A Cross-job Framework for MapReduce Scheduling

Xuejie Xiao, Jian Tang, Zhenhua Chen, Jielong Xu and Chonggang Wang

***Abstract*—In this paper, we present a novel cross-job framework for MapReduce scheduling, which aims to minimize the total processing time of a sequence of related jobs by combining reduce and map phases of two consecutive jobs and streaming data between them . The proposed framework has the following desirable properties: (1) It can accelerate the execution of a sequence of related MapReduce jobs by achieving a good tradeoff between data locality and parallelism. (2) It can support all the existing MapReduce applications with no changes to their source code. (3) It is a general framework, which can work with different scheduling algorithms. We built a new MapReduce runtime system called *cross-job Hadoop* by integrating the proposed cross-job framework into Hadoop. We conducted extensive experiments to evaluate its performance using PageRank and an Apache Pig application. Our experimental results show that the cross-job Hadoop can significantly reduce both the total processing time of a job sequence and the size of data transferred over the network.**

***Index Terms*—Big Data, MapReduce, Task Scheduling, Resource Management.**

I. Introduction

MapReduce [9] has become the *de facto* model for big data processing. With the problems becoming more and more complicated, we are more often in a situation where multiple MapReduce jobs need to be chained together to solve a problem such that the output of one job is the input of another job. Popular MapReduce applications, such as PageRank [15] and many Apache Pig applications [3], fall into this case.

Given a sequence of related jobs, current MapReduce systems, such as Apache Hadoop [1], usually treat each job as an individual and process them one by one, which is obviously not efficient. In this paper, we present a novel cross-job framework for MapReduce scheduling, which aims to minimize the total processing time of the job sequence by combining reduce and map phases of two consecutive jobs and streaming data between them. The proposed framework has the following desirable properties:

(1) It can accelerate the execution of a sequence of related MapReduce jobs by achieving a good tradeoff between data locality and parallelism.
(2) All the existing MapReduce applications can be run on the MapReduce system integrated with the proposed framework without any changes to their source code since our framework does not violate any fundamental constraints of a MapReduce system.
(3) It is a general framework, into which any task scheduling algorithm can be integrated.

We validated our idea by integrating the proposed framework into Hadoop and building a new MapReduce system called *cross-job Hadoop*. We conducted extensive experiments to evaluate its performance using PageRank and an Apache Pig application. The experimental results show that the cross-job Hadoop can significantly reduce the total processing time of the job sequence and the size of data transferred over the network. Note that in the cross-job Hadoop, if a MapReduce application only involves a single job (instead of a sequence of jobs), it will be handled in the same way as that in the original Hadoop.

Xuejie Xiao, Jian Tang, Zhenhua Chen and Jielong Xu are with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse. Email: {xxiao04, jtang02, zchen03, jxu21}@syr.edu. Chonggang Wang is with InterDigital Communications Inc. This research was supported by NSF grant #1218203.

II. System Model

In this section, we briefly introduce the system model and describe the scheduling problem that needs to be solved.

As mentioned above, we focus on MapReduce applications that involve a sequence of jobs where the output of one job is the input of another job. A corresponding job scheduling problem needs to be efficiently solved: given a sequence of n jobs (each job is further divided into multiple map tasks and reduce tasks) and m workers, find an assignment that assigns tasks to workers for processing such that the total processing time (a.k.a. makespan, i.e., from when the first task starts to when the last task ends), is minimized subject to the following constraints:

- For a worker p, it can only execute one task at any time. In other words, if multiple tasks are assigned to p, these tasks will be executed sequentially.
- A task s can only start when all the tasks that it depends on are completed (the set of tasks that task s depends on is denoted as D_s). Particularly, in a job, all its map tasks must be completed before its reduce tasks start to run.

Given a sequence of jobs, current MapReduce implementations usually treat each job as an individual and process them one by one, which is illustrated in Fig. 1. In this example, there are two MapReduce jobs and the output of the first job is the input of the second job. The first job has 3 map tasks and 2 reduce tasks, while the second one includes 4 map tasks and 2 reduce tasks. In the figure, we use a widely-used task graph to show these tasks and their dependencies. In the task graph, each vertex corresponds to a task (map or reduce) and there is a link from vertex u to vertex v if the task corresponding to u must be completed before the task corresponding to v. The map tasks of the first job will be assigned to workers to run first. When they are all completed, the reduce tasks of the first job will be executed. The output of the first job will then be written into data storage (usually a distributed file system such as Hadoop Distributed File System (HDFS) [1].

The processing time of a task s on worker p, $T(s,p)$, consists of two parts: 1) the time needed to transfer the outputs of all tasks s depends on to worker p, which is called *transfer time*; and 2) the time needed to process task s on worker p, which is called *running time*. In order to minimize the makespan, an important tradeoff between data locality and parallelism needs to be addressed: On one hand, running time

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

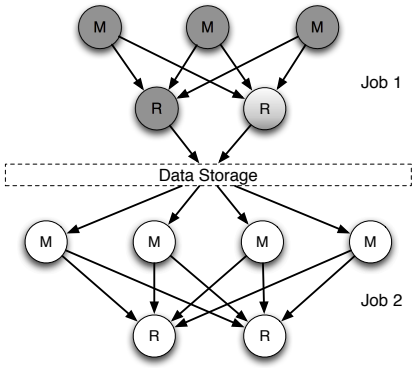


Fig. 1. Original MapReduce scheduling

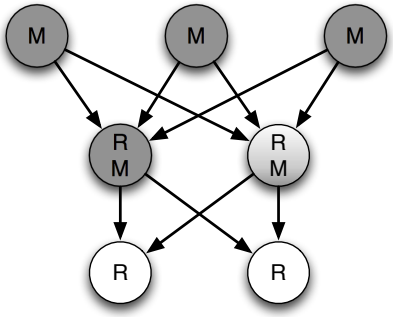


Fig. 2. Cross-job scheduling

can be significantly reduced by running tasks on as many as possible workers concurrently; on the other hand, transfer time can be greatly reduced by running tasks on a small subset of workers.

III. Cross-job Scheduling Framework

In this section, we present the proposed cross-job scheduling framework.

In most current MapReduce systems, for a sequence of jobs, map tasks in one job need to wait for all reduce tasks in previous job to be completed. In other words, even if the input data for a map task has already become available, it still needs to wait for other irrelevant reduce tasks. If there are a large number of reduce tasks, this delay may be significant.

Another problem of the original MapReduce scheduling model lies in storage. In most cases, we only care about the final output of the job sequence, which is the output of the final MapReduce job. Hence there is no need to store outputs of intermediate jobs. For those tasks running on the same worker, we can simply keep output of intermediate jobs in memory. We can accelerate job processing further by combining a reduce task with a map task of the next job and streaming output records between them. However, in order to achieve combination and streaming, we need to ensure that the number of reduce tasks of the current job is the same as the number of map tasks of the next job. For example, in Fig. 1, the first job has 2 reduce tasks, while the second one has 4 map tasks. In order to use the proposed cross-job scheduling framework, we need to adjust the number of map tasks of the second job to 2. Each map task is combined with a reduce task of the first job to form a new type of task, namely, *reduce-map* task, as shown in Fig. 2. We will discuss how we adjust the map task number in our implementation in Section IV.

We argue that this adjustment will not affect the performance: A typical MapReduce cluster has much more (usually over 100 times more) tasks than workers. With a well-designed partitioning function, it can be ensured that workloads are well balanced and every worker has roughly the same amount of data to process. Even though our adjustment may reduce the number of map tasks (such that each map task has more data to process), it will not change the total amount of data to be processed by each worker (with the help a well-designed partitioning function that can achieve good load balancing), which has been verified by our Haddop-based implementation and experiments. Moreover, less tasks can even reduce the overhead and save time needed to transfer data between tasks.

In a new task graph based on cross-job scheduling, which we will call *cross-job task graph* in the following, each vertex corresponds to a map/reduce/reduce-map task. The cross-job task graph can be viewed as a layered graph with $(n+1)$ layers (where n is the number of jobs), as shown in Figure 2. In the cross-job task graph, the tasks in the first layer are map tasks of the first job; the tasks in the intermediate layers are reduce-map tasks; and the tasks in the last layer are reduce tasks of the last job. It is easy to see that a cross-job task graph has the following property: a task s in layer i ($2 \leq i \leq n+1$) can start if and only if all tasks in layer $(i-1)$ are completed. This is because of a fundamental scheduling constraint of a MapReduce system: in a MapReduce job, a reduce task can start only and only if all map tasks are completed. This desirable property enables us to schedule tasks on a layer-by-layer basis.

In each layer, it does not matter in which order the tasks assigned to a worker are executed since these tasks are independent. Hence, a schedule in this case is essentially a worker assignment $\mathcal{A}$ which specifies which worker a task s is assigned to. Then the total processing time (makespan) of an assignment $\mathcal{A}$, $T(\mathcal{A}) = \max_{p \in P}(\sum_{\mathcal{A}(s)=p} T(s,p))$, where $T(\cdot)$ gives the time needed to process task s on worker p. Since our goal is to minimize the makespan, we can solve the following scheduling problem in each layer:

Definition 1 (Single-Layer Scheduling (SLS)): Given a set S_i of tasks in layer i and a set P of workers, along with the time needed to process task $s \in S_i$ on worker $p \in P$, $T(s,p)$, the *Single-Layer Scheduling (SLS)* problem seeks an assignment $\mathcal{A} : S_i \mapsto P$ such that the makespan $T(\mathcal{A})$ is minimized.

Mathematically, the SLS problem is the same as the *problem of scheduling tasks on unrelated processors* [8], which has been studied by a few works in the literature. The processors are *unrelated* in the sense that the time required for processing a task on a processor is a function of both the task and the processor, in other words, it is not always the case in which a fast processor requires less time than a slow processor, irrespective of the task being executed [8]. This model is quite suitable here because as mentioned above, the processing time of a task consists of transfer time and running time.

Schedule Map tasks in job 1 using an SLS algorithm;

for $i \leftarrow 2$ **to** n **do**
    Adjust the number of map tasks in job i;
    Combine reduce tasks in job $i-1$ and map tasks in job i to form reduce-map tasks;
    Schedule them using an SLS algorithm;
end

Schedule reduce tasks in job n using an SLS algorithm.

Algorithm 1: The cross-job scheduling framework

The problem of scheduling tasks on unrelated processors has been shown to be NP-hard [8]. Several approximation algorithms and heuristic algorithms have been proposed in the literature. The proposed framework does not put any restriction on scheduling algorithms. Hence, any algorithm that can solve this problem can be used as a subroutine for SLS in our framework. The algorithm presented in [14] has an approximation ratio of 2, which is the best-known approximation ratio for this problem. However, this algorithm is a complicated Linear Programming (LP) rounding based algorithm, which needs to solve an LP problem multiple times. Therefore, it has a high time complexity and more importantly, it is difficult to implement in a real system. We adopt the simple and fast greedy algorithm presented in [8] for SLS in our implementation. The algorithm is an adaptation of the list scheduling algorithm which has often been used for scheduling problems. It starts by constructing a list of the tasks, either in an arbitrary order or in accordance with some heuristic. The algorithm deals with the tasks on the list one by one, and every time it assigns the next task on the list to the processor that will first complete the tasks it has already been assigned. It has been shown that the makespan given by the algorithm is guaranteed to be at most $(1+\sqrt{2})\sqrt{m}$ times longer than optimal, and has a time complexity of $O(mn \log n)$, where n and m are the numbers of tasks and processors respectively. We omit some details here, which can be found in [8]. This algorithm is easy to implement, and works very well on average cases, which has been verified by our experiments.

The special features and benefits of our cross-job scheduling framework are summarized as follows:

- It is a general scheduling framework, into which any algorithms that solves the SLS problem can be integrated.
- It can reduce the total processing time of a job sequence by combining reduce and map phases in two consecutive jobs and streaming data between them, and by achieving a good tradeoff between data locality and parallelism using an efficient SLS algorithm.
- It honors all the original MapReduce scheduling constraints. Hence, users do not need to change the source code of their MapReduce applications. Moreover, it can guarantee the correctness of the final results.

IV. Implementation

In this section, we present the details of integrating the proposed cross-job MapReduce framework into Apache Hadoop [1].

We chose Hadoop to build our system since it is currently the most popular open-source MapReduce system and quite a few popular MapReduce application tools, such as Apache Pig [3], were built based on Hadoop. We call the modified Hadoop system, *cross-job Hadoop*. In our implementation, the original Hadoop API is preserved as much as possible. Hence, users of Hadoop can run their applications on our cross-job Hadoop system without changing their map and reduce functions.

A. Implementation of Reduce-Map Tasks

The key idea of our cross-job scheduling framework is the combination of reduce tasks and map tasks (in the next job). So we discuss how to implement such a combination first. In the Hadoop, the classes *MapTask* and *ReduceTask* are in charge of running map and reduce tasks respectively. Hadoop creates a MapTask object for each map task, which first reads the meta information of the input split. Then the actual input data are fetched based on the locations in the meta information. The input records are thus fed into the (user-defined) map function sequentially. When the intermediate results are gathered, the MapTask object first splits the output records into partitions according to the partitioning function. The output data are then written into local machine's storage on a partition basis.

Similarly, a ReduceTask object is created for each reduce task. It monitors the completion events of all map tasks. When a map task is completed, it queries the corresponding machine via the task tracker, fetches the input, then sorts existing partial inputs. When all the input parts are collected and sorting is finished, it groups those records according to their keys, then uses the grouped records as input to launch the reduce function. After the reduce function processes these data and produces output, the output is first written to local storage, then is copied to the HDFS for future use (probably for other jobs).

In our implementation, a new class called *ReduceMapTask* is created to handle reduce-map tasks. Whenever such a task is started, a ReduceMapTask object is created from the class. It performs operations of both a ReduceTask object and a MapTask object as described above. Our ReduceMapTask behaves similarly as the original MapTask in the sense that it only writes into local storage. Hence, in a job sequence, for all the jobs except the last one, the output will be written into local storage (instead of HDFS), which can save transfer time.

As described above, to apply the proposed cross-job scheduling framework, particularly to achieve the combination, we need to ensure that the number of map tasks in a job (except the first one) is the same as the number of reduce tasks in the previous job. In the original Hadoop, the number of map tasks is determined by the number of splits of input data, and the user specifies the number of reduce tasks. In the cross-job Hadoop, similarly, the user specifies the number of reduce tasks for each job in the configuration. Accordingly,

the runtime system will create the same number of map tasks in each layer to form reduce-map tasks. Note that for different jobs, the number of reduce tasks can be set differently. So the cross-job Hadoop does not add any additional burden to the user.

B. Job Execution Logic

When running a job sequence, the original Hadoop only allows one job to be running at a time. To use the proposed framework, we need to allow multiple jobs to be in running state. For all but the last job, the output is only written to local storage rather than HDFS, which will be cleaned up as soon as the current job is completed. However, to use our cross-job framework, a job needs to read the output of the previous job in the local storage. In other words, a job needs to remain in the running state until the next job obtains necessary data from its local storage. In addition, the Hadoop system leverages early copying to speed up data processing. Specifically, as long as 5% of all map tasks are finished, some of the ReduceTask objects start to run to copy the intermediate results to the corresponding workers.

With these considerations, the cross-job Hadoop allows multiple jobs to be in the running state. In the worst case, 4 jobs need to be in the running state, which are described in the following:

- Job k: Cleanup phase;
- Job $k+1$: All its reduce-map tasks have finished, but it still waits for all reduce-map tasks from the next jobs to be finished, so the cleanup phase cannot start;
- Job $k+2$: Some but not all of its reduce-map tasks are finished;
- Job $k+3$: Some of its reduce-map tasks start running due to early copying.

Even though 4 jobs might be in the running state in the cross-job Hadoop system, at a particular time, the system only runs tasks for one job (It is Job $k+2$ in the example presented above). For all the other running jobs, the system just either keeps their data still available or does some preparation works such as early copying. Hence, compared to the original Hadoop, the performance of the cross-job Hadoop will not be degraded because of this change.

C. Estimation of Task Processing Time

As mentioned above, we need to estimate or predict the processing time of a task s on worker p before scheduling, which consists of two parts: running time and transfer time. In our implementation, for running time estimation, we assume that it is linear to input size. This model has been used by other works on MapReduce [20]. The data processing rates of map and reduce tasks are obtained via profiling. For the transfer time estimation, we also use a simple method: for local transfer, we assume the transfer time is 0; for remote transfer, we simply assume the transfer time is linear to the size of transferred data. Similarly, the transmission rate is estimated via a traditional profiling method. Our experimental results show that this simple estimation method works very well in our Hadoop cluster.

Our cross-job framework does not put any constraints on the estimation method and our implementation ensures that any method can be easily integrated into our system.

V. Performance Evaluation

In order to justify the effectiveness of the proposed scheduling framework, we conducted experiments to compare our cross-job Hadoop with the stable version of Hadoop system using two popular MapReduce applications: PageRank and an Apache Pig application for log file analysis. We discuss the experimental setup and results in the following.

A. Experimental Setup

As described in Section IV, we implemented our cross-job framework on top of Apache Hadoop [1]. We used Cloudera's Distribution Including Apache Hadoop (CDH) 3 Update 4(cdh3u4) [7] for implementation. CDH is an Apache Hadoop distribution published by Cloudera, Inc. Based on the open source Hadoop distribution, it was built specifically to meet enterprise demands. With custom patches provided by Cloudera (also open source), it makes Hadoop more secure, stable and scalable, which is why we chose CDH for our implementation.

The computer cluster used for experiments consists of five physical servers. Each server has a quad-core Intel Xeon E5506 CPU and 18GB memory. Each server has a small disk of 5GB for storing operating system, and all the five servers are also attached to a big disk array providing storage for the Hadoop cluster. On top of these servers, we installed the VMWare vSphere 5.0 [19] to build a virtualized computing environment and created 9 Virtual Machines (VMs) to host the Hadoop system. Each VM was configured to have 2 CPU cores and 8GB memory. One of the VMs was designated as the master of the Hadoop cluster, while the rest eight VMs were assigned as workers.

We tested our system with two popular applications in the experiments: (1) PageRank was chosen to represent iterative applications. Apache Nutch [2] was employed to crawl a total of 20 million webpages into our cluster, from which a giant link graph was built as the input to PageRank. In our experiments, a PageRank application (consisting of n jobs) first started a job to read input from files generated by Apache Nutch, then the intermediate $(n-2)$ jobs performed the actual PageRank computations. Finally, the last MapReduce job wrote the final output to HDFS in a readable format. (2) We used an Apache Pig application to represent applications with heterogeneous jobs. Our choice of Pig Latin script was the Pig script 1 in the Apache Pig tutorial [18], which is also called the query phrase popularity script. It processes a search log file and finds search phrases with particular high occurring frequencies during certain times of a day. In the experiment, the script was transformed into a sequence of three MapReduce jobs. This represents a different use case from the PageRank application: The PageRank application iteratively runs the same MapReduce job many times; the Pig script, however, generates three different MapReduce jobs and run them in a sequence. We used Faker [13] to randomly generate

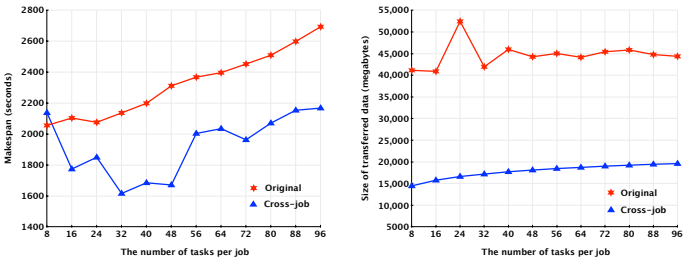
search queries. Our final test set contains around 28 million queries.

Both the number of jobs and the number of (reduce) tasks per job (specified by the user) may make a significant impact on the performance. For the PageRank application, we came up with the following testing scenario: fix the number of jobs to 8, let the number of (reduce) tasks in each job vary from 8 to 96.

In the Apache Pig experiments, we can only control the number of tasks per job because the number of jobs is predetermined by the Pig Latin script. The total processing time of the job sequence (i.e., makespan) and the total number of data bytes transferred over the network were used as performance metrics. In the following figures, we label the cross-job Hadoop system and the original Hadoop system by *"Cross-job"* and *"Original"*, respectively.

B. Experimental Results

First, we present the experimental results corresponding to the PageRank application. Figs. 3(a) and 3(b) show the results obtained from running the PageRank application with different numbers of tasks per job. As mentioned above, there is an important tradeoff between data locality and parallelism: On one hand, in order to achieve better parallelism, we tend to create more tasks; on the other hand, in order to reduce the size of transferred data and transfer time, we tend to have less tasks. Therefore, the makespan does not increase or decrease monotonically with the number of tasks, which can be observed from Fig. 3(a). In our experiments, for the original Hadoop, setting the number of (reduce) tasks per job to 8 or 24 lead to short makespans. The cross-job Hadoop achieved the minimal makespan when this number was set to 32. With this setting, our cross-job Hadoop used 24% less time than the original Hadoop.



(a) Makespans with different number of tasks per job

(b) Transferred data sizes with different numbers of tasks per job

Fig. 3. Performance of cross-job Hadoop with PageRank

Fig. 3(b) shows that the number of data bytes transferred over the network increases slowly with the number of tasks per job. The figure also shows that our cross-job Hadoop can reduce more than half of the number of transferred data bytes by combining map and reduce phases and maximizing local data use.

It is worth mentioning that from these two figures, we can observe that reducing the size of transferred data does not necessarily lead to a shorter makespan. When running the cross-job Hadoop, the minimal transferred data size was achieved when the number of (reduce) tasks per job was set to 8. But the makespan at this point is particularly high (2138 seconds). The minimal makespan was obtained when the number of (reduce) tasks per job was set to 32. This observation validates our earlier claim: we need to achieve a good tradeoff between parallelism and data locality in order to reduce the makespan.

We also want to point out that our Hadoop cluster was set to perform no replications on the data. In a production cluster with replications enabled, more speed boost and transferred data reduction can be expected since it takes time to store the output from one MapReduce job to HDFS multiple times, while this can be avoided using our cross-job framework. The results presented here can serve as the baseline for the performance gains that can be brought by the proposed cross-job scheduling framework. More gains are expected to be achieved if running our cross-job Hadoop in a large-scale production cluster.

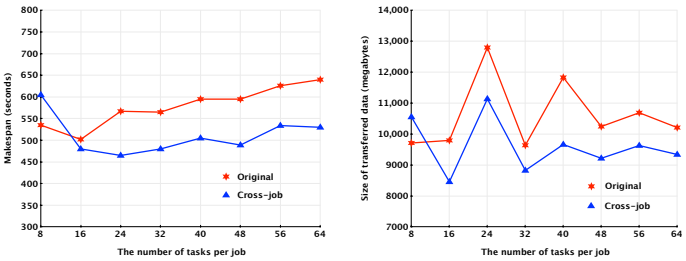
The experimental results corresponding to the Apache Pig application are shown in Figs. 4(a) and 4(b). Similar to the PageRank application, we can see that the minimal makespan was achieved when the number of (reduce) tasks was set to 16 for the original Hadoop, while the cross-job Hadoop gave the minimal makespan when it was set to 24. Essentially, the cross-job Hadoop can achieve a speedup between 4% and 18%. The speedup here is smaller than that for the PageRank application mainly because we had much smaller number of jobs (3 for Apache Pig vs. 8 jobs for PageRank).

In Fig. 4(b), we see more variations on the size of data transferred over the network. When the number of (reduce) tasks per job was set to 24, the cross-job Hadoop achieved the minimal makespan, however, the system experienced maximal number of transferred data bytes. The minimal number of data bytes were transferred when the number of (reduce) tasks per job was set to 16. In this case, 8458 megabytes of data were transferred in the cross-job Hadoop, while 9796 megabytes in the original Hadoop. However, the original Hadoop experienced the minimal number of transferred data bytes (9639 megabytes) when the number of (reduce) tasks per job was set to 32. These observations again verify the claim that reducing the size of transferred data does not necessarily lead to a shorter makespan because of the tradeoff between parallelism and data locality. Overall, the cross-job Hadoop achieved about $8\% - 18\%$ savings on the size of transferred data. As described above, our cluster was configured to use no replications. In a large-scale production Hadoop cluster with replications, the cross-job Hadoop is expected to achieve more speedup and savings on the size of transferred data.

In summary, we can observe from the experimental results that the cross-job Hadoop can significantly reduce both the makespan and the size of transferred data, compared to the original Hadoop. However, in order to achieve the best performance, some parameters, such as the number of (reduce) tasks per job, need to be carefully selected.

VI. Related Work

MapReduce and related resource allocation problems have attracted extensive research attentions recently. In a pioneering



(a) Makespans with different number of tasks per job

(b) Transferred data sizes with different numbers of tasks per job

Fig. 4. Performance of cross-job Hadoop with the Pig application

work [9], Dean and Ghemawat, introduced the MapReduce programming model and a runtime system. In [6], the authors considered the problem of jointly scheduling all three phases of the MapReduce process. They presented guaranteed approximation algorithms and outlined several heuristics to solve the joint scheduling problem. In a recent paper [16], the authors presented Purlieus, a MapReduce resource allocation system aimed at enhancing the performance of MapReduce jobs in the cloud. Purlieus provisions virtual MapReduce clusters in a locality-aware manner enabling MapReduce VMs access to input data and importantly, intermediate data from local or close-by physical machines. Zaharia *et al.* presented a fair scheduler for MapReduce at Facebook in [21], which is focused on multi-user workloads but can also raise throughput for the single-user case. Incoop [4] is a generic MapReduce framework for incremental computations. Incoop detects changes to the input and automatically updates the output by employing an efficient, fine-grained result reuse mechanism. Even though these related works provide great insight into MapReduce scheduling, none of them carefully addressed the case which involves a sequence of related jobs, which, however, is the main focus of this paper.

Several recent works studied the multi-job case for MapReduce systems. The two closest works are [5] and [11]. In [5], the authors presented HaLoop, a modified version of the Hadoop MapReduce framework that is designed to serve iterative applications. HaLoop not only extends MapReduce with programming support for iterative applications, but also dramatically improves their efficiency by making the task scheduler loop-aware and by adding various caching mechanisms. Similarly, the paper [11] presented the programming model and the architecture of Twister, which is an enhanced MapReduce runtime that supports iterative MapReduce computations efficiently. The cross-job framework proposed in this paper is more general since it can not only deal with iterative applications with homogeneous jobs but also applications with heterogeneous jobs (such as Apache Pig applications).

In a recent paper [12], the authors presented ReStore, a system that manages the storage and reuse of intermediate results for MapReduce applications. Our work is also different from ReStore because the objective of ReStore is to maximize the reuse of intermediate results, however, our cross-job framework aims to minimize the total processing time of a job sequence by exploiting the tradeoff between data locality and parallelism.

VII. Conclusions

In this paper, we presented a novel cross-job framework for MapReduce scheduling, which can accelerate the execution of a sequence of related jobs by achieving a good tradeoff between data locality and parallelism. Moreover, all the existing MapReduce applications can be run on the MapReduce system integrated with the proposed framework without any changes to their source code. We integrated the proposed framework into Hadoop and built a new Hadoop system called *cross-job Hadoop*. We conducted extensive experiments to evaluate its performance using the PageRank application and an Apache Pig application. The experimental results show that the cross-job Hadoop can significantly reduce both the total processing time of a job sequence (by over 20% in most cases) and the size of the data transferred over the network (by over 50% in most cases), compared to the original Hadoop.

References

[1] Apache Hadoop, http://hadoop.apache.org/

[2] Apache Nutch, http://nutch.apache.org/

[3] Apache pig, *http://pig.apache.org/*

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquini, Incoop: Mapreduce for incremental computations, *Proceedings of ACM SOCC'2011*.

[5] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, HaLoop: efficient iterative data processing on large clusters, *Proceedings of VLDB Endowment*, Vol. 3, No. 1-2, 2010, pp. 285–296.

[6] F. Chen, M. S. Kodialam, and T. V. Lakshman, Joint scheduling of processing and shuffle phases in mapreduce systems, *Proceedings of IEEE Infocom'2012*, pp. 1143–1151.

[7] Cloudera's distribution including Apache Hadoop(CDH), *http://www.cloudera.com/hadoop*

[8] E. Davis, and J. M. Jaffe, Algorithms for scheduling tasks on unrelated processors, *Journal of ACM*, Vol. 28, 1981, pp. 721–736.

[9] J. Dean, and S. Ghemawat, MapReduce: simplified data processing on large clusters, *Proceedings of USENIX OSDI'2004*.

[10] J. Ekanayake, S. Pallickara, and G. Fox, Mapreduce for data intensive scientific analyses, *Proceedings of IEEE eScience'2008*, pp. 277–284.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, Twister: a runtime for iterative mapreduce, *Proceedings of ACM HPDC'2010*, pp. 810–818.

[12] I. Elghandour, and A. Aboulnaga, Restore: reusing results of mapreduce jobs, *Proceedings of VLDB Endowment*, Vol. 5, No. 6, 2012, pp. 586–597.

[13] Faker, *https://github.com/stympy/faker*

[14] J. K. Lenstra, D. B. Shmoys and E. Tardos, Approximation algorithms for scheduling unrelated parallel machines, *Mathematical Programming*, Vol. 46, 1990, pp. 259–271.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd, The pagerank citation ranking: Bringing order to the web, *Technical Report 1999-66, Stanford InfoLab*, 1999, Previous number = SIDL-WP-1999-0120.

[16] B. Palanisamy, A. Singh, L. Liu, and B. Jain, Purlieus: locality-aware resource allocation for MapReduce in a cloud, *Proceedings of ACM SC'2011*.

[17] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, Evaluating mapreduce for multi-core and multiprocessor systems, *Proceedings of IEEE HPCA'2007*, pp. 13–24.

[18] The Apache Pig tutorial, *http://pig.apache.org/docs/r0.8.1/tutorial.html*

[19] VMware vSphere, *http://www.vmware.com/products/datacenter-virtualization/vsphere*

[20] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, Improving mapreduce performance in heterogeneous environments, *Proceedings of USENIX OSDI'2008*, pp. 29–42.

[21] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, Job scheduling for multi-user mapreduce clusters. *Tech. Rep. UCB/EECS-2009-55*, EECS Department, University of California, Berkeley, 2009.