Scheduling Executors with Time-varying Resource Demands on Data-Parallel Computation Frameworks

Hu Yang¹, Su Lin²

Abstract

Efficiently scheduling execution instances of data-parallel computing frameworks, such as Spark and Dryad, on a multi-tenant environment is critical to applications’ performance and systems’ utilization. To this end, one has to avoid resource fragmentation and over-allocation so that both idleness and contention of resources can be minimized. To make effective scheduling decisions, a scheduler has to be informed of and exploit resource demands of individual execution instances, including both short-lived tasks and long-lived executors. The issue becomes particularly challenging when resource demands greatly vary over time within each instance. Prior studies often assume that a scheduling instance is either short lived or of gradually varying resource demands.

However, when in-memory computing platforms, such as Spark, become increasingly popular, the assumption no longer holds. The execution instance for scheduling becomes executor, which executes an entire application once it is scheduled. Usually it is not short lived. Its resource demands are significantly time-varying. To address the inefficacy of current cluster schedulers, we propose a scheduling approach, namely Prophet, which takes resource demand variation within each executor into the scheduling decision. It leverages the fact that execution of a data-parallel application is pre-defined by a DAG structure and resource demands at various DAG stages are highly predictable. With this knowledge, Prophet schedules executors to minimize resource fragmentation and over-allocation. To deal with unexpected resource contention, Prophet adaptively backs off selected task(s) to reduce the contention. We have implemented Prophet in Apache Yarn running Spark. We evaluated it on a 16-server cluster, using 10 categories of a total of 90 application benchmarks. Compared to Yarn’s default capacity and fair schedulers, Prophet reduces application make span by up to 39% and reduces their median completion time by 23%.

Key words: Spark; Dryad; Prophet.

I. INTRODUCTION

Applications running on today’s large-scale data-parallel processing frameworks, such as Apache Hadoop [1], Dryad [2] and Spark [3], usually have DAG (Directed Acyclic Graph) composed of stages in their execution durations. Each stage consists of a number of tasks conducting the same type of data processing. A task requires multiple resources for its running, including CPU, memory, as well as disk and network bandwidths. While tasks belonging to the same stage are of similar demands for each of the resources, those belonging to different stages can have very different demands for different resources. For example, in machine learning applications, such as K-Means and SVM (Support Vector Machine) [4], [5],

¹Airport Management College, Guangzhou Civil Aviation College, Guangzhou 510403, China. E-mail: huyangfox@163.com
²Department of Computer Science and Engineering, Guangdong Peizheng College, Guangzhou 510830, China. E-mail:sulincookie@163.com
tasks of their map stages are I/O- and CPU-intensive and tasks of the reduce stages are network-intensive. The frameworks, such as Hadoop and Spark, usually run on a resource management system. YARN[6] and Mesos[7] are current popular systems that are responsible for resource allocation and sharing. It is critical for task schedulers in a YARN-like system to efficiently schedule the tasks of vastly diverse multi-resource demands onto a cluster of servers, so that both applications’ execution time and the cluster’s throughput can be maximized.

Scheduling tasks with multi-resource demands onto servers of limited amount of resources (CPU, memory, disk, and network) is often formulated as a multidimensional bin packing problem. As long as these demands are known a priori or can be accurately estimated, this problem can be solved heuristically in a polynomial time [8]. A common technique used for this estimation is to profiling tasks by leveraging the fact that jobs of an application are recurring and they “repeat hourly (or daily) to do the same computation on newly arriving data.” [8].

Such a profiling strategy is not sufficient to fully address the issue by itself in practice, as the demands measured during a task’s run vary (sometimes dramatically). While it is known that a multidimensional bin packing problem is NP-hard and has to be solved with heuristics, it is almost impossible to accommodate time-varying demands into the model to efficiently produce an effective scheduling decision. A conservative alternative is to use peak usage of a resource to represent the varying demands of a task during running to prevent resource over-allocation [8], which occurs when aggregate demand from all running tasks exceeds available resources. It often leads to interference between tasks and serious performance degradation. However, this conservative approach generates risk of resource fragmentation, which occurs when resources are idle but tasks with demands on them that ready for scheduling cannot use them.

To achieve high scheduling efficiency, a scheduler has to simultaneously minimize fragmentation and over-allocation of resources [8]. When each application can have a large number of tasks and each task has a relatively short execution time, using peak demand may not create extremely large pockets of fragmentation in terms of wasted resource time. However, this becomes a serious issue with in-memory computing frameworks, such as Spark [3] and Storm [9], where scheduling units have long execution time with varying demands.

An application of a Spark-like in-memory computing framework, does not expose its tasks to the underlying resource management system, like YARN and Mesos. Instead, the concept of executor is introduced as the scheduling instance in these systems. Once executors of an application are launched on servers by the system’s scheduler, the application’s scheduler is responsible for scheduling its tasks to these pre-allocate executors. Specifically, an executor is usually a Java virtual machine (JVM) and tasks are threads running on the JVM. Each Spark application has a set of executors scheduled by the resource manager to different servers and they stay alive until all tasks of the application are completed. This two level scheduling is adopted for two reasons. One is to cache a subset of data in memory to enable in-memory reuse of data across tasks in an executor in a fault-tolerant manner. The other is to significantly reduce overhead of launching tasks, which is critical for in-memory computing. In contrast, in a Hadoop application each task runs on a dedicated JVM, which is scheduled by the system’s resource manager.

While there are two levels of scheduling for in-memory computing, the executors’ scheduling plays a more performance-critical role as it represents the resources allocation and sharing between applications. Recent work like Tetris [8] exploits the knowledge of future (peak) resource demands of tasks. However, it cannot be applied directly on executors’ scheduling. When an executor becomes the scheduling object, the
rationale made by existing schedulers based on peak resource usage to represent the object’s varying resource demand is less likely to be valid. An executor runs multiple batches of tasks belonging to different DAG stages may have (very) different resource demands. Therefore, using the peak demand to represent different demands of a resource during the lifetime of an executor for resource allocation can cause serious resource fragmentation (or wastage).

Additionally, for a smooth run of tasks in an executor without interference from other application executors, it might be desired to have all four major required resources (CPU, memory, disk, and network) pre-allocated or reserved. Users only need to pre-specify their resource demands on CPU (number of cores) and memory (size of memory) for an executor. As these demands usually represent the bottom line of a user’s requirement on quality of service, the requested resources are pre-reserved at the time of executor scheduling. However, network and disk resources are shared among executors on a server without isolation or reservation. They are more likely to incur over-allocation, and tend to cause disk seeks or network in cast that may significantly compromise system’s throughput. In addition, neither users nor current cluster managers [6],[7] would specify network and disk demands of executors, let alone consider their highly variable demands. This may lead to application performance degradation and poor resource efficiency.

To improve cluster efficiency and speed up individual applications’ performance for in-memory computation, we design an executor scheduler, namely Prophet, which can select an executor whose scheduling would result in the smallest amount of fragmentation and over-allocation of network and disk resources. With the knowledge of an executor’s future varying (peak) disk and network demands at any stage during its lifetime and of each stage’s start time and its duration, Prophet can estimate resource availability at any time frame in the near future and make an informed scheduling decision accordingly to minimize resource fragmentation and over-allocation. To deal with unexpected resource contention, Prophet selects task(s) in an executor to back off to adaptively ameliorate the contention. In summary, we make the following contributions in the paper.

• We identify a performance-critical issue about the executor scheduling on in-memory data parallel computing platforms. We show that without considering resource demand variation within an executor, one can hardly enable an effective scheduling. By showing stability and predictability of resource demands in an executor, we make it possible to take the dynamics on the resource demands into account.

• We design an online executor scheduler, named Prophet, that adopts a greedy approach by choosing the currently optimal executors in terms of expected resource fragmentation and over-allocation to dispatch. It also dynamically avoids severe resource contention and subsequent dramatic performance degradation due to unexpected over allocation with its task back off mechanism.

• We have implemented Prophet on YARN and Spark1.5 to support Spark and evaluated it on a 16-server cluster. Experiments show that Prophet can minimize resource fragmentation while avoiding over-allocation. It can substantially improve cluster resource utilization, minimize application make span, and speed up application completion time. Compared to Yarn’s default capacity and fair schedulers, Prophet reduces the make span of workloads in Spark Bench [4] by 39% and the median job completion time by 23%.

The rest of the paper is organized as follows. Section II describes motivation of the work and demonstrates predictability of resource demands in an executor. Section III describes design of the Prophet scheduling scheme. Section IV describes the implementation and evaluation of Prophet. Section V reviews the related work, and Section VI concludes the paper.
II. MOTIVATION AND BACKGROUND

A. Workload Analysis

To illustrate the potential efficiency loss due to resource fragmentation and over-allocation, we use four Spark benchmarks and their input data generators available in Spark-Bench [4], to reveal their executors’ resource demand variations. Among the four benchmarks, two (K-means and SVM) represent machine learning workloads, and the other two (Page Rank and SVD++) represent graph computation workloads. They are briefly described in the below.

- K-means is a machine learning workload clustering a set of data into K clusters.
- SVM (Support Vector Machine), is a machine learning classifier workload analyzing data and recognizing patterns of high dimensional feature spaces while efficiently conducting non-linear classifications.
- Page Rank is a graph computation workload ranking web-site pages and estimating their importance.
- SVD++ is a graph computation collaborative filtering workload improving the quality of recommendation system based on the users’ feedbacks.

Figures 1 and 2 show disk and network bandwidth demands of the four Spark benchmarks (Spark 1.5.0) on Hadoop Yarn 2.4.0, respectively. Each executor is exclusively run on a server of 24 cores, 32GB of
memory, three 7200 RPM disk drives, and 1Gbps NIC. It is obvious that for both disk and network usages the amount of requested bandwidth varies from almost 0 MB/s to around 300MB/s for disk or around 160MB/s for network. Their very low resource demands can stay for more than half of some executor’s lifetimes, such as for network usages of K-means and SVM, while their peak demands are still very high, such as around 160MB/s. Should the resources be allocated according to the peak demands, they would be significantly wasted due to the serious fragmentation. Even worse, starvation may occur on applications with both high peak network and disk demands as servers may not have available resources to meet both peak demands simultaneously (even though such an availability is not necessary). On the other hand, if they are not pre-allocated, multiple executors on the same server may simultaneously experience high demand on the same resource, causing resource over-allocation. This can lead to severe interference (disk seeks or network in cast) between the executors, which can sharply degrade applications’ performance.

It is necessary to take resource variation of executors into their scheduling decision so that both resource fragmentation and over-allocation can be minimized. This is a highly challenging issue considering that even scheduling objects of constant resource demands (e.g., using peak demands) can be NP-hard [8].

B. Predictability

Recent studies on large-scale data-parallel systems reveal that most applications in production clusters exhibit recurring execution behaviors with predictable future resource demands and mostly constant execution time in each DAG stage for given CPU cores and with sufficient memory [8], [12], [13], [14], [15]. Therefore, tasks’ statistics measured in their prior runs enable effective estimation. Specifically, “since tasks in a phase perform the same computation on different partitions of data, their resource use is statistically similar.” [8]. An offline or online profiling of tasks’ runs would provide a scheduler with knowledge on tasks’ resource demands. To illustrate this, in addition to the aforementioned four benchmarks, we select another six Spark benchmarks. Three of them (LR, Triangle Count, and Tera Sort) are from Spark Bench [4], and the other three (Word Count, Sort, and Grep) are from Big Data Bench [16]. They are described in the below.

- Logistic Regression (LR) is a machine learning classifier benchmark to predict continuous or categorical data.
- Triangle Count is a fundamental graph analytics counting number of triangles in a graph to detect spam or hidden structures in web pages.
- Tera Sort is a sorting benchmark using map/reduce to sort input data into a total order.
- Word Count reads Wikipedia text entries as input, and counts how often words occur.
- Sort is a benchmark designed for sorting words from a Wikipedia dataset.

For each of the ten benchmarks, we used 9 settings, including three CPU core numbers for each executor (one, three, and five) and three categories of input dataset sizes (small, medium, and large). The dataset sizes for each benchmark and category are shown in Table I. Each of the settings run five times with different input datasets of the same size. For each of the five runs in a dedicated cluster of 16 nodes, we collect each stage’s start time and peak disk/network bandwidths of an executor and compute their relative standard errors over the five runs. Figure 3 plots the errors with CDF (cumulative distribution function) curves. As shown, the relative errors are mostly smaller than 10%. Though contents of the input data sets have the potential of affecting executor’s behaviors, such as number of iterations to reach a convergence in machine learning applications, the impact is small. More importantly, each stage’s start time is very stable (with a 5% or smaller relative standard error).
TABLE I. Three categories of input dataset sizes for each of 10 benchmarks.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>SVM</th>
<th>KMeans</th>
<th>LR</th>
<th>Page Rank</th>
<th>SVD++</th>
<th>Triangle Count</th>
<th>Terasort</th>
<th>Word Count</th>
<th>Sort</th>
<th>Gre</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Input Dataset</td>
<td>38.3G</td>
<td>21.9G</td>
<td>37.1G</td>
<td>4.0G</td>
<td>365.6M</td>
<td>364.7M</td>
<td>37.3G</td>
<td>44G</td>
<td>44G</td>
<td>44G</td>
<td></td>
</tr>
<tr>
<td>Medium Input Dataset</td>
<td>19.2G</td>
<td>10.9G</td>
<td>18.5G</td>
<td>1.9G</td>
<td>163.3M</td>
<td>167.2M</td>
<td>18.6G</td>
<td>22G</td>
<td>22G</td>
<td>22G</td>
<td></td>
</tr>
<tr>
<td>Small Input Dataset</td>
<td>9.6G</td>
<td>5.5G</td>
<td>9.3G</td>
<td>933.1M</td>
<td>78.1M</td>
<td>86.5M</td>
<td>9.3G</td>
<td>11G</td>
<td>11G</td>
<td>11G</td>
<td></td>
</tr>
</tbody>
</table>

•**Grep** is a benchmark filtering and finding specified words from a Wikipedia dataset.

Figure 3. Relative standard errors of disk/network bandwidth and stage start time over the five runs of each of 10 benchmarks with different settings on CPU core and input size. Each run uses a different input dataset.

Because usually the same setting (CPU cores for each executor and input dataset size) remains in use for an application for an extended time period [14], [15], [13], profiling results about stage start time and peak resource demands of a run is sufficient for an executor scheduler to make an informed decision for its future runs. However, when an application uses a new setting that has not been profiled, we need a method to estimate the results. To this end, we adopt a supported vector machine (SVM) with linear regression technique. In particular, we feed results from 25 profiling runs covering representative settings into the machine to build a prediction model. The model then takes in a new setting (about CPU cores and dataset size) and produces its predicted stage start time and peak resource demands. Because changing CPU core count and input size usually does not lead to disruptive change of an executor’s behaviors, the model consistently provides high-quality estimations (mostly less than 10% errors).

III. DESIGN OF PROPHET

As an executor scheduler, in addition to its main objective of minimizing resource fragmentation and over-allocation, Prophet has two other objectives. One is fairness across applications, and the other is load balance across servers running applications. In the scheduling, all arriving applications will be placed into a waiting queue. When an application is submitted, its required reserved CPU, memory, and number of executors are specified by users. When there exist applications whose specified CPU and memory resource demands can be met by currently available resources in the cluster, Prophet greedily chooses one that would result in minimal fragmentation and over-allocation of network and disk resources for dispatching. Then the required number of executors are created on different servers. Note that for load balance across servers in an application’s execution, Prophet always creates the required number of executors at the time when the
application is scheduled. It does not create executors fewer than the required ones when resources are not sufficient. Otherwise, if the number of executors is allowed to increase, all newly created executors will request data from existing ones and cause them to become performance bottleneck. For fairness and starvation avoidance, Prophet chooses an application for scheduling from a subset of pending applications that have waited for the longest time (by default 50% of all pending ones). Each application is also assigned a deadline when it arrives at the queue. It will be scheduled immediately when its deadline is passed. The deadline can be assigned according to current average waiting time (e.g. three times of its average).

A. Prophet’s Scheduling Algorithm

Prophet’s scheduling algorithm is designed under the assumption that future resource demands of an executor, either one that is running or one that is candidate to be scheduled, is known in advance (or can be predicted). By knowing the total demands of executors currently running at a server, Prophet can compute how much the resource would be available in the near future. This is illustrated in Figure 4 for disk bandwidth of a server with two executors being scheduled on it. In the figure, each executor has two stages of different peak disk bandwidth demands. However, their combined effect leaves the available resource of four distinct values, or four resource availability stages. At this time we have two candidate applications’ executors for Prophet to decide which one to schedule, as shown in Figures 5(a) and (b), respectively. If only disk bandwidth is considered, Prophet needs to examine future fragmentation areas (FAs) and over-allocated areas (OAs) in Figure 5. FA or OA refers to the area between the two lines for available bandwidth and the demand in the

![Figure 4. Illustration of predicting available disk bandwidth. With known peak demands on disk bandwidth across stages of two executors (see (a) and (b)), the shaded area in (c) between their combined demand and the disk’s capacity represents the disk’s bandwidth to be available.](image)

![Figure 5. Illustration of how fragmentation area (FA) and over-allocation area (OA) of disk bandwidth are formed for two executors. For each executor (see (a) or (b)), the graph at the top shows its peak demands on disk bandwidth across stages, and the graph at the bottom shows the demand and available disk resource (shaded area computed in Figure 4) overlap with each other to form FAs, such as A1, A2, and A3, and OAs, such as B1 and B2.](image)
If available bandwidth is larger than the demand, it is FA, such as \( A_i \) (\( i = 1, 2, ..., 5 \)). Otherwise, it is OA, such as \( B_i \) (\( i = 1, 2, 3 \)). FA represents wasted resource and OA suggests resource contention and performance degradation. A good scheduler would simultaneously minimize the two areas. In this example, Prophet will schedule the executor shown in Figure 5(a), as it has much smaller aggregate FA/OA area than that in Figure 5(b). This example also indicates a scheduler that is unaware of future resource demands and availability might schedule the executor shown in Figure 5(b), leading to much worse performance. To formally describe the design of the scheduling algorithm,

we introduce a number of notations as shown in Table II. Note that in the notations, quantities about duration and times \( (t_{k,i}\^{\text{start}}, t_{k,i}\^{\text{end}}, T_{s,i}\^{\text{start}} \text{ and } T_{s,i}\^{\text{end}}) \) are not defined specifically for certain resource. Instead, they are specified according to change of stages for any resources.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_i^r )</td>
<td>Capacity of Resource ( r ) on Server ( i )</td>
</tr>
<tr>
<td>( P_{k,i}^r )</td>
<td>Peak demand of Resource ( r ) from Executor ( j ) at its Stage ( k )</td>
</tr>
<tr>
<td>( A_{i,s}^r )</td>
<td>Available Resource ( r ) of Server ( i ) at resource Stage ( s )</td>
</tr>
<tr>
<td>( t_{k,i}^{\text{start}}, t_{k,i}^{\text{end}} )</td>
<td>Start and end times of Stage ( k ) at Executor ( j )</td>
</tr>
<tr>
<td>( T_{s,i}^{\text{start}}, T_{s,i}^{\text{end}} )</td>
<td>Start and end times of resource Stage ( s ) at Server ( i )</td>
</tr>
</tbody>
</table>

To quantify fragmentation and over-allocation for candidate application’s executors, we may simply add FA or OA of an executor’s every stage, and consider the sum as the executor’s fragmentation score or over-allocation score, or F and O in short, respectively. However, for an executor of many stages, prediction on demands and resource availabilities at the earlier stages, or those closer to the current time, is usually more accurate than that on later stages, because the latter is more likely to be influenced by unaccounted noises. For this reason, we give earlier stages a higher weight. Specifically, if the executor has \( n \) stages, the weight for Stage \( i \) (\( i = 0, 1, ..., n-1 \)) is \( w_i = \frac{1}{n} \). Therefore, the two scores can be computed for Resource \( r \) as following.

For any \( P_{k,i}^r \leq A_{i,s}^r \),

\[
F_r = \sum_k \left( \sum_s \text{Formula}(s, k) \right) ;
\]  

(1)

For any \( P_{k,i}^r > A_{i,s}^r \),

\[
O_r = \sum_k \left( \sum_s \text{Formula}(s, k) \right) ;
\]  

(2)

If \( t_{k,i}^{\text{end}} \leq T_{s,i}^{\text{start}} \) or \( T_{s,i}^{\text{end}} \leq t_{k,i}^{\text{start}} \),

\[
\text{Formula}(s, k) = 0
\]

(3)

else if \( t_{k,i}^{\text{start}} \leq T_{s,i}^{\text{start}} \leq t_{k,i}^{\text{end}} \leq t_{k,i}^{\text{end}} \),

\[
\text{Formula}(s, k) = \left[ |P_{k,i}^r - A_{i,s}^r| \times (T_{s,i}^{\text{end}} - T_{s,i}^{\text{start}}) \right] \times w_k
\]  

(4)

else if \( t_{k,i}^{\text{start}} \leq T_{s,i}^{\text{start}} \leq t_{k,i}^{\text{end}} \leq T_{s,i}^{\text{end}} \),

\[
\text{Formula}(s, k) = \left[ |P_{k,i}^r - A_{i,s}^r| \times (T_{s,i}^{\text{end}} - t_{k,i}^{\text{start}}) \right] \times w_k
\]  

(5)

else if \( T_{s,i}^{\text{start}} \leq t_{k,i}^{\text{start}} \leq T_{s,i}^{\text{end}} \leq t_{k,i}^{\text{end}} \),

\[
\text{Formula}(s, k) = \left[ |P_{k,i}^r - A_{i,s}^r| \times (T_{s,i}^{\text{end}} - T_{s,i}^{\text{start}}) \right] \times w_k
\]  

(6)
In theory, to minimize both fragmentation and over-allocation in the selection of applications for scheduling, we might simply use the sum of the two scores as the metric for the selection. However, resource over-allocation can cause contention among executors and slow down all involved ones. More seriously, the slowdown may lead to more idleness (fragmentation) of other resources. To address the issue, we give $O$ a higher weight when computing the overall score.

$$\text{OverallScore}_v = (1 - \eta)O_v + \eta * E_v.$$  (8)

In our prototype, we set $\eta$ as 0.3 by default, which is experimentally determined to balance the risks of severe performance degradation and wastage of resources. We leave a comprehensive study of this parameter as future work. While for each resource (disk or network resources) Prophet can compute an overall score, for all resources it obtains a vector of overall scores for an application’s executor. To convert the vector into a one-dimensional quantity for comparison across candidate applications, we use the Euclidean norm of the vector. Accordingly Prophet selects an application whose executors have the smallest norm.

### B. Ameliorating Contention with Task Back off

While Prophet attempts to avoid expected over-allocations, there still can be unexpected ones or expected minor one turn out to be major over-allocations. As we have indicated in Section I, severe over-allocation leads to intensive interference. For disk and network, such an interference can cause their effective bandwidths to be much lower than their normal peak ones due to reasons such as random access and in cast, respectively. When interference essentially blocks tasks of an executor from moving forward, the executor’s reserved CPU cores and memory are also wasted. To address the issue, Prophet has an exception handling mechanism built in the Spark’s task scheduler. When it is observed that effective disk or network bandwidth is substantially lower than their peak one but the disk or network stays busy to serve requests at a server, a serious over-allocation is detected at the server. Prophet will examine the profiled resource demands of each executor on the server and identify ones that are most likely to overuse the contested resource. It then activates a back off mechanism by reducing number of tasks dispatched to the executors until the effective bandwidth approaches the peak one or the resource is not busy anymore. Note that the mechanism is enabled only temporarily, usually lasting for only a few task scheduling rounds, as an over action could compromise utilization of CPU and memory.
IV. CONCLUSION

Existing task schedulers are not suitable for scheduling executors with time-varying resource demands on an in-memory data-parallel computing platform, such as Spark. They suffer from serious over-allocation and fragmentation problems and can substantially compromise application performance and system resource utilization. Motivated by observations on recurring resource usage patterns in the platform, we propose a scheduling scheme, Prophet, to learn and leverage the patterns in the executors’ scheduling. In particular, Prophet can accurately predict resource availability at a server and varying demands from executors in the near future. It allows the demands to best match available resources. This will help with both application performance and system utilization. In addition, Prophet has a task back off mechanism to accommodate unexpected over-allocation to improve the system’s robustness.

We have implemented Prophet on Yarn and Spark. Extensive experiments with publicly available benchmarks show that Prophet can reduce make span by up to by 39% and median application completion time by 23%, compared to Yarn’s default capacity and fair schedulers.

REFERENCES