

A Heterogeneity-Aware Task Scheduler for Spark

Luna Xu*, Ali R. Butta*, Seung-Hwan Lim[†], Ramakrishnan Kannan[†]

*Virginia Tech, [†]Oak Ridge National Laboratory

*{xuluna, butta}@cs.vt.edu, [†]{lims1, kannanr}@ornl.gov

Abstract—Big data processing systems such as Spark are employed in an increasing number of diverse applications—such as machine learning, graph computation, and scientific computing—each with dynamic and different resource needs. These applications increasingly run on heterogeneous hardware, e.g., with out-of-core accelerators. However, big data platforms do not factor in the multi-dimensional heterogeneity of applications and hardware. This leads to a fundamental mismatch between the application and hardware characteristics, and the resource scheduling adopted in big data platforms. For example, Hadoop and Spark consider only data locality when assigning tasks to nodes, and typically disregard the hardware capabilities and suitability to specific application requirements.

In this paper, we present RUPAM, a heterogeneity-aware task scheduling system for big data platforms, which considers both task-level resource characteristics and underlying hardware characteristics, as well as preserves data locality. RUPAM adopts a simple yet effective heuristic to decide the dominant scheduling factor (e.g., CPU, memory, or I/O), given a task in a particular stage. Our experiments show that RUPAM is able to improve the performance of representative applications by up to 62.3% compared to the standard Spark scheduler.

I. INTRODUCTION

Modern computer clusters that support big data platforms such as Spark [40] and Hadoop [2] are increasingly heterogeneous. The heterogeneity can arise from nodes comprising out-of-core accelerators, e.g., FPGAs, or staged upgrades resulting in nodes with varying performance and capabilities. Similarly, the resource needs of today’s applications are also heterogeneous and dynamic. Modern applications in machine learning, data analysis, graph analysis, etc., can have varying resource requirements during the application lifetime. For instance, a machine learning job may be I/O-bound and need I/O resources for input data processing in the initial stage, and then be memory- and compute-bound during the later processing stages. However, big data platforms, e.g., Spark, are typically oblivious of the dynamic resource demands of applications and the underlying heterogeneity, as the unit of per-task resource allocation is a homogeneous container (i.e., the abstraction does not capture heterogeneity of resources such as CPU cores, RAM, and disk I/O). This fundamental

mismatch between the high level software platforms and the underlying hardware results in degraded performance, and wastage of resources due to inability to efficiently utilize different resources.

We focus on the Spark scheduler in this paper, as Spark has become the de facto standard for big data processing platforms. Similar to most task schedulers in the MapReduce framework, the Spark scheduler mainly considers data locality for scheduling, and does not differentiate between the various resource capabilities, e.g., CPU power between the nodes, assuming them to be uniform, and is not aware of other resources that a node may have such as GPUs and SSDs. Varying resource demands within an application are also not captured. While, some external resource managers such as YARN [33] and Mesos [17] have begun to support heterogeneous clusters, these external managers are not aware of the internal heterogeneity of tasks within applications, and rely entirely on applications to request the right resources for further second-level (i.e., task-level) scheduling. This needlessly burdens the application developer. Other approaches either require historical data for periodical jobs [18] or require job profiling [9], [11], [35]. Moreover, they do not count the emerging accelerators and storage devices. Note that this paper focuses on internal task-level scheduling, which is orthogonal to aforementioned job-level resource managers.

Extant approaches [3], [16], [36] often make the assumption that tasks perform generic computations, and tasks in the same Map/Reduce stage would have same resource consumption patterns. As a result, such approaches focus on general purpose CPU architectures and often optimize for a dominant resource bottleneck for tasks in a Map/Reduce stage, regardless of the differences among tasks in a single stage, as well as the tasks that may benefit from special devices such as GPUs [37]. A recent study [23] shows that a task in Spark requires multiple resources when executing, and the completion time of a stage depends on the time spent on each resource. Consequently, a scheduler should consider the heterogeneity of the resource consumption characteristics of tasks in an environment with heterogeneous hardware, as disregarding such factors leads to suboptimal scheduling and overall inefficient use of resources.

In this paper, we address the above problems and propose RUPAM, a heterogeneity-aware task scheduler for distributed data processing frameworks. RUPAM considers both heterogeneity in the underlying resources as well as the various resource usage characteristics of each task in each stage of an application. RUPAM manages the life cycle of all tasks

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>).

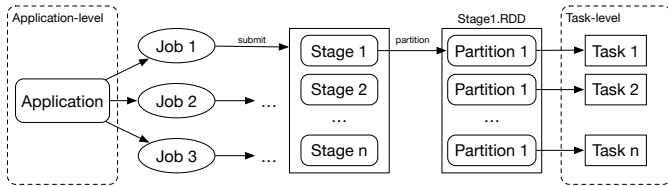


Fig. 1: Spark application architecture.

inside a job. To this end, RUPAM uses a self-adaptable heuristic that does not sacrifice data locality, to efficiently match a machine with the right task that can benefit most from the machine’s resources in a dynamic fashion. The heuristic coupled with real-time resource utilization, enables RUPAM to avoid resource contention and overlapping tasks with different resource demands, and thus yields high overall performance.

Specifically, this paper makes the following contributions.

- 1) We present a motivational study to demonstrate the diverse resource utilization and dynamic characteristics of tasks in Spark applications and show how the current Spark task scheduler is unable to properly handle heterogeneity in resources and application demands.
- 2) We design a self-adaptable heuristic based algorithm to automatically schedule tasks based on task characterization, cluster node capabilities, runtime resource utilization, and data locality.
- 3) We implement RUPAM atop Spark and evaluate RUPAM using SparkBench [21]. Results show that compared to the current Spark task scheduler, RUPAM reduces workload execution time by up to $2.5\times$. The performance increases (up to $3.4\times$ in our tests) with more application iterations (common in emerging deep learning applications).

II. BACKGROUND AND MOTIVATION

In this section, we first discuss the standard scheduling process of Spark. Next, we present an experimental study to motivate the need for RUPAM.

A. Spark scheduling

As shown in Figure 1, a typical Spark application comprises multiple “jobs” triggered by various actions on Spark Resilient Distributed Dataset (RDDs). A job is further divided into multiple “stages” differentiated by shuffle dependencies, which perform the data transformation operations for an RDD. The “tasks” within a stage perform the same operation on different data partitions. Here, two levels of scheduling are performed for a Spark application, namely, application level scheduling that provisions nodes in a cluster for an application, and task level scheduling.

Application scheduling is done by cluster managers, either the internal Spark standalone cluster manager or external managers such as Mesos and YARN. The cluster manager provisions a subset of cluster nodes for an application, which Spark then uses to run jobs and tasks. Spark and major resource managers provision resources into abstract units of number of CPU cores and sizes of memory. However, other resources

of a node, e.g., storage setup, network configuration, accelerators, etc. are not considered in the scheduling decisions. Mesos and YARN do provide support for accelerators, e.g., GPUs, but that too is limited and requires user configuration and labeling. Moreover, such configuration-based approaches are static throughout the application life cycle and unable to capture the dynamic application requirements. The main reason for this limitation is that the resource managers are not aware of the application task characteristics, and programmatic approaches need to be employed to negotiate and renegotiate between application and resource managers to dynamically require resources only when needed.

Once the resources are acquired from the resource manager, task-level scheduling is performed to assign the tasks to each executor running on a node. Current Spark task schedulers assign one task per a CPU core. Given a node, as long as it has cores available, the scheduler finds a task that has data residing on that node, and schedules the task to the node. Thus, only data locality is considered, while other factors are ignored. For example, a node with available cores may not be suitable for running more tasks say because it does not have enough memory left, and a new task would fail with an out of memory error. Existing heterogeneity-aware schedulers only focus on the heterogeneity of the underlying nodes, and often assume that all tasks in the same Map/Reduce stage have the same characteristics, which as we show does not always hold true.

B. Motivational study

To show that the assumptions in current Spark scheduler are not always true and can lead to inefficiencies, we performed several illustrative experiments with Spark. We use a simple 2-node setup, each node having 16 CPU cores and 48 GB memory. However, we configured the nodes with different CPU frequencies and network throughputs to emulate a heterogeneous environment. We explore more heterogeneous resources including memory, storage, and accelerators in Section IV. We configured node-1 to have 1.6 GHz CPU frequency and 10 GbE network speed, and node-2 to have 2.4 GHz CPU frequency and 1 GbE network speed. We set up the latest version of Spark 2.2 with one master and two workers.

1) *Resource utilization of different stages in a single application:* In our first test, we study the dynamic demands for resources during an application’s execution. Here we employ a crucial kernel used in numerous machine learning applications [37], [42]–[44], matrix multiplication, to make our case. We multiply two $4K \times 4K$ matrices as input and monitor the resource utilization during the execution. Figure 2 shows the CPU and memory utilization, network utilization for both inbound and outbound traffics, and disk utilization for both read and write. We observe that memory utilization remains high with an initial slope and a slight reduction in the final stages. In contrast, CPU usage is high for only the last stages where the actual multiplication happens, and shows a spike in the beginning data processing stage. Network utilization shows spikes in both beginning and final stages due to the

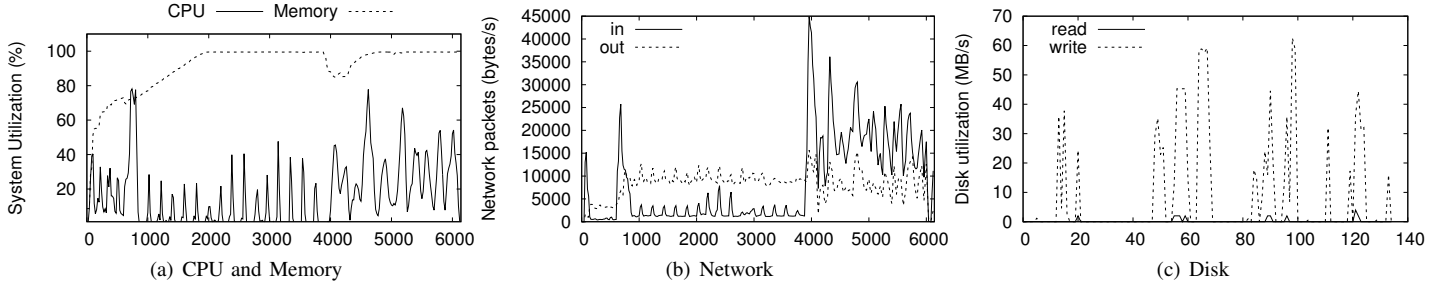


Fig. 2: System resource utilization under $4K \times 4K$ matrix multiplication. X-axis represents relative timestamp (trimmed for Disk performance).

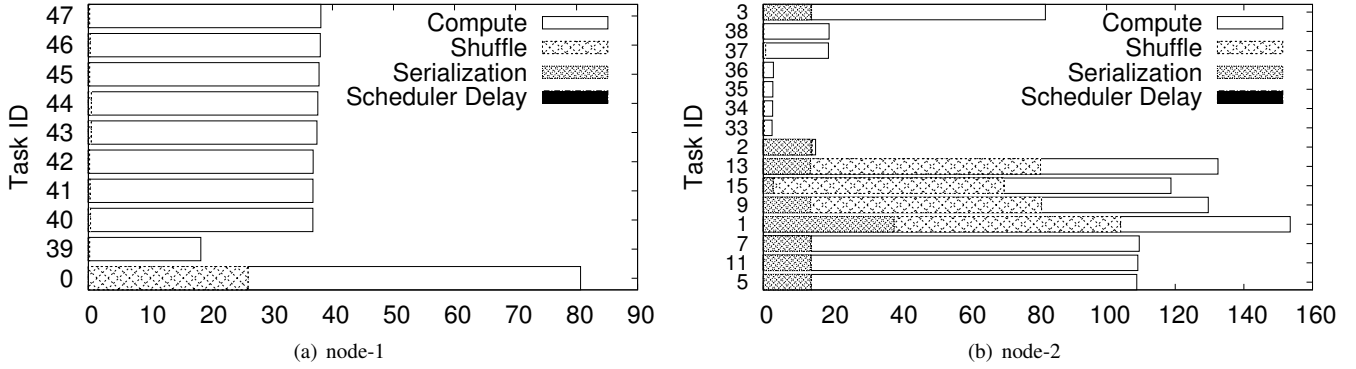


Fig. 3: Task distribution and execution breakdown of PageRank.

reduce operations. The application exhibits relatively low disk reads but high disk writes during different shuffle stages.

We see from the graph that the matrix multiplication application requires multiple resources including CPU, memory, network and storage to execute, and the demand for different resources changes with the different execution stages. For example, it is CPU dominant in the beginning stages, memory dominant in the middle stages, and finally network dominant. Statically allocating a subset of cores and memory to a given application, as is the case in current schedulers, does not consider this diversity and inconsistency in the needed resources.

2) *Task skewness in a single stage*: Existing heterogeneity-aware schedulers assume tasks in the same stage to have similar characteristics as they perform the same computation. However, this assumption does not hold true due to data skewness, shuffle operations, etc. To demonstrate the diverse task characteristics in the same stage, we perform a PageRank calculation with 20 GB input data on the 2-node cluster. Figure 3 shows the task assignments on the two nodes. We further break down the execution time into four categories: compute, shuffle, data serialization, and scheduler delay. Here y-axis represents task ID, and x-axis represents the execution time in seconds. Note that node-1 has a higher CPU processing capacity and lower network throughput than node-2. We can see that tasks in the same stage have different execution times, with the difference being as much as $31\times$ between the two nodes. Also we can see some tasks, such as task 47, are CPU

intensive where they spend most of time on compute, while other tasks, such as task 13, are shuffle intensive. However, Spark task scheduler does not consider the characteristics of the tasks and assigns most CPU intensive tasks to node-1, and shuffle intensive tasks to node-2. It also does not consider overlapping tasks with different resource demands on the same node. For example, most tasks in node-1 are compute intensive, and compete for CPU resource. Moreover, node-1 has 10 tasks assigned while node-2 has 15 tasks. The uneven task scheduling can also cause unbalanced load among the cluster.

These experiments show that, a Spark application may require varying resources even within the application life cycle. The tasks therein also have varying characteristics. Moreover, the resource heterogeneity amplifies the challenge of matching resources to appropriate tasks. RUPAM aims to manage this heterogeneity in an efficient manner.

III. DESIGN

In this section, we first describe the architecture of RUPAM. Then, we detail the heterogeneity-aware scheduling of RUPAM.

A. System architecture

Figure 4 shows an overview of the RUPAM architecture, and highlights key components in addition to the original Spark cluster and the interactions therein. The goal of RUPAM is to match the best resource with a Spark task, given a heterogeneous resource pool allocated to the application.

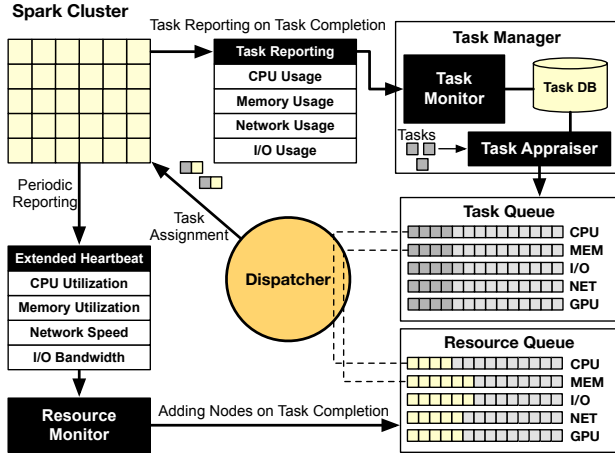


Fig. 4: RUPAM system architecture.

Note that RUPAM is a task-level scheduler that works with any application-level and job-level schedulers such as Mesos, YARN, or Spark standalone scheduler. In this work, we showcase RUPAM atop Spark standalone scheduler.

RUPAM has three major components: **Resource Monitor (RM)**, **Task Manager (TM)**, and **Dispatcher**. **RM** is in charge of real-time resource monitoring of the system. It has a central *Monitor* running in Spark master and a distributed *Collector* running on each Spark worker node. The *Collector* reports resource usage such as CPU, memory, network, I/O, and GPU on each node, and the *Monitor* collects and records the information from the nodes. **RM** can be extended to collect more information based on the resource capabilities, e.g., NVM devices. **TM** tracks the tasks resource usage to determine any resource bottlenecks for a task. **Dispatcher** component implements the main logic flow of RUPAM such as determining the size of executor to launch per node, number of tasks to launch on a specific node, matching a task to a suitable node, and scheduling tasks based on multiple factors.

RM starts when Spark is initiated on a cluster. When an application is submitted, Spark instantiates a centralized application-level resource scheduler that can be *CoarseGrainedScheduler* for Spark standalone mode, and *ExternalClusterManager* for external resource managers. In the meantime, the Spark task scheduler (*TaskScheduler*) is also launched to take control of task life cycles. When tasks are submitted to *TaskScheduler*, it queues the tasks and waits for the resource scheduler to release a node. Then *TaskScheduler* schedules a task for a given resource based on locality. Here RUPAM launches **TM** and **Dispatcher** instead of *TaskScheduler*. Instead of scheduling tasks based on locality alone, **TM** keeps track of resource utilization of each task and decides its crucial characteristics. The information is then used by **Dispatcher** to schedule task to suitable resources based on both resource characteristics of tasks and nodes.

B. Real-time resource and task monitoring

The dynamic scheduling mechanism of RUPAM relies on real-time resource monitoring and task characterization.

Specifically, when a node becomes available for running a task, RUPAM should be able to assign a task that has matching performance characteristics to the resource characteristics of the node (Section II-B2 showed tasks in the same stage with different resource bottlenecks). For instance, if the node is equipped with large amount of memory, a task requiring large memory capacity should be assigned to the node. However, the resource utilization of a node does not stay the same. Available resources can change as tasks are launching and finishing in the node. It is crucial to keep the most updated resource utilization for each node. Moreover, the resource bottleneck of the same task may change along with the status of the node it is executed on. For example, a task may have a bottleneck on CPU when executed on a node with poor CPU clock speed, but the same task may spend a large amount of time shuffling data over the network when it is executed on a powerful node (to remedy the CPU bottleneck) in the next iteration. The resource contention among tasks in a node can also affect such characteristics. Hence, it is also important to keep track of task characteristic in the entire life cycle to decide the best node for a given task with trade-offs. To this end, **RM** monitors resource utilization of each node real-time to provide updated utilization metrics, and **TM** keeps track of each task as the application executes and collects the statistics when the task finishes.

1) *Resource monitoring*: Spark adopts a master-slave deployment architecture. When a Spark worker launches, it registers itself with Spark master using a message including its ID, IP address, CPU cores, and memory. With this information, Spark master can later on launch executors with certain CPU cores and memory on each worker. To tolerate node failure, Spark master requires a simple heartbeat acknowledgement message to each worker periodically to decide whether a node is healthy. RUPAM takes advantage of this mechanism, and piggy-backs real-time resource monitoring data on the heartbeats, thus providing scalable monitoring without introducing extra communication overhead. To consider heterogeneous hardware configurations in the cluster, RUPAM supports multi-dimensional resource characterization and availability reporting. Table I (left side) shows a list of supported monitoring metrics. For static properties such as SSD and maximum network bandwidth, *collector* only sends the information once when registering to the master. For real-time properties, RUPAM collects the information periodically via our extended heartbeat messages. We consider CPU frequency as a dynamic value due to the workload-aware energy saving features in modern CPUs. In addition, RUPAM can be easily extended to support other resource types.

RM records all of the collected information from the nodes and later on pass to the **Dispatcher**. Since every node has different types of resources as listed in Table I, the key challenge is how to organize the metrics for different nodes as the system scales. To this end, RUPAM again leverages an existing object of *executorDataMap* inside Spark to reduce memory overhead. After receiving the heartbeat messages from the *collectors*, **RM** first stores all of the information

Node metrics	Description	Task metrics	Description
cpuFreq	CPU clock frequency.	computeTime	Time the task spent on computation (including serialization and deserialization).
gpu	The number of idle GPUs in the machine.	gpu	Whether the task uses GPUs.
ssd	The disk that Spark uses for intermediate data storage is an SSD device or not.	peakMemory	The maximum memory size used by the task during execution.
netBandwith	The bandwidth of the network.	shuffleRead	Time the task spent on reading shuffle data.
freeMemory	Size of free memory in the node.	shuffleWrite	Time the task spent on writing shuffle data to the disk.
cpuUtil	CPU load of the node.	optExecutor	The executor where the task has the lowest runtime by far.
diskUtil	The I/O load of the node.	historyResource	The history resource bottlenecks that TM has determined for this task.
netUtil	The network load of the node.		

TABLE I: The node metrics that **RM** monitors on each node (left) and the task metrics that **TM** monitors (right).

in the *executorDataMap* object. However, just keeping the information does not help decide appropriate nodes for given tasks. Multi-dimensional resource availability complicates this process. To help **Dispatcher** figure out the best node for each resource category, RUPAM uses one priority queue for each resource type (“Resource Queue” in Figure 4), i.e., CPU, GPU, network, storage, and memory. Each queue is sorted with capacity in descending order (most powerful/capable/capacity first) and associated utilization in ascending order (least used first). In order not to overwhelm the memory usage by keeping these queues, we only insert a record whenever a node is ready to execute a task. Based on our observation, instead of making bulk scheduling when all nodes become available, Spark detects whether a node is ready for tasks with heartbeat messages and immediately schedules a task whenever a subset of nodes are available. It then blocks incoming messages until tasks are scheduled, before going to the next round of making offers. As a result, RUPAM only needs to sort out the small subset of nodes in a single round, and all of the queues can be emptied before the next round of offers. In this way, we keep the size of our resource queues in check and associated sorting time complexity low. This minimizes the overhead of RUPAM.

2) *Task monitoring*: To select an appropriate task for a given node with certain capabilities, RUPAM also needs to determine the task characteristics. To this end, **TM** monitors the resource usage of tasks for every application and records the information. RUPAM uses a task characteristics database (DB_{task_char}) to store the task metrics based on the observation that data centers usually run the same application on input data with similar patterns periodically [31], [34]. Table I (right side) shows the task metrics that RUPAM maintains. Once tasks are submitted to **TM**, RUPAM first searches for the task in DB_{task_char} and retrieves its characteristics. To separate tasks with different resource needs in a stage, **TM** also keeps a queue for each resource type (“Task Queue” in Figure 4). As stated earlier, these queues will be reset when current tasks finish and before a new task wave arrives. Algorithm 1 gives a detailed view of the steps **TM** takes to determine resource bottlenecks. Here, Res_factor is a parameter that decides

the sensitivity to resource bottlenecks. For example, a task is considered compute-bound if it spends $2\times$ more time than shuffle operation. Users can adjust the sensitivity, and RUPAM will modify the frequency of task re-characterization and re-scheduling accordingly.

If there is no record for a task in DB_{task_char} , i.e., this is the first time the task has been submitted, RUPAM first checks the current stage of the task. If it is in map stage (*ShuffleMapTask*), RUPAM considers it to be bounded by all types of resources and thus enqueue it in all queues. A task in a reduce stage (*ResultTask*) is considered to be network bound, since a reduce task typically first reads data from shuffling and then sends the results back to the Spark driver program; activities that are all network intensive (this assumption can be relaxed by **TM** for later iterations.). When a task finishes, RUPAM combines the task metric information from Spark and records the information in DB_{task_char} for future use, i.e., future task iterations and job runs.

For tasks that use special accelerators such as GPUs, **TM** checks with **RM** to see if any GPU is used during the task execution period, and marks all the tasks in the same stage to be GPU tasks. This is because the tasks in the same stage usually perform the same computation. The **TM** updates the task metrics in DB_{task_char} whenever a task finishes. This is because the same task may show different resource usage when executing in a different environment as discussed earlier. This ensures that RUPAM has the most updated information for its decision making. However, this also creates the challenge of how to manage the overhead of frequent DB_{task_char} accesses. To address this, RUPAM creates a helper thread for accessing DB_{task_char} . All DB_{task_char} write requests are queued and served by the helper thread. For read requests, the helper thread first checks the queue to see if the task has written to the database yet, and if it has, the request is served from the enqueue requests if any before accessing the database.

C. Task scheduling

We model the problem as the scheduling of n tasks onto m parallel machines. Our objective is to minimize the total processing time of all tasks for all machines, $T_{max} = \max(T_i =$

Algorithm 1: Task characterization procedure

Input: *taskSet*, *computeTime*, *gpu*, *shuffleRead*, *shuffleWrite*, *Res_factor*

```
1 begin
2   for each task in taskSet do
3     if gpu then
4       pendingGpuTasks.enqueue(task);
5     end
6     else if computeTime >
7       Res_factors × max(shuffleRead, shuffleWrite)
8     then
9       pendingCpuTasks.enqueue(task);
10    end
11    else if
12      shuffleRead > Res_factor × shuffleWrite then
13      pendingNetTasks.enqueue(task);
14    end
15    else
16      pendingDiskTasks.enqueue(task);
17    end
18  end
19 end
```

$\sum_{\forall j} p_j$), where p_j denotes the processing time of a task j . T_{max} represents the maximal makespan among machines, which is equivalent to the makespan in parallel machines or the total processing time of all tasks for all machines. In order to capture the different hardware capability, capacity of nodes, and data locality, we assume that the processing time of the task j on each machine i , $p_{i,j}$, varies across machines. Our goal is to minimize $T_{max} = \max(T_i = \sum_{\forall j} p_{i,j})$ under the following constraints:

$$\forall i, \sum_j x_{i,j}^r \leq C_i^r$$
$$\forall i, j, x_{i,j} \in \{0, 1\},$$

where C_i^r denotes the capacity of resource r on machine i , and $x_{i,j} = 1$ if and only if a task j is scheduled on the machine i . If the resource r is not available on the machine i , $C_i^r = 0$, which prevents task j from mapping to machine i .

The relevant underlying theoretical problem that applies to our conditions is unrelated parallel machine scheduling [28], for which obtaining optimal solution has been shown to be NP-hard [4], [6], [16]. The most popular solution to this problem is list scheduling algorithm, a greedy algorithm that maps tasks to available machines without introducing idle times, if it is not needed (e.g., dependencies between tasks.) Since list scheduling provides a practical solution with a reasonably good theoretical bound [28], RUPAM adopts a heuristics based on the greedy algorithm.

Specifically, the **TM** determines resource bottleneck for each task, and passes it to **Dispatcher** for scheduling. The **Dispatcher** waits for underlying node(s) to be available. Combined with the task metrics from **TM** and node information from **RM**, **Dispatcher** is able to match a task to a node. There are several factors that RUPAM needs to consider for heterogeneous resource aware scheduling:

- hardware capability/capacity of nodes (e.g. w/wo SSDs,

Algorithm 2: Task scheduling algorithm

Input: *taskSet*, *speculativeTaskSet*

```
1 begin
2   {res, node} ← dequeue_node_rr;
3   task ← schedule_task(taskSet, res, node);
4   if task is null then
5     /*check stragglers*/
6     task ← schedule_task(speculativeTaskSet, res,
7       node);
8   end
9   Function schedule_task(pendingTasks, res, node)
10  taskList ← get_tasks_with_res(res,
11    pendingTasks); taskWithBestLoc ← null;
12  for each task in taskList do
13    if task.peakMemory > node.freeMemory then
14      /*the task has been identified to
15        bottlenecked by all of the 5 resources
16        and the history shows running on node
17        yields best performance.*/
18      if task.historyResource.size = 5 and
19        task.optExecutor = node then
20        return task;
21      end
22      else if task.get_locality(node) =
23        PROCESS_LOCAL then
24        return task;
25      end
26      else if task.get_locality(node) >
27        taskWithBestLoc.get_locality(node) then
28        taskWithBestLoc ← task;
29      end
30    end
31  end
32  return taskWithBestLoc;
```

w/wo GPU(s);

- resource consumption for each task;
- resource contention for each resource in a node;
- number of tasks running in the same node; and
- data locality.

1) *Scheduling policy:* Data locality based scheduling mitigates performance degradation due to network transfers, but is unable to capture resource and task heterogeneity as discussed earlier. In contrast, RUPAM uses multi-dimensional characteristics for scheduling. The **Dispatcher** matches the task with the right resources with the help of “Task Queue” and “Resource Queue”. Algorithm 2 describes the steps **Dispatcher** takes to schedule tasks. After **RM** populates the “Resource Queue”, RUPAM dequeues one node from each resource queue at a time in a round-robin fashion to make sure no task with a single resource type is starved. Since the first node dequeued from a specific priority queue will have the highest capacity/capability and the least utilization of the available resource type, **Dispatcher** goes over the tasks in the queue of that resource type, makes sure that the node has enough free memory to launch the task, and finally find the task with the best locality to that node in the order of PROCESS_LOCAL (data is inside the java process), NODE_LOCAL (data is on the node), RACK_LOCAL (data is on the node in the same rack) and ANY (data on a node in a different rack). This heuristic

greedily finds a node N with the best capability and lowest contention for a resource R , and then schedules to N a task T , that had R as the bottleneck in its previous run, and now N offers the best locality for T . RUPAM does not try to find the optimal scheduling strategy for each task, as that may cause a huge scheduling delay and will be counterproductive. Instead, RUPAM tries different node assignments for a task, e.g., with well-endowed CPUs or better I/O throughput, and records the node where the task has achieved the best performance. This node is used to schedule the task, even if the task has some bottleneck for say CPU on that node. This “locking” of a task to the node on which it gives the best observed performance, also prevents moving tasks back and forth between nodes due to temporary fluctuations in the task characteristics.

2) *Resource allocation*: Spark launches executors with a fixed number of cores and memory, and considers a node to be available if there are free cores in the node. This static configuration is inefficient. First, in a heterogeneous environment, nodes have different memory size and number of cores. RUPAM schedules tasks beyond this size limitation, i.e., based on the resource availability for each node. For example, RUPAM changes the executor size when necessary so that different nodes will have executors with different memory sizes. Second, determining the number of task slots based on the number of CPU cores in a node is not accurate. For example, a node with no free CPU cores may only have 10% CPU utilization if all tasks assigned the node are I/O bound. On the other hand, a node that only has one task may be using 100% of the CPU. In this case, scheduling more CPU intensive tasks will cause resource contention and slowdown the application progress. To this end, RUPAM treats a node to be available as long as it has enough resources to execute a task. The usage of priority queue makes sure that the node that is dequeued has the least utilization for such a resource. By over-committing a node that has some idle resources and matching the node with the right task, RUPAM can overlap tasks with different resource demands. For instance, a node that has all cores that are occupied by CPU intensive tasks, may have idle GPUs. It will be the first node in the GPU priority queue, and RUPAM can use it to run a GPU-friendly task. Thus making more efficient use of available resources. Such resource overlapping is possible because Spark often launches tasks from different stages at the same time whenever possible (as long as there are no data dependencies between the stages).

3) *Straggler and task relocation*: To remedy the possible suboptimal decisions **Dispatcher** may make, RUPAM also works with recently introduced Spark speculative execution system to launch copies of stragglers in available nodes. The Spark speculative execution system monitors the current runtime of the tasks. When the number of tasks completed reaches a threshold (default 75% of total tasks), the system searches the tasks that take more time than a factor ($1.5\times$ by default) of mean execution time of finished tasks and mark them as stragglers. A copy of the stragglers will be executed in the next available node to compete with the original copy. Besides

Name	CPU (GHz)	Memory (GB)	Network (GbE)	SSD	GPU	#
thor	3.2	16	1	Y	N	6
hulk	2.5	64	1	N	N	4
stack	2.4	48	1	N	Y	2

TABLE II: Specifications of Hydra cluster nodes.

the standard stragglers detected by Spark, RUPAM also detects resource stragglers due to the heterogeneous environment. For this purpose, we change the `checkSpeculatableTasks()` function in Spark to consider resource usage when marking tasks as stragglers. For example, BLAS application is designed atop underlying libraries that either use CPU (OpenBLAS) or GPU (NVBLAS) for acceleration [37]. Although such task would be marked as GPU tasks in RUPAM, RUPAM does not wait until GPU nodes are available to execute the tasks, instead it will also schedule such tasks to available nodes with powerful and idle CPU. Whichever version finishes first will continue, while the unfinished version is aborted and the resources freed. On the other hand, memory is a crucial resource because *OutOfMemory* error can cause the application to abort. In the case when the OS rejects the JVM memory allocation request, the whole JVM can be killed by the OS, which is then followed by a catastrophic failure of the Spark worker. To prevent such a scenario, RUPAM also takes an aggressive approach to detect memory stragglers. First, whenever **RM** detects a node that has low free memory, it sends a message to **TM** and by examining the currently running tasks, **TM** marks the task that has the highest memory consumption as straggler, and terminates the task in the node. After marking a task as a straggler, a copy of the task is sent to **TM**, and it analyzes the task metrics to determine the bottleneck and enqueues it to the “Task Queue” again. The **Dispatcher** can then again assign the task to a node that has appropriate idle resources for the task.

IV. EVALUATION

We evaluate RUPAM using a local heterogeneous cluster, Hydra, consisting of 12 nodes with three types of resources as Table II, namely thor, hulk, and stack. Each thor node has an 8-core AMD FX-8320E processor, and a 512 GB Crucial SSD. However, these nodes have the lowest memory capacity of 16 GB each. The hulk machines have 32-core AMD Opteron Processor 6380. The hulk machines have the highest memory capacity of 64 GB each and a high network bandwidth of 10 GbE. Finally, stack machines have 16-core Intel Xeon E5620 CPUs and a moderate memory size of 48 GB. Each stack nodes is equipped with an NVIDIA Tesla C2050 GPU. All nodes besides thor have a 1 TB Seagate HDD device as storage. We have 6 thor nodes, 2 stack nodes, and 4 hulk nodes in our cluster. We set up Spark 2.2 with one master node and 12 worker nodes with the master running on a node that is also a worker. We set the executor memory size to 14 GB to accommodate the thor machines in our cluster for default Spark setup. We evaluate RUPAM using a variety of applications (Table III) covering

Workload	Input size (GB)
Logistic Regression (LR)	60
TeraSort	40
SQL	35
PageRank (PR)	0.95 (500K vertices)
Triangle Count (TC)	0.95 (500K vertices)
Gramian Matrix (GM)	0.96 (8K*8K matrix)
KMeans	3.7

TABLE III: Studied workloads and input sizes.

SysBench	stack	hulk	thor
CPU (sec)/latency (ms)	10.14/4.63	10.06/2.23	2.16/1.73
I/O read (MB/s)	63.87	148.92	236.69
I/O write (MB/s)	6.21	52.58	134.18
Network (Mbits/s)	809	934	813

TABLE IV: Hardware characteristics benchmarks.

graph, machine learning applications, SQL, and TeraSort from SparkBench [21]. For our evaluation, we also utilize a widely-used GPU-intensive application employed in machine learning algorithms, Gramian Matrix calculation [37]. Note that Gramian Matrix and KMeans utilize GPUs for acceleration.

A. Hardware capabilities

To study the hardware capabilities of each machine group in our heterogeneous cluster, we first use SysBench [19] to benchmark the CPU and I/O performance of a node from each group. We also use Iperf [1] to determine the real network bandwidth between the workers and the master nodes. Table IV shows the results. For CPU tests, we use the default CPU test workload from SysBench that calculates 20 K prime numbers using all available cores. We record the time in seconds and latency in milliseconds. We can see that thor machines perform the best and are $5\times$ faster than stack and hulk machines. Thor also has the lowest latency. Hulk machines performs slightly better than stack machines. We also run the default I/O tests with a 10 GB file, using direct I/O to avoid memory cache effect. We observe that the thor machines have the best read and write performance, given it has the attached SSDs. Finally, we use Iperf with master node (stack1) set as the server, and test the UDP (protocol used in Spark) performance from different set of machines. Since all machines are connected through a 1 GbE network, the results are similar for all the machines. In a large-scale environment, more complicated network topology would result in a more disparate network bandwidth availability among node in different subnets.

B. Overall performance

In our first set of tests, we study the overall performance impact of RUPAM. For a fair comparison, we also enable speculative task execution (via `spark.speculation`) for default Spark in these tests. We run all workloads five times and clear `DBtask_char` after each run, and record the average execution time and 95% confidence interval under both default Spark and RUPAM. The workloads include both compute intensive (KMeans, GM, etc.) and shuffle intensive (SQL, TeraSort, etc.)

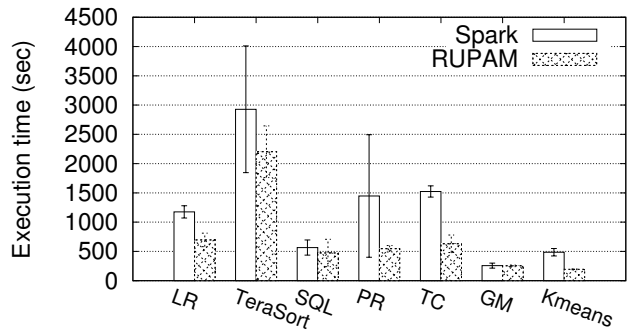


Fig. 5: Overall performance of studied workloads under default Spark and RUPAM.

applications. Some workloads, such as PR, are memory intensive such that default Spark fails with memory error in some runs. Figure 5 shows the results. We observe that all workloads experience performance improvement under RUPAM, with an average of 37.7% over default Spark. This is because RUPAM selects a node that offers the best resource for a task of each type of workload, while Spark only considers data locality, PR yields the highest speedup of $2.65\times$. This is due to the memory error failure and recovery during the execution (which also causes the large error bar for PR with default Spark). In contrast, RUPAM finishes without memory errors due to the dynamic executor memory configuration based on each node’s memory capacity as well as the memory usage aware task scheduling policy discussed in Section III-C. KMeans also achieves a $2.49\times$ speedup, but GM only shows a negligible 1.4% performance improvement. This is because GM only has one iteration of computation, which makes it difficult for RUPAM to test and determine an appropriate resource for the workload, while KMeans’ five iterations enable RUPAM to better match tasks with suitable resources. The behavior is also observed for other workloads with only one iteration such as SQL (per query) and TeraSort, which only have moderate speedups of $1.19\times$ and $1.32\times$, respectively. Workloads with multiple iterations (PR, LR, TC, KMeans) have an average speedup of $2.31\times$. This shows that RUPAM performs better when there are multiple iterations in a workload due to the log based task characterization of RUPAM. The more iterations an application has, the better matching RUPAM can achieve for the application.

In order to have a clearer view of the relationship between the performance of RUPAM and the number of iterations of a workload, we experiment further with LR. Here, we use the same input size but alter the times of regression to vary the number of iterations of the workload. We record the speedup of RUPAM compared to default Spark in Figure 6. We can see that as the number of iteration increases, the speedup achieved by RUPAM also increases, up to $3.4\times$. Note that regardless of iterations, RUPAM is able to match or outperform the default Spark scheduler, thus making it a desirable approach for all types of applications.

C. Impact on locality

RUPAM attempts to find the task whose requirements best

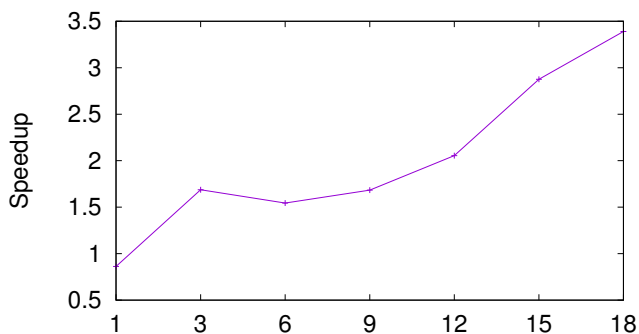


Fig. 6: Speedup of LR under RUPAM wrt. default Spark with increasing number of workload iterations.

match a node’s resource, as well as will achieve the best data locality on the node. However, it is possible that a node N that offers the best locality (`NODE_LOCAL` and `ANY`) will affect a task T ’s performance significantly when considering other resources beside locality, and thus, unlike default Spark N will not be picked by RUPAM for running T . In our next test, we record the number of tasks that are scheduled by RUPAM on a node with different locality than a node chosen by default Spark. We use this number as a measure of RUPAM’s impact on preserving data locality similar to default Spark. Table V shows the results. Note that all workloads have zero `RACK_LOCAL` tasks. We can see that for all workloads, default Spark has more `PROCESS_LOCAL` tasks than RUPAM. This is expected, as Spark aims to schedule task with the best data locality available for a node. For some workloads, such as TeraSort and TC, Spark has more total number of tasks than RUPAM. This is due to the fail and retry of some tasks with the out-of-memory error under Spark when they are scheduled to a node with less memory capacity and high memory contention. In such cases, Spark still strives to allocate tasks with the best data locality, so we still observe a lower number of `NODE_LOCAL` and `ANY` tasks under such scenario. However, RUPAM has more tasks with poorer data locality for such workloads because RUPAM relocates tasks to a node with higher memory capacity and lower contention. Thus, RUPAM trade-offs locality for better matching resources in such cases, with the goal to achieve higher end-to-end performance (in this even task completion and error avoidance). As the goal of any big data platform is faster time to solution, and not preserving locality for the sake of it, such trade-offs are justified and necessary.

D. Performance breakdown

In our next experiment, we select three representative workloads for each category—LR (machine learning), SQL (database), and PR (graph)—and study the breakdown of performance into five categories: compute, garbage collection (GC), network shuffle, disk shuffle (read and write), and scheduler delay. Figure 7 shows the results. We find that all selected workloads have improved compute times, which underscores RUPAM’s ability to schedule compute-intensive tasks to nodes with better computational power and less CPU utilization to reduce contention (Section III-C). For the LR

workload, we observe less GC overhead for RUPAM, but similar or higher GC overhead for PR and SQL, respectively. Combined with Figure 8(b), we see that SQL consumes the largest amount of memory among the three studied workloads. Furthermore, SQL has only one iteration per SQL query with no data that needs to be preserved across queries, but involves a lot of shuffle operations for data join, so GC is triggered often to free space for shuffle. Moreover, RUPAM increases the memory usage up to the node capacity compared to a conservative configuration of Spark, thus resulting in JAVA spending more time to search the whole JVM memory space for GC, resulting in a big GC overhead compared to default Spark. On the other hand, LR has moderate memory usage and intermediate data needs to be kept across iterations. In this case, larger memory capacity provided by RUPAM is able to cache more data compared with the static memory configuration of default Spark where more GC operations are triggered for LRU cache management. Thus, RUPAM entails less GC operations and experiences a lower GC overhead.

For SQL workload, RUPAM yields a high shuffle overhead than Spark. This is because SQL only has one iteration and RUPAM simply treats the tasks to be general without specific resource bottleneck as described in Section III-B2. Moreover, RUPAM achieves worse data locality compared to Spark as shown in Section IV-C, which result in worse shuffle performance. For other workloads, such overhead is mitigated by the performance improvement due to correct characterization of tasks. From Table V, we see that for LR, RUPAM has 15 `NODE_LOCAL` tasks and 0 `ANY` tasks, while Spark has 12 `NODE_LOCAL` tasks and 0 `ANY` tasks. Thus, we do not observe much shuffle overhead over network, but we observe a higher shuffle overhead from disk for Spark than for RUPAM. This is because although RUPAM has similar number of `NODE_LOCAL` tasks to Spark, RUPAM schedules the I/O intensive tasks to nodes with SSDs. On the other hand, we see that RUPAM has 13 `ANY` tasks but Spark only has 1. This creates the larger shuffle network overhead of RUPAM. PR has a similar number of `ANY` tasks, but RUPAM has twice the number of `NODE_LOCAL` tasks than Spark. However, here again we observe similar shuffle disk overhead of RUPAM due to the I/O task scheduling of RUPAM.

Finally, we observe that although RUPAM takes more steps for task scheduling, by tracking resource and task monitoring, and using the simple heuristic, the resulting scheduler delay under RUPAM is moderate compared to default Spark.

Discussion: We carefully select the input data size to fully saturate the capacity of the scale of our setup, such that the task would execute without crashes under default Spark. All task slots are filled during the experiments. We expect that bigger data size would generate more tasks to be scheduled in the queue. With the same hardware configuration, RUPAM continues to strategically assign tasks based on real-time resource consumption of both tasks and the underlying nodes, while Spark’s static task slot-based scheduling policy can cause suboptimal scheduling decision and resource contention. Though speculative execution can ease the problem, more

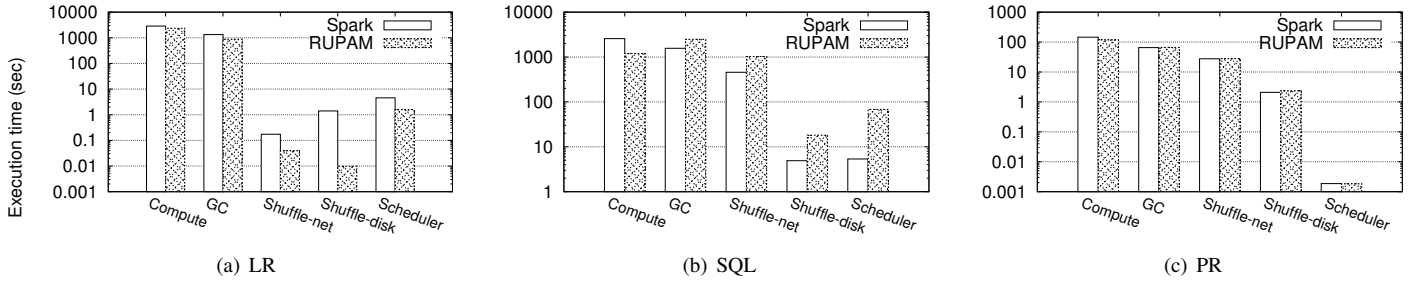


Fig. 7: Performance breakdown of selected workloads under default Spark and RUPAM. Note that y-axis is log-scaled.

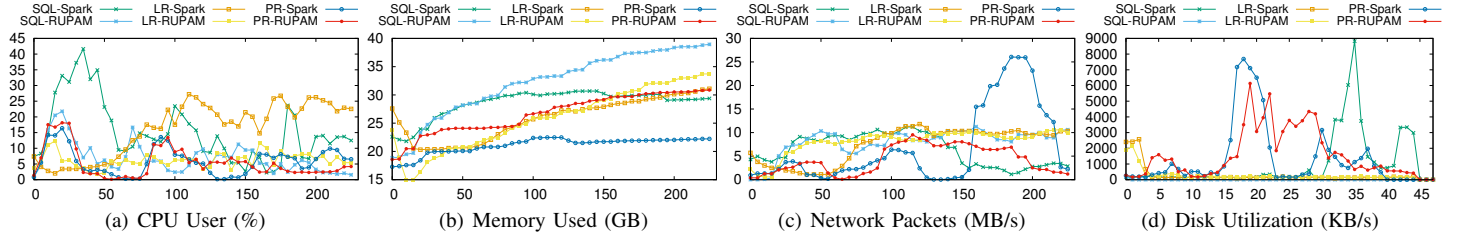


Fig. 8: Average system utilization of the nodes with selected workloads under default Spark and RUPAM.

Workload	PROCESS		NODE		ANY	
	Spark	RUPAM	Spark	RUPAM	Spark	RUPAM
LR	295	292	12	15	0	0
TeraSort	2490	1803	339	188	30	12
PR	13800	13785	16	33	134	132
TC	2052	3924	612	1605	556	949
SQL	492	480	0	0	1	13
GM	512	256	64	320	0	0
Kmeans	2004	2259	95	101	10	0

TABLE V: Number of tasks with different locality level under default Spark and RUPAM.

frequent launching and relaunching of tasks also increase the scheduling overhead.

E. Impact on system utilization

In our final test, we investigate how RUPAM impacts resource utilization. We repeat the previous experiment (with LR, SQL, and PR) and measure the average utilization of CPU, memory, network, and disk I/O of the 12 nodes in our cluster. Figure 8 shows the results. We see that for CPU utilization, RUPAM shows a lower average CPU user percentage compared with default Spark. This is because RUPAM takes the real-time CPU utilization into consideration when assigning tasks, which can help balance CPU load and reduce CPU contention. The same trend can also be observed for network and disk I/O utilization. However, for memory, RUPAM shows a higher usage than default Spark for all workloads. This is because default Spark takes a static global configurable memory size for launching the executors on each worker node. In our setup, we have to accommodate the node with the smallest memory size in order to launch the executors without memory errors. However, RUPAM is able to launch executors with different

memory sizes on different nodes with different memory capacity. Thus, RUPAM yields a higher overall/average memory usage.

Next, to test RUPAM’s impact on the resource load balance, we also calculate the standard deviation of the resource utilization among the nodes in the cluster during the execution of workloads. To get a clear view, Figure 9 only shows the result for PR. However, the other workloads show similar patterns. Here we omit the results for memory usage due to the RUPAM’s design of using all available memory of the node. We observe from the figure that, in general, RUPAM shows a lower standard deviation of CPU utilization than default Spark. For network and disk I/O, Spark shows spikes, while RUPAM keeps a low and stable standard deviation. This is because PR performs heavy shuffle operations in the late stages, and stresses the use of network and disk. The low standard deviation among the nodes of RUPAM shows that RUPAM scheduling is able to dispatch tasks to nodes with less contention on the resources and balance the resource utilization among the nodes in a cluster, while Spark scheduler only considers data locality and may cause an unbalanced workload and contention on individual nodes, which results in a higher standard deviation in utilization among the nodes.

V. RELATED WORK

Rolling server upgrades is a common practice in large scale clusters and data centers, which inherently make the systems more heterogeneous [20], [24]. Thus, extensive work [5], [14], [17], [25], [26], [29], [30], [32] has been conducted for sharing heterogeneous clusters among multiple applications for sharing heterogeneous clusters among multiple applications focus on multi-tenancy environment and allocate resources in a heterogeneous environment to an application based on

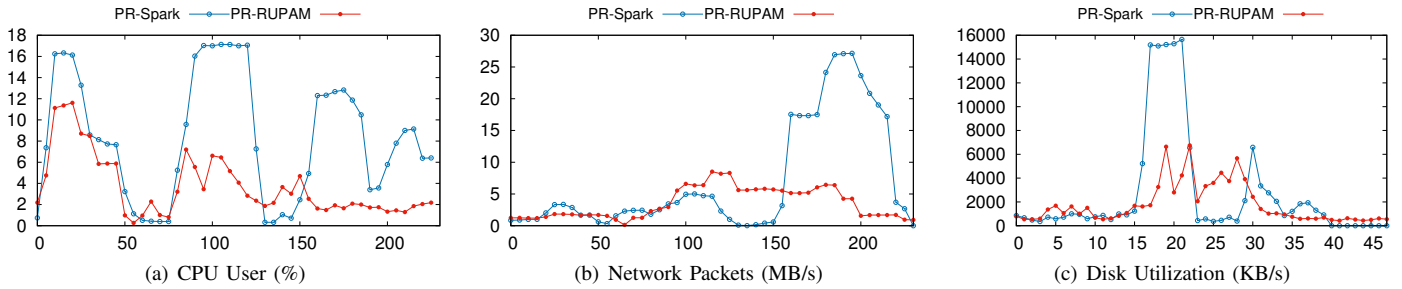


Fig. 9: Standard deviation of system utilization among the nodes with selected workloads under the default Spark and RUPAM.

application types. In contrast, RUPAM monitors and captures the diverse task and hardware characteristics, and thus aims to create a fine-grained resource scheduler with the aim to improve performance for individual tasks instead of just at application level (as applications can have tasks with varying demands). The design of RUPAM is orthogonal to works on heterogeneous resource management, and can co-exist and benefit from them as a second-level scheduler.

Scheduling optimization for MapReduce jobs in heterogeneous environment is also extensively studied both for Hadoop [3], [7], [10], [12], [34], [36], [41] and Spark [39]. Here, techniques such as straggler detection via progress rate and time estimation for better schedule and reschedule tasks are used. These approaches rely on application monitoring to make reactive scheduling decisions. In contrast, RUPAM considers task characteristics as well as resource heterogeneity and utilization when scheduling a task. RUPAM also adopts the idea of reactive scheduling as needed to avoid suboptimal decisions. Another aspect is taken up by works such as [3], [5], [8], [13], [27], which consider the resource utilization when scheduling tasks. However, heterogeneous cluster scheduling that considers multiple resources is a hard problem, both in theory [15] and practice [3], [16], [22]. The challenges here stem from the dynamic interactions between processes (or tasks) within a single application such as synchronization, load balancing, and the inter-dependency between processes; and the dynamic interactions between applications such as the interleaving resource usages with other collocated independent workloads [38]. To reduce the complexity of the problem, these works focus on dominant resource optimization per a stage and assumes all tasks in the same Map/Reduce stage share the same resource characteristics. In contrast, RUPAM considers resource usage pattern for each task and adopts a heuristic to reduce complexity and thus improve performance. Tetris [16] proposes to pack multiple resources in a cluster for scheduling tasks. It applies the heuristics for the multi-dimensional bin packing algorithm. However, Tetris works with YARN in a homogeneous environment and assumes that tasks of a job have the same resource requirements to reduce the search space. In contrast, RUPAM takes a step further to consider both heterogeneous cluster resources and heterogeneous tasks, within Spark’s own task scheduler.

VI. CONCLUSION

In this paper, we present RUPAM, a heterogeneity-aware task scheduling system for big data platforms. RUPAM goes beyond just using data locality for task scheduling, and also factors in both task-level resource characteristics and underlying hardware characteristics including network, storage, and out-of-core accelerators in addition to the extant CPU and memory. RUPAM adopts a self-adaptable heuristic for scheduling tasks based on the collected metrics without loss of data locality. Experiments with an implementation of RUPAM atop Spark shows an overall performance improvement by up to 62.3% compared to the extant Spark task scheduler. In our future work, we plan to explore machine learning techniques to further fine-tune, enhance, and adapt RUPAM to dynamic workloads and heterogeneous hardware.

ACKNOWLEDGMENTS

This work is sponsored in part by the NSF under the grants: CNS-1405697, CNS-1422788, and CNS-1615411. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

REFERENCES

- [1] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. <https://iperf.fr>. [Online; accessed 12-Dec-2017].
- [2] Apache Hadoop. <http://hadoop.apache.org>, 2017. [Online; accessed 12-Dec-2017].
- [3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *ACM SIGARCH Computer Architecture News*, 2012.
- [4] M. K. Ahsan and D.-b. Tsao. Solving resource-constrained project scheduling problems with bi-criteria heuristic search techniques. *Journal of Systems Science and Systems Engineering*, 12(2):190–203, Jun 2003.
- [5] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007.
- [6] W. Chen, Y. Xu, and X. Wu. Deep reinforcement learning for multi-resource multi-machine job scheduling. *arXiv preprint arXiv:1711.07440*, 2017.
- [7] D. Cheng, J. Rao, Y. Guo, C. Jiang, and X. Zhou. Improving performance of heterogeneous mapreduce clusters with adaptive task tuning. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):774–786, March 2017.
- [8] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

- [9] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, pages 127–144, New York, NY, USA, 2014. ACM.
- [10] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju. Marla: Mapreduce for heterogeneous clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 49–56. IEEE, 2012.
- [11] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, pages 99–112, New York, NY, USA, 2012. ACM.
- [12] R. Gandhi, D. Xie, and Y. C. Hu. Píkachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *USENIX Annual Technical Conference*, pages 61–66, 2013.
- [13] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.
- [14] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378. ACM, 2013.
- [15] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.
- [16] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14*, page 455, New York, NY, USA, 2014. ACM.
- [17] 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*, volume 11, pages 22–22, 2011.
- [18] 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. Morphheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, 2016. USENIX Association.
- [19] A. Kopytov. *SysBench*. <http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf>.
- [20] G. Lee. *Resource allocation and scheduling in heterogeneous cloud environments*. University of California, Berkeley, 2012.
- [21] 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, CF '15*, pages 53:1–53:8, New York, NY, USA, 2015. ACM.
- [22] S.-H. Lim, J.-S. Huh, Y. Kim, G. M. Shipman, and C. R. Das. D-factor: a quantitative model of application slow-down in multi-resource shared systems. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):271–282, 2012.
- [23] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proc. SOSP*, 2017.
- [24] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.
- [25] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé. *Resource-Aware Adaptive Scheduling for MapReduce Clusters*, pages 187–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [26] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.
- [27] B. Sharma, T. Wood, and C. R. Das. Hybridmr: A hierarchical mapreduce scheduler for hybrid data centers. In *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.
- [28] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. *SIAM journal on computing*, 24(6):1313–1331, 1995.
- [29] Z. Tan and S. Babu. Tempo: robust and self-tuning resource management in multi-tenant parallel databases. *Proceedings of the VLDB Endowment*, 9(10):720–731, 2016.
- [30] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *The 37th IEEE International Conference on Distributed Computing Systems*, 2017.
- [31] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, pages 1013–1020, New York, NY, USA, 2010. ACM.
- [32] C. Tian, H. Zhou, Y. He, and L. Zha. A dynamic mapreduce scheduler for heterogeneous workloads. In *2009 Eighth International Conference on Grid and Cooperative Computing*, pages 218–224, Aug 2009.
- [33] 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 International Conference on Cloud Computing*, page 5. ACM, 2013.
- [34] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC '11*, pages 235–244, New York, NY, USA, 2011. ACM.
- [35] A. Verma, L. Cherkasova, and R. H. Campbell. Profiling and evaluating hardware choices for mapreduce environments: An application-aware approach. *Performance Evaluation*, 79:328 – 344, 2014. Special Issue: Performance 2014.
- [36] B. Wang, J. Jiang, and G. Yang. Actcap: Accelerating mapreduce on heterogeneous clusters with capability-aware data placement. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 1328–1336. IEEE, 2015.
- [37] L. Xu, S.-H. Lim, A. R. Butt, and R. Kannan. Scaling up data-parallel analytics platforms: Linear algebraic operation cases. In *Proceedings of the 2017 IEEE International Conference on Big Data*, 2017.
- [38] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, 2013.
- [39] H. Yang, X. Liu, S. Chen, Z. Lei, H. Du, and C. Zhu. Improving spark performance with mpte in heterogeneous environments. In *2016 International Conference on Audio, Language and Image Processing (ICALIP)*, pages 28–33, July 2016.
- [40] 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*, page 2. USENIX Association, 2012.
- [41] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OsdI*, volume 8, page 7, 2008.
- [42] X. Zhang, Z. Chen, L. Zhao, A. P. Boedihardjo, and C. T. Lu. Traces: Generating twitter stories via shared subspace and temporal smoothness. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1688–1693, Dec 2017.
- [43] X. Zhang, L. Zhao, A. P. Boedihardjo, C. Lu, and N. Ramakrishnan. Spatiotemporal event forecasting from incomplete hyper-local price data. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 507–516, 2017.
- [44] X. Zhang, L. Zhao, Z. Chen, A. P. Boedihardjo, J. Dai, and C. T. Lu. Trendi: Tracking stories in news and microblogs via emerging, evolving and fading topics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1590–1599, Dec 2017.