Flow Scheduling: An Efficient Scheduling Method for MapReduce Framework in the Decoupled Architecture

Chin-Jung Hsu
Department of Computer Science
North Carolina State University
chsu6@ncsu.edu

Vincent W. Freeh
Department of Computer Science
North Carolina State University
vwfreeh@ncsu.edu

ABSTRACT

Hadoop is a popular implementation of the MapReduce programming model for data processing. We first compare different Hadoop models and discuss their advantages and limitation. The traditional Hadoop system is scalable because a machine serves both computation and storage function. However, this principle imposes a strong constraint on system design and does not quite fit enterprise and cloud application, which require to decouple computation and storage nodes. Any naive Hadoop implementations may fail to be optimized because they are designed to preserve data locality, which does not exist in the decoupled model. In this paper, we propose a flow scheduling method: it eliminates undesired factors that can decrease processing performance. We model the cost of task assignment based on the penalty of violating flow demand and convert this problem to the network optimization problem. We have implemented Flow Scheduler for Hadoop and the experiment results show that it can maximize the processing flow rate while improving the system throughput by up to 30%. More interestingly, our flow scheduling method can provide more smooth task execution time, which suggests it can eliminate stragglers that caused by resource contention.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*scheduling*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms, network problem*

General Terms

Design, Algorithms, Performance

1. INTRODUCTION

This is an era of big data, and IDC even estimated the exponential growth of data by a factor of 10 [21, 29]. The growing data set provides huge potential for business and research to explore. Thus, big data analytics has become critical demand to dig valuable information from explosive growing data. The giant volume of data along with its variety and velocity poses great challenge to big data analytics[35], and therefore its optimization remains an active research domain.

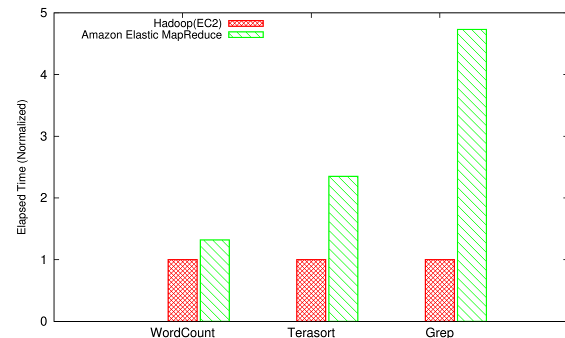


Figure 1: Hadoop performance in the decoupled model. Four x1.large instances are used, and Amazon EMR uses Amazon S3 as the storage for input and output files. Amazon EMR does not utilize all available machines because its resource scheduler does not fit the decoupled Hadoop model.

The MapReduce programming model has gained increasing success for parallel data processing because of its horizontal scale and fault tolerance [13]. Apache Hadoop [4] and Microsoft Dryad [18], for example, are popular systems that embrace this model. In order to handle large-scale data, these systems usually run on top of large clusters of commodity machines and tightly couple computation and storage nodes. This design enables a system to scale out easily because the ratio between computation power and I/O capability remains constant [13, 4, 10].

However, this principle imposes a strong constraint on system design, and thus cannot apply to many use cases, such as enterprises application [23, 27], scientific computing [11, 28], and cloud computing [1, 6, 16]. For example, storage area network (SAN) is popular in enterprise and Amazon Simple Storage Service (Amazon S3) is the recommended storage for Amazon web services. These scenarios prefer a separate storage architecture because of high efficient stor-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

age, data lifecycle management and elastic cloud storage. More importantly, this separate architecture provides high flexibility of system deployment, which is a big challenge in enterprise and datacenter management [30, 27]. For these reasons, we believe the decoupled model will become more important for MapReduce and Hadoop.

The decoupled mode has emerged but only few research studies pay attention to this scenario. This situation leads to poor performance when running Hadoop with the decoupled model as shown in Figure 1. To understand the performance of the decoupled model, we ran three different types of jobs on Amazon EC2 for the Hadoop reference model and Amazon Elastic MapReduce for the decoupled one. The result shows that the decoupled model has system throughput that is much lower than we expected. We find that existing Hadoop schedulers cannot work well when data locality no longer exists. Besides, we doubt that the overhead of large data movement can greatly affect system throughput because the storage system and the network infrastructure cannot fulfill the demand of Hadoop applications.

In this paper, we view the decoupled Hadoop model as a flow network in which data flows through computation and storage facilities. The data flow rate is the amount of data processed or transferred per unit of time, and the system throughput can be measured by the flow rate. We argue that maintaing high data flow rate can increase the system throughput. In the decoupled model, there are two possible conditions that a computation facility does not fully utilize its processing power: 1) the flow rate of data supply from the remote storage facility is not fast enough and 2) the computing facility cannot process data flow fast enough. The first case can happen when the storage facility cannot handle a large number of simultaneous data access and when network infrastructure cannot sustain such a large amount of data transfer, especially when cross-rack communication happens. The second case comes from the overhead of operating system and the data access over network. We believe eliminating undesired factors that affects the processing flow rate can increase the system throughput in the meanwhile.

We propose a flow scheduling method that models the penalty cost of task assignments. Given the cost model, we encode the scheduling problem as the min-cost flow problem so that we can derive the optimal task assignment. We have designed Hadoop Flow Scheduler that implements our flow scheduling method. This Flow Scheduler requires job profile and machine profile in order to decide the optimal task assignment. We first estimate the flow demand of tasks and the flow capability of facilities and then feed this information to our Flow Scheduler. Our experiment result shows that we can improve the system throughput by up to 30%. More importantly, our flow scheduling method can eliminate stragglers in the decoupled model and provide more smooth task execution time.

This paper is organized as follows. Section 2 provides the background on the new Hadoop design and the decoupled model. Section 3 describes the ideal behind our flow scheduling method, the definition of data flow rate and the cost model for task assignment. Section 4 details the Flow Scheduler implantation for Hadoop and how we estimate the data flow rate. We evaluate Flow Scheduler in Section 5 and discuss the limitation of current implementation. Section 6 gives the most related work and we conclude in Section 7.

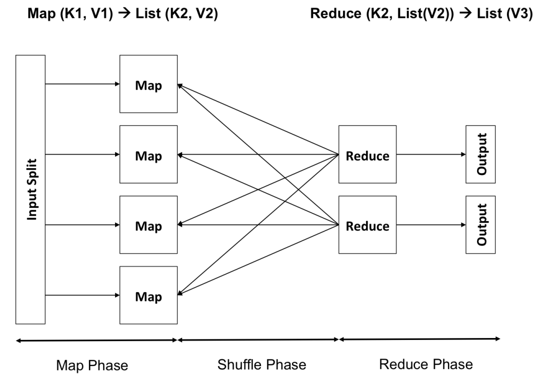


Figure 2: A MapReduce job consists of the map, shuffling and reduce phase. The map task processes a portion of input data and the reduce task aggregate the output from map tasks. The phase between the map and the reduce phase to dispatch intermediate result is the shuffling phase.

2. BACKGROUND

2.1 MapReduce Programming Model

The MapReduce programming model [13] is a simplified model for data processing. In the MapReduce model, a data processing job consists of three phases. First, the map phase processes a portion of input data and generates output as a list of key/value pairs. Next, the shuffle phase is the period to transfer data between map tasks and reduce tasks Finally, the reduce phase merges the output, which is aggregated by key, from the map phase. Many data processing applications can benefit from this simplified model, such as large-scale indexing, machine learning problems and graph computation [13]. Google and Yahoo, for example, apply this model to accelerate large-scale data processing and empower their online advertisement business.

Figure 2 shows the three phases in a MapReduce job. This simplified model is relatively easy for programmers to develop a MapReduce application. Programmers only need to implement the map function and the reduce function. Each map function processes a portion of input data at a time, e.g. one line of a file or a XML document, and the output of the map function is a list of key-value pairs. The reduce function then aggregates those key-value pairs and generates the final output. Programmers do not need to specify the details of the shuffling phase. The MapReduce runtime handles complex data exchange: it aggregates the results based on the output key from the map phase, and then transfers the output to corresponding reduce tasks.

The MapReduce model simplifies data processing because it hides complicated data exchange among tasks and makes fault tolerance easier to support. Unlike the Message Passing Interface (MPI) [17], a MapReduce job does not have to specify the host to send and receive data; the MapReduce runtime, instead, automatically transfer data in the shuffle phase. This embarrassingly parallel design [25, 14] is not flexible and is limited to some data processing applications. However, it can ease the pain when developing data processing programs. On the other hand, tasks in a MapReduce job do not have strong dependency. This fact enables to support fault tolerance without complex checkpointing mecha-

nism; it needs only re-run the failed task again. In short, the MapReduce programming is simple but yet powerful enough to achieve the goal of many practical data processing applications.

2.2 Hadoop

Hadoop [4] is an open-source implementation of the MapReduce programming model, which provides a reliable and scalable system for data processing. The latest Hadoop project includes four modules: 1) Hadoop YARN is a cluster resource management framework, 2) Hadoop MapReduce is the implementation to support the MapReduce programming model based on YARN, 3) Hadoop HDFS is a distributed storage system for Hadoop application data, and 4) Hadoop Common is the common utilities that are required by the above modules.

2.3 Hadoop Job Execution Flow

[1]

The Resource Manager (RM) is responsible for resource allocation. Once received a MapReduce job, RM allocates resource to initialize a AppMaster. This AppMaster is created to negotiate computing resources with the preconfigured resource scheduler, e.g. FIFO Schduler, Capacity Scheduler [2], Fair Scheduler [3]. These schedulers allocate resources (or slots) to the AppMaster based on objectives such as data locality or fairness. Those allocated resources can be used to run map or reduce tasks. Different from Hadoop 1.x (non YARN based Hadoop), the number of map and reduce slots is required to define explicitly, and research studies [9, 32] show that the optimal configuration of this value is not intuitive. The YARN resource framework, on the other hand, is flexible because it determines these numbers dynamically based on the application request and the node capability.

After obtaining computation resources, the AppMaster starts several node containers to process input splits. Each map task handles an input split, and the size is usually 64MB or 128MB (the block size of HDFS). This input split is parsed by the RecordReader object and the map task processes a record at a time. Each record read operation creates the FSDataInputStream object to access the file input stream as shown in Figure 3. The source of the input split can be a block from HDFS or it can an data object from Amazon S3 or Azure Blob Storage service.

2.4 Hadoop Models

The most common way to deploy a Hadoop system is to configure a node to run both Hadoop MapReduce and Hadoop HDFS, which can avoid bringing data to computation. However, data management, in many cases, requires separating computing and storage nodes for flexibility and efficiency. For example, enterprises prefer silos in order to manage high-value data. Amazon Elastic MapReduce primary uses Amazon S3 for its persistent data storage. Moreover, many high performance files systems are considered to replace HDFS in order to support both MapReduce application and other workload. As a result, different scenarios require different ways to deploy Hadoop. As shown in Figure 3, we can define three major types of Hadoop configuration as follows.

[1]The latest YARN framework allocates resources based on memory and CPU will be considered soon

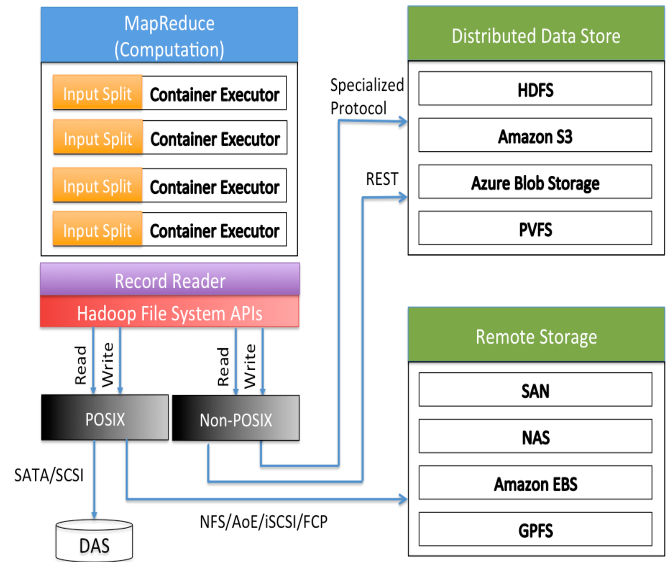


Figure 3: System configuration of Hadoop. Each slot handles one input split and uses the RecordReader object to read data from a POSIX compliant file system or a non-POSIX compliant file system, e.g. distributed data store.

1. **The reference model** is the most common configuration for dedicated Hadoop clusters. Each node in a Hadoop cluster serves both computing and storage functions. This model has the scale-out benefit because the ratio between computation and storage remains constant. Each computation node own its own private storage (local disks), which can provide Hadoop jobs enough data access bandwidth. Most popular Hadoop schedulers aim to maximize the number of local map tasks so that data access is at local but not from remote. This approach guarantees enough I/O bandwidth when the number of computation nodes increase. On the other hand, the output data of reduce tasks will be written to the local HDFS (the data node), and then replicated to other remote data nodes.

2. **Remote Storage Model** is common in enterprises or high performance computing (HPC). This model has the similar definition of AoE architecture [30]. NFS and SAN are quite common to store data, and Hadoop might require to access data in a separate storage, other than HDFS. In this case, each computation node can access data via file-level protocol or block-level protocol (mounted as a local file system). The mounted point should appear the same across all computation nodes, and thus data can be accessed on any nodes. An good example is to replace HDFS with GPFS [7].

3. **Cloud Storage Model** is defined as a separate architecture, which is similar to the remote storage model but with specialized data access protocol. This is also similar to the Split architecture [30] but we don't limit this model to HDFS only. The so called model can be found easily in many cloud computing platforms, like Amazon Elastic MapReduce or Azure HDInsight.

Table 1: Comparisons of Hadoop models with different storage architecture

	Hadoop Reference Model	Remote Storage Model	Cloud Storage Model
Resource sharing	Dedicated Hadoop cluster	Enable time and space sharing	Same as the remote storage model
Deployment flexibility	The ratio between computation and storage resources remains fixed	Highly flexible to determine the size of computation and storage resource	Same as the remote storage model
Performance scalability	Scale-out easily	Proportional to storage I/O capability and network bandwidth	Network limitation
Read/Write Performance	Depends on local disk configuration	Support concurrent read/write (high I/O performance)	Long latency for REST and SOAP protocol
Replication	Triple replication is common	Block-level replication	Geo-replication is considered to reduce latency
Hadoop compatibility	Native support	Mounted as local storage, e.g. NFS	Only Amazon S3 is supported by default
Hadoop optimization	Schedulers are optimized for data locality	Not optimized	Not optimized
Extra	Hadoop is optimized for this model, but fails to apply to many use cases	Enterprise storage supports rich functions, e.g. de-duplication, archiving, data lifecycle management	Best fits the cloud computing scenario

This configuration forces a task to read and write data from and to the remote storage service.

Contrary to the reference model, we classify the remote storage model and the cloud storage model as the decoupled model. Because the decoupled Hadoop model can provide more flexibility in deployment and thus the decoupled model would be our focus in this paper.

Different from traditional Hadoop model, data locality is no longer valid for scheduling jobs, which can cause existing scheduler not suitable for the decouple Hadoop model. Moreover, the decoupled Hadoop model can face performance degradation because all the input data have to be transferred over the network In such a case, the system throughput can be affected because CPU can be idle while waiting for the data to be ready. For these reasons, the decouple model requires a new scheduling algorithm for better performance.

In order to optimize the system throughput (minimizing turnaround time), we argue that maximizing the data processing rate on computing nodes can greatly increase the system throughput. The data processing rate can be affected if the input data cannot be pulled quickly enough or the output data cannot be pushed to remote storage nodes. To address this challenge, we introduce flow scheduling in the next.

3. FLOW SCHEDULING

In this section, we describe the concept of flow scheduling and then explain how we model the assignment cost based on the penalty of violating flow demand. We also introduce how to encode the scheduling problem as a min-cost flow problem.

3.1 Concept of Data Flow

A MapReduce job includes several map and reduce tasks, and each of them (on a computing facility) reads data, processes data and writes data. The data flow rate is the size of data that goes through a facility per unit time. For example, the processing flow rate is how fast a machine can process data, and a faster processing rate also suggests higher system throughput. In a decoupled model, all of the read and write operations involve network activities, which can be costly and decrease the processing flow rate. Therefore, our flow scheduling tries to maximize processing flow rate on computing facilities so that the system throughput can be increased.

The idea of flow scheduling is similar to the water treatment system. Water supply to a desired end-user must go through several processing steps before water is delivered to users. Users may complain if water supply is not fast enough. This situation can happen if water supply is scarce or if the intermediate facilities cannot process water quickly or if the pipeline is not large enough or if too many users request water at the same time. Thus, to meet the demand from users, a water treatment system should satisfy the above conditions as mush as possible. This analogy truly reflects those factors that affect the system throughout in a decoupled Hadoop model: ensuring the quality of data supply and the flow rate of data processing can increase the system throughput. More precisely, if the flow demand of a task cannot be satisfied by facilities, the scheduling decision is considered costly.

Flow rate is defined as how fast a machine can process data or a network can transfer data.

$$R = \frac{D}{T}$$

Here, D is the size of data and T is the total time to process or transfer the data. Throughput this paper, we use *second* for the time unit and *mega bytes* for the data size unit.

Flow capability of facilities: We define three types of flow capability which are read, write and process. Read flow capability is the maximum flow rate that a facility can pull data from other facilities and write flow capability is similar but its the maximum flow rate that can push data to other

facilities. The process flow capability is defined by how fast a facility can process data. Facilities can be classified as computing nodes, storage nodes and network infrastructure. R^s_{in} is the read flow capability of the storage node and R^s_{out} is the write one; similarly, R^n_{in} and R^n_{out} are for network infrastructure, and the computing node uses R^c_{in} and R^c_{out} for read and write capability. Besides, only the computing node has the process flow capability, R^c_p.

Flow demand of tasks: The flow demand describes the characteristic of tasks and can be used to classify tasks into CPU-intensive (low flow demand) and network-intensive (high flow demand) tasks. A flow rate can vary during task execution, and we assume the flow rate is relatively stable, which can be reasonable because either a map task or a reduce task repeats a piece of the same code based on key-value pairs. The R^t_{in} and R^t_{out} are the read and write flow demand of tasks.

3.2 Cost Model

In this section, we describe how to model the cost of task assignment. The decoupled model can be model as $\{C, S, I\}$, where C is the set of computing facilities, S is the set of storage facilities and I is the network infrastructure. Let $c_i \in C$, $s_i \in S$, where i is an integer. The flow capability of facilities is defined as, for example, $R^{c_i}_p$ for the processing capability on the computing node i and $R^{s_i}_{out}$ for the write capability on the storage node i. Let r^c_p be the processing flow rate on a computing facility. When r^c_p approaches to R^c_p, the system throughput is considered increasing. The tasks to be scheduled are t_{ij}, where i is the *ith* job in the system and j is the *jth* task of the job. Similar to flow capability, $R^{t_{ij}}_{in}$ and $R^{t_{ij}}_{out}$ are the read and write flow demand of tasks.

A scheduling problem is to assign $T = \{t_{ij}\}$ to $C = \{c_n\}$, and our flow scheduling tries to minimize the assignment cost based on the flow rate. Given a task, the cost of an assignment can be defined as the penalty cost that facilities cannot satisfy the flow demand of the task. To fulfill the flow demand of tasks, we should ensure quality flow supply and quality processing flow. There are two conditions in which an assignment can occur high penalty cost: 1) a storage facility is overloaded when $\sum_{t_{ij} \in s_i} R^{t_{ij}}_{in}$ is high, especially when it exceeds $R^{s_i}_{out}$ and 2) a computing facility is filled with network-intensive jobs, which means $\sum_{t_{ij} \in c_i} R^{t_{ij}}_{in}$ is high.

To assign a map task t_{ij}, suppose the input data stores on s_n, the cost to assign the task on c_n is the sum of

$$R^{t_{ij}}_{in} \times (1 + \frac{r^{s_n}_{out}}{R^{s_n}_{out}} + f_c)$$

and

$$R^{t_{ij}}_{in} \times (1 + e_m),$$

where f_c is the parameter if s_n and c_m is not in the same rack, and e_m is the effective load on c_m. The effective load can be defined as $\sum_{t_{ij} \in c_m} R^{t_{ij}}_{in} - stdev(R^{t_{ij}}_{in})$. If flow demands of tasks are diverse on a computing facility, we consider the assignment cost lower because overlapping CPU-intensive and network-intensive tasks can utilize resource efficiently.

When t_{ij} is a reduce task, it usually reads data from multiple computing facilities; thus, we need to count all of these read cost. Suppose t_{ij} is assigned to c_m and the reduce task need to read data from other computing facilities c_k, the total read cost is

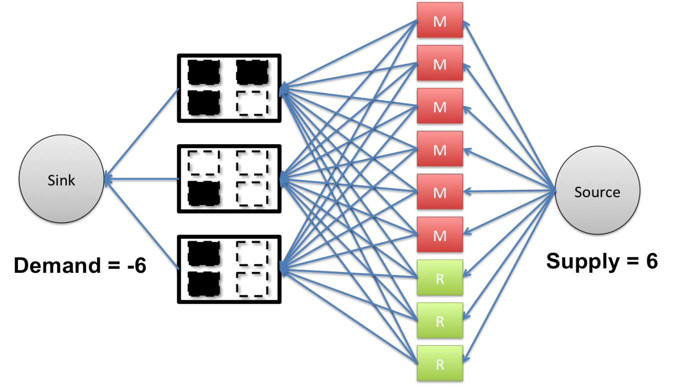


Figure 4: Flow scheduling uses the penalty cost to build the min-cost flow network. The number of supply means the number of tasks to be scheduled.

$$\sum_{c_k} R^{t_{ij}}_{in} \times (1 + \frac{r^{c_k}_{out}}{R^{c_k}_{out}} + f_c),$$

where f_c has the definition similar to the one above. If c_m and c_k are the same node, the parameter is zero, and otherwise, the parameter is small for in-rack communication and large for cross-rack data transfer.

3.3 Min-cost flow optimization

We argued that maximizing the processing flow rate can increase the system throughput. As described in Section 3.2, we model the cost of task assignments as the penalty of violating flow demand. Given the cost model, we encode the scheduling problem as a min-cost flow optimization problem. The min-cost flow problem is a network optimization problem that looks for the cheapest way to allow a certain amount of flow through a network. Given the flow demand of map and reduce tasks, we can decide the best path of network flow with minimum penalty cost. For example, our flow scheduling avoids to assign a task to facilities if the flow capability of the facilities can not meet the flow demand of the tasks. Figure 4 depicts how to encode the scheduling problem to a min-cost flow problem.

The min-cost flow problem can be stated as follows:

$$Minimize\ z(x) = \sum_{(i,j) \in A} c_{ij} x_{ij}$$

subject to

$$\sum_{j:(i,j) \in A} x_{ij} - \sum_{j:(j,i) \in A} x_{ji} = b(i)\ \forall i \in N$$

$$0 \leq x_{ij} \leq u_{ij}\ \forall (i,j) \in A$$

In this equation, $N = \{T_{ij}, C_k, source, sink\}$ and the source node produces flow of capacity n, which is the number of available slots on computing facilities and the sink node has the flow of capability $-n$. Each arc from T_{ij} to C_k has the cost that is defined in Section 3.2, and each arc has a lower capacity zero and a upper capacity one. On the other had, the lower capacity and higher capacity of the arc from the

source to T_{ij} are both *one*, and them of the arc from C_k to the sink are the number of available slots on C_k. After encoding the scheduling problem, we then solve the min-cost flow problem to decide the optimal task assignment.

4. FLOW SCHEDULER FOR HADOOP

Hadoop supports pluggable resource schedulers that makes developing the flow scheduler not being a difficult challenge.

4.1 System Architecture

Our flow scheduler requires job profile (flow demand of tasks) and machine profile (flow capability of facilities). Besides, we use a database to keep track of resource allocation so that next time we can understand the flow rate on facilities. The flow scheduler runs upon AppMaster requests resources or periodically, e.g. every 5 second. In each run, it calculate the penalty cost based on flow demand of tasks and flow capability of facilities, and then encodes the scheduling problem as stated in Section 3.3. After building the cost mode, we use the scaling push-relabled method as described in [15] and their solver program to derive the optimal task assignment.

4.2 Estimating Flow Rate

In this section, we describe how to estimate the flow demand of tasks and flow capability of facilities. Basically, we overload facilities to get flow capability and run a single task to derive the flow demand. First, we measure R^c_{out} of storage nodes to get the flow rate that a storage node can supply. We create a NoComputation map task that only reads input data, but does not do computation and does not generate output. We execute enough NoComputation map tasks at the same time to ensure that the out-bound bandwidth of a storage node is saturated. The result shows that R^{cap}_{out} of storage nodes is roughly close to 65% of the theoretical network bandwidth. Similarly, R^s_{in} is close to R^s_{out}; therefore, we use the same number. Regarding the flow capability of network infrastructure, we use the same number with that of storage nodes. This is true for node communication in the same rack; however, the cross-race communication can be limited by aggregate bandwidth available at top-of-rack switches. The real bandwidth of cross-bandwidth is hard to measure and monitor, and instead, we increase the cost of cross-rack communication in our cost model as described in 3.3.

For the flow capability of computing nodes, R^c_{in} and R^c_{out} are set to the same with storage nodes because this number is mainly limited by the network bandwidth. Regarding R^c_p, we ran different types of Hadoop jobs to estimate the processing capability of computing nodes. In order to faithfully measure the processing capability, we increased the number of map tasks while keeping only one reduce tasks running on the same node. As shown in Figure 5, the maximum flow capability happens when four maps are executed on a four-core node. Even with NoComputation jobs, the max R^c_p is around $60MB/s$, which suggests the limitation of network bandwidth and the overhead of Hadoop framework affects the maximum flow rate of processing in the decoupled Hadoop model. Unless we can break these bottlenecks, we will not be able to increase the the flow rate of processing. Table 2 shows the flow capability that will be used later in our evaluation.

To estimate the flow demand of tasks, we vary input sizes

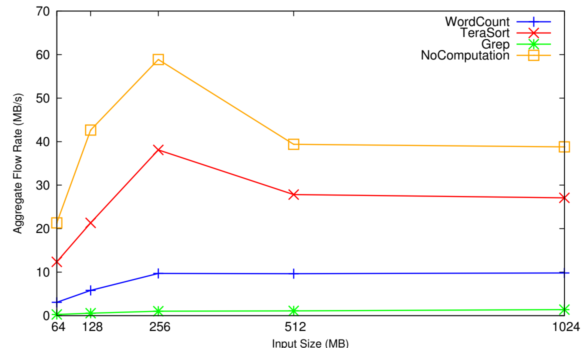


Figure 5: Estimating flow rate of processing capability. Increasing the number of concurrent tasks does not necessarily increase the aggregate throughput; instead, it decreases the throughput, especially when the tasks are network intensive. The computing facility in Cluster 1 has the processing flow capability lower than 60MB/s in

to measure R^t_{in} for map tasks, and we fixed the number of reduce tasks to one. We pick up the flow rate when only one map executes in a computing facility. This ensures that other tasks would not compete resources with the map task we measure. R^t_{out} of map tasks is set to zero because the output data would be staged and later will be pulled by reduce tasks.

It is more tricky to estimate R^t_{in} of reduce tasks because there are multiple sources of flow supply (the shuffle phase contains many-to-many communication [12]) and a reduce task can start even before all map tasks complete. Moreover, the input size can also affect the flow demand. These cases can make the elapse time of reduce tasks longer and would affect the accuracy of estimating flow demand. To eliminate the impact, we use only one reducer in our estimation and we calculate the real flow demand in runtime, which is described as in Section 3.3.

Table 2: Estimated flow capability of facilities. Cluster 1 is powerful than Cluster 2 and their detailed configurations are describe in Section 5

Facility Type	R^c_p	R^s_{out}
Cluster 1	60 MB/s	85 MB/s
Cluster 2	20 MB/s	85 MB/s

5. EVALUATION

5.1 Experiment setup

We evaluate the flow scheduling on NCSU VCL [5]. In our evaluation, two types of nodes are includes: 1) 2 x 2.0GHz and 8GB memory, and 2) 4 x 2.4 GHz and 16GB memory. Each of them is equipped with a 1-Gbps NIC. We refer to the first group as Cluster 1 and the second groups as Cluster 2. Both have Ubuntu 12.04 installed and Hadoop 2.0.3-alpha deployed. One node in Cluster 2 is used for Resource

Table 3: Estimating flow demand of tasks. Only one map and one reduce execute at a time to ensure the estimation accuracy. Lower flow rate suggests it is a CPU-intensive task and it is likely to be a network-inattentive task if flow rate is hight. Terasort requires high demand of bandwidth and Grep requires more computing power (with search pattern .*kinmen.*)

	Cluster 1		Cluster 2		
Job Type	$r^t_{in}(map)$	$r^t_{in}(reduce)$	$r^t_{in}(map)$	$r^t_{in}(reduce)$	*in/out ratio*
WordCount	3.04 MB/s	7.63 MB/s	3.20 MB/s	7.63 MB/s	20%
Terasort	9.14 MB/s	13.31 MB/s	12.8 MB/s	22.18 MB/s	100%
Grep	0.23 MB/s	very small	0.27 MB/s	very small	very small
NoComputation	16.0 MB/s	0 MB/S	21.3 MB/s	0 MB/S	0%

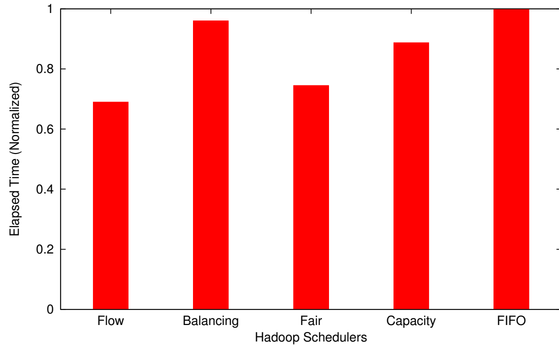


Figure 6: Elapsed time comparison. The FIFO scheduler is the baseline. Nine jobs with 1GB input are submitted every 5 second.

Manger, and another one is for the name node of HDFS. For computing facilities, two clusters are in different network segment, and Cluster 1 has six nodes and Cluster 2 has five nodes. Regarding storage facilities, two data nodes of HDFS are in Cluster 1 and one is in Cluster 2. Each has around 30GB disk space and we set the replication number of HDFS to two.

5.2 Heterogeneous Cluster Setting

We analyze the behavior of our flow scheduler and other existing Hadoop schedulers. We compare Flow Scheduler with FIFO Scheduler, Fair Scheduler, and Capacity Scheduler. Moreover, we create a Balancing Scheduler that distributes the flow demand to computing facilities evenly. In this experiment, we setup a Hadoop system with Cluster 1 and Cluster 2. Then we submit jobs (Wordcount, Terasort and Grep) to the Hadoop system every five seconds. The wordcount and terasort job has an input size 1GB and we submit four times for each of them. Since the grep job has longer running time, we submit only one job with input size 1GB. As shown in Figure 6, FIFO Scheduler has the longest elapsed time and Flow Scheduler can complete all the jobs in a shorter time. Figure 7 shows that Flow Scheduler has lower average execution time of tasks in most of cases. Besides, Flow Scheduler demonstrates more smooth execution time of tasks, as shown in Figure 7(b), because Flow Scheduler has lower values of the standard deviation of execution time.

5.3 Scheduling Overhead

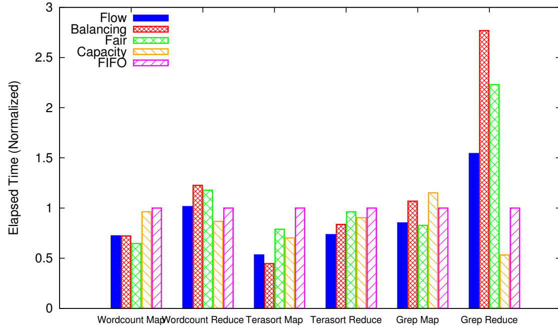
At each scheduling run, we filter out the computing facilities without available slots, and then calculate the penalty cost of each arc in the min-cost flow network. This encoding is exported as an input to the solver to derive the optimal assignment. The solver accounts most of scheduling overhead. The graph size in our experiment is greater than 100 at peak time ($|T| > 100$ and $|C| = 11$). The maximum execution time is around $97ms$ and the minimum one is $2ms$, and the average solving time is $8.3ms$. For a large-scale graph size, as indicated in [19], the solving time of the min-cost flow optimization problem can be acceptable.

5.4 Discussion

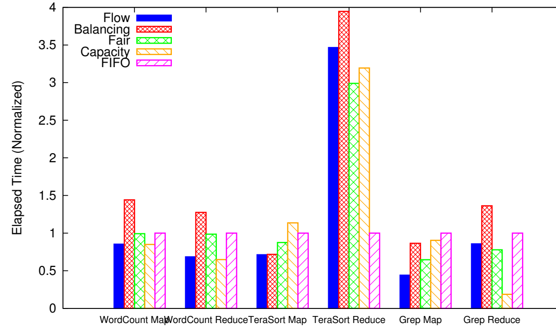
In the traditional Hadoop model, data locality plays an important role in system performance so that most existing schedulers focus on maximizing the number of local map tasks. However, the decoupled Hadoop model does not have this property so that most existing schedulers does not fit in this model. From our experiments, we observed that those schedulers might assign tasks directly to the first node with available slots, which happens on FIFO Scheduler more often. FIFO Scheduler first considers node-level data locality and then rack-level data locality. This scheduling method can cause problems because many tasks of a job would be assigned to a computing facility. This would lead to resource competition especially when high flow demand of tasks are executed on the same node. Figure 5 shows a similar result when multiple higher flow demand tasks are running on a node. The drop percentage appears to be the largest one for the NoComputation line.

The proposed flow scheduling tries to avoid the task assignment that can affect the flow rate of processing. Our cost model considers the quality of data supply (R^s_{out} and R^c_{out}) and the qualify of processing flow rate on computing facilities. We argue that maintaining the flow rate of processing can increase the system throughput, and Figure 5 greatly support our argument. More interestingly, our flow scheduling method can somehow eliminate stragglers [13, 34]. Our result shows that Flow Scheduler derives more smooth execution time of map tasks.

Unlink the graph for the min-cost flow problem in Quincy [19] and CAM [22], we do not construct a hierarchical graph to reflect the hierarchical data center networking. The reason is that we can not convert our desired cost model into a hierarchical graph. Suppose we take the top-of-rack switch into account, we introduce a new switch node SW. We then construct an arc for each task assignment that produces data



(a) Average of execution time



(b) Standard deviation of execution time

Figure 7: The average execution time and the standard deviation of execution time. Flow Scheduler shows more smooth execution time and this somehow suggests flow scheduling can eliminate stragglers that are caused by resource contention.

flow through SW. We also construct another arc for computing facilities that connects to SW. In such a scenario, we cannot decide the cost for the arc from SW to C_m because we lose the information of flow demand. For this reason, Quincy has a cost, zero, for the arc from the rack to computing nodes. In our cost model, we simply raise the cost of an arc that causes cross-rack communication. To address this problem, we probably can look into convex network optimization that can support this scenario so as to delivering a more accurate cost model.

6. RELATED WORK

6.1 The decoupled model

In previous work [30], the author compared the performance of Hadoop integrated with different types of storage architecture. The author found that in the split architecture, Hadoop has imbalance issue access to remote data storage, which can lead to poor I/O performance. In the following, we discussed the use cases that integrate Hadoop with separate storage. After that, we describe scheduling methods for Hadoop and cluster that are related to our work.

6.1.1 Parallel File System

Several research studies show their interests in replacing HDFS with other high performance storage systems. W. Tantisiriroj et al [31] argues that parallel file systems can support diverse workloads and provides a better tradeoff between performance and reliability. The authors proposed a PVFS shim layer to incorporate data layout of PVFS to achieve data locality. Maltzahn et al considered Ceph as a scalable alternative to HDFS [24], and they create a mapping layer which is similar to the PVFS one. GPFS is a shared-disked file system developed by IBM, and widely adopted in supercomputers. R. Ananthanarayanan et al [7] from IBM Research modify data layout in GPFS and expose this information to Hadoop. These are the earliest studies that tried to replace HDFS, but none of them considers optimizing Hadoop at the job scheduler level.

6.1.2 Enterprise Storage

Another research study [27] analyzed the feasibility to use a very powerful storage node to accommodate Hadoop, and he finds that Hadoop performance is dominated by the bandwidth between computing and storage facilities.

Recently, the researchers in NetApp Inc. argued that it is required to decouple compute and storage nodes in big data analytics because enterprise IT often deploys *silos* to manage high-value data [23]. The decoupled Hadoop model would incur high cost on data loading from the backend storage system to compute nodes. They propose MixApart, which includes the data-aware task scheduler, the task-aware data scheduler and a caching mechanism, to optimize Hadoop performance. However, they mainly focused on highly data-reuse workload.

6.1.3 Cloud Storage Service

Cloud computing has emerged as an important technology for the pay-as-you-go model [8, 20]. In such a platform, object storage, e.g. Amazon S3 and Azure Blob Storage, is primarily chosen for persistent data. Amazon Elastic MapReduce and Azure HDInsight are two popular Hadoop platforms on the cloud [1, 6]. Both cases enable Hadoop to support data access to object storage.

6.2 Resource and Job Scheduling

Hadoop has the builtin FIFO scheduler, which allocates resources based on first-come-first-serve policy and data locality. The Fair Scheduler was originally developed by Facebook with the objective of resource sharing, and Yahoo proposed Capacity Scheduler to support fairness and priority sharing; both of them aim at achieving data locality and fairness in a large cluster. LATE [34] improvs the speculative execution by accurately estimating the remaining time of tasks, which can better support Hadoop in heterogeneous environment. HFS [33] uses delay scheduling to solve the conflict between data locality and fairness, and the job throughput can be improved by almost 2x. All of these schedulers are suitable for the Hadoop reference model but not the decoupled model.

Hadoop has poor resource utilization due to the fixed number of map and reduce slots [26]. Polo et al proposed a resource-aware scheduler that incorporates offline job profil-

ing so that resource utilization can be increased. This work does not consider the decoupled Hadoop model, and the new Hadoop YARN supports flexible slot allocation; however, their job profiling can be applied to our system. Moreover, instead of choosing the optimal slot number, our flow scheduling can utilize resource more efficiently because computing facilities can maintain high processing flow rate.

A scheduling problem can be solved as the network optimization problem. Quincy [19] adopts the min-cost flow network to achieve fair scheduling in a distributed commuting system. Our flow scheduling is similar to this approach; however, our cost model is based on flow rate but not the data size that is required by a computing task. We believe flow rate is a better choice because it can be a good indicator to determine the type of a task. For example, a low flow rate task is a CPU-intensive task; however, the data size itself is not enough to determine the right task type. CAM [22] argues that the decoupled model is not suitable for virtualized clouds, and it attempts to co-allocate data with virtual machines. CAM utilizes the network topology information and builds the min-cost flow network to reconcile data placement and VM placement.

7. CONCLUSION

The decoupled Hadoop model is flexible and much more preferable in many scenarios. However, existing Hadoop schedulers do not consider this model and hence the scheduling method fails to optimize the system throughput. Our flow scheduling method uses the penalty cost for task assignments in order to increase the processing flow rate on computing facilities. We encode this problem as the min-cost flow problem and then we can obtain the optimal assignment. We have implemented a pluggable Flow Scheduler for Hadoop YARN and it supports the latest version of Hadoop. Our experiment results have shown that our flow scheduling can greatly improve the system throughput by about 30% so as to eliminate stragglers. These results support that the proposed flow scheduling can maintain the flow rate of processing.

Flow scheduling seems efficient for the decoupled model, but there still remains large space to improve. For our current implementation, Flow Scheduler requires job profile and machine profile, which is not practical. We believe we can estimate the flow demand of tasks and the flow capability of facilities at runtime. A naive approach is to sample the flow demand of a task and then use this information to decide the cost of the remaining tasks of the same job. Another approach is to monitor the flow rate of tasks so that we can adjust the penalty cost dynamically. We can also decide the flow capability of facilities in a similar way. Overall, we are positive about flow scheduling but more extensive evaluations have to be conducted before we can conclude.

8. REFERENCES

[1] Amazon Web Services. http://aws.amazon.com/.

[2] Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[3] Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[4] The Hadoop project. http://hadoop.apache.org/.

[5] Virtual Computing Lab. http://vcl.ncsu.edu.

[6] Windows Azure. http://www.windowsazure.com/.

[7] R. Ananthanarayanan and K. Gupta. Cloud analytics: Do we really need to reinvent the storage stack. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 1–5, 2009.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[9] S. Babu. Towards automatic optimization of MapReduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 137, New York, New York, USA, June 2010. ACM Press.

[10] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110–112, Jan. 2006.

[11] J. Bresnahan, D. Labissoniere, T. Freeman, and K. Keahey. Cumulus : An Open Source Storage Cloud for Science. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 25–31, 2011.

[12] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 conference*, volume 41, pages 98–109, Oct. 2011.

[13] J. Dean and S. Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. In *the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, Dec. 2004.

[14] I. Foster. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Jan. 1995.

[15] A. V. Goldberg. An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. *Journal of Algorithms*, 22(1):1–29, Jan. 1997.

[16] Google. Google Cloud Platform. http://cloud.google.com/, 2012.

[17] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MPI, 1999.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, volume 41, pages 59–72, Mar. 2007.

[19] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, page 261, New York, New York, USA, Oct. 2009. ACM Press.

[20] K. Kambatla, A. Pathak, and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud.

[21] R. T. Kouzes, G. A. Anderson, S. T. Elbert, I. Gorton, and D. K. Gracio. The Changing Paradigm of Data-Intensive Computing. *Computer*, 42(1):26–34, Jan. 2009.

[22] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar. CAM: A Topology Aware Minimum Cost Flow Based Resource Manager for MapReduce

Applications in the Cloud. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing - HPDC '12*, page 211, New York, New York, USA, June 2012. ACM Press.

[23] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza. MixApart: Decoupled Analytics for Shared Storage Systems. In *HotCloud*, 2012.

[24] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J., Nelson, S. A. Brandt, and S. Weil. Ceph as a scalable alternative to the Hadoop Distributed File System. http://static.usenix.org/publications/login/2010-08/openpdfs/maltzahn.pdf, 2010.

[25] C. Moler. Matrix Computation on Distributed Memory Multiprocessors. In *Hypercube Multiprocessors*. 1986.

[26] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé. Resource-aware adaptive scheduling for MapReduce clusters. pages 180–199, Dec. 2011.

[27] G. Porter. Decoupling storage and computation in Hadoop with SuperDataNodes. *ACM SIGOPS Operating Systems Review*, 44(2):41, Apr. 2010.

[28] L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf. Defining future platform requirements for e-Science clouds. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 101, New York, New York, USA, June 2010. ACM Press.

[29] S. Sakr, A. Liu, D. M. Batista, and M. Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys & Tutorials*, 13(3):311–336, 2011.

[30] J. Shafer. *A Storage Architecture for Data-Intensive Computing*. PhD thesis, Rice University, 2010.

[31] W. Tantisiriroj, S. Patil, G. Gibson, S. W. Son, S. J. Lang, and R. B. Ross. On the duality of data-intensive file system design: Reconciling HDFS and PVFS. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC))*, pages 1–12, 2011.

[32] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: a slot allocation scheduling optimizer for MapReduce workloads. pages 1–20, Nov. 2010.

[33] 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 *Proceedings of the 5th European conference on Computer systems - EuroSys '10*, page 265, New York, New York, USA, Apr. 2010. ACM Press.

[34] M. Zaharia, A. Konwinski, and A. Joseph. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, San Diego, California, 2008.

[35] P. Zikopoulos and C. Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. Oct. 2011.