Symbiosis: Network-Aware Task Scheduling in Data-Parallel Frameworks

Jingjie Jiang*, Shiyao Ma*, Bo Li*, and Baochun Li†
*Department of Computer Science and Engineering, Hong Kong University of Science and Technology
†Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—Even with the recent proliferation of in-memory computation in data-parallel frameworks (such as Spark), transfers over the network are still time-consuming. Similar to computation, network transfers serve as main roadblocks as we try to minimize job completion times. Existing schedulers were designed as isolated solutions that focused on computation or network performance only. Without any coordination, the utilization of computation and network resources may become unbalanced, leading to a reduced level of overall resource utilization. In this paper, we design, implement, and evaluate *Symbiosis*, a network-aware task scheduler designed to coordinate computation-bound and network-bound tasks in a large cluster, so that resources are utilized in a more balanced fashion. *Symbiosis* is an online scheduler that predicts resource imbalance before launching tasks, and correct such imbalance by co-locating computation-bound and network-bound tasks in the same executor process. As a guiding principle, it is engineered to be practically implemented within and to complement existing data-parallel frameworks. We have implemented *Symbiosis* within Spark, and carried out our experiments on a 100-node cluster. We show convincing evidence that *Symbiosis* reduces job completion times by 11.9% in comparison to Spark's current scheduler with little overhead.**

I. Introduction

Traditional data-parallel computing frameworks, such as MapReduce [1] and Pregel [2], serve as important foundations for big data processing in cloud datacenters. They assist to accelerate the processing of large datasets by dividing a job into multiple tasks, and by distributing them to machines (called *worker nodes*) across the cluster for parallel execution. In addition, a job involves multiple computation stages [3], and tasks in a downstream stage have to wait for intermediate results to be produced from their upstream stages. In MapReduce, for example, intermediate results need to be transferred from mapper tasks to reducer tasks with large volumes of data to be transmitted. Called a *shuffle* phase, such high-volume data transfers affect job completion times substantially. On average, MapReduce jobs spend about 33% of their runtime in the shuffle phase [3].

With the recent proliferation of in-memory computing frameworks, such as Spark [4] and Flink [5], Resilient Distributed Datasets (RDD) are introduced to allow in-memory sharing across pipelined tasks, mitigating the volume of data transfers. Still, for those tasks that are dependent upon a large number of other tasks in upstream stages, the shuffle phase is indispensable: jobs running on Spark spend more than 12% of their runtime during the shuffle phase [3]. A typical production cluster hosts multiple types of data-parallel frameworks, running a mix of background batch jobs and interactive query jobs on a common dataset. In such production environments, network transfer times are critical contributing factors to job completion times.

Nevertheless, existing schedulers such as HFS [6], Sparrow [7] and KMN [8], have only considered computation resources, such as CPU and memory, when assigning tasks to worker nodes. In contrast, existing bandwidth allocation schemes in the research literature, such as Orchestra [3], Varys [9], Baraat [10] and Aalo [11], have instead emphasized network transfer times, with a sole focus on network resources. Without coordination, these isolated resource allocation strategies may lead to under-utilized resources, due to a lack of balance when both computation and network resources are considered. For example, if a network-intensive task is assigned to a CPU core on which no other tasks are running, the CPU core may be left mostly idle, since data transfers over the network is the performance bottleneck.

At first glance, it may appear that *multi-resource* scheduling policies proposed in the literature [12], [13], [14] may be the best candidate to rectify such an imbalance in resource utilization across multiple types. Unfortunately, in production clusters and data-parallel frameworks, there are practical limitations that make it difficult to apply existing multi-resource scheduling policies. For example, it is difficult to have *a priori* knowledge of the precise *resource demand* in a typical job, background or interactive. Resource demands are not typically specified in job descriptions, and even when they are, they are ballpark estimates that can only serve as hints and guidelines. At runtime, neither network transfer times nor computation times are deterministic or predictable, as co-located tasks share cluster resources. Further, existing multi-resource scheduling policies are typically designed with a "clean-slate" approach, engineered to replace current data-parallel frameworks, rather than complementing them.

In this paper, we design, implement, and evaluate *Symbiosis*, a *network-aware* task scheduler that complements and improves current scheduling policies in existing data-parallel frameworks, with a coordinated emphasis on both computation and network resources when scheduling decisions are made.

The research was supported in part by grants from RGC under the contracts 615613 and 16211715, a grant from NSFC and Guangdong joint project under the contract U1301253, and the NSERC Strategic Networks grant titled "Smart Applications on Virtual Infrastructure."

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

Symbiosis is first and foremost designed to be *practical*: it predicts and rectifies a lack of balance in utilizing computation and network resources at runtime, without any *priori* knowledge or online estimation about resource demands.

As its name suggests, *Symbiosis* rectifies any resource imbalance by scheduling computation-intensive tasks (with data locality) on the same CPU core as network-intensive tasks (without data locality), within the same executor process managed by existing data-parallel frameworks, such as Spark. When multiple computation-intensive tasks compete for the same *symbiotic* opportunity on an idle CPU core hosting a network-intensive task, we propose a priority-based strategy to choose the best candidate. When multiple network-intensive tasks compete for the same symbiotic chance on a worker node without any network transmissions, we prefer the candidate that is nearer to its upstream tasks to reduce cross-rack traffic.

As a new task scheduler, *Symbiosis* is designed as a plugin to existing data-parallel frameworks, and co-exists well with existing two-tier resource managers in traditional data-parallel frameworks. It remains transparent to applications, and requires no modifications to existing programming interfaces. We have implemented it in Spark, and extensively evaluated *Symbiosis* in testbeds with modest and large scales, ranging from a few to over a hundred worker nodes on Linode [15], with a diverse range of workloads. In comparison to Spark, we have shown convincing evidence that *Symbiosis* improves CPU utilization in the cluster, mitigates the lack of balance when multiple resources are considered, and reduces the average job completion time by 11.9% on average.

II. Network-Aware Task Scheduling

In the era of big data processing, data-parallel computing frameworks, such as MapReduce [1] and Spark [4], have been widely adopted in datacenters. In these frameworks, it is typical for a computation job to process datasets through a sequence of *stages* [3], and each stage consists of multiple tasks, running on their respective worker nodes across the cluster. In input stages, tasks need to read input datasets from the file system; with in-memory processing in Spark, data locality in these stages are further improved. In intermediate stages, task need to retrieve intermediate results from previous stages for subsequent processing, leading to network transfers across the network during the *shuffle* phase.

While existing schedulers in the literature focused only on computation resources (*e.g.*, Sparrow [7], [6], Mesos [16] and YARN [17]) or network resources (*e.g.*, Varys [9], Baraat [10] and Aalo [11]), our design objective is to take both computation and network resources into account, as jobs require both resources to complete.

To be more general, we consider a cluster that simultaneously runs several different computing frameworks, or multiple instances of the same framework with different resource bounds. In such a cluster, a cluster resource manager, such as Mesos [16], is responsible for offering computation resources to different frameworks. Each worker node in the cluster can launch multiple *executor* processes, and run tasks on these executors through multithreading. Whenever an executor becomes idle, the cluster manager will select and inform one of the frameworks about the resources that has just become available, referred to as a *resources offer*.

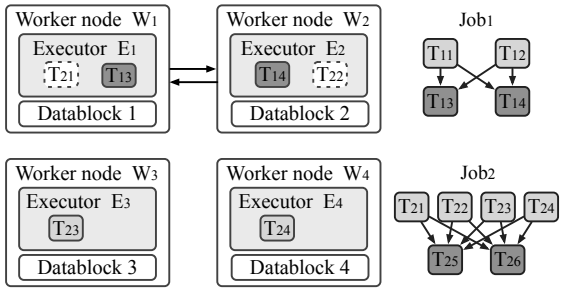


Fig. 1. A motivating example: T_{11} and T_{12} have finished and released the executors E_1 and E_2. T_{13} and T_{14} are then distributed to E_1 and E_2. Without network-aware scheduling, the scheduler would only launch two tasks of Job_2, T_{23} and T_{24}, for now since it believes that no other executor is idle.

Upon receiving the resource offer, the selected framework calls its own task scheduler to decide whether to accept such an offer, and if so, which task should be distributed. Nevertheless, the selected task might need to wait for its input data shuffled from previous stages before it starts computing. The computation resources allocated to it are thus left idle, waiting for the network transfer to complete. If we can discover and correct such resource imbalance, more tasks can be accommodated, and job performance will be improved as a result.

A. Network-Aware Task Scheduling: An Example

To understand the benefits of network-aware scheduling, consider the scenario shown in Fig. 1. Each worker node has one CPU core, stores one data block, and launches one executor process. When Job_2 is submitted to the system, the map tasks of Job_1, T_{11} and T_{12}, have finished and written the output to local disks. According to the first-in-first-out strategy, the two reduce tasks of Job_1 could first select the offers from the executors. Therefore, the scheduler would distribute them to E_1 and E_2 to reduce the volume of network transfer. Each input task of Job_2 needs to read one data block: T_{21} needs Datablock_1; T_{22} needs Datablock_2; T_{23} needs Datablock_3; T_{24} needs Datablock_4. To achieve data locality, the scheduler first launches T_{23} to E_3 and T_{24} to E_4 since the executors that T_{21} and T_{22} can achieve locality is currently occupied.

Without network-aware scheduling, a scheduler would believe that the cluster is already fully utilized, since every CPU core is allocated with one task. T_{21} and T_{22} have to wait for the tasks running on their existing executors to finish. More specifically, they have to wait for the network-intensive tasks if they would like to achieve data locality.

However, since T_{13} and T_{14} are network-bound, the CPUs of their executors are mostly idle. Intuitively, if we assign the network-free tasks T_{21} and T_{22} to E_1 and E_2 respectively, T_{21} and T_{22} can then utilize the computation resources on these executors, while T_{13} and T_{14} receive their intermediate results from the mappers. The cluster can thus run all six ready tasks

in parallelg. On one hand, the system throughput and resource utilization are improved. On the other hand, the completion time of Job_2 is reduced since its final result can be achieved only after all its constituent tasks finish. At the same time, the completion time of Job_2 is not affected. Therefore, the overall job completion performance has been improved.

B. Symbiosis: Objectives and Practical Considerations

It is our objective that *Symbiosis* should, first and foremost, be designed to be simple and practical. In the existing literature, scheduling jobs with multiple resource constraints is a classical problem in the domain of operation research [18], with the objective of equal utilization of all the type of resources. In the context of cluster computing, however, the resource demands cannot be explicitly quantized. The demand for network resources cannot be directly transferred to bandwidth requirements, and the demand for computation resources cannot be directly converted to CPU and memory constraints. Without knowing the desired network transfer times and computation times, we cannot apply existing multi-resource scheduling heuristics in *Symbiosis*.

Even if we assume that cluster users could acquire accurate information about the resource demands of each task beforehand, the application interfaces have to be modified to embrace such information into the task description. This is practically undesirable, since it breaks backward compatibility with a vast collection of existing applications. Ideally, our task scheduler in *Symbiosis* should be network-aware while still keeping the modifications *transparent* to the cluster users, without the need of modifying their applications.

In a similar vein, it is not practically feasible to incorporate a "clean-slate" redesign, by replacing existing schedulers with a new scheduling framework. The existing scheduling framework in Spark, for example, has its own merits when diversified workloads with batch and low-latency interactive jobs coexist. It is a much better design philosophy to *incrementally* improve the current scheduling framework, and keep the network-aware scheduler *compatible* to existing task schedulers in different frameworks. Last but not the least, the scheduling logic should be *light-weight* and *highly efficient*, in order to make sure that the scheduler itself does not become a performance bottleneck. An algorithm designed to require too much information would incur a significant amount of computation and control overhead, and may slow down the jobs being scheduled in the worst case.

The critical challenge is: how should *Symbiosis* be designed to be network-aware in the best possible way, while still maintaining its practicality, efficiency, and backward compatibility? In what follows, we will take a bottom-up approach, starting by motivating and designing our main ideas on identifying and correcting a lack of resource balance at runtime.

III. *Symbiosis*: Design

A. Spotting Resource Imbalance

We focus on two types of resources: bandwidth on access links and computing slots on CPU cores. The utilization of links in the core network layers depends on routing strategies [19] [20] and is beyond the scope of this paper.

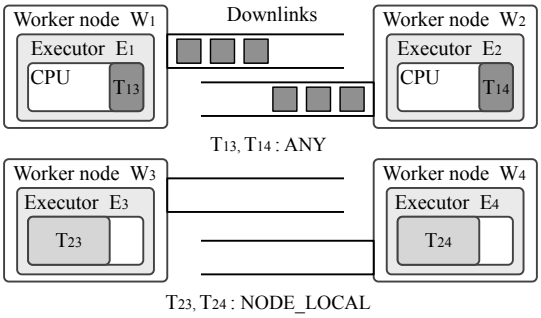


Fig. 2. In *Symbiosis*, we do not actively monitor the CPU and network utilization. We predict that E_1 and E_2 may experience CPU under-utilization by discovering T_{13} and T_{14} do not have any locality; we predict that E_3 and E_4 do not utilize the network since T_{23} and T_{24} have achieved node locality.

A worker node is considered resource balanced, if its utilization of all of its resources are the same [18]. According to this definition, it seems straightforward to identify a lack of resource balance by monitoring network and CPU usage at runtime. However, we believe that the overhead of real-time resource monitoring is not negligible, and must be seriously considered. But more importantly, real-time monitoring can only detect that resource usage is not balanced after the fact, which may be too late for the purposes of scheduling. For instance, after an under-utilized CPU core has been reported to the scheduler, the network-intensive task may have completed retrieving data and started computation already, with better utilization of the CPU. If the scheduler still attempts to make decisions to improve CPU utilization, it would now negatively affect the resource balance.

Even if our runtime resource monitoring mechanism is perfect and reports accurate real-time system states with no overhead, how the *resource balance* is traditionally defined is still impractical. On one hand, worker nodes possess heterogeneous resources: some nodes have more powerful CPUs, whereas others have faster network access. Given the same tasks, the resource utilization on these two type of nodes could be very different. On the other hand, jobs running in a cluster with multiple concurrent frameworks may have diversified and dynamic demand for different types of resources. It is therefore impractical to impose equal utilization as the standard for a balanced use of multiple resource types.

Instead, *Symbiosis* is designed to *predict* a lack of resource balance at runtime on a per-task basis. As a high-level overview of our design, we show an example in Fig. 2, where the jobs and tasks are the same as in Fig. 1. T_{13} and T_{14} need to retrieve a certain amount of input data from tasks in the preceding stage before they can start processing the data. Therefore, the access links of W_1 and W_2 are fully occupied, whereas the CPU cores are under-utilized. In contrast, since the input data of T_{23} and T_{24} are already on the corresponding worker nodes, it is possible to cache entire data blocks in their main memory, speeding up their access to the data.

Therefore, these tasks can directly start processing and fully utilize the CPUs. It is clear that a network-bound task always incurs a lack of resource balance due to CPU under-utilization, whereas a computation-bound task with data locality always under-utilizes its network links. Based on these observations, in *Symbiosis*, we simply predict that a worker node *will* be unbalanced in its resource usage if a single network-bound or a computation-bound (but network-free) task has been assigned to an executor process on that worker node.

It now remains to be seen how a task can be correctly identified to be network- or computation-bound. In practice, *Symbiosis* takes advantage of the *NameNode* [21] in the underlying file system, HDFS. The NameNode is a unique central repository that manages the system metadata, including the directory tree of all the files in the system, and tracks where across the entire cluster the data is stored. A large number of *DataNodes* in the system store the actual data blocks and interact with the NameNode to respond to client requests.

Symbiosis first submits a query to the NameNode to locate the input data blocks of the task in question. For an input task, the NameNode returns a list of relevant DataNodes that store or cache its input data blocks through the `LocalityQuery` procedure. If the executor process that the task will be launched to runs on one of the returned DataNodes, the task achieves data locality. We then determine its locality level with respect to this specific executor. If the executor caches the data block in its main memory, the task is *process-local*. If the corresponding worker node of the executor stores the data block in local disks, the task is *node-local*. Process- or node-local tasks are considered computation-bound (and network-free). Otherwise, if a task does not exhibit either process- or node-locality, it is considered to be network-bound. Depending on the distance between the executor and a node storing the data block, a network-bound task can be rack-local or without any locality. With respect to those tasks in intermediate stages, since they depend on multiple outputs from tasks in upstream stages, it is impossible to achieve data locality for all the input. Therefore, we can directly identify them as network-bound tasks from the stages they belong to.

B. Correcting Resource Imbalance

Now that resource imbalance stems from running a network-bound or a computation-bound task alone on an executor, it is intuitive to co-locate these two types of tasks to correct such imbalance. This intuition is conceptually simple; the key question, however, is how it can be implemented in practice.

An application independently defines the number of cores a task needs, and relies on the cluster manager to allocate resources. Each worker node launches multiple executors to run tasks from different frameworks. With modern container techniques, such as *Docker containers* [22] supported by Spark, the cluster manager is able to isolate co-located executors, and to limit the resources each of them can access. Therefore, each executor essentially possesses a fixed number of cores throughout its lifetime. Assigned to an executor, each task can only access the resources of its own executor, without the ability to utilize the idle resources allocated to other executors. The aggregated resources of co-located executors cannot exceed the worker node's capacity, and the total resources allocated to co-located tasks cannot surpass the executor's capacity.

To improve CPU utilization, one possible solution is to devise a "forging" mechanism that makes a worker node believe it has more cores than its physical capacity. As a result, the worker node either launches more executors [23], or launches executors with higher capacities [16]. Either way, more tasks would be assigned to these worker nodes by the scheduler. It is then possible that a network-bound task and a computation-bound task may share the same CPU. However, the forged information about the number of CPU cores must be *manually* set by the operator across the entire cluster. Such manual configurations do not have the flexiblity to adapt to different worker states across the cluster. For workers whose running tasks are all computation-bound, this method would further exacerbate the burden on the CPUs, potentially harming the computing performance.

To correct resource imbalance automatically under all circumstances, in *Symbiosis*, decisions are made at runtime within the task scheduler within each framework, rather than centrally within the cluster manager. Recall that as resources become available, the cluster manager will selectively inform one of the frameworks about the executor with the available resources, called a *resource offer*. Traditionally, the task scheduler in the selected framework would determine which task is to accept the offer. At this moment, *Symbiosis* steps in and scrutinizes the description of the selected task, before it is actually assigned to the executor in the resource offer.

By scrutinizing its description, *Symbiosis* attempts to determine if the selected task is a network-bound task, with the hope that a computation-bound task can be found to co-locate with it on the same executor, referred to as a *symbiotic* task henceforth in this paper. To look for such computation-bound symbiotic tasks, *Symbiosis* scans through all the pending tasks in the same application to see if any of them can achieve data locality on the executor in the resource offer. If so, these tasks are considered network-free with respect to the executor in the resource offer, and are good candidates as the symbiotic task. It then assigns one of these network-free tasks to the same executor, by accepting the same resource offer.

As shown in Fig. 3, after we discover T_{13} and T_{14} are network-bound tasks, we scan the pending tasks of Job_2 and find that T_{21}'s input data is on W_1, and T_{22}'s input is on W_2. Therefore, we assign T_{21} to E_1 and T_{22} to E_2, such that the corresponding cores are almost fully utilized. The timelines shown in Fig. 4 have clearly shown that with network-aware scheduling, job performance is improved since Job_2 is accelerated, while Job_1 is not slowed down.

But what if an existing task running on the executor turns out to be a computation-bound task? The instinctive reaction may be to find a network-bound task as its symbiotic task, sharing the same executor. However, a network-bound task may not be computation-free, since it is able to process the

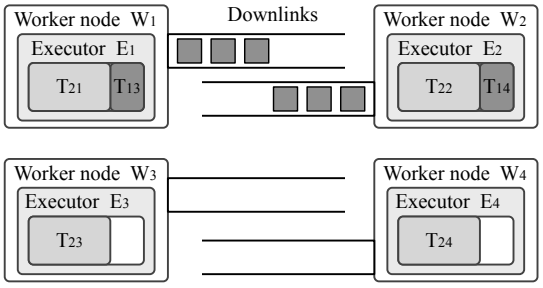


Fig. 3. Making a computation-bound task co-locate with a network-bound task corrects resource imbalance through improving CPU utilization: T_{21} and T_{13} utilize E_1's resources, and T_{22} and T_{14} utilize E_2's resources.

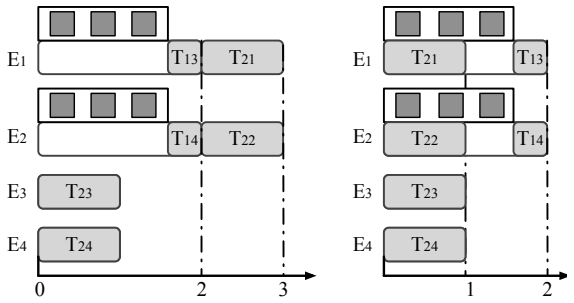


Fig. 4. Job performance without (left) and with (right) network-awareness: the completion time of Job_1 remains 2 time units, whereas the completion time of Job_2 decreases since its map stage decreases from 3 to 1 time unit.

data it has received so far while receiving the remaining input data. In addition, network bandwidth is usually shared across all the executors in the same worker node, without the enforced isolation that is adopted with CPU cores. For these reasons, it is not always beneficial if a network-bound task is chosen as a symbiotic task to a computation-bound one. We leave this as a configurable option; once enabled, *Symbiosis* scans through all existing tasks running on the same worker node as the executor in the resource offer. If all these tasks are found to incur no network transfer, it selects and assigns *one* network-bound task to the executor in the resource offer.

C. Selecting Symbiotic Tasks

As we have discussed previously, for each executor predicted to be resource unbalanced, there might exist multiple corresponding network-free tasks competing for the same symbiotic opportunity on the executor. On the other hand, since a framework might be offered multiple executors where a task can achieve data locality, there might exist multiple symbiotic opportunities for a single task on different executors. It is thus necessary to coordinate among multiple candidate tasks and multiple symbiotic opportunities to best improve the system performance.

Given a set of resource unbalanced executors and a set of candidate symbiotic tasks, the problem of selecting the best symbiotic tasks to maximize the aggregated performance gain can be formally formulated as below:

$$\max \quad \sum_j \sum_i p_i \cdot x_{ij} \tag{1}$$

$$\text{subject to} \quad \sum_j x_{ij} \leq 1, \tag{2}$$

$$\sum_i p_i \cdot x_{ij} \leq \text{transfer}_j, \forall j \tag{3}$$

where $x_{ij} \in \{0, 1\}$ indicates whether a candidate symbiotic task t_i with respect to the executor e_j is distributed to e_j; and p_i is the processing time of t_i on a CPU core. transfer_j represents the time to transfer the input to the network-bound task on e_j. Since the network-bound task distributed to e_j has a higher priority over the candidate symbiotic tasks, Constraint (3) regulates that all the symbiotic tasks must finish before the network-bound task receives all its input data. This is a *Multiple Knapsack Problem* [24], which can be solved through an existing polynomial time approximation scheme [25] if we have *a priori* knowledge of transfer_j and p_i.

Nevertheless, estimating transfer times not only requires monitoring the dynamic flow rates with extra overhead, but also needs to modify application interfaces to reveal the flow sizes in advance. Furthermore, accurately estimating task processing times is almost impossible for online schedulers.

Instead of using such selection strategies that require additional monitoring and modifications to application interfaces, in *Symbiosis*, we propose to select one task at a time based on the predefined priorities (line 6-9 of Alg. 1). We prioritize tasks belonging to a high priority job, and for tasks within the same job, a process-local task is preferred. *Symbiosis* continues to allocating network-free tasks to the executor until the corresponding network-bound task is ready to compute (line 16 of Alg. 1). In this way, as the network-bound task starts its computation phase, at most one task would compete with it for CPU cycles, without slowing it down significantly.

Algorithm 1 Network-aware task scheduling

```
1:  procedure NETWORK-FREE TASKS(e_j)
2:      Initiate: Ω_j = Φ, Σ = the set of pending tasks
3:      for t ∈ Σ do
4:          if e_j ∈ LOCALITYQUERY(t) then
5:              Ω_j = Ω_j + {t}
6:  procedure PRIORITY-BASED SELECTION(e_j)
7:      Ω_j = NETWORK-FREE TASKS(e_j)
8:      Π = SORTTASKS(Ω_j)
9:      Distribute t_Π(0) to e_j and start processing t_Π(0)
10: procedure MAIN
11:     Initiate: the newly received resource offer (c_i, e_j)
                ▷ the (number of cores, offering machine) pair
12:     t_0 = TASKSCHEDULER(e_j)
13:     if t_0 is not empty then
14:         Accept the offer and distribute t_0 to e_j
15:     if e_j ∉ LOCALITYQUERY(t_0) then ▷ network-bound
16:         while t_0 is receiving do ▷ add symbiotic tasks
17:             PRIORITY-BASED SELECTION(e_j)
```

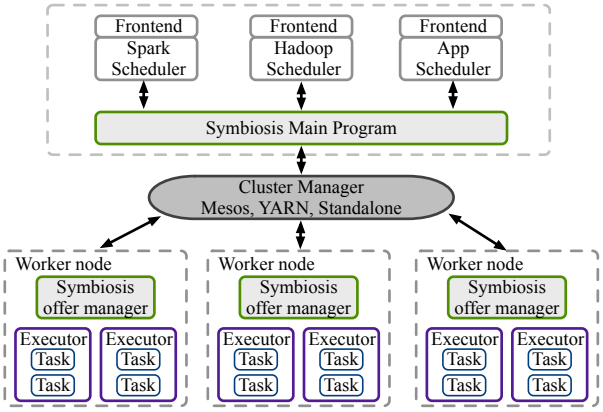


Fig. 5. *Symbiosis* in a multi-framework cluster.

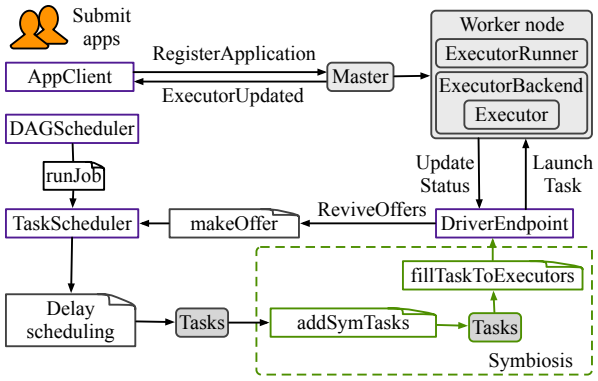


Fig. 6. *Symbiosis* on top of Spark: *Symbiosis* works as a scheduling plugin, requiring no modifications to existing application interfaces.

Whenever a task finishes, the executor will inform the cluster manager of this event, which triggers a new resource offer to a framework. The scheduler in the framework, unaware of the existence of *Symbiosis*, will select a new task to be launched on the executor. When a symbiotic task finishes, if the cluster runs through the same procedure, it is possible that the newly selected task is network-bound, which aggravates the already busy network on the executor. To avoid such situations, we block all the messages to the cluster manager that indicate the completion of symbiotic tasks on their corresponding executors. Instead, whenever a symbiotic task finishes, a message is explicitly sent to *Symbiosis* to select and launch another symbiotic task. We treat the original task and the symbiotic task as a whole, and release their allocated resources only after both of them finish. The logic of *Symbiosis* is shown in Algorithm 1.

D. Symbiosis in a Multi-Framework Cluster

The design of *Symbiosis* is complementary to the functionality provided by per-framework task schedulers and cluster resource managers, and can be easily deployed in a multi-framework cluster, as shown in Fig. 5. Each framework has a long-running front-end to receive high-level user requests such as online search queries and data analytic requests. The cluster manager offers available resources to a framework as *resource offers*, using a strict priority or weighted fair sharing strategy. The framework depends on its own scheduler to distribute its ready tasks (whose upstream tasks have all finished) to different executors in the resource offer.

Symbiosis incrementally improves such a hierarchical scheduling design, by adding a plugin to each application in different frameworks. Each application adopts its own customized scheduling strategies to place tasks onto executors in the resource offer. *Symbiosis* examines these placement decisions to identify if there exists a lack of resource balance, and adjusts these placement decisions by adding a symbiotic task to each executor without resource balance. *Symbiosis* also introduces an *offer manager* on each worker node, in order to control when to send a message to the cluster manager to claim that an executor is ready to accept new tasks. *Symbiosis* is designed to be flexible and completely transparent, in that it does not rely on the scheduling policies used in each framework, or require any extra user input or modifications to application interfaces. These design decisions have made it more practical, readily deployable in production clusters.

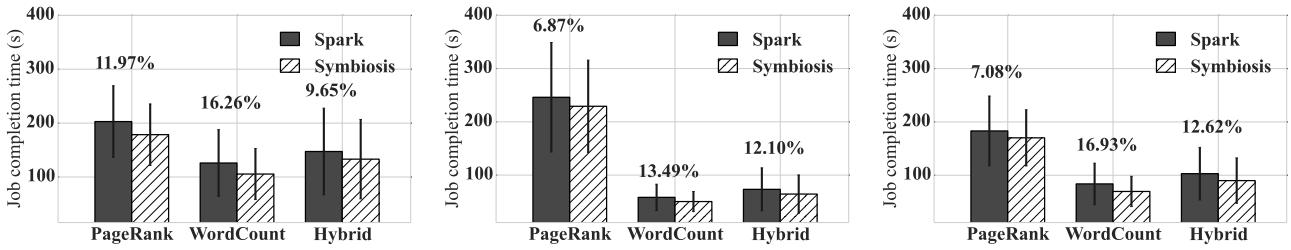
IV. *Symbiosis*: System Implementation

We have built *Symbiosis* on top of Spark 1.4.0 [26] as a scheduling plugin in the Scala programming language, as shown in Fig. 6.

When a user submits an application to Spark, a `SparkContext` is generated for this application. An application `Client` in `SparkContext` then registers the newly submitted application to the master, which then allocate executors to this application. Once executors are allocated to the application, a `Driver` program can directly exchange messages with executors to acquire their up-to-date information.

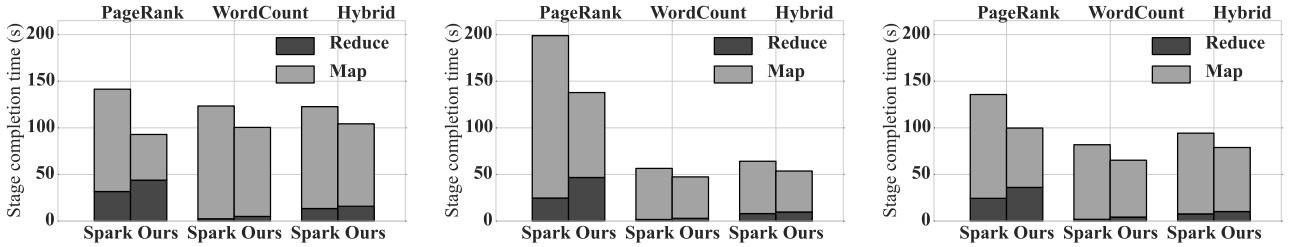
After initiation, the application's `DAG scheduler` first divides jobs into different stages of tasks and then calls the `task scheduler` to submit the tasks in the ready stage. Meanwhile, submitting tasks invokes the `makeOffers` function to pass the resource offers from executors to the application. For each resource offer, the `task scheduler` sorts all the ready tasks based on the their locality level on the corresponding executor. Spark allocates a fixed number of cores on an executor to each task. Tasks can continue to accept an offer if the relevant executor still has enough available cores.

Symbiosis steps in when the task scheduler has selected tasks to accept the resource offers. It examines whether there exists a `Shuffle` task or an input task that is `RACK_LOCAL` or `ANY` with respect to the corresponding executor. Once found, it scans through all the pending input tasks with data blocks on the offering executor, (*i.e.*, `PROCESS_LOCAL` or `NODE_LOCAL` tasks) in the `addSymTasks` module. It then finds executors to submit these symbiotic tasks to, and asks the `Driver` to launch both the symbiotic tasks and the original selected tasks to the executors through the `fillTaskToExecutors` module. *Symbiosis* only selects one symbiotic task for each network-



(a) Cluster size = 25 (b) Cluster size = 50 (c) Cluster size = 100

Fig. 7. Average job completion times for different workloads under three cluster sizes: jobs are expedited by more than 6% in all the experiments.



(a) Cluster size = 25 (b) Cluster size = 50 (c) Cluster size = 100

Fig. 8. The completion times of map and reduce stages for different workloads under all the three clusters. Compared with Spark, *Symbiosis* significantly speeds up the map stages due to early launching symbiotic tasks. In contrast, reduce stages are slightly slowed down due to the potential competition between a symbiotic task and the reduce task on the same executor.

bound task to share the same executor. In the worst case, the symbiotic task may still be running and competes with the network-bound task, after the latter enters the computation phase. Though unfortunate and difficult to avoid, such competition for CPU only lasts for the remaining duration of the symbiotic task.

V. Evaluation

We evaluate *Symbiosis* through a series of experiments on the Linode[15] cloud hosting platform. We begin with three different types of workloads that evaluate how *Symbiosis* behaves under different settings and go on to analyze the four key factors that determines the performance of *Symbiosis*.

A. Setup

The master of our cluster runs on an instance with 20 CPU cores and 96 GB of memory. Each worker node in our cluster is equipped with 8 cores, 16GB of memory and 384GB SSD storage. The bandwidth limit for uplink is 2000Mbps whereas downlink bandwidth is as large as 40Gbps.

We have designed a customized module based on the deployment tool, Ansible [27], to launch our cluster and uniformly configure the worker nodes. We run Spark 1.4.0 together with Hadoop 2.6 in the cluster. The size of each data block is 128 MB as seen in the Facebook's production cluster [6]. Two executors are launched on each node to run tasks.

To comprehensively evaluate the performance of *Symbiosis*, we run two types of popular analytics workloads, WordCount and PageRank [28] on a 32 GB Wiki dump [29] that tracks all internal links in the Wiki [30]. The jobs in the WordCount workload computes the occurrence frequency of each word in the input data file. Such a job is a typical network-light MapReduce job which involves a small amount of network transfer between its tasks in map and reduce stages. Each WordCount job processes a 8 GB data file.

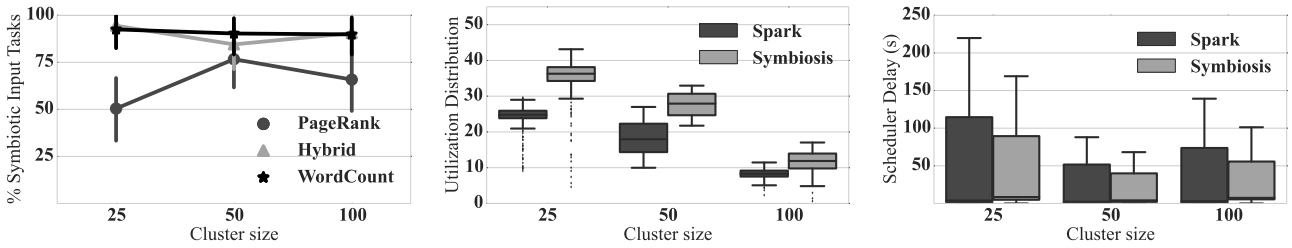
PageRank [28] is an iterative graph processing algorithm that is designed for Google's search engine. A job in the PageRank workload usually incurs a large amount of network traffic to transfer intermediate results among multiple stages. The PageRank is thus marked as network-heavy in our experiments. Each job in the PageRank workload processes a 1 GB data file extracted from the original Wiki dump.

In all the following experiments, three applications registers with the cluster manager and 30 jobs are submitted to each application. The submission schedule is generated to ensure the comparability among different rounds of experiments. The distribution of inter-arrival times follows an exponential distribution with a mean of 14 seconds as observed in the Facebook cluster [6].

B. Benefits of Symbiosis

In Fig. 7, we present the job completion times of the network-heavy PageRank workload, the network-light WordCount workload and the network-medium hybrid workload. The performance gains in terms of average job completion times vary from 6.87% to 16.93% in different clusters. More specifically, WordCount jobs benefit most from *Symbiosis* with the average performance gain equal to 15.56% on average.

By comparing the job completion times in Fig. 7 and the stage completion time in Fig. 8, we find that the improvement of job performance mainly stems from the shortened completion time of input tasks in the map stages. Essentially,



(a) Ratio of symbiotic tasks (b) Average CPU utilization (c) Scheduler delay

Fig. 9. (a) The ratio between symbiotic tasks and input tasks in each job; (b) the CPU utilization under the three clusters; (c) the scheduler delay under *Symbiosis* and Spark.

symbiotic tasks are all input tasks that achieve data locality and avoid network transmissions. Without *Symbiosis*, it is possible that a candidate symbiotic task waits too long to be offered with an executor that stores its data. Based on delay scheduling, it has to be launched onto an executor without its input data after a certain amount of time. In other words, symbiotic tasks not only are scheduled earlier but also achieves higher locality. As a result, the map stage is accelerated.

C. *Does* Symbiosis *come into effect?*

To validate that our scheduling policy contributes to *Symbiosis*'s performance gain, we first count the number of symbiotic tasks that actually share the executor with a network-bound task. In Fig. 9(a), we present the ratio between the number of symbiotic tasks and the total number of input tasks in a job. *Symbiosis* successfully schedules more than 90% of input tasks in a WordCount job as symbiotic tasks and launch them to executors. In contrast, PageRank jobs have relatively fewer symbiotic tasks under all the clusters. Consequently, the performance gain of PageRank is lower than WordCount as shown in Fig. 7. This is due to the multiple intermediate stages caused by iterations in PageRank jobs. Many tasks in the intermediate stages are reduce tasks that involve network transfers. However, by analyzing application logs we find that there are some process-local tasks that are launched onto the same executor immediately after the tasks in the previous iteration finishes. As a result, the symbiotic opportunities are not as many as in WordCount jobs.

D. *Is the CPU utilization improved?*

The motivation of *Symbiosis* is to maximally utilize the computation and network resources at the same time. To evaluate the influence of *Symbiosis* on CPU utilization, we sample the utilization of each core on a worker node every second. We can thus compute the average utilization of all the CPU cores in the cluster. The utilization distribution shown in Fig. 9(b) depicts the variation of the average cluster utilization throughout the application runtime. *Symbiosis* significantly improves CPU utilization in all the three clusters. In addition, we find that the average utilization decreases with the increasing number of workers in the cluster since the loads of the three clusters are similar.

E. *What is* Symbiosis*'s overhead?*

Since a symbiotic task will share the computation resources allocated to the corresponding network-intensive tasks, it is possible that the symbiotic task and the network-intensive task will compete for resources. We evaluate the influence of symbitoic tasks over reduce tasks in Fig. 8, which are the main network-intensive tasks in our workloads. From the results we can see that although reduce tasks are indeed slowed down under *Symbiosis*, such loss of performance is offset by the faster map tasks.

We further evaluate the scheduler delay of *Symbiosis*, which is the time period after a task is submitted to the system and before it is launched to an executor. Using *Symbiosis*, a symbiotic task that originally waits for executors can be launched to the executor hosting a network-bound task in advance. Its scheduler delay is thus shortened. We evaluate the scheduler delay of all tasks belonging to a WordCount job under different clusters in Fig. 9(c). It is clear to see that *Symbiosis* does not incur extra delay in comparison with Spark's standard scheduler. Furthermore, *Symbiosis* reduces the scheduler delay for tasks that wait for a long time in Spark.

VI. Related Work

Task schedulers. Zaharia *et al.* designed a Hadoop fair scheduler, HFS [6], to balance between inter-job fairness and data locality. The delay scheduling algorithm used in HFS makes jobs that should be scheduled next (according to fairness) wait for a small amount of time before it can launch a local task. Other jobs can thus launch tasks first and utilize the computation resources. This strategy has been adopted in the current version of Spark [26] due to its simplicity. Nevertheless, since it only considered data locality for input tasks, and neglected the influence of inevitable data shuffle, the resource imbalance addressed in *Symbiosis* cannot be handled efficiently.

To improve the scalability of a centralized scheduler, Ousterhoud *et al.* designed Sparrow [7], a sampling-based scheduling framework that randomly probes available worker nodes and selects less loaded nodes to launch tasks. Leveraging late binding, tasks can dynamically adjusts their initial choices and thus experience the minimum amount of waiting time. In production clusters, however, the per-application task scheduler barely becomes the performance bottleneck: CPU is often the

scarce resource [31]. Without network-awareness, Sparrow not only failed to achieve data locality, but also cannot utilize the idle CPU resources incurred by network transmission.

Venkataraman *et al.* further designed a data-aware scheduling framework, KMN [8], to minimize the time taken by input tasks to read datasets. KMN also reduced the time taken by the data shuffle, by launching additional tasks that avoid congested links and by co-locating upstream and downstream tasks. In contrast, with *Symbiosis*, we have improved cluster utilization regardless of the placement of tasks. As long as network transmissions still exist between upstream and downstream tasks, the CPU cores allocated to the reduce tasks are mostly idle. By enabling network-free tasks to share the resource offer of a network-intensive task, we have effectively improved resource utilization, and shortened job completion times.

Cluster managers. Resource utilization in a cluster highly depends on the inter-framework resource allocation policy embraced in the cluster manager. Mesos [16] introduced a two-level scheduling scheme to decide how many resources to offer each framework. Schwarzkopf *et al.* [32] argued that Mesos and other existing cluster managers [17] lacked sufficient resource visibility. Instead, they designed a new scheduler architecture using shared state and optimistic concurrency. Although our task scheduler is built upon resource offers in two-level cluster managers, it can be easily extended to shared state managers. When a scheduler successfully claims a resource combination for a network-intensive task, our scheduler can immediately decides whether to select other computation-intensive tasks to share the resources.

Multi-resource scheduling. Theoretical work in multi-resource scheduling [12], [13], [18] only deals with offline job scheduling through complicated graph or combinatoric algorithms. Their approaches cannot be applied to our online scheduling scenario since the demand for network and computation resources cannot be quantized in advance.

VII. Conclusion

In this paper, we have designed, analyzed and evaluated a network-aware task scheduler, *Symbiosis*, designed to improve resource utilization and job performance at the same time. The upshot of *Symbiosis* lies in its ability to identify and correct unbalanced utilization of multiple resources at runtime. By selecting computation-intensive tasks to share the resource offered to a network-intensive task, computation and network resources can be fully utilized simultaneously. Our experiments on the Linode cluster demonstrate that *Symbiosis* effectively reduces job completion times and improves resource utilization without incurring extra scheduling delay. Furthermore, its compatibility with existing parallel computing frameworks makes it ready to be deployed in production datacenters.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. USENIX OSDI*, 2004.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD*, 2010.

[3] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[4] 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 *Proc. USENIX NSDI*, 2012.

[5] "Apache Flink," https://flink.apache.org/.

[6] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. European conference on Computer systems (Eurosys)*, 2010, pp. 265–278.

[7] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 69–84.

[8] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proc. USENIX OSDI*, 2014, pp. 301–316.

[9] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *Proc. ACM SIGCOMM*, 2014, pp. 443–454.

[10] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. ACM SIGCOMM*, 2014.

[11] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM SIGCOMM*, 2015, pp. 393–406.

[12] J. H. Patterson, "A comparison of exact approaches for solving the multiple constrained resource, project scheduling problem," *Management science*, vol. 30, no. 7, pp. 854–867, 1984.

[13] J. P. Stinson, E. W. Davis, and B. M. Khumawala, "Multiple resource–constrained scheduling using branch and bound," *AIIE Transactions*, vol. 10, no. 3, pp. 252–259, 1978.

[14] E. Demeulemeester and W. Herroelen, "A branch-and-bound procedure for the multiple resource-constrained project scheduling problem," *Management Science*, vol. 38, no. 12, pp. 1803–1818, 1992.

[15] "Linode: SSD cloud hosting," http://www.linode.com/.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX NSDI*, 2011.

[17] "Apache Hadoop NextGen MapReduce (YARN)," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[18] W. Leinberger, G. Karypis, and V. Kumar, "Job scheduling in the presence of multiple resource requirements," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 1999, p. 47.

[19] C. Hopps, "Analysis of an equal-cost multi-path algorithm," United States, 2000.

[20] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX NSDI*, 2010, pp. 19–19.

[21] "Hadoop NameNode," http://wiki.apache.org/hadoop/NameNode.

[22] "Docker: an open platform for distributed applications for developers and sysadmins," https://www.docker.com/.

[23] "Spark standalone mode," http://spark.apache.org/docs/latest/spark-standalone.html.

[24] "Multiple Knapsack Problem," http://en.wikipedia.org/wiki/List_of_knapsack_problems#Direct_generalizations.

[25] C. Chekuri and S. Khanna, "A polynomial time approximation scheme for the multiple knapsack problem," *SIAM Journal on Computing*, vol. 35, no. 3, pp. 713–728, 2005.

[26] "Apache Spark," https://github.com/apache/spark.

[27] "Ansible is simple IT automation," http://www.ansible.com/.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank citation ranking: bringing order to the Web.* Stanford InfoLab, 1999.

[29] "Wikimedia downloads," http://dumps.wikimedia.org/.

[30] "Pagelink tables," http://www.mediawiki.org/wiki/Manual:Pagelinks_table.

[31] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. USENIX NSDI*, 2015, pp. 293–307.

[32] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. ACM European Conference on Computer Systems (EuroSys)*, 2013.