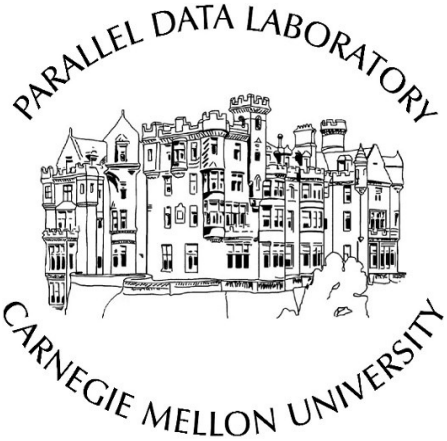



# Understanding Inefficiencies in Data-Intensive Computing

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

Jan 2012

**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 such systems perform relative to the capabilities of the hardware on which they run. This paper describes a simple model of I/O resource consumption that predicts the ideal lower-bound runtime of a parallel dataflow on a particular set of hardware. Comparing actual system performance to the model's ideal prediction exposes the inefficiency of a scale-out system. 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), but that a gap of up to 20% remains for workloads using up to 45 nodes. Guided by the model, we analyze inefficiencies exposed in both the disk and networking subsystems—issues that will be faced by any DISC system built atop popular commodity hardware and OSs.*

**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 and will be a big part of future cloud computing. 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 their efficacy and robustness.

To assist programmers with DISC, new programming frameworks (e.g., MapReduce [11], Hadoop [1] and Dryad [16]) 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, 27].

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 of some bottleneck resource. For example, reported record-setting Hadoop and MapReduce sort benchmark results required 3–10$\times$ more server time than if they were hardware efficient [7]. This effectively wastes a large fraction of the computers over which jobs are scaled; 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 frameworks that are ideal.

An important first step is to understand 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" lower-bound parallel dataflow runtimes and evaluating the actual performance of systems relative to the model. The model predicts the ideal runtime by assuming an even data distribution, that data is perfectly pipelined through sequential operations, and that the underlying I/O resources are utilized at their full bandwidths whenever applicable. 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 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. To focus the evaluation on the efficiency of the programming framework, and not the entire software stack, measured values of I/O bandwidths are used as inputs to the model. An ideal run is "hardware-efficient," if the realized goodput matches the maximum achievable bandwidth for some bottleneck resource, given its use (i.e., the amount of data transferred). Our model can expose how close (or far) a given system is from this ideal. Such throughput will occur if the framework provides sufficient parallelism to keep the bottleneck fully utilized, but not 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 that become wastefully overprovisioned.

To establish an efficiency floor, we focus on a limited but efficient parallel dataflow system called Parallel DataSeries (PDS) that lacks many features of other frameworks, but through careful engineering and a stripped-down feature-set demonstrates that near-ideal hardware efficiency (within $\sim$20% of the ideal) is possible. In addition to validating the model, we use PDS to explore the fundamental sources of inefficiency common to commodity disk and network-dependent DISC frameworks; any remaining inefficiencies of

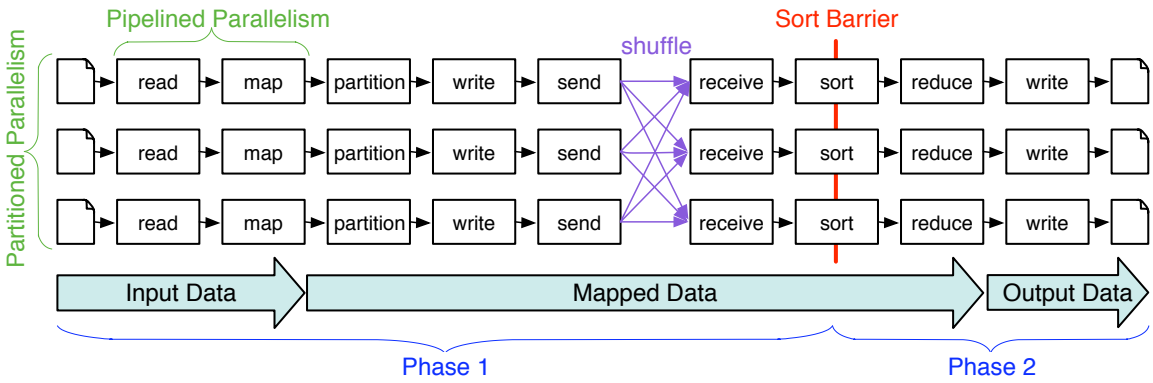



Figure 1: **A map-reduce dataflow.** Parallel dataflow systems make use of partitioned parallelism across nodes and pipelined parallelism across operations. In Phase 1, data is read from disk in a staged pipeline that includes the map and shuffle operations up until the sort barrier. Once all data is received at the destination node, the pipeline begins again with Phase 2, where data is sorted, passed to the reduce operator, and eventually written back to disk (possibly replicated). The total amount of data flowing through the system depends on the input and any changes from the map and reduce operations.

popular map-reduce systems must then be specific to the framework or additional features that are not part of PDS (e.g., distributed file system integration, dynamic task distribution, or fault tolerance). PDS also provides an interesting foundation for subsequent analyses of the incremental costs associated with these extra features.

Our experiments expose straggler effects from disk-to-disk variability and network slowdown effects from the all-to-all data shuffle. The straggler effects are caused by differences in disk performance that are unrelated to failures or bad components, but are instead caused by a new manufacturing method used by all major disk vendors, called *adaptive zoning* [19]. Even assuming that the network is capable of full bisection bandwidth, slowdown effects during the data shuffle still arise from TCP's inability to maximize each receiver's link capacity amongst many competing high-bandwidth flows. Because all these issues arise from characteristics of the underlying hardware and OS, they are faced by any DISC framework built on these standard underlying services.

The remainder of this paper is organized as follows. Section 2 provides background on dataflow parallelism and map-reduce computing along with related work. Section 3 describes the idealized runtime model, including assumptions and usage. Section 4 discusses how ideal performance is computed, describes the experimental setup, and presents disk and network microbenchmark results. Section 5 introduces Parallel DataSeries and uses it to both validate the model and evaluate the effects responsible for lost efficiency. Section 6 discusses more sources of inefficiency in other DISC systems, and Section 7 concludes.

## 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 across different processors and running the same operator in parallel over the partitions. 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 [2] 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 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 the boundaries where the amount of data "flowing" through the system changes throughout the computation. The amount of input data per node ($d_i$) is possibly transformed first by the map operator ($d_m$) and then by the reduce operator to produce the final output ($d_o$). 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. 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 decreases the amount of data ($d_i = d_m = d_o$).

## 2.1 Related Work

Concerns about the performance of map-reduce style systems have recently taken hold in the broader systems community, although similar data processing tasks have also been tackled by commercially available database systems. 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 [31]. In a workshop position paper by two of the authors of this paper [7], 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. By calculating the actual I/O bandwidth of record-breaking sort benchmarks [23], the disk I/O efficiency of DISC systems was estimated to be about 6%. For example, the 2009 Yahoo! Hadoop

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

PetaSort benchmark [24] had an average per-node disk bandwidth of 4.5 MB/s, while other comparable disks are capable of 80-100 MB/s. Similar levels of poor efficiency were found for other published benchmarks, such as the 2008 Google MapReduce PetaSort benchmark [10]. This paper builds on such observations by analyzing every operation of a parallel dataflow and exploring efficiency loss in a quantitative manner.

In other relevant work, Wang et al. use simulation to evaluate how certain design decisions (e.g., network layout and data locality) will affect the performance of Hadoop jobs [35]. Specifically, their MRPerf simulator instantiates fake Hadoop jobs and studies the impact of different parameters, such as job startup times. The fake jobs generate network traffic and disk I/O, all of which are simulated. Using execution characteristics measured from small job instances, MRPerf accurately predicts the performance of larger clusters within 5-12%. Although simulation techniques like MRPerf are useful for exploring different designs, by relying on measurements of actual behavior, such simulations will also emulate any inefficiencies particular to the specific implementations. Our goal is different—rather than seeking to predict the performance of a particular system, we build a simple model to guide exploration of how far from ideal a system's performance may differ.

Other work towards building more efficient and balanced systems is emerging from the TritonSort project [28]. TritonSort is a specialized system that holds the 2010 record for the Indy GraySort competition [23] (i.e., sorting 100 TB of fixed 100-byte records with 10-byte keys). In the past, these competitions have been won by scaling map-reduce systems up to thousands of nodes, such as the 3,452 node cluster that Yahoo! used with a tuned version of Hadoop to win the 2009 (and 2010) Daytona variant of the same benchmark (i.e., sorting 100 TB with a general-purpose framework). Alternatively, DEMSort [26] used only 195 nodes with their own specialized sort implementation using MPI over InfiniBand to win the 2009 Indy GraySort. Remarkably, TritonSort used only 52 higher-end commodity nodes over 10 Gbit Ethernet to win the Indy GraySort in 60% of the time of DEMSort's previous record. These specialized systems help to demonstrate the current inefficiency of general-purpose DISC systems in comparison. They also provide interesting alternative architectures (e.g., for managing a buffer pool across connections) and performance optimizations (e.g., rate-limiting each flow according to its pre-computed share, partitioning disks into read-only and write-only sets). Our idealized runtime model and analysis tools are general enough to evaluate most DISC systems for their ideal efficiency levels, and we seek to understand the fundamental reasons for efficiency loss.

# 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 workloads.

**Assumptions**: The idealized runtime model is intended for data-intensive workloads where computation time is negligible in comparison to I/O speeds. This assumption holds for a large class of data-intensive workloads, including but not limited to grep- and sort-like jobs, such as those described by Dean and Ghemawat [11] as being representative of most MapReduce jobs at Google. 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, oversubscription is not the bottleneck for any of our systems. Furthermore, there is ongoing research on building such networks out of commodity components [3].

The model assumes that input data is evenly distributed across all participating nodes in the cluster, that the cluster is dedicated to the workload, and that each node retrieves its initial input from local storage.

| 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 of the job's output data. If replication is not 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.

The model also simplifies the computation of ideal efficiency by assuming that nodes have homogeneous performance characteristics. In practice, homogeneity and uniform performance across machines isn't possible [19, 22], but variations in performance can at least be limited with similar hardware, or the framework can help to mask heterogeneity with smarter job scheduling so that average hardware performance will be indicative of actual system performance [8, 14]. Most map-reduce systems use dynamic scheduling techniques and 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**: The model calculates the total time by breaking a map-reduce dataflow into two pipelined phases, where the pipeline is as fast as the slowest I/O component in each phase.

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-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

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

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 backup files, sent over the network, and written to the output files. Thus, the best-case runtime is determined by the time to read the initial input from the disk:

$$tgrep = \frac{i}{nD_r} \tag{1}$$

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, and if we assume no additional replication or backup write, then the top equation of Table 3 is used,

simplifying to:

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

In this equation, the time to complete the first phase is the maximum of the two pipelined operations of reading data from local disk ($\frac{1}{D_r}$) and shuffling the data over the network. The $\frac{n-1}{n}$ term is the fraction of data sent over the network. The time for the second phase is just the time taken to write the output data back to local disk ($\frac{1}{D_w}$).

**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 the behavior of a hypothetical system. These parameters to the model, such as the number of nodes and the speeds of the network and disks, are fixed for a given system configuration. Other input parameters are tied to patterns of the workload and of the data itself. For these parameters, instances of the workload can be measured to determine appropriate values, such as the amounts of data filtered by the map and reduce operators, or even a possible increase in intermediate data. For a hypothetical system, or if actual system measurements are not available, estimates of these values can also 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 options of a backup write and sort type, is also largely dependent on the workload characteristics in combination with the system configuration. Specifically, the type of sort depends on the relationship between the mapped data $d_m$ and the amount of main memory available for the sort operator. The backup write option is a softer choice, 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. Hadoop is configured do the backup write by default, at least to the local file system cache even though it is not explicitly flushed to the disk.

Finally, 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 than nominal raw performance, and can have surprising behavior.

# 4 Exploring the efficiency of data-intensive computing

The model provides an aggressive lower bound for ideal efficiency, especially considering the current inefficiencies of DISC computing, but close-to-ideal performance is still achievable. 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 cluster configuration and the disk and network microbenchmarks performed with a few different I/O measurement tools. Section 5 uses a fast stripped-down framework to see how close a real system can approach the model's ideal runtime. Section 6 discusses these results and ties together our observations of other sources of inefficiency with opportunities for future work in this area.

**Cluster configuration**: 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, for reasons which are explained later in Section 5, C1 consists of 45 nodes and C2 of 41 nodes.

### 4.1 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 expect that ideal performance will only be achieved by maintaining full streaming utilization of the I/O bottleneck during periods when data is queued to be read or written.

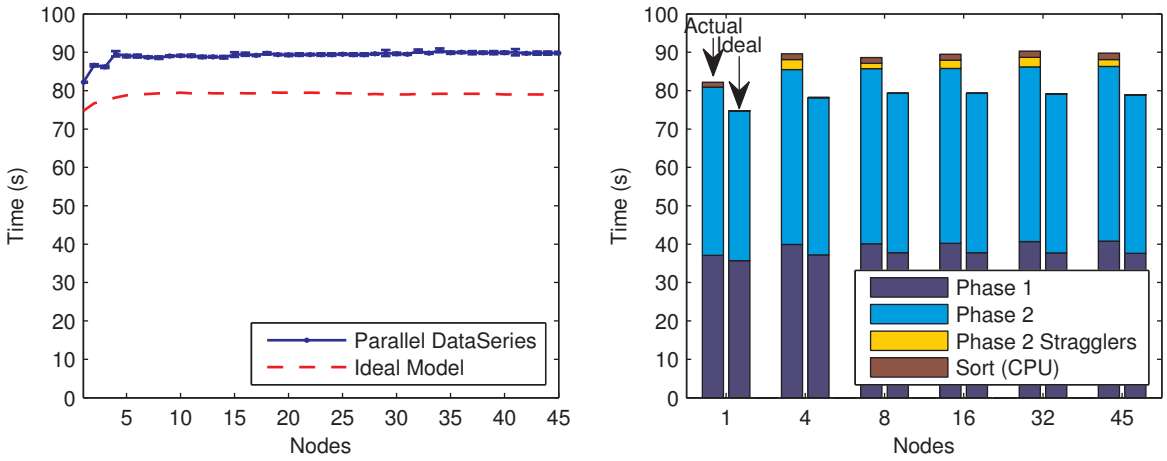
**Disk**: Throughput of the disk is measured with the `dd` utility, using the sync option to force all data to be written to disk. We use 4 GB file sizes and 128 MB block sizes, measuring bandwidth to and from the raw device and also the file system. A separate streaming read benchmark measures the throughput of every consecutive 4 GB LBN range on disk. When files and blocks are sufficiently large for sequential disk transfers, seek times have a negligible effect on performance; aside from any delays imposed by the file system, 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 [29]. For modern disks, "sufficiently large" is on the order of 8 MB [34].

Looking at a sample disk from the cluster, we observe 104 MB/s raw read speeds from the beginning of the disk, which is in line with the disk specifications. While all of our experiments 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 empty 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 induces 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.[2] The measured difference between read and write bandwidth prompted us to use two values for disk speeds 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 idealized model is $N = 110$ MB/s.

We also built an all-to-all network transfer microbenchmark 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.

[2] 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 [21].




(a) PDS sort time.

(b) Time breakdown.

Figure 2: **PDS sort performance on a fast network.** 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 time predicted by our model. PDS scales well up to 45 nodes in **(a)**, sorting 4 GB per node. 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).

## 5 Evaluating the model with Parallel DataSeries

To validate the model, and to explore where efficiency is lost in practice, we present Parallel DataSeries (PDS), a stripped-down data analysis tool that attempts to closely approach ideal efficiency.

**PDS Design**: Parallel DataSeries builds on DataSeries, an efficient and flexible data format and open source runtime library optimized for analyzing structured data [6]. 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.

**Parallel sort evaluation**: 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 our experiments, we used *Gensort*, which is the standard sort benchmark [23] input generator that produces a uniform and random distribution of 100-byte input records with 10-byte key values. Taking advantage of its random distribution, the 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.

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.

**PDS sort on a 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 2(a) graphs the total

sort time achieved in comparison to the ideal model, and Figure 2(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 ideal. 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 *Stragglers* category in the time breakdown represents the average time that a node wastes by waiting for the last node in the job to finish. The Phase 2 straggler effect accounts for about half of the additional inefficiency. 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 ideal 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 were limited and not a continuously growing effect.

**FS-based and disk-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 [5, 36] and was not unexpected. However, it was surprising that even when using a perfectly balanced workload, without any data skew, there was still an effect. Especially since performance laggards in Phase 2 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 3, where files are placed towards the beginning of each disk for what is otherwise an empty filesystem except for the input and output data of our experiments. With this configuration, read speeds on a particular node are quite consistent, but 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 can fit into the shorter inner tracks [32, 30]. Groups of zones each have different sectors-per-track values and, thus, media transfer rates. The raw disk measurements come from the first disk zone, and filesystems will favor the faster zones when a disk is otherwise empty, which is the case for these experiments. However, the filesystem still decides how to break a disk down into subvolumes that span different disk zones. XFS, for instance, uses many allocation groups to manage free blocks and journaling

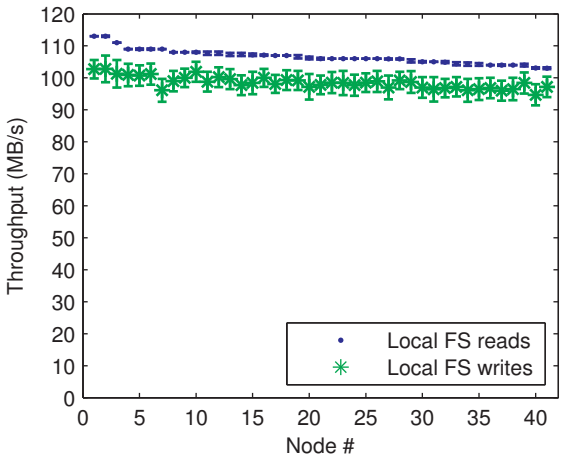



Figure 3: **Read and write bandwidth measurements through the XFS filesystems on our C2 cluster.** Even after pruning away slower nodes, variance across nodes is significant, and writes are slower than reads.

over a disk, and more allocation groups are used for larger disks. 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.

In the last few years, traditional zoning schemes have been extended to aggressively increase areal density with a technique that we refer to as *adaptive zoning* [19]. This 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 (much like CPU binning). Adaptive zoning is responsible for a large amount of the variability in bandwidth of our otherwise identical make and model disks. It is also responsible for larger differences in performance across the multiple disk platters within a single disk.

The configurations of our experiments provide a more controlled environment to understand the factors that affect performance. Some of these factors might have otherwise hurt performance in other settings. For example, we control the use of a new and mostly empty filesystem (so files are stored on the faster outer zones of a disk). Running PDS on an older and fragmented filesystem would reduce efficiency. Also, the nodes that make up our C1 and C2 clusters were selected from a larger set of nodes based on their disk speed measurements, to avoid slower disks on nodes that would have contributed to even higher variations of performance. Avoiding or minimizing the effect of slower nodes is integral to any system that is tied to the performance of the slowest node. PDS's current simple static partitioning strategy that distributes data evenly across nodes would have been especially vulnerable to even one significantly slower node. It is also easy to extend PDS with other partitioners that, for example, can send an amount of data to each node that corresponds to the speed of that node's disks. In the interest of experimental simplicity, a setup that exhibits less variation in node performance matches up better with the assumptions of our model and is consistent with our goal of achieving close-to-ideal performance.

To expose the underlying disk effects further, Figure 4 shows the average disk throughput over ten runs for every consecutive 4 GB block of a disk. Results are shown for a representative sample of a set of twenty-five disks that were not pruned for performance. These modern 2009-era disks are of the same make

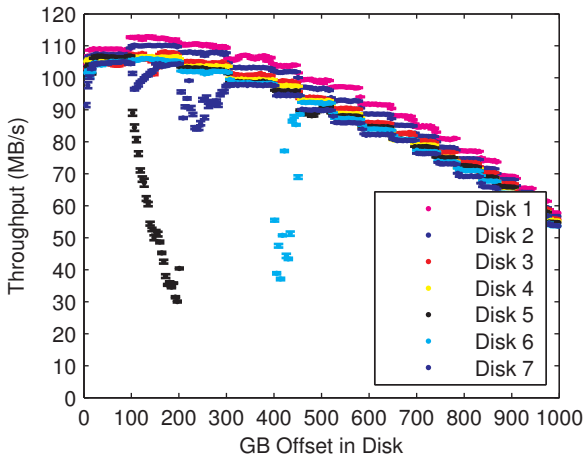


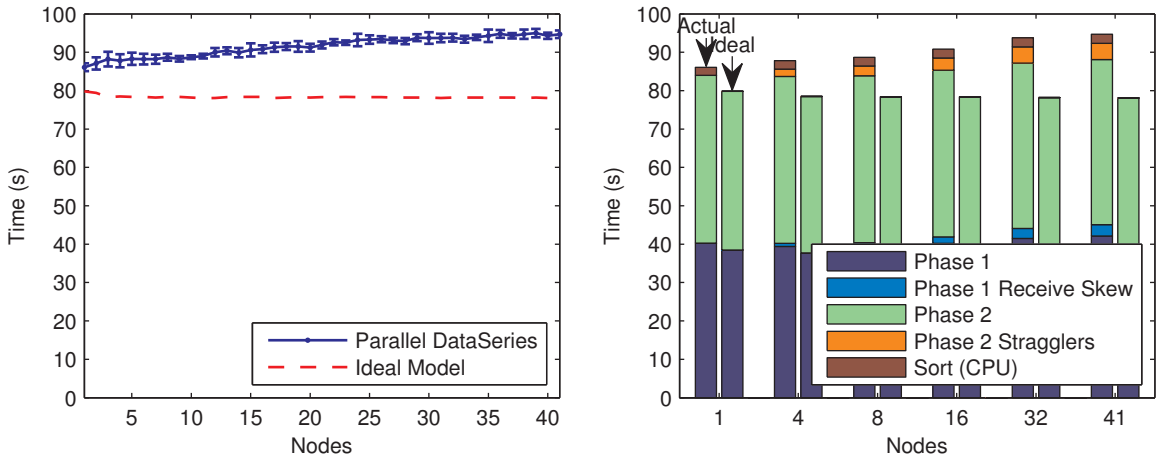

Figure 4: **Full-disk streaming read performance.** Each of the seven representative disks were selected from a set of twenty-five identical-model 1 TB 2009-era disks. Because of new disk manufacturing methods such as adaptive zoning, disk cluster performance has become increasingly variable, creating a heterogeneous environment that complicates efficient job placement strategies.

and model as those from the C1 and C2 clusters. Because of adaptive zoning, even the first blocks on each disk can vary by 20% or more, in this case between 90 and 110 MB/s. The same range of LBNs across disks can vary similarly for the entire disk. A few disks also show consistent stretches of around 100 GB with significantly worse performance, with throughput dropping as low as 30 MB/s. Some areas of poor throughput could be due to other defects or bad sector remappings, although a SMART disk test didn't find any issues and it is more likely that these larger areas of performance differences are another consequence of new manufacturing processes like adaptive zoning.

With these new disk characteristics, placement decisions can have an even greater impact on performance, especially when partitioning work and inaccurately assuming homogeneity. Dynamic task assignment, such as that used by Hadoop or MapReduce, could reduce this load imbalance across the cluster if the associated feature overhead is small enough and the tasks are appropriately sized.

**PDS sort on a 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. Actual performance of our sort experiments reveal that the network speeds are not that high. As presented in Figure 5, parallel sort times continue to slowly increase from 8 to 21% longer than the ideal 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 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 a 4% overhead from ideal. The presence of receive skew doesn't necessarily mean that the transfer is inefficient, only that the 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 and discovered large amounts of network skew and general slowdown effects from many competing transfers during the network shuffle. Figure 6 graphs the resulting throughput to




(a) PDS sort time.

(b) Time breakdown.

Figure 5: **PDS sort performance with a network bottleneck.** 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 an ideal prediction that assumes constant 110 MB/s 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.

complete a full network shuffle microbenchmark, including the time to connect to every server and finish transferring data to all the nodes. With the small overhead of our microbenchmark tracing infrastructure, we achieve 112 MB/s for a 2-way transfer, which matches a basic bidirectional iperf test. A quick drop in network performance occurs when moving from 2 nodes to a 3-way shuffle with six flows (106 MB/s) and a 4-way shuffle with twelve flows (104 MB/s). At around 16 nodes, network speeds become slower than disk read throughput, making the network the bottleneck resource for Phase 1 of a map-reduce. Throughput then continues to slowly decrease down to 98 MB/s at thirty-two nodes and under 94 MB/s at sixty-four nodes. The TritonSort project reported running into an even more severe shuffle slowdown on 10 Gbps Ethernet for an earlier implementation of their system, attributing the slowdown to thread scheduling issues from using smaller 16 KB receiver window sizes [28]. Their final implementation avoided the large trending throughput decline by using a single-threaded non-blocking receiver, but still an initial drop in performance could not be avoided. In a different setting of 1 Gbps Ethernet on our cluster, the CPU can drive the network without any problems, but network speeds are still an issue—that is, something other than CPU overhead is the culprit.

Figure 7 presents the aggregate network link utilization into each node for the last ten 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 [15] 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 [17]. Some of TCP's, and specifically CUBIC's, unfairness and stability issues are known and are prompting continuing research toward better algorithms.

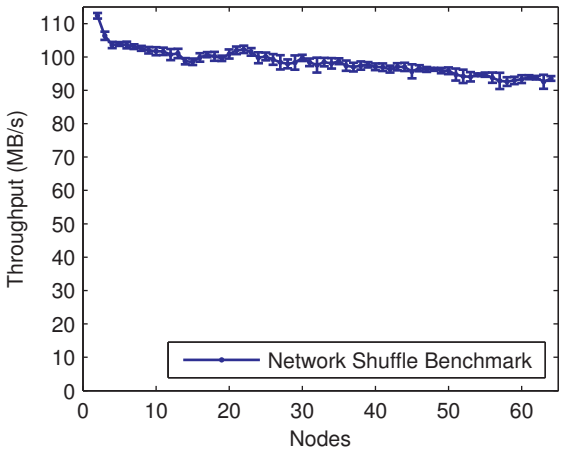



Figure 6: **Network shuffle throughput.** A separate network shuffle microbenchmark measures the average throughput per node, starting just after initiating all connections and ending when the last bit of data has been received at all the nodes. The quick drop in throughput from two to four nodes, and a gradual slowdown afterwards, confirms that the data shuffle is not effectively using the full link capacity of the network.

The closest other issue with TCP that we know of is the incast problem [25], which traditionally manifests itself as a 2–3$\times$ order of magnitude collapse in network speeds when performing many small bursty synchronized reads in an all-to-one pattern. Common application-level solutions for incast, such as introducing jitter and throttling transfer rates, only delay the onset of the problem [20]. A safe and effective solution for incast uses microsecond-granularity TCP timeouts [33]. In our case, a longer all-to-all data shuffle creates even more competing flows that can overflow the switch buffers and cause timeouts, but it also provides more data per connection for TCP to balance over time and it avoids the many synchronization steps that would exacerbate these effects. By recompiling a kernel with 1ms TCP retransmission timeouts instead of the default 200ms, we have verified that a partial incast solution did not improve performance for a few nodes. This is expected because there are very few actual TCP timeouts at that scale, so shortening the timeout wait period has a limited effect.

We have found no other sources exposing problems with longer (i.e., gigabytes per node) all-to-all network transfers in high-bandwidth and low-latency environments. For smaller-sized synchronized reads, the Data Center TCP (DCTCP) project [4] found that multiple simultaneous incast-like effects happen in production environments, and queuing delays still appeared in their cluster for TCP configured with 10ms timeouts. Specifically, requests to read 1 MB of total data spread equally across 40 nodes took around 64ms instead of the fastest time of 8ms for a single-source over 1 Gbps Ethernet. The DCTCP project also noticed that an all-to-all incast variant over 41 machines caused over 55% of the queries to experience at least one TCP timeout. We have identified a sharp increase in TCP fast retransmissions (from packet drops) of around 200 to 10,000 when moving from a two-way to a three-way shuffle. A fast retransmission corresponds at the very least with halving the congestion window, and can easily lead to a full TCP timeout with further drops. We have verified that actual TCP timeouts are not always occuring in tandem with the all-to-all network shuffle slowdown, but just the events of dropped packets are enough to affect performance. We have measured our network shuffle microbenchmark on other clusters and switches with similar results.

To zero in even further on the network effects and deduce their impact on the increasing PDS sort completion times, 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

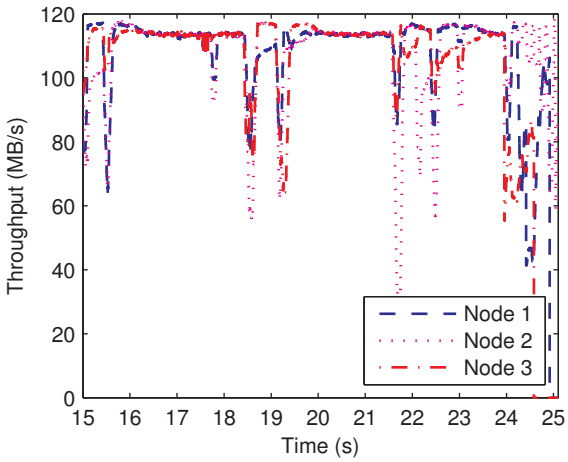



Figure 7: **Aggregate per-node shuffle throughput.** Trace information from the network shuffle microbenchmark is shown for the last 10 seconds of a three-way all-to-all shuffle, smoothed with a 100 ms sliding window. Unexpected network fluctuations and flow unfairness across nodes contribute to partial link utilizations and slower network throughput.

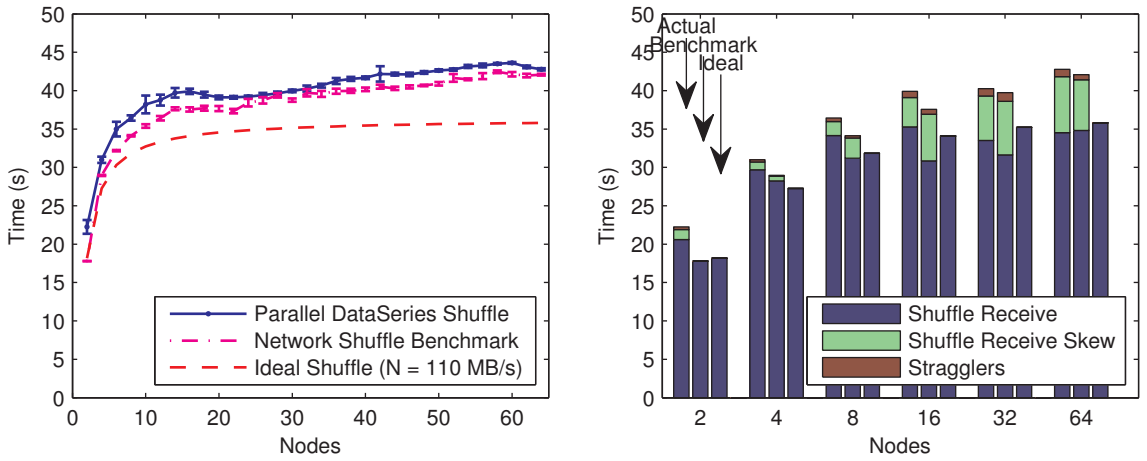
written to disk at the destination. Our idealized model for this scenario just removes the disk components and focuses on the network, where the $\frac{n-1}{n}N$ term in the model predicts increasingly long sort times that converge in scale as more nodes participate. Figure 8 demonstrates that our actual 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, as the small 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 from the network, along with a network slowdown that is responsible for the loss of efficiency.

# 6 Other sources of inefficiency in DISC systems

The experiments with PDS demonstrate that our model is not wildly optimistic—it is possible to get close to the ideal runtime, but popular DISC frameworks don't perform as efficiently. We haven't yet determined exactly why specific widely-used data-intensive computing frameworks don't achieve full per-node efficiency, but our experiments from Section 5 uncovered some fundamental issues with disk stragglers and network slowdown effects that impact all DISC systems, and we have better insights into these issues after tuning and experimenting with PDS in tandem with the idealized runtime model. While the design and engineering details of each particular system have their own associated performance issues, this section discusses other general sources of inefficiency that may come into play.

One class of inefficiencies comes from duplication of work or unnecessary use of a bottleneck resource. Some amount of duplication is commonly used to deal with failures. 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. For longer jobs, these operations are expected and our model takes it into account, so it would not be classified as an inefficiency.

Speculative execution, on the other hand, involves a tradeoff between efficiency and job performance




(a) PDS network shuffle time.

(b) Time breakdown.

Figure 8: **PDS network shuffle performance over 1 Gbps Ethernet.** By focusing on just the PDS network shuffle, increases in the completion times are more than the model predicts, mostly because of network slowdown effects. The measured PDS network shuffle times in **(a)** are slower than the ideal shuffle prediction of the model evaluated with 110 MB/s network speeds, but appear much closer to the measured network shuffle microbenchmark results. The time breakdown in **(b)** confirms that most of the increased PDS network shuffle time includes receive skew, which measures the skew of incoming flow completion times on every node, as well as a general network slowdown effect that also appears in the microbenchmark.

by reserving some percentage of nodes to replicate work that is lagging on another node. Because a job can only complete when the last node finishes its work, the small loss in efficiency from duplicating tasks is usually preferable. Stragglers are a significant issue faced by map-reduce systems, and failures become increasingly important at scale. In our experiments, stragglers and their combination with network skew effects account for most of PDS's efficiency loss when moving from one node to many 10s of nodes, but still are less than a 10% additional slowdown from ideal. For larger scale systems, such as the 1,000+ node systems used to achieve record-breaking sort benchmark results, this straggler effect would be expected to be more significant. Dynamically distributing map and reduce tasks among nodes in combination with speculative execution, as is done by MapReduce and Hadoop on larger clusters, can help mitigate straggler effects [5, 36] in addition to enhancing fault tolerance.

It is tempting to blame lack of sufficient bisection bandwidth in the network topology for much of the network 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 results on our cluster, because all nodes are attached across two switches with sufficient backplane bandwidth. The network topologies for most published benchmarks [23] are not disclosed, but for many we don't believe that bisection bandwidth was an issue for them either. For example, the Hadoop PetaSort benchmark [24] 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.

Other potential sources of inefficiency are poor scheduling and task assignment. PDS's static workload

partitioning over disks with different performance characteristics creates an imbalance proportional to the disk performance skew. Even DISC frameworks that dynamically schedule tasks need to exploit locality and avoid long tail effects from slower nodes. When a node finishes processing all of its local input data, Hadoop will assign it a task with input that is located on another node, necessitating another network transfer delay. This strategy is preferable because it alleviates work from the slower nodes, but it also causes additional I/O overhead.

Some of the inefficiency of other systems 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. Managing flows across system layers (e.g., between HDFS and Hadoop) also appears to be an issue. For Hadoop, part of the inefficiency is also commonly attributed to the CPU and data marshalling overhead induced by its Java-based implementation.

The achievements of specialized systems for sorting data, like TritonSort [28] or DEMSort [26], are noteworthy for their architectures, algorithms, and optimizations. Eventually, some of these systems may also reach the same scale and generality of Hadoop, although not all of the optimizations possible for a specialized system will extend to a general one at the same scale. With TritonSort, for instance, the ability to partition disks into read-only or write-only sets depends on the initial location of the data, for which there is less control on a larger multi-purpose system. Rate-limiting flows according to each machine's calculated throughput also may not generalize, unless the job has full control of its nodes.

Naturally, deep instrumentation and analysis of each system will provide more insight into its inefficiencies. Also, PDS in particular has provided a promising starting point for understanding the sources of inefficiency from the ground up, and it is a good framework for further extensions. 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.

# 7 Conclusion

Data-parallel computation and scale-out performance are here to stay. However, we hope that the low efficiency of popular DISC frameworks is not. Experiments with Parallel DataSeries, our simplified dataflow processing tool, demonstrate that the model's runtimes are approachable in practice. Using the model as a guide, our experiments also identify and quantify hardware and OS inefficiencies that will affect any DISC framework built atop standard underlying services. By gaining a better understanding of the sources of inefficiency, and exploring 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] *HDFS*, http://hadoop.apache.org/core/docs/current/hdfs_design.html.

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

[4] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan, *Data Center TCP (DCTCP)*, SIGCOMM, ACM, 2010, pp. 63–74.

[5] 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.

[6] 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.

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

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Fail-Stutter Fault Tolerance*, 8th Workshop on Hot Topics in Operating Systems (HotOS VIII), 2001, pp. 33–38.

[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] Steven D. Gribble, *Robustness in Complex Systems*, 8th Workshop on Hot Topics in Operating Systems (HotOS VIII), 2001, pp. 21–26.

[15] 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.

[16] 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.

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

[18] 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.

[19] Elie Krevat, Joseph Tucek, and Gregory R. Ganger, *Disks Are Like Snowflakes: No Two Are Alike*, Hot Topics in Operating Systems (HotOS), May 2011.

[20] Elie Krevat, Vijay Vasudevan, Amar Phanishayee, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan, *On Application-level Approaches to Avoiding TCP Throughput Collapse in Cluster-based Storage Systems*, Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW), 2007, pp. 1–4.

[21] 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.

[22] Jeffery C. Mogul, *Emergent (Mis)behavior vs. Complex Software Systems*, EuroSys, ACM, 2006, pp. 293–304.

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

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

[25] 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.

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

[27] 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.

[28] Alexander Rasmussen, George Porter, Michael Conley, Harsha Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat, *TritonSort: A Balanced Large-Scale Sorting System*, Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI) (Boston, MA), April 2011.

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

[30] 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.

[31] 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).

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

[33] 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*, SIGCOMM, ACM, 2009.

[34] 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.

[35] 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.

[36] 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.