Applying Simple Performance Models to Understand Inefficiencies in Data-Intensive Computing

Elie Krevat*, Tomer Shiran*, Eric Anderson†,
Joseph Tucek†, Jay J. Wylie†, Gregory R. Ganger*
**Carnegie Mellon University* †*HP Labs*

CMU-PDL-11-103
Feb 2011

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

New programming frameworks for scale-out parallel analysis, such as MapReduce and Hadoop, have become a cornerstone for exploiting large datasets. However, there has been little analysis of how these systems perform relative to the capabilities of the hardware on which they run. This paper describes a simple analytical model that predicts the theoretic ideal performance of a parallel dataflow system. The model exposes the inefficiency of popular scale-out systems, which take 3–13× longer to complete jobs than the hardware should allow, even in well-tuned systems used to achieve record-breaking benchmark results. Using a simplified dataflow processing tool called Parallel DataSeries, we show that the model's ideal can be approached (i.e., that it is not wildly optimistic), coming within 10–14% of the model's prediction. Moreover, guided by the model, we present analysis of inefficiencies which exposes issues in both the disk and networking subsystems that will be faced by any DISC system built atop standard OS and networking services.

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM, Intel, LSI, Microsoft Research, NEC Laboratories, NetApp, Oracle, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Yahoo! Labs) for their interest, insights, feedback, and support. This research was sponsored in part by an HP Innovation Research Award and by CyLab at Carnegie Mellon University under grant DAAD19–02–1–0389 from the Army Research Office.

Keywords: data-intensive computing, cloud computing, performance and efficiency

1 Introduction

"Data-intensive scalable computing" (DISC) refers to a rapidly growing style of computing characterized by its reliance on huge and growing datasets [9]. Driven by the desire and capability to extract insight from such datasets, DISC is quickly emerging as a major activity of many organizations. With massive amounts of data arising from such diverse sources as telescope imagery, medical records, online transaction records, and web pages, many researchers are discovering that statistical models extracted from data collections promise major advances in science, health care, business efficiencies, and information access. Indeed, statistical approaches are quickly bypassing expertise-based approaches in terms of efficacy and robustness.

To assist programmers with DISC, new programming frameworks (e.g., MapReduce [11], Hadoop [1] and Dryad [15]) have been developed. They provide abstractions for specifying data-parallel computations, and they also provide environments for automating the execution of data-parallel programs on large clusters of commodity machines. The map-reduce programming model, in particular, has received a great deal of attention, and several implementations are publicly available [1, 24].

These frameworks can scale jobs to thousands of computers, however, they currently focus on scalability without concern for efficiency. Worse, anecdotal experiences indicate that they fall far short of fully utilizing hardware resources, neither achieving balance among the hardware resources, nor achieving full goodput on some bottleneck resource. This effectively wastes a large fraction of the computers over which jobs are scaled. If these inefficiencies are real, the same work could (theoretically) be completed at much lower costs. An ideal approach would provide maximum scalability for a given computation without wasting resources such as the CPU or disk. Given the widespread use and scale of DISC systems, it is important that we move closer to ideal.

An important first step is understanding the degree, characteristics, and causes of inefficiency. Unfortunately, little help is currently available. This paper begins to fill the void by developing and using a simple model of "ideal" map-reduce job runtimes and the evaluation of systems relative to it. The model's input parameters describe basic characteristics of the job (e.g., amount of input data, degree of filtering in the map and reduce phases), of the hardware (e.g., per-node disk and network throughputs), and of the framework configuration (e.g., replication factor). The output is the theoretic ideal job runtime.

Our goal for this model is not to accurately predict the runtime of a job on any given system, but to indicate what the runtime theoretically *should be*. An ideal run is "hardware-efficient," meaning that the realized goodput matches the maximum goodput for some bottleneck resource, given its use (i.e., amount of data moved over it). Our model can expose how close (or far) a given system is from this ideal. Such throughput will not occur, for example, if the framework does not provide sufficient parallelism to keep the bottleneck fully utilized, or if it makes poor use of a particular resource (e.g., inflating network traffic). In addition, our model can quantify resources wasted due to imbalance—in an unbalanced system, one under-provisioned resource bottlenecks the others, which are wastefully overprovisioned.

To illustrate these issues, we applied the model to a number of benchmark results (e.g., for TeraSort and PetaSort) touted in the industry. These presumably well-tuned systems achieve runtimes that are 3–13$\times$ longer than the ideal model suggests should be possible. We also report on our own experiments with Hadoop, confirming and partially explaining sources of inefficiency.

To confirm that the model's ideal is achievable, and explore foundation induced inefficiencies, we present results from an efficient parallel dataflow system called Parallel DataSeries (PDS). PDS lacks many features of the other frameworks, but its careful engineering and stripped-down feature-set demonstrate that near-ideal hardware efficiency (within $\sim$20%) is possible. In addition to validating the model, PDS provides an interesting foundation for subsequent analyses of the incremental costs associated with features, such as distributed file system functionality, dynamic task distribution, fault tolerance, and task replication.

To demonstrate the utility of the model, we use it to explore some sources of unexpected inefficiency encountered by PDS, which we expected to be even closer to ideal than it was. For example, we found

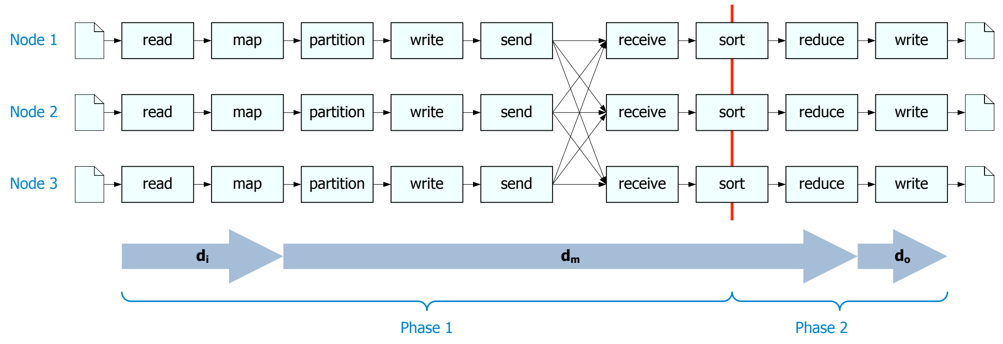


Figure 1: A map-reduce dataflow

that variability in file system performance across nodes in a homogeneous cluster induces a straggler effect unrelated to failures or consistently slow components, slowing overall job completion times. In addition, we found that the all-to-all communication pattern inherent to the "shuffle" step of map-reduce computations (in which map output is distributed among nodes performing reduces) results in steadily decreasing aggregate network throughput. Since these issues arise from characteristics of the underlying OS and networking infrastructure, they are faced by any DISC framework built on these standard underlying services.

2 Dataflow parallelism and map-reduce computing

Today's data-intensive computing derives much from earlier work on parallel databases. Broadly speaking, data is read from input files, processed, and stored in output files. The dataflow is organized as a pipeline in which the output of one operator is the input of the following operator. DeWitt and Gray [12] describe two forms of parallelism in such dataflow systems: partitioned parallelism and pipelined parallelism. Partitioned parallelism is achieved by partitioning the data and splitting one operator into many running on different processors. Pipelined parallelism is achieved by streaming the output of one operator into the input of another, so that the two operators can work in series on different data at the same time.

Google's MapReduce[1] [11] offers a simple programming model that facilitates development of scalable parallel applications that process a vast amount of data. Programmers specify a *map* function that generates values and associated keys from each input data item and a *reduce* function that describes how all data matching each key should be combined. The runtime system handles details of scheduling, load balancing, and error recovery. Hadoop [1] is an open-source implementation of the map-reduce model. Figure 1 illustrates the pipeline of a map-reduce computation involving three nodes. The computation is divided into two phases, labeled Phase 1 and Phase 2.

Phase 1 begins with the reading of the input data from disk and ends with the sort operator. It includes the map operators and the exchange of data over the network. The first write operator in Phase 1 stores the output of the map operator. This "backup write" operator is optional, but used by default in the Google and Hadoop implementations of map-reduce, serving to increase the system's ability to cope with failures or other events that may occur later.

Phase 2 begins with the sort operator and ends with the writing of the output data to disk. In systems that replicate data across multiple nodes, such as the GFS [13] and HDFS [3] distributed file systems used

[1]We refer to the programming model as map-reduce and to Google's implementation as MapReduce.

with MapReduce and Hadoop, respectively, the output data must be sent to all other nodes that will store the data on their local disks.

Parallelism: In Figure 1, partitioned parallelism takes place on the vertical axis; the input data is split between three nodes, and each operator is, in fact, split into three sub-operators that each run on a different node. Pipelined parallelism takes place on the horizontal axis; each operator within a phase processes data units (e.g., records) as it receives them, rather than waiting for them all to arrive, and passes data units to the next operator as appropriate. The only breaks in pipelined parallelism occur at the boundary between phases. As shown, this boundary is the sort operator. The sort operator can only produce its first output record after it has received all of its input records, since the last input record received might be the first in sorted order.

Quantity of data flow: Figure 1 also illustrates how the amount of data "flowing" through the system changes throughout the computation. The amount of input data per node is d_i, and the amount of output data per node is d_o. The amount of data per node produced by the map operator and consumed by the reduce operator is d_m. In most applications, the amount of data flowing through the system either remains the same or decreases (i.e., $d_i \geq d_m \geq d_o$). In general, the mapper will implement some form of select, filtering out rows, and the reducer will perform aggregation. This reduction in data across the stages can play a key role in the overall performance of the computation. Indeed, Google's MapReduce includes "combiner" functions to move some of the aggregation work to the map operators and, hence, reduce the amount of data involved in the network exchange [11]. Many map-reduce workloads resemble a "grep"-like computation, in which the map operator decreases the amount of data ($d_i \gg d_m$ and $d_m = d_o$). In others, such as in a sort, neither the map nor the reduce function decrease the amount of data ($d_i = d_m = d_o$).

2.1 Related work

Concerns about the performance of map-reduce style systems have emerged from the parallel databases community, where similar data processing tasks have been tackled by commercially available systems. In particular, Stonebraker et al. compare Hadoop to a variety of DBMSs and find that Hadoop can be up to 36x slower than a commercial parallel DBMS [29]. In a workshop position paper [8], it was argued that many parallel systems (especially map-reduce systems) have focused almost exclusively on absolute throughput and high-end scalability to the detriment of other worthwhile metrics, such as efficiency. This paper builds on such observations by exploring efficiency quantitatively.

In perhaps the most relevant prior work, Wang et al. use simulation to evaluate how certain design decisions (e.g., network layout and data locality) will effect the performance of Hadoop jobs [33]. Specifically, their MRPerf simulator instantiates fake jobs, which impose fixed times (e.g., job startup) and input-size dependent times (cycles/byte of compute) for the Hadoop parameters under study. The fake jobs generate network traffic (simulated with ns-2) and disk I/O (also simulated). Using execution characteristics accurately measured from small instances of Hadoop jobs, MRPerf accurately predicts (to within 5-12%) the performance of larger clusters. Although simulation techniques like MRPerf are useful for exploring different designs, by relying on measurements of actual behavior (e.g., of Hadoop) such simulations will also emulate any inefficiencies particular to the specific implementation simulated. Our goal is different—rather than seeking to predict the performance of any actual system, we build a simple model to guide exploration of how far from optimal a system's performance may differ.

3 Performance model

This section presents a model for the runtime of a map-reduce job on a perfectly hardware-efficient system. It includes the model's assumptions, parameters, and equations, along with a description of common

Symbol	Definition
n	The number of nodes in the cluster.
D_w	The aggregate disk *write* throughput of a single node. A node with four disks, where each disk provides 65 MB/s writes, would have D = 260 MB/s.
D_r	The aggregate disk *read* throughput of a single node.
N	The network throughput of a single node.
r	The replication factor used for the job's output data. If no replication is used, $r = 1$.
i	The total amount of input data for a given computation.
$d_i \left(= \frac{i}{n}\right)$	The amount of input data per node, for a given computation.
$d_m \left(= \frac{i \cdot e_M}{n}\right)$	The amount of data per node after the map operator, for a given computation.
$d_o \left(= \frac{i \cdot e_M \cdot e_R}{n}\right)$	The amount of output data per node, for a given computation.
$e_M \left(= \frac{d_m}{d_i}\right)$	The ratio between the map operator's output and its input.
$e_R \left(= \frac{d_o}{d_m}\right)$	The ratio between the reduce operator's output and its input.

Table 1: Modeling parameters that include I/O speeds and workload properties.

workloads.

Assumptions: For a large class of data-intensive workloads, which we assume for our model, computation time is negligible in comparison to I/O speeds. Among others, this assumption holds for grep- and sort-like jobs, such as those described by Dean and Ghemawat [11] as being representative of most MapReduce jobs at Google, but may not hold in other settings. For workloads fitting the assumption, pipelined parallelism can allow non-I/O operations to execute entirely in parallel with I/O operations, such that overall throughput for each phase will be determined by the I/O resource (network or storage) with the lowest effective throughput. For modeling purposes, we presume that the network is capable of full bisection bandwidth (e.g., full crossbar). Although many current networks are not capable of this, we have reason to believe oversubscription was not the bottleneck for any of the systems we examine. Further, there is ongoing research on building such networks out of commodity components [5].

The model assumes that input data is evenly distributed across all participating nodes in the cluster, that the cluster is dedicated to the one job, that nodes are homogeneous, and that each node retrieves its initial input from local storage. Most map-reduce systems are designed to fit these assumptions. The model also accounts for output data replication, assuming the common strategy of storing the first replica on the local disks and sending the others over the network to other nodes.

Deriving the model from I/O operations: Table 1 lists the I/O speed and workload property parameters of the model. They include amounts of data flowing through the system, which can be expressed either in absolute terms (d_i, d_m, and d_o) or in terms of the ratios of the map and reduce operators' output and input (e_M and e_R, respectively).

Table 2 identifies the I/O operations in each map-reduce phase for two variants of the sort operator. When the data fits in memory, a fast *in-memory sort* can be used. When it does not fit, an *external sort* is used, which involves sorting each batch of data in memory, writing it out to disk, and then reading and merging the sorted batches into one sorted stream. The $\frac{n-1}{n} d_m$ term appears in the equation, where n is the number of nodes, because in a perfectly-balanced system each node partitions and transfers that fraction of its mapped data over the network, keeping $\frac{1}{n}$ of the data for itself.

Incorporating the modeling parameters and patterns of I/O operations, Table 3 provides the model equations for the execution time of a map-reduce job in each of four scenarios, representing the cross-

	$d_m <$ memory (in-memory sort)	$d_m \gg$ memory (external sort)
Phase 1	Disk read (input): d_i Disk write (backup): d_m Network: $\frac{n-1}{n} d_m$	Disk read (input): d_i Disk write (backup): d_m Network: $\frac{n-1}{n} d_m$ Disk write (sort): d_m
Phase 2	Network: $(r-1)\,d_o$ Disk write (output): rd_o	Disk read (sort): d_m Network: $(r-1)\,d_o$ Disk write (output): rd_o

Table 2: I/O operations in a map-reduce job.

	$d_m < memory$ (in-memory sort)
Without backup write	$\frac{i}{n}\left(max\left\{\frac{1}{D_r}, \frac{\frac{n-1}{n}e_M}{N}\right\} + max\left\{\frac{re_Me_R}{D_w}, \frac{e_Me_R(r-1)}{N}\right\}\right)$
With backup write	$\frac{i}{n}\left(max\left\{\frac{1}{D_r} + \frac{e_M}{D_w}, \frac{\frac{n-1}{n}e_M}{N}\right\} + max\left\{\frac{re_Me_R}{D_w}, \frac{e_Me_R(r-1)}{N}\right\}\right)$
	$d_m \gg memory$ (external sort)
Without backup write	$\frac{i}{n}\left(max\left\{\frac{1}{D_r} + \frac{e_M}{D_w}, \frac{\frac{n-1}{n}e_M}{N}\right\} + max\left\{\frac{e_M}{D_r} + \frac{re_Me_R}{D_w}, \frac{e_Me_R(r-1)}{N}\right\}\right)$
With backup write	$\frac{i}{n}\left(max\left\{\frac{1}{D_r} + \frac{2e_M}{D_w}, \frac{\frac{n-1}{n}e_M}{N}\right\} + max\left\{\frac{e_M}{D_r} + \frac{re_Me_R}{D_w}, \frac{e_Me_R(r-1)}{N}\right\}\right)$

Table 3: Model equations for the execution time of a map-reduce computation on a parallel dataflow system.

product of the Phase 1 backup write option (*yes* or *no*) and the sort type (*in-memory* or *external*). In each case, the per-byte time to complete each phase (map and reduce) is determined, summed, and multiplied by the number of input bytes per node $\left(\frac{i}{n}\right)$. The per-byte value for each phase is the larger (max) of that phase's per-byte disk time and per-byte network time. Using the last row (external sort, with backup write) as an example, the map phase includes three disk transfers and one network transfer: reading each input byte $\left(\frac{1}{D_r}\right)$, writing the e_M map output bytes to disk (the backup write; $\frac{e_M}{D_w}$), writing e_M bytes as part of the external sort $\left(\frac{e_M}{D_w}\right)$, and sending $\frac{n-1}{n}$ of the e_M map output bytes over the network $\left(\frac{\frac{n-1}{n}e_M}{N}\right)$ to other reduce nodes. The corresponding reduce phase includes two disk transfers and one network transfer: reading sorted batches $\left(\frac{e_M}{D_r}\right)$, writing e_Me_R reduce output bytes produced locally $\left(\frac{e_Me_R}{D_w}\right)$ and $(r-1)e_Me_R$ bytes replicated from other nodes $\left(\frac{(r-1)e_Me_R}{D_w}\right)$, and sending e_Me_R bytes produced locally to $(r-1)$ other nodes $\left(\frac{e_Me_R(r-1)}{N}\right)$. Putting all of this together produces the appropriate equation.

Applying the model to common workloads: Many workloads benefit from a parallel dataflow system because they run on massive datasets, either extracting and processing a small amount of interesting data or shuffling data from one representation to another. We focus on parallel sort and grep as representative examples in analyzing systems and validating our model.

For a grep-like job that selects a very small fraction of the input data, $e_M \approx 0$ and $e_R = 1$, meaning that only a negligible amount of data is (optionally) written to the backup files, sent over the network, and written to the output files. Thus, the best-case runtime is determined by the initial input disk reads:

$$t_{grep} = \frac{i}{nD_r} \tag{1}$$

A sort workload maintains the same amount of data in both the map and reduce phases, so $e_M = e_R = 1$.

If the amount of data per node is small enough to accommodate an in-memory sort and not warrant a Phase 1 backup, the top equation of Table 3 is used, simplifying to:

$$t_{sort} = \frac{i}{n}\left(max\left\{\frac{1}{D_r}, \frac{n-1}{nN}\right\} + max\left\{\frac{r}{D_w}, \frac{r-1}{N}\right\}\right) \tag{2}$$

Determining input parameters for the model: Appropriate parameter values are a crucial aspect of model accuracy, whether to evaluate how well a production system is performing or to determine what should be expected from a hypothetical system. The n and r parameters are system configuration choices that can be applied directly in the model for both production and hypothetical systems.

The amount of data flowing through various operators (d_i, d_m, or d_o) depend upon the characteristics of the map and reduce operators and of the data itself. For a production system, they can be measured and then plugged into a model that evaluates the performance of a given workload run on that system. For a hypothetical system, or if actual system measurements are not available, some estimates must be used, such as $d_i = d_m = d_o$ for sort or $d_m = d_o = 0$ for grep.

The determination of which equation to use, based on the backup write option and sort type choices, is also largely dependent on the workload characteristics, but in combination with system characteristics. Specifically, the sort type choice depends on the relationship between d_m and the amount of main memory available for the sort operator. The backup write option is a softer choice, worthy of further study, involving the time to do a backup write $\left(\frac{d_m}{D_w}\right)$, the total execution time of the job, and the likelihood of a node failure during the job's execution. Both Hadoop and Google's MapReduce always do the backup write, at least to the local file system cache.

Appropriate parameter values depend on what is being evaluated. If we are evaluating the efficiency of an entire software stack, from the OS up, then using nominal performance values (e.g., 1 Gbps for Ethernet) is appropriate. If the focus is on the programming framework, it is more appropriate to consider the goodput feasible after the losses in lower components (e.g., the operating system's TCP implementation), which can be determined via measurements. As we show, such measured values can be substantially lower then nominal raw performance, and can have surprising behavior.

4 Popular data-intensive computing systems far from optimal

Our model indicates that, though they may scale beautifully, popular data-intensive computing systems leave a lot to be desired in terms of efficiency. Figure 2 compares optimal times, as predicted by the model, to reported measurements of a few benchmark landmarks touted in the literature, presumably on well-tuned instances of the programming frameworks utilized. These results indicate that far more machines and disks are often employed than would be needed if the systems were hardware-efficient. The remainder of this section describes the systems and benchmarks represented in Figure 2 along with some of our own experiences with Hadoop.

Hadoop – TeraSort: In April 2009, Hadoop set a new record [21] for sorting 1 TB of data in the Sort Benchmark [20] format. The setup had the following parameters: $i = 1$ TB, $r = 1$, $n = 1460$, $D =$ 4 disks $\cdot$65 MB/s/disk[2] $= 260$ MB/s, $N = 110$ MB/s, $d_m = i/n = 685$ MB. With only 685 MB per node, the data can be sorted by the individual nodes in memory. A phase 1 backup write is not needed, given the short runtime. Equation 2 gives a best-case runtime of 8.86 seconds. After fine-tuning the system for this specific benchmark, Yahoo! achieved 62 seconds—7$\times$ slower. An optimal system using the same hardware would achieve better throughput with 209 nodes (instead of 1,460).

[2]For the Hadoop and Google Tera/Peta sorts, we conservatively assume 65 MB/sec, as actual disk speeds were not reported.

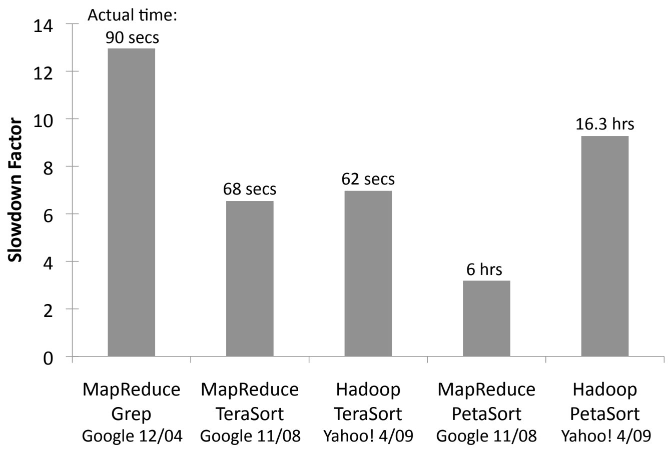


Figure 2: Published benchmarks of popular parallel dataflow systems. Each bar represents the reported throughput relative to the ideal throughput indicated by our performance model, parameterized according to a cluster's hardware.

MapReduce – TeraSort: In November 2008, Google reported TeraSort results for 1000 nodes with 12 disks per node [10]. The following parameters were used: $i = 1$ TB, $r = 1$, $n = 1000$, $D = 12 \cdot 65 = 780$ MB/s, $N = 110$ MB/s, $d_m = i/n = 1000$ MB. Equation 2 gives a best-case runtime of 10.4 seconds. Google achieved 68 seconds—over 6× slower. An optimal system using the same hardware would achieve better throughput with 153 nodes (instead of 1,000).

MapReduce – PetaSort: Google's PetaSort experiment [10] is similar to TeraSort, with three differences: (1) an external sort is required with a larger amount of data per node ($d_m = 250$ GB), (2) output was stored on GFS with three-way replication, (3) a Phase 1 backup write is justified by the longer runtimes. In fact, Google ran the experiment multiple times, and at least one disk failed during each execution. The setup is described as follows: $i = 1$ PB, $r = 3$, $n = 4000$, $D = 12 \cdot 65 = 780$ MB/s, $N = 110$ MB/s, $d_m = i/n = 250$ GB. The bottom cell of Table 3 gives a best-case runtime of 6,818 seconds. Google achieved 21,720 seconds—approximately 3.2× slower. An optimal system using the same hardware would achieve better throughput with 1,256 nodes (instead of 4,000). Also, according to our model, for the purpose of sort-like computations, Google's nodes appear to be overprovisioned with disks. In an optimal system, the network would be the bottleneck even if each node had only 6 disks instead of 12.

Hadoop – PetaSort: Yahoo!'s PetaSort experiment [21] is similar to Google's, with one main difference: The output was stored on HDFS with two-way replication. The setup is described as follows: $i = 1$ PB, $r = 2$, $n = 3658$, $D = 4 \cdot 65 = 260$ MB/s, $N = 110$ MB/s, $d_m = i/n = 273$ GB. The bottom cell of Table 3 gives a best-case runtime of 6,308 seconds. Yahoo! achieved 58,500 seconds—about 9.3× slower. An optimal system using the same hardware would achieve better throughput with 400 nodes (instead of 3,658).

MapReduce – Grep: The original MapReduce paper [11] described a distributed grep computation that was executed on MapReduce. The setup is described as follows: $i = 1$ TB, $n = 1800$, $D = 2 \cdot 40 = 80$ MB/s, $N = 110$ MB/s, $d_m = 9.2$ MB, $e_M = 9.2/1000000 \approx 0$, $e_R = 1$. The paper does not specify the throughput of the disks, so we used 40 MB/s, conservatively estimated based on disks of the timeframe (2004). Equation 1 gives a best-case runtime of 6.94 seconds. Google achieved 150 seconds including startup overhead, or 90 seconds without that overhead—still about 13× slower. An optimal system using the same hardware would achieve better throughput with 139 nodes (instead of 1,800). The 60-second startup time experienced by MapReduce on a cluster of 1,800 nodes would also have been much shorter on a cluster of 139 nodes.

It is worth noting that not all submitted sort benchmark results are as inefficient as those based on

Hadoop Setting	Default	Tuned	Effect
Replication level	3	1	The replication level was set to 1 to avoid extra disk writes.
HDFS block size	64 MB	128 MB	Larger block sizes better amortize map task start-up costs.
Speculative exec.	$true$	$false$	Failures are uncommon on small clusters, avoid extra work.
Max maps/node	2	4	Our nodes can handle more map tasks in parallel.
Max reduces/node	1	4	Our nodes can handle more reduce tasks in parallel.
Map tasks	2	$4n$	For a cluster of n nodes, maximize the map tasks per node.
Reduce tasks	1	$4n$	For a cluster of n nodes, maximize the reduce tasks per node.
Java VM heap size	200 MB	1 GB	Increase the Java VM heap size for each child task.
Daemon heap size	1 GB	2 GB	Increase the heap size for Hadoop daemons.
Sort buffer memory	100 MB	600 MB	Use more buffer memory when sorting files.
Sort streams factor	10	30	Merge more streams at once when sorting files.

Table 4: Hadoop configuration settings used in our experiments.

general DISC frameworks. For example, reported DEMsort results [23] for a 100 TB sort are only 39% slower than our model's ideal. Reported TritonSort results [25] also appear to be more efficient.

4.1 Experiences with Hadoop

We experimented with Hadoop on a 25 node subset of our C2 cluster (described in Section 5) to confirm and better understand the inefficiency exposed by the above analysis.

Tuning Hadoop's settings: Default Hadoop settings fail to use most nodes in a cluster, using only two (total) map tasks and one reduce task. Even increasing those values to use four map and reduce tasks per node, a better number for our cluster, with no replication, still results in lower-than-expected performance. We improved the Hadoop sort performance by an additional $2\times$ by adjusting a number of configuration settings as suggested by Hadoop cluster setup documentation and other sources [2, 28, 4]. Table 4 describes our changes, which include reducing the replication level, increasing block sizes, increasing the number of map and reduce tasks per node, and increasing heap and buffer sizes.

Interestingly, we found that speculative execution did not improve performance for our cluster. Occasional map task failures and lagging nodes can and do occur, especially when running over more nodes. However, they are less common for our smaller cluster size (one NameNode and up to 25 slave nodes), and surprisingly they had little effect on the overall performance when they did occur. When using speculative execution, it is generally advised to set the number of total reduce tasks to 95–99% of the cluster's reduce capacity to allow for a node to fail and still finish execution in a single wave. Since failures are less of an issue for our experiments, we optimized for the failure-free case and chose enough Map and Reduce tasks for each job to fill every machine at 100% capacity.

Sort measurements and comparison to the model: Figure 3 shows sort results for different numbers of nodes using our tuned Hadoop configuration. Each measurement sorts 4 GB of data per node (up to 100 GB total over 25 nodes). Random 100 byte input records were generated with the *TeraGen* program, spread across active nodes via HDFS, and sorted with the standard *TeraSort* Hadoop program. Before every sort, the buffer cache was flushed (with `sync`) to prevent previously cached writes from interfering with the measurement. Additionally, the buffer cache was dropped from the kernel to force disk read operations for the input data. The sorted output is written to the file system, but not synced to disk before completion is reported; thus, the reported results are a conservative reflection of actual Hadoop sort execution times.

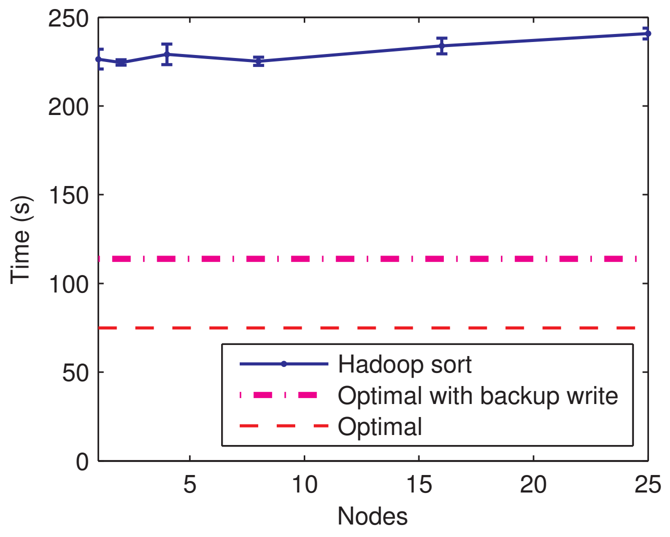


Figure 3: On a tuned Hadoop cluster, sort runtimes are ∼3× longer than ideal and 2× longer when allowing for Hadoop's unnecessary (in this context) backup write for map output. Data size scales with cluster size.

The results confirm that Hadoop scales well, since the average runtime only increases by 6% (14 seconds) from 1 node up to 25 nodes (as the workload increases in proportion). We also include the ideal sort times in Figure 3, calculated from our performance model parameterized to fit the C2 cluster (see Section 5). Comparison to the model reveals a large constant inefficiency for the tuned Hadoop setup—each sort requires 3× the ideal runtime to complete, even without syncing the output data to disk.

The 6% higher total runtime at 25 nodes is due to skew in the completion times of the nodes, but the vast majority of the inefficiency appears unrelated to scale. Section 5 identifies and analyzes inefficiencies (∼20%) induced by underlying OS, disk, and network services, but that also doesn't account for most of the lost efficiency. Up to ∼25% of the inefficiency, relative to ideal, may come from Hadoop's use of a backup write for the map output, despite the relatively short runtimes and small scale of our experiments. As shown by the dotted line in Figure 3, using the model equation that includes a backup write would accomodate for an ideal runtime that is 39 seconds longer, assuming the backup write goes to disk. Since the backup write is sent to the file system but not necessarily synced to disk, though, it is unclear what fraction of the 25% is actually from backup writes—with 4 GB of map output per node and 16 GB of memory per node, most of the backup write data may not actually be written to disk before the corresponding files are deleted.

Another possible source of inefficiency could be unbalanced distribution of the input data or the reduce data. However, we found that the input data is spread almost evenly across the cluster. Also, the difference between the ideal split of data and what is actually sent to each reduce node is less than 3%. Therefore, the random input generation along with TeraSort's sampling and splitting algorithms is partitioning work evenly, and the workload distribution is not to blame for the loss of efficiency.

Another potential source of inefficiency could be poor scheduling and task assignment by Hadoop. However, Hadoop actually did a good job at scheduling map tasks to run on the nodes that store the data, allowing local disk access (rather than network transfers) for over 95% of the input data. The fact that this value was below 100% is due to skew of completion times where some nodes finish processing their local tasks a little faster than others, and take over some of the load from the slower nodes.

We do not yet have a full explanation for the bulk of Hadoop's inefficiency. Although we have not been able to verify in the complex Hadoop code, some of the inefficiency appears to be caused by insufficiently pipelined parallelism between operators, causing serialization of activities (e.g., input read, CPU processing, and network write) that should ideally proceed in parallel. Part of the inefficiency is also commonly attributed to the CPU and data marshalling overhead induced by Hadoop's Java-based implementation. More diagnosis of Hadoop's inefficiency is left for continuing research.

5 Exploring the efficiency of data-intensive computing

The model indicates that there is substantial inefficiency in popular data-intensive computing systems. The remainder of the paper reports and analyzes results of experiments confirming the sanity of the model and exploring and quantifying the fundamental factors that affect scale and efficiency in data-intensive workloads. This section describes our experimental setup, I/O measurement tools, and data-intensive workloads. Section 6 uses a fast stripped-down framework to see how close a real system can approach the model's ideal runtime. Section 7 discusses these results and ties together our observations of the sources of inefficiency with opportunities for future work in this area.

Experimental hardware: Our experiments were performed on two different computing clusters with similar hardware, the primary difference being a faster 10 Gbps Ethernet network over two Arista 7148S switches on the first cluster (C1) and a 1 Gbps Ethernet network over four Force10 switches on the second cluster (C2). Another small difference between the clusters is that C1 uses a Linux 2.6.32 kernel while C2 uses a Linux 2.6.24 kernel that is running under the Xen hypervisor, however all experiments were run directly on the host's domain 0 and not through a virtual machine. Both kernels use TCP CUBIC with 1,500 byte packets. Each node is configured with two quad-core Intel Xeon E5430 processors, 16 GB of RAM, and at least two 1 TB Seagate Barracuda ES.2 SATA drives—the first disk reserved for the operating system, the second configured with the XFS filesystem and used for reading and writing benchmark and application data. After pruning away nodes with slower disks, C1 consists of 45 nodes and C2 of 41 nodes.

5.1 Microbenchmarks

Understanding the performance characteristics and operating quirks of the individual components, both disk and network, is essential for evaluating the performance of a cluster. To that end, we used simple single-purpose tools to measure the performance of each component. We measured the maximum achievable throughput and expect "optimal" performance to maintain full streaming utilization during periods when data is queued to be read or written.

Disk: For the disk, speeds are measured with the `dd` utility and the sync option to force all data to be written to disk. We use sufficiently large 4 GB file sizes and 128 MB block sizes, measuring bandwidth to and from the raw device and also the file system. For sufficiently large or sequential disk transfers, seek times have a negligible effect on performance; raw disk bandwidth approaches the maximum transfer rate to/from the disk media, which is dictated by the disk's rotation speed and data-per-track values [26]. For modern disks, "sufficiently large" is on the order of 8 MB [32].

Looking at a sample disk from the cluster, we observe 104 MB/s raw read speeds, which is in line with the specifications for our disks. While all of our experiment use the XFS filesystem, we considered both `ext3` and XFS initially during our early measurements, and both `ext3` and XFS achieve within 1% of the raw bandwidth for large sequential reads. Writes through the filesystem, however, are more interesting. "Copying" from `/dev/zero` to the filesystem gave average write bandwidths of 84 MB/s and 97 MB/s for `ext3` and XFS, respectively, while the raw disk writes were only around 3 MB/s slower than the raw disk reads. This 3 MB/s difference is explainable by sequential prefetching for reads without any inter-I/O bandwidth loss, but the larger filesystem-level differences are not. Rather, those are caused by file system decisions regarding coalescing and ordering of write-backs, including the need to update metadata. XFS and ext3 both maintain a write-ahead log for data consistency, which induce seek overhead on new data writes. ext3's higher write penalty is likely caused by its block allocator, which allocates one 4 KB block at a time, in contrast to XFS's variable-length extent-based allocator. [3] The measured difference between read and write bandwidth prompted us to use two values for disk speeds, D_r and D_w, in the model.

Network: For the network, although a full-duplex 1 Gbps Ethernet link could theoretically transfer 125 MB/s in each direction, maximum achievable data transfer bandwidths are lower due to unavoidable protocol overheads. We initially used the iperf tool to measure the bandwidth between machines, transferring 4 GB files and using the maximum kernel-allowed 256 KB TCP window size. We measured sustained bi-directional bandwidth between two nodes of 112 MB/s, which is in line with the expected best-case data bandwidth, but were unsure whether those speeds would hold over larger clusters with more competing flows performing an all-to-all shuffle. Therefore, an approximate initial estimate used in our optimal model is $N = 110$ MB/s.

We also built an all-to-all network transfer micro-benchmark to mimic just the network shuffle phase of a map-reduce dataflow, provide global throughput statistics, and trace the progress of every flow over time. The network shuffle microbenchmark was written in C++ and uses POSIX threads to parallelize the many send and receive tasks across all bi-directional flows. So that only the network is measured, we send dummy data from the same buffers on the sender and silently discard the data at the receiver.

Parallel sort benchmark: As mentioned in Section 3, a parallel sort benchmark serves as our primary workload since it effectively stresses all I/O components and is a common map-reduce dataflow computation. To generate input data for experiments, we used *Gensort*, which is the sort benchmark [20] input generator on which *TeraGen* is based. Taking advantage of its random distribution, the Gensort input set is partitioned equally across the nodes by deterministically mapping equally-sized ranges of the first few bytes of a key to a particular node. In the next section, we study the performance of this sort benchmark over both of our clusters, with the objective of determining how close to ideal runtimes are possible, and what are the causes of any inefficiency.

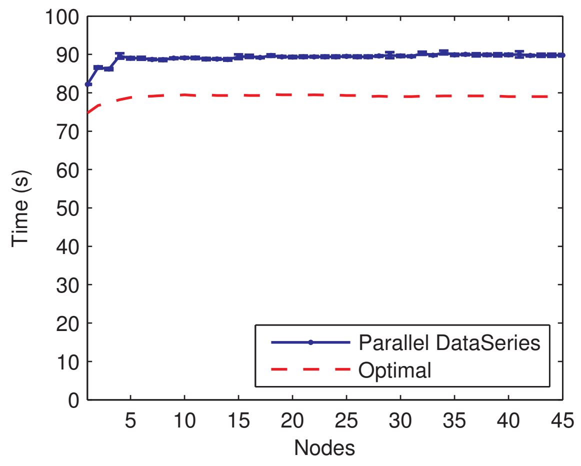
6 Evaluating the model with Parallel DataSeries

The results of our local Hadoop experiments and the published benchmarks at massive scale clearly diverge from the predicted ideal. The large extent to which they diverge, however, brings the accuracy of our simple model into question. To validate the model, and explore where efficiency is lost in practice, we present Parallel DataSeries (PDS), a data analysis tool that attempts to closely approach ideal hardware-efficiency.

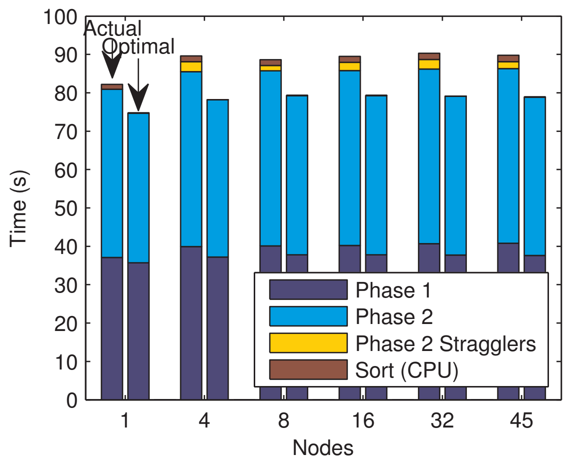
PDS Design: Parallel DataSeries builds on DataSeries, an efficient and flexible data format and open source runtime library optimized for analyzing structured data [7]. DataSeries files are stored as a sequence of *extents*, where each extent is a series of records. The records themselves are typed, following a schema defined for each extent. Data is analyzed at the record level, but I/O is performed at the much larger extent level. DataSeries supports passing records in a pipeline fashion through a series of modules. PDS extends DataSeries with modules that support parallelism over multiple cores (intra-node parallelism) and multiple nodes (inter-node parallelism), to enable parallel flows across modules.

Sort evaluation: We built a parallel sort module in PDS that implements a map-reduce dataflow. To setup the cluster before each experiment, GenSort input data is first converted to DataSeries format without compression. Since PDS doesn't utilize a distributed filesystem, the input data is manually distributed as 40 million 100-byte records ($\sim$4 GB) per node. All experiments use at least 10 repetitions of each cluster size, starting from a cold cache, partitioning and shuffling data across the network in Phase 1, performing a local sort at the start of Phase 2, and syncing all data to local disk (without replication) before terminating.

[3]To address some of these shortcomings, the ext4 file system improves the design and performance of ext3 by adding, among other things, multi-block allocations [19].



(a) PDS sort benchmark on a fast network.



(b) Time breakdown.

Figure 4: Using Parallel DataSeries to sort up to 180 GB on a fast 10 Gbps Ethernet, it is possible to achieve within 10 to 14% of the ideal sort time predicted by our performance model. PDS scales well up to 45 nodes in **(a)**, sorting 4 GB per node up to 45 nodes. A breakdown of time in **(b)** shows that the increases are mostly due to wasted time from nodes in Phase 2 that take longer to complete their assigned work, in addition to a similar Phase 1 increase from disk read skew effects (not explicitly broken out).

PDS sort on fast network: We first measured PDS with the standard sort benchmark on up to 45 nodes on the C1 cluster which is disk-bound because of its 10 Gbps network. Figure 4(a) graphs the total sort time achieved in comparison to optimal, and Figure 4(b) graphs this same information but further breaks down the time spent into the different phases of a map-reduce dataflow. Additionally, the time breakdown includes a *Stragglers* category that represents the average wait time of a node from the time it completes all its work until the the last node involved in the parallel sort also finishes. The straggler time is a measure of wasted system resources in the system due to skew in completion times, which challenges the model's assumption of unvarying homogeneity across the nodes.

PDS performed very well, within 10-14% of optimal. Most of the constant overhead that exists for a single node is accounted for. About 4% of efficiency is lost from converting the input data into DataSeries format, which, because of a known alignment overhead with the particular sizes of the input fields, causes a small data expansion. The in-memory sort time takes about 2 seconds, which accounts for another 2–3% of the overhead. Much of this CPU time could be overlapped with I/O (PDS doesn't currently), and it is sufficiently small to justify excluding CPU time from the model. These two factors explain the majority of the 10% overhead of the single node case, leaving less than 4% runtime overhead in the framework, concentrated during Phase 2, where we believe some of the inefficiency comes from a small delay in forcing the sync to flush the system buffers and write to disk.

As the parallel sort is performed over more nodes on the fast network, besides the small additional coordination overhead from code structures that enable partitioning and parallelism, the additional 4% inefficiency is explained by disk and straggler effects. The Phase 2 straggler effect accounts for about half of the additional inefficiency, which is separated out into the *stragglers* category. The other half is lost during Phase 1, where there is a similar skew effect when a node waits to receive all its data before performing a sort at the start of Phase 2. The quick increase in both the sort times and optimal prediction when moving from one to two nodes is due to randomly choosing a particularly fast node as the first member of our cluster. We were also relieved to see that parallel sort times stabilized at around 4 nodes, so that the straggler effects

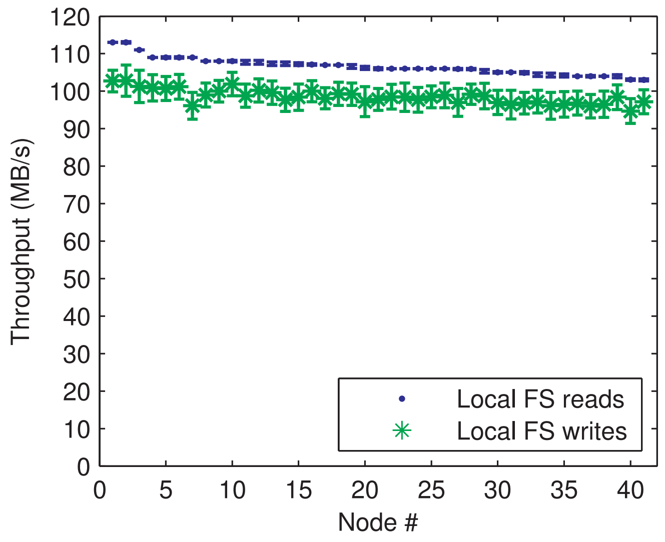


Figure 5: Read and write bandwidth measurements through the XFS filesystems on our C2 cluster. Variance is significant across nodes, and writes are slower than reads.

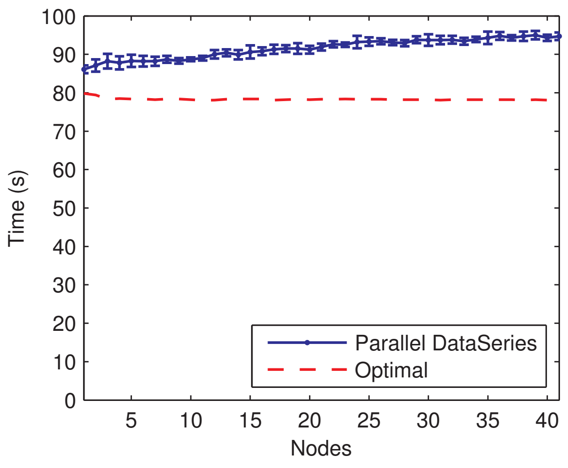
were limited and not a continuously growing effect.

FS-based straggler effects: Our approach to investigating stragglers is to understand why they fundamentally exist and how much of their impact is necessary. The straggler problem is well-known [6, 34] and was not unexpected. However, it is surprising that even when using a perfectly balanced workload, without any data skew, there was still an effect. Especially since performance laggards in the second phase can only be explained by disk performance, we revisited our disk microbenchmarks in detail.

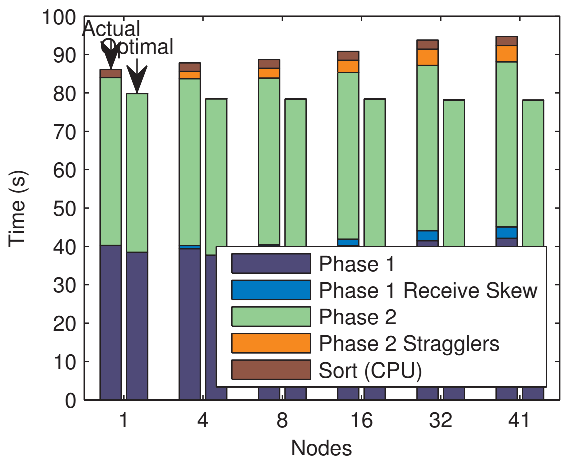
Although XFS shows a relatively close correlation between read and write speeds, looking more closely at the data we see that an individual XFS filesystem shows significant variance. The local XFS filesystem read and write speeds for C2 are shown in Figure 5. Although read speeds on a particular node are quite consistent, write speeds can vary between 88–100 MB/s over 30 trials even on one disk. Even more variation in FS speeds occurs across different nodes and disks, causing straggler effects. From 30 repetitions of our `dd` utility experiments, the average read bandwidth across the cluster is 106.6 MB/s while the average write bandwidth is 98.5 MB/s. However the speeds can vary over the entire cluster from 102–113 MB/s for reads and 87–107 MB/s for writes, both with a standard deviation of 2. Each node exhibits a similar difference in read and write speeds, and slower reads are correlated with slower writes.

Some of the variation in FS write speeds are due to the placement of data into different areas on disk, where traditional zoning techniques allow for more data to be packed into the longer outer tracks of a disk while less data would fit into the shorter inner tracks [30, 27]. Multiple zones each have different sectors-per-track values and, thus, media transfer rates. The raw disk measurements come from the first disk zone, but a filesystem decides how to break the disk down into subvolumes that span different disk zones. XFS, for instance, uses many allocation groups to manage free blocks and journaling over the disk, and more allocation groups are used for larger disk sizes. We can partially verify this with the results of our disk microbenchmarks on C2. The disk bandwidth numbers from C1 are about the same, with read speeds of 106.5 and slightly slower writes at 96.8 MB/s. However, C1 was set up to use only a small 10 GB partition instead of the entire disk. Hence, writes on C1 have an average standard deviation of only 0.64 MB/s compared to 3.3 MB/s on C2.

With modern disks, traditional zoning schemes have been extended to even more aggressively increase



(a) PDS sort benchmark with a network bottleneck.



(b) Time breakdown.

Figure 6: Using Parallel DataSeries to sort up to 164 GB over 1 Gbps Ethernet, network effects cause completion times to slowly increase from 8 to 21% higher than a predicted ideal that inaccurately assumes constant 110 MB/s data transfer bandwidth into each node. The increase in sort time up to 41 nodes in **(a)** is only explained partly by the Phase 2 straggler effects as broken down in **(b)**. The longer network shuffle time and skew from Phase 1 is the largest component of the slowly growing parallel sort times

areal density with a technique that we refer to as *adaptive zoning* [18]. This relatively new technique by disk manufacturers avoids preconfiguring each zone boundary, instead maximizing the platter densities post-production according to the capabilities of each disk head, which have a good deal of process variation. Adaptive zoning is responsible for a large amount of the variability in bandwidth of our otherwise identical make and model disks, and also produces larger differences in performance across the multiple disk platters within a single disk.

Even after pruning away slower nodes in our clusters, the variability of FS speeds across the nodes was enough to cause disk skew and straggler effects for our PDS sort experiments. If we hadn't pruned the nodes to all be within a particular range of speeds, even just one particularly slow node would delay the entire job. Dynamic task assignment, such as that used by Hadoop or MapReduce, could reduce this load imbalance across the cluster if the overhead associated with this feature is small enough and the tasks are appropriately sized. Although this task has been made more difficult with the variability of modern adaptive zoning, It may also be worthwhile for a DISC-centric distributed filesystem to measure and attempt to match disk zones across the cluster, to reduce the variance for an individual job.

PDS sort on slower network: To see how both PDS and the model react to a slower 1 Gbps network, we also ran PDS parallel sort experiments on the C2 cluster; here, the model assumes a constant network bandwidth of 110 MB/s, which would make the disk the slowest component, but the actual performance of our sort experiments reveal that the network speeds become slower. As presented in Figure 6, parallel sort times continue to slowly increase from 8 to 21% longer than ideal (the model) for up to 41 nodes. Since the disk effects were bounded in our first experiments, and the Phase 2 straggler time accounts for only 5% of the efficiency loss, the growing divergence from the model is explained by network effects. The addition of a *Phase 1 Receive Skew* category in the time breakdown measures the component of Phase 1 time during the shuffle phase that is the average difference of each node's last incoming flow completion time and the average incoming flow completion time. Flows complete with a receive skew that is 4% overhead from ideal. The presence of receive skew doesn't necessarily mean that the transfer is inefficient, only that the

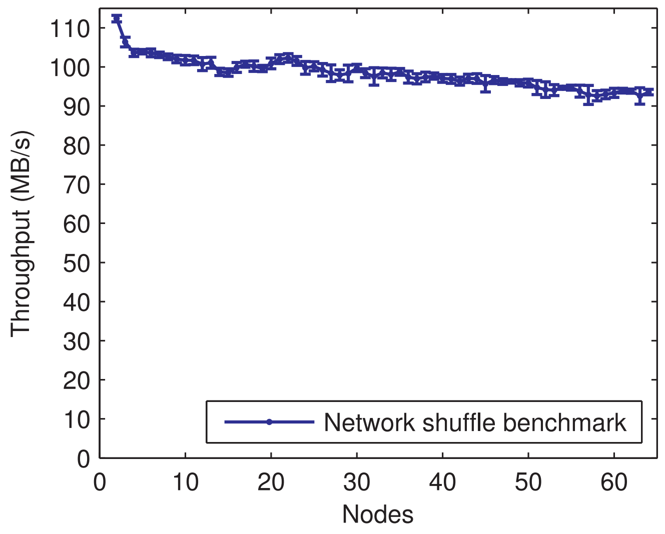


Figure 7: A separate network shuffle microbenchmark, which measures the full time from initiating all connections until the last bit of data has been received at all the nodes, confirms that, even with no other network traffic, network skew and slowdown effects make it infeasible for more than two all-to-all network transfers to achieve the full link capacity that is possible with a single bidirectional transfer.

flows complete in an uneven manner. However, for a protocol like TCP, it suggests an inherent unfairness to time sharing of the link.

Network skew and slowdown effects: To discover the reasons behind the network effects, we performed more measurements of the network and discovered large amounts of network skew and general slowdown effects from many competing transfers during the network shuffle. For example, on even a 3-way all-to-all network transfer, with 6 bi-directional flows, `iperf` could only sustain 104 MB/s on average into each node (rather than 110 MB/s). Figure 7 graphs the resulting throughput of a full network shuffle microbenchmark, including the time to set up all the threads, initiate connections, and completely finish transferring data to all the nodes. With the small overhead of our microbenchmark tracing infrastructure and setting up and joining over threads, we achieve 112 MB/s for a 2-way flow, which also matches a basic iperf test, and 106 MB/s for a 3-way flow, which is 2 MB/s faster than a basic coordinated all-to-all exchange using a few instances of `iperf`. Network shuffle speeds drop quickly to under 104 MB/s from 2 to 4 nodes, and then continue to slowly decrease down to 98 MB/s at 32 nodes and under 94 MB/s at 64 nodes.

Figure 8 presents the aggregate network link utilization into each node for the last 10 seconds of a representative 3-way network shuffle, revealing large intermittent drops in link utilization that are correlated across the different nodes. This 3-way shuffle should have completed 2 seconds earlier if it was achieving 110 MB/s average bandwidth into each node. Instead, one slightly faster node finishes receiving all its data half a second earlier than the slowest node, achieving an aggregate throughput of 108 MB/s, and then must wait for all other nodes to finish receiving their data before the entire shuffle can complete.

These experiments were run using the newer CUBIC [14] congestion control algorithm, which is the default on Linux 2.6.26 and is tuned to support high-bandwidth links. TCP NewReno performed just a little slower, and is known to have slow convergence on using full link bandwidths on high-speed networks [16]. Some of TCP's and, specifically, CUBIC's unfairness and stability issues are known and are prompting continuing research toward better algorithms [16]. However, we have found no previous research exposing problems with TCP's all-to-all network performance in high-bandwidth low-latency environments. The

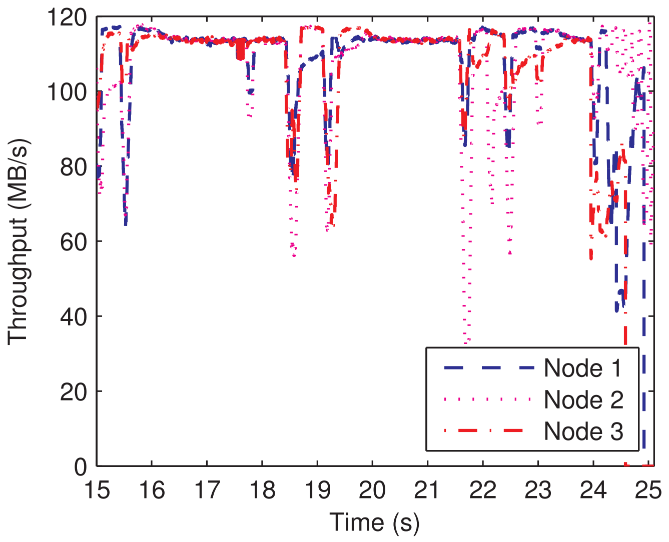


Figure 8: A trace of the last 10 seconds of instantaneous aggregate throughput into each node of a 3-way all-to-all network shuffle, smoothed with a 100 ms sliding window. Unexpected network fluctuations and flow unfairness across nodes contribute to partial link utilization and slower network throughput.

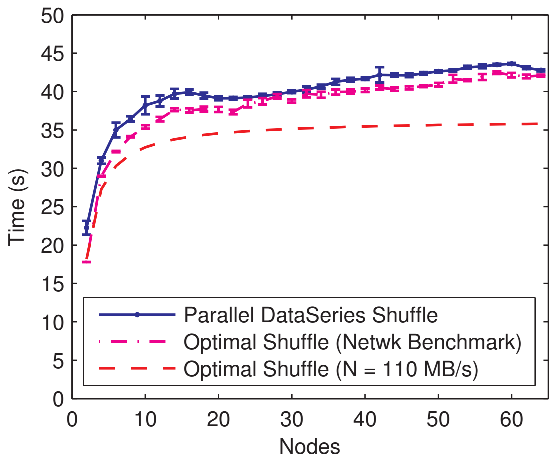
closest other known issue with TCP that we know of is the Incast problem [22, 31], which manifests as a 2–3× order of magnitude collapse in network speeds when performing many small bursty synchronized reads in an all-to-one pattern. An all-to-all shuffle creates more competing flows, but also avoids the many synchronization steps necessary to induce Incast, with only a single barrier upon completion of all incoming flows. Also, while Incast was solved with high-frequency timers and microsecond-granularity TCP retransmission timeouts, we have verified that TCP timeouts are not always occurring in tandem with the all-to-all network shuffle slowdown.

To zero in even further on the network effects, to deduce their impact on increasing the PDS sort numbers, we performed a separate network shuffle within PDS. Data was read from the cache, the sort was eliminated (but still included a synchronization barrier at the start of Phase 2), and nothing was written to disk at the destination. Our optimal model for this scenario just removes the disk components and focuses on the network, where the $\frac{n-1}{n}N$ term in the model predicts increasingly long sort times that converge in scale as more nodes participate. Figure 9 demonstrates that our actual shuffle results with PDS match up very well to that pattern, but that network shuffle speeds of 110 MB/s are infeasible. In fact, the average PDS network shuffle speeds dip just under 94 MB/s at 64 nodes, when the overhead of running a shuffle through PDS becomes amortized with longer shuffle times. The PDS shuffle times also match up closely to the shuffle microbenchmark at 64 nodes, as both experience a large amount of receive skew that is solely from the network, along with a network slowdown that is responsible for the loss of efficiency.

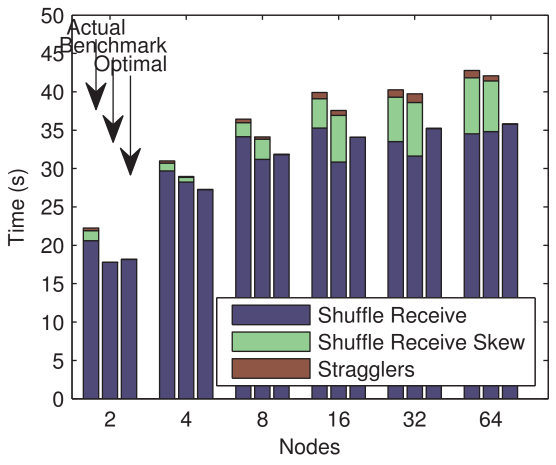
7 Discussion

The experiments with PDS demonstrate that our model is not wildly optimistic—it is possible to get close to the optimal runtime. Thus, the inefficiencies indicated for our Hadoop cluster and the published benchmark results are real. We do not have complete explanations for the 3–13× longer runtimes for current data-intensive computing frameworks, but we have identified a number of contributors.

One class of inefficiencies comes from duplication of work or unnecessary use of a bottleneck resource.



(a) PDS network shuffle on 1 Gbps Ethernet.



(b) Time breakdown.

Figure 9: By focusing just on the PDS network shuffle times on the slower 1 Gbps Ethernet, the network time increase is more than the model predicts. Two ideal value lines are given in **(a)**, in addition to the measured PDS network shuffle values, assuming different average network bandwidths during the shuffle: the original 110 MB/s (applied to every point on the x-axis) and the decreasing-with-scale measured value from the network shuffle microbenchmark for each number of nodes. The time breakdown in **(b)** confirms that most of the increased runtime is due to shuffle receive skew, which measures the skew of incoming flow completions across nodes, as well as a general network slowdown effect.

For example, Hadoop and Google's MapReduce always write Phase 1 map output to the file system, whether or not a backup write is warranted, and then read it from the file system when sending it to the reducer node. This file system activity, which may translate into disk I/O, is unnecessary for completing the job and inappropriate for shorter jobs.

A significant issue faced by map-reduce systems is that a job only completes when the last node finishes its work. In our experiments, straggler effects account for less than 10% of the inefficiency at 10s of nodes. Thus, it is not the source of most of the inefficiency at that scale. For larger scale systems, such as the 1,000+ node systems used for the benchmark results, this straggler effect is expected to be more significant—it may explain much of the difference between our measured 3× higher-than-ideal runtimes and the published 6× higher-than-ideal runtime of the Hadoop record-setting TeraSort benchmark. The straggler effect is also why Google's MapReduce and Hadoop dynamically distribute map and reduce tasks among nodes. Support for speculative execution also can help mitigate this effect, as demonstrated in recent work [6, 34], in addition to enhancing fault tolerance.

It is tempting to blame lack of sufficient bisection bandwidth in the network topology for much of the inefficiency at scale. This would exhibit itself as over-estimation of each node's true network bandwidth, assuming uniform communication patterns, since the model does not account for such a bottleneck. However, this is not an issue for the measured Hadoop results on our small-scale cluster because all nodes are attached across two switches with sufficient backplane bandwidth. The network topology was not disclosed for most of the published benchmarks, but for many we don't believe bisection bandwidth was an issue. For example, MapReduce grep involves minimal data exchange because $e_M \approx 0$. Also, for Hadoop PetaSort, Yahoo! used 91 racks, each with 40 nodes, one switch, and an 8 Gbps connection to a core switch (via 8 trunked 1 Gbps Ethernet links). For this experiment, the average bandwidth per node was 4.7 MB/s. Thus, the average bandwidth per uplink was only 1.48 Gb/s in each direction, well below 8 Gbps. Other benchmarks may have involved a bisection bandwidth limitation, but such an imbalance would have meant that far more machines were used per rack (and overall) than were appropriate for the job, resulting in significant wasted resources.

Naturally, deep instrumentation and analysis of Hadoop will provide more insight into its inefficiency. Also, PDS in particular provides a promising starting point for understanding the sources of inefficiency. For example, replacing the current manual data distribution with a distributed file system is necessary for any useful system. Adding that feature to PDS, which is known to be efficient, would allow one to quantify its incremental cost. The same approach can be taken with other features, such as dynamic task distribution and fault tolerance.

8 Conclusion

Data-parallel computation and scale-out performance are here to stay. However, we hope that the low efficiency of popular DISC frameworks, such as job runtimes for popular map-reduce systems that have been shown to be 3–13$\times$ longer than their hardware resources should allow, is not. Experiments with Parallel DataSeries, our simplified dataflow processing tool, demonstrate that the model's runtimes can be approached, validating the model and confirming the inefficiency of Hadoop and Google's MapReduce. Using the model as a guide, additional experiments identify and quantify file system and networking inefficiencies that will affect any DISC framework built atop common OSes. Furthermore, most of the 3–13$\times$ slowdown arises from inefficiency in the frameworks themselves. By gaining a better understanding of inefficiency, and understanding the limits of what is achievable, we hope that our work will lead to better insight into and continued improvement of commonly used DISC frameworks.

References

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

[2] *Hadoop Cluster Setup Documentation*, http://hadoop.apache.org/common/docs/r0.20.2/cluster_setup.html.

[3] *HDFS*, http://hadoop.apache.org/core/docs/current/hdfs_design.html.

[4] *Intel White Paper. Optimizing Hadoop Deployments*, October 2009.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat, *A Scalable, Commodity Data Center Network Architecture*, ACM SIGCOMM (Seattle, WA), August 2008.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, and B. Saha, *Reining in the Outliers in Map-Reduce Clusters using Mantri*, USENIX Symposium on Operating Systems Design and Implementation, 2010.

[7] Eric Anderson, Martin Arlitt, Charles B. Morrey, III, and Alistair Veitch, *Dataseries: an efficient, flexible data format for structured serial data*, SIGOPS Oper. Syst. Rev. **43** (2009), no. 1, 70–75.

[8] Eric Anderson and Joseph Tucek, *Efficiency Matters!*, HotStorage '09: Proceedings of the Workshop on Hot Topics in Storage and File Systems (2009).

[9] Randal E. Bryant, *Data-Intensive Supercomputing: The Case for DISC*, Tech. report, Carnegie Mellon University, 2007.

[10] Grzegorz Czajkowski, *Sorting 1PB with MapReduce*, October 2008, http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

[11] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: simplified data processing on large clusters*, Communications of the ACM **51** (2008), no. 1, 107–113.

[12] David DeWitt and Jim Gray, *Parallel database systems: the future of high performance database systems*, Commun. ACM **35** (1992), no. 6, 85–98.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The Google file system*, ACM Symposium on Operating Systems Principles, 2003.

[14] Sangtae Ha, Injong Rhee, and Lisong Xu, *CUBIC: A new TCP-friendly high-speed TCP variant*, SIGOPS Oper. Syst. Rev. **42** (2008), no. 5, 64–74.

[15] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly, *Dryad: distributed data-parallel programs from sequential building blocks*, ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007.

[16] Vishnu Konda and Jasleen Kaur, *RAPID: Shrinking the Congestion-control Timescale*, INFOCOM, April 2009.

[17] Michael A. Kozuch, Michael P. Ryan, Richard Gass, Steven W. Schlosser, James Cipar, Elie Krevat, Michael Stroucken, Julio Lopez, and Gregory R. Ganger, *Tashi: Location-aware Cluster Management*, Workshop on Automated Control for Datacenters and Clouds, June 2009.

[18] Elie Krevat, Joseph Tucek, and Gregory R. Ganger, *Disks Are Like Snowflakes: No Two Are Alike*, Tech. report, Carnegie Mellon University, February 2011.

[19] Aneesh Kumar K.V, Mingming Cao, Jose R. Santos, and Andreas Dilger, *Ext4 block and inode allocator improvements*, Proceedings of the Linux Symposium (2008), 263–274.

[20] Chris Nyberg and Mehul Shah, *Sort Benchmark*, http://sortbenchmark.org/.

[21] Owen O'Malley and Arun C. Murthy, *Winning a 60 Second Dash with a Yellow Elephant*, April 2009, http://sortbenchmark.org/Yahoo2009.pdf.

[22] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan, *Measurement and analysis of TCP throughput collapse in cluster-based storage systems*, USENIX Conference on File and Storage Technologies, 2008.

[23] Mirko Rahn, Peter Sanders, and Johannes Singler, *Scalable Distributed-Memory External Sorting*, October 2009, available at http://arxiv.org/pdf/0910.2582.

[24] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems*, IEEE International Symposium on High Performance Computer Architecture, 2007.

[25] Alexander Rasmussen, Harsha V. Madhyastha, Radhika Niranjan Mysore, Michael Conley, Alexander Pucher, George Porter, and Amin Vahdat, *TritonSort*, May 2010, available at http://sortbenchmark.org/tritonsort_2010_May_15.pdf.

[26] Chris Ruemmler and John Wilkes, *An introduction to disk drive modeling*, IEEE Computer **27** (1994), 17–28.

[27] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger, *Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics*, USENIX Symposium on File and Storage Technologies, 2002.

[28] Sanjay Sharma, *Advanced Hadoop Tuning and Optimisation*, December 2009, http://www.slideshare.net/ImpetusInfo/ppt-on-advanced-hadoop-tuning-n-optimisation.

[29] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin, *MapReduce and Parallel DBMSs: Friends or Foes?*, Communications of the ACM (2010).

[30] Rodney Van Meter, *Observing the effects of multi-zone disks*, Proceedings of the annual conference on USENIX Annual Technical Conference, 1997.

[31] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller, *Safe and effective fine-grained tcp retransmissions for datacenter communication*, ACM SIGCOMM Conference on Data Communication, 2009.

[32] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger, *Argon: Performance insulation for shared storage servers*, USENIX Conference on File and Storage Technologies, 2007.

[33] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta, *A Simulation Approach to Evaluating Design Decisions in MapReduce Setups*, IEEE/ACM MASCOTS, September 2009.

[34] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica, *Improving mapreduce performance in heterogeneous environments*, USENIX Symposium on Operating Systems Design and Implementation, 2008.