuffle, the *sort-and-reduce* phase runs on each node (Figure 2). In this pass, entire partition files containing map output records with the same key are loaded into memory. These files are sorted, and a reduce function is applied to each record before being written back to disk. The sort-and-reduce phase runs locally on each node, so no network transfer occurs in this pass.

In order to run the sort benchmark on Themis MapReduce, we simply have to run the full MapReduce job with identity map and reduce functions, which leave the data records unchanged. The globally sorted data set can then be represented as the concatenation of all the reduce output files, so long as an order-preserving, range partitioning function is used.

Themis MapReduce contains a sampling phase (not shown) that runs before the map-and-shuffle phase to determine the proper partitioning function. Because the Indy GraySort benchmark permits us to take advantage of the uniformly distributed data set, we can disable this sampling phase and run a custom partitioning function that range-partitions the data set into uniformly sized partitions. Some of the evaluations described in this report run with sampling enabled, and others with it disabled. It will be clearly noted, for each cluster, whether sampling is enabled or disabled.

3. Data Analysis Tools

We now briefly describe the tools we use to analyze performance bottlenecks.

3.1 System Resource Monitoring

We collect logs from the `sar`, `iostat`, and `vnStat` resource monitoring tools [14, 15]. We measure the CPU and memory utilization reported by `sar` every second during each phase of the sort. The `iostat` and `vnStat` tools measure the disk and network utilizations respectively, and we run each averaging over five second intervals.

3.2 Application Framework Logging

The Themis MapReduce framework logs information about the processing time, idle times, and blocked times of various threads and operations on each node. We use these time breakdowns to compute the fraction of time each thread spends performing useful work, waiting for new work, or waiting while blocked on resources or other threads.

The pipelined design employed by Themis makes it relatively easy to assign a root cause to a period of idleness. For example, a mapper thread may be idle because it runs faster than a downstream sender thread. In this case, user configurable limits prevent the mapper from getting too far ahead, causing it to block. The same thread might also be idle because it runs faster than an upstream reader thread. In this case, the mapper has periods of time with no records to process. The Themis logging infrastructure differentiates these cases, making analysis as straightforward as possible.

From these logs and the data sizes involved, we can compute the throughput of each thread in MB/s and compare across threads that perform compute-, memory-, disk-, or network-intensive operations. We can then determine which component of Themis is running at the slowest rate, and is therefore the bottleneck.

Virtual CPU Cores	32
Memory	120 GB
Persistent Disk	None
Cost	\$1.60 / hr
Local SSDs	4x 375 GB
SSD Cost	\$0.452 / hr
Total Cost	\$2.052 / hr

Table 1: The `n1-standard-32` VM with four local SSDs.

While resource bottlenecks can change over time, particularly in the network, we make the simplifying assumption that resource utilization is relatively constant during each phase of the sort job. Therefore, when we make claims that a particular phase is SSD-bound, for example, we mean that in that phase, the slowest component of Themis is the storage subsystem. This does not necessarily indicate that we are running at the full throughput of the underlying device. For example, changes in I/O patterns can cause a storage-bound system to run at sub-optimal speeds.

4. Cluster Configuration

Next we describe our cluster configurations on Google Compute Engine and the Themis MapReduce configuration for running on these clusters.

4.1 Google Compute Engine

For the purpose of this evaluation, we use the `n1-standard-32` virtual machine configured with no persistent disks and four locally attached SSDs (Table 1). We show the full price without sustained use because this represents the on-demand price of spinning up a cluster to run a particular job. In an actual deployment, sustained-use prices may reduce total costs.

Each cluster is configured with one master VM and a number of slave VMs. When we refer to a cluster of size N we mean N slave nodes and one master. The master VM runs a monitoring script that checks for node liveness. It also supports a web GUI for operating on the cluster, serves as a head node for examining logs, and launches jobs on nodes in the cluster. Slave nodes are responsible for actually carrying out the MapReduce logic.

Although we run all virtual machines in the `us-central1-f` zone, some of the experiments in this report are run on different days. In particular, we evaluate two clusters A and B for each cluster size. We run both A clusters on September 4, 2015, and both B clusters on September 9, 2015. We therefore have two days worth of data points, day A and day B, for each cluster size. It should also be noted that the A clusters run the sampling step described in Section 2.2.

4.2 Themis MapReduce

We configure Themis MapReduce for the Indy GraySort benchmark as follows. We allocate two of the four local SSDs for input and output files, and the remaining two SSDs for intermediate files. Each SSD is formatted with the `ext4` file system and mounted with the `noatime`, `discard`, and `data=writeback` options.

We configure read and write sizes to be 4 MiB and use the `O_DIRECT` flag to bypass the file buffer cache. We issue all reads and writes asynchronously using `libaio`, the native asynchronous I/O library for Linux. We issue two simultaneous asynchronous reads to each of the two reading SSDs in each phase. In the map-and-shuffle phase, we issue four simultaneous asynchronous writes to each of the two writing SSDs. However, in the sort-and-reduce phase, we only issue two simultaneous asynchronous writes to each of the two writing SSDs. These values were determined experimentally based on the I/O patterns of our MapReduce framework and the properties of the SSDs.

The shuffle component of Themis MapReduce runs an all-to-all network transfer across the cluster. Each of the N nodes maintains N sending and N receiving sockets, for a total of 2N open TCP connections per node. Themis also supports an option to open multiple parallel connections to nodes to increase network throughput for high speed networks. Using a factor of P parallel connections brings the total to 2PN open TCP connections per node. At small cluster sizes this is very reasonable even with P as high as 4, but at larger cluster sizes we can run into problems. We therefore set P to 1 in most of the clusters evaluated.

We run Themis on a Linux image derived from the Debian 7.8 image with backports kernel that is available on Google Compute Engine and built on August 18, 2015. This image runs Linux 3.16.0 and is stock except for a set of installed dependencies necessary to run Themis.

5. Mid-scale Evaluation

We now describe our "mid-scale" evaluation on clusters of 100 VMs, which is approximately one third the size necessary to run the 100 TB sort operation. Prior to running this evaluation, the largest measured cluster size was 15 VMs. We chose to evaluate 100 VMs before running the full 100 TB sort in order to incrementally assess the scaling properties of Google's cloud infrastructure, as opposed to making a 20x jump in scale all at once.

5.1 Cluster A-100

Cluster A-100 consists of 100 slave VMs in the `n1-standard-32` configuration described in Table 1. We use a `n1-standard-2` VM as the master with no local SSDs. This machine type is a much smaller VM that only has 2 virtual CPU cores and 7.5 GB of memory. As noted in Section 4, we use the `us-central1-f` zone.

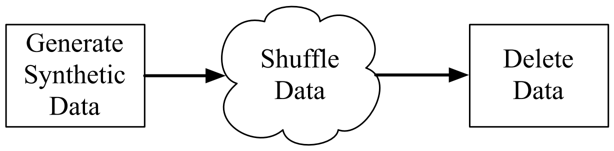

Figure 3: The *NetBench* application-level benchmark measures the network performance of the entire cluster with an all-to-all shuffle without involving storage devices. Figure reproduced from [5].

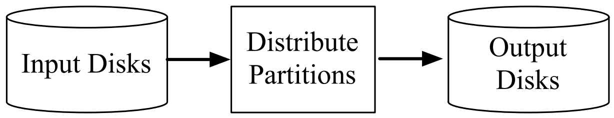

Figure 4: The *DiskBench* application-level benchmark measures the storage performance of individual cluster VMs locally without involving network transfer. Figure reproduced from [5].

5.1.1 A-100 Benchmarks

Before running the sort operation, we run the application-level benchmarks NetBench and DiskBench [5] to assess the network and storage properties of the cluster. These benchmarks, illustrated in Figures 3 and 4, give us an idea of how well Themis MapReduce ought to perform without actually running the full application logic.

We run NetBench configured with four parallel TCP connections per pair of nodes. The benchmark measures the network performance of the shuffle component of Themis MapReduce at a average throughput of 1531.76 MB/s, or 12.26 Gb/s, on cluster A-100. We report a standard deviation of 16.35 MB/s, although the all-to-all nature of this benchmark makes it difficult to tease apart straggler flows, which can block faster flows if enough data backs up in the benchmark pipeline.

Next we run DiskBench to assess the storage performance of the A-100 VMs. As described in Section 4, we configure the benchmark to read from two of the SSDs and write to the other two SSDs to more closely mirror the performance of Themis MapReduce. We use the same asynchronous I/O and I/O size configurations used in the map-and-shuffle phase. DiskBench reports an average read/write throughput of 810.05 MB/s and a standard deviation of 3.97 MB/s, indicating that each SSD is capable of writing at approximately 400 MB/s, and that performance is relatively consistent across VMs.

5.1.2 Sorting on A-100

Next, we run a sort operation over 33.3 TB of data divided evenly across the 100 VMs in the cluster. While not strictly necessary given the rules of the Indy GraySort benchmark, we enable sampling for this sort job to test the throughput of the default mode of operation for our MapReduce framework.

Time (s)	
Sampling and Overhead	32
Map and Shuffle	468
Sort and Reduce	430
Total	930
Bottlenecks	
Map and Shuffle	Mappers
Sort and Reduce	SSD Writes
Cost	
VM Cost	$2.052/hr
Cluster Size	100
Master VM Cost	$0.10/hr
Total	$53.04

Table 2: Sorting results on the A-100 cluster.

The sort job completes in 930 seconds. About 468 seconds are spent on map-and-shuffle and 430 seconds on sort-and-reduce, with the remaining 32 seconds spent on sampling and coordination overhead. Our log analysis tools show that the map-and-shuffle phase is CPU-bound by the mapper threads, and the sort-and-reduce phase is SSD-bound by the writer threads.

In a perfect world, both the map-and-shuffle and sort-and-reduce phases would be bound by the write speed of the SSDs, yielding a per-VM throughput of 800 MB/s in each phase. In this particular sort job, we report per-VM throughputs of 712 MB/s and 774 MB/s in the map-and-shuffle and sort-and-reduce phases respectively.

From log analysis, we can observe that the throughput of the mapper threads varies wildly from VM to VM. The fastest VM fully maps all its data in about 378 seconds, while the slowest requires 453 seconds. Our logs report that in both of these cases, the mapper threads are almost never idle, indicating that they are running as fast as the system can support, i.e. they are CPU-bound, or perhaps memory-bound due to caching effects. For reference, the mean map completion time is about 401 seconds, and the standard deviation is 13 seconds.

In the sort-and-reduce phase, it is instructive to look at the observed processing throughput of writers. This metric is defined to be the throughput in MB/s of the writer threads from the moment they see the first data buffer to be written to the moment they finish writing the last data buffer. This metric ignores the first several seconds of the sort-and-reduce phase, while the pipeline is filling up and writers are idle waiting for their first piece of work. Using this metric, we see the fastest VM writes at a speed of 814 MB/s and the slowest at a speed of 787 MB/s. These speeds are in-line with the maximum SSD write speeds measured by DiskBench, so we conclude this phase is SSD-bound by the writer threads.

The results from the A-100 evaluation are shown in Table 2. We report a total cost of about $53 to sort the 33.3 TB data set. This suggests an approximate cost of $160 for the full 100 TB sort under the assumption of perfect scalability.

5.2 Cluster B-100

We run the B-100 cluster a few days later on a different set of 100 slave VMs in the same `us-central1-f` zone. This time we use `n1-standard-32` as our master VM type, again with no local disks. We upgrade the master VM in order to provide faster log access due to CPU limitations accessing Google Cloud Storage with only 2 cores, as we will describe in Section 8.1.

Our goal with cluster B-100 is solely to tweak the performance of the map-and-shuffle phase of the sort because we are already SSD-bound in the sort-and-reduce phase in cluster A-100. We therefore only run partial sort operations on this cluster.

5.2.1 B-100 Benchmarks

As before, we run NetBench and DiskBench to assess the performance of the cluster before running the sort operation. This time, we configure NetBench to use a single TCP connection per pair of hosts. We measure an average network bandwidth of 1654.82 MB/s, or 13.2 Gb/s, with a standard deviation of 7.70 MB/s. DiskBench reports an average read/write bandwidth of 813.54 MB/s, with a standard deviation of 3.37 MB/s. These measurements peg cluster B-100 at roughly the same storage performance as cluster A-100, although with slightly more network bandwidth.

5.2.2 Sorting on B-100

As mentioned above, we are interested in tweaking the performance of map-and-shuffle. We therefore do not run sort-and-reduce on this cluster.

No Sampling We begin by running map-and-shuffle *without* sampling on the 33.3 TB data set. Unlike the map-and-shuffle phase in cluster A-100, this operation uses a simpler partitioning function that assumes the uniform data distribution of Indy GraySort. We report an execution time of 431 seconds for map-and-shuffle, which yields a per-VM throughput of 774 MB/s. Log analysis shows the writer threads write to the SSDs at an average speed of 798 MB/s/VM, which is essentially optimal. Further analysis shows the mapper threads are idle waiting for records to process approximately 16% of the time, indicating that mappers are no longer a bottleneck in this configuration.

Sampling We now consider the effect of enabling sampling to create a configuration that is closer to the A-100 cluster. We report a sampling time of 32 seconds and a map-and-shuffle time of 438 seconds, which is slightly worse than disabling sampling. Log analysis shows that mapper threads in this configuration are actually a bottleneck again, causing writer threads to run at average speed of 778 MB/s/VM. This speed is slightly slower than optimal, although the difference is not as pronounced as in the A-100 cluster.

Five Mappers A CPU-intensive thread bottlenecking the job, such as a mapper thread, can often be remedied by increasing the number of threads up to the available number of free CPU cores in the system. We therefore increase the number of mapper threads from four to five, and re-run the job. This time we report a sampling time of 33 seconds, and a map-and-shuffle time of 433 seconds. Log analysis shows that mappers are no longer a bottleneck, with an average idle time of about 13%.

A summary of the results of the B-100 evaluation is shown in Table 3. The results from clusters A-100 and B-100 show that Themis MapReduce is capable of running a 33.3 TB sort on 100 VMs at full speed, namely approximately 800 MB/s/VM as dictated by the available SSD write bandwidth.

6. Large-scale Evaluation

Armed with the knowledge and experience of running mid-scale clusters of 100 VMs, we now turn our attention to the main focus of this work, which is running 100 TB sort operations on approximately 300 VMs. As before, we evaluate two different clusters, A-299 and B-299, which are run on the same days as the A-100 and B-100 clusters respectively.

6.1 Cluster A-299

We launch cluster A-299 with 299 slave `n1-standard-32` VMs and a single `n1-standard-2` master VM, again in the `us-central1-f` zone.

6.1.1 A-299 Benchmarks

We run NetBench and DiskBench on the cluster to get baseline bandwidths of the network and storage devices. We run NetBench with a single TCP connection per pair of hosts and report an all-to-all average network bandwidth of 1216.59 MB/s, or 9.7 Gb/s, with a standard deviation of 4.94 MB/s. DiskBench measures the read/write storage bandwidth to be 801.84 MB/s on average, with a standard deviation of 7.81 MB/s.

6.1.2 Sorting on A-299

Four Mappers We run the 100 TB sort operation with sampling and four mapper threads. The job completes in 976 seconds, about 45 of which consist of sampling and framework overheads. The map-and-shuffle phase takes 497 seconds, while the sort-and-reduce takes 434 seconds, yielding per-VM phase throughputs of 672 MB/s and 771 MB/s respectively.

Log analysis shows that the map-and-shuffle phase is, as in the A-100 case, bound by the speed of the mapper threads. The average mapper idle time is less than 1%, although there is significant variability between the speeds of the different VMs. We observe a mean mapper completion time of

Configuration	Map and Shuffle	Mapper Idle	SSD Write	Bottleneck
No Sampling, 4 Mappers	430s	16%	798 MB/s	SSD Writes
Sampling, 4 Mappers	438s	1%	778 MB/s	Mappers
Sampling, 5 Mappers	433s	13%	793 MB/s	SSD Writes

Table 3: Sorting results on the B-100 cluster.

432.8 seconds and a standard deviation of 10.9 seconds. The fastest VM finishes mapping its data in about 412 seconds, while the slowest requires 483 seconds. Writer threads remain idle approximately 12% of the time, indicating that there is room for improvement if mapper performance can be increased.

The sort-and-reduce phase is bound by the write speed of the SSDs. VMs write to their SSDs at an average of 800.4 MB/s, with a standard deviation of 5.3 MB/s. The fastest VM writes at 812 MB/s, while the slowest writes at 785 MB/s. These measurements suggest that the sort-and-reduce phase is running at the speed of the underlying hardware, and cannot be improved.

Five Mappers We repeat the sort experiment with five mapper threads in order to alleviate the bottleneck at the mappers. This sort job completes in 976 seconds, with 47 seconds for sampling and framework overheads, 484 seconds for map-and-shuffle, and 445 seconds for sort-and-reduce.

The map-and-shuffle phase is still bound by the speed of the mapper threads, which have an average idle time of less than 1%. The average map completion time is 420.9 seconds, with a standard deviation of 16.5 second, minimum of 384 seconds and maximum of 468 seconds. We observe that while this is slightly faster than the map time in the first sort job, the average speed of each individual mapper thread is about 19% slower, indicating that there is an underlying resource contention issue, perhaps in the CPU, memory, or cache.

This time in the sort-and-reduce phase we measure an average writer throughput of 785.6 MB/s, with a standard deviation of 6.8 MB/s, minimum of 769 MB/s, and maximum of 804 MB/s. Log analysis shows that, although writes are slower, writers are still the bottleneck.

It is unclear exactly why writes are slower compared to the first sort operation. Examining logs from `iostat` shows a slight difference in the average queue size and begin-to-end I/O service time between the runs, which could account for the approximately 2% difference in total write throughput.

The fact that writes are slower across the board for all VMs suggests that this slowdown might have something to do with the flash itself. While we do mount the `ext4` filesystem with the `discard` option, the various layers of virtualization do not give us any guarantee as to what exactly is going on at the physical layer in the flash. Because intermediate and output files have to be erased and re-written between runs, repeating the 100 TB sort can alter the performance properties of the underlying flash devices in unpredictable ways.

	Time (s)	
	Four Mappers	Five Mappers
Sampling	45	47
Map and Shuffle	497	484
Sort and Reduce	434	445
Total	976	976
	Bottlenecks	
Map and Shuffle	Mappers	Mappers
Sort and Reduce	SSD Writes	SSD Writes
	Cost	
VM Cost	\$2.052/hr	\$2.052/hr
Cluster Size	299	299
Master VM Cost	\$0.10/hr	\$0.10/hr
Total	\$166.32	\$166.32

Table 4: Sorting results on the A-299 cluster.

We summarize the results of the A-299 sort operations in Table 4. In particular, both configurations sort 100 TB in about 976 seconds on 299 VMs, resulting in a total cost of \$166.32.

6.2 Cluster B-299

The B-299 cluster consists of 299 slave `n1-standard-32` VMs and a master node, which is also `n1-standard-32`. Again, we launch all VMs in the `us-central1-f` zone. We run many different sorting configurations on the cluster, although a large number are omitted for brevity.

6.2.1 B-299 Benchmarks

Again we run NetBench and DiskBench before beginning our sort operations. NetBench, configured with a single TCP connection per pair of hosts, reports an average network bandwidth of 1237.19 MB/s, or 9.9 Gb/s, with a standard deviation of 11.44 MB/s. DiskBench reports a read/write storage bandwidth average of 811.01 MB/s, with a standard deviation of 3.83 MB/s.

6.2.2 Sorting on B-299

Baseline We first measure the performance of the map-and-shuffle phase of a 100 TB sort without sampling in order to see if it is mapper-bound or SSD-bound. Map-and-shuffle completes in 475 seconds and is mapper-bound on at least some of the VMs. The average map completion time is

416.2 seconds with a standard deviation of 7.7 seconds, minimum of 402 seconds, and maximum of 459 seconds.

From the logs, we find that mappers do not spend any time waiting on upstream reader threads. While there is a small amount of time spent waiting on downstream sender threads, we note that the slower VMs spend very little mapper time idle. This suggests that improving mapper performance will increase overall cluster performance.

Core Offlining We next try turning a CPU core offline to see if yielding a core to the hypervisor improves performance. This configuration completes the map-and-shuffle phase in about 520 seconds, which is substantially slower.

The mapper threads complete with an average time of 439.7 seconds and standard deviation of 31.8 seconds. The fastest VM completes in 381 seconds and the slowest takes 505 seconds. Log analysis shows that mappers spend almost no time idle, indicating that the system is mapper-bound on all VMs, although there is now significant variability between VMs. Additionally, while some VMs now run faster, the vast majority run far slower, indicating that core offlining reduces the performance of sorting with Themis MapReduce in this particular configuration.

Eight Mapper Threads The experiments on cluster B-100 and A-299 suggest that increasing the number of mapper threads can improve performance. While we did not observe a significant performance boost increasing the number of mappers from four to five, we did notice a dramatic shift in cluster performance using eight mapper threads per VM.

Running the map-and-shuffle phase with eight mapper threads requires 497 seconds, which is still slower than our baseline measurement with four mapper threads. However, this configuration exhibits very different mapper properties. In particular, mappers now spend an average of 38% of their runtime waiting on downstream stages. This suggests that we are no longer mapper bound. Further log analysis suggests that we may in fact be network-bound in this particular configuration, although likely by the speed of our network processing threads and not by the speed of the underlying network, which is capable of roughly 10 Gb/s all-to-all as measured by NetBench.

In fact, further investigation suggests we may not be running an optimal configuration for a cluster of this size. Mapper output buffers are configured to be 2 MiB in size, and mappers are only allowed to get at most 1 GB ahead of the sender threads, or equivalently 476 map output buffers. Given that each VM is transferring data buffers to 299 VMs, it may be the case that some connections do not have data available when the sender threads need to send data.

Compounding this problem is the fact that each mapper thread maintains a very large number of buffers that must fill up before a network transfer can be performed, essentially introducing unnecessary burstiness, which may further reduce performance.

Better Map Output Memory Utilization To address these issues, we keep the eight mapper threads, but reconfigure Themis to use 256 KiB map output buffers, rather than the 2 MiB buffers used above. In addition, we increase the slack between the mapper threads and sender threads by allowing mappers to get at most 3 GB ahead of senders, as opposed to 1 GB. We make these changes with the intention of more evenly spreading the map output data across the available network connections.

This particular configuration runs the map-and-shuffle phase in 457 seconds, which is a marked improvement from the previous attempts. We again find that mappers spend a significant amount of time waiting for downstream sender threads, but this time the total time taken to execute the map is significantly shorter.

In fact, this configuration is storage-bound by the write speed of the SSDs on at least some of the slower VMs. The slowest VM takes 455 seconds to write all of its data, and its writer is never idle, indicating that this VM is writing at the maximum speed permitted by its SSDs under the given workload. This particular VM writes at a speed of 747 MB/s, or 374 MB/s per SSD, which is within about 5% of optimal.

The average VM's writer threads spend about 3% of their time idle, indicating that variance in the ability of each VM to write this particular workload to its SSDs causes some VMs to perform the task faster, and others slower. We note that although our benchmarks report very little storage variance, the benchmark writes a much smaller amount of data and has no interference from other application threads, such as the shuffle component of Themis or the map and partitioning logic. Further analysis is needed to pin down which of these factors causes Themis MapReduce to perform slightly worse than the benchmark applications.

Putting it Together We now have enough information to run the full 100 TB sort at near-optimal speeds. Before running the full sort, we remove three VMs from the cluster due to errors described in in Section 8.2. Therefore we run the full 100 TB sort on 296 VMs, rather than on all 299 VMs.

The sort completes in 915 seconds, with 469 seconds for the map-and-shuffle phase, 444 seconds for the sort-and-reduce phase, and 2 seconds for framework overhead.

As before, we find that the slowest VMs are SSD-bound by the writer threads in the map-and-shuffle phase, with the slowest writing at a total speed of 740 MB/s, or 370 MB/s per SSD.

The sort-and-reduce phase is also SSD-bound by the writer threads for all VMs. Here too we find some variability between VM write speeds, with the fastest VM finishing its write workload in 421 seconds, and the slowest in 444 seconds, which is about 386 MB/s per SSD.

The results of these evaluations are summarized in Tables 5 and 6. We are able to sort 100 TB of data at a total cost of $154.78.

Configuration	Map and Shuffle	Bottleneck
Baseline	475s	Mappers
Core Offline	520s	Mappers
Eight Mappers	497s	Network
Better Map Memory	457s	SSD Writes

Table 5: Map-and-shuffle performance on the B-299 cluster.

100 TB Sort	
Overhead	2s
Map and Shuffle	469s
Sort and Reduce	444s
Total	915s
Bottlenecks	
Map and Shuffle	SSD Writes
Sort and Reduce	SSD Writes
Cost	
VM Cost	$2.052/hr
Cluster Size	296
Master VM Cost	$1.60/hr
Total	$154.78

Table 6: Sorting results on the B-299 cluster.

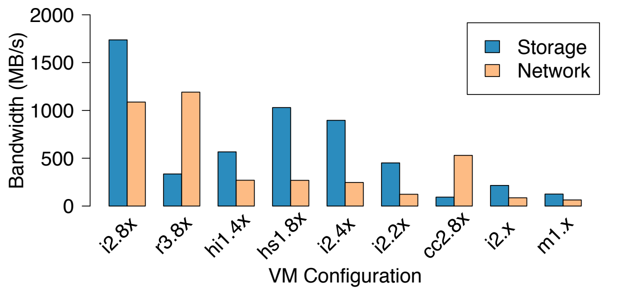

Figure 5: Performance of DiskBench and NetBench on a subset of EC2 VMs. Figure reproduced from [5].

7. Comparison with Amazon EC2

We now discuss how our experiences on Google Compute Engine compare to our experiences on Amazon Elastic Compute Cloud, and analyze the cost and performance tradeoffs of each service.

7.1 Cost and Performance of Sorting on EC2

In previous work, we performed a comprehensive analysis of the storage and networking properties of VMs on Amazon Elastic Compute Cloud (EC2) in order to determine time- and cost-efficient configurations for sorting data on EC2 [5]. We now present some of the highlights of this work.

As part of the study, we run DiskBench and NetBench on a variety of VMs on EC2. A subset of those results are shown in Figure 5.

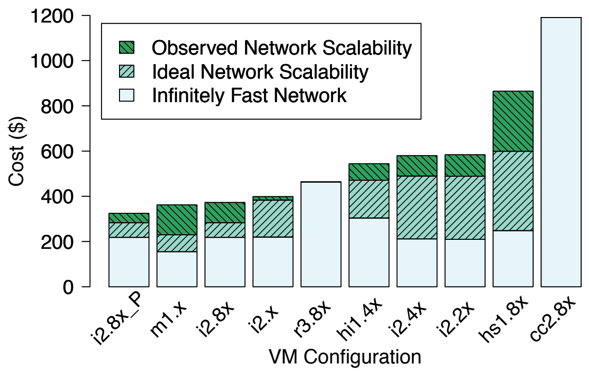

Figure 6: Predicted dollar cost of sorting 100 TB under various network assumptions on EC2. Figure reproduced from [5].

Interestingly, we find that on EC2, most VMs have networks that are far slower than their storage devices, unlike the GCE VMs measured in this report thus far. We also find that the fastest VM, `i2.8xlarge`, has storage devices capable of read/write bandwidths in excess of 1700 MB/s, which is more than twice as fast as the local SSDs on GCE. However, the network capabilities of this VM limit the maximum throughput of the map-and-shuffle phase to the slower network speed of almost 1100 MB/s.

Based on these observations, we predict the total dollar cost of running a 100 TB sort on various EC2 VMs. We make this prediction under a variety of network assumptions. We first assume the network is not a bottleneck, meaning that the performance, and therefore the cost, of the sort is limited only the available storage bandwidth (infinitely fast network). Next, we update the prediction with NetBench measurements made a small scale, and assume that the network scales linearly, which is the ideal case. Finally, we update the prediction again using NetBench measurements observed at a much larger scale. These cost predictions are shown in Figure 6.

From the figure, we see that the predicted cost to sort 100 TB on EC2 VMs is under $400 for a handful of VMs, and is just over $300 for the cheapest VM type. In fact, one of the evaluations in the EC2 study [5], which was also our Indy GraySort submission for 2014 [4], pegged the actual cost of a 100 TB sort on `i2.8xlarge` at $299.45. This sort completed in 888 seconds on 178 VMs.

7.2 Cost and Performance of Sorting on GCE

Prior to the evaluations in this report, we performed a similar measurement of the storage and networking capabilities of the VMs on Google Compute Engine [3]. While this study was not as comprehensive as the study on EC2, we still learned a great deal about the general performance and cost properties of GCE.

In particular, many of the VMs on GCE have networking bandwidths far greater than the available storage band-

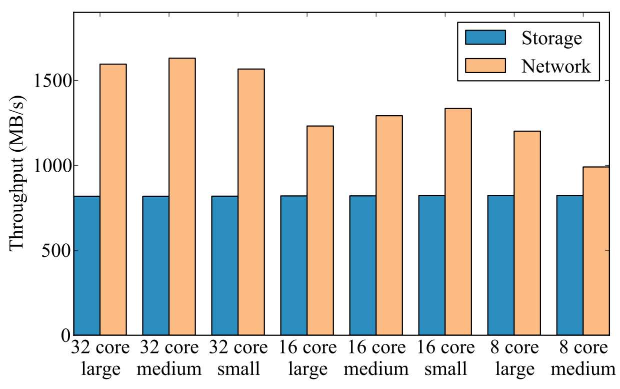

Figure 7: Storage and network bandwidths on a subset of GCE VMs. Figure reproduced from [3].

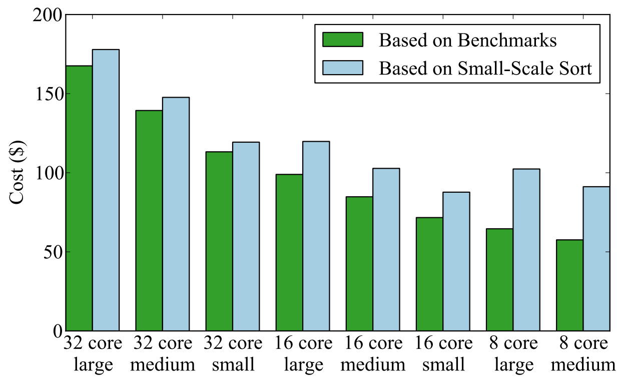

Figure 8: Predicted cost of a 100 TB sort on a subset of GCE VMs. Figure reproduced from [3].

width of the local SSDs. Figure 7 shows the measured storage and network bandwidths on a subset of the GCE VMs. In all cases, local SSD read/write storage bandwidth is approximately 800 MB/s, but network speeds are far greater and vary by VM type.

In the referenced study, we also run small-scale sort jobs on each VM type, and predict the cost of a 100 TB using data from 1) the benchmarks alone, and 2) the small-scale 1.2 TB sort operation on 10 VMs. These cost predictions are shown in Figure 8.

In all cases, the predicted cost is less than $180. In particular, the `n1-standard-32` VM evaluated in this report, which is represented as "32 core medium" in the figure, has a predicted 100 TB sort cost of $147.64 based on a small-scale sort job, and $139.33 based on benchmarks alone.

In fact, in Section 6.2, we measure the actual cost of a 100 TB sort operation on `n1-standard-32` to be $154.78, which is $7.14, or 5%, more expensive than the predicted cost based on the 1.2 TB sort job on 10 VMs.

	EC2	GCE
VM Type	`i2.8xlarge`	`n1-standard-32`
Cluster Size	178	296
Sort Time	888s	915s
Sort Cost	$299.45	$154.78

Table 7: Comparison of 100 TB Indy GraySort results on EC2 and GCE.

7.3 Sorting 100 TB on EC2 vs. GCE

Table 7 shows a side-by-side comparison of our Indy GraySort evaluation on both cloud services. We observe that EC2 is slightly faster, and permits significantly smaller cluster sizes, but that GCE is about half as expensive in terms of total dollar cost.

8. Discussion

We now discuss some scalability concerns that cropped up in the course of this work, and also some issues with the local SSDs on GCE.

8.1 Scalability and Themis MapReduce

Modern large-scale data processing frameworks, such as Hadoop [16], are well-suited to handling large cluster sizes. Themis MapReduce was designed for smaller cluster sizes, and the evaluations in this report stressed both the MapReduce binary and the cluster infrastructure scripts in unpredictable ways.

We first note that the all-to-all shuffle component of Themis, which opens TCP connections to every node in the cluster, experiences performance loss at the scale of a few hundred VMs. Performance could potentially be improved by implementing a more complicated shuffle that uses coordination to reduce the number of concurrent TCP connections, but further evaluation is needed.

Next, we note that the Themis infrastructure is not well suited for handling clusters of hundreds of nodes. Executing operations and changing configurations of the entire cluster requires minutes, so a more efficient coordination mechanism is needed.

Finally, we observe significant performance loss accessing Google Cloud Storage at a large scale. Themis stores configuration and log files locally on each VM, but the infrastructure supports permanently saving these files on cloud-based storage services. Uploading or downloading a large number of large log files from these services can take minutes. In particular, we observed the performance of the `gsutil rsync` command on Google Cloud Storage to be very slow on lower power VMs, such as the `n1-standard-2` master VM in some of our earlier evaluations. For this reason, we switched to `n1-standard-32` as the master VM in the later evaluations.

8.2 Local SSD Errors

We encountered several errors with the local SSDs during our evaluations and the work that preceded them. We now briefly describe the issues we encountered and how we dealt each.

4K Zero Blocks Initial evaluation of GCE revealed a bug with the local SSDs where a particular write configuration would cause 4 KiB blocks of data towards the end of certain files to be written as all `0`'s with high probability. The particular configuration that triggered this issue was writing 4 MiB chunks of data with `O_DIRECT` using the `libaio` asynchronous I/O library to files pre-allocated with `fallocate()` on the `XFS` files system.

An initial solution was disabling the `fallocate()` call, which caused all data to be written properly. However, we later found that switching from `XFS` to `ext4` also solved the issue, so we decided to use `ext4` in the evaluations in this report.

Capacity Errors When preparing for the 100 TB sort evaluation, we noticed that the SSDs would sometimes report device-out-of-capacity errors under our write workload when the SSDs reached 60%-80% capacity. While the SSD documentation suggests that some space needs to be set aside as reserved, we felt that this much waste was unnecessary, and made large-scale evaluation difficult due to resource allocation issues and user quotas.

We found that switching from the `XFS` file system to `ext4` resolved the issue, and permitted filling the devices up to 99% capacity without errors. However, DiskBench reports lower read/write storage throughput on `ext4` than on `XFS` with an otherwise identical configuration. We found that enabling `fallocate()` resolved this performance issue, and did not cause the zero block issue described above to occur on `ext4`.

Device I/O Errors While running the B-299 cluster we encountered two VMs that returned a device I/O error when reading intermediate files in the sort-and-reduce phase. We also encountered a third VM that had its disks unmounted during the course of the sort. These three faulty VMs were removed from the cluster before running the final 100 TB sort on B-299, which used the remaining 296 VMs.

9. Conclusion

Google Compute Engine provides a cheap and efficient way to run large-scale I/O-intensive applications, such as sorting. In this report, we evaluate several mid- and large-scale clusters on GCE, and sort 100 TB of data in 915 seconds at a cost of $154.78 on 296 VMs, which is 48% cheaper than our previous experience sorting 100 TB on Amazon EC2.

We further show that, with significant application tuning, it is possible to run the sort job in a way that is storage-bound by the write speeds of local SSDs on GCE, meaning that we are running at or near the maximum available speeds of the underlying server hardware.

Acknowledgments

We wish to thank David Wetherall, Tony Voellm, Paul Newson, Ivan Filho, Kirill Tropin, Dave Hiniker-Roosa, Scott Trump, Tino Tereshko, Henry Robertson, Scott Van Woudenberg, Sophia Yang, Al Hamel, Stanley Feng, and Mohsin Imam from Google for their help in running the evaluations presented in this work.

This work was supported in part by UCSD's Center for Networked Systems, the AWS Educational Grant program, National Science Foundation (CNS-1116079, CNS-1314921), and a Google Focused Research Award.

References

[1] Amazon Web Services. `http://aws.amazon.com/`.

[2] Microsoft Azure. `http://azure.microsoft.com/`.

[3] M. Conley. Achieving efficient I/O with high-performance data center technologies, 2015. ProQuest Dissertations and Theses. Order no. 3712210.

[4] M. Conley, A. Vahdat, and G. Porter. TritonSort 2014. `http://sortbenchmark.org/TritonSort2014.pdf`.

[5] M. Conley, A. Vahdat, and G. Porter. Achieving cost-efficient, data-intensive computing in the cloud. In *ACM SoCC*, 2015.

[6] Google Cloud Platform. `http://cloud.google.com/`.

[7] T. Graves. GraySort and MinuteSort at Yahoo on Hadoop 0.23. `http://sortbenchmark.org/Yahoo2013Sort.pdf`.

[8] D. Jiang. Indy Gray Sort and Indy Minute Sort. `http://sortbenchmark.org/BaiduSort2014.pdf`.

[9] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat. Themis: An I/O efficient MapReduce. In *ACM SoCC*, 2012.

[10] A. Rasmussen, M. Conley, G. Porter, and A. Vahdat. TritonSort 2011. `http://sortbenchmark.org/2011_06_tritonsort.pdf`.

[11] A. Rasmussen, H. V. Madhyastha, R. N. Mysore, M. Conley, A. Pucher, G. Porter, and A. Vahdat. TritonSort. `http://sortbenchmark.org/tritonsort_2010_May_15.pdf`.

[12] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A balanced large-scale sorting system. In *NSDI*, 2011.

[13] Sort Benchmark. `http://sortbenchmark.org/`.

[14] SYSSTAT. `http://sebastien.godard.pagesperso-orange.fr/`.

[15] vnStat - a network traffic monitor for Linux and BSD. `http://humdi.net/vnstat/`.

[16] Apache Hadoop. `http://hadoop.apache.org/`.