Check for updates

acmqueue

Hadoop Superlinear Scalability

The perpetual motion of parallel performance

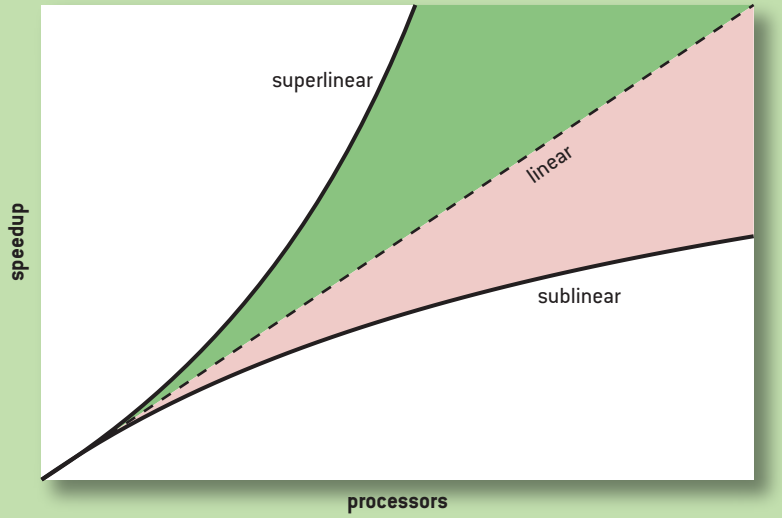
Neil Gunther, Performance Dynamics
Paul Puglia
Kristofer Tomasette, Comcast

"We often see more than 100 percent speedup efficiency!' came the rejoinder to the innocent reminder that you can't have more than 100 percent of anything. But 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—the USL (Universal Scalability Law)—failed when applied to superlinear speedup data.

Indeed, superlinear speedup is a bona fide, measurable 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, USL is not only capable of accommodating superlinear speedup in a surprisingly simple way, it reveals that superlinearity, although alluring, is as illusory as perpetual motion.

To elaborate, figure 1 shows conceptually that linear speedup (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 percent efficiency. More

FIGURE 1 **Qualitative Comparison of Sublinear, Linear and Superlinear Speedup Scalability**



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 speedup 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).

Superlinearity is a genuinely measurable effect,[4,12,14,21,22,23,24,25] so it's 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.[26] 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 hard part is not deciding whether the claim violates the law of conservation of energy; the hard 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 you tolerate? That question can't 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 fulfills that role in a relatively simple way.[6,7,8,9,10,19,20] The next section introduces the USL performance model in preparation for applying it to superlinear data. In Appendix A, we also show how the USL has been used to validate applications with more orthodox scalability characteristics.

UNIVERSAL SCALABILITY MODEL

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

$$S_p = \frac{T_1}{T_p}$$

Equation 1

where T_p is the measured runtime on $p = 1, 2, 3, \ldots$ 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 *concave* discrete function of p. The following special cases can be identified.

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

• **Sublinear speedup.** If $T_p > T_1 / p$ for each cluster configuration, then successive speedup values will be *inferior* to the linear scalability bound in figure 1—in other words, *sublinear* speedup (red curve). For example, if $p = 2$ and $T_2 = 3\ T_1 / 4$, then $S_2 = 1.33$. Since this 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 speedup.** If $T_p < T_1 / p$, then successive speedup values will be superior to the linear bound, as represented by the blue curve in figure 1—in other words, superlinear speedup. For example, if $p = 2$ and $T_2 = T_1 / 3$, then $S_2 = 3$, which is greater than linear speedup.

It's important to note that the definition of speedup in equation 1 is based on *measured*, not theoretical, values. The scalability of any computer system can be validated by comparing the measured speedup with the theoretically expected speedup, defined in the following section.

COMPONENTS OF SCALABILITY

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

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

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

Each of these scaling effects can be represented as separate terms in an analytic performance model: USL.[8,9,19] The theoretical speedup is shown in equation 2 as:

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

Equation 2

where the coefficient σ represents the degree of *contention* in the system and the coefficient κ represents the lack of *coherency* in the distributed data.

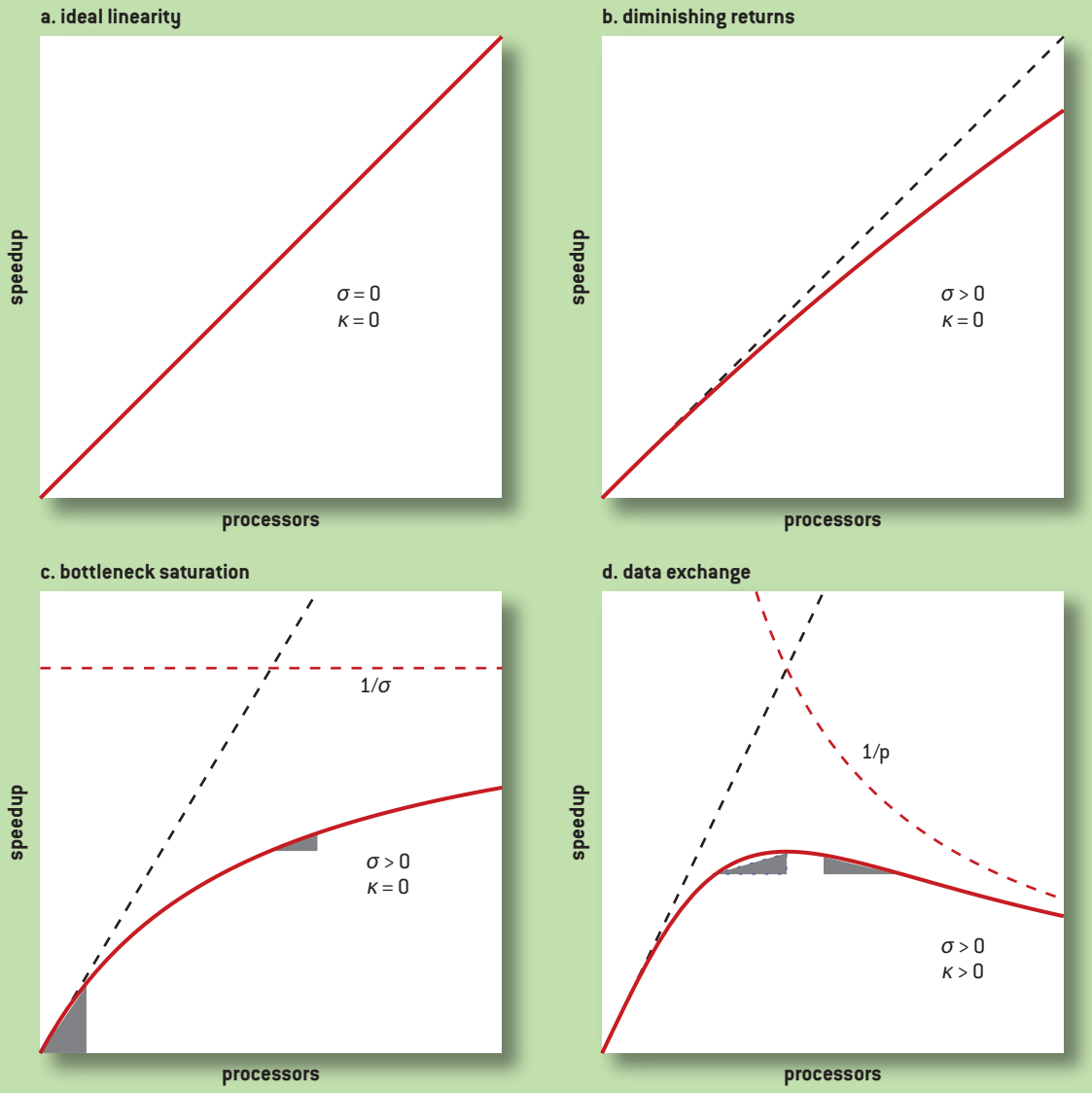
The contention term in equation 2 grows linearly with the number of cluster nodes, p, since it represents the cost of waiting for a shared resource such as message queueing. The coherency term grows quadratically with p because it represents the cost of making distributed data consistent (or coherent)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

INTERPRETING THE COEFFICIENTS

If both the coefficients σ and κ are zero in equation 2, then the speedup simply reduces to $S_p = p$, which corresponds to figure 2a. If σ is nonzero, the speedup starts to fall away from linear, even when the node configuration is relatively small, as in figure 2b. As the number of nodes continues to grow, the speedup approaches the ceiling, $S_{ceiling} = 1/\sigma$, indicated by the horizontal dashed line in figure 2c. The two triangles in figure 2c indicate that 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 κ is also nonzero, then the speedup will eventually degrade like $1/p$ toward the x axis. That

FIGURE 2 How the USL Model in Equation 2 Characterizes Scalability



implies the continuous scalability curve must pass through a maximum or peak value, as in figure 2d. Although the two triangles are congruent, the triangle on the right side of the peak is reversed, indicating that the slope has become negative. Hence, this is not just a region of diminishing returns, but *negative* returns.

From a mathematical perspective, USL is a parametric model based on rational functions,[9] and one could imagine continuing to add successive polynomial terms in p to the denominator of equation 2, each with its attendant coefficient. For a nonzero κ coefficient, 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 maximum altogether if possible—hence the use of the word *universal*.

The central idea used throughout this article is to match the measurement-based definition of speedup in equation 1 with the performance-model definition in equation 2. For a given node configuration p, this can be achieved only by adjusting the values of the coefficients σ and κ. In practice, this is accomplished using nonlinear statistical regression.[8,19]

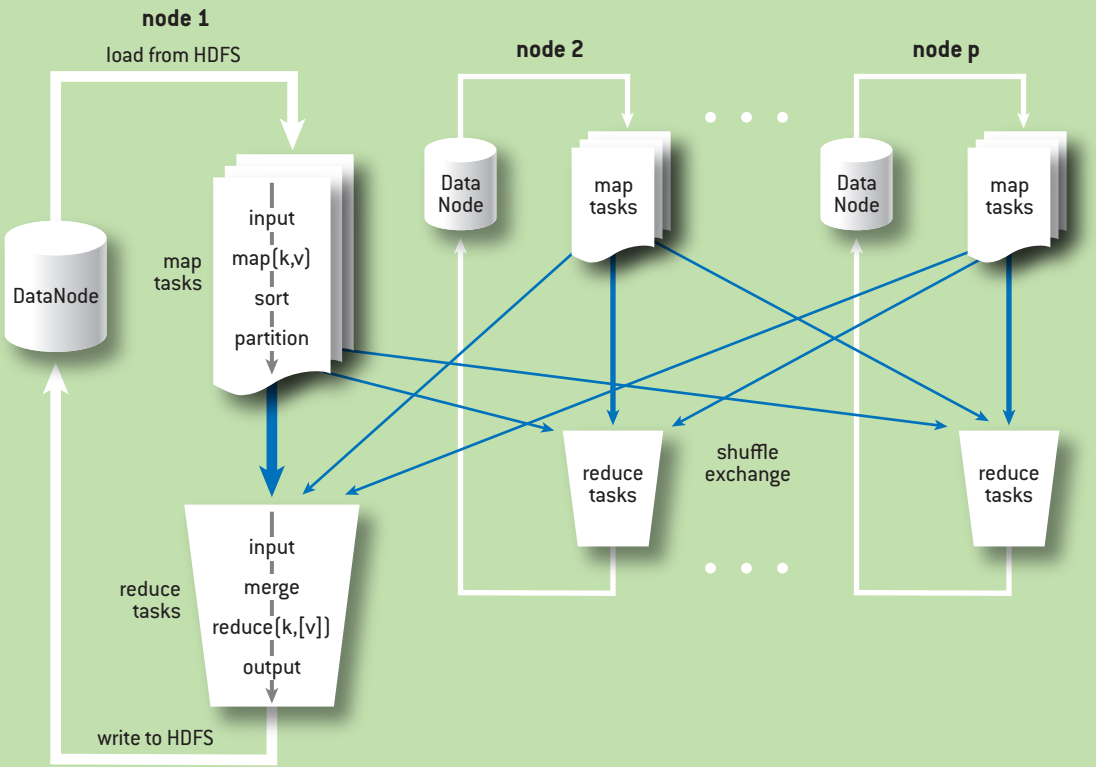
In Appendix A at the end of this article, readers can see how USL is applied to applications that do not exhibit superlinear scalability—such as Varnish, Memcached, and ZooKeeper. Those examples also serve to illustrate how the USL model can be used for both prediction and explanation. The use of a performance model for prediction is widely known and assumed; explanation, however, is not often recognized as a reason to apply a performance model to data, but that is primarily the way we use USL here.

HADOOP TERASORT IN THE CLOUD

To explore superlinearity in a controlled environment, we used a well-known workload, the TeraSort benchmark,[16,17] running on the Hadoop MapReduce framework.[3,27] Instead of using a physical cluster, however, we installed it on AWS (Amazon Web Services) 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.

FIGURE 3

Hadoop MapReduce Dataflow with Node 1 Expanded to Show Tasks Detail



Appendix B provides a high-level overview of the Hadoop framework and its terminology,[27] focusing on the components that pertain to the performance analysis later in this article.

It is noteworthy that 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 (compare with 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 are likely to occur with different magnitudes in different Hadoop applications. The actual magnitude of the physical coherency effect is determined by the value of the coefficient κ that results from USL analysis of the Hadoop performance data.

RUNNING TERASORT ON AWS

TeraSort is a synthetic workload that has been used recently to benchmark the performance of Hadoop MapReduce by measuring the time taken to sort 1 TB of randomly generated data—hence the name. The input data, which is generated by a separate program called TeraGen, consists of 100-byte records with the first 10 bytes used as a key. The output data has a replication factor of one in TeraSort, not the default factor of three in Hadoop. TeraSort is a good choice for exploring superlinearity because the scripts for setting it up on a Hadoop cluster are readily available.

TeraSort relies on the fact that the MapReduce framework sorts the output of Map tasks before executing the Reduce tasks. In the Map phase, TeraSort simply outputs each key-value pair that it reads from the input file. The MapReduce framework then sorts the keys (the Sort box in figure 3). A custom partitioning algorithm breaks the keys into sorted subsets (the Partition box in figure 3). Each partition is assigned to a Reduce task, then the shuffle-exchange process gathers all the partitions assigned to a given Reduce task into its Input box (shown in figure 3). The Reduce task writes its results (the Output box in figure 3) to an HDFS (Hadoop distributed file system) directory. This ensures that the files in that directory, taken collectively, are totally ordered.

It is important to emphasize that the goal here is to examine the phenomenon of superlinear speedup with TeraSort, not to tune the cluster to produce the shortest runtimes, as would be demanded for competitive benchmarking.[16,17]

To keep the time and cost of running multiple experiments manageable, we limited the amount of data TeraGen generates to just 100 GB and the Amazon EC2 (Elastic Compute Cloud) configurations to fewer than 200 nodes. This choice mimics typical configurations that might be seen in practice. The particular EC2 cluster configurations used for our Hadoop measurements are summarized in table 1. They use local instance storage rather than Elastic Block storage.

EC2 instance types m2.2xlarge and c1.xlarge are distinguished by the former having five times

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

more memory but only a single 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 obscure Amazon instance type nomenclature, we refer to m2.2xlarge and c1.xlarge, respectively, using the more descriptive names BigMem and BigDisk (see table 1) to emphasize the key capacity difference, which will turn out to be important for the later performance analysis.

Amazon EC2 supports the rapid and cheap provisioning of clusters with various instance types and sizes, such as those in table 1. We needed a way to bootstrap the EC2 cluster, install Hadoop, and prepare and run TeraSort, as well as collect performance metrics. We also wanted to manipulate parameters such as cluster size and instance type in an easily repeatable way. That was accomplished with Apache Whirr[1] and some custom bash scripts.

Whirr is a set of Java libraries for running cloud services. Since it supports Amazon EC2, it was a natural choice. We configured Whirr to create a cluster consisting 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 data generation and the TeraSort MapReduce jobs. Whirr can read the desired configuration from a properties file, as well as receiving properties passed in from the command line. This allows permanent storage of the parameters that do not change (e.g., the OS version and Amazon credentials). We were then able to manipulate the cluster size and instances as command-line parameters.

Three sets of 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 is the elapsed time of the TeraSort job, which is recorded using the Posix timestamp in milliseconds (since EC2 hardware supports it) via the shell command:

```
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
```

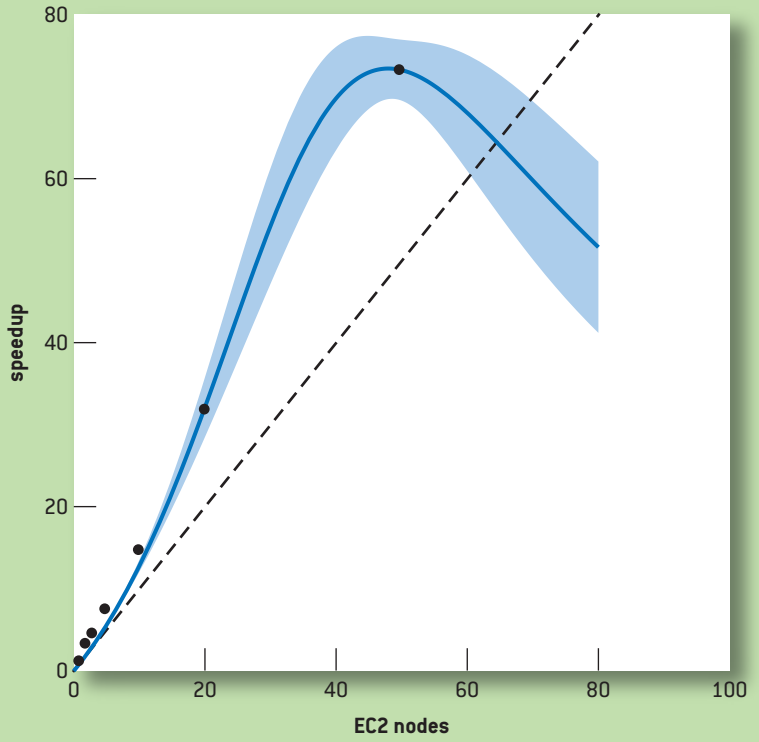
Runtime performance metrics, such as memory usage, disk I/O metrics, and processor utilization, were captured for each EC2 node instance using the resident Linux performance tools `uptime`, `vmstat`, and `iostat`. The performance data was parsed, and output as comma-separated values was appended to a file every two seconds.

A SIGN OF PERPETUAL MOTION

Figure 4 shows the TeraSort speedup data (dots) together with the USL projected scalability curve (blue). The linear bound (dashed line) has been included for reference. The fact that the data all lie on or above the linear bound provides immediate visual evidence that the speedup is indeed superlinear. Rather than a linear fit,[23] the USL regression curve exhibits a convex trend near the origin that is consistent with the generic superlinear profile in figure 1.

FIGURE 4

USL Analysis of Superlinear Speedup for p ≤ 50 BigMem Nodes



The entirely unexpected result of the USL regression analysis is that the contention coefficient develops a negative value, $\sigma = -0.0288$, as distinct from the conventional positive value seen for applications such as Varnish and Memcached in Appendix A. It also (superficially) contradicts the assertion that both σ and κ must be positive for physical consistency.[8 §5.5.4] This is the likely source of the criticism voiced at the beginning of this article that the USL failed when applied to 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 system-level tasks such as I/O requests. In that sense, the same processor capacity can be consumed by work other than the application itself. Therefore, the application takes longer to complete, and the throughput is less than the expected linear bang for the capacity buck.

That kind of capacity *consumption* accounts for the sublinear scalability component in figure 2b. Conversely, a negative value of σ can be identified with some kind of capacity *boost*, the source of which has to be determined. This interpretation will be explained shortly.

In addition, the USL regression analysis produces a coherency coefficient with a positive value of $\kappa = 0.000447$. As seen in figure 2d, that means there must be a peak value for the speedup, which the USL predicts to be $S_{max} = 73.48$ occurring at $p = 48$ nodes. More significantly, it also means that the scalability curve must cross the linear bound and enter the payback region shown in figure 5. That's

where you pay the piper for (apparently) getting a superlinear ride for free.

The USL model predicts that 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 multiplied by (p – 1) in equation 2. Therefore, as the number of nodes increases, the difference, 1 – σ (p – 1), in the denominator of equation 2 becomes progressively smaller such that S_p is eventually dominated by the quadratic coherency term, κ p (p – 1), in the denominator.

Figure 6 includes additional speedup measurements (squares). The fitted USL coefficients are now significantly smaller than those in figure 4. The maximum speedup, S_{max}, is therefore about 30 percent higher than predicted on the basis of the data in figure 4 and now 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 would reach a maximum value and then rapidly decline into the payback region.

Based on the USL regression coefficients, the scalability curve is expected to cross the linear bound at p_x nodes given in equation 3:

$$p_x = \left| \frac{|\sigma|}{\kappa} \right|$$

Equation 3

For the dashed curve in figure 6, the crossover occurs at p_x = 65 nodes, whereas for the solid curve it occurs at p_x = 99 nodes. As with S_{max}, the difference in the two p_x predictions comes from the difference in the amount of information contained in the two sets of measurements.

FIGURE 5 **Superlinearity and Its Associated Payback Region**

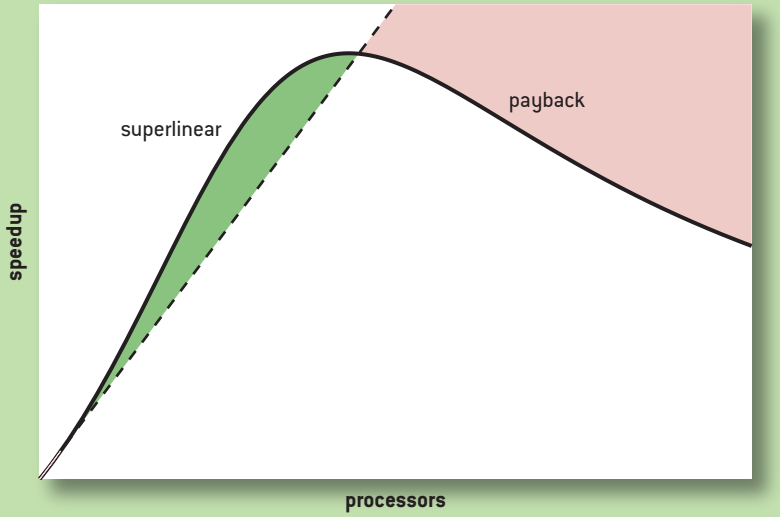
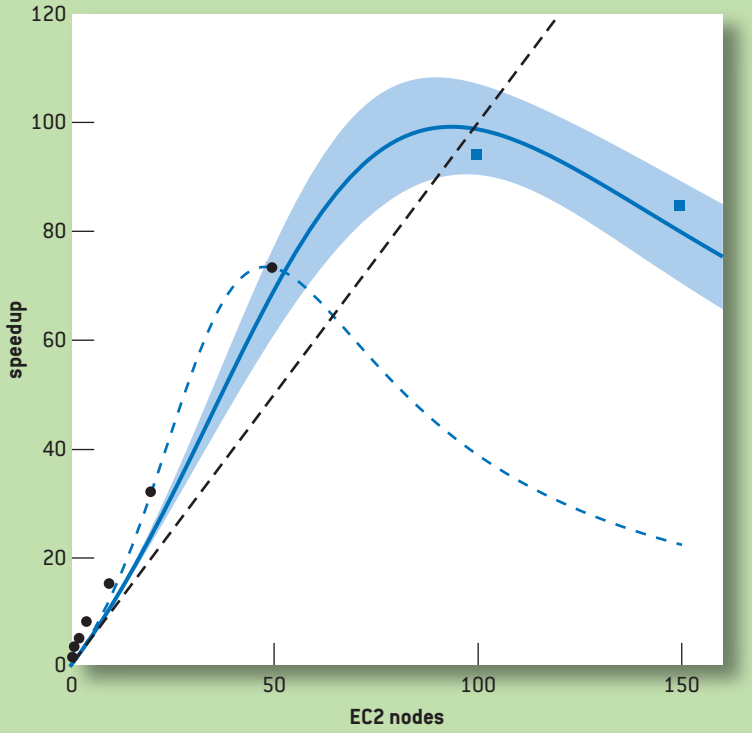


FIGURE 6

USL Analysis of p ≤ 50 BigMem Nodes (Solid Blue Curve) with Figure 4 (Dashed Blue Curve) Inset for Comparison



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

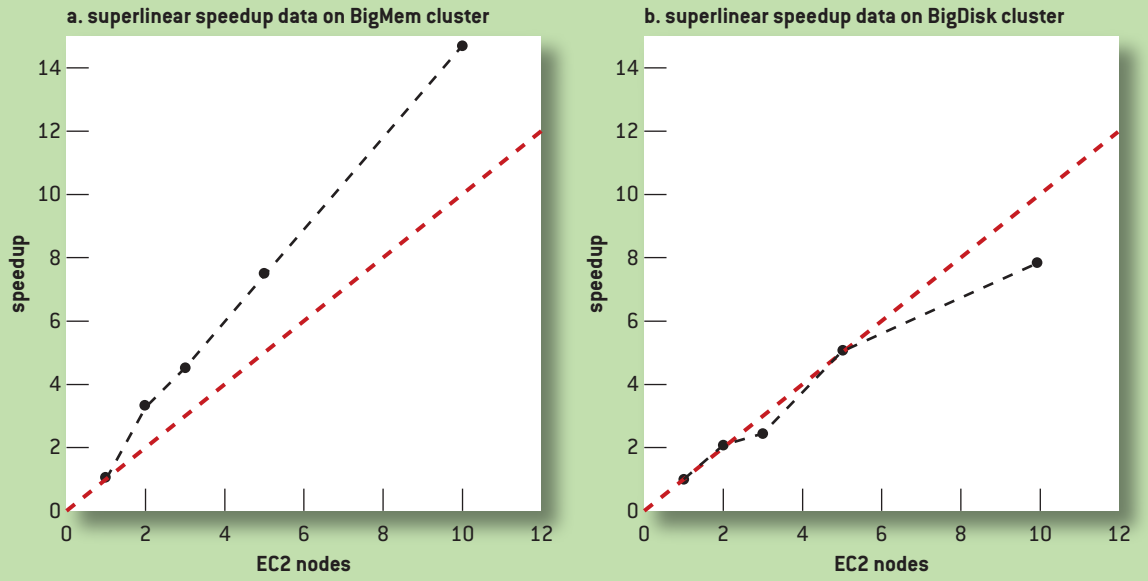
In a typical load test or performance testing environment, each load point in plots such as those in figures 9-11 represents the time-series average for a single run. The common excuse for not doing multiple runs is lack of time, which is more of an indictment of the particular in-house engineering philosophy than scheduling constraints. Even worse, determining the measurement error is not possible from a single run. If you don't know the measurement error, how are you going to know when something is wrong?

To make a statistical determination of the error in our 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 (r.e.), 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 is the ratio of

FIGURE 7

Superlinear Hadoop Terasort Speedup Essentially Eliminated by Increasing Nodal Disk I/O Bandwidth



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 three percent, which is nominal, to nine percent, which likely warrants further attention. This variation in the measurement error does not mean that the measurement technique is unreliable; rather, it means there is a higher degree of dispersion or variance 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 measured 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."*[16]

Some of the Yahoo team's sources of variability may differ from ours (e.g., the 10-times-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."*[17]

A SURPRISING HYPOTHESIS

The physical cluster configuration used by the Yahoo benchmark team consisted of nodes with two quad-core Xeon processors (i.e., a total of eight cores per node) and four SATA disks.[17] This is very similar to the BigDisk EC2 configuration in table 1. We therefore repeated our TeraSort scalability measurements on the BigDisk cluster. The results for `p = 2, 3, 5,` and `10` clusters are compared in figure 7.

Consistent with figure 4, the BigMem speedup values in figure 7a are superlinear, whereas the

BigDisk nodes in figure 7b 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 I/O 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 the BigMem and BigDisk configurations.

BigMem has the larger memory configuration, which possibly provides more CentOS buffer caching for the TeraSort data, and that could be the source of the capacity boost associated with the negative USL contention coefficient described earlier. 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 the case of 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 in the BigMem configuration. The single-disk DataNode in figure 3 implies that all disk I/O is serialized. In this sense, when disk writes (including replications) occur, TeraSort is I/O-bound—most particularly in the single-node case. As the cluster configuration gets larger, this latent I/O 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 configurations therefore exhibit runtimes that are shorter than the single-node case, which results in the superlinear speedup values shown in figure 7a.

Conversely, although the BigDisk configuration has a smaller amount of physical memory per node, it has four disks per DataNode, which means each node has greater disk bandwidth to accommodate more concurrent I/O. TeraSort is therefore far less likely to become I/O-bound. Since there is no latent single-node I/O constraint, there cannot be any capacity boost at play. As a result, the speedup values are more orthodox and fall into the sublinear region of figure 7b.

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 $1/p$.

CONSOLE STACK TRACES

The next step was to try and validate the I/O bottleneck hypothesis in terms of Hadoop metrics collected during each run. While TeraSort was running on certain BigMem configurations, task failures were observed in the Hadoop JobClient console that communicates with the Hadoop JobTracker (see appendix B). The following is an abbreviated form of a failed task status with the salient identifiers shown in bold.

```
14/10/01 21:53:41 INFO mapred.JobClient: Task Id :  atte
mpt_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)
```

Since the TeraSort job continued and all tasks ultimately completed successfully, we originally discounted these failure reports. Later, with the earlier I/O bottleneck hypothesis in mind, we realized that 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, the progress of that Reduce task became retrograde. Moreover, given that the failure in the stack trace above involved the Java class `DFSOutputStream`, we surmised that 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:

```
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)
```

This record indicates that the Reduce task actually failed on the Hadoop cluster, as opposed to the JobClient. Since the failure occurred during the invocation of `DFSOutputStream`, it further suggests that there was an issue while physically writing data to HDFS. Furthermore, a subsequent record in the log with the same task ID,

```
ReduceAttempt TASK_TYPE="REDUCE" TASKID="task_201410011835_0002_r_000000" TASK_ATTEMPT_
ID= "attempt_201410011835_0002_r_000000_1" TASK_STATUS= "SUCCESS"
```

had a newer `TASK ATTEMPT ID` (namely, a trailing 1 instead of a trailing 0) that was successful.

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

Table 3 has 12 rows corresponding to 12 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. The most significant of these metrics were extracted by parsing the log with Hadoop log tools.[13]

As described earlier, the 840 Map tasks in the first column are determined by the TeraSort job partitioning 100 (binary) GB of data into 128 (decimal) MB HDFS blocks. No Map failures occurred. The fourth column shows that the total number of Reduce tasks was set to three times the number of cluster nodes (`p = 1` in this case). The fifth column reveals that 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 job runtimes in the last column are used to determine the average

TABLE 2 **Runtime error analysis**

T_1	=	13057	±	606	seconds (r.e.5%)
T_2	=	6347	±	541	seconds (r.e.9%)
T_3	=	4444	±	396	seconds (r.e.9%)
T_5	=	2065	±	147	seconds (r.e.7%)
T_{10}	=	893	±	27	seconds (r.e.3%)

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

runtime.

For a single BigMem node, T_1 = 13057078.67 milliseconds is in agreement with table 2. Additional statistical analysis reveals a strong correlation between the number of Reduce task retries and longer runtimes. If the average single-node runtime T_1 is longer than successive values of $p\ T_p$, the speedup, as defined earlier in this article, will be superlinear.

WHENCE REDUCE FAILS?

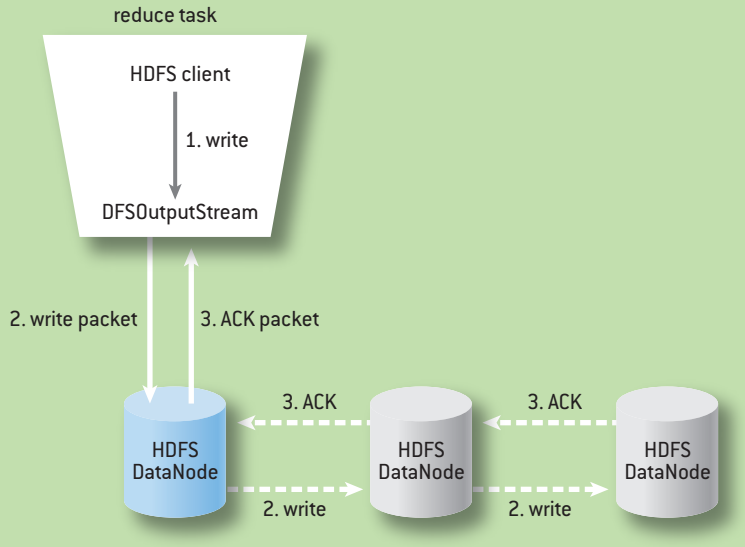
The number of failed Reduces in table 3 indicates that a write failure in the Reduce task causes it to retry the write operation—possibly multiple times. In addition, failed Reduce tasks tend to incur longer runtimes as a consequence of those additional retries. The only question remaining is, what causes 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.

Closer scrutiny of the earlier failed Reduce stack trace reveals the following lines, with important keywords shown in bold:

```
ReduceAttempt TASK_TYPE="REDUCE" ... TASK_STATUS="FAILED" … ERROR="java.
io.IOException: All datanodes are bad. Aborting …
…
```

FIGURE 8

HDFS DataNode Pipeline Showing Single Replication (Blue) and Default Triple Replication(Blue and Gray)



`.setupPipelineForAppendOrRecovery(DFSOutputStream.java:1000)`

The "All datanodes are bad" Java IOException means that the HDFS DataNode pipeline in figure 8 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 operating smoothly, a Reduce task makes a call into the `HDFSClient`, which then initiates the creation of a HDFS DataNode pipeline (see figure 8). The `HDFSClient` opens a `DFSOutputStream` and readies it for writing ("1. Write" in figure 8) 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" in figure 8), 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" in figure 8) from each DataNode that the write operation completed successfully.

When an exception is thrown (e.g., in the stack trace), the `DFSOutputStream` attempts to remedy the situation by reprocessing the packets to complete the HDFS write operation. 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 that the DataNodes will be able to keep up and respond with acknowledgments. If, however, the underlying I/O 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 as AWOL.

The `DFSOutputStream` throws an I/O exception that propagates back up to the Reduce task in figure 8. Since the Reduce task doesn't know how to handle this I/O 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.

To elaborate briefly on how an HDFS packet can go AWOL, one possibility is that TeraSort attempts to write 100 GB of data through a 32-GB buffer cache. The first 32 GB of data may be written very quickly to memory while CentOS is asynchronously writing that data to disk. If TeraSort is writing to the buffer cache faster than it can commit writes to disk, however, CentOS will exhaust the buffer cache. At that point, writes become synchronous, which makes them appear to slow down by several orders of magnitude relative to memory operations. If too many HDFS packets become enqueued because of the slower synchronous writes, then some HDFS packets could time out.

Given the mechanism for Reduce failures based on AWOL HDFS write packets, we can turn that operational insight around to construct a list of simple tactics for dealing with the associated retries and runtime stretching:

- Resize the buffer cache.
- Tune kernel parameters to increase I/O throughput.
- Reconfigure Hadoop default timeouts.

If maintaining a BigMem-type cluster is dictated by nonengineering requirements (e.g., budgetary constraints), then any of these steps could prove helpful in ameliorating superlinear effects.

CONCLUSION

The large number of controlled measurements performed while running Hadoop TeraSort on Amazon EC2 exposed the underlying causes of superlinearity, which would otherwise be difficult to resolve in the wild. Fitting our speedup data to the USL performance model produced a negative contention coefficient ($\sigma < 0$), a telltale sign that superlinearity was present 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_x in equation 3. At that point, Hadoop TeraSort superlinear scalability returns to being sublinear in the payback region. The cluster size p_x provides an estimate of the minimal node capacity needed to ameliorate superlinear speedup on BigMem clusters.

Although superlinearity is a reproducibly measurable phenomenon, like perpetual motion it is ultimately a performance illusion. For TeraSort on BigMem, the apparent capacity boost—identified by negative σ in USL—can be traced to successively relaxing the latent I/O bandwidth constraint per node as the cluster size grows. This I/O 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, in the simplest case, then successive speedup measurements will be superlinear. Increasing the I/O bandwidth per node, as we did with BigDisk clusters, diminishes or eliminates superlinear speedup by reducing T_1 stretching.

This USL analysis suggests that superlinear scalability is not peculiar to TeraSort on Hadoop but may arise with any MapReduce application. Superlinear speedup has also been reported to occur in relational database systems.[2] For high-performance computing applications, however, superlinear

speedup may have different causes than presented here.[4,14,22]

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 shown to work on a small number of cluster nodes, the Hadoop framework facilitates scaling it out to an arbitrarily large number of nodes with no additional effort. For 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.[28]

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 additional and potentially increasing 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 often far exceed the flat scalability assumption (see Memcached in appendix A). The USL provides a valuable tool to software engineers who want to analyze Hadoop scalability.

ACKNOWLEDGMENTS

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

APPENDIX A: SUBLINEAR SCALABILITY EXAMPLES

For comparison with the superlinear scalability analysis discussed in this article, this appendix presents the USL analysis of three topical applications that exhibit orthodox scalability: Varnish, Memcached, and ZooKeeper.

VARNISH SCALABILITY

The first example quantifies the scalability of the Varnish HTTP accelerator. The speedup data (dots) in figure 9a are derived from HTTP GET operations.[18] They confirm, very strikingly, that Varnish is a rare example of an application that exhibits near-linear scalability.

A cautionary remark is in order. Throughout this article, we are considering the speedup S_p as a function of p, the number of distributed cluster nodes or processors. In figure 9, and later in figure 10, p represents the number of *processes*, not processors. The platform on which data is collected (e.g., an HTTP server) has a fixed number of processors for all the speedup measurements. This is the typical situation found in a QA/load test environment. Remarkably, the same USL in equation 2 accommodates both perspectives.[9]

USL regression analysis of the Varnish speedup data for $p \leq 400$ processes (red curve in figure 9a) produces a contention coefficient value of $\sigma = 0.000169$. Although it is not zero, it is indeed very small and explains the near-linear scalability as deriving from a minimal sharing of resources (compare with figure 2b). The 95 percent confidence bands are shown in blue.

Similarly, the coherency coefficient value is $\kappa = 0$, so a maximum of the type shown in figure 2d cannot develop. Although it is not apparent from the measurements in figure 9a, the USL predicts that the near-linear scalability cannot be expected to scale indefinitely because of the theoretical

FIGURE 9

USL Analysis of Varnish Scalability

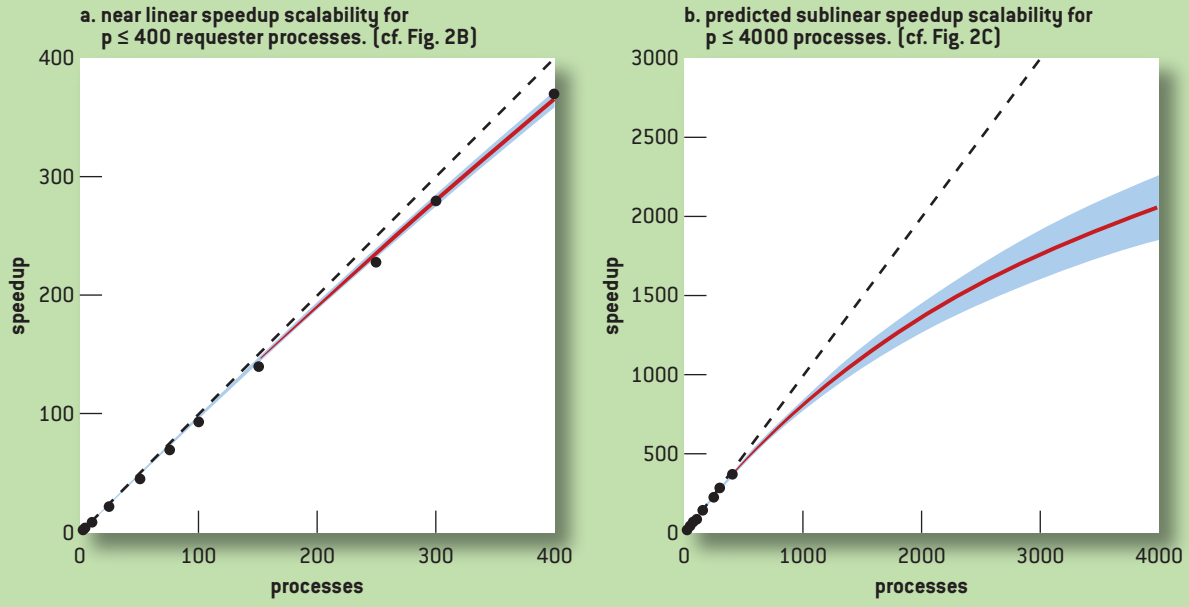
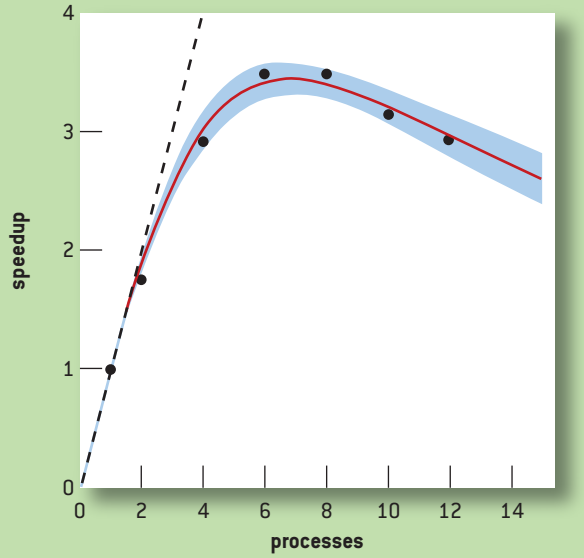


FIGURE 10

USL Analysis of Memcached Scalability



ceiling in the speedup at $S_{ceiling} = 1/\sigma = 5917$ (not shown). This prediction is made more evident by the red curve in figure 9b, where the USL curve in figure 9a has been projected out to $p = 4000$ processes. Since the USL fit to the Varnish data produces a very small value of σ and a κ of zero, the measurements and the model are consistent, and that provides data validation of the type mentioned in the first section of this article.

MEMCACHED SCALABILITY

The second, and more typical, example of orthodox scalability is provided by Memcached,[11] the caching daemon intended to decrease the load on a back-end database by storing objects as key-value pairs in memory. Speedup measurements (dots), derived from key-value retrieval operations, are shown in figure 10. Once again, p represents processes, not processors.

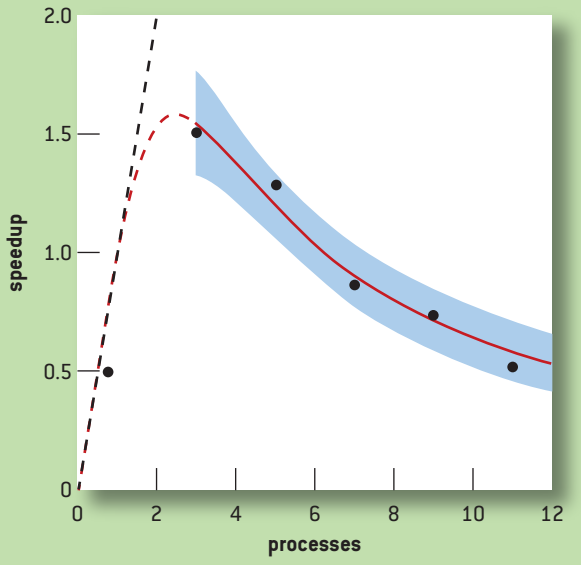
Applying USL regression analysis to the data verifies that a maximum in the speedup occurs at around a half-dozen thread processes in Memcached version 1.2.8. The USL model (red curve) in figure 10 is an empirical expression of the scaling characteristic depicted schematically in figure 2d with nonzero values for both coefficients, σ and κ.

There's little point in predicting scalability beyond $p = 6$ threads. Rather, the incentive should be to remove the peak, not characterize it with greater precision. For Memcached, the first step is to explain why the peak occurs where it does based on the regression values of the σ and κ coefficients. That explanation may then provide some engineering insight into ameliorating the situation. In fact, Sun Microsystems developed a software patch for Memcached 1.3.2, on a Solaris SPARC multicore platform, that moved the peak out to around $p = 50$ threads.[11]

ZOOKEEPER SCALABILITY

As the final and rather exotic example, the USL model is applied to the Apache ZooKeeper scalability

FIGURE 11 **USL Analysis of ZooKeeper Scalability**



data[15] in figure 11. Speedup measurements (dots) are based on a mix of read and write operations. Notice that all the speedup data are not only sublinear (i.e., to the right of the linear bound indicated by a dashed line), but also decreasing in a way that is consistent with purely negative scalability (compare with figure 2d).

In this distributed coordination application, votes must be exchanged between at least three distributed servers in order to determine a majority. Because the workload consists almost entirely of exchanging data between pairs of servers, the coherency penalty is very high, with a coefficient of `κ = 0.1635`, while the contention coefficient, σ, is relatively small. Consequently, a severe maximum occurs in the USL model (dotted curve), and the best-case speedup data start on the "downhill" side of the USL maximum (red curve). In other words, purely retrograde scalability is optimal for ZooKeeper.

This example serves as a stark reminder that all computer system performance is about tradeoffs. Sometimes the *best* means the *least worst* that can be achieved under what would otherwise be considered utterly adverse constraints.

APPENDIX B: HADOOP FRAMEWORK OVERVIEW

The Hadoop framework is designed to facilitate writing large-scale, data-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.
- **HDFS:** a distributed file system that allows data to be stored on any node and to be 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 tasks.** The function of the Map task is to take a slice of the entire input data set and transform it into key-value pairs, commonly denoted by `<key,value>` in the context of MapReduce. (See the detailed Map tasks data flow 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 also sorts the data by key and stores the sorted `<k,v>` objects so that they can easily be exchanged with a Reduce task.
- **Reduce tasks.** 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 the value of `v` is a list `[v1,v2, ...]` of all the values that are `< k, [v1,v2, ...]>` objects whose key is the specific key across the entire input data set. (See the detailed Reduce tasks dataflow in Node 1 of figure 3.)

A MapReduce application processes its input data set using the following workflow:

1. On startup, the application creates and schedules one Map task per slice of the input data set, as well as creating a user-defined number of Reduce tasks.
2. 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 together.
3. 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, [v1,v2,...]>` objects. This is referred to as the *shuffle exchange* process, 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 execution service called the JobTracker and a number of slave services called TaskTrackers.[27]

The JobTracker is responsible for scheduling tasks and transferring them to the TaskTrackers residing on each cluster node. Another feature of the JobTracker is that it can detect and restart tasks that might fail. It provides a level of fault tolerance to application execution. The user interacts with the Hadoop framework via a JobClient component, such as TeraSort, which 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 developers with a reliable distributed file service that allows Hadoop applications to read and write very large data files at high throughput to fixed-size blocks (128 MB in the case of TeraSort[3]) across the cluster. The master node in an HDFS cluster is the NameNode, which is responsible for regulating client access to files, as well as managing the file-system namespace by mapping file blocks to its storage location, which can reside on the DataNodes (i.e., slave nodes to the NameNode). A key feature of HDFS is its built-in resilience to node or disk failure, which is accomplished by replicating blocks across multiple DataNodes. The default replication factor is three, but this is set to one for the TeraSort workload.

REFERENCES

1. Apache Whirr; https://whirr.apache.org.
2. Calvert, C., Kulkarni, D. 2009. *Essential LINQ*. Boston, MA: Pearson Education Inc.
3. Cloudera Hadoop; http://www.cloudera.com/content/cloudera/en/downloads/cdh/cdh-4-7-0.html.
4. Eijkhout, V. 2014. Introduction to high-performance scientific computing. Lulu.com.
5. Feynman, R. P. The Papp perpetual motion engine; http://hoaxes.org/comments/papparticle2.html.
6. Gunther, N. J. 1993. A simple capacity model of massively parallel transaction systems. In *Proceedings of International Computer Measurement Group Conference*; http://www.perfdynamics.com/Papers/njgCMG93.pdf.
7. Gunther, N. J. 2001. Performance and scalability models for a hypergrowth e-commerce Web site. In *Performance Engineering, State of the Art and Current Trends*. (Eds.) Dumke, R. R., Rautenstrauch, C., Schmietendorf, A., Scholz, A. Lecture Notes in Computer Science 2047: 267-282. Springer-Verlag.
8. Gunther, N. J. 2007. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Springer; http://www.springer.com/computer/communication+networks/book/978-3-540-26138-4.
9. Gunther, N. J. 2008. A general theory of computational scalability based on rational functions; http://arxiv.org/abs/0808.1431.
10. Gunther, N. J. 2012. PostgreSQL scalability analysis deconstructed;http://perfdynamics.blogspot.com/2012/04/postgresql-scalability-analysis.html.
11. Gunther, N. J., Subramanyam, S., Parvu, S. 2010. Hidden scalability gotchas in Memcached and friends. VELOCITY Web Performance and Operations Conference;

http://velocityconf.com/velocity2010/public/schedule/detail/13046.
12. Haas, R. 2011. Scalability, in graphical form, analyzed; 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., Patterson, D. A. 1996. *Computer Architecture: A Quantitative Approach*. Second edition. Waltham, MA: Morgan Kaufmann.
15. Hunt, P., Konar, M., Junqueira, F. P., Reed, B. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the Usenix Annual Technical Conference*; https://www.usenix.org/legacy/event/usenix10/tech/full_papers/Hunt.pdf.
16. O'Malley, O. 2008. TeraByte Sort on Apache Hadoop; http://sortbenchmark.org/YahooHadoop.pdf.
17. O'Malley, O., Murthy, A. C. 2009. Winning a 60 second dash with a yellow elephant; http://sortbenchmark.org/Yahoo2009.pdf.
18. Parvu, S. 2012. Private communication.
19. Performance Dynamics Company. 2014. How to quantify scalability (including calculator tools); http://www.perfdynamics.com/Manifesto/USLscalability.html.
20. Schwartz, B. 2011. 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/.
21. sFlow. 2010. SDN analytics and control using sFlow standard—Superlinear; http://blog.sflow.com/2010/09/superlinear.html.
22. Stackoverflow. Where does superlinear speedup come from?; http://stackoverflow.com/questions/4332967/where-does-super-linear-speedup-come-from.
23. Sun Fire X2270 M2 super-linear scaling of Hadoop TeraSort and CloudBurst benchmarks. 2010; https://blogs.oracle.com/BestPerf/entry/20090920_x2270m2_hadoop.
24. Sutter, H. 2008. Going superlinear. *Dr. Dobb's Journal* 33(3); http://www.drdobbs.com/cpp/going-superlinear/206100542.
25. Sutter, H. 2008. Super linearity and the bigger machine. *Dr. Dobb's Journal* 33(4); http://www.drdobbs.com/parallel/super-linearity-and-the-bigger-machine/206903306.
26. TechCrunch. 2015. AuroraTek tried to pitch us a gadget that breaks the laws of physics at CES; http://techcrunch.com/2015/01/08/auroratek-tried-to-pitch-us-a-gadget-that-breaks-the-laws-of-physics-at-ces/.
27. White, T. 2012. *Hadoop: The Definitive Guide. Storage and Analysis at Internet Scale*, 3rd edition. O'Reilly Media, Inc.
28. Yahoo! Hadoop Tutorial; https://developer.yahoo.com/hadoop/tutorial/module1.html#scalability.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

NEIL J. GUNTHER, M.Sc., Ph.D., is a researcher and teacher at Performance Dynamics (www.perfdynamics.com), where he originated the PDQ open-source performance analyzer and the USL and wrote some books based on both. He is a senior member of ACM and received the A.A. Michelson Award in 2008. Blogging at http://perfdynamics.blogspot.com is sporadic, but tweets are more consistent as @DrQz.

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. He holds an M.S. in applied mathematics from SUNY at Stony Brook and is a member of ACM.

KRISTOFER TOMASETTE (ktomasette@gmail.com) is a principal 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. He first observed superlinear scaling in 2011 while attempting to establish the capacity of a Hadoop cluster. Later he applied the USL to the Sirius project (http://comcast.github.io/sirius/), a distributed trace library, and several middleware applications.

© 2015 ACM 1542-7730/14/0400 $10.00