Applying performance models to understand data-intensive computing efficiency

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

Abstract

New programming frameworks for scale-out parallel analysis, such as MapReduce and Hadoop, have become a cornerstone of exploiting large datasets. But, 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 optimal performance of a parallel dataflow system. This 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. To validate the sanity of our model, we present small-scale experiments with Hadoop and a simplified dataflow processing tool called Parallel DataSeries. Parallel DataSeries achieves performance close to the analytic optimal, showing that the model is realistic and that large improvements in parallel analysis efficiency are possible.

1 Introduction

"Data-intensive scalable computing" (DISC) refers to a rapidly growing style of computing characterized by its reliance on huge and growing datasets [7]. Driven by the desire and capability to extract insight from such datasets, data-intensive computing 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 data-intensive computing, new programming frameworks (e.g., MapReduce [9], Hadoop [1] and Dryad [13]) 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, 20].

These frameworks can scale jobs to thousands of computers, which is great. But, they currently focus on scalability without concern for efficiency. Worse, anecdotal experiences indicate that they fall far short of fully utilizing hardware resources, effectively wasting large fractions 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 while not wasting resources (e.g., CPU or disk) applied to a given computation. Given the spread of data-intensive computing combined with its scale, we need to move toward such an 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 with a simple model of "ideal" map-reduce job runtimes and 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 ideal job runtime.

An ideal run is "hardware-efficient," meaning that the realized throughput matches the maximum throughput for the bottleneck hardware resource, given its usage (i.e., amount of data moved over it). Our model can expose how close (or far, currently) a given system is from this ideal. Such throughput won't occur, for example, if the framework does not provide sufficient parallelism to keep the bottleneck resource fully utilized, or it makes poor use of a particular resource (e.g. inflating network traffic). In addition, our model can be used to quantify resources wasted due to imbalance—in an unbalanced system, one resource (e.g., network, disk, or CPU) is under-provisioned relative to others and acts as a bottleneck. The other resources are wasted to the extent that they are over-provisioned and active.

To illustrate these issues, we applied the model to a number of benchmark results (e.g., for the TeraSort and PetaSort benchmarks) touted in the industry. As shown in Figure 2, these presumably well-tuned systems achieve runtimes that are 6–13$\times$ 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, 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 enable demonstration that near-ideal hardware-efficiency (within $\approx$20%) is possible. In addition to validating the model, PDS provides an interesting foundation for subsequent analyses of the incremental costs

associated with various features, such as distributed file system functionality, dynamic task distribution, fault tolerance, and task replication.

Data-parallel computation is here to stay, as is scale-out performance. However, we can hope that the low efficiency indicated by our model is not. By gaining a better understanding of where the bottlenecks in these computations, and understanding the limits of what is achievable, we hope that our work can lead to improvements in commonly used DISC frameworks.

2 Dataflow parallelism and map-reduce computing

Today's "data-intensive computing" derives much from early 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 [10] describe two forms of parallelism in such dataflow systems: partitioned parallelism and pipelined parallelism. Partitioned parallelism is achieved by partitioning the data and splitting an operator into multiple operators running on different processors. Pipelined parallelism is achieved by streaming the output of one operator into the input of another operator so that the two operators can work in series on different data at the same time.

Google's MapReduce[1] [9] 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 (computers). The computation is divided into two phases, labeled Phase 1 and Phase 2.

Phase 1: 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 cause a reduce

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

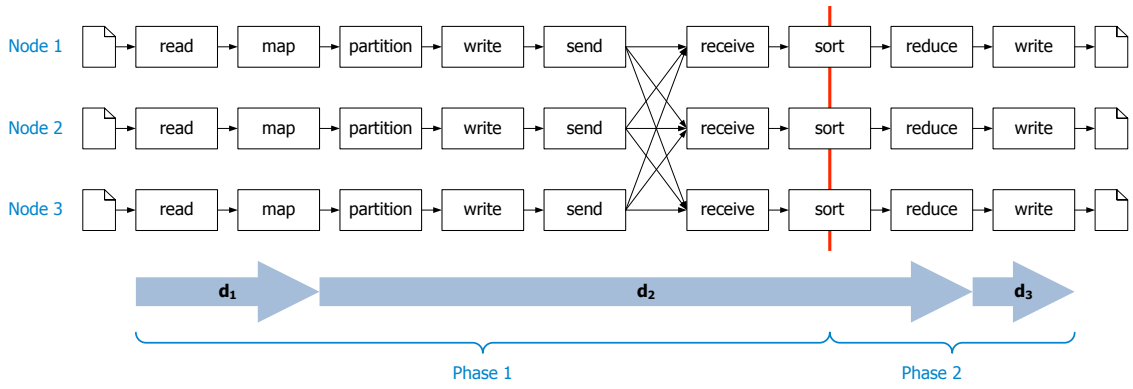


Figure 1: A map-reduce dataflow.

task to stop running.

Phase 2: 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 [11] and HDFS [3] distributed file systems used 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_1, and the amount of output data per node is d_3. The amount of data per node produced by the map operator and consumed by the reduce operator is d_2. In most applications, the amount of data flowing through the system either remains the same or decreases (i.e., $d_1 \geq d_2 \geq d_3$). In general, the map will implement some form of select, filtering out rows, and the reduce 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 [9]. Many map-reduce workloads resemble a "grep"-like computation, in which the map operator decreases the amount of data ($d_1 \gg d_2$ and $d_2 = d_3$). In others, such as in a sort, neither the map nor the reduce function decrease the amount of data ($d_1 = d_2 = d_3$).

2.1 Related work

Concerns about the performance of map-reduce style systems 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 [25]. In [5], two of the authors of our paper pointed out that many parallel systems (especially map-reduce systems, but also other parallel systems) have focused almost exclusively on the headline performance number and high-end scalability. This focus, as the authors quantify by back-of-the-envelope comparisons, has been at the detriment of other worthwhile metrics.

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 [27]. 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 to 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.

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 assumptions, parameters, and equations of the model, along with a description of common 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 [9] 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 also do not consider specific network topologies or technologies, and we assume that the network core is over-provisioned enough that the internal network topology does not impact the speeds of inter-node data transfers. From our experience, unlimited backplane bandwidth without any performance degradation is probably impractical, and this assumption of full bisection bandwidth is revisited in Section 8.

The model also assumes that input data is evenly distributed across all participating nodes in the cluster, that nodes are homogeneous, and that each node retrieves its input from local storage. Most map-reduce systems are designed to fit these assumptions. The model 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.

The model assumes that a single job has full access to the cluster at a time, with no competing jobs or other activities. Production map-reduce clusters may be shared by more than one simultaneous job, but understanding a single job's performance is a useful starting point.

Deriving the model from I/O operations: Table 1 identifies the I/O operations in each map-reduce phase, for two variants of the sort operator. When the data to be sorted 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 $\frac{n-1}{n} d_2$ term appears in the equation, where n is the number of nodes, because each node partitions and transfers that fraction of its mapped data over the network, keeping $\frac{1}{n}$ of the data for itself.

Table 2 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_1, d_2, and d_3) or in terms of the ratios of the map and reduce operators' output and input (e_M and e_R, respectively).

	$d_2 <$ memory (in-memory sort)	$d_2 \gg$ memory (external sort)
Phase 1	Disk read (input): d_1 Disk write (backup): d_2 Network: $\frac{n-1}{n} d_2$	Disk read (input): d_1 Disk write (backup): d_2 Network: $\frac{n-1}{n} d_2$ Disk write (sort): d_2
Phase 2	Network: $(r-1)d_3$ Disk write (output): rd_3	Disk read (sort): d_2 Network: $(r-1)d_3$ Disk write (output): rd_3

Table 1: I/O operations in a map-reduce job. The first disk write in Phase 1 is an optional backup in case of node failure.

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.
$d_1 \left(= \frac{i}{n}\right)$	The amount of input data per node, for a given computation.
$d_2 \left(= \frac{i \cdot e_M}{n}\right)$	The amount of data per node after the map operator, for a given computation.
$d_3 \left(= \frac{i \cdot e_M \cdot e_R}{n}\right)$	The amount of output data per node, for a given computation.
$e_M \left(= \frac{d_2}{d_1}\right)$	The ratio between the map operator's output and its input.
$e_R \left(= \frac{d_3}{d_2}\right)$	The ratio between the reduce operator's output and its input.

Table 2: Modeling parameters that include I/O speeds and workload properties.

Table 3 gives the model equations for the execution time of a map-reduce job in each of four scenarios, representing the cross-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 ($\frac{i}{n}$). 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 ($\frac{1}{D_r}$), 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 ($\frac{e_M}{D_w}$), and sending $\frac{n-1}{n}$ of the e_M map output bytes over the network ($\frac{\frac{n-1}{n} e_M}{N}$) to other reduce nodes. The corresponding reduce phase includes two disk transfers and one network transfer: reading sorted batches ($\frac{e_M}{D_r}$), writing $e_M e_R$ reduce output bytes produced locally ($\frac{e_M e_R}{D_w}$) and $(r-1)e_M e_R$ bytes replicated from other nodes ($\frac{(r-1)e_M e_R}{D_w}$), and sending $e_M e_R$ bytes produced locally to $(r-1)$ other nodes ($\frac{e_M e_R (r-1)}{N}$). Putting all of this together produces the equation shown.

	$d_2 < memory$ (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	$\frac{i}{n}\left(max\left\{\frac{1}{D_r} + \frac{e_M}{D_w}, \frac{\frac{n-1}{n}e_M}{N}\right\} + max\left\{\frac{re_Me_R}{D_w}, \frac{e_Me_R(r-1)}{N}\right\}\right)$
	$d_2 \gg memory$ (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.

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 in analyzing systems and validating our model, which Dean and Ghemawat [9] indicate are representative of most programs written by users of Google's MapReduce.

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 using the model 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 (i.e., d_1, d_2, and d_3) 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 for evaluating the performance of a given workload run on that system. For a hypothetical system, some estimates must be used, such as "$d_1 = d_2 = d_3$ for sort" or "$d_2 = d_3 = 0$ 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_2 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 ($\frac{d_2}{D_w}$), 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.

The appropriate values for I/O speed depends on what is being evaluated. For both production and hypothetical systems, specification values for the hardware can be used—for example, 1 Gbps for the network and the maximum streaming bandwidth specified for the given disk(s). This approach is appropriate for evaluating the efficiency of the entire software stack, from the operating system up. But, if the focus is on the programming framework, using raw hardware specifications can indicate greater inefficiency than is actually present. In particular, some efficiency is generally lost in the underlying operating system's conversion of raw disk and network resources into higher level abstractions, such as file systems and network sockets. To focus attention on programming framework inefficiencies, one should use measurements of the disk and network bandwidths available to applications using the abstractions. As shown in our experiments, such measured values are lower than specified values and often have non-trivial characteristics, such as dependence on file system age or network communication patterns.

4 Existing data-intensive computing systems are 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/benchmarks represented

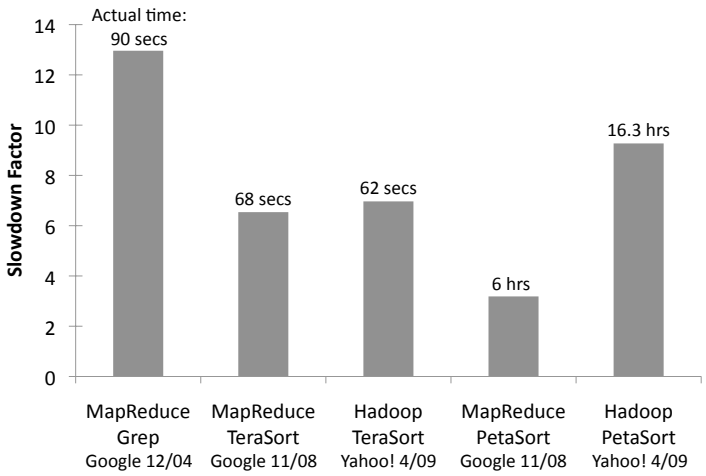


Figure 2: Published benchmarks of popular parallel dataflow systems. Each bar represents the reported throughput relative to the ideal indicated by our performance model, parameterized according to the cluster's hardware.

in Figure 2.

Hadoop – TeraSort: In April 2009, Hadoop set a new record [18] for sorting 1 TB of data in the Sort Benchmark [17] format. The setup had the following parameters: $i = 1$ TB, $r = 1$, $n = 1460$, $D = 4 \cdot 65 = 260$ MB/s, $N = 110$ MB/s, $d_2 = i/n = 685$ 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× slower. An optimal system using the same node hardware would achieve better throughput with 209 nodes (instead of 1460).

MapReduce – TeraSort: In November 2008, Google reported TeraSort results for 1000 nodes with 12 disks in each node [8]. The following parameters were used: $i = 1$ TB, $r = 1$, $n = 1000$, $D = 12 \cdot 65 = 780$ MB/s, $N = 110$ MB/s, $d_2 = i/n = 1000$ 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 node hardware would achieve better throughput with 153 nodes (instead of 1000).

MapReduce – PetaSort: Google's PetaSort experiment [8] is similar to TeraSort, with three differences: (1) an external sort is required with a larger amount of data per node ($d_2 = 250$ 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, $d_2 = i/n = 250$ GB. The bottom cell of Table 3 gives a best-

case runtime of 6818 seconds. Google achieved 21,720 seconds—approximately 3.2× slower. An optimal system using the same node hardware would achieve better throughput with 1256 nodes (instead of 4000). Also, according to our model, for the purpose of sort-like computations, Google's nodes are over-provisioned 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 [18] is similar to Google's (above), with one 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, $d_2 = i/n = 273$ GB. The bottom cell of Table 3 gives a best-case runtime of 6308 seconds. Yahoo! achieved 58,500 seconds—about 9.3× slower. An optimal system using the same node hardware would achieve better throughput with 400 nodes (instead of 3658).

MapReduce – Grep: The original MapReduce paper [9] 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, $d_2 = 9.2$ 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 node hardware would achieve better throughput with 139 nodes (instead of 1800). The 60-second startup time experienced by MapReduce on a cluster of 1800 nodes would also have been much shorter on a cluster of 139 nodes.

5 Exploring 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 exploring such inefficiency. This section describes our cluster and quantifies efficiency lost to OS functionality. Section 6 confirms the Hadoop inefficiency indicated in the benchmark analyses, and Section 7 uses a stripped-down framework to validate that the model's optimal runtimes can be approached. Section 8 discusses these results, toward additional attributing of inefficiency.

Experimental cluster: Our experiments used 1–25 nodes of a 25-node cluster. Each node is configured with 2 quad-core Intel Xeon E5430 processors, 4 1TB Seagate Barracuda ES.2 SATA drives, 16 GB of RAM, and 1 Gigabit

Ethernet link to a Force10 switch. The I/O speeds indicated by the hardware specifications are $N = 1$ Gbps and $D_r = D_w = 108$ MB/s (for the outer-most disk zone). All machines run the Linux 2.6.24 Xen kernel, but none of our experiments were run in virtual machines—they were all run directly on domain zero. The kernel's default TCP implementation (TCP NewReno using up to 1500 byte packets) was used. Except where otherwise noted, the XFS file system was used to manage a single one of the disks in our experiments.

Disk bandwidth for applications: 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 [21]. For modern disks, "sufficiently large" is on the order of 8 MB [26]. Most applications do not access the raw disk, instead accessing the disk indirectly via a file system. Using the raw disk, we observe the 108 MB/s indicated by the specifications for our disks. Nearly the same bandwidth (within 1%) can be achieved for large sequential file reads on `ext3` and XFS file systems. For writes, our measurements indicate more interesting behavior. Using the `dd` utility with the sync option, a 64MB block size, and input from the `/dev/zero` pseudo-device, we observe steady-state write bandwidths of 84 MB/s and 102 MB/s, respectively. When writing an amount of data less than or close to the file system cache size, the reported bandwidth is up to another 10% lower, since the file system does not start writing the data to disk immediately; that is, disk writing is not occurring during the early portion of the utility runtime.

This difference between read and write bandwidths causes us to use two values (D_r and D_w) in the model; our original model used one value for both. The difference is not due to the underlying disks, which have the same media transfer rate for both reads and writes. Rather, it is 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 also induces some overhead on new data writes. `ext3`'s relatively 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. 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 [16].

The 108 MB/s value, and the `dd` measurements discussed above, are for the first disk zone. Modern disks have multiple zones, each with a different data-per-track value and, thus, media transfer rate [22]. When measuring an XFS filesystem on a partition covering the entire disk, read speeds remained consistent at 108 MB/s, but write speeds

fluctuated across a range of 92-102 MB/s with an average of 97 MB/s over 10 runs. In reporting "optimal" values for experiments with our cluster, we use 108 MB/s and 97 MB/s for the disk read and write speeds, respectively.

Network bandwidth for applications: Although a full-duplex 1Gbps Gigabit Ethernet link could theoretically transfer 125MB/s in each direction, maximum achievable data transfer bandwidths are lower due to unavoidable protocol overheads. Using the `iperf` tool with the maximum kernel-allowed 256KB TCP window size, we measured sustained bandwidths between two machines of approximately 112.5 MB/s, which is in line with expected best-case data bandwidth. But, we observed lower bandwidths with more nodes in the all-to-all pattern used in map-reduce jobs. For example, in a 5–16 node all-to-all network transfer, we observed 102–106 MB/s aggregate node-to-node bandwidths over any one link. These lower values are caused by NewReno's known slow convergence on using full link bandwidths on high-speed networks [14]. Such bandwidth reductions under some communication patterns may make the use of a single network bandwidth (N) inappropriate for some environments. For evaluating data-intensive computing on our cluster, we use a conservative value of $N = 110$ MB/s.

We also ran experiments using the newer CUBIC [12] congestion control algorithm, which is the default on Linux 2.6.26 and is tuned to support high-bandwidth links. It achieved higher throughput (up to 115 MB/s per node with 10 nodes), but exhibited significant unfairness between the flows, yielding skews in completion times of up to 86% of the total time. CUBIC's unfairness and stability problems are known and are prompting continuing research toward better algorithms [14].

6 Experiences with Hadoop

We experimented with Hadoop on our cluster to confirm and better understand the inefficiency exposed by our analysis of reported benchmark results.

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 at least two maps and one reduce per node (and no replication) results in lower-than-expected performance. We improved the Hadoop sort performance by an additional 2× by adjusting a number of configuration settings as suggested by Hadoop cluster setup documentation and other sources [2, 24, 19]. Table 4 describes our changes, which include reducing the replication level, increasing block sizes,

Hadoop Setting	Default	Tuned	Effect
Replication level	3	1	The replication level was set to 1, to avoid any extra writes to disk.
HDFS block size	64 MB	128 MB	Larger block sizes in HDFS make large file reads and writes faster, amortizing the overhead for starting each map task.
Speculative execution	*true*	*false*	Failures are less common on smaller clusters, so avoid extra tasks.
Maximum map tasks per node	2	4	Our machines can handle more maximum map tasks per node.
Maximum reduce tasks per node	1	4	Our machines can handle more maximum reduce tasks per node.
Map tasks per job	2	$4n$	For a cluster of n nodes, we maximize the map tasks per node.
Reduce tasks per job	1	$4n$	For a cluster of n nodes, we 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.

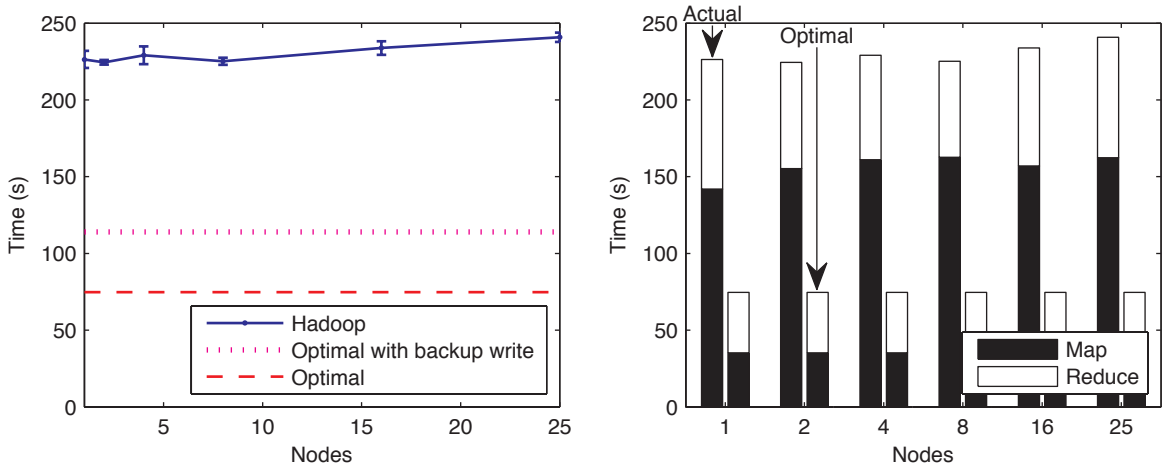
Table 4: Hadoop configuration settings used in our experiments.

increasing the numbers 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. But, they are less common for our smaller cluster size (one NameNode and 1–25 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 synched to prevent previously cached writes from interfering with the measurement and flushed to force disk reads for the input. The sorted output is written to the file system, but not synched to disk, before completion is reported; thus, the reported results are a conservative reflection of actual Hadoop sort execution times.

The results confirm that Hadoop scales well, since the average runtime only increases 6% (14 seconds) from 1 node



(a) Scaling a Hadoop sort benchmark up to 25 nodes.

(b) Time breakdown into phases.

Figure 3: Measured and optimal sort runtimes for the tuned Hadoop cluster. sorts 4GB per node for up to 25 nodes about 3 times slower than optimal, and 2 times slower than an optimal sort that includes an extra backup write for the map output, which is currently Hadoop's behavior. Each sort is of 4 GB per node. Hadoop scales well but is inefficient. The measured runtimes, optimal, and optimal with backup write are shown in **(a)**. The breakdown of runtime into map and reduce phases are shown in **(b)**.

up to 25 nodes (as the workload increases in proportion). For comparison, we also include the optimal sort times in Figure 3, calculated from our performance model. The model's optimal values reveal a large constant inefficiency for the tuned Hadoop setup—each sort requires 3× the optimal runtime to complete, even without synching the output data to disk.

The 6% higher total runtime at 25 nodes is due to skew in the completion times of the nodes—this is the source of the ≈9% additional inefficiency at 25 nodes. The inefficiency due to OS abstractions is already accounted for, as discussed in Section 5. One potential explanation for part of the inefficiency is that Hadoop uses a backup write for the map output, even though the runtimes are short enough to make it of questionable merit. As shown by the dotted line in Figure 3a, using the model equation with a backup write would yield an optimal runtime that is 39 seconds longer. This would explain approximately 25% of the inefficiency. But, as with the sort output, the backup write is sent to the file system but not synched to disk—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 during the map phase. It is unclear what fraction of the potential 25% is actually explained by Hadoop's use of a backup write.

Another possible source of inefficiency could be unbalanced distribution of the input data or the reduce data. But, 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. But, 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 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 commonly attributed to CPU overhead induced by Hadoop's Java-based implementation. And, of course, Hadoop may not be using the I/O resources at full efficiency. More fully diagnosing Hadoop's inefficiency is a topic for continuing research.

7 Verifying the model with Parallel DataSeries

The Hadoop results above clearly diverge from the predicted optimal. The large extent to which they diverge, however, brings the accuracy of the model into question. To validate the sanity of our model, we present Parallel DataSeries (PDS), a data analysis tool that attempts to closely approach the maximum possible throughput.

PDS Design: Parallel DataSeries builds on DataSeries, an efficient and flexible data format and runtime library optimized for analyzing structured data [4]. 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

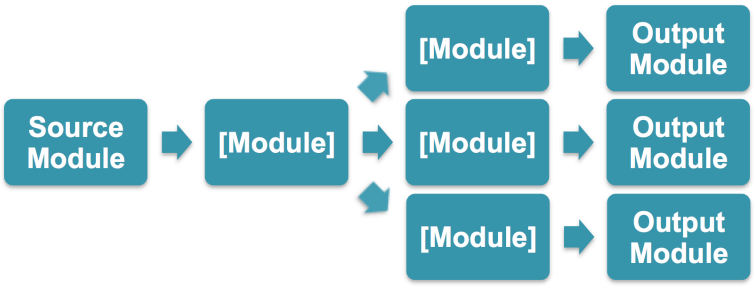


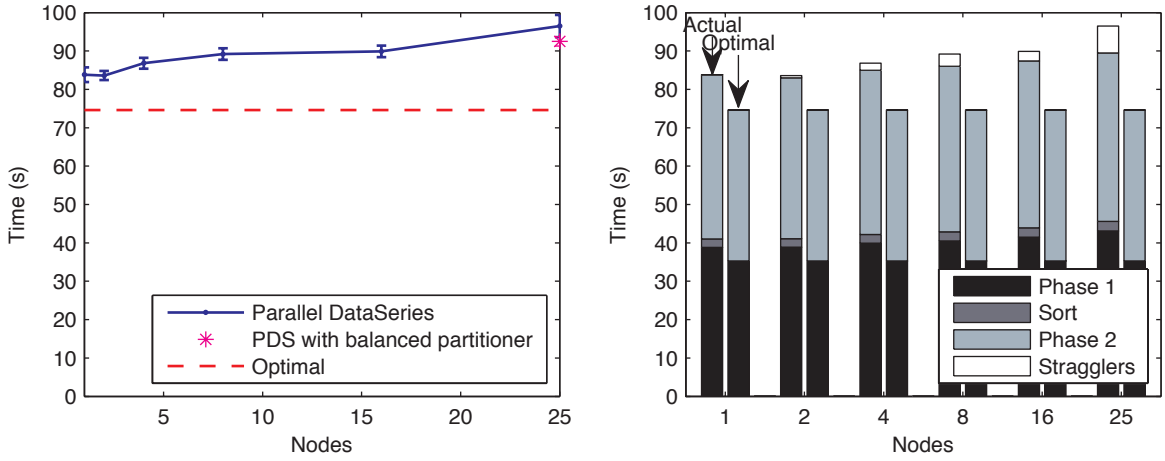
Figure 4: Parallel DataSeries is a carefully-tuned parallel runtime library for structured data analysis. Incoming data is queued and passed through a number of modules in parallel, in a fast and optimized pipeline.

cores (intra-node parallelism) and multiple nodes (inter-node parallelism), to support parallel flows across modules as depicted in Figure 4. Although we have not tuned these parallel modules, PDS inherits high performance for each single thread from DataSeries.

Sort evaluation: We built a parallel sort module in PDS that implements a dataflow pattern similar to map-reduce. In Phase 1, data is partitioned and shuffled across the network. As soon as a node receives all data from the shuffle, it exits Phase 1 and begins Phase 2 with a local sort. To generate input data for experiments, we used *Gensort*, which is the sort benchmark [17] input generator on which *TeraGen* is based. The Gensort input set is separated into partitions, one for each node. PDS doesn't currently utilize a distributed filesystem, so we manually partition the input, with 40 million records (≈4GB) at each node. We converted the GenSort data to DataSeries format without compression, which expands the input by 4%.

We measured PDS to see how close to the optimal predicted performance we could get on the cluster used in the Hadoop experiments. Figure 5 presents the same sort task as run for Hadoop. We repeated all experiments 10 times, starting from a cold cache and syncing all data to disk before terminating the measurement. As with the earlier Hadoop measurements, time is broken down into each phase. Furthermore, average per-node times are included for the actual sort, as well as 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.

PDS performed well at 12% of optimal, and stays within 20% of optimal up to 16 nodes. At 25 nodes, there is a noticeable increase, but still within 30% of optimal. About 4% of that is the aforementioned input expansion. The sort time remains constant at about 2 seconds, or 3%. Much of this CPU could be overlapped with IO (PDS doesn't currently), and is sufficiently small to justify excluding CPU time from the model.



(a) Scaling a Parallel DataSeries sort benchmark up to 25 nodes.

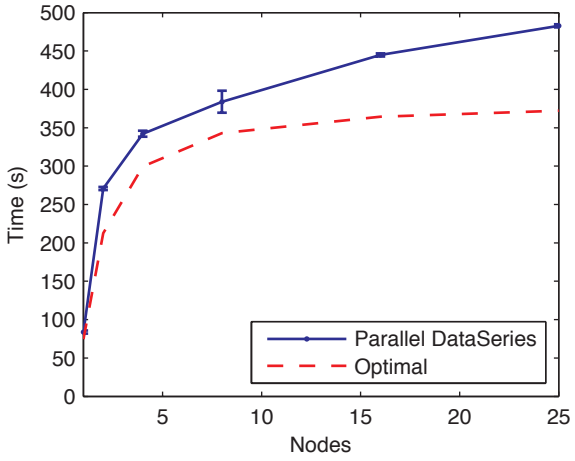
(b) Time breakdown into phases.

Figure 5: Using Parallel DataSeries to sort up to 100GB, it is possible to approach within 12-30% of the optimal sort times as predicted by our performance model. PDS scales well for an in-memory sort with 4GB per node up to 25 nodes in **(a)**, although there is a small time increase around 4 nodes due to network effects, and a larger increase at 25 nodes because of an unbalanced partitioner (our new partitioner achieves 24% overhead for 25 nodes). A breakdown of time in **(b)** shows that the time increases at scale are mostly in the first phase of a map-reduce dataflow, which includes the network data shuffle, and due to the time nodes spend waiting for stragglers, due to skew effects.

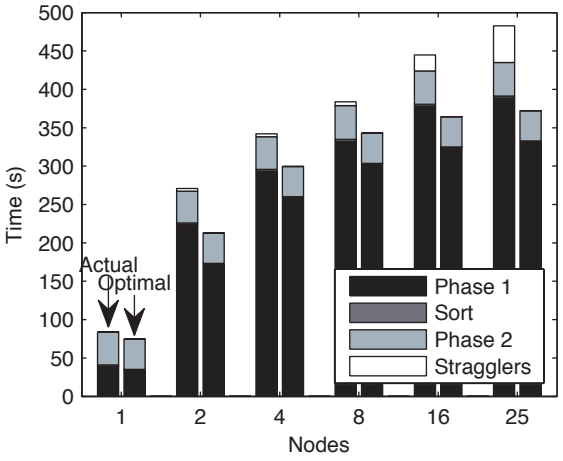
However, the actual performance diverges from the model more than the 7% that has already been accounted for, especially at the 25-node case. Looking at the 25 node runs more carefully, we noticed that 6 of the nodes consistently finished later than the other 19. This led us to notice that those 6 nodes were processing about 10% more work than the others. It turns out that our data partitioner uses only the first byte of the key; hence it partitions the data unevenly for clusters that are not a power of 2. After designing a fairer partitioner and applying it to the 25 node parallel sort, we were able to bring down the overhead to 24%.

The remaining 13% divergence (from the balanced 16-node case) can be explained by two factors: (1) straggler nodes, and (2) network slowdown effects from many competing transfers. Stragglers (broken out in Figure 5b) can be the result of generally slow (i.e., "bad") nodes, skew in network transfer, or variance in disk write times. The 3% straggler time observed is reasonable. The network slowdown effects were identified in Section 5 using `iperf` measurements, and are mostly responsible for the slight time increase around 4 nodes. As more nodes are added at scale, both the straggler effects and network slowdowns become more pronounced.

To see how both the model and PDS react to the network as a bottleneck, we configured our network switches to



(a) Scaling a Parallel DataSeries sort benchmark up to 25 nodes with a slower network.



(b) Time breakdown into phases.

Figure 6: With 100Mbps Ethernet as the bottleneck resource, a 100GB sort benchmark on Parallel DataSeries matches up well with the model's prediction and stays within 13-30% of optimal. As more data is sent over the network with larger cluster sizes in **(a)**, both the model and PDS predict longer sort times that eventually converge. A breakdown of time in **(b)** shows that the predicted and actual time increases occur during the first map-reduce phase, which includes the network data shuffle.

negotiate 100Mbps Ethernet. Just as the $\frac{n-1}{n}N$ term in the model predicts increasingly longer sort times which converge in scale as more nodes participate, Figure 6 demonstrates that our actual results with PDS match up very well to that pattern. The PDS sort results vary between 12-30% slower than optimal, with the same outlier at 25 nodes due to poor partitioning. At 16 nodes, 5% of the time is spent waiting for stragglers. The slow speed of the network amplifies the effect of skew; we observed a few nodes finishing their second phase before a the most delayed nodes had even received all of their data in phase one.

8 Discussion

The experiments with PDS demonstrate that our model is not wildly optimistic—it is possible to get close to its optimal runtimes. Thus, the inefficiencies indicated for our cluster running Hadoop and the analyzed results of the published benchmarks 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. 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 node that will run the corresponding reduce task. This file system activity, which may translate into disk I/O, is unnecessary for completing the job and inappropriate for jobs that can take little time.

One significant effect faced by map-reduce systems is that a job only completes when the last node finishes its work. For our cluster, we analyzed the penalty induced by such stragglers, finding that grows to 6% of the runtime for Hadoop over 25 nodes. Thus, it is not the source of most of the inefficiency at that scale. For much larger scale systems, such as the 1000+ node systems used for the benchmark results, this straggler effect can be expected to much more significant—it is possible that this effect explains much of the difference between our measured 3× higher-than-optimal runtimes and 6× higher-than-optimal runtime of the record-setting TeraSort benchmark result. This straggler effect is why Google's MapReduce and Hadoop dynamically distributed map and reduce tasks among nodes. Support for speculative execution also can help mitigate this effect, although fault tolerance is its primary value. If the straggler effect really is the cause of poor end-to-end performance at scale, then it motivates that these new data-parallel systems examine and adapt the load balancing techniques used in works like River [6] or Flux [23].

It is tempting to blame lack of sufficient bisection bandwidth in the network topology for the inefficiency, 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. But, for our small-scale cluster, all nodes are attached to a single switch that has sufficient backplane bandwidth—so, for our measured Hadoop results, this is not an issue. For most of the analyzed benchmark results, the network topology is not disclosed. For at least some of those, as well, we can dismiss this possible explanation. For example, it is not the explanation for the MapReduce grep (which involves minimal data exchange, because $e_M \approx 0$) or for Hadoop PetaSort. For the latter, Yahoo! used 91 racks, each with 40 nodes, one switch, and an 8 Gbps connection to a core switch (via 8 trunked 1Gbps Ethernet links). For Hadoop PetaSort, the average bandwidth per node was 4.7 MB/s. Thus, the bandwidth used on each uplink was only 1.48 Gb/s in each direction, which is well below 8 Gbps. The other benchmarks may have involved a bisection bandwidth limitation, but such an imbalance would mean that far more machines were used per rack (and, thus, 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 (in)efficiency. but, PDS also provides a promising starting point for understanding 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 the incremental cost of that feature. The same approach can be taken with other features, such as dynamic task distribution and fault tolerance.

9 Conclusion

Data-intensive computing is a growing activity that is being served by scalable, but inefficient, systems. A simple model of optimal map-reduce job runtimes shows that popular map-reduce systems take 3–13× longer to execute jobs than their hardware resources should allow. A simplified dataflow processing tool, called PDS, demonstrates that the model's runtimes can be approached, validating the model and confirming the inefficiency of Hadoop and Google's MapReduce. Our model and results highlight and begin to explain the inefficiency of existing systems, providing impetus for continued improvements in the systems supporting this increasingly popular style of computing.

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] 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.

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

[6] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick, *Cluster i/o with river: making the fast case common*, IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems (New York, NY, USA), ACM, 1999, pp. 10–22.

[7] Randal E. Bryant, *Data-intensive supercomputing: The case for disc*, Tech. report, Carnegie Mellon University, 2007.

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

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

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

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The Google file system*, SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM, 2003, pp. 29–43.

[12] 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.

[13] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly, *Dryad: distributed data-parallel programs from sequential building blocks*, EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (New York, NY, USA), ACM, 2007, pp. 59–72.

[14] Vishnu Konda and Jasleen Kaur, *RAPID: Shrinking the congestion-control timescale*, april 2009, pp. 1 –9.

[15] Michael A. Kozuch, Michael P. Ryan, Richard Gass, Steven W. Schlosser, James Cipar, Elie Krevat, Michael Stroucken, Julio Lpez, and Gregory R. Ganger, *Tashi: Location-aware cluster management*, June 2009.

[16] 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.

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

[18] Owen O'Malley and Arun C. Murthy, *Winning a 60 second dash with a yellow elephant*, April 2009, http://sortbenchmark.org/Yahoo2009.pdf.

[19] Intel White Paper, *Optimizing hadoop deployments*, October 2009.

[20] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems*, HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (Washington, DC, USA), IEEE Computer Society, 2007, pp. 13–24.

[21] Chris Ruemmler and John Wilkes, *An introduction to disk drive modeling*, IEEE Computer **27**, no. 3, 17–28.

[22] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger, *Track-aligned extents: matching access patterns to disk drive characteristics*, USENIX Association, pp. 259–274.

[23] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin, *Flux: An adaptive partitioning operator for continuous query systems*, Data Engineering, International Conference on **0** (2003), 25.

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

[25] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin, *MapReduce and parallel DBMSs: Friends or foes?*, CACM (2010).

[26] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger, *Argon: performance insulation for shared storage servers*.

[27] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta, *A simulation approach to evaluating design decisions in MapReduce setups*, september 2009.