DOI:10.1145/2719919

Article development led by acmqueue
queue.acm.org

The perpetual motion of parallel performance.

BY NEIL J. GUNTHER, PAUL PUGLIA, AND KRISTOFER TOMASETTE

Hadoop Superlinear Scalability

"WE OFTEN SEE more than 100% speedup efficiency!" came the rejoinder to the innocent reminder that you cannot have more than 100% of anything. This was just the first volley from software engineers during a presentation on how to quantify computer-system scalability in terms of the speedup metric. In different venues, on subsequent occasions, that retort seemed to grow into a veritable chorus that not only was superlinear speedup commonly observed, but also the model used to quantify scalability for the past 20 years—Universal Scalability Law (USL)—failed when applied to superlinear speedup data.

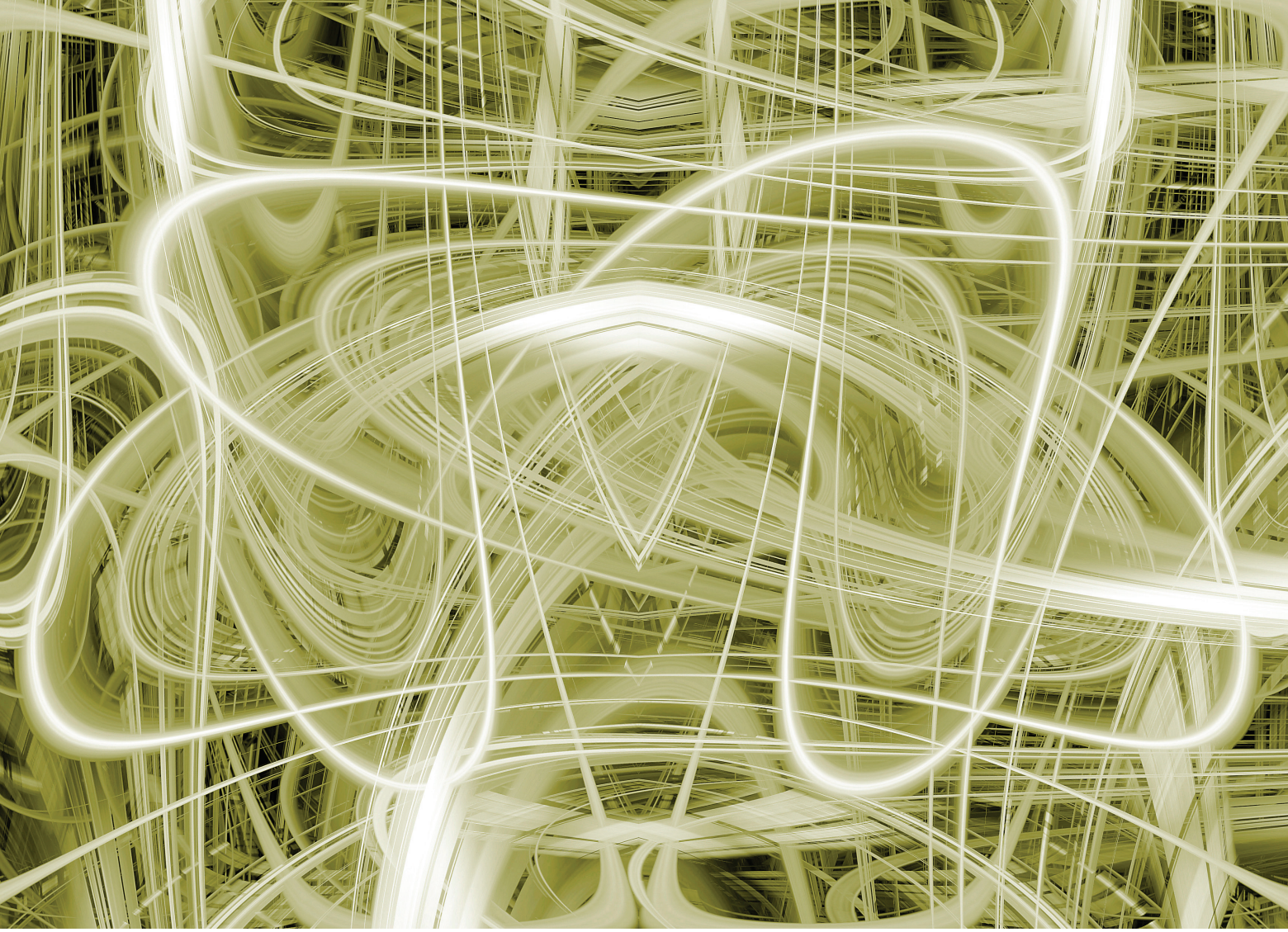
Indeed, superlinear speedup is a bona fide phenomenon that can be expected to appear more frequently in practice as new applications are deployed onto distributed architectures. As demonstrated here using Hadoop MapReduce, however, the USL is not only capable of accommodating superlinear speedup in a surprisingly simple way, it also reveals that superlinearity, although alluring, is as illusory as perpetual motion.

To elaborate, Figure 1 shows conceptually that linear speedup (the dashed line) is the best you can ordinarily expect to achieve when scaling an application. Linear means you get equal bang for your capacity buck because the available capacity is being consumed at 100% efficiency. More commonly, however, some of that capacity is consumed by various forms of overhead (red area). That corresponds to a growing loss of available capacity for the application, so it scales in a sublinear fashion (red curve). Superlinear speedup (blue curve), on the other hand, seems to arise from some kind of hidden capacity boost (green area).

As we will demonstrate, superlinearity is a genuinely measurable effect,[4,12,14,19,20,21–23] so it is important to understand exactly what it represents in order to address it when sizing distributed systems for scalability. As far as we are aware, this has not been done before.

Measurability notwithstanding, superlinearity is reminiscent of *perpetuum mobile* claims. What makes a perpetual motion machine attractive is its supposed ability to produce more work or energy than it consumes. In the case of computer performance, superlinearity is tantamount to speedup that exceeds the computer capacity available to support it. More importantly for this discussion, when it comes to perpetual motion machines, the difficult part is not deciding if the claim violates the conservation of energy law; the difficult part is *debugging* the machine to find the flaw in the logic. Sometimes that endeavor can even prove fatal.[5]

If, prima facie, superlinearity is akin to perpetual motion, why would some software engineers be proclaiming its ubiquity rather than debugging it? That kind of exuberance comes from an overabundance of trust in performance data. To be fair, that misplaced trust likely derives from the way performance data is typically presented without any indication of measurement error. No open source or commercial performance tools of which we are aware display measurement errors,



even though all measurements contain errors. Put simply, all measurements are "wrong" by definition: the only question is, how much "wrongness" can be tolerated? That question cannot be answered without quantifying measurement error. (Later in this article, Table 2 quantifies Hadoop measurement errors.)

In addition to determining measurement errors, all performance data should be assessed within the context of a validation method. One such method is a performance model. In the context of superlinear speedup, the USL[6–11,17,18] fulfills that role in a relatively simple way.

Universal Scalability Model

To quantify scalability more formally, we first define the empirical speedup metric in equation 1:

$$S_p = \frac{T_1}{T_p} \quad (1)$$

where T_p is the measured runtime on p = 1, 2, 3,... processors or cluster nodes.[14] Since the multinode runtime T_p is expected to be shorter than the single-node runtime T_1, the speedup is generally a discrete concave function of p. The following special cases can be identified.

▸ *Linear scalability.* If $T_p = T_1/p$ for each cluster configuration, then the speedup will have values S_p = 1, 2, 3, ... for each p, respectively. The speedup function exhibits linear scalability (the dashed line in Figure 1).

▸ *Sublinear scalability.* If $T_p > T_1/p$ for each cluster configuration, then successive speedup values will be *inferior* (red curve) to the linear scalability bound in Figure 1. For example, if $p = 2$ and $T_2 = 3T_1/4$, then $S_2 = 1.33$. Since it is less than $S_2 = 2$, the speedup is sublinear. The red curve is the most common form of scalability observed on both monolithic and distributed systems.

▸ *Superlinear scalability.* If $T_p < T_1/p$, then successive speedup values will be superior (blue curve) to the linear bound in Figure 1. For example, if p = 2 and $T_2 = T_1/3$, then $S_2 = 3$, which is greater than linear speedup.

The scalability of any computer system can be validated by comparing the measured speedup in equation 1 with the theoretically expected speedup, defined here.

Components of scalability. Scalability, treated as an aggregation of computer hardware and software, can be thought of as resulting from several physical factors:

1. Ideal parallelism or maximal concurrency.
2. Contention for shared resources.
3. Saturation resulting from the primary bottleneck resource.
4. Exchange of data between nonlocal resources to reach consistency or data coherency.

This does not yet take superlinearity into consideration. The individual effect of these factors on scalability, as measured by the speedup metric in equation 1, is shown schematically in Figure 2.

Each of these scaling effects can also be represented as separate terms in an analytic performance model: the

IMAGE BY STEVE BALL

USL. In our usual notation,[7,8] we write the theoretical speedup in equation 2 as:

$$S_p = \frac{p}{1 + \sigma p + \kappa p\,(p-1)} \qquad (2)$$

where the coefficient σ represents the degree of contention in the system, and the coefficient κ represents the incoherency in distributed data.

The *contention* term in equation 2 grows linearly with the number of cluster nodes, p, since it represents the cost of waiting for a shared resource such as message queuing. The coherency term grows quadratically with p because it represents the cost of making distributed data consistent (or coherent) via a pairwise exchange between distributed resources (for example, processor caches).

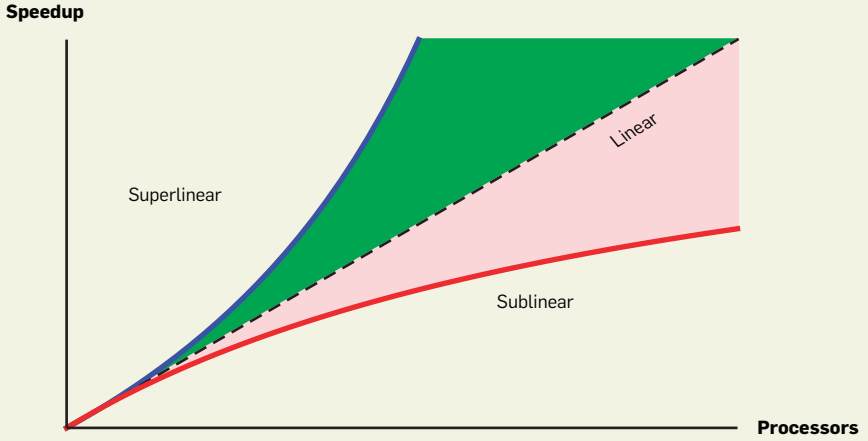
Interpreting the coefficients. If $\sigma = 0$ and $\kappa = 0$, then the speedup simply reduces to $S_p = p$, which corresponds to Figure 2a. If $\sigma > 0$, the speedup starts to fall away from linear (Figure 2b), even when the node configuration is relatively small. As the number of nodes continues to grow, the speedup approaches the ceiling, $S_{ceiling} = \sigma^{-1}$, indicated by the horizontal dashed line in Figure 2c. The two triangles in Figure 2c indicate this is a region of diminishing returns, since both triangles have the same width but the right triangle has less vertical gain than the left triangle.

If $\kappa > 0$, the speedup will eventually degrade like p^{-1}. The continuous scalability curve must therefore pass through a maximum or peak value, as in Figure 2d. Although both triangles are congruent, the one on the right side of the peak is reversed, indicating the slope has become negative—a region of negative returns.

From a mathematical perspective, the USL is a parametric model based on rational functions,[7] and one could imagine continuing to add successive polynomial terms in p to the denominator of equation 2, each with its attendant coefficient. For $\kappa > 0$, however, a maximum exists, and there is usually little virtue in describing analytically how scalability degrades beyond that point. The preferred goal is to remove the peak altogether, if possible—hence the name *universal*.

The central idea is to match the measured speedup in equation 1 with the USL defined in equation 2. For a given node configuration p, this can be achieved only by adjusting the σ and κ coefficients. In practice, this is accomplished using nonlinear statistical regression.[8,17] (The scalability of Varnish, Memcached, and Zookeeper applications are discussed in the *ACM Queue* version of this article).

Figure 1. Qualitative comparison of sublinear, linear, and superlinear speedup scalability.



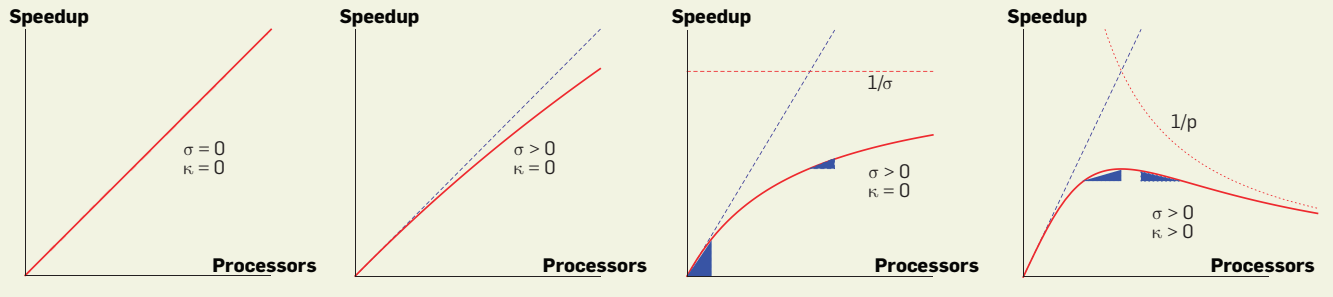
Hadoop Terasort in the Cloud

To explore superlinearity in a controlled environment, we used a well-known workload, the TeraSort benchmark,[15,16] running on the Hadoop MapReduce framework.[3,24] Instead of using a physical cluster, however, we installed it on Amazon Web Services (AWS) to provide the flexibility of reconfiguring a sufficiently large number of nodes, as well as the ability to run multiple experiments in parallel at a fraction of the cost of the corresponding physical system.

Hadoop framework overview. This discussion of superlinear speedup in TeraSort requires some familiarity with the Hadoop framework and its terminology.[24] In particular, this section provides a high-level overview with the primary focus on just those Hadoop components that pertain to the later performance analysis.

The Hadoop framework is designed to facilitate writing large-scale, data-

Figure 2. How the USL model in equation 2 characterizes scalability.



intensive, distributed applications that can run on a multinode cluster of commodity hardware in a reliable, fault-tolerant fashion. This is achieved by providing application developers with two programming libraries:

▸ *MapReduce*, a distributed processing library that enables applications to be written for easy adaptation to parallel execution by decomposing the entire job into a set of independent tasks.

▸ *Hadoop Distributed File System* (HDFS), which allows data to be stored on any node but remain accessible by any task in the Hadoop cluster. An application written using the MapReduce library is organized as a set of independent tasks that can be executed in parallel. These tasks fall into two classes:

▸ *Map task.* The function of the Map task is to take a slice of the entire input dataset and transform it into key-value pairs, commonly denoted by <*key*, *value*> in the context of MapReduce. See the detailed Map-tasks dataflow in Node 1 of Figure 3 where the Map task is represented schematically as a procedure Map(k, v). Besides performing this transform, the Map task also sorts the data by key and stores the sorted <k, v> objects so they can easily be exchanged with a Reduce task.

▸ *Reduce task.* The function of the Reduce task is to collect all the <k, v> objects for a specific key and transform them into a new <k, v> object, where the value of the key is the specific key and whose value is a list [v_1 , v_2 ,...] of all the values that are < k, [v_1, v_2,...]> objects whose key is the specific key across the entire input data set. Node 1 of Figure 3 shows the detailed Reduce-task dataflow.

A MapReduce application processes its input dataset using the following workflow. On startup, the application creates and schedules one Map task per slice of the input dataset, as well as creating a user-defined number of Reduce tasks. These Map tasks then work in parallel on each slice of the input data, effectively sorting and partitioning it into a set of files where all the <k, v> objects that have equal key values are grouped. Once all the Map tasks have completed, the Reduce tasks are signaled to start reading the partitions to transform and combine these intermediate data into new <k, [v_1, v_2,...]> objects. This process is referred to as *shuffle exchange*, shown schematically in Figure 3 as arrows spanning physical nodes 1, 2,..., p.

To facilitate running the application in a distributed fashion, the MapReduce library provides a distributed execution server composed of a central service called the JobTracker and a number of slave services called TaskTrackers.[24] The JobTracker is responsible for scheduling and transferring tasks to the TaskTrackers residing on each cluster node. The JobTracker can also detect and restart tasks that might fail. It provides a level of fault toler-

Figure 3. Hadoop MapReduce dataflow with Node 1 expanded to show tasks detail.

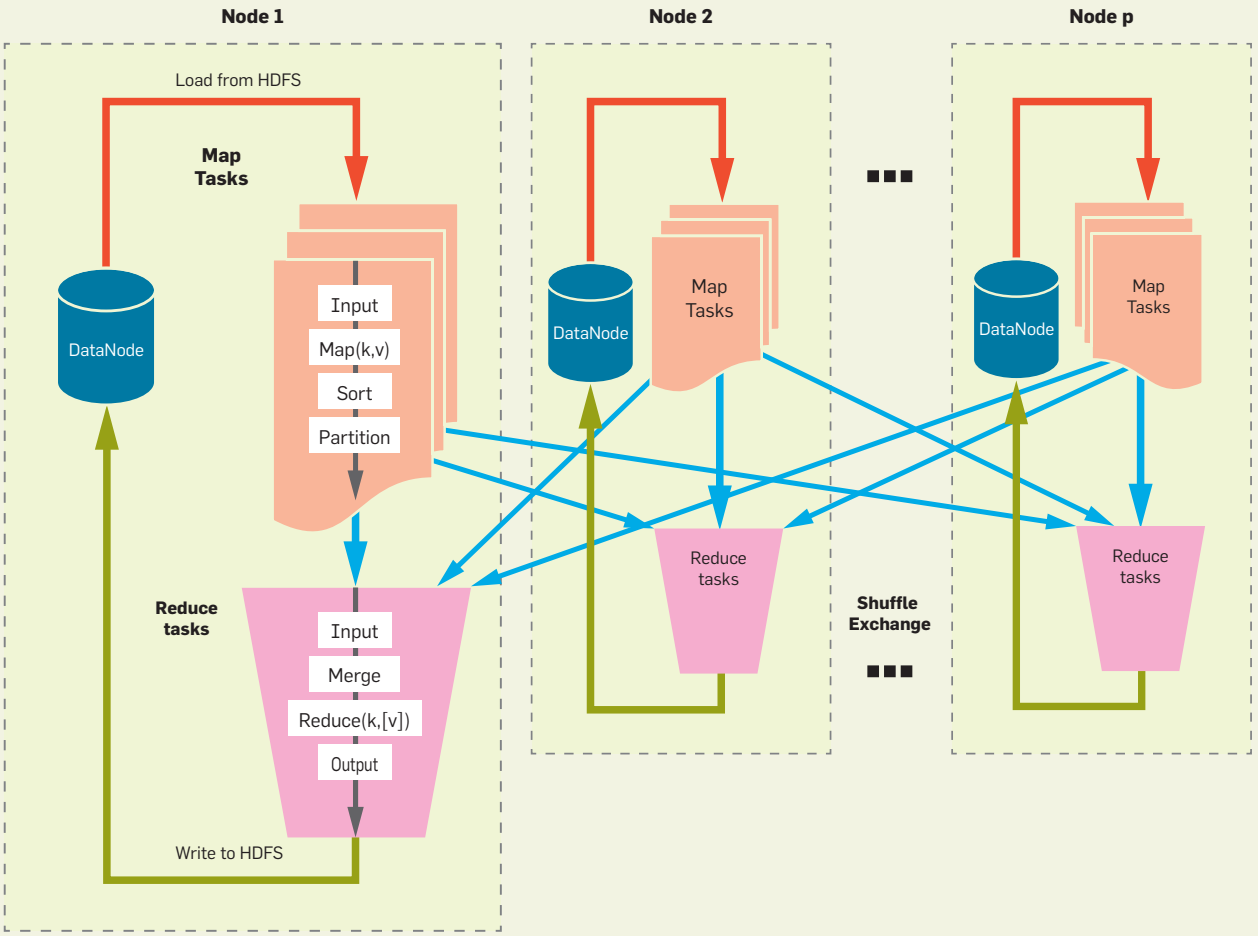


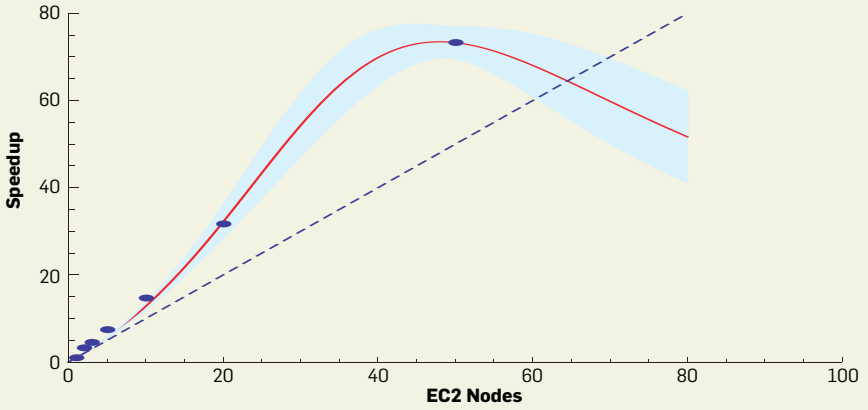
Table 1. Amazon EC2 instance configurations.

	Instance Type	Optimized for	Processor Arch.	vCPU number	Memory (GiB)	Instance Storage (GB)	Network Performance
BigMem	m2.2xlarge	Memory	64-bit	4	34.2	1 x 850	Moderate
BigDisk	c1.xlarge	Compute	64-bit	8	7	4 x 420	High

Figure 4. Bash script used to record Terasort elapse times.

```
BEFORE_SORT='date +%s%3N`
hadoop jar $HADOOP_MAPRED_HOME/hadoop-examples.jar terasort /user/hduser/
terasort-input
/user/hduser/terasort-output
AFTER_SORT='date +%s%3N'
SORT_TIME='expr $AFTER_SORT - $BEFORE_SORT` echo "$CLUSTER_SIZE, $SORT_TIME"
>> sort_time
```

Figure 5. USL analysis of superlinear speedup for p ≤ 50 BigMem nodes.



ance. The user interacts with the Hadoop framework via a JobClient component such as TeraSort that provides monitoring and control of the MapReduce job.

To support the execution of MapReduce tasks, the Hadoop framework includes HDFS, which is implemented as a storage cluster using a master-slave architecture. It provides a reliable distributed file service that allows Hadoop applications to read and write very large data files at high throughput of fixed-sized blocks (128MB in TeraSort[3]) across the cluster. The master node in a HDFS cluster is the NameNode, which is responsible for regulating client access to files, as well as managing the filesystem namespace by mapping file blocks to its storage location, which can reside on the DataNodes (that is, slave nodes to the NameNode). A key feature of HDFS is its built-in resilience to node disk failure, accomplished by replicating blocks across multiple DataNodes. The default replication factor is three, but this is set to one in TeraSort.

It is noteworthy the shuffle-exchange process depicted in Figure 3 involves the interaction between Map and Reduce tasks, which, in general, causes data to be 'Reduced' on different physical nodes. Since this exchange occurs between MapReduce pairs, it scales quadratically with the number of cluster nodes, and that corresponds precisely to the USL coherency term, $p(p-1)$, in equation 2 (Figure 2d). This point will be important for the later performance analysis of superlinear speedup. Moreover, although sorting represents a worst-case MapReduce workload, similar coherency phases will occur with different magnitudes in different Hadoop applications. The actual magnitude of the physical coherency effect is reflected in the value of the κ coefficient that results from USL analysis of Hadoop performance data.

Running TeraSort on AWS. TeraSort is a synthetic workload that has been used recently to benchmark the performance of Hadoop MapReduce[15,16] by measuring the time taken to sort 1TB of randomly generated data. A separate program called TeraGen generates the input data, consisting of 100-byte records with the first 10 bytes used as a key.

The scripts for setting TeraSort up on a Hadoop cluster are readily available. The performance goal here was to use TeraSort to examine the phenomenon of superlinear scalability, not to tune the cluster to produce the shortest runtimes as demanded by competitive benchmarking.[15,16]

Amazon's Elastic Compute Cloud (EC2) provides rapid and cheap provisioning of clusters with various instance types and sizes (for example, those in Table 1). To keep the time and cost of running multiple experiments manageable, data generation was limited to 100GB and the cluster configurations kept to less than 200 EC2 nodes using local instance storage rather than Elastic Block Storage.

EC2 instance types `m2.2xlarge` and `c1.xlarge` are distinguished by the former having five times more memory, but only one hard disk, half the number of cores, and higher network latencies, whereas the latter has four hard disks and lower network latency. Rather than adhering to the clumsy Amazon nomenclature, we refer to them here as BigMem and BigDisk, respectively. These designations emphasize the key capacity difference that will turn out to be important for the later performance analysis.

Apache Whirr[1] is a set of Java libraries for running cloud services that supports Amazon EC2. We used it together with custom bash scripts to: specify cluster size and instance type; bootstrap the EC2 cluster; install Hadoop; prepare and run TeraSort; and collect performance metrics.

Whirr was configured to create a cluster made of EC2 instances running

Linux CentOS 5.4 with the Cloudera CDH 4.7.0 distribution of Hadoop 1.0 installed.[3] Included in that distribution is the `Hadoop-examples.jar` file that contains the code for both the TeraGen and TeraSort MapReduce jobs. Whirr can read the desired configuration from a properties file, as well as receiving properties passed from the command line. This allowed permanent storage of the parameters that did not change (for example, the operating system version and Amazon credentials).

Three sets of performance metrics were gathered:

- The elapsed time for the TeraSort job (excluding the TeraGen job).
- Hadoop-generated job data files.
- Linux performance metrics.

Of these, the most important metric was the elapsed time, which was recorded using the Posix time stamp in milliseconds (since EC2 hardware supports it) via the shell command illustrated in Figure 4.

Runtime performance metrics (for example, memory usage, disk IO rates, and processor utilization) were captured from each EC2 node using the resident Linux performance tools uptime, vmstat, and iostat. The performance data was parsed and appended to a file every two seconds.

A sign of perpetual motion. Figure 5 shows the TeraSort speedup data (dots) together with the fitted USL scalability curve (blue). The linear bound (dashed line) is included for reference. That the speedup data lies on or above the linear bound provides immediate visual evidence that scalability is indeed superlinear. Rather than a linear fit,[21] the USL regression curve exhibits a convex trend near the origin that is consistent with the generic superlinear profile in Figure 1.

The entirely unexpected outcome is that the USL contention coefficient develops a *negative* value: $\sigma = -0.0288$. This result also contradicts the assertion[8] that both σ and κ must be positive for physical consistency—the likely source of the criticism that the USL failed with superlinear speedup data.

As explained earlier, a positive value of σ is associated with contention for shared resources. For example, the same processor that executes user-level tasks may also need to accommodate operating-system tasks such as IO requests. The processor capacity is consumed by work other than the application itself. Therefore, the application throughput is less than the expected linear bang for the capacity buck.

Capacity consumption ($\sigma > 0$) accounts for the sublinear scalability component in Figure 2b. Conversely, $\sigma < 0$ can be identified with some kind of capacity boost. This interpretation will be explained shortly.

Additionally, the (positive) coherency coefficient $\kappa = 0.000447$ means there must be a peak value in the speedup, which the USL predicts as $S_{max} = 73.48$, occurring at $p = 48$ nodes. More significantly, it also means the USL curve must cross the linear bound and enter the payback region shown in Figure 6.

The USL model predicts this crossover from the superlinear region to the payback region must occur for the following reason. Although the magnitude of σ is small, it is also multiplied by $(p - 1)$ in equation 2. Therefore, as the number of nodes increases, the difference, $1 - \sigma (p - 1)$, in the denominator of equation 2 becomes progressively smaller such that S_p is eventually dominated by the coherency term, $\kappa\, p(p - 1)$.

Figure 7 includes additional speedup measurements (squares). The fitted USL coefficients are now significantly smaller than those in Figure 5. The maximum speedup, S_{max}, therefore is about 30% higher than predicted with the data

Table 2. Runtime error analysis.

T1	=	13057	±	606 seconds (r.e. 5%)
T2	=	6347	±	541 seconds (r.e. 9%)
T3	=	4444	±	396 seconds (r.e. 9%)
T5	=	2065	±	147 seconds (r.e. 7%)
T10	=	893	±	27 seconds (r.e. 3%)

Figure 6. Superlinearity and its associated payback region (see Figure 1).

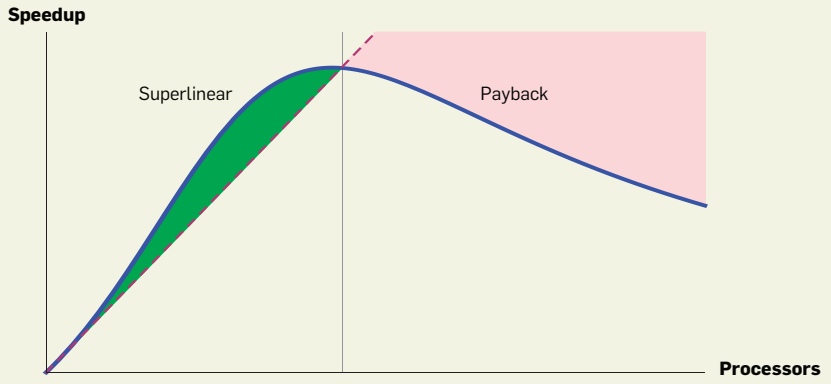


Figure 7. USL analysis of p ≤ 150 BigMem nodes (solid blue curve) with Figure 4 (dashed blue curve) inset for comparison.

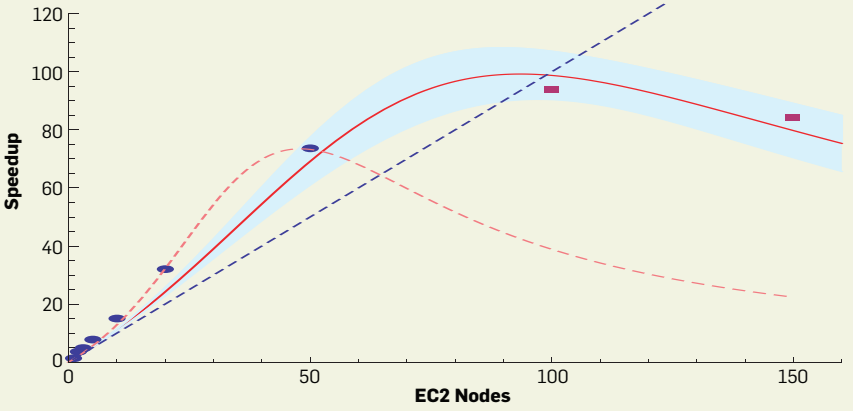
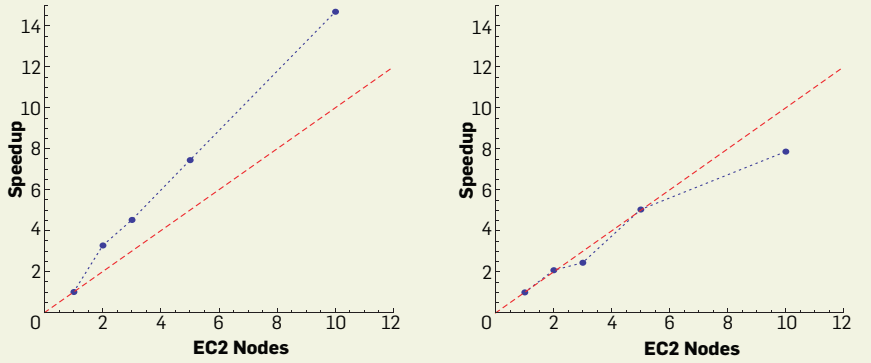


Fig 5 $P \leq 50$ nodes shows more Superlinearity
Fig 7 $P \geq 150$ " " less "

Figure 8. Hadoop TeraSort superlinearity is eliminated by increasing nodal disk IO bandwidth.



in Figure 5 and occurs at $p = 95$ nodes. The measured values of the speedup differ from the original USL prediction, not because the USL is wrong but because there is now more information available than previously. Moreover, this confirms the key USL prediction that superlinear speedup reaches a maximum value and then rapidly declines into the payback region.

Based on USL analysis, the scalability curve is expected to cross the linear bound at $p_\times$ nodes given by equation 3:

$$p_\times = \left\lceil \frac{|\sigma|}{\kappa} \right\rceil \qquad (3)$$

For the dashed curve in Figure 7, the crossover occurs at $p_\times = 65$ nodes, whereas for the solid curve it occurs at $p_\times = 99$ nodes. Like predicting S_{max}, the difference in the two $p_\times$ predictions comes from the difference in the amount of information contained in the two sets of measurements.

Hunting the Superlinear Snark

After the TeraSort data was validated against the USL model, a deeper performance analysis was needed to determine the cause of superlinearity. Let's start with a closer examination of the actual runtime measurements for each EC2 cluster configuration.

Runtime data analysis. To make a statistical determination of the error in the runtime measurements, we performed some runs with a dozen repetitions per node configuration. From that sample size a reasonable estimate of the uncertainty can be calculated based on the standard error, or the relative error, which is more intuitive.

For each of the runtimes in Table 2, the number before the ± sign is the sample mean, while the error term following the ± sign is derived from the sample variance. The relative error (r.e.) is the ratio of the standard error to the mean value reported as a percentage.

What is immediately evident from this numerical analysis is the significant variation in the relative errors with a range from 3%, which is nominal, to 9%, which likely warrants further attention. This variation in the measurement error does not mean the measurement technique is unreliable; rather, it means there is a higher degree of dispersion in the runtime data for reasons that cannot be discerned at this level of analysis.

Nor is this variation in runtimes peculiar to our EC2 measurements. The Yahoo TeraSort benchmark team also noted significant variations in their execution times, although they did not quantify them. *"Although I had the 910 nodes mostly to myself, the network core was shared with another active 2000 node cluster, so the times varied a lot depending on the other activity."*[15]

Some of the Yahoo team's sources of variability may differ from ours (for example, the 10x larger cluster size is likely responsible for some of the Yahoo variation). *"Note that in any large cluster and distributed application, there are a lot of moving pieces and thus we have seen a wide variation in execution times."*[16]

A surprising hypothesis. The physical cluster configuration used by the Yahoo benchmark team comprised nodes with two quad-core Xeon processors (that is, a total of eight cores per node) and four SATA disks.[16] This is very similar to the BigDisk EC2 configuration in Table 1. We therefore repeated the TeraSort scalability measurements on the BigDisk cluster. The results for $p = 2, 3, 5$, and 10 clusters are compared in Figure 8.

Consistent with Figure 5, BigMem speedup values in Figure 8a are superlinear, whereas the BigDisk nodes in Figure 8b unexpectedly exhibit speedup values that are either linear or sublinear. The superlinear effect has essentially been eliminated by increasing the number of local spindles from one to four per cluster node. In other words, increasing nodal IO bandwidth leads to the counterintuitive result that scalability is degraded from superlinear to sublinear.

In an attempt to explain why the superlinear effect has diminished, we formed a working hypothesis by identifying the key performance differences between BigMem and BigDisk.

BigMem has the larger memory configuration, which possibly provides more CentOS buffer caching for the TeraSort data, and that could be thought of as being the source of the capacity boost associated with the negative USL contention coefficient. Incremental memory growth in proportion to cluster size is a common explanation for superlinear speedup.[4,14] Increasing memory size, however, is probably not the source of the capacity boost in Hadoop TeraSort. If the buffer cache fills to the point where it needs to be written to disk, it will take longer because there is only a single local disk per node on BigMem. A single-disk DataNode in Figure 3 implies all disk IO is serialized. In this sense, when disk writes (including replications) occur, TeraSort is IO bound—most particularly in the single-node case. As the cluster configuration gets larger, this latent IO constraint becomes less severe since the amount of data per node that must be written to disk is reduced in proportion to the number of nodes. Successive cluster sizes therefore exhibit runtimes that are shorter than the single-node case, and that results in the superlinear speedup values shown in Figure 8a.

Conversely, although BigDisk has a smaller amount of physical memory per node, it has quad disks per

DataNode, which means each node has greater disk bandwidth to accommodate more concurrent IOs. TeraSort is therefore far less likely to become IO bound. Since there is no latent single-node IO constraint, there is no capacity boost at play. As a result, the speedup values are more orthodox and fall into the sublinear region of Figure 8b.

Note that since the Yahoo benchmark team used a cluster configuration with four SATA disks per node, they probably did not observe any superlinear effects. Moreover, they were focused on measuring elapsed times, not speedup, for the benchmark competition, so superlinearity would have been observable only as execution times T_p falling faster than p^{-1}.

Console stack traces. The next step was to validate the IO bottleneck hypothesis in terms of Hadoop metrics collected during each run. While TeraSort was running on BigMem, task failures were observed in the Hadoop JobClient console that communicates with the Hadoop JobTracker. The following is an abbreviated form of a failed task status message with the salient identifiers shown in bold in Figure 9.

Since the TeraSort job continued and all tasks ultimately completed successfully, we initially discounted these failure reports. Later, with the IO bottleneck hypothesis in mind, we realized these failures seemed to occur only during the Reduce phase. Simultaneously, the Reduce task %Complete value decreased immediately when a failure appeared in the console. In other words, progress of that Reduce task became retrograde. Moreover, given that the failure in the stack trace involved the Java class `DFSOutputStream`, we surmised the error was occurring while attempting to write to HDFS. This suggested examining the server-side Hadoop logs to establish the reason why the Reduce failures are associated with HDFS writes.

Hadoop log analysis. Searching the Hadoop cluster logs for the same failed `TASK_ATTEMPT_ID`, initially seen in the JobClient logs, revealed the corresponding record as shown in Figure 10.

This record indicates the Reduce task actually failed on the Hadoop cluster, as opposed to the JobClient. Since the failure occurred during the invocation of `DFSOutputStream`, it also suggests there was an issue while physically writing data to HDFS.

Furthermore, a subsequent record in the log with the same task ID, as shown in Figure 11, had a newer `TASK_ATTEMPT_ID` (namely, a trailing 1 instead of a trailing 0) that was successful.

This log analysis suggests if a Reduce task fails to complete its current write operation to disk, it must start over by rewriting that same data until it is successful. In fact, there may be multiple failures and retries (see Table

Figure 9. Failed Reduce task as seen in the Hadoop job-client console.

```
14/10/01 21:53:41 INFO mapred.JobClient: Task Id :
attempt_201410011835_0002_r_000000_0,
Status : FAILED java.io.IOException: All datanodes 10.16.132.16:50010 are bad.
Aborting . . .
. . .
at org.apache.hadoop.hdfs.DFSOutputStream$DataStreamer.run(DFSOutputStream.
java:463)
```

Figure 10. Hadoop cluster log message corresponding to Figure 9.

```
ReduceAttempt TASK_TYPE="REDUCE" TASKID="task_201410011835_0002_r_000000"
TASK_ATTEMPT_ID="attempt_201410011835_0002_r_000000_0" TASK_STATUS="FAILED"
FINISH_TIME="1412214818818" HOSTNAME="ip-10-16-132-16.ec2.internal"
ERROR="java.io.IOException: All datanodes 10.16.132.16:50010 are bad. Aborting
. . .
. . .
at org.apache.hadoop.hdfs.DFSOutputStream$DataStreamer.run(DFSOutputStream.
java:463)
```

Figure 11. Successful retry of failed Reduce task.

```
ReduceAttempt TASK_TYPE="REDUCE" TASKID="task_201410011835_0002_r_000000"
TASK_ATTEMPT_ID=
"attempt_201410011835_0002_r_000000_1" TASK_STATUS= "SUCCESS"
```

Figure 12. Origin of Reduce task failure as seen in the Hadoop log.

```
ReduceAttempt TASK_TYPE="REDUCE" ... TASK_STATUS="FAILED" . . .
ERROR="java.io.IOException: All datanodes are bad. Aborting . . .
. . .
.setupPipelineForAppendOrRecovery(DFSOutputStream.java:1000)
```

Table 3. Single-node BigMem metrics extracted from Hadoop log.

Job ID	Finished Maps	Failed Maps	Finished Reduces	Failed Reduces	Job Runtime
1	840	0	3	0	9608794
2	840	0	3	2	12167730
3	840	0	3	0	10635819
4	840	0	3	1	11991345
5	840	0	3	0	11225591
6	840	0	3	2	12907706
7	840	0	3	2	12779129
8	840	0	3	3	13800002
9	840	0	3	3	14645896
10	840	0	3	4	15741466
11	840	0	3	2	14536452
12	840	0	3	3	16645014

3). The potential difference in runtime resulting from Reduce retries is obscured by the aforementioned variation in runtime measurements, which is also on the order of 10%.

Table 3 shows 12 rows corresponding to 12 parallel TeraSort jobs, each running on its own BigMem single-node cluster. A set of metrics indicating how each of the runs executed is stored in the Hadoop job-history log and extracted using Hadoop log tools.[13]

The 840 Map tasks are determined by the TeraSort job partitioning 100 (binary) GB of data into 128 (decimal) MB HDFS blocks. No Map failures occurred. The number of Reduce tasks was set to three per cluster node. The number of failed Reduce tasks varied randomly between none and four. In comparison, there were no Reduce failures for the corresponding BigDisk case.

The average runtime for Hadoop jobs was 13057078.67 ms, shown as T_1 in Table 2. Additional statistical analysis reveals a strong correlation between the number of Reduce task retries and longer runtimes. Recalling the definition of speedup, if the mean single-node runtime, T_1, is longer than successive values of pT_p, then the speedup will be superlinear.

Whence reduce fails? The number of failed Reduces in Table 3 indicates that a write failure in the Reduce task causes it to retry the write operation—possibly multiple times. In addition, failed Reduce tasks tend to incur longer runtimes as a consequence of those additional retries. The only outstanding question is, what causes the writes to fail in the first place? We already know that write operations are involved during a failure, and that suggests examining the HDFS interface.

Returning to the earlier failed Reduce stack trace, closer scrutiny reveals the following lines, with important key words shown in bold in Figure 12.

The "`All datanodes are bad`" Java IOException means the HDFS DataNode pipeline in Figure 13 has reached a state where the `setupPipelineForAppendOrRecovery` method, on the `DFSOutputStream` Java class, cannot recover the write operation, and the Reduce task fails to complete.

When the pipeline is unhindered, a Reduce task makes a call into the HDFSClient, which then initiates the creation of a HDFS DataNode pipeline. The HDFSClient opens a *DFSOutputStream* and readies it for writing (`1. Write` in Figure 13) by allocating a HDFS data block on a DataNode. The `DFSOutputStream` then breaks the data stream into smaller packets of data. Before it transmits each data packet to be written by a DataNode (`2. Write packet`), it pushes a copy of that packet onto a queue. The DFSOutputStream keeps that packet in the queue until it receives an acknowledgment (`3. ACK packet`) from each DataNode that the write operation completed successfully.

When an exception is thrown (for example, in the stack trace) the `DFSOutputStream` attempts to remedy the situation by reprocessing the packets to complete the HDFS write. The `DFSOutputStream` can make additional remediation attempts up to one less than the replication factor. In the case of TeraSort, however, since the replication factor is set to one, the lack of a single HDFS packet acknowledgment will cause the entire `DFSOutputStream` write operation to fail.

The `DFSOutputStream` endeavors to process its data in an unfettered way, assuming the DataNodes will be able to keep up and respond with acknowledgments. If, however, the underlying IO subsystem on a DataNode cannot keep up with this demand, an outstanding packet can go unacknowledged for too long. Since there is only a single replication in the case of TeraSort, no remediation is undertaken. Instead, the `DFSOutputStream` immediately regards the outstanding write packet to be AWOL and throws an exception that propagates back up to the Reduce task in Figure 13.

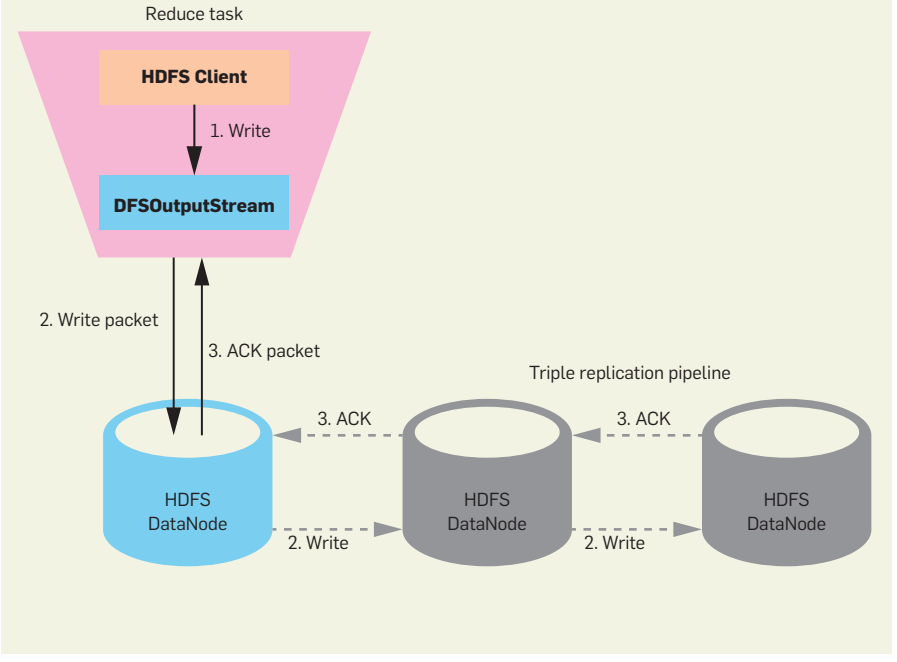
Since the Reduce task does not know how to handle this IO exception, it completes with a `TASK_STATUS="FAILED"`. The MapReduce framework will eventually retry the entire Reduce task, possibly more than once (see Table 3), and that will be reflected in a stretched T_1 value that is ultimately responsible for the observed superlinear speedup.

This operational insight into Reduce failures can be used to construct a list of simple tactics to avoid runtime stretching.

1. Resize the buffer cache.
2. Tune kernel parameters to increase IO throughput.
3. Reconfigure Hadoop default timeouts.

If maintaining a BigMem-type cluster is dictated by nonengineering requirements (for example, budgetary

Figure 13. HDFS DataNode pipeline showing single replication (blue) and default triple replication blue and gray).



constraints), then any of these steps could be helpful in ameliorating superlinear effects.

Conclusion

The large number of controlled measurements performed by running Hadoop TeraSort on Amazon EC2 exposed the underlying cause of superlinearity that would otherwise be difficult to resolve in the field. Fitting our speedup data to the USL performance model produced a negative contention coefficient as a telltale sign of superlinearity on BigMem clusters.

The subtractive effect of negative σ introduces a point of inflection in the convex superlinear curve that causes it ultimately to become concave, thus crossing over the linear bound at $p_{\times}$ in equation 3. At that point, Hadoop TeraSort superlinear scalability returns to being sublinear in the payback region. The cluster size $p_{\times}$ provides an estimate of the minimal node capacity needed to ameliorate superlinear speedup on BigMem clusters.

Although superlinearity is a bona fide phenomenon, just like perpetual motion it is ultimately a performance illusion. For TeraSort on BigMem, the apparent capacity boost can be traced to successively relaxing the latent IO bandwidth constraint per node as the cluster size grows. This IO bottleneck induces stochastic failures of the HDFS pipeline in the Reduce task. That causes the Hadoop framework to restart the Reduce task file-write, which stretches the measured runtimes. If runtime stretching is greatest for T_1, then successive speedup measurements will be superlinear. Increasing the IO bandwidth per node, as we did with BigDisk clusters, diminishes or eliminates superlinear speedup by reducing T_1 stretching.

This USL analysis suggests superlinear scalability is not peculiar to TeraSort on Hadoop but may arise with any MapReduce application. Superlinear speedup has also been observed in relational database systems.[2,12] For high-performance computing applications, however, superlinear speedup may arise differently from the explanation presented here.[4,14,20]

Superlinearity aside, the more important takeaway for many readers may be the following. Unlike most software-engineering projects, Hadoop applications require only a fixed development effort. Once an application is demonstrated to work on a small cluster, the Hadoop framework facilitates scaling it out to an arbitrarily large number of nodes with no additional effort. For many MapReduce applications, scale-out may be driven more by the need for disk storage than compute power as the growth in data volume necessitates more Maps. The unfortunate term *flat scalability* has been used to describe this effect.[25]

Although flat scalability may be a reasonable assumption for the initial development process, it does not guarantee that performance goals—such as batch windows, traffic capacity, or service-level objectives—will be met without significant additional effort. The unstated assumption behind the flat-scalability precept is that Hadoop applications scale linearly (Figure 2a) or near-linearly (Figure 2b). Any shuffle-exchange processing, however, will induce a peak somewhere in the scalability profile (Figure 2d). The Hadoop cluster size at which the peak occurs can be predicted by applying the USL to small-cluster measurements. The performance-engineering effort needed to temper that peak will typically far exceed the flat-scalability assumption. As this article has endeavored to show, the USL provides a valuable tool for the software engineer to analyze Hadoop scalability.

Acknowledgments

We thank Comcast Corporation for supporting the acquisition of Hadoop data used in this work.

Related articles on queue.acm.org

Hazy: Making it Easier to Build and Maintain Big-Data Analytics
Arun Kumar, Feng Niu, and Christopher Ré
http://queue.acm.org/detail.cfm?id=2431055

Data-Parallel Computing
Chas. Boyd
http://queue.acm.org/detail.cfm?id=1365499

Condos and Clouds
Pat Helland
http://queue.acm.org/detail.cfm?id=2398392

References

1. Apache Whirr; https://whirr.apache.org.
2. Calvert, C. and Kulkarni D. *Essential LINQ*. Pearson Education, Boston, MA, 2009.
3. Cloudera Hadoop; http://www.cloudera.com/content/cloudera/en/downloads/cdh/cdh-4-7-0.html/.
4. Eijkhout, V. *Introduction to High Performance Scientific Computing*. Lulu.com, 2014.
5. Feynman, R.P. Papp perpetual motion engine; http://hoaxes.org/comments/papparticle2.html.
6. Gunther, N.J. A simple capacity model of massively parallel transaction systems. In *Proceedings of International Computer Measurement Group Conference*, (1993).
7. Gunther, N.J. A general theory of computational scalability based on rational functions, 2008; http://arxiv.org/abs/0808.1431.
8. Gunther, N.J. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Springer, New York, NY, 2007.
9. Gunther, N.J. Performance and scalability models for a hypergrowth e-commerce Web site. *Performance Engineering*. R.R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, eds. A. Lecture Notes in Computer Science 2047 (2001). Springer-Verlag 267-282.
10. Gunther, N.J. PostgreSQL scalability analysis deconstructed. *The Pith of Performance*, 2012; http://perfdynamics.blogspot.com/2012/04/postgresql-scalability-analysis.html.
11. Gunther, N.J., Subramanyam, S. and Parvu, S. Hidden scalability gotchas in memcached and friends. VELOCITY Web Performance and Operations Conference, (2010).
12. Haas, R. Scalability, in graphical form, analyzed, 2011; http://rhaas.blogspot.com/2011/09/scalability-in-graphical-form-analyzed.html.
13. Hadoop Log Tools; https://github.com/melrief/Hadoop-Log-Tools.
14. Hennessy, J.L. and Patterson, D.A. *Computer Architecture: A Quantitative Approach*. Second edition. Morgan Kaufmann, Waltham, MA, 1996.
15. O'Malley, O. TeraByte sort on Apache Hadoop, 2008; http://sortbenchmark.org/YahooHadoop.pdf.
16. O'Malley, O., Murthy, A. C. 2009. Winning a 60-second dash with a yellow elephant; http://sortbenchmark.org/Yahoo2009.pdf.
17. Performance Dynamics Company. How to quantify scalability, 2014; http://www.perfdynamics.com/Manifesto/USLscalability.html.
18. Schwartz, B. Is VoltDB really as scalable as they claim? Percona MySQL Performance Blog; http://www.percona.com/blog/2011/02/28/is-voltdb-really-as-scalable-as-they-claim/.
19. sFlow. SDN analytics and control using sFlow standard — Superlinear; http://blog.sflow.com/2010/09/superlinear.html.
20. Stackoverflow. Where does superlinear speedup come from?; http://stackoverflow.com/questions/4332967/where-does-super-linear-speedup-come-from
21. Sun Fire X2270 M2 superlinear scaling of Hadoop TeraSort and CloudBurst benchmarks, 2010; https://blogs.oracle.com/BestPerf/entry/20090920_x2270m2_hadoop.
22. Sutter, H. Going superlinear. *Dr. Dobb's J. 33*, 3 (2008); http://www.drdobbs.com/cpp/going-superlinear/206100542.
23. Sutter, H. Super linearity and the bigger machine. *Dr. Dobb's J. 33*, 4 (2008); http://www.drdobbs.com/parallel/super-linearity-and-the-bigger-machine/206903306.
24. White, T. *Hadoop: The Definitive Guide*, third edition. O'Reilly Media, 2012.
25. Yahoo! Hadoop Tutorial; https://developer.yahoo.com/hadoop/tutorial/module1.html#scalability.

Neil J. Gunther (http://perfdynamics.blogspot.com; tweets as @DrQz) is a researcher and teacher at Performance Dynamics where he developed the USL and the PDQ open source performance analyzer.

Paul Puglia (pjpuglia@gmail.com) has been working in IT for more than 20 years doing Python programming, system administration, and performance testing. He has authored an R package, SATK, for fitting performance data to the USL, and contributed to the PDQ open source performance analyzer.

Kristofer Tomasette (ktomasette@gmail.com) is a senior software engineer on the Platforms & APIs team at Comcast Corporation. He has built software systems involving warehouse management, online banking, telecom, and most recently cable TV.