Data Locality in MapReduce: A Network Perspective

Weina Wang and Lei Ying
School of Electrical, Computer and Energy Engineering
Arizona State University, Tempe, AZ 85287
{weina.wang, lei.ying.2}@asu.edu

***Abstract*—In MapReduce, placing computation near its input data is considered to be desirable since otherwise the data transmission introduces an additional delay to the task execution. This data locality problem has been studied in the literature. Most existing scheduling algorithms in MapReduce focus on improving performance through increasing locality. In this paper, we view the data locality problem from a network perspective. The key observation is that if we make appropriate use of the network to route the data chunk to the machine where it will be processed in advance, then processing a remote task is the same as processing a local task. In other words, instead of bringing computation close to data, we can also bring data close to computation to improve the system performance. However, to benefit from such a strategy, we must (i) balance the tasks assigned to local machines and those assigned to remote machines, and (ii) design the routing algorithm to avoid network congestion. Taking these challenges into consideration, we propose a scheduling/routing algorithm, named the Joint Scheduler, which utilizes both the computing resources and the communication network efficiently. To show that the Joint Scheduler has superior performance, we prove that the Join Scheduler can support any load that can be supported by some other algorithm, i.e., achieve the maximum capacity region. Simulation results demonstrate that with popularity skew, the Joint Scheduler improves the throughput significantly (more than 30% in our simulations) compared to the Hadoop Fair Scheduler with delay scheduling, which is the de facto industry standard. The delay performance is also evaluated through simulations, where we can see a significant delay reduce under the Joint Scheduler with moderate to heavy load.**

I. Introduction

The MapReduce framework [1] has been widely deployed in large computing clusters for the growing need of big data analysis. Hadoop [2] is one of the most popular implementations of the MapReduce framework and has been adopted by various organizations.

MapReduce are implemented on top of distributed file systems such as the Google File System (GFS) [3] and the Hadoop Distributed File System (HDFS) [4], which divide large datasets into data chunks and store multiple replicas of each chunk on different machines. MapReduce jobs are submitted to request data processing and each job is divided into a number of *map* tasks and *reduce* tasks. A map task reads one data chunk from the distributed file system and processes it to generate intermediate results. Reduce tasks fetch these intermediate results and conduct further computations to get the final results.

Map tasks and reduce tasks are assigned to machines according to the scheduling algorithm. During task scheduling, an important consideration is to place computation near data, i.e., to assign a task on or close to the machine that stores its input data on local disks. This is commonly referred to as the *data locality* problem. The data locality problem is particularly crucial for map tasks since they read data from the distributed file system and map functions are data-parallel. Besides, according to an empirical trace study from a production MapReduce cluster [5], majority of jobs are map-intensive, and many of them are map-only. Therefore we focus on data locality in map task scheduling algorithms and assume that reduce tasks are not the bottleneck of the job processing or the communication network. Data locality has been studied intensively in the literature [1], [6]–[15]. Most existing scheduling algorithms optimize data locality in certain ways in order to achieve better efficiency. Detailed discussions of the related work and comparisons with our work are provided in Section II.

We call a machine a *local machine* for a map task if it has the input data chunk of this task on its local disks, and we call this map task a *local task* on this machine; otherwise, the machine is called a *remote machine* for this map task and this map task is called a *remote task* on the machine. We also use the term *locality* to refer to the fraction of tasks that are executed on local machines. Previously, launching a map task on a remote machine is usually considered to be inefficient since the machine needs to first retrieve the input data from other machines through the communication network before processing it, which introduces an additional delay to the task execution. So local and remote tasks are often modeled with different processing time.

This view of data locality in MapReduce is arguable. If the communication network that connects the machines had infinite capacity and could transfer data instantaneously, there would be no difference between assigning a task to its local machines or to other machines. Thus the time of processing a remote task depends on the capacity of the communication network and the scheduling algorithm that allocates tasks. If *data can be routed in advance so that machines do not spend time on waiting for input data before executing tasks*, then even though the network capacity is finite, we can still achieve the same throughput as if all tasks are local. Inspired by this intuition, we study the data locality problem from a

network perspective beyond just abstracting the effect of the communication network as a "longer processing time". We focus on characterizing fundamental limits on the capacity of a MapReduce cluster with network constraints and designing optimal scheduling algorithms.

To optimize the performance, we face the following challenges: *how to strike the right balance local and remote tasks*, and *how to route the traffic in the network appropriately to avoid congestion.* Failure to meet these challenges may result in slow data transmission and waste of computing resource and may even harm the throughput performance. These challenges are more pronounced when data are not ideally uniformly distributed across the cluster, in which case placing all the computation near data results in heavy-congestion on some machines while leaving other machines lightly utilized or idle.

In this paper, we first quantify the capacity region of a MapReduce cluster with communication network constraints. Then we develop scheduling and routing algorithms by taking the above challenges into consideration, called the Joint Scheduler, and prove that it can support any load that is in the capacity region. We will refer to it as JS in this paper. Our contributions are summarized as follows.

- We advocate studying the data locality problem from a network perspective. The efficiency of a MapReduce cluster can be improved not only by placing tasks close to data, but also by transferring data in advance to machines where the tasks will be executed. Joint task scheduling and routing, with communication network constraints taken into consideration, should be designed to fully exploit the system capacity.
- We consider a queueing model for the communication network in a MapReduce cluster to capture the bandwidth constraints. Based on this model, we propose a map task scheduling/routing algorithm, which assigns newly arrived tasks using join the shortest queue policy, and routes tasks in the communication network also using join the shortest queue policy, but with blocking under some conditions.
- We characterize the capacity region of a MapReduce cluster with data locality and communication network constraints, where the capacity region consists of all the arrival rate vectors for which there exists a scheduling/routing algorithm that stabilizes the system. Then we prove that the proposed Joint Scheduler stabilizes any arrival rate vector strictly within the capacity region, i.e., it is throughput optimal. The proposed algorithm utilizes the computing resources efficiently by balancing the tasks assigned to local machines and those assigned to remote machines, and explores the communication network capability by balancing the traffic load to avoid congestion.

II. Related Work

Data locality in MapReduce has received a lot of attention in designing scheduling algorithms [6]–[9, 16]. Some widely used schedulers, including the default FIFO scheduler [16], the Fair Scheduler for Hadoop [6], and Quincy for Dryad [7], try to improve the performance by increasing locality, i.e., by launching more tasks on local machines. However, they all use the request-and-wait procedure to obtain input data before processing the task. As we argued, this may lead to underutilization on both the computing resources and the communication network capability.

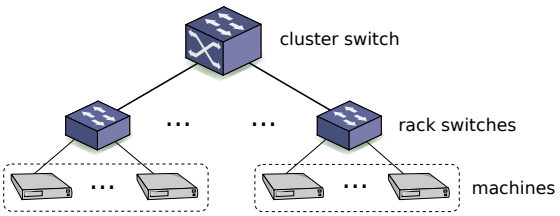


Fig. 1: Hierarchical network architecture.

Most related to our work is the prefetching idea in [10]. However, the prefetching scheme developed in this paper puts little effort on load balancing among machines and congestion control for the network. The decision of which data chunks need to be prefetched and where they are prefetched to is not closely coupled with the current work loads and network traffic.

Although we focus on addressing the data locality problem within the MapReduce framework, we note that approaches from the distributed file system side have also been proposed. The authors of [11, 12] exploit the variance in data popularity and access patterns and present algorithms named Scarlett and DARE, respectively, to replicate data chunks based on their data access patterns. These algorithms share the same spirit with ours in that we both make popular data accessible to more machines.

III. System Model

We consider a MapReduce computing cluster with M machines, indexed as $1, 2, \cdots, M$. The cluster adopts a multi-level hierarchical network architecture depicted in Fig. 1, where machines are grouped into R racks of machines at the lowest level, and one or more levels of aggregation switches connects the racks. This hierarchical network architecture is commonly used by data centers [1, 6, 17, 18].

Jobs come in stochastically, and when a job comes in, it brings a random number of map tasks, which need to be served by the machines. We assume that data chunks have the same fixed size (e.g., by default 64MB in Hadoop). Each data chunk is replicated and placed at three different machines. Therefore, each task is associated with three local machines. When a task is launched on a non-local machine, the machine cannot start processing the task until necessary data arrives. According to the associated local machines, tasks can be classified into *types* denoted by

$$\vec{L} = (m^1_{\vec{L}}, m^2_{\vec{L}}, m^3_{\vec{L}}),$$

where $m^i_{\vec{L}}, i = 1, 2, 3$ are the indices of the three local machines. For example, if the data chunk associated with a task is stored at machines 1, 21 and 23 then $\vec{L} = (1, 21, 23)$. We assume machine $m^1_{\vec{L}}$ is in rack $r^1_{\vec{L}}$, and machine $m^2_{\vec{L}}$

and $m^3_{\vec{L}}$ are in the same rack $r^2_{\vec{L}}$ according to Hadoop's data replication policy [16].

The following notations are used throughout this paper. We use $\mathcal{M}_r$ to denote the set of machines in rack r, and r_m to denote the index of the rack that machine m is in. For each task type $\vec{L}$, the set of local machines is

$$\mathcal{M}_{\vec{L}} = \{m^1_{\vec{L}}, m^2_{\vec{L}}, m^3_{\vec{L}}\}.$$

For each machine m, the set of local tasks is

$$\mathcal{L}_m = \{\vec{L} \in \mathcal{L} \colon m \in \mathcal{M}_{\vec{L}}\}.$$

A. Arrivals and Service

We consider slotted time. Let $A_{\vec{L}}(t)$ denote the number of type $\vec{L}$ tasks arriving at the beginning of time slot t. We assume that this arrival process is temporally i.i.d. with arrival rate $\lambda_{\vec{L}}$. Furthermore, we assume that the arrival processes are bounded processes with $\sum_{\vec{L}} A_{\vec{L}}(t) \leq C_A$ for some constant C_A. Let $\lambda = (\lambda_{\vec{L}} \colon \vec{L} \in \mathcal{L})$ be the arrival rate vector, where $\mathcal{L}$ is the set of task types with arrival rates greater than zero; i.e., $\mathcal{L} = \{\vec{L} \colon \lambda_{\vec{L}} > 0\}$.

Machine m starts to serve a task after it gets the corresponding data. A task served by machine m in time slot t completes service with some probability φ. Specifically, we consider a binary variable $S_m(t)$ and a task served by machine m during time slot t completes service if $S_m(t) = 1$; otherwise it remains in the system. This assumes a geometric distributed service time.

B. Network Queueing Model

In the considered hierarchical network architecture, a set of machines are mounted within a rack and interconnected by a rack switch. These rack switches have a number of uplink connections to the cluster switch, which can use 1-Gbps or 10-Gbps links. For economic considerations, the design of the network connections usually introduces oversubscription since all the interrack traffic needs to go through the cluster switch. For example, in the network that 40 servers in the same rack connect to the rack switch by 1-Gbps ports, rack switches may have between four and eight 1-Gbps uplinks to the cluster switch, corresponding to an oversubscription factor between 5 and 10 for communication across racks [18]. Therefore the cluster-level bandwidth resource is relatively scarce compared with the rack-level.

Data transmission in the network consumes bandwidth. Since each machine connects to the network through the rack switch, the bandwidth between the machine and the rack switch constrains the incoming and outgoing data transmission rates of the machine. When there is a large amount of data that needs to be sent or received by a machine, the unfinished data will be backlogged at some queues. *Let this constraint on the incoming and outgoing data transmission rates of each machine be B_1 data chunks per time slot.* For the interrack traffic, due to oversubscription, the machines in one rack cannot communicate with machines in other racks at their full bandwidth simultaneously. *There are constraints on the overall incoming and outgoing data transmission rates of the rack. Let this constraint be B_2 data chunks per time slot.* This rack bandwidth is shared by machines in the same rack.

Based on the network hierarchy, the communication network in the cluster is modeled as depicted in Fig. 2. For each machine m, $Q_{m,1}$ and $Q_{m,2}$ are the queues for the outgoing traffic and incoming traffic of the machine, respectively. Therefore, at most B_1 data chunks depart from each queue during one time slot. Similarly, for each rack r, $X_{r,1}$ and $X_{r,2}$ are the queues for the outgoing traffic and incoming traffic of the rack, respectively, and at most B_2 data chunks depart from each queue during one time slot. We call $Q_{m,1}$ and $Q_{m,2}$ the *machine outgoing queue* and the *machine incoming queue*, and we call $X_{r,1}$ and $X_{r,2}$ the *rack outgoing queue* and the *rack incoming queue*. Other labels and queues in Fig. 2 will be described in later sections. Machine m communicates with machine a in the same rack through the path $(Q_{m,1}, Q_{a,2})$. An example is shown in Fig. 2 for $m = 1, a = 2$ with corresponding path $(Q_{1,1}, Q_{2,2})$. Machine m communicates with machine b in another rack through the path $(Q_{m,1}, X_{r_m,1}, X_{r_b,2}, Q_{b,2})$, as shown in Fig. 2 for $m = 1, b = 20$ with path $(Q_{1,1}, X_{1,1}, X_{r,2}, Q_{20,2})$. Since each map task is associated with a data chunk and the network is used to transfer the data chunks to be processed non-locally, B_1 and B_2 can also be viewed as number of tasks per time slot. The phrases transmitting tasks and transmitting data are used interchangeably in the context of communication. The lengths of the queues are counted as the number of tasks that have input data chunks in the queues.

Remark 1. We assume without loss of generality that B_1 is larger than or equal to 1 since otherwise we can rescale the duration of the time slot. The rack switches usually use 1-Gbps uplinks. This transmission rate is usually larger than the data processing rate performed by map functions. So we assume a service rate φ smaller than 1 after rescaling. The rate B_2 is larger than B_1 and smaller than $R \cdot B_1$ due to oversubscription.

IV. Map Task Scheduling/Routing

The task scheduling/routing problem is to assign incoming tasks to machines and route tasks in the communication network. Due to data locality, the task scheduling/routing algorithm can significantly affect the performance of the system. If a task scheduling/routing algorithm assigns too many tasks to remote machines or route tasks using improper balancing, data transmission will take up significant time and even congest the communication network. To achieve high performance, a task scheduling/routing algorithm needs to make fully use of both the computing resources and the network resources.

A. Data Locality

In task scheduling/routing, an important consideration is to place computation near the data, i.e., to place a map task on or close to the machine that stores the required data chunk. This makes the processing of tasks more efficiently since the network bisection bandwidth is much smaller than the bandwidth of local disks. Existing task scheduling algorithms

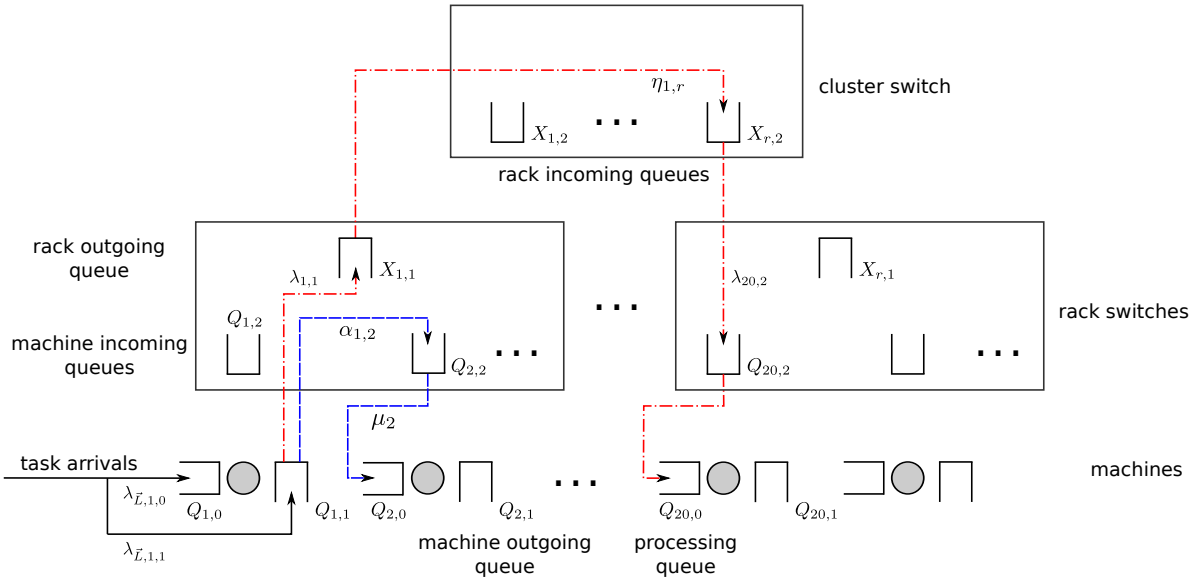


Fig. 2: Network queueing model.

exploit these bandwidth differences and make efforts to increase data locality. For example, the Hadoop Fair Scheduler [6] improves locality by slightly relaxing fairness. However, it is not always possible to launch a task on its local machine. When concurrent tasks access data uniformly from all the machines, most machines can easily find a local task to serve and very few machines need to retrieve data from other machines, bringing about a high throughput. However, trace study on an production MapReduce cluster shows a wide variance in data popularity and access patterns [11]. Some data exhibits high access concurrency. In this case, some tasks have to be launched on non-local machines and the corresponding data chunks need to spend time on going through queues in the communication network.

B. Task Scheduling/Routing Algorithm

Rethinking of the data locality problem we find that the low efficiency of non-local task processing is due to the lack of the foresight to route data in advance. The work flow of processing a non-local task in existing algorithms is as follows: when a machine m is idle, the scheduler assigns a task to it; if the data associated with the assigned task is not stored locally, machine m requests the data from another machine and then that machine sends the data through the communication network to machine m. Therefore, the system does not start routing a data chunk until some idle machine requests it. When an idle machine is assigned to serve a non-local task, it remains idle until the necessary data arrives. This actually underutilizes the computing resources and leaves machines not work conserving in some sense. If we route some of the data chunks in advance, machines can process the ready tasks while waiting for the data of future tasks, which makes the machines more work conserving. Routing in advance also makes better use of network resources in the sense that communication network does not need to wait for data requests to start routing.

Based on this insight, we propose a new algorithm that performs task scheduling and routing jointly. We call this scheduler the Joint Scheduler, which is referred as JS in the rest of this paper. This algorithm includes two parts: the first part assigns tasks to some machines to serve or to the communication network to transmit as tasks arrive at the cluster; the second part routes the tasks in the communication network. Before we describe our algorithm in detail, we first elaborate our model of the cluster.

We have derived the queueing model of the communication network in Section III-B. The architecture of the processing queues in JS is very simple. The scheduler maintains a queue $Q_{m,0}$ for each machine m to buffer the tasks that have data stored on machine m or have data arrived at machine m from other machines. These tasks are ready to be served and the service process has been introduced in Section III-A as $\{S_m(t), t \geq 0\}$.

For each queue Q in the communication network, the set of queues that can receive data from Q is called the set of *candidate destinations* of Q and is denoted by $\mathcal{D}(Q)$. These sets represent the connectivity of the system. We describe this candidate destination set for each queue as follows.

- For each machine m, the machine outgoing queue $Q_{m,1}$ can send data to the rack outgoing queue and to the machine incoming queues in the same rack, so
$$\mathcal{D}(Q_{m,1}) = \{X_{r_m,1}\} \cup \{Q_{m',2} \mid m' \in \mathcal{M}_{r_m}\}.$$
- The machine incoming queue $Q_{m,2}$ can only send data to the task queue $Q_{m,0}$, so
$$\mathcal{D}(Q_{m,2}) = \{Q_{m,0}\}.$$
- For each rack r, the rack outgoing queue $X_{r,1}$ can send data to the rack incoming queues of all the racks, so
$$\mathcal{D}(X_{r,1}) = \{X_{r',2} \mid r' = 1, \cdots, R\}.$$

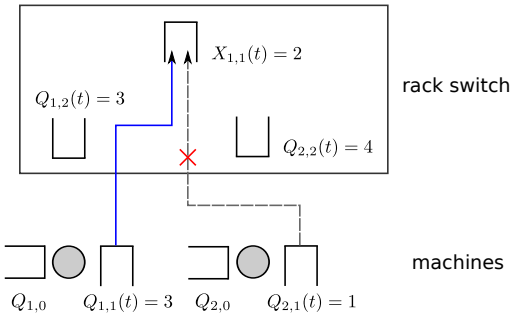


Fig. 3: A Toy Example for the Routing Algorithm in the Joint Scheduler.

- The rack incoming queue $X_{r,2}$ can send data to the machine incoming queues in its own rack, so

$$\mathcal{D}(X_{r,2}) = \{Q_{m,2} \mid m \in \mathcal{R}_r\}.$$

These connections are illustrated in Fig. 2. Now we present the Joint Scheduler as follows.

- **Task Assignment for New Tasks.** When a type $\vec{L}$ task comes in, the scheduler assigns it to the shortest queue in $\mathcal{Q}_{\vec{L}} = \{Q_{m,i} \mid m \in \mathcal{M}_{\vec{L}}, i = 0, 1\}$. Note that if a task is assigned to $Q_{m,0}$, it means that the task will be processed at local machine m; if it is assigned to $Q_{m,1}$, it means machine m needs to transmit the data chunk associated with the task to another (remote) machine to process.
- **Routing in the Communication Network.** For each queue in the communication network, we use a join the shortest queue algorithm with blocking to route tasks: each queue Q in the communication network first finds the shortest queue in the candidate destination set $\mathcal{D}(Q)$, denoted by D_Q^t; then it compares the queue length $D_Q^t(t)$ with $Q(t)$. If $D_Q^t(t) < Q(t)$, Q sends data to D_Q^t; otherwise Q does not send any data. If Q is the machine incoming queue or machine outgoing queue, then by the bandwidth constraint it can send at most B_1 data chunks during each time slot; similarly, if Q is the rack incoming queue or rack outgoing queue, then it can send at most B_2 data chunks during each time slot according to the bandwidth constraint.

  A Toy Example. We use the rack in Fig. 3 to illustrate the procedure of this routing algorithm. The candidate destination set of $Q_{1,1}$ is

$$\mathcal{D}(Q_{1,1}) = \{Q_{1,2}, Q_{2,2}, X_{1,1}\}.$$

  At time slot t, the shortest queue in this set is $X_{1,1}(t) = 2$, and since $Q_{1,1}(t) = 3 > X_{1,1}(t)$, $Q_{1,1}$ sends B_1 data chunks to $X_{1,1}$. For $Q_{2,1}$, the shortest queue in $\mathcal{D}(Q_{2,1})$ is also $X_{1,1}$. However, since $Q_{2,1}(t) = 1 < X_{1,1}(t)$, $Q_{2,1}$ blocks the outgoing traffic and does not send any data at the current time slot.

Remark 2. This algorithm balances the usage of the computing resources and the usage of the network resources by starting routing some of the data chunks as the corresponding tasks arrive. When a subset of the machines need to be accessed by many concurrent tasks, these tasks compete for a limited number of local machines. This algorithm assigns some of these tasks to the local machines and spreads other tasks to non-local machines in advance. Since these tasks cannot be all launched on local machines eventually, this foresight of routing reduces the waiting time of non-local machines and improves the throughput.

Remark 3. We focus on improving the throughput performance of a computing cluster and have not put much effort on delay performance for now. We aim at *achieving throughput optimality*. We have not integrated job-level scheduling into this scheduling algorithm either. This can be done through many ways under our framework for the purpose of achieving fairness or improving delay performance on the job level. For example, a simple way to achieve fairness is to use subqueues in the system. We can divide each queue into multiple subqueues according to the job that tasks come from; i.e., per job subqueues. Then we give the tasks from the jobs with fewer running tasks higher priority during scheduling. We can also use the remaining time to decide priority for better job delay performance. However, in this paper we just focus on the throughput performance and leave these choices for future work. We note that this paper is the first one that studies data locality from a network perspective and quantifies the fundamental limits due to communication network constraints.

C. Queue Dynamics

Scheduling decisions are made at the beginning of each time slot t and the service is performed after arrivals. For each queue Q in the system with arrival process A and service process S, the queue dynamics is expressed as

$$Q(t+1) = (Q(t) + A(t) - S(t))^+. \tag{1}$$

The arrivals of type $\vec{L}$ tasks are assigned to queues in $\mathcal{Q}_{\vec{L}}$. Let $A_{\vec{L},m,i}(t)$ with $m \in \mathcal{M}_{\vec{L}}$, $i = 0, 1$, denote the arrivals assigned to $Q_{m,i}$. Other arrivals in the system are internal arrivals, which are the departures of some other queues at the last time slot. Notations for those arrivals are as follows:

- $A_{m,1}$: arrivals coming from $Q_{m,1}$ to $X_{r_m,1}$;
- $A^Q_{m',m}$: arrivals coming from $Q_{m',1}$ to $Q_{m,2}$;
- $A^X_{r',r}$: arrivals coming from $X_{r',1}$ to $X_{r,2}$;
- $A_{m,2}$: arrivals coming from $X_{r_m,2}$ to $Q_{m,2}$;
- $A_{m,0}$: arrivals coming from $Q_{m,2}$ to $Q_{m,0}$.

Under the traffic constraints, the service processes of the queues in the communication network are defined as follows.

- For each m, the service processes of $Q_{m,1}$ and $Q_{m,2}$ are $\{S_{m,1}(t), t \geq 0\}$ and $\{S_{m,2}(t), t \geq 0\}$ defined as

$$S_{m,i}(t) = \begin{cases} B_1 & \text{if } Q_{m,i}(t) > D^t_{Q_{m,i}}(t), \\ 0 & \text{otherwise.} \end{cases}$$

- For each r, the service processes of $X_{r,1}$ and $X_{r,2}$ are $\{S^X_{r,1}(t), t \geq 0\}$ and $\{S^X_{r,2}(t), t \geq 0\}$ defined as

$$S^X_{r,i}(t) = \begin{cases} B_2 & \text{if } X_{r,i}(t) > D^t_{X_{r,i}}(t), \\ 0 & \text{otherwise.} \end{cases}$$

The service process of $Q_{m,0}$ is $\{S_m(t), t \geq 0\}$. Notice that the relations between internal arrivals and service are

$$S_{m,1}(t-1) = A_{m,1}(t) + \sum_{m' \in \mathcal{M}_{r_m}} A^Q_{m,m'}(t) \tag{2a}$$

$$S^X_{r,1}(t-1) = \sum_{r'} A^X_{r,r'}(t) \tag{2b}$$

$$S^X_{r,2}(t-1) = \sum_{m \in \mathcal{M}_r} A_{m,2}(t) \tag{2c}$$

$$S_{m,2}(t-1) = A_{m,0}(t). \tag{2d}$$

We assemble all the queue lengths into a vector

$$Z = (Q_{m,i}, X_{r,j} \colon m = 1, \cdots, M, i = 0, 1, 2, \\ r = 1, \cdots, R, j = 1, 2).$$

Then under the statistical assumptions we have made and the Joint Scheduler, the queueing process $\{Z(t), t \geq 0\}$ is a Markov chain. We assume that the state space $\mathcal{S}$ consists of all the states which can be reached from the zero vector.

Remark 4. This Markov chain $\{Z(t), t \geq 0\}$ is irreducible and aperiodic for the following reasons. For any state Z in the state space, since every queue length in the system is finite, the Markov chain can reach the zero state from Z within finite time slots when there are no arrivals. This probability is positive, so Z can reach the zero state and hence the Markov chain is irreducible. We can also see that the transition probability from the zero state to itself is positive, so the Markov chain is also aperiodic.

V. Throughput Optimality

In this section we prove that the Joint Scheduler achieves the maximum throughput. The system is said to be *stabilized* if the number of backlogged tasks $\{\Phi(t), t \geq 0\}$ satisfies

$$\lim_{C \to \infty} \lim_{t \to \infty} \Pr(\Phi(t) > C) = 0. \tag{3}$$

The *capacity region* is defined to be the set of arrival rate vectors for which there exists a scheduling/routing algorithm that stabilizes the system. Since the queueing process $\{Z(t), t \geq 0\}$ in the Joint Scheduler is an irreducible and aperiodic Markov chain, positive recurrence of this Markov chain implies stability of the system. We first characterize the capacity region of the system using an outer bound, and then prove that the Joint Scheduler stabilizes the system for any arrival rate vector strictly within this outer bound.

A. Characterization of the Capacity Region

The *capacity region* $\mathcal{C}$ of the system (stability region in [19]) is defined as the set of all the arrival rate vectors for which there exists a scheduling algorithm that stabilizes the system. We characterize $\mathcal{C}$ by first consider some necessary conditions for an arrival rate vector λ to be in $\mathcal{C}$.

Assume that the system is stable under some scheduling algorithm for an arrival rate vector λ and that the system is in steady state. The rates in the system are denoted by the following notations, which are also labeled in Fig. 2.

- $\lambda_{\vec{L},m,0}$: the rate of $\{A_{\vec{L},m,0}(t), t \geq 0\}$;
- $\lambda_{\vec{L},m,1}$: the rate of $\{A_{\vec{L},m,1}(t), t \geq 0\}$;
- $\lambda_{m,1}$: the rate of $\{A_{m,1}(t), t \geq 0\}$;
- $\alpha_{m',m}$: the rate of $\{A^Q_{m',m}(t), t \geq 0\}$;
- $\eta_{r',r}$: the rate of $\{A^X_{r',r}(t), t \geq 0\}$;
- $\lambda_{m,2}$: the rate of $\{A_{m,2}(t), t \geq 0\}$;
- μ_m: the rate of $\{A_{m,0}(t), t \geq 0\}$.

The arrival processes above are as defined in Section IV-C. We assemble these rates into a flow vector f for conciseness. Since the system is in steady state, the rate with which tasks arrive at each queue should be equal to the rate with which tasks depart from the queue; i.e., f satisfies the following flow conservation equations at each queue. For each type $\vec{L}$, each machine m and each rack r,

$$\lambda_{\vec{L}} = \sum_{m \in \mathcal{M}_{\vec{L}}} \lambda_{\vec{L},m,0} + \sum_{m \in \mathcal{M}_{\vec{L}}} \lambda_{\vec{L},m,1} \tag{4a}$$

$$\sum_{\vec{L} \in \mathcal{L}_m} \lambda_{\vec{L},m,1} = \lambda_{m,1} + \sum_{m' \in \mathcal{M}_{r_m}} \alpha_{m,m'} \tag{4b}$$

$$\sum_{m \in \mathcal{M}_r} \lambda_{m,1} = \sum_{r'} \eta_{r,r'} \tag{4c}$$

$$\sum_{r' \in \mathcal{R}} \eta_{r',r} = \sum_{m \in \mathcal{M}_r} \lambda_{m,2} \tag{4d}$$

$$\sum_{m' \in \mathcal{M}_{r_m}} \alpha_{m',m} + \lambda_{m,2} = \mu_m. \tag{4e}$$

We call such f a λ-*admissible flow vector*. A flow vector f is said to be *supportable* if the corresponding arrival rate to each queue is less than the service rate of that queue; i.e., for each machine m and each rack r,

$$\sum_{\vec{L} \in \mathcal{L}_m} \lambda_{\vec{L},m,1} \leq B_1 \tag{5a}$$

$$\sum_{m \in \mathcal{M}_r} \lambda_{m,1} \leq B_2 \tag{5b}$$

$$\sum_{r' \in \mathcal{R}} \eta_{r',r} \leq B_2 \tag{5c}$$

$$\sum_{m' \in \mathcal{M}_{r_m}} \alpha_{m',m} + \lambda_{m,2} \leq B_1 \tag{5d}$$

$$\sum_{\vec{L} \in \mathcal{L}_m} \lambda_{\vec{L},m,0} + \mu_m \leq \varphi. \tag{5e}$$

Let $\mathcal{F}_\lambda$ be the set of all λ-admissible flow vectors. Then a necessary condition for the arrival rate vector λ to be in the capacity region is that there exists a supportable $f \in \mathcal{F}_\lambda$.

Consider the set Λ defined as follows:

$$\Lambda = \{\lambda \mid \text{there exists a supportable } f \in \mathcal{F}_\lambda\}.$$

Then since such arrival rate vectors satisfy some necessary conditions to be in the capacity region $\mathcal{C}$, Λ gives an outer bound of $\mathcal{C}$; i.e., $\mathcal{C} \subseteq \Lambda$.

B. Achievability

Stability region $\mathcal{C}_{JS}$ of the Joint Scheduler is defined to be the set of all arrival rate vectors for which the system is stable under JS [19]. The proposed Joint Scheduler achieves the maximum throughput as stated in the following theorem.

Theorem 1 (Throughput Optimality)**.** *The Joint Scheduler stabilizes the system for any arrival rate vector strictly within* Λ*. Therefore* Λ *characterizes the capacity region in the following sense*

$$\Lambda^o \subseteq \mathcal{C}_{JS} \subseteq \mathcal{C} \subseteq \Lambda,$$

where Λ^o *is the interior of* Λ*, and hence the Joint Scheduler is throughput optimal.*

Due to space limitations, we omit the proof of this theorem and refer the reader to our technical report [20].

VI. Simulations

In this section we use simulations to compare the performance of the Joint Scheduler (JS) and the Hadoop Fair Scheduler (HFS) with delay scheduling. HFS with delay scheduling takes data locality into account and shows great performance improvement compared with the default FIFO scheduler and the naïve fair sharing without delay scheduling [6]. As remarked in Remark 3, we mainly focus on the throughput performance and demonstrate that JS achieves the maximum capacity region while HFS cannot. Even though JS has not been fine-tuned to decrease task delay, the simulation results show delay reduce under JS compared to HFS with moderate to heavy load.

A. Settings

We simulate a MapReduce computing cluster with two hundred machines organized into ten racks. A distributed database is stored on machines in the cluster, with three replicas of each data chunk, and jobs are submitted to process part of the data. This mimics the scenario such as data chunks constitute a database like the user profile database of Facebook, and each job is some manipulation of the data like searching for some user.

We scale the time slot in the system such that transmitting one data chunk between machines in the same rack takes one time slot; i.e., $B_1 = 1$. For the interrack traffic, we assume an oversubscription factor 4; i.e., $B_2 = 5$. The service rate is set to $\varphi = 0.25$ since typically processing a data chunk is slower than transmitting it. With this processing capability, the total arrival rate λ_Σ of map tasks should be no larger than $200 \times \varphi = 50$. We run the simulations for the two algorithms under several total arrival rates. For each arrival rate we run the simulation for 10^6 time slots and evaluate the performance using results from the last 5×10^4 time slots, during which the system is either unstable or in the steady state. Job sizes are generated following the analysis of workload from [11], which shows that the number of tasks in a job follows a power-law distribution. For HFS, we treat each job as a separate user so that jobs share the cluster equally. For JS we use subqueues for fairness purpose as we remarked in Remark 3.

We consider two different data access patterns for the task arrival processes. The first pattern accesses data uniformly from all the machines, and the second pattern accesses data from half of the racks more frequently than from the other half, which mimics the scenario with popularity skew.

B. Uniform Data Access

As we analyzed for the data locality problem, HFS performs well under the arrivals with uniform data access pattern since most machines can easily find a local task to serve. In this scenario, JS and HFS perform similarly in throughput performance. Fig. 4a shows the average number of backlogged tasks in the last 5×10^4 time slots. The system is stable under both algorithms for all the arrival rates shown in the figure. Fig. 4b shows the average task delay in steady state, from which we can see that the task delay performance is very similar (HFS is slightly better than JS).

C. Data Access with Popularity Skew

This is the scenario that JS benefits from routing in advance under moderate to heavy load, since many tasks need to be launched on non-local machines under HFS in this case. The average number of backlogged tasks in the last 5×10^4 time slots is shown in Fig. 5a. Under HFS, the average number of backlogged tasks shows a sudden increase at the total arrival rate $\lambda_\Sigma = 39$ tasks per time slot, which indicates that the system is unstable at arrival rates greater than that, while under JS the system remains stable for all the total arrival rates shown in the figure.

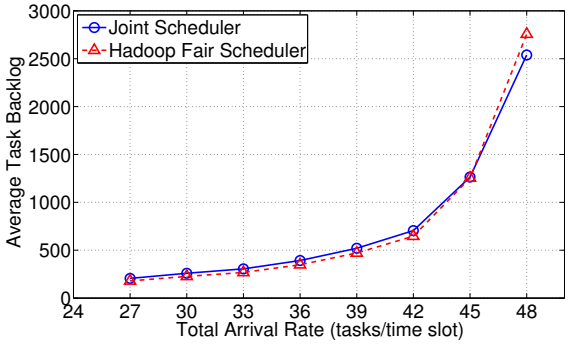
The average task delay is shown in Fig. 5b. When the total arrival rate is small, the task delay performance is still similar (HFS is slightly better than JS). As the arrival rate increases, the task delay performance under HFS becomes worse. The average task delay under HFS becomes larger than that under JS when the arrival rate is greater than 35 tasks per time slot. For arrival rate greater than 39 tasks per time slot, the average task delay under HFS becomes very large (it is already over 2000 for $\lambda_\Sigma = 39$) due to instability. Thus to get a clear comparison figure, we did not show the average task delay under HFS for arrival rate larger than 38 tasks per time slot. Overall, JS significantly reduces the task delay under moderate to heavy load.

VII. Conclusion

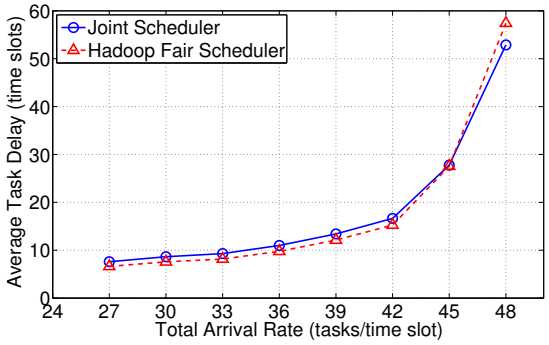
In this paper, we considered the data locality problem in MapReduce computing clusters. We proposed a scheduling/routing algorithm, the Joint Scheduler, that addresses the data locality problem from a network perspective. The Joint Scheduler makes fully use of the computing resources by routing the input data of some tasks in advance without waiting for an available machine to request it. This speeds up the processing of tasks in the existence of data popularity skew. Then we characterized the capacity region of a MapReduce computing cluster with data locality and communication network constraints, and proved that the Joint Scheduler achieves the maximum capacity region. This scheduling/routing algorithm can be extended immediately to communication networks with more levels of hierarchy.

VIII. Acknowledgement

This work was supported in part by NSF Grant ECCS-1255425.

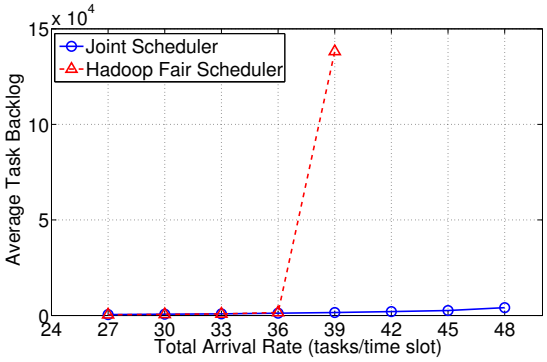


(a) Throughput performance.

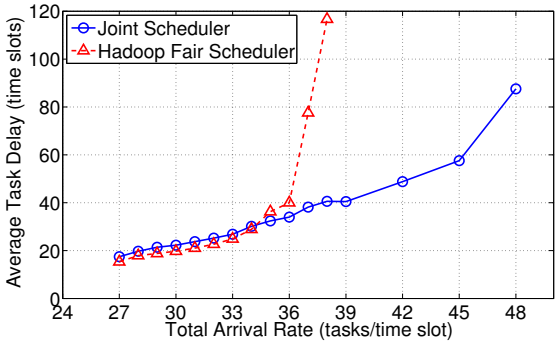


(b) Task delay performance.

Fig. 4: Performance under uniform data access.



(a) Throughput performance.



(b) Task delay performance.

Fig. 5: Performance under data access with popularity skew.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *ACM Commun.*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] "Hadoop," http://hadoop.apache.org.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, Bolton Landing, NY, 2003, pp. 29–43.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *IEEE Symp. Mass Storage Systems and Technologies (MSST)*, Incline Villiage, NV, May 2010, pp. 1–10.

[5] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *Proc. IEEE/ACM Int. Conf. Cluster, Cloud and Grid Computing (CCGRID)*, Melbourne, Australia, 2010, pp. 94–103.

[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 Conf. Computer Systems (EuroSys)*, Paris, France, 2010, pp. 265–278.

[7] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, Big Sky, MT, 2009, pp. 261–276.

[8] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, Lisbon, Portugal, 2011, pp. 187–207.

[9] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "Bar: An efficient data locality driven task scheduling algorithm for cloud computing," in *Proc. IEEE/ACM Int. Conf. Cluster, Cloud and Grid Computing (CCGRID)*, Newport Beach, CA, 2011, pp. 295–304.

[10] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, "Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment," in *IEEE Int. Conf. Cluster Computing (CLUSTER)*, New Orleans, LA, 2009.

[11] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proc. European Conf. Computer Systems (EuroSys)*, Salzburg, Austria, 2011, pp. 287–300.

[12] C. Abad, Y. Lu, and R. Campbell, "DARE: Adaptive data replication for efficient cluster scheduling," in *IEEE Int. Conf. Cluster Computing (CLUSTER)*, Austin, TX, 2011, pp. 159–168.

[13] Q. Xie and Y. Lu, "Degree-guided map-reduce task assignment with data locality constraint," in *Proc. IEEE Int. Symp. Information Theory (ISIT)*, Cambridge, MA, 2012, pp. 985–989.

[14] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," in *Proc. IEEE Int. Conf. Computer Communications (INFOCOM)*, Turin, Italy, 2013, pp. 1609–1617.

[15] ——, "A throughput optimal algorithm for map task scheduling in mapreduce with data locality," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 33–42, Mar. 2013.

[16] T. White, *Hadoop: The definitive guide*. Yahoo Press, 2010.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. Ann. ACM SIGCOMM Conf.*, Seattle, WA, 2008, pp. 63–74.

[18] L. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures on Comput. Architecture*, vol. 4, no. 1, pp. 1–108, 2009.

[19] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Trans. Autom. Control*, vol. 4, pp. 1936–1948, Dec. 1992.

[20] W. Wang and L. Ying, "Data locality in mapreduce: A network perspective," Arizona State University, Tech. Rep., Jul. 2013.