2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)

Sparker: Optimizing Spark for Heterogeneous Clusters

Nishank Garg
Department of CSE
Indian Institute of Technology, Madras
Email: nishankg@cse.iitm.ac.in

Dharanipragada Janakiram
Department of CSE
Indian Institute of Technology, Madras
Email: djram@iitm.ac.in

***Abstract*—Spark is an in-memory big data analytics framework which has replaced Hadoop as the de facto standard for processing big data in cloud platforms. These frameworks run on cloud platforms where heterogeneity is a common scenario. Heterogeneity gets introduced due to the failure, addition or upgradation of nodes in the cloud platforms. It can arise from various factors such as variation in the number of CPU cores, amount of memory, disk read/write latencies across the nodes, etc. These factors have a significant impact on the performance of Spark jobs. Spark supports execution of a job on equal-sized executors which can result in under allocation of resources in a heterogeneous cluster. Insufficient resources can severely degrade the performance of CPU and memory intensive applications like machine learning, graph processing, etc. Existing techniques use equal-sized executors which can degrade the performance of jobs in heterogeneous environments. In this paper, we propose Sparker, an efficient resource-aware optimization strategy for Spark in heterogeneous clusters. It overcomes the limitation of heterogeneity in terms of CPU and memory resources by modifying the size of the executor. The executors are re-sized based on the available resources of the node. We have modified Spark source code to incorporate executor re-sizing strategy. Experimental evaluation on SparkBench benchmark shows that our approach achieves a reduction of up to 46% in execution time.**

***Index Terms*—Heterogeneity, Executors, Apache Spark**

I. Introduction

MapReduce [1] is a popular programming model for distributed data processing. It has revolutionized the processing of big data on the cluster of commodity hardware. Hadoop [2] uses MapReduce model for processing and Hadoop Distributed File System (HDFS) [3] for storage of big data. Such a design approach is not efficient for iterative and interactive applications, as they perform repetitive accesses to the disk. Spark [4] resolves this problem by introducing in-memory data abstraction known as Resilient Distributed Datasets (RDDs) [5]. It outperforms Hadoop for these applications by storing the frequently accessed data in the memory. However, Spark's executors are designed for homogeneous clusters which can introduce significant challenges in cloud platforms and data centers where heterogeneity of resources is a common scenario.

Heterogeneity can be caused due to variation in read-/write latencies of disks, amount of memory and number of CPU cores available across the nodes. It can result from virtualization of resources in the cloud platforms [6] and differences in the generation of systems that get added up in the cluster over a period of time [7], [8].

Executors are distributed agents which are launched as Java Virtual Machine (JVM) instances to run tasks in parallel. The size[1] of an executor for a job is determined based on the smallest worker node in the cluster. However, in heterogeneous clusters, equal-sized executors may not result in optimal utilization of resources. For instance, consider three nodes (node 1, node 2 and node 3) in the cluster with 8 cores 16 GB memory, 6 cores 4 GB memory and 2 cores 2 GB memory respectively. The executor size is configured equal to smallest worker node i.e. 2 cores 2 GB memory. In this case, seven executors will be launched. Each executor will incur 300 MB of reserved memory [9]. Therefore, total reserved memory overhead would be 2 GB (approx) which can not be used for execution. Moreover, 8 GB of memory will remain unused in node 1 and two cores will be idle on node 2 as the resources are not a multiple of executor size. Hence, heterogeneity can severely impact the execution time of jobs which have high resource requirement such as machine learning, graph processing, etc. Also, the variation in the disk bandwidth (higher the disk bandwidth, lower the disk read/write latencies) across the nodes can result in overall reduced data read/write rate, affecting the execution time of the job.

Existing works [10]–[12] use homogeneous (or equal-sized) executors which is the minimum unit of resource assignment for execution of jobs. There exists a one-size-fits-all configuration for an executor in homogeneous clusters but no such configuration exists for executors in heterogeneous clusters. Other techniques [13], [14] focus on job scheduling and load balancing in MapReduce for heterogeneous clusters and do not consider resource assignment aspects of the job. Tula [15] is a balancer for Hadoop which balances the HDFS blocks across the nodes based on the variation in read/write latencies of the disks but does not consider variation in terms of compute and memory resources. Since existing approaches do not provide heterogeneous resource assignment, an efficient solution is needed for utilization of resources in heterogeneous

[1]We define the size of the executor as the number of CPU cores and amount of memory allocated to it.

978-1-5386-7899-2/18/$31.00 ©2018 IEEE
DOI 10.1109/CloudCom2018.2018.00017

clusters.

To address these challenges, we propose Sparker, a resource-aware optimization strategy for Spark in heterogeneous clusters. This will help in utilizing unprovisioned compute and memory resources of the cluster. The methodology of Sparker can be adapted to various JVM based frameworks such as Apache Flink [16], Presto [17], etc. We demonstrate the working of Sparker on iterative applications such as machine learning and graph processing as these applications exhibit a high demand for memory and CPU cores. Sparker first identifies the resources available on each node. If the resources allocated to a job significantly deviates from the available resources on the cluster, Sparker modifies the executor size. The effect of this change is reflected in the improved CPU and memory utilization of the cluster, resulting in the reduction of the execution time of the job.

The major contributions of this paper are:

- Analysis of the limitations of Spark's resource provisioning in heterogeneous clusters.
- Support for launching heterogeneous executors in Spark.
- Experimental evaluation and comparison of Sparker against Spark's resource provisioning.

The paper is organized as follows. In Section II, we discuss the related background on Apache Spark and its resource management. Section III presents the design of Sparker. Section IV describes the executor re-sizing algorithm. Section V empirically evaluates it. Finally, we conclude and indicate future directions in Section VI.

II. Background

A. Apache Spark

Spark is an in-memory distributed processing framework designed to overcome the limitations of Hadoop for iterative applications and interactive analytics. For each iteration of application, data is fetched, processed and written back to HDFS which causes a lot of disk read/write overhead. Spark resolves this problem by keeping the repetitively used data to be used repetitively in the memory thereby making it much faster than the Hadoop framework. This in-memory data abstraction is called as RDDs which is split across the cluster in the form of partitions.

Spark follows the master-slave architecture for distributed data processing. Master runs the driver program and slaves run the worker programs. Executors are Java processes which are launched inside workers and execute tasks parallelly to process the partitions. Each executor stores these partitions in the heap space allocated to its JVM. The fraction of partitions cached decides the performance of the application as the higher the partitions cached in memory, the lesser their access time.

Spark can take input from multiple sources like HDFS, HBase [18], Cassandra [19], etc. Although Spark is an in-memory framework, disks play a vital role in the performance of job [20] [21]. The disk operations are required during loading and storing of data. Disk accesses are also needed for reading the RDDs if they are not fully cached. The percentage of time spent on reading and writing big data to disk is higher than the processing time of data. Therefore, disk read/write latency is a bottleneck for job processing in Spark.

B. Executor Memory Management in Spark

Executors are JVM instances which are launched from the *buildCommand()* function defined in the *WorkerCommandBuilder* class inside the *core.src.main.scala.org.apache.spark.launcher* package. Each executor is assigned equal number of CPU cores and amount of memory using the *spark.executor.cores* and *spark.executor.memory* arguments respectively. These arguments can be passed at run-time or by setting them in the *spark-env.sh* configuration file. Executors are responsible for running tasks parallelly on the assigned CPU cores and memory. Executor memory [9] is divided into four regions: reserved memory, user memory, execution memory and storage memory. Spark memory consists of storage memory and execution memory. Reserved memory is set to 300MB per executor and can not be used by Spark. User memory is the memory pool which can be used for user needs like storing data structures. In the default case, it is allocated 40% of the total executor memory and remaining 60% is assigned as Spark memory. Storage memory is used for the purpose of caching the RDDs while execution memory is used to store intermediate buffers during shuffle operations like joins, groupby, etc. Spark memory, execution memory and storage memory can be represented in mathematical form as

$$SparkMemory = (JVM - n * RM) * 0.6 \tag{1}$$

$$ExecutionMemory = (JVM - n * RM) * 0.6 * (1 - sf) \tag{2}$$

$$StorageMemory = (JVM - n * RM) * 0.6 * sf \tag{3}$$

where JVM is the total executor memory, n is the number of executors, RM is the reserved memory and sf is the $spark.memory.storageFraction$ which is the fraction of Spark memory allocated for storage memory.

The above formula holds true till Spark 1.5. However, from Spark 1.6 unified memory management [22] was introduced to avoid the manual tuning of storage and execution memory. Unified memory management allows automatic tuning of storage and execution memory based on the type of the application. For instance, linear regression is an RDD bound application, therefore the Spark's memory manager assigns the higher percentage of Spark memory to storage memory region. Whereas sorting types of applications are given more execution memory than storage memory. Although unified memory management is a step in the right direction, it does not work well for all applications [23].

C. Tula

Tula [15] is a balancer for Hadoop which shows the role of variation in disk latencies on the performance of jobs. It balances the HDFS blocks based on the capability of individual nodes using the *datanode utilization (du)* metric. *du* is the product of *disk space utilization (dsu)* and *disk latency (dl)*. The disk read/write latencies are determined using message filters [24] which are wrappers around system calls and give controlled access to them. Latency aware balancing policy considers the capability of a node to be directly proportional to the disk read/write latency. Thus, it prevents nodes with different latencies to receive the same number of blocks. Tula minimizes the variation of *du* metric across the cluster by moving blocks from higher *du* nodes to lower *du* nodes. We use Tula to address heterogeneity in terms of disk read/write latencies by moving HDFS blocks.

D. Related Work

Resource Management. Resource-aware optimization has been adopted by many cluster managers such as Mesos [10], YARN [11], etc., to maximally utilize the cluster resources. They allow multiple frameworks to run simultaneously in the cluster. These managers lease containers in order to isolate executors of different frameworks and run executors inside these containers. Although containers can be expanded or shrunk, the executor size once configured can not be modified. This brings inefficiency in utilizing the resources in heterogeneous clusters. Also, Spark provides Dynamic Resource Allocation [25] mechanism by increasing and decreasing the number of executors based on the demand of the application. This approach will incur higher reserved memory overhead when the number of executors is increased. We adopt a better approach by re-sizing the existing executors. MEMTUNE [26] is a framework for dynamically changing the memory partitions based on the workload demand. [27] predicts the precise memory requirement of a job using regression analysis. They found out that different applications have different memory demands. Project Tungsten [28] aims at optimizing the memory management of Spark by moving it to off-heap management.

Scheduling in Heterogeneous Cluster. Multiple approaches [13] [14] have focused on improving the performance of jobs in a heterogeneous environment by considering load balancing and scheduling. Paragon [8] is a heterogeneous and interference aware scalable online data center scheduler. It classifies an application for different hardware platforms using the collaborative filtering technique. In [13], the authors propose a technique for better straggler management on heterogeneous clusters. TARAZU [14] goes further and argues that heterogeneity causes excessive network communication at the Map phase and load imbalance at Reducer side and proposes a method to schedule tasks based on the application and cluster configuration. HeteroSpark [29] incorporates GPU in Spark for improving the task parallelism. However, these works do not consider the resource provisioning aspects which is crucial for data-intensive applications. Our work can complement these works by allowing efficient utilization of cluster resources.

III. Sparker

Sparker is a resource-aware executor re-sizing strategy for Spark in a heterogeneous environment. The main objective of Sparker is to leverage the unprovisioned resources in the worker nodes so that executors can utilize these resources for caching the RDDs and increasing parallelism. Sparker finds the underutilized nodes by monitoring the size of launched executors. It utilizes unallocated resources of nodes by increasing the memory and CPU cores of the launched executor.

Performance of iterative applications like linear regression, k-means clustering, PageRank, connected components, etc., heavily depends on the size of the RDDs cached. The size of the RDDs cached will depend on the memory allocated to the job. Therefore, the performance of these applications will depend on the memory allocated to them.

Let ET_i denote the execution time of application i.

$$ET_i \propto 1/rdd\_cached \tag{4}$$

$$rdd\_size \propto dataset\_size \tag{5}$$

$$rdd\_cached \propto memory\_allocated \tag{6}$$

From equations (4) and (6),

$$ET_i \propto 1/memory\_allocated \tag{7}$$

where *rdd_cached* is the size of RDDs cached by the job, *rdd_size* is the total size of RDDs generated by the job and *memory_allocated* is the physical memory allocated to the job.

Application	Dataset size	RDD size
Linear Regression	110 GB	45.11 GB
Decision Tree	110 GB	26.28 GB
SVM	110 GB	40.75 GB
PageRank	4 GB	67.43 GB
Connected Component	4 GB	42.5 GB

Table I: Relation between dataset size and RDD size for different applications.

The size of the RDDs generated by different applications is shown in Table I. It shows that for the same size of the dataset, applications can have RDDs of different sizes. Moreover, the size of RDDs generated by graph applications is much larger than the RDDs generated by machine learning applications. Therefore, memory insufficiency will have the higher impact on graph applications.

Figure 1 shows the executor re-sizing strategy for improving resource utilization in a heterogeneous system. The under-utilized nodes are encircled in dotted red lines in Spark. The idle CPU cores and unused memory resources

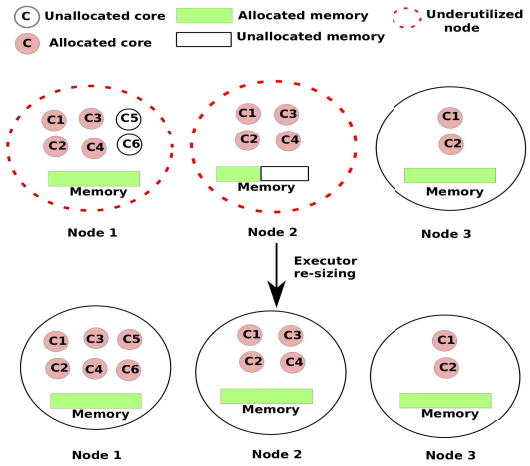

Figure 1: Executor re-sizing to utilize resources on under-utilized nodes

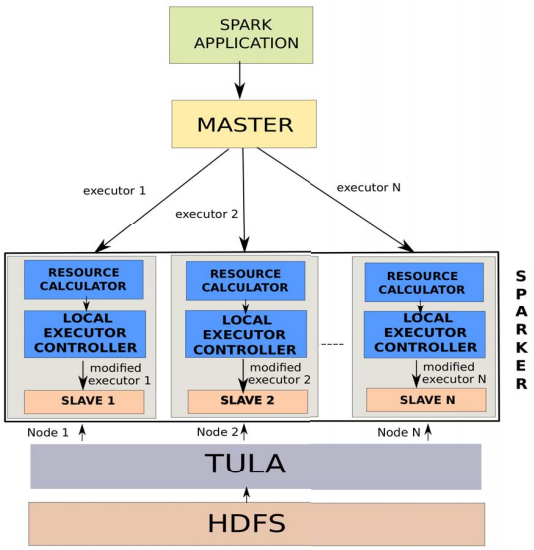

Figure 2: Sparker Architecture

are depicted in white. Figure 1 depicts that CPU cores are idle in node 1, memory is unused in node 2 while all cores and memory are allocated in node 3. The executor size is determined by the worker node with least resources. A job is executed with the executor size based on the resources available on node 3 as it has the least amount of resources. A total of five executors can be launched, two each on nodes 1 and 2 and one on node 3. Sparker decides to utilize the unallocated resources by re-sizing the executors. Since CPU cores and memory are unallocated on node 1 and node 2 respectively, Sparker re-size the executors in terms of cores and memory on these nodes. There is no scope of executor re-sizing in node 3, hence it is left unchanged.

A. Architecture

Figure 2 shows the architecture of Sparker. It consists of two main distributed components, *Resource Calculator* and *Local Executor Controller* which are executed on each node. *Resource Calculator* computes the available CPU and memory resources on each node. It communicates this information to the *Local Executor Controller*. *Local Executor Controller* performs the core idea of Sparker by modifying the size of the executors based on the data provided by the *Resource Calculator*.

We use Tula for balancing the HDFS blocks based on the variations in read/write latencies of disks. Using Tula, reading and writing of data from/to HDFS become faster because of the higher number of blocks placed on the disks with lower latencies.

After the executors are launched by the Spark framework, the decision about re-sizing of the executor is taken by the *Local Executor Controller* running on each node. The Controller analyzes the information and takes a decision to re-size the executor using the data provided by the *Resource Calculator*. If re-sizing is to be done, it assigns the executors with maximum available resources on that node. Therefore, the idle cores and unused memory will be used for increasing task parallelism and caching of RDDs respectively.

IV. Heterogeneity Aware Executor Re-sizing

Sparker optimizes the resource utilization of the cluster based on the size of the executors launched. Sparker can be enabled from the *spark-env.sh* configuration file. Once enabled, it checks the need for re-sizing the executors. If re-sizing is required, it modifies the size of the executor by increasing the CPU cores and allocated memory.

A. Implementation

We have implemented Sparker in Spark version 2.2.0 to support executor re-sizing for utilizing the unallocated resources in heterogeneous clusters.

1) Modifying the Executor size : Executors are the Java processes which run on worker nodes and their responsibility is to execute tasks in parallel. They are launched in Spark using the following command.

> *JAVA_HOME/bin/java -cp path-of-jars -Xmx<Memory> --driver-url <driverUrl> --executor-id <executorId> --hostname <hostname> --cores <cores> --app-id <appid> --worker-url <workerUrl>*

CPU cores and memory are configured in this command using the *--core* and *--Xmx* arguments. The <cores> can take positive integer value like 2, 4, 8, etc. The <Memory> can take values in MBs or GBs, such as 1024m or 1g. The number of cores and memory allocated per executor are

defined before submitting the job. After submitting the job, the master creates the executor launch commands.

The activation of Sparker is described in Algorithm 1. The capability of heterogeneous executors in Spark can be enabled by adding an environment variable 'SPARK_HETERO_EXECUTOR' in the *spark-env.sh* configuration file. This variable is checked just before the launch of the executor. This condition is introduced in *WorkerCommandBuilder.scala* in the package *core.src.main.scala.org.apache.spark.launcher*. If the variable is enabled, *Local Executor Controller* is called with the initial executor command. Sparker will intercept the executor launch command before it is launched on the worker. Algorithm 2 highlights the steps used by *Local Executor Controller* for re-sizing of executors in heterogeneous settings. It takes the executor launch command as input and receives information about available CPU cores and memory from the *Resource Calculator*. In the first step, *Resource Calculator* is invoked. The details of its working are described in Algorithm 3. It calculates the available memory and cores in the system using the *meminfo* and *cpuinfo* files available in *proc* [30] filesystem in Linux. This information is used by *Local Executor Controller* for modifying the executor size. It parses the executor launch command for the value of CPU cores and memory. If the assigned value for cores and memory is found to be lesser than the available cores and memory on the node, it modifies the executor command with maximum available cores and memory. Instead of allocating all the memory to the executor, we leave a certain amount of memory for operating system (OS) overheads. At the last step of the algorithm, modified executor command is launched.

Algorithm 1: Activate Sparker

```
Input : Spark Configuration file spark-env.sh,
        Initial executor command executor_cmd
var enable = getenv("SPARK_HETERO_EXECUTOR")
if enable == TRUE
 then
    LocalExecutorController (executor_cmd)
end
```

Algorithm 2: Local Executor Controller

```
Input : Executor launch command executor_cmd
Output: Modified Executor launch command
avail_core, avail_memory = Resource Calculator()
for each argument args ∈ executor_cmd do
    if args == -- cores
     then
        Replace the args with avail_cores in
         executor_cmd
    end
end
for each argument args ∈ executor_cmd do
    if args == substring(Xmx)
     then
        Replace the args with avail_memory in
         executor_cmd
    end
end
Launch modified executor_cmd
```

Algorithm 3: Resource Calculator

```
Input : cpuinfo and meminfo files from /proc directory
Output: Available Memory and Available Cores
avail_memory ←retrieve the total memory from
  meminfo file
avail_cores ←retrieve the number of CPU cores from
  cpuinfo file
return avail_memory and avail_cores
```

V. Evaluation

A. Workloads

We have used the SparkBench [31] benchmark suite for evaluating Sparker. This benchmark suite includes a variety of applications such as

- **Graph**. PageRank, Connected Component, etc.
- **Machine Learning**. Linear Regression, Support Vector Machine (SVM), Decision Tree, etc.
- **Clustering**. k-means

Experiments are done in such a way to cover applications from each category mentioned above. Realistic datasets are generated using the generation utility provided by the benchmark.

B. Environment

The experiments are performed on a 15-node heterogeneous cluster with standalone deploy mode of Spark. Each node is a VM and runs on the BOSS MOOL [32] operating system which uses object oriented abstractions to reduce coupling and improve maintainability in the Linux kernel. BOSS MOOL is used to incorporate Tula with Sparker. However, our strategy is generic enough to work on any OS. The number of cores in the cluster range from 2 to 8 and RAM size varies between 4 GB and 32 GB. The disk latencies of the nodes have been normalized between 0 to 1. HDFS Block size is kept at 128MB. Replication factor is set as 1 to make the evaluation of Tula prominent. Table II shows the configuration of the nodes.

C. Results

We study the effectiveness of Sparker in leveraging the unallocated resources of nodes for different workloads in a heterogeneous cluster.

1) Overall Performance of Sparker: We evaluate the performance of Sparker on the four workloads selected from the SparkBench suite. Table III give details about the datasets used for the experimentation. Among these workloads, PageRank generates maximum size of RDDs (Table I). All the workloads have RDDs size lesser than the available

Node#	Cores	Memory(GB)	Average Disk latency
1	8	32	0.028
2	4	8	0.028
3	6	8	0.028
4	2	4	0.116
5	6	5	0.058
6	4	12	0.028
7	6	4	0.116
8	2	4	0.116
9	4	6	0.058
10	4	6	0.058
11	4	5	0.058
12	4	8	0.028
13	2	4	0.116
14	2	4	0.106
15	2	4	0.058

Table II: Details of Nodes

Application	**Dataset**	**Records**	**Dimensions**
PageRank	4 GB	2M	-
Linear Regression	110 GB	15M	400
SVM	110 GB	15M	400
k-means	110 GB	15M	400

Table III: Datasets

RDDs storing capacity of cluster except for PageRank. Figure 3 shows the runtime of different applications under three scenarios: Sparker, Sparker with Tula and default Spark. We observe that Sparker outperforms default Spark for all applications by utilizing unallocated CPU cores and memory.

The percentage improvement varies across the application because of the size of RDDs cached by them. Linear regression shows a maximum reduction in execution time (46%) as it increases the percentage of caching by 26% whereas PageRank gives least reduction (21.2%) as it increases the percentage of caching by only 7%. An average performance improvement of 32.5% over default Spark is obtained across the studied workloads. Despite having equal dataset size, linear regression performs better than SVM. This happens due to the fact that the increase in the percentage of RDDs cached by linear regression (26%) is higher than that of SVM (23%).

Incorporating Tula with Sparker further improves the performance because all the three aspects of heterogeneity i.e., read/write latencies of the disks, amount of memory and number of CPU cores are considered. With Tula, the overall rate of loading and storing data from/to HDFS is reduced since it places a higher number of HDFS blocks on the nodes with lower disk read/write latencies. We observed a maximum reduction in execution time of 53% by including Tula with Sparker for linear regression. An average reduction of 7% in execution time is observed across the studied workloads by incorporating Tula with Sparker. In the experiments, we have used 4 GB dataset (32 HDFS blocks) for PageRank (higher dataset size than 4 GB results in failure of the job because of out of memory error) compared to 110 GB dataset (880 HDFS blocks) for others. Therefore, the reduction in execution time by Tula with Sparker is higher in linear regression, k-means and SVM as compared to PageRank because the impact of Tula is significant when the number of HDFS blocks are higher.

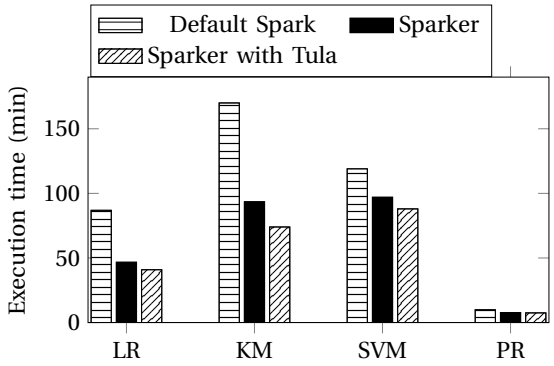

Figure 3: Execution time of different workloads under Sparker, Sparker with Tula and default Spark. LR: *Linear Regression*, KM: *k-means*, SVM: *Support Vector Machine*, PR: *PageRank*.

2) Increased RDDs Caching: We have collected the information about the size of RDDs cached by each application during its execution. Figure 4 depicts the higher percentage of RDDs cached with Sparker as compared to the default Spark for different workloads. This happens because default Spark leaves a significant amount of unused memory due to the use of equal-sized executor. However, Sparker launches heterogeneous executors and is able to utilize this unused memory. We achieved a maximum of 26% increase in RDD caching for linear regression.

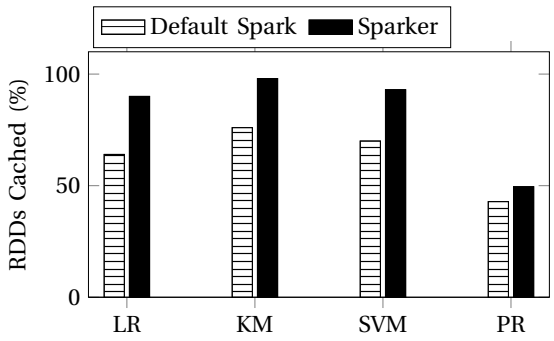

Figure 4: Comparison of percentage of RDDs cached between default Spark and Sparker

3) Improved CPU and Memory utilization: Executor resizing by Sparker results in better resource utilization of the cluster. Figure 5 shows the CPU usage for linear regression

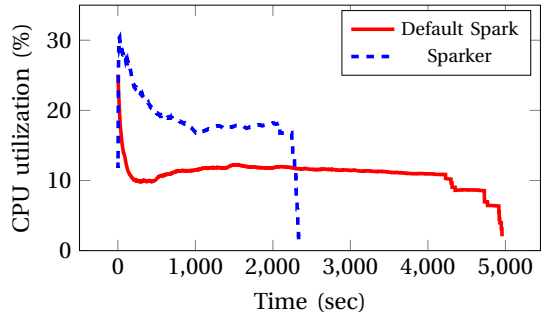

Figure 5: CPU utilization of Linear Regression (110 GB)

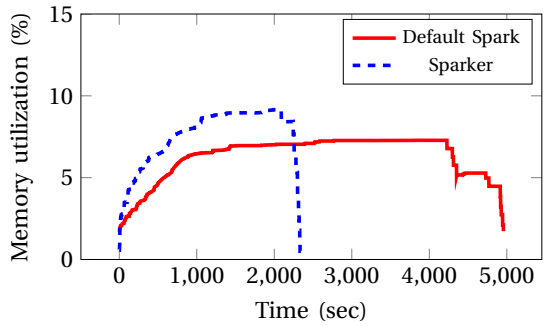

Figure 6: Memory utilization of Linear Regression (110 GB)

for 110 GB dataset. CPU utilization is higher with Sparker than the default Spark during the complete execution of the job. This is achieved by leveraging the unallocated CPU cores to improve the task parallelism. The extra number of cores that is utilized can be calculated as

$$\Delta cores = \sum_i cores_{Sparker_i} - \sum_j cores_{Default_j} \quad (8)$$

where $cores_{Sparker_i}$ is the number of cores allocated to the executor i in Sparker and $cores_{Default_j}$ is the number of cores allocated to the executor j in default Spark.

Figure 6 shows the memory usage of linear regression. It shows that the memory usage is greater than default case throughout its execution. The extra memory allocated to the executors is indicated below

$$\Delta memory = \sum_i memory_{Sparker_i} - \sum_j memory_{Default_j} \quad (9)$$

where $memory_{Sparker_i}$ is the executor memory assigned to executor i in Sparker and $memory_{Default_j}$ is the executor memory assigned to executor j in default Spark.

4) Impact of dataset size on performance of Sparker: Figure 7 shows the impact of varying the dataset size on the performance of linear regression. Increasing the dataset size will increase the RDD size (equation 5). Performance improvement is not observed with the smaller dataset (e.g. 10 GB) because RDDs are fully cached. Sparker shows the improvement in performance with the increase in dataset

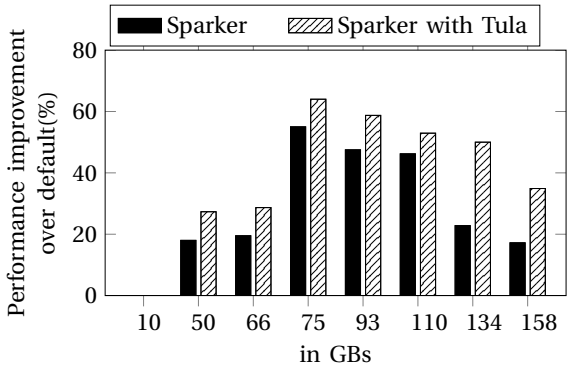

Figure 7: Impact of dataset size on the performance of Linear Regression.

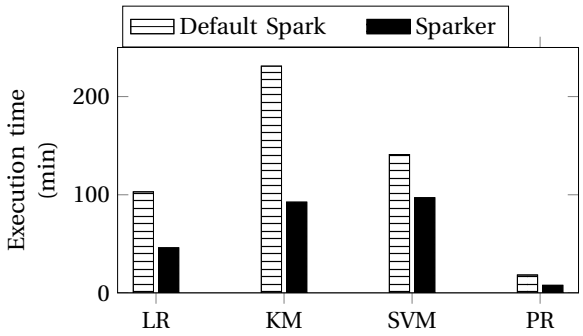

Figure 8: Comparing performance of default Spark (with executor size as 1 core and 1 GB memory) and Sparker.

size since it re-sizes the executor to cache more RDDs, which are otherwise spilled to the disk in default Spark. We observed that Sparker is able to fully cache the RDDs up to 75 GB dataset. However, beyond the 75 GB dataset, RDDs generated exceeds the RDDs storing capacity of the cluster. In such scenarios, the performance improvement with Sparker is relatively lower than the scenarios where Sparker is able to fully cache the RDDs. The reduction in performance improvement after 75 GB dataset occurs because of the disk overhead caused due to the spilling of RDDs to the disk. Moreover, combining Tula with Sparker helps in improving the performance in such scenarios by placing the higher number of HDFS blocks on disks with lower latencies.

5) Impact of many small sized-executors on performance of job: In order to fully allocate all the resources on a heterogeneous cluster, we could launch many small executors with the size of 1 GB memory and 1 CPU core. Figure 8 shows the comparison between default Spark (with executor size as 1 core 1 GB) and Sparker. We could launch a maximum of 52 executors based on the cluster configuration. We observed that Sparker has much lesser execution time than using many small sized-executors. This is because Spark reserves 300MB of executor memory per

executor. Therefore, with many executors, this overhead will increase and memory available per executor for caching is reduced.

D. Discussion and Limitations

Experiments have been done considering a single job. However, our approach can also be applied to multi-job and multi-tenant environments. But in that case, we need to consider multiple other factors such as QoS requirements, the fairness of resource distribution among jobs, etc. Moreover, *Resource Calculator* computes the maximum resources available on a node but not the optimal resources. Sparker only has a local view of the heterogeneous cluster. A global view with optimal resource requirement of job can help in better management and utilization of resources.

VI. Conclusion and Future Work

Distributed big data processing frameworks such as Spark do not efficiently utilize the resources on the heterogeneous clusters. Existing works consider equal-sized executors for processing of jobs which may result in unallocated CPU and memory resources on the heterogeneous clusters. The inadequacy of these resources can significantly affect the performance of memory and CPU bound jobs.

In this paper, we propose Sparker, a resource efficient strategy for Spark to improve the performance of jobs in heterogeneous clusters. Sparker uses the unprovisioned resources by launching unequal-sized executors in a heterogeneous environment. We have evaluated Sparker on a 15-node cluster of heterogeneous systems. We observed that Sparker outperforms default Spark for machine learning and graph applications. Integrating Tula with Sparker further improves the performance. Experimental results show an average improvement of 32.5% with Sparker and 39.5% by including Tula with Sparker. In the future, we plan to estimate the optimal memory requirements of jobs and extend our work to multi-tenant environments.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Apache Hadoop," 2009, http://hadoop.apache.org; last accessed: June 2018.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *2010 IEEE 26th symposium on Mass storage systems and technologies (MSST)*. IEEE, 2010, pp. 1–10.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[5] 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*. USENIX Association, 2012, pp. 2–2.

[6] "Amazon EC2," http://aws.amazon.com/ec2/; Accessed: April 2018.

[7] L. A. Barroso, "Warehouse-scale computing: entering the teenage decade," 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2019527

[8] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[9] "Spark memory management," https://spark.apache.org/docs/latest/tuning.html;lastaccessed:May2018.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[11] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[12] "Spark standalone mode," https://spark.apache.org/docs/latest/spark-standalone.html; last accessed: June 2018.

[13] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments." in *Osdi*, vol. 8, no. 4, 2008, p. 7.

[14] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: Optimizing MapReduce on Heterogeneous Clusters," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 61–74.

[15] J. Dharanipragada, S. Padala, B. Kammili, and V. Kumar, "Tula: A disk latency aware balancing and block placement strategy for Hadoop," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 2853–2858.

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[17] A. Hadoop, "Distributed SQL query engine for big data," https://prestodb.io; last accessed: April 2018.

[18] "Apache HBase," https://hbase.apache.org/; last accessed: April 2018.

[19] "Apache cassandra project," http://cassandra.apache.org/; last accessed: April 2018.

[20] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. Spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[21] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. A. Mckee, Z. Jia, and N. Sun, "Understanding the behavior of in-memory computing workloads," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 22–30.

[22] "Consolidate storage and execution memory management," https://issues.apache.org/jira/browse/SPARK-10000; last accessed: June 2018.

[23] "Mayuresh Kunjir, Yuzhang Han, and Shivnath Babu. 2016. Where does Memory Go?: Study of Memory Management in JVM-based Data Analytics," https://pdfs.semanticscholar.org/8590/b5d66e0dc429578cf6ac64b8abda6a125701.pdf.

[24] S. Nadella and D. Janakiram, "Message filters for hardening the Linux kernel," *Software: Practice and Experience*, vol. 41, no. 1, pp. 51–62, 2011.

[25] "Dynamic resource allocation," https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation;lastaccessed: April,2018.

[26] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "MEMTUNE: Dynamic memory management for in-memory data analytic platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 383–392.

[27] V. S. Marco, B. Taylor, B. Porter, and Z. Wang, "Improving Spark application throughput via memory aware task co-location: a mixture of experts approach," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 95–108.

[28] "Project Tungsten," https://issues.apache.org/jira/browse/SPARK-7075; last accessed: April 2018.

[29] P. Li, Y. Luo, N. Zhang, and Y. Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in *IEEE International Conference on Networking, Architecture and Storage (NAS), 2015*. IEEE, 2015, pp. 347–348.

[30] Linux, "/proc," http://man7.org/linux/man-pages/man5/proc.5.html; lastaccessed:"April,2018".

[31] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015, p. 53.

[32] "BOSS MOOL," http://dos.iitm.ac.in/projects/bossmool/bossmool.html; last accessed: June 2018.