Libra and the Art of Task Sizing in Big-Data Analytic Systems

Anonymous
Submission Type: Research

Abstract

Despite extensive investigation of job scheduling in data-intensive computation frameworks, less consideration has been given to optimizing job partitioning for resource utilization and efficient processing. Instead, partitioning and job sizing are a form of dark art, typically left to developer intuition and trial-and-error style experimentation.

In this work, we propose that just as job scheduling and resource allocation are out-sourced to a trusted mechanism external to the workload, so too should be the responsibility for partitioning data as a determinant for task size. Job partitioning essentially involves determining the partition sizes to match the resource allocation at the finest granularity. This is a complex, multi-dimensional problem that is highly application specific: resource allocation, computational runtime, shuffle and reduce communication requirements, and task startup overheads all have strong influence on the most effective task size for efficient processing. Depending on the partition size, the job completion time can differ by as much as 10 times!

Fortunately, we observe a general trend underlying the tradeoff between full resource utilization and system overhead across different settings. The optimal job partition size balances these two conflicting forces. Given this trend, we design Libra to automate job partitioning as a framework extension. We integrate Libra with Spark and evaluate its performance on EC2. Compared to state-of-the-art techniques, Libra can reduce the individual job execution time by 25% to 70%.

1 Introduction

With the wild popularity of distributed data-intensive computation frameworks (e.g., MapReduce [19], Hadoop [1] and Spark [49]) and services [2], recent years have witnessed considerable amounts of investigation on how jobs should be scheduled within a distributed system. Despite all the efforts, relatively less attention has been paid to exactly what the most appropriate schedulable entity should be to optimize for resource utilization. We have seen proposals at both ends of the spectrum, ranging from running an end-to-end job entirely on a single core [37] to running massive numbers of very small jobs [41] on a server cluster. However, little formal consideration has been invested in studying the tradeoffs involved in effectively partitioning the work into tasks that happens in a schedulable job. Instead, partitioning and task sizing are a form of dark art, typically left to developer intuition and trial-and-error style experimentation.

In this work, we argue that task sizing should be an independent engine, just as job scheduling and resource allocation are out-sourced to a trusted mechanism external to the workload. The reasons are two folds: task sizing could greatly impact job performance (by more than 10X times in our experiments), and it could help people decide the right level of parallelism for running the job.

As an illustration of how important task sizing is to job performance, Figures 1 and 2 show how the completion time varies with task size. The minimum point on each curve corresponds to the optimal size. Not only do the optimal task sizes vary significantly across applications or even different parts of the same application workload, the completion time can vary by as much as 10 times! Section 2 further analyzes the most significant factors that should be considered in performing partitioning.

However, we observe that this is a complex, multi-dimensional problem that is highly application specific: resource allocation, computational runtime, shuffle and reduce communication requirements, and task startup overheads all have strong influence on the most effective task size for efficient processing (Section 3.2).

Despite the complexity involved in optimal task sizing, we have two key insights from extensive experiments (Section 2). First, we observe a general trend of the job completion time variation with the task sizes resembling a U-shaped curve across applications (Figures 1 and 2). This trend reflects the tradeoff between full resource utilization and system overhead (such as the task initialization overhead, metadata management overhead, and task scheduling delay) [37]. The optimal task size balances these two conflicting forces. Given this trend, we can employ an adaptive mechanism to find the optimal size dynamically, using a small number of probes from the early portion of the job. This contrasts sharply with the current practice of repeatedly running and profiling *entire* jobs many times [26, 30].

Second, while the trend is noisy in practice, the same adaptation technique still applies as long as we can filter out enough noise from the probe results (Section 3).

Our goal is to provide a dynamic, automatic, pluggable task sizing engine. It should be able to dynamically adjust task sizes to address the runtime variances during job execution. Also, it has to work automatically without user intervention. Lastly, it needs to be a pluggable engine, which can easily integrate with existing systems without requiring changes to existing codebase.

Libra is designed based on the above insights (Section 4). We implement Libra on Spark [49] (Section 5) and evaluated the system on EC2 under a range of setups (Section 6). Compared to state-of-the-art techniques, Libra reduces the job completion time by 25% to 70%.

With Libra, we make the following contributions:

First, we highlight the importance of optimizing task sizes, the challenges involved, and argue for framework level support.

Second, we observe a well-defined, general trend underlying the variation of the job completion time with the task size. This lends to a simple optimization mechanism, using a small number of automatic probes, in contrast to the current practice of repeatedly running and profiling *entire* jobs many times.

Third, we employ an effective adaptation framework following the above insights and integrate it with Spark. The logic is conceptually complementary to existing programming models and independent of specific frameworks. Our implementation shows simple extensions to Spark deliver significant performance benefit.

2 The Elusive Optimal Task Size

2.1 Parallelism in data analytic frameworks

The execution flow for an analytics job is often expressed as a directed acyclic graph (DAG), where each node indicates specific processing logic and the directed edges indicate the flow of data. Several programming models [19, 27, 49, 39, 40] for data analytics systems have been proposed to provide different DAGs to express diverse application semantics. Further, the runtime frameworks implementing these programming models support parallel execution based on the DAGs in an application-independent manner, though at the granularity of nodes (often referred to as *stages*) in a DAG.

However, existing framework-level parallelism support does not specify how to optimally parallelize the *per-stage* execution. A stage is typically further divided into *tasks* by partitioning the input data for that stage, with each task processing a portion of the input data. Tasks within a stage are run in parallel on multiple servers. The developer is free to request resources and partition the job accordingly to fully utilize the available resource. Unfortunately, this also places the burden of optimizing job partitioning on the framework user.

Fundamentally, task sizing reflects resource allocation at the finest granularity. When the tasks are too small, the performance is dominated by various system overhead, such as the task initialization time and scheduling delay; when the tasks are too large, the performance is bottlenecked by the amount of resource available, such as the number of CPU cores and the memory. The completion time for a stage could vary by as much as 10 times depending on the task size (Figure 2). While existing resource allocation solutions abound, they typically address issues such as fairness [22, 12], job performance prediction [45, 46, 23], and guaranteeing the service level objectives [28, 21]. None has optimized the task sizes according to the resource specification.

Existing job partitioning techniques [33, 30, 51, 41] instead address *skewness*. However, [33, 30, 51] are application-specific techniques while [34, 41] will incur high overhead. Again, none of these techniques optimizes for the tradeoff between full resource utilization and low system overhead.

The most commonly used dataflow frameworks Hadoop [1] and Spark [49] currently provide generic guidelines about using a task size of 128 MB, which is the block size of the HDFS (Hadoop distributed file system). Furthermore, Spark recommends a minimum task size that would correspond to at least 100 ms execution time in order to avoid high scheduling overhead.

2.2 Motivating experiments

Consider a simple application, *Sort*, with only two stages, *map* (spreading input data across available CPU cores in parallel) and *reduce* (collecting and merging all intermediate results from the previous stage). Intuitively, one should simply partition the input data into equal-sized portions to take advantage of the parallelism in the system. However, this turns out to be non-trivial even on a single machine dedicated to this workload.

We ran several applications individually on Spark on a single workstation with 4 "executors" (Java virtual machines) in parallel, each given 1 CPU core and 1 GB of memory, with minimal interfering background activity on the machine. These applications are drawn from the HiBench benchmark suite (see Section 6 for detail). We partition each workload into different numbers of fixed-sized tasks at the beginning of each run[1]. We run the experiments multiple times, and show the results as the average of multiple runs, though the errors were small.

[1]In Spark, the data at each stage, whether the initial input data or the intermediate data generated from the previous stage, are partitioned into the same number of portions, which is set in the configuration file.

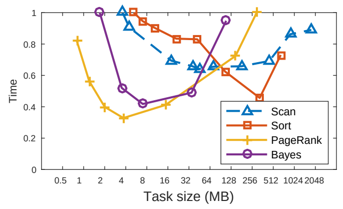

Figure 1: Normalized stage completion time varies with task sizes.

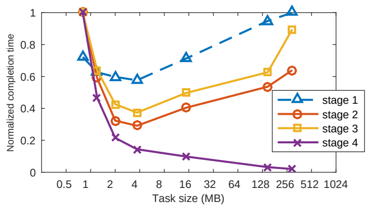

Figure 2: Optimal task sizes for different stages of *PageRank*.

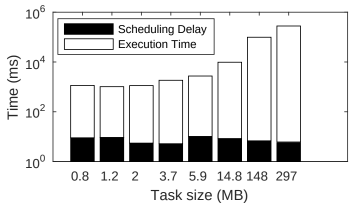

Figure 3: Per-task average overhead breakdown for *PageRank*.

Figure 1 shows how the normalized *stage* completion time varies with the set task sizes for *PageRank* (2.9 GB initial input, third stage), *Scan* (17.4 GB input, first stage), *Sort* (13.3 GB input, first stage), and *Bayes* (5.7 GB input, second stage). We see a U-shaped trend for all four cases, but the exact shape varies by application. Figure 2 further shows a distinct U-curve for each stage of *PageRank*. This follows from Figure 1 since different stages typically involve different operations.

2.3 Observations and analysis

First, very small task sizes may be suboptimal. This is because the various overhead will dominate for small tasks. For each task, its completion time includes significant contributions from the system overhead incurred, such as the scheduling overhead and the task thread startup overhead.

We find that the task scheduling delay is the main overhead within the task execution span. Figure 3 plots the average *per-task* scheduling delay and net execution time for *PageRank* stage 1. We see that the per-task scheduling delay is roughly constant for different task sizes, since the operations involved in scheduling are independent of the task itself. The net execution time per task is roughly constant for sizes from 0.8 MB to 2 MB before increasing exponentially. This is because small tasks cannot maintain high CPU utilization and are more susceptible to other OS overhead. Smaller task sizes also correspond to a larger number of tasks and hence a higher overall scheduling delay and execution time.

Second, very large task sizes are also undesirable. This is because the performance is then bottlenecked on the dominant resource. We can see this more clearly in Figure 4, which plots the number of I/O operations in each stage of *PageRank*. When the task size exceeds 148.5 MB (297 MB and larger), the number of total I/O operations increases sharply for stages 1 to 3. This is correlated with each task being allocated 1 GB memory in our experiments. Based on the Spark default configuration [3], only 60% of (heap space - 300 MB) is used for execution and storage, and the remaining space is reserved for Spark internal metadata, safeguarding against out-of-memory issues. Within the 420 MB memory for execution and storage, by default half of that amount is reserved for caching data, so only about 210 MB memory is really dedicated to execution. Therefore, a task size of 148.5 MB falls within the allowance, while a size of 297 MB triggers virtual memory swapping to the disk.

For stage 4 of *PageRank*, the total number of I/O operations decreases when the task size reaches 297 MB. This is because that stage of the application mainly writes the final data to HDFS, which involves little computation but mostly disk I/O operations, and large task sizes can benefit from batch processing.

Third, Figures 1 and 2 both show convex functions, each with roughly a single optimal point and suggests a well-defined performance optimization goal. This is because of the inherent tradeoff between the performance bounded by the dominant resource (the memory in these cases) at full utilization and the per-task overhead.

Fourth, this optimal task size is specific to a particular experiment setting (application logic, execution environment, and so on), and the optimal *number of tasks* is not determined by the apparent parallelism (measured in the number of executors or cores) in the system. For example, Figure 1 shows that the optimal *number of tasks* for *PageRank*is 800 (the *PageRank* job size divided by the optimal task size), but there are only 4 cores available in our experiments. Also, it is worth noting that we observed the U-curve behavior in Hadoop jobs as well.

In general, the memory resource is always scarce in data analytic frameworks. When a larger amount of memory is available, it will only shift the bottom of the U-curve to a larger task size. If the task size increases beyond a certain value, the amount of I/O time will dominate. On the other hand, if the task size (say, the recommended 128 MB) is smaller than the optimal task size, the framework overhead will dominate.

Figure 4 shows that the resource consumption behavior varies during job lifespan, another reason why the exact shape of the U-curve varies by stage.

When the resource allocated to a workload varies, the exact shape of the U-curve will change accordingly. However, existing data analytics systems usually employ static resource allocation, and therefore the corresponding U-

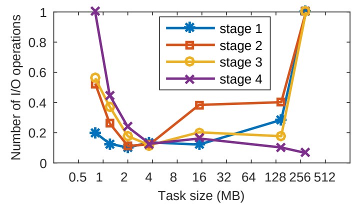

Figure 4: Number of I/O operations for different stages of *PageRank*.

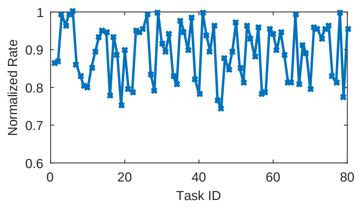

Figure 5: Task processing rate naturally fluctuates during a run.

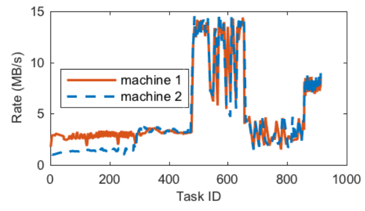

Figure 6: Normalized per-machine task processing rate over time.

curve remains the same during the run.

Given the U shape, using different task sizes could produce very different performance, and the completion time can vary by as much as *10 times*. (Figure 2)! From Figure 1, we estimate that the completion time for the particular stages of *PageRank*, *Bayes*, *Sort*, and *Scan* could be reduced by up to 55%, 74%, 44%, and 65% respectively. Though approximate, these estimates suggest significant room for improvement.

2.4 Summary

To summarize, task sizing has a significant impact on the job completion time, and the optimal task size should be individualized per (stage of a) workload and per resource allocation to minimize the job completion time.

However, the optimal size is difficult to pre-determine purely from the application semantics. Even if we found it, it hardly remains the ideal one throughout the job execution, since these jobs are typically sharing resource on a cluster. Therefore, we should dynamically achieve this U-curve minimum via adaptation.

3 Right-sizing tasks with Libra

3.1 Intuition for task size adaptation

Fortunately, our analysis above provides some intuition for an adaptive mechanism.

The U-curve suggests a well-defined *optimization goal* that is agnostic to the specifics of the application or the run time environment, even though the exact shape of the curve is application dependent. Therefore, the adaptive mechanism should be provided at the framework level. Further, we only need a small number of measurements during the run time to identify the optimal size corresponding to the U-curve minimum. Since the optimal task size varies by stage, we need to reset and restart the adaptation whenever a new stage starts.

Task processing rate. Intuitively, task size adaptation within a job run should be based on the observed processing rate of the earlier portion of the job. The processing rate can be defined as the amount of data already processed divided by the time taken to process the data[2].

The starting point. If given a particular task size by itself, we have no clue whether the next task size should be increased or decreased. Therefore, we always start the adaptation sequence with a small task size, and increase the size after each task completes, until the observed task processing rate starts to drop.

Further, since the U shape varies by the stage within an application (Figure 2), it is difficult to customize the start task size statically based on the application semantics. Therefore, we simply adopt a fixed start size across applications (and stages).

3.2 Bracing for Real-World Complexity

In practice, the exact U-curve is fuzzier than shown in Figures 1 and 2 for many reasons: random fluctuation and data skewness even on a single machine, and differences across machines (static due to heterogeneous hardware or dynamic due to contention for the same resource). Much work so far has shown that it is non-trivial to deal with even a single issue, let alone their combined effects. Even in these two figures (obtained under ideal machine conditions), each point shows the average of multiple runs, though the errors were small.

Our key insight is that, whatever the cause for the inaccuracy, if all the processing rates are measured under roughly the same conditions, they can be compared directly to indicate the direction to go on the U-curve towards the optimal task size. Our main challenge is to ensure these processing rates are *comparable*, per machine and across machines.

Guarding against estimation noise per machine. Figure 5 plots the task processing rate for each task during first stage of *PageRank*, and they fluctuate a lot. So we want to average over multiple tasks to filter out noise.

There are two main sources of noise. First, the normal fluctuation due to measurement accuracy. It can be filtered

[2]In other words, this measures the *processing throughput*, different from the "processing rate" definition (the proportion of the task already completed) in [50].

out with exponentially weighted moving average.

Second, there might be two types of skewness. In a JOIN workload, for example, a skewed task may generate much more output than others due to the uneven distribution of keys. This is *data skewness*. The other case is commonly seen in graph algorithms, where a high-degree node may incur more computation than other nodes. This is *computation skewness*. There are several proposed skewness mitigation techniques [34, 33, 30, 44, 51], and we want to have a pluggable design to easily integrate these techniques into our system.

The machine factor. Data analytics jobs are usually run in parallel on multiple *virtual* machines of some form (JVMs, Linux containers, etc.), called *executors* on Spark. These executors are given the same resource specification and thus will see the same U-curve. Therefore, we should collect data points across machines.

Unfortunately, the *physical* machines hosting the executors may be heterogeneous due to either static hardware capability differences or different utilization levels induced dynamically at run time, the latter especially the case in the face of multiple co-locating applications sharing and contending for the same resource [20]. Therefore, we need to ensure the *executors* are comparable before comparing data samples from different executors. If an executor is unusually slow (often referred to as a *straggler* [50]), we can simply skip it in favor of another.

Slow executors can be detected by assessing how their processing rates deviate from the average among the executors. If we group tasks by executor and plot one line per executor, most lines should more or less overlap while the straggler line should be lower than others, as shown in Figure 6. There are 2 executors in the figure, processing a large *PageRank* job. One of the executors has a concurrent job contending for CPU added from the beginning to 300 s, shown by the lowest line initially. The rises and drops in the figure represent different stages of *PageRank*.

Interaction between the job and the executors. In practice, skewness and slow executors are present simultaneously. They can both be detected by assessing the processing rate deviation from the average, across tasks per executor and across executors respectively. These two effects interact, so the detection is relative to the average condition across tasks and across machines. This requires per-executor "noise filtering" and executor selection to work together.

3.3 System overview

Following from the above discussion, the primary component of Libra is *dynamic task sizing* (DTS). The basic idea is to start with a small task on each *executor* (i.e., a container) as a probe, collects task completion time and task size[3] when it finishes, adapt and converge to the optimal task size as the job progresses. Furthermore, Libra will filter out the observation noises in task completion time along the run. Libra also has a secondary component, *dynamic executor selection* (DES), to detect contention issues across executors and proactively switch to a better one if contention detected.

Libra is a simple add-on to JobManager[4] in existing data analytics frameworks, such as Hadoop and Spark. At the start of the job, we expect the native framework scheduler to allocate resource in terms of some number of executors, each given some CPU cores and memory. This allocation stays the same throughout the job run, as is current practice. DTS then takes over. The initial task size is set to a predefined value. When a task finishes, it will report its size and completion time to the job manager. The task processing rate can then be calculated, which DTS uses to determine the "right" task size for each executor. The average processing rate is also monitored on all machines. If a slow executor is detected, DES will be triggered to find a better executor. When DES switches to this new executor, the task size will be set to the latest size calculated by DTS for fast ramp-up.

Libra represents a pluggable control framework that adapts in a intra-job manner to the *allocated* resource. Further, it simplifies the configuration and user sophistication in an application-agnostic manner.

4 Libra design details

4.1 Preprocessing

Figure 5 shows that task processing rates tend to be noisy due to measurement accuracy and other fluctuations, so we use exponential smoothing [4] to filter out the noise. More specifically, we record the processing rates of X tasks, $R_t, t = 1, 2, ..., X$. These X tasks must have the same size S_i. We calculate the filtered task processing rate C_{S_i} corresponding to size S_i:

$$C_{S_i} = \alpha \times C_{S_{i-1}} + (1-\alpha) \times R_t, t = 1, 2, ..., X (0 \leq \alpha < 1)$$

Currently X is set to $\max(3, 10\%\ of\ parallelism)$.

α is the smoothing factor, and through experiments across many jobs, we found that setting α to be 0.6 yields good results across the board.

Similarly, if task i with processing rate R_i is executed on executor j, we will update the filtered executor throughput as:

$$C_j = \alpha \times C_j + (1 - \alpha) \times R_i$$

[3]For map tasks, it is input partition size; for reduce tasks, it is the size of the aggregated intermediate results sent to one reduce task.

[4]For example, in YARN it is ApplicationMaster daemon, and in Spark it is SchedulerBackend daemon.

4.2 Dynamic Task Sizing

Strawman solution. As discussed in Section 3.1, Figure 1 and Figure 2 exhibit U-curve pattern, which suggests a well-defined optimization goal. We can use Gradient Descent to solve the optimization problem.

More specifically, all executors start with the same task size S_0 at the beginning. When task k finishes (size S_k, filtered processing rate C_{S_k}), the gradient of the task size is calculated as $dS_k = \frac{C_{S_k} - C_{S_{k-1}}}{S_k - S_{k-1}}$.

Then, the next task size is updated as (α_{learn} is the learning rate of gradient descent algorithm): $S_{k+1} = S_k + \alpha_{learn} \times dS_k$.

However, the solution above suffers from multiple issues. Firstly, measurements with larger gradient dS_k will have unfairly higher impact on new task size. This may cause the algorithm miss the optimal point. Secondly, noisy gradient dS_k could add randomness to S_{k+1}. Thirdly, it cannot guarantee convergence.

Refinements. To solve the above issues, we use ADAM technique [32] to improve the gradient descent algorithm. The main idea is to normalize dS_k's effect on S_{k+1}. Also, dS_k needs to be exponentially filtered to remove the noise.

$$m_k = \beta_1 \times m_{k-1} + (1 - \beta_1) \times dS_k$$
$$v_k = \beta_2 \times v_{k-1} + (1 - \beta_2) \times dS_k^2$$
$$S_{k+1} = S_k + \alpha_k \times \frac{m_k}{\sqrt{v_k} + \epsilon}$$

We use v_k to characterize the size of the gradient dS_k, and counteract its effect on next task size S_{k+1} with $\frac{1}{\sqrt{v_k}}$. To guarantee acceleration, we let the learning rate α_k decay by $\frac{1}{\sqrt{k}}$ in each iteration, i.e., $\alpha_k = \frac{\alpha_0}{\sqrt{k}}$. ϵ is used to avoid division by zero.

We set β_1 to 0.9, β_2 to 0.999, α_0 to 0.001 and ϵ to 10^{-8}, as is recommended by [32].

Discussion. DTS is initialized and applied *per stage* of an application workload. This is different from Hadoop or Spark, which only sets the task size at the beginning of *each job*, for all stages. DTS does not affect the number of stages run, and we leave it to the application to determine the number of stages needed.

Further, the DTS operations at different stages are independent. Therefore, we only need to consider the *input data* size to each stage, whether these are the initial input to the entire job (for the first stage) or the intermediate data from the previous stage. The amount of output data generated by each task only affects the DTS operations of *the next stage*. A particular stage of a workload might be completed with different numbers of tasks *from one run to another*, and the task sizes could vary *within a run* and across runs, depending on the machine conditions. Different applications and stages are typicially completed with different numbers of tasks of variable sizes.

4.3 Dynamic Executor Selection

To address the machine factor, DES module will detect and remove the contended executors, ensuring that the processing throughputs between executors are comparable, i.e., all executors see a similar U-curve, so that DTS can work efficiently.

The input to DES is a sequence of executor processing throughputs, and DES will run outlier detection algorithm to identify and remove the contended containers.

When each task i completes on executor j, DES checks container's filtered processing throughput C_j, and compares it to C, the exponentially weighted moving average of the task processing rate across all executors.

If $C_j < (1 - threshold)C$, executor j is potentially too slow. The $threshold$ is a multiple of the standard deviation of C. We suspend executor j, launch the next d tasks (1 partition each) on d "backup executors" (called *idle executors*). These executors are initialized at the beginning of the job but do not run any specific tasks until activated. They both execute the task and serve as the probe, leveraging the power-of-d choices [38]. We use a small size for the probe tasks for fast feedback.

When all the probe tasks finish, the executor with the highest processing rate is selected as the new active executor in place of j. We wait for all executors to finish before making the selection. Since d is usually very small (around 4 based on our experiments) and the executor selection process is typically not on the critical path of the next task launch, the delay incurred waiting for all executors to finish is negligible.

It is worth noting that ourlier detection module is a pluggable design, and users can freely integrate other outlier detection algorithms such as K-nearest Neighbour [5] and Support Vector Machine [6].

5 Implementation

We implement Libra on Spark [7]. We plan to release the source code soon.

5.1 Job execution on Spark

On Spark, tasks are executed in *executors*, with a fixed amount of resource at the beginning of each job run. When a job is submitted, the static job partitioner (`DAGScheduler`) will divide it into multiple stages, each stage then statically partitioned into parallel tasks.

The `SchedulerBackend` manages the executors and launches a new task when an executor emerges with available resource.

`Map` stage tasks read data from HDFS, while `reduce` stage tasks read from the output in the previous stage, which are hashed into a few buckets, and one `reduce` stage task will read one bucket of data.

5.2 Libra additions to Spark

Libra mainly adds two modules, DTS and DES, to the `SchedulerBackend`. A *Contention Detector* keeps track of the executor capability variation. When a task completes and frees up an executor, the `SchedulerBackend` will first check if this executor experiences contention. DES will select a new executor if needed. Otherwise, DTS computes the right size for the next task on this executor.

Calculating task processing rates. When a task completes, it sends a status update (including task size and execution time) to the `SchedulerBackend`. The `SchedulerBackend` then divides task size by execution time to compute the raw task processing rate.

Dynamic Task Sizing. For `map` stage tasks, we change task size by allowing the task to read multiple consecutive partitions from HDFS. For `reduce` stage tasks, we hash each stage output into a large number of buckets, and assign various number of buckets to `reduce` stage tasks to change their sizes.

Dynamic Executor Selection. `SchedulerBackend` runs the DES algorithm. In order to reduce the JVM launch overhead, we launch a few idle executors at the start of the job. The idle executors only add a small amount of maintenance overhead and slight memory overhead. In our setting, they do not consume CPU resource until activated.

Patches to the Spark API and HDFS. The Spark API assumes a static, fixed-size data partition for each task, and we replace that with a call to obtain a variable-size partition. We also modify HDFS API to enable merging multiple physical chunks stored in HDFS.

6 Evaluation

6.1 General setup

Cluster. We run all experiments on EC2. Small-scale experiments use 8 m4.xlarge VMs and large-scale experiments use 100 `m4.2xlarge` VMs. All the microbenchmarks are run on the small-scale cluster for easy control and comparison, since we need to manually introduce contention in several experiments.

Given the possibility of performance variation across EC2 VMs, we always run comparative experiments in quick succession on the same VMs. We ran the experiments repeatedly and the results were consistent. Therefore, error bars are omitted from the plots for legibility.

Table 1: Task Size Ramp-up Time (min/mean/max)

	Rounds of tasks	Completed stage portion
PageRank	3 / 5 / 9	4.5% / 9.4% / 13.2%
Sort	2 / 5 / 8	10.7% / 16.6% / 24.4%

Workloads. We use application benchmarks from the HiBench suite [8][5], using *Sort*, *Scan*, *PageRank* (implemented on GraphX), and *Bayes* as representative workloads for batch processing, database queries, graph computation, and machine learning based applications respectively. The first two are I/O intensive, while the other two are CPU intensive. *PageRank* is the most prone to skewness, while *Scan* the least. The other applications in the HiBench suite behave in similar ways to the representative in their respective category.

The input data are generated by HiBench following suitable Zipf distributions and show skewness.

In most experiments, we run both a long job and a short job for these applications. Long jobs typically mean *PageRank* with 4.6 GB data, *Bayes* 10.52 GB, *Scan* 17.4 GB, and *Sort* 29.8 GB. The sizes are chosen to ensure similar job completion times across workloads. Short jobs vary in size, and will be stated for individual experiments. We also tried different data sizes, but the results are qualitatively similar to those under "long jobs" and "short jobs" correspondingly.

Default Spark setup. We use the *spread-out* mode, allocating enough memory to avoid out-of-memory issues. The number of executors launched decides job parallelism. Unless otherwise stated, each executor is given 1 CPU core and 1 GB of memory [6].

The current Spark documentation recommends setting the task size to be either at least 100 ms long or 128 MB, the HDFS block size. Therefore, "default Spark" in later experiments means the task size is 128 MB.

Performance metrics. We measure the overall job completion time in most experiments, and sometimes overhead in terms of the scheduling latency.

6.2 Control parameters

We first perform experiments to identify suitable values for the control parameters.

Initial task size. Table 1 illustrates the ramp-up speed of the DTS algorithm. We run *PageRank* and *Sort* with the initial task size ranging from 1 MB to 30 MB. For each stage in each initial task size setting, we record how many

[5]We tune HiBench to generate jobs with different input sizes, but we do not change the DAGs of the jobs across the run.

[6]Note that we make no claims about whether this is the most appropriate resource specification for the workloads we run. The goal of Libra is to optimize for *already allocated resource*, in terms of the number of executors and the resource for each, based on whatever criteria deemed appropriate by the cluster wide resource manager.

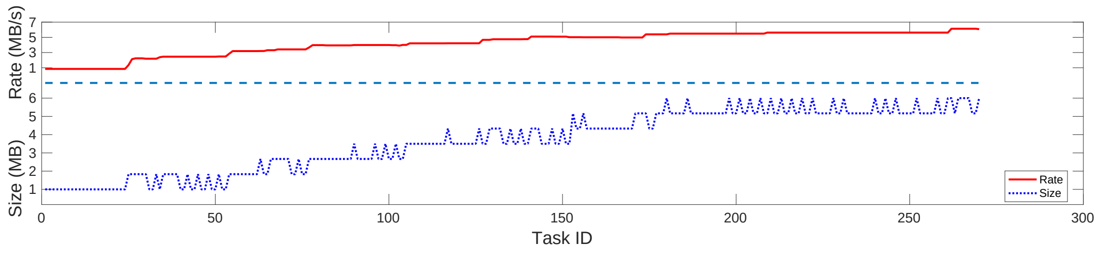

Figure 7: DTS processing rate ramp-up to convergence.

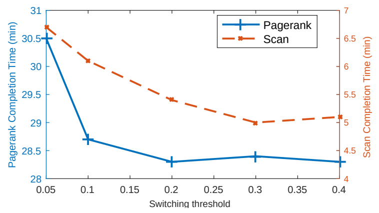

Figure 8: Job completion time vs executor switching threshold.

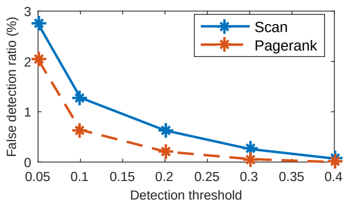

Figure 9: False detection ratio vs executor switching threshold.

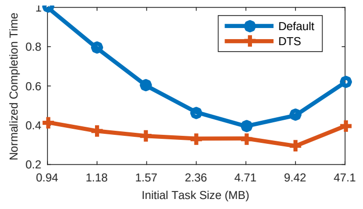

Figure 10: DTS vs default Spark under various initial task sizes.

rounds of tasks are needed and calculate the percentage of the stage (in the amount of input data processed) already completed before convergence.

Figure 7 illustrates DTS in action in the third stage[7] of an example *PageRank* run. Each data partition size is 1 MB. Successive task sizes will continuously adapt to achieve the optimal task size, and the task processing rate will converge to a high value. Note that our task size increments are discrete, and can only take integer multiples of the initial partition size.

Since we leverage the static partitioner in Spark, the initial task size is the same as partition size. While these two can be decoupled, it is preferable to start with a small task anyway and then ramp up the size. Given we also prefer to adjust task sizes in fine steps, we set the intial task size (and the partition size) to a small number, 5 MB, in subsequent experiments, unless otherwise noted.

Executor switching threshold. We need a threshold to detect a particularly slow executor. Intuitively, this should be a multiple of the standard deviation of the processing rates across executors to detect outliers. In practice, we find that a fixed fractional deviation suffices, defined as the reduction from the average rate across executors. The ideal value should respond to slow executors quickly without causing unnecessary executor switching. This is especially important for small jobs for which there are few rounds for adaptation.

We ran *PageRank* and *Scan*, using 8 executors on 8 VMs, each executor given 1 core and 700 MB memory. In the first experiment, a background workload is added to one VM via the Linux `stress` and `dd` commands to simulate constant contention, and we measure the job completion time as the detection threshold varies (Figure 8). In the second experiment, there is no background contention, and so any executor switching is considered a false positive and contributes to the DES overhead. The false detection rate is calculated as the number of executor switching instances during an entire run, and plotted against the switching threshold (Figure 9, note the small scales on the y-axis in that figure). Although an optimal value could be set per application, in general the performance appears insensitive to a threshold larger than 0.3. This is because, with exponential weighted moving average applied, the weighted processing rates for successive tasks are generally within 10% of one another, and so a detection threshold of 0.3 can always accurately distinguish contention from normal fluctuation. We therefore empirically set it to 0.3 for all applications.

The number of executors to probe. We adopt power-of-d choices in DES. Experiments show that the value of d affects the performance less than the number of idle executors. Therefore, we omit detailed results and set d to $\min(4, number\ of\ idle\ executors)$. Libra favors at least 4 idle executors. We maintain idle executors instead of initializing an executor on a new machine on demand, because the JVM initialization takes around 1 s. The cost of maintaining idle executors is low, including a few heartbeat messages and a little memory overhead. There is no CPU overhead unless a probe task is launched.

[7]This stage involves both I/O and some computation, so the processing rate is a mixture of the I/O and computation throughput.

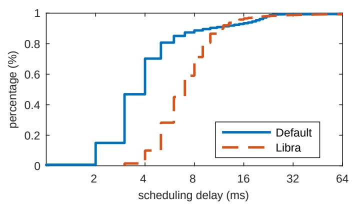

Figure 11: Added per-task scheduling overhead due to DTS.

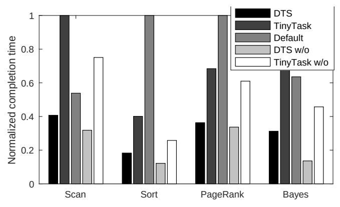

Figure 12: DTS performance vs TinyTasks and default Spark.

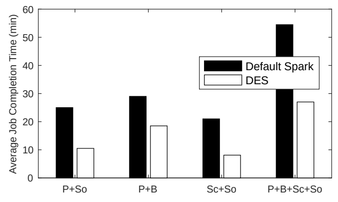

Figure 13: DES performance under dynamic contention.

6.3 Task Sizing Microbenchmarks

We next run microbenchmarks for the task sizing component (DTS) of Libra. The experiments in this section were run free of contention, with DES disabled.

DTS scheduling overhead. Since DTS needs to calculate the task sizes, we measure the task scheduling delay thus incurred. Figure 11 shows that Libra adds a median delay of around 4 ms in Libra compared to default Spark from an example run. This delay is independent of the application workload and negligible compared with the task execution time.

The benefit of per-stage task sizing. Figure 2 shows each stage of *PageRank* favors a distinct optimal task size, whereas the default Spark sets the same size for *all* stages. Therefore, we compare DTS and default Spark under various initial task sizes[8]. Figure 10 shows the normalized *PageRank* completion times for both. DTS always outperforms default Spark, even when default Spark approaches its own optimal performance point. This confirms that DTS responds to per-stage dynamics.

DTS vs TinyTasks and default Spark. In the literature, most task sizing discussion revolves around skewness mitigation or some application specific techniques (such as for graph computation). TinyTask [41] made a case for small task sizes to mitigate skewness in an application independent manner, although at the expense of generating a large number of tasks and incurring a high scheduling overhead. We consider TinyTask without scheduling delay as the state of the art regarding skewness mitigation. We therefore compare DTS performance to TinyTasks and default Spark on the 8 VMs.

The TinyTask proposal did not explicitly recommend a task size, while the Spark documentation suggests a minimum execution time of 100 ms per task, or the overhead would dominate. Therefore, we use the corresponding size (around 0.8 MB) for TinyTask. Since the default Spark scheduler incurs high overhead when running tiny tasks, we also compare the job completion time without including the scheduling delay.

[8]by partitioning the input data into 100, 500,1000,2000,3000,4000 and 5000 pieces

Figure 12 shows that DTS outperforms both default Spark and TinyTasks by significant margins. This is because DTS tries to operate at or near the optimal task size, whereas the other two schemes do not, as per the U-curve behavior noted in Section 2. DTS even outperforms TinyTasks without scheduling overhead. This is consistent with Figure 3. TinyTasks performs the worst for *Scan* and *Bayes* because both workloads involve substantial numbers of I/O operations, which favor large task sizes. Default Spark performs the worst for *Sort* and *PageRank* because these workloads are most susceptible to skewness, which can be mitigated by TinyTasks.

6.4 Executor Selection Microbenchmarks

Recall that DTS relies on DES to ensure the executors can achieve comparable performance. We next study the executor selection component (DES) of Libra through microbenchmarks to assess how well it reacts to executor capability variation. The experiments are run with static, fixed task sizes and various contention levels on the executors, with the task sizing component disabled. Although DES would use idle executors for execution, the total amount of resource used by DES and the schemes compared against are the same.

DES in the face of contention. We ran the 4 large jobs in 16 executors on 8 VMs for all the experiments.

First, we launch pairs of applications (or all) simultaneously. Figure 13 plots the average job completion time across all workloads in each run, and shows that DES reduces this by 40% to 60% compared to the default Spark.

The exact reduction depends on the initial contention. Recall that the four applications (*PageRank*, *Scan*, *Sort*, and *Bayes*) have different resource usage profiles. When a pair is initially contending with each other for the same type of resource, DES could bring huge benefit by switching a workload to other executors to avoid further contention. Even if the initial placement of the jobs does not incur significant contention, it is possible that some level of contention still appears mid-job, which can also be reduced by DES.

Second, we use the Linux `stress` command to generate controlled background workloads, and compare the

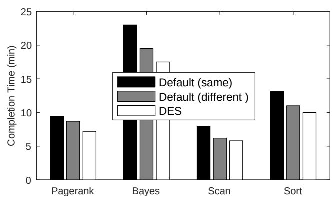

Figure 14: DES performance under controlled contention.

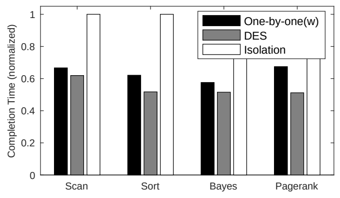

Figure 15: Job performance in different machine sharing scenarios.

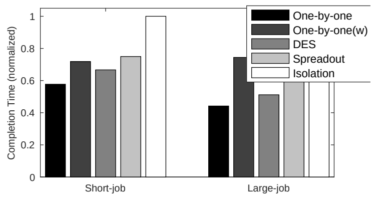

Figure 16: Job performance in different machine sharing settings.

performance between three cases: Default Spark with the same type of single-source contention (CPU, memory, or disk I/O) across executors, Default Spark with different single-source contention, and DES with the same type of single-source contention (Figure 14). Default Spark suffers most when the workloads contend for the same type of resource, while DES actively rearranges co-locating workloads via executor selection to avoid this contention.

This shows that even if a contending workload has different resource usage profile than the foreground workload (e.g., an I/O intensive foreground over a CPU-intensive background), contention still happens. In other words, even if workloads are strategically placed together based on prior knowledge of their overall behavior, some contention is still inevitable. Therefore, it is essential to activate DES during the run time. The performance improvement from DES is not significant in this case because we only created contention on one of the VMs to simplify the experiments.

Finally, we measure the overhead of DES without contention. This arises from unnecessary executor switching due to misdetection and is very small, ranging from 5-7% for the four application workloads.

DES vs isolating applications. Since DES reacts to contention by using an alternative, which may also be contended, we assess how it compares with enforcing resource isolation between applications. Arguably, resource isolation provides the most stable performance.

In this set of experiments, we launch 4 executors on 4 VMs, and launch applications in four ways: (i) The applications are launched one at a time ("one-by-one"), each using all 4 VMs; (ii) The applications are evenly spread on 4 VMs, i.e., each VM sees all 4 applications ("spreadout"); (iii) and (iv) Each application is confined to one distinct VM initially, but (iii) enables DES functionality, and allows applications to switch to a different VM during the run, whereas (iv) confines the application to the same VM throughout ("DES" and "isolation" respectively). Case (i) enforces *temporal isolation* between applications, whereas case (iv) enforces *spatial isolation*. Figure 15 shows the individual job completion times for cases (iii) and (iv), and Figure 16 shows the average job completion time across all four applications in each case. For (i), we also calculate the average completion time across jobs with the job launch latency included, indicated by the "one-by-one (w)" bars in both figures. The exact average time varies with the job launch order, but the qualitative behavior stays the same.

As expected, DES outperforms the default Spark and spatial isolation. The latter fails to efficiently utilize the available resource. This suggests "spatial partitioning" [35] of resource can sometimes be suboptimal. Case (iii), with no better machines available, is the most challenging scenario for DES, but DES still provides benefit by regrouping the contending applications. Compared to temporal isolation, while DES cannot achieve the same individual job performance, it does reduce the wait time before job launches.

DES vs Speculative execution. The standard approach to mitigate stragglers is to speculatively launch a duplicate copy of the task [19], proactively or reactively. When one copy finishes, the unfinished copy is stopped. While speculation does not maintain machine capability, it mitigates the effect of unpredictable dynamics.

We run an emulation of speculative execution as a proxy for the optimal performance[9] for DES for comparison. We run each job twice, ensuring that a task is run on different VMs in the two runs. To compute the job completion time, we first find the shorter execution time between the two runs per task and then take into account parallelism. We emulate the scheduling process by always assigning the next pending task to the executor which sees the smallest accumulated task completion time. After scheduling all pending tasks, we choose the shortest accumulated task completion time as the overall job completion time. Note that this emulated process strictly outperforms and upperbounds any *actual* speculative execution schemes.

Figure 17 shows the performance for the regular large job workload, where the applications contend with one another to generate variable machine conditions. DES usually achieves within 10% of the speculation performance. This shows that, by minimizing contention, DES achieves very good performance.

[9]It is impractical due to requiring 100% resource overprovisioning.

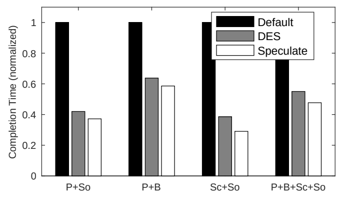

Figure 17: DES vs emulation of speculative execution.

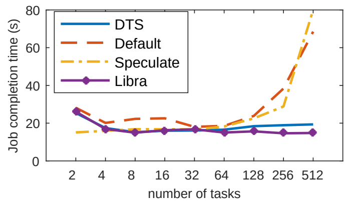

Figure 18: Comparison of short job performance.

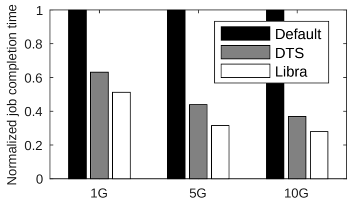

Figure 19: Long *PageRank* job performance.

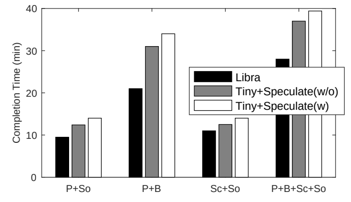

Figure 20: Libra vs TinyTasks combined with speculation.

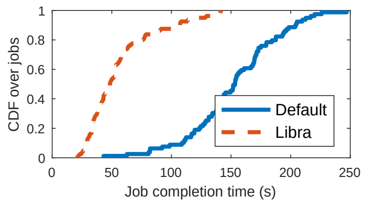

Figure 21: Scan completion time on a large cluster.

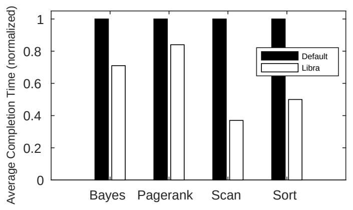

Figure 22: Average job completion time on a large cluster.

6.5 How DES helps DTS

Small jobs. We use 3 VMs for this experiment, launching 2 active executors and 1 idle executor. *Scan* (174 MB) and *Sort* are run concurrently, the latter intended to simulate a background application contending for resource. Figure 18 shows that Libra performance is generally optimal and stable, regardless of how many initial tasks are launched.

Large jobs. We then run *PageRank* with 1 GB, 5 GB, and 10 GB input sizes on 8 VMs, adding contention to half of the VMs using the Linux commands. Figure 19 shows that Libra as a whole outperforms DTS.

These figures show that DES is essential to DTS. This is because DTS requires executor capability to be comparable, which is not met in the presence of contention; DES restores the condition.

Libra vs combination of skewness and straggler mitigation. Libra effectively jointly considers issues from the job side and the machine side. Therefore, we compare it with a system naively combining the state-of-the-art skewness and straggler mitigation schemes, i.e., TinyTasks (without scheduling delay) combined with speculation (using twice the resource)

Figure 20 shows that, for the large jobs, Libra *with scheduling overhead* outperforms TinyTasks and speculation *without scheduling overhead* by 25% to 33%, and more if the scheduling overhead for the latter is included.

Table 2: Workload distribution on a large cluster

Bin	Tasks	Job Type	# of Jobs Run
1	1-10	Scan	85
2	11-50	Pagerank	4
3	51-150	Sort	8
4	>150	Bayes	3

6.6 Large-scale performance

We run a large-scale experiment on 100 `m4.2xlarge` VMs. The workload is synthesized based on existing workload characterization of the Facebook trace [16] as shown in Table 2, 85% of which are small jobs, and 15% large jobs, covering the typical types of the applications seen in the original trace. Note that we *cannot replay any existing cluster traces*, because Libra changes the task size during the run, whereas those traces are based on static task size settings. We generated a submission plan of 100 jobs whose inter-arrival times are 14 s, so there is a range of interleaving between jobs during the entire run.

Figure 21 plots the CDF of *Scan* completion times. Libra reduces the median job completion time by more than 60%, and reduces the tail job completion time by approximately 40%. For Libra, the 80th-percentile job completion time ranges from 25 s to 75 s, much smaller than the 40 s to 180 s for default Spark. In other words, Libra provides more stable performance across jobs.

Figure 22 plots the average job completion time for each type of jobs. The median job completion time for Libra is a third of that for the default Spark. The smallest improvement is still around 20%, suggesting that Libraworks well in large-scale clusters.

7 Related work

Job partitioning. A lot of efforts have been made on graph partitioning, typically partitioning the input graph statically at the beginning of the execution [15, 24] or dynamically [26, 31]. However, all these algorithms have to leverage the graph structure information, and many leverage specific properties of the graph algorithm, hence they are not suitable for more generic data analytics systems.

Naiad [39] includes a static, hash-based job partitioner. AdaptStream [17] dynamically adjusts batch sizes specifically for streaming workloads, mainly to prevent the data flow from blocking. Instead, Libra calculates the optimal task size dynamically regardless of the workload characteristics.

Dynamic task management. Various works [40, 36, 29, 9] essentially permit dynamically changing the DAG. Libra does not affect the DAG, only how each node in the DAG is executed. DES is related to *work stealing* proposed in high performance computing [13], although the trigger and work (task) migration mechanisms differ.

Performance profiling and tail reduction techniques. A few studies characterize the performance bottlenecks or outliers [18, 35, 25, 42]. The causes of the tails are variously attributed to skewness (job issue), stragglers (slow machines), and data locality issue, and many solutions have been proposed to address these individually (e.g., [34, 33, 30, 44, 51, 41] to mitigate skewness, and [10, 50, 48, 14, 47, 11, 43] to mitigate stragglers).

In contrast, Libra does not explicitly address skewness but employs proactive dynamic repartitioning of the job. The DES component can mitigate stragglers. Libra optimizes the overall job completion performance, but may reduce performance tails *as a side effect*.

Prediction techniques. Libra implicitly predicts job performance based on earlier part of the *same run* following a data-light, control theoretic approach, *assuming the U-curve behavior*. In comparison, other schemes (such as Wrangler [48] and Quasar [20]) adopt general machine learning techniques, requiring comprehensive resource usage data (data-heavy) from previous completed runs of the same job to make a one-time decision, *without assumptions about the application behavior*. Ernest [45] is the closest to our system by leveraging predictable structures in the job, though still following a machine learning approach. The control approach is orthogonal to the learning based approaches. Libra can incorporate learning techniques to make more accurate decisions at each adaptation step.

8 Conclusion

In this paper, we propose Libra, to auto-tune job partitioning in data analytics systems. The key observation is that there is an optimal task size per application (stage). This optimal varies by the application logic, and so we need to dynamically determine task sizes during run time. We can start with a small task size and gradually increase the size until the task processing rate starts to drop. Libra takes a control approach, collecting information and making decisions local to a job on the fly.

We implemented Libra on Spark and showed Libra could reduce job completion times by a significant margin compared to state-of-the-art alternatives.

References

[1] `http://hadoop.apache.org`.

[2] `https://aws.amazon.com/emr/`.

[3] `http://spark.apache.org/docs/latest/configuration.html`.

[4] `https://en.wikipedia.org/wiki/Exponential_smoothing`.

[5] `https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm`.

[6] `https://en.wikipedia.org/wiki/Support_vector_machine`.

[7] `https://spark.apache.org`.

[8] `https://github.com/intel-hadoop/HiBench`.

[9] `https://issues.apache.org/jira/browse/SPARK-9850`.

[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[11] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[12] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 4:1–4:15, New York, NY, USA, 2013. ACM.

[13] R. D. Blumofe, P. A. Lisiecki, et al. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, 1997.

[14] E. Bortnikov, A. Frank, E. Hillel, and S. Rao. Predicting execution bottlenecks in map-reduce clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[15] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[16] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, Aug. 2012.

[17] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA, 2014. ACM.

[18] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[20] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 127–144, 2014.

[21] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[23] Z. Gong, X. Gu, and J. Wilkes. PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25-29, 2010*, pages 9–16, 2010.

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[25] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 22:1–22:14, New York, NY, USA, 2011. ACM.

[26] J. Huang and D. J. Abadi. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment*, 9(7):540–551, 2016.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[28] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, GA, 2016. USENIX Association.

[29] Q. Ke, M. Isard, and Y. Yu. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 15–28, New York, NY, USA, 2013. ACM.

[30] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[31] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[32] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[33] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant Parallel Processing of Feature-extracting Scientific User-defined Functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM.

[34] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in Mapreduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[35] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[36] Y. Lu, A. Chowdhery, and S. Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 57–70, New York, NY, USA, 2016. ACM.

[37] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[38] M. Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[39] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[40] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[41] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 14–14, Berkeley, CA, USA, 2013. USENIX Association.

[42] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 293–307, 2015.

[43] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 379–392, New York, NY, USA, 2015. ACM.

[44] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM.

[45] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, Mar. 2016. USENIX Association.

[46] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang. Probabilistic performance modeling of virtualized resource allocation. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 99–108, New York, NY, USA, 2010. ACM.

[47] N. Xu, L. Chen, and B. Cui. LogGP: A Log-based Dynamic Graph Partitioning Method. *Proc. VLDB Endow.*, 7(14):1917–1928, Oct. 2014.

[48] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 26:1–26:14, New York, NY, USA, 2014. ACM.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[50] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[51] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing Data Shuffling in Data-parallel Computation by Understanding User-defined Functions. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 22–22, Berkeley, CA, USA, 2012. USENIX Association.