IEICE TRANS. INF. & SYST., VOL.E96–D, NO.12 DECEMBER 2013

PAPER Special Section on Parallel and Distributed Computing and Networking

An Efficiency-Aware Scheduling for Data-Intensive Computations on MapReduce Clusters

Hui ZHAO[†a)], **Shuqiang YANG**[†], *Nonmembers*, **Hua FAN**[†], *Student Member*, **Zhikun CHEN**[†], *and* **Jinghu XU**[†], *Nonmembers*

SUMMARY Scheduling plays a key role in MapReduce systems. In this paper, we explore the efficiency of an MapReduce cluster running lots of independent and continuously arriving MapReduce jobs. Data locality and load balancing are two important factors to improve computation efficiency in MapReduce systems for data-intensive computations. Traditional cluster scheduling technologies are not well suitable for MapReduce environment, there are some in-used schedulers for the popular open-source Hadoop MapReduce implementation, however, they can not well optimize both factors. Our main objective is to minimize total flowtime of all jobs, given it's a strong NP-hard problem, we adopt some effective heuristics to seek satisfied solution. In this paper, we formalize the scheduling problem as job selection problem, a load balance aware job selection algorithm is proposed, in task level we design a strict data locality tasks scheduling algorithm for map tasks on map machines and a load balance aware scheduling algorithm for reduce tasks on reduce machines. Comprehensive experiments have been conducted to compare our scheduling strategy with well-known Hadoop scheduling strategies. The experimental results validate the efficiency of our proposed scheduling strategy.

key words: *data-intensive computation, MapReduce, Hadoop, algorithm design, scheduling, grid computing, data locality, cloud computing, flowtime*

1. Introduction

MapReduce [1] has emerged as one of the most popular frameworks for data-intensive distributed cloud computing. The benefits of this simple yet powerful programming model has been demonstrated on a wide spectrum of domains, ranging from search and ads analysis (e.g., [2]–[4]), bioinformatics (e.g. [5], [6]), to artificial intelligence, machine learning and data mining (e.g. [7]–[9]).

MapReduce defines a simple computation as a MapReduce job, which is divided into two phases, named map and reduce. A job starts from the map phase, which consists lof a cluster of parallel map tasks, job's inputs are divided into several small pieces, named splits, each of which is used as input of a map task and usually located in a computer to prevent network cost. In map phase, a customized function on input data is applied, converting each $< k1, v1 >$ into $< k2, v2 >$ as intermediate output, which are sorted, partitioned and optionally written into local disks if pull-based data exchange strategy is used, finally each partition will be copied to a corresponding reduce task. After all intermediate results are copied to reduce nodes, the job can start reduce phase, which also consists of a number of parallel reduce tasks, each of which uses a partition of map phase output as its input. Figure 1 illustrates the dataflow of a typical MapReduce job in detail.

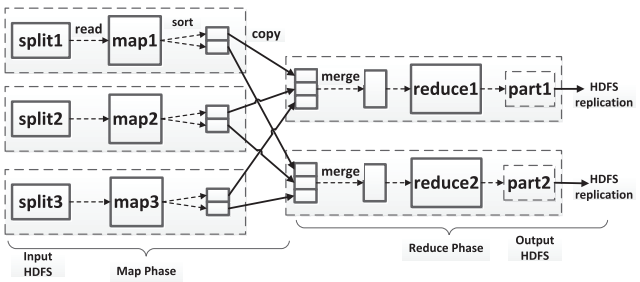

Fig. 1 The dataflow of a typical MapReduce job.

Compared to other designs, MapReduce provides some new features described as follows. (i) It abstracts an simple but expressive programming language interface. Although MapReduce only provides two functions map() and reduce(), a large number of data analytical workloads can be expressed as a set of MapReduce jobs, including all kinds of SQL query, data mining, machine learning and graph processing etc. The programming model is also independent of the underlying storage system and is able to process various types of data, structured or unstructured. This storage-independent design is considered to be indispensable in a production environment where mixed storage systems are deployed [10]. (ii) MapReduce achieves elastic scalability through block level scheduling. The runtime system automatically splits the input dataset into even-sized data blocks and dynamically schedules them to the available compute nodes for processing. MapReduce is proven to be highly scalable in real systems. Installation of MapReduce on a shared-nothing cluster with 4,000 nodes has been reported in [11]. (iii) MapReduce provides fine-grained fault tolerance whereby only tasks on failed nodes have to be restarted.

However, the performance of MapReduce is still far from state of the art parallel database systems. According to a recent comparing study, Hadoop [12], the open source implementation of MapReduce, is slower than two parallel database systems by a factor of 3.1 to 6.5 [13]. Thereafter, optimization problem for MapReduce systems has become an active research topic in industry and academia.

Data locality is an important heuristic to improve system efficiency on massive data processing, which means

Manuscript received December 30, 2012.
Manuscript revised April 3, 2013.

[†]The authors are with the National University of Defense Technology, China.
a) E-mail: splendidjao@gmail.com

DOI: 10.1587/transinf.E96.D.2654

moving computation to where its input is located. The only challenge of data locality is data location knowledge, in MapReduce systems (such as Google' MapReduce implementation or Hadoop MapReduce implementation), there is a distributed file system (such as GFS [14] or HDFS [15]) to maintain data location information, so data locality is very easy to achieve, however, only high data locality does not means high efficiency, the second challenge is load balance. To achieve better load balance, current Hadoop schedulers adopt a greedy resource allocation strategy, i.e. as soon as a node becomes free, it requests a task from the central scheduler through a heartbeat message, then the scheduler assigns a task to it according to "best effort" data locality (choose a task whose input data is on the requesting node from a task set, otherwise, a remote task is chosen). FIFO scheduler queues jobs by job submission time, when a free node requests a task, the scheduler only find a task from the head job of the job queue according to "best effort" data locality. Hadoop Fair Scheduler (HFS) [16] is a more popular scheduler under multi-user environment, which mainly concerns on the fairness among users and jobs. As [17] pointed out, FIFO and HFS both suffer from poor data locality, and a simple scheduling strategy named "Delay Scheduling" is proposed to tackle this issue.

In this paper, we pursue high efficiency of many independent jobs executing in MapReduce cluster rather than only data locality as [17] does. Actually, we think efficiency mainly depends on two aspects: rationality of resource allocation and load balancing among the cluster. For MapReduce clusters, network bandwidth is considered as scarce resource, so most MapReduce schedulers adopt data locality during assigning map tasks as a rule to improve rationality of resource allocation. Unfortunately, most schedulers adopt "best effort" data locality strategy to achieve best load balancing under some job sequencing strategy, such as FIFO, Fairness etc. We think this two important factors can be achieved simultaneously through an sophisticated job sequencing strategy. To simplify the problem of data locality, we adopt strict data locality rather than "best effort" strategy when scheduling map tasks, i.e. pushing map tasks to the places their inputs reside, regardless of whether computing nodes are busy or not. Under this premise, we dynamically change job sequence to pursue better load balancing among the whole cluster. We think this simple scheduling strategy can achieve better efficiency than in-used Hadoop schedulers, such as FIFO and HFS, no matter under "best effort" data locality or strict data locality. Finally, we conduct comprehensive experiments to compare our efficiency-aware scheduling strategy (denoted as EAS) with FIFO and HFS, under "best effort" locality and strict locality respectively, denoted as FIFO-BE, FIFO-S, HFS-BE, HFS-S respectively.

2. Related Work

There are lots of research work on scheduling problem of distributed and parallel systems. In high performance computing (HPC) realm, schedulers like Torque [18] usually support job priority and resource-consumption-aware scheduling and the workloads are always batch-oriented. However, HPC jobs are usually CPU or communication bound, so data locality is not the key consideration. Grid schedulers like Condor [19] also support locality constraints, but usually at the level of geographic sites, because the jobs are more compute-intensive than MapReduce.

Recently, there are some studies on improving the performance of MapReduce systems, A. Abouzeid et al. proposed a hybrid parallel data management system named HadoopDB [20], which combines local databases with Hadoop, making databases be the underlying processing engines to utilize sophisticated database technologies. Dittrich et al. provides a customized index structure and the corresponding executing algorithms to optimize selection and join operations on Hadoop, naming Hadoop++ [21]. Afrati and Ullman worked on optimizing joins in MapReduce environment [22]. These works all have improved the efficiency of Hadoop significantly, however, they are mainly on how to improve efficiency of a job execution. In contrast, our aim is to study how to efficiently execute a cluster of independent jobs on a given cluster of machines, which must be achieved by efficient scheduling strategy.

Now, we introduce the most recently work on scheduling optimization for MapReduce systems. Pietro Michiardi et al. design a scheduler labelled FSP [23], which considers both fairness and efficiency rather than ours efficiency-only objective, and fairness-only scheduling as HFS [16], similar as Delay Scheduling [17] expect for the job-level resource provision like ours rather than task-level used in lots of current Hadoop schedulers, what's more, FSP permits preemption by job suspension. Joel Wolf et al. propose a scheduling optimizer for MapReduce workloads with shared scans named as CIRCUMFLEX [24], which aims on optimizing concurrent jobs with share inputs, on the other hand, we assume jobs are totally independent, however, we will do this kind of optimization in future work. Hammoud et al. propose center-of-gravity reduce task scheduling aiming to lower MapReduce network traffic [25], which model reduce input distribution as mass distribution model, by properly assign reduce tasks to save network cost, so we can call it data locality in reduce phase, which is not a consideration in our study but in future work. Benjamin Moseley et al. study the scheduling problem in MapReduce and Flowshops [26], which formalize job scheduling in map-reduce as a novel generalization of the two-stage classical flexible flow shop (FFS) problem: instead of a single task at each stage, a job now consists of a set of tasks per stage. He et al. develop a new MapReduce scheduling technique to enhance map task's data locality named as MaBtchmaking [27], which is a very good and efficient scheduling technique, motivated by improving data locality of map tasks on the basis of well known in-used Hadoop schedulers, such as FIFO and Hadoop Fair Scheduler. Like Delay Scheduling, they all inherit the resource provision strategy on task granularity rather than ours on job granularity, maybe this strategy

is more flexible, however, we think it will hurt pipeline optimization principle illustrated as Fig. 2, since too many jobs contend the limited resource.

The closest work we know are Delay Scheduling [17]. In Delay Scheduling, when a node requires a task and current selected job according to some job scheduling algorithm has none local task, the scheduler will skip the job and search for the next job in the light of the same job scheduling algorithm. When there is a job which has a local task at the requiring node after some round of skip or the delay threshold is broken, this task assignment is finished. Though this method can improve data locality, but it must sacrifice some utilization while waiting, and in some bad conditions it must wait much time to assign a local task or abandon locality after sacrificing lots of CPU cycles. What is more, Delay Scheduling achieves data locality in task level instead of job level, i.e. it delays free slot allocation to a task from the front job to the rear job in a given job sequence rather than change the jobs order, which will result in too many jobs compete for limited resource in their map phase and reduce phase. This phenomenon has serious impact on the resource utilization and throughput of reduce machines.

3. Problem Definition

3.1 Assumptions

To simplify discussion, we make some reasonable assumptions which is relevant to implementation rather than model. These assumptions are summarized as follows.

(i) No pipeline between map and reduce phase, i.e. for each job, only when all its map tasks finished, its reduce tasks can be scheduled. (ii) A node either acts as map machine or reduce machine, i.e. computing nodes of the whole cluster are divided into two disjoint sets, map machines and reduce machines. Though this is not the fact in real systems such as Hadoop cluster, which consider a machine as both map machine and reduce machine by abstracting a certain number of virtual map slots and reduce slots on physical processors, we think both settings perform nearly the same, because map phase and reduce phase usually use different type of resource, the former is disk-bound, the latter is network-bound. (iii) For all jobs, their input data in map phase are uniformly distributed, i.e. if total input data size is d_i, then each map machine has d_i/N_m data, where N_m represents the total map machines in the whole cluster as defined in 3.3. Each map task uses a block as its input according to current Hadoop input splitting strategy. (iv) Preemption is not permitted as current Hadoop schedulers's assumption, given each task is very small and will finish in more or less one minute. (v) We assume strict data locality, i.e. map tasks are only permitted to execute on the node where its input data is located.

3.2 Job Profile

A job consists of two sets of tasks, namely, map task set and reduce task set, where tasks in a set can be run in parallel, but the sets themselves have to be run sequentially, where no reduce task can be started until all map tasks for the job are completed. Thus, the scheduling problem is precedence constrained.

There are n MapReduce jobs, denoted as $J_1, J_2, \ldots, J_n$, with $n \in \mathbb{N}$ and $n \geq 1$. Each job J_i consists of m_i map tasks $(J_i^1, J_i^2, \ldots, J_i^{m_i})$ followed by r_i reduce tasks $(J_i^{m_i+1}, J_i^{m_i+2}, \ldots, J_i^{m_i+r_i})$, with $m_i, r_i \in \mathbb{N}$ and $m_i \geq 1$, for all $1 < i < n$. Let B_i be the input size of J_i^m, for all $1 \leq m \leq m_i$, and R_i represents the number of input replicas of J_i.

3.3 Hardware Model

The cluster is consists of N_m identical map machines (used to run map tasks), N_r identical reduce machines (used to run reduce tasks), and a managing machine (used to monitor the whole cluster and scheduling workloads.) For each map machine, its data processing speed is p bytes per second under the condition of disk IO. For each reduce machine, its data processing speed is q bytes per second in the case of network IO.

3.4 Scheduling

Let π be a schedule of a batch independent jobs. Given π, for a task of job J_i, let the function $s_\pi(J_i^m)$ denote its starting time and the function $f_\pi(J_i^m)$ denote its completion time, both with respect to the schedule π. We also define $s_\pi(J_i) = \min_m s_\pi(J_i^m)$ and $f_\pi(J_i) = \max_m f_\pi(J_i^m)$, as the starting and finishing times for job J_i. Let $m_\pi(J_i^m)$ be the machine the task J_i^m is assigned under π. A schedule π for job J_i is feasible if and only if the following conditions are satisfied: (i) for each J_i^m, with $1 \leq m \leq m_i$, $m_\pi(J_i^m) \in N_m$, i.e. all map tasks of J_i are scheduled only on the map machines, (ii) for each J_i^r, with $m_i+1 \leq r \leq m_i+r_i$, $m_\pi(J_i^r) \in N_r$, i.e. all reduce tasks of J_i are scheduled only on the reduce machines, (iii) for each J_i^r and J_i^m, with $m_i + 1 \leq r \leq m_i + r_i$ and $1 \leq m \leq m_i$, $s_\pi(J_i^r) \geq \max_m f_\pi(J_i^m)$, i.e. each reduce task for job J_i can not be scheduled until all map tasks for job J_i are completed.

Let a_i be the arrival time of job J_i. The *flowtime* of job J_i with respect to a schedule π is $flow_\pi(J_i) = f_\pi(J_i) - a_i$;let $flow_\pi = \sum_{i=1}^n flow_\pi(J_i)$ be the total *flowtime*. Our objective is to find a schedule π to minimizing the total *flowtime* for a time interval I. Actually, the classical two-stage flow shop problem (FlS) is an important special case of MapReduce scheduling problem, where there is only one map machine and one reduce machine, and FlS is known to be strongly NP-hard [28]. So, in this paper, we want to utilize some heuristics to improve our scheduling objective as far as possible. We divide the problem into two steps, one is map tasks scheduling on map machines (denoted as π_m), the other is reduce tasks scheduling on reduce machines (denoted as π_r). We also divide the original objective into two indirect objectives (maximize the throughput of map machines and reduce machines). For each $m \in N_m$ and $r \in N_r$, let TP_π^m

and TP_π^r represent their throughputs under schedule π, D_π^m and D_π^r are the total input data size of machine m and r respectively under schedule π, T_π^m and T_π^r are the completion times of the last map task and reduce task under schedule π respectively, with $T_\pi^m < T_\pi^r$. Then, we can compute each machine's throughput according to (1) and (2).

$$TP_\pi^m = D_\pi^m / T_\pi^m \tag{1}$$

$$TP_\pi^r = D_\pi^r / T_\pi^r \tag{2}$$

Then, the total throughput of N_m and N_r can be expressed as (3) and (4).

$$TP_\pi^{N_m} = \sum_{m=1}^{N_m} TP_\pi^m \tag{3}$$

$$TP_\pi^{N_r} = \sum_{r=1}^{N_r} TP_\pi^r \tag{4}$$

Finally, we define our objective function as (5).

$$\max_\pi \{TP_\pi^{N_m} + TP_\pi^{N_r}\} \tag{5}$$

During π_m we adopt two heuristics to improve (1), one is data locality, the other is load balance among map machines. During π_r we also adopt two heuristics to enhance (2), one is to balance the load dynamically among reduce machines, the other is to increase the producing rate of schedulable reduce tasks as soon as possible by confining concurrent jobs in π_m, given map phase and reduce phase of the same job accord with producer-consumer model.

4. Scheduling Algorithms

In this section, we design our efficiency-aware scheduling strategy based on the objective mentioned above, which can be achieved by three algorithms: load balance-aware job selection algorithm, strict data locality map task scheduling algorithm, and load balance-aware reduce task scheduling algorithm.

4.1 Algorithm Design

In job scheduling, we formalize it as a job selection problem among a cluster waiting jobs to permit the selected job to obtain resource, aiming to balance load among all map machines, here we specify a job can obtain as many resource as it needs under the precedence of its map phase and reduce phase if it is scheduled, regardless there is free resource or not. This requirement does not follow current Hadoop schedulers, which assigned a resource to a task of a selected job only when the resource is free, next resource will assign to a task of another job, such as HFS [16] and Delay scheduler [17], except for the default FIFO scheduler. We think scheduling the whole job by sequence can mitigate the contention of the same type machines (map machines and reduce machines), enabling pipeline between map phase and reduce phase of different jobs (Fig. 2 illustrates this pipeline feature by a simple example), resulting in higher efficiency of the whole cluster.

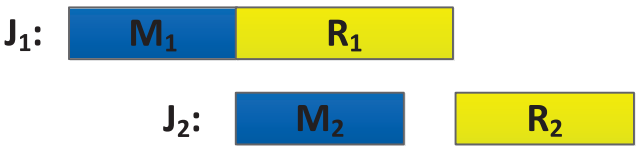

Fig. 2 Pipelined execution of two MapReduce jobs J_1 and J_2, M_i and R_i are the map phase and reduce phase of job J_i, here we assume the size of each M_i and R_i are big enough to occupy the whole map machines and reduce machines.

Algorithm 1 Load balance-aware job selection algorithm

Input: $\mathbb{J} = \{J_1, \ldots, J_n\}$ *The waiting job set*
Output: J *The selected job*
1: **for all** $J_i \in \mathbb{J}$ **do**
2: computing $LB_\pi^{N_m,i}$;
3: **if** $OPT\_LB > LB_\pi^{N_m,i}$ **then**
4: $OPT\_LB = LB_\pi^{N_m,i}$;
5: $J = J_i$;
6: **end if**
7: **end for**
8: **return** J;

Algorithm 2 Map tasks scheduling algorithm

Input: M_i *The map task set of the selected job J_i*
Output: $AM < a, b >$ *The assignment plan of M_i, a represents a map machine, b is a list of map tasks assigned to a*
$\{L^{mm}\}$ *Current load on each map machine*
$\{L_m\}$ *The load of each map task in M_i*
$t = \phi$ *A pointer to a map machine*
when job J_i is selected:
1: **for all** $m \in M_i$ **do**
2: MM=obtain input locations of m;
3: **for all** $mm \in MM$ **do**
4: **if** $t == \phi$ **then**
5: $t = mm$;
6: **else if** $L^t > L^{mm}$ **then**
7: $t = mm$;
8: **end if**
9: **end for**
10: append $< t, m >$ to AM;
11: $L^t = L^t + L_m$;
12: $t = \phi$;
13: **end for**
14: **return** AM;

Firstly, we describe our job scheduling algorithm by Algorithm 1, which compute the load balance feature among map machines by simulating to schedule each job in the waiting job set according to map task scheduling strategy (show as Algorithm 2), in each simulation, we estimate the instant load of each map machine if all map tasks of the job are scheduled, then compute the average load among all map machines, finally compute the standard deviation of load as load balance value, when all simulations finished, the algorithm selects the job which make the best load balance.

Secondly, we give map tasks scheduling algorithm for a selected job J_i (illustrated by Algorithm 2), which will

Algorithm 3 Load balance-aware reduce tasks scheduling algorithm

Input: R_i *The reduce task set of the selected job J_i*
Output: $AR < a, b >$ *The assignment plan of R_i, a represents a reduce machine, b is a list of reduce tasks assigned to a*
$\{L^{rm}\}$ *Current load on each reduce machine*
$\{L_r\}$ *The load of each reduce task in R_i*
$t = \phi$ *A pointer to a reduce machine*
when all map tasks in M_i are finished:

```
for all r ∈ R_i do
    for all rm ∈ N_r do
        if t == φ then
            t = rm;
        else if L^t > L^rm then
            t = rm;
        end if
    end for
    append < t, r > to AR;
    L^t = L^t + L_r;
    t = φ;
end for
return AR;
```

happen as soon as an appropriate job is selected, and it is also used by Algorithm 1 for n times to estimate load balance feature of each job. In this algorithm, each map task will be assigned to the map machine where its input data is located, if there are more than one machine (a common case in MapReduce systems, given data replicas), the task will be assigned to the lowest load machine.

Finally, an load balance-aware reduce tasks scheduling algorithm is proposed (see Algorithm 3 for details.), which happens after all map tasks of the same job are finished. For each reduce task, assign it to a reduce machine with the lowest load, then update the load of this machine.

4.2 Discussions of Time Complexity

First of all, we discuss the time complexity of Algorithm 2 and Algorithm 3, which have impaction on time complexity of Algorithm 1. Obviously, for a given cluster, N_m and N_r are both constants, so the time complexity of Algorithm 2 and Algorithm 3 are $O(M)$ and $O(R)$, in which M and R represent the task numbers of map phase and reduce phase respectively of the selected job.

Then, we consider the complexity of Algorithm 1, which contains a possible non-constant computation step (step 2) in each loop. Given the load of each map machine can be acquired from each machine, the time complexity of this computation step is actually the complexity of Algorithm 2 plus $O(N_m)$, so complexity of Algorithm 1 is $O(n * (N_m + O(M))$, in which N_m is a constant for a given cluster, and n is also a constant for off-line scheduling, in which all jobs have arrive at the beginning of scheduling, so Algorithm 1 has the same time complexity with Algorithm 2 under off-line scheduling; however, under on-line condition, which is a more popular mode, n is a variable which depends on the distribution of job arriving time, the job scheduling frequency, so Algorithm 1 has $O(n * M)$ complexity for on-line scheduling. Obviously, denser distribution of job arriving time is, bigger n is, more frequently jobs are scheduled, smaller n will be. Smaller n makes smaller job selection space, resulting in smaller optimization space, however, smaller n will make better time complexity of Algorithm 1. So, we adopt an adaptive job scheduler opportunity, i.e. dynamically decide when to schedule the next job. We set a threshold for n to prevent it is too big. Under this constraint, we delay job scheduling so long as all map machines are busy.

5. Evaluations

In this section, we do some simple experiments to evaluation of our scheduling strategy. Two in-used Hadoop scheduling strategies are chosen as comparing objectives, one is FIFO scheduling and the other is Hadoop Fair scheduling (HFS), for each scheduling strategy, we consider two sub-strategies, one is "best effort" data locality and the other is strict data locality. So, the actually comparing objectives are FIFO with "best effort" data locality (denoted as FIFO-BE), FIFO with strict data locality (denoted as FIFO-S), HFS with "best effort" data locality (denoted as HFS-BE) and HFS with strict locality (denoted as HFS-S). Our efficiency-aware scheduling strategy (denoted as EAS).

5.1 Experimental Setup

The cluster consists of 21 identical computers, one of which is the master, i.e. the JobTracker, responsible for management and scheduling. The other computers act as slaves, which are named TaskTrackers, 15 of which act as map machines and provide data service of HDFS (we configure the block size as 128MB which is used and considered more efficient than the default 64MB in Facebook as [17] said.) and take care of map tasks running, the other 5 act as reduce machines and take the responsibility of executing reduce tasks. Each of the TaskTrackers has the cpu of Intel (R) Xen (R) CPU x3220 2.40GHz, 4GB Ram, 500GB disk, Ubuntu 10.10 server version operating system, jdk1.6.0.2 and hadoop-0.20.2-cdh3u0 distributed by Cloudera company.

In our evaluations, we adopt the default slot number defined in the configuration file named as 'mapred-default.xml' in the 'src' directory, i.e. each map machine has 2 map slots, and each reduce machine has 2 reduce slots.

Actually, in real production environments, nearly all MapReduce jobs have a common feature that map task number is much larger than reduce task number; especially, there are some map-only jobs, which do not need reduce phase, such as select operation in [21]. So we think 15 map machine plus 5 reduce machine can be a representation, which maybe a small cluster instance, evaluations under large cluster will be conducted in the near future. Given that we choose a job from the waiting job set and assign map tasks and reduce tasks of the appropriate job based on the real time load on each machine, to simplify load computation,

Table 1 A common workload in all evaluations.

Job type	#Maps	#Reduces	#Jobs
wordcount	1	1	25
sort	2	1	14
wordcount	5	2	7
sort	30	10	4

we adopt dedicated workers, i.e. map machines and reduce machines.

Now, we explain why the abovementioned settings will not be disadvantageous to other scheduling algorithms. For the configuration of slots, which defines the parallelizable map tasks and reduce tasks per machine, though they can be random configured, should consider the physical parallelism of each machine, such as CPU core number and disk number. We think if slot number is too small, the inner parallel capability can't be perfectly utilized, on the other hand, it will lead to the resource contention on some machines and resource underutilization on other machines to set the slot number too large. So, we think the impact depends on the relationship between slots configuration and physical configuration of each machine, rather than the scheduling algorithms. We think map tasks are usually IO-bound while reduce tasks are communication-bound, i.e. they use different kinds of resource, so dedicated worker strategy and the default mixed worker strategy have the same impact on different scheduling algorithms.

5.2 Simulated Workload

We choose two types of classical MapReduce jobs (wordcount and sort). For wordcount, we permit the combine function during map phase to optimize network I/O, and for both, pipeline between map and reduce of the same job is forbidden, backup write of output in map phase is permitted. Based on these two type of jobs, we generate a common submission of a workload consisting of 50 jobs which are depicted in Table 1, and the submission times are randomly generated among the time interval [0,120]. We generate map number, reduce task numbers and job number based on part of the workload illustrated in [17] which said it is based on real job distribution in Facebook production environment.

In our experiments, the overall map input is about 26GB, under our scheduling algorithm they are all from local disks. However, other previous scheduling algorithms cannot achieve this. Total output of map tasks, i.e. the total input of reduce tasks, which should be transferred over network from map machines to reduce machines, is about 20GB, 91% of which is from sort MapReduce jobs and 9% is from wordcount MapReduce jobs. Under our scheduling algorithm, we try to make these intermediate data transferred over the network more evenly in the timeline and among reduce machines at each time point according to the pipeline principle illustrated as Fig. 2.

5.3 Data Locality Evaluation

In MapReduce environment, high data locality is considered as a prerequisite to achieve high throughput. So, we first compare data locality of different scheduling strategies. Its metric is the percentage of local map tasks under a give scheduling strategy, formalized as (6), where LM represents the number of local map tasks and TM represents the total number of map tasks.

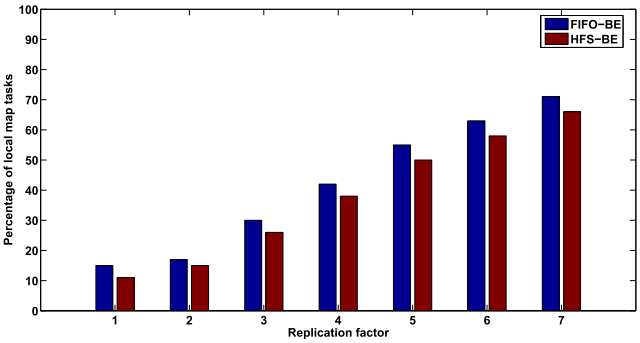

Fig. 3 Data locality comparisons.

$$Percentage = LM/TM \tag{6}$$

Actually, our scheduling strategy and other strict data locality scheduling strategies (such as FIFO-S and HFS-S) always have the optimal data locality, the aim of comparison is to study how data locality is for "best effort" data locality scheduling strategies (such as FIFO-BE and HFS-BE), and the relationship between the replication factor of input and data locality. The comparison results are illustrated as Fig. 3).

From Fig. 3, we see both of FIFO-BE and HFS-BE suffer from poor locality problem. Increasing replication factor can help to improve data locality, however, this is not an advisable method, given huge storage space waste and cost of data replication. For large MapReduce cluster which consists of thousands of computing nodes like Google, Yahoo! and Facebook, the default replication factor is only three. Thus, we think data locality problem on current Hadoop schedulers will prevent performance improvement.

5.4 Load Balance Evaluation of Map Machines

Except for data locality, load balance is considered as another factor which has serious impact on throughput of map machines. So, in this sub-section, we evaluate the load balance feature among map machines during system running.

Given all map machines have the same processing rate if data is on local disk and all map tasks are strictly local, we use the size of un-processed data of un-finished map tasks on a map machine to represent its load. For "best effort" data locality scheduling strategies, given map tasks will read data from other machines, we assume the load of the a map machine as two parts, one is the input size of local map tasks, the other is input data size of remote map tasks which is reading data from it. For EAS, FIFO-BE and FIFO-S, when there are $t = i * N_m$ ($i \in \mathbb{N}$ and $i \geq 1$) map tasks scheduled, we compute the load of each map machine (denoted as L_m^t,

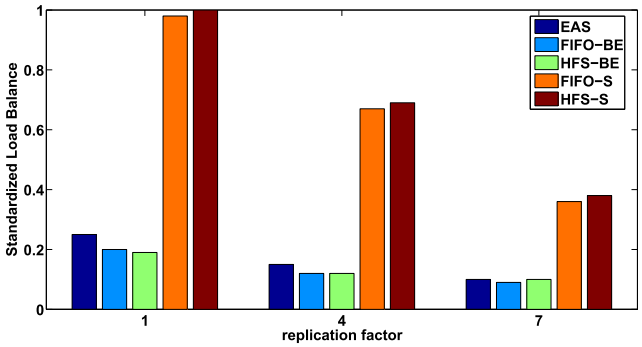

Fig. 4 Load balance comparison.

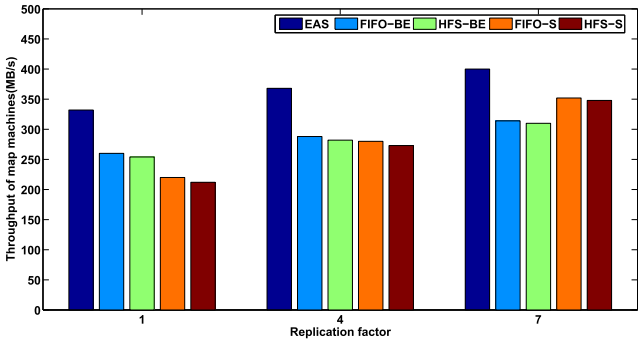

Fig. 5 Comparison of total throughputs on map machines.

where $1 \leq m \leq |N_m|$), then compute the average load among this machines (denoted as $\overline{L^t_{N_m}}$), next we compute the standard deviation as load balance value (denoted as $LB^t_{N_m}$) at this time point, last we compute $\sum_t LB^t_{N_m}$ as the final load balance value. To simplify the representation, we normalize final load balance value to [0,1]. Further, we study the relationship between load balance value and replication factor. The results are illustrated through Fig. 4.

From the results, we can see EAS, FIFO-BE, and HFS-BE have much better load balance feature than FIFO-S and HFS-S. We think this is the reason why current Hadoop schedulers all adopt "best effort" data locality, strict data locality have serious load balance problem, resulting some machines are overloaded while others are still not fully utilized. However, we think the key reason is current job scheduling strategies are not optimized for efficiency of the cluster, through rational job sequencing, we can achieve both data locality and load balance, EAS has nearly the same load balance feature as "best effort" scheduling strategies. When replication factor increases, load balance values of all scheduling strategies become better, and FIFO-S and HFS-S decrease much faster than others.

5.5 Throughput Evaluation of Map Machines

Now, we compare the throughput of map machines on different scheduling strategies. Throughput value is computed according to (1) and (3).

Figure 5 illustrates the results, we can see EAS has the highest throughput of map machines. When replication factor is 1, FIFO-BE and HFS-BE have higher throughput than FIFO-BE and HFS-BE; when replication factor is 7, FIFO-S and HFS-S are better. Throughputs of all scheduling strategies increase when replication factor increases, however, strict data locality scheduling strategies grown faster than "best effort" data locality schedulings, we think the reason is in these schedulings, data locality depends not only on replication factor but also on how a map task is scheduled, when a free node request a task, scheduler try best to search a local map task from the first job (in FIFO-BE) or the most unfair job (in HFS-BE) rather than from all jobs, which confines the possibility of finding a local task, so they still have poorer data locality than strict data locality strategies. On the other hand, when replication factor increases, strict data locality schedulings can better balance the load among map machines. So, their overall throughputs perform better.

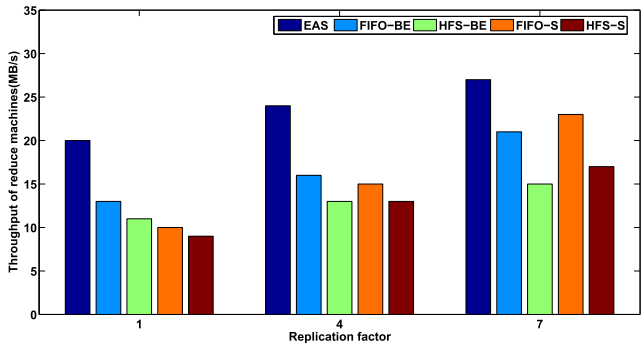

Fig. 6 Comparison of total throughputs on reduce machines.

5.6 Throughput Evaluation of Reduce Machines

In reduce tasks scheduling, given each task's input usually scatters all map tasks evenly, data locality is usually not considered, in which case load balance among reduce machines will be easy to achieve if there are enough schedulable reduce tasks. However, reduce tasks only become schedulable when all map tasks of the same job completes. So, we think the productivity of schedulable reduce tasks has significant impact on the throughput of reduce machines.

In our study, we allocate resource on job level to eliminate resource contention, i.e. if a job is scheduled, all its map tasks should be scheduled before next selected job, which is like FIFO, instead of task level, such as HFS [16] and Delay scheduling [17], in which too many jobs contend for limited resource if there are lots of continuous jobs to be processed, resulting in poor utilization and throughput of reduce machines. Now, we compare different scheduling strategies on the throughput of reduce machines. Throughput value is computed according to (2) and (4). Figure 6 gives the comparison results, from which we can conclude that our efficiency aware scheduling has the best throughput of reduce machines, although there is only 10 MB/s average throughput on each reduce machines at best (when replication factor is 7) .

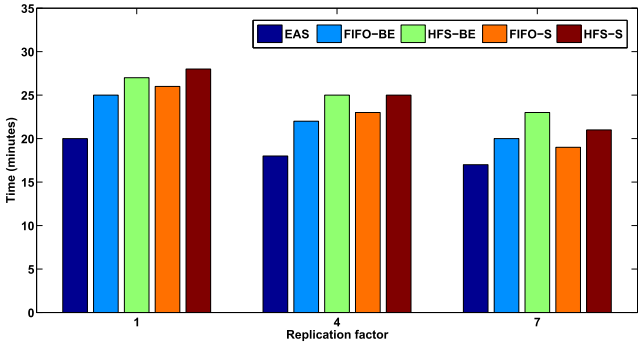

Fig. 7 Comparison of flowtimes.

5.7 Evaluation of flowtime

Above evaluations are indirect evaluations of efficiency of the whole system which are based on the throughput of map machines and reduce machines. For more really evaluating the efficiency of our scheduling strategy, we compare the total *flowtime* for a cluster jobs under different schedulings, the total *flowtime* metric has already been defined in 3.4. The results are described by Fig. 7, from which, we find the total *flowtime* of EAS is always the best under the same replication factor, however, when replication factor increases, the total *flowtime* of EAS does not decrease as much as others, FIFO-S and HFS-S have the most obviously reduction of *flowtime*.

6. Conclusion and Future Work

In this paper, motivated by current job scheduling strategies have either poor data locality problem or poor load balance problem, resulting in inefficiency, we proposed an efficiency-aware scheduling strategy, to achieve this strategy we designed a job selection algorithm based on load balance feature, a strict data locality map tasks scheduling algorithm and a load balance aware reduce tasks scheduling algorithm. By comparing with two popular Hadoop scheduling strategies, we find our strategy performs best on several metrics.

For simplicity, in this study, we adopt push-based task scheduling. However, pull-based task scheduling is more flexible, and easy to fault-tolerance, in fact, our strategy is independent of the implementation of task scheduling. So, in next work, we want to implement our job scheduling strategy under pull-based tasks, in which how to estimate free machine set will be very important when doing job selection. What is more, Fair scheduling strategy is an important work under multi-user and multi-job environment, however, when job number is too big, fairness among all submitted jobs will hurt performance seriously, in which case, our scheduling strategy can combine with Fair scheduling strategy to make fairness among a smaller job set, which will improve the data locality of HFS, this is also my future direction.

Acknowledgements

This research is supported by the Key Technologies R&D Program of China (No. 2012BAH38B-04 and 2012BAH38B06), National High-Tech R&D Program of China (No. 2010AA012505, 2011AA010702, 2012AA01A401 and 2012AA01A402), National Natural Science Foundation of China (No. 60933005), National Information Security 242 Program of China (No. 2011A010).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," Proc. 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, p.10, Berkeley, CA, USA, 2004.
[2] N. Archak, V.S. Mirrokni, and S. Muthukrishnan, "Mining advertiser-specific user behavior using adfactors," Proc. 19th International Conference on World Wide Web, WWW '10, pp.31–40, New York, NY, USA, 2010.
[3] A. Goyal, F. Bonchi, and L.V. Lakshmanan, "Learning influence probabilities in social networks," Proc. Third ACM International Conference on Web Search and Data Mining, WSDM '10, pp.241–250, New York, NY, USA, 2010.
[4] D. Hillard, S. Schroedl, E. Manavoglu, H. Raghavan, and C. Leggetter, "Improving ad relevance in sponsored search," Proc. Third ACM International Conference on Web Search and Data Mining, WSDM '10, pp.361–370, New York, NY, USA, 2010.
[5] G.S. Sadasivam and G. Baktavatchalam, "A novel approach to multiple sequence alignment using hadoop data grids," Proc. 2010 Workshop on Massive Data Analytics on the Cloud, MDAC '10, pp.2:1–2:7, New York, NY, USA, 2010.
[6] M. Suzanne and W. Tiffani, "Mrsrf: an efficient mapreduce algorithm for analyzing large collections of evolutionary trees," 2010.
[7] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on mapreduce," Proc. 1st International Conference on Cloud Computing, CloudCom '09, pp.674–679, Berlin, Heidelberg, 2009.
[8] J.H. Böse, A. Andrzejak, and M. Högqvist, "Beyond online aggregation: parallel and incremental data mining with online mapreduce," Proc. 2010 Workshop on Massive Data Analytics on the Cloud, MDAC '10, pp.3:1–3:6, New York, NY, USA, 2010.
[9] Q. Gao and S. Vogel, "Training phrase-based machine translation models on the cloud open source machine translation toolkit chaski. the prague," Bulletin of Mathematical Linguistics, vol.93, no.1, pp.37–46, 2010.
[10] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," Commun. ACM, vol.53, no.1, pp.72–77, Jan. 2010.
[11] YAHOO!, "Scaling hadoop to 4000 nodes at yahoo!," Website. http://developer.yahoo.com/blogs/hadoop/posts/2008/09/scaling_hadoop_to_4000_nodes_a/
[12] Apache, "Hadoop," Website. http://hadoop.apache.org
[13] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," Proc. 35th SIGMOD International Conference on Management of Data, SIGMOD '09, pp.165–178, New York, NY, USA, 2009.
[14] S. Ghemawat, H. Gobioff, and S.T. Leung, "The google file system," SIGOPS Oper. Syst. Rev., vol.37, no.5, pp.29–43, Oct. 2003.
[15] Apache, "Hdfs architecture guide," Website. http://hadoop.apache.org/docs/hdfs/current/hdfs_design.html
[16] Apache, "Hadoop fair scheduler," Website. http://hadoop.apache.org/docs/mapreduce/r0.21.0/fair_scheduler.html
[17] 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," Proc. 5th European conference on Computer systems, EuroSys '10, pp.265–278, New York, NY, USA, 2010.
[18] "Torque resource manager," Website. http://www.adaptivecomputing.com/products/open-source/torque/
[19] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience," Concurrency and Computation: Practice and Experience, vol.17, no.2-4, pp.323–356, 2005.
[20] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads," Proc. VLDB Endow., vol.2, no.1, pp.922–933, Aug. 2009.
[21] J. Dittrich, J.A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," Proc. VLDB Endow., vol.3, no.1-2, pp.515–529, Sept. 2010.
[22] F.N. Afrati and J.D. Ullman, "Optimizing joins in a map-reduce environment," Proc. 13th International Conference on Extending Database Technology, EDBT '10, pp.99–110, New York, NY, USA, 2010.
[23] P. Michiardi, A. Barbuzzi, and D. Carra, "Shared cluster scheduling: a fair and efficient protocol," 2011.
[24] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.L. Wu, and R. Vernica, "Circumflex: a scheduling optimizer for mapreduce workloads with shared scans," ACM SIGOPS Operating Systems Review, vol.46, no.1, pp.26–32, 2012.
[25] M. Hammoud, M.S. Rehman, and M.F. Sakr, "Center-of-gravity reduce task scheduling to lower mapreduce network traffic," 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp.49–58, 2012.
[26] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," Proc. SPAA, 2011.
[27] C. He, Y. Lu, and D. Swanson, "Matchmaking: A new mapreduce scheduling technique," Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on, pp.40–47, 2011.
[28] M.R. Garey, D.S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," Mathematics of Operations Research, vol.1, pp.117–129, 1976.

Hui Zhao was born in Liaoning Province of China in 1982. He is a Ph.D. student at School of Computer in National University of Defense Technology. His interests include big data, database, data mining, cloud computing and scheduling. Currently, his research is mainly on scheduling for data-intensive computings.

Shuqiang Yang was born in Hunan Province of China in 1969. He is a Ph.D. supervisor and professor at National University of Defense Technology. His current research interests include big data management, data warehouse and cloud computing.

Hua Fan was born in Shanxi Province of China in 1984. He is a Ph.D. student at School of Computer in National University of Defense Technology. His research interests include stream data management and Sensor network.

Zhikun Chen was born in Fujian Province of China in 1983. He is a Ph.D. student at School of Computer in National University of Defense Technology. His interests include big data management and cloud computing.

Jinghu Xu was born in Chongqing City Province of China in 1985. He is a Ph.D. student at School of Computer in National University of Defense Technology. His current interests include data mining and intrusion detection. Currently, he works on Alert Correlation in the domain of network security.