Future Generation Computer Systems 36 (2014) 80–90

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Morpho: A decoupled MapReduce framework for elastic cloud computing

Lu Lu, Xuanhua Shi *, Hai Jin, Qiuyue Wang, Daxing Yuan, Song Wu

Services Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China

HIGHLIGHTS

- A decoupled MapReduce computing-storage system for cloud computing is proposed.
- Data perception mechanism after decoupling makes tasks read the closest data.
- We design a VM placement strategy to exploit the data locality of tasks.
- A load-aware data placement strategy is complementary to the VM placement.

ARTICLE INFO

Article history:
Received 21 December 2012
Received in revised form
26 October 2013
Accepted 3 December 2013
Available online 19 December 2013

Keywords:
Cloud computing
MapReduce
Data placement
Virtual machine scheduling

ABSTRACT

MapReduce as a service enjoys wide adoption in commercial clouds today [3,23]. But most cloud providers just deploy native Hadoop [24] systems on their cloud platforms to provide MapReduce services without any adaptation to these virtualized environments [6,25]. In cloud environments, the basic executing units of data processing are virtual machines. Each user's virtual cluster needs to deploy HDFS [26] every time when it is initialized, while the user's input and output data should be transferred between the HDFS and external persistent data storage to ensure that the native Hadoop works properly. These costly data movements can lead to significant performance degradation of MapReduce jobs in the cloud.

We present Morpho—a modified version of the Hadoop MapReduce framework, which decouples storage and computation into physical clusters and virtual clusters respectively. In Morpho, the map/reduce tasks are still running in VMs without corresponding ad-hoc HDFS deployments; instead, HDFS is deployed on the underlying physical machines. When MapReduce computation is performing, the map tasks can get data directly from physical machines without any extra data transfers. We design data location perception module to improve the cooperativity of the computation and storage layers, which means that the map tasks can intelligently fetch information about the network topology of physical machines and the VM placements. Additionally, Morpho also achieves high performance by two complementary strategies for data placement and VM placement, which can provide better map and reduce input locality. Furthermore, our data placement strategy can mitigate the resource contentions between jobs.

The evaluation of our Morpho system prototype shows it achieves a nearly 62% speedup of job execution time and a significant reduction in network traffic of the entire system compared with the traditional cloud computing scheme of Amazon and other cloud providers.

1. Introduction

The ability of processing and analyzing big data plays a key role in most modern enterprises today. So, the technology of MapReduce, which has the ability to automatically parallelize the application on a cluster of commodity hardware and can efficiently and quickly process terabytes and petabytes of data, has becomes popular to all sizes of enterprises. It allows the details of distributed execution, network communication and fault tolerance to be handled by the MapReduce framework [1].

This technology is suitable and valuable to businesses, but building and operating even a relatively small cluster can be a formidable undertaking, requiring not just money but physical space, power, and management resources [2]. For example, static data center provisioning for the peak load should lead to underutilization at other times. The virtualization technology widely adopted by most IaaS platforms not only eases the resource management, system administration and deployment for cloud providers, but also allows users to easily customize their executing environments. Alternatively, cloud services (e.g., Amazon

* Corresponding author.
E-mail address: xhshi@hust.edu.cn (X. Shi).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.12.026

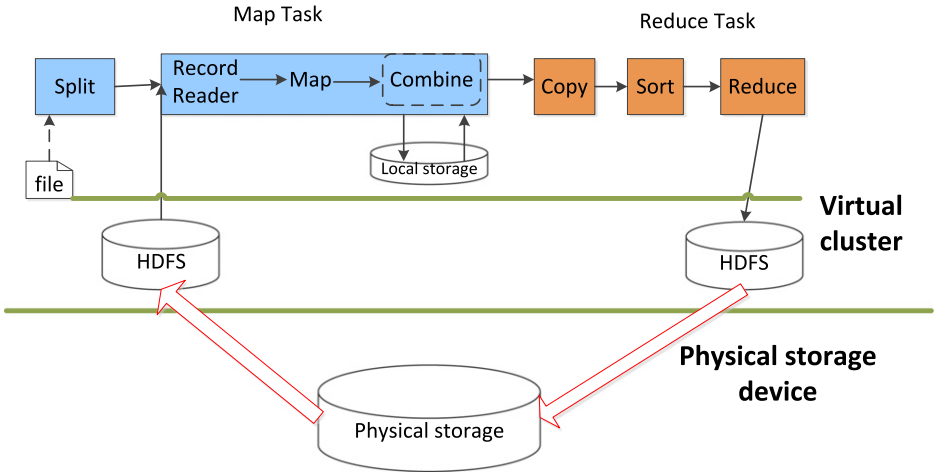


Fig. 1. The traditional cloud computing mode.

Elastic MapReduce) have become a good choice for most enterprises. Amazon Elastic MapReduce works in conjunction with EC2 (Elastic Compute Cloud) and Amazon S3 (Simple Storage Service) [3–5]. The users upload into Amazon S3 the data that they want to analyze, along with the mapper and reducer executables that will process the data, and then send a request to Amazon EC2. Subsequently, Amazon EC2 will start a cluster which loads and runs Hadoop and executes a job flow by downloading data from Amazon S3 onto the cluster of slave nodes. When the data processing is done, the results need to be uploaded back from the cluster to Amazon S3 [6]. That represents the most common cloud computing mode today, just as the Fig. 1 shows.

Currently, most cloud providers directly deploy MapReduce systems to the cloud environment without adaptation to the cloud environment. Both Amazon and Microsoft provide their MapReduce cloud services as extensions of the basic IaaS. Users can simply create virtual MapReduce clusters to cost-effectively analyze their large amounts of data. As described in the technical papers of Amazon Elastic MapReduce and Microsoft Azure HDInsight, the users' virtual clusters for MapReduce computing are hosted on the computing cluster (e.g. EC2), while the input (output) data have to be read from (written to) the storage cluster into (from) the local HDFS before (after) the job begins (finishes). The virtual cluster will be destroyed just after the final output files have been persistently stored on the storage cluster. To ensure that MapReduce in VMs can work in its original way, we have to build the traditional environment in a virtual cluster, including the computing and storage. But we cannot store the users' data in VMs permanently as the VMs are used by different users at different times. Thus, we have to store the data in a physical file system, inducing the serious problem that every time a request arrives the virtual cluster needs to load and run the HDFS and download and upload data between the HDFS in the VMs and the file system in the physical machines. These factors can lead to significant a cross-rack data movement overhead and prolong the job execution time [7]. Many optimizations concerning data transfer between physical storage and the virtual HDFS have been proposed [8] but removal of the virtual HDFS is unique and the first attempt in this field. We store the data of users on the physical machines which run HDFS and VMs. Furthermore, neither computation nor storage both run in VMs. We present a novel mechanism of decoupling the storage and computation in VMs, namely, just computation runs in the VMs and there is no longer any import and export. Fig. 2 shows the changes of the process as the framework is simplified.

Morpho decouples computation and storage layers but deploys them on the same physical cluster. This kind of overlap of computation and storage is similar to the traditional private Hadoop cluster deployment. Data blocks and tasks are distributed by central schedulers (e.g. JobTracker, NameNode, Mesos and Hadoop YARN resource manager) to all machines in the cluster. Thanks to the fine-grained task-level scheduling and delay scheduling, the central task scheduler can balance data locality and resource fair sharing among different jobs. This goal is hard to achieve in virtualized cloud platforms because the basic scheduling units are virtual machines which have much higher re-launch and migration cost than tasks, while the users SLA of their cloud services cannot be broken like soft fair sharing constraints in private Hadoop clusters. Decoupling of computation and storage brings a series of challenges along with the advantage of degradation of extra overheads. As Fig. 2 shows, after decoupling, the computation and storage layers become two relatively independent systems: virtual machines and physical machines. Because of this, the computing processes cannot fetch the block location information as before. On the other hand, since the data is stored on physical machines which run the VMs simultaneously, the VMs assignments can bring about great performance variations. At this point, the problem of data-location-aware VM placement should be taken into consideration. Moreover, for each physical machine, its load capacity is limited, which means the users whose data are stored on this machine will compete for the limited capacity (VM slots) of this machine. For example, if there are 10 concurrent running jobs on the platform, the location-aware VM scheduler may try to assign one VM for each job to node a for data locality, because it stores input data blocks of all 10 jobs locally. Suppose the PM a can only host up to 8 VMs under the soft constraint of overprovisioning or the hard constraint of non-overprovisioning (for simplicity we assume all the VMs are homogeneous), the VM scheduler have to re-assign 2 VMs to another idle PM that stores identical replicas of the corresponding targeted data blocks due to the competition of the 8 VM slots of PM a. It is important to mitigate these contentions to some degree if not all. By addressing these issues, Morpho makes a number of contributions:

(1) *Decoupling of computation and storage.* To our knowledge, Morpho is the first system to decouple the storage and computation in its true sense for a MapReduce service in a cloud. In another words, it is the first system where the map tasks in VMs can directly fetch the data they need from physical machines without any downloading, and vice versa. In comparison, recent commercial systems all use a mechanism where map tasks get data from a coupled HDFS in VMs, inducing extra uploading and downloading.
(2) *Data perception mechanism after decoupling.* Morpho introduces an efficient and low-cost mechanism which allows the

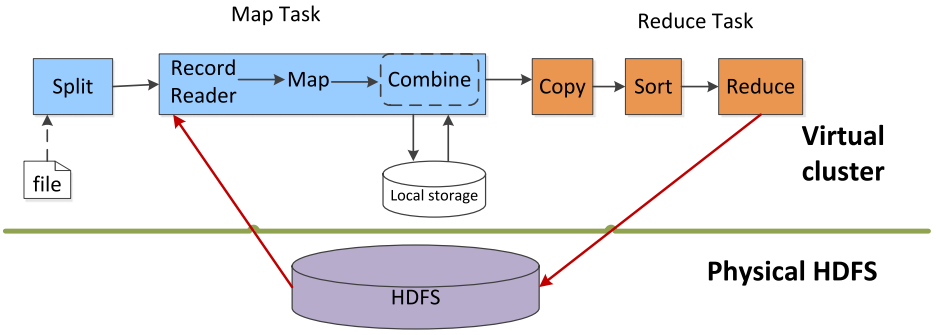


Fig. 2. The decoupled cloud computing mode.

independent computing framework running in VMs to intelligently choose the data blocks in physical machines to process, while the VMs are in the middle. "*Intelligently*" here means that a map task will choose the closest physical machine that stores the data blocks it needs—generally, the closest machine is its host machine.

(3) *VM placement.* As the data has been stored in physical machines, this strategy determines which physical machines run the VMs. The first principle is that of placing VMs on the physical machines that contain the input data chunks for the map task. But the first principle alone is not enough, there are always exceptions. Usually, a user will specify the number of VM instances and the hardware configuration of the virtual cluster applied for his/her MapReduce jobs according to the applications business requirement and this user's budget for the application. Intuitively, the cluster size will be proportional to the job input data size for the same application. But this assumption is often broken in reality, especially considering that it is difficult to achieve linear speedup when you scale out the virtual cluster due to the extra networking overhead. A user's VMs and the physical machines that contain his data do not always exactly match. How we handle these circumstances will be discussed in particular later.

(4) *Data placement.* This strategy aims at optimization of the placement of input data blocks. The data placement and the VM placement are complementary, the latter is specific to the reduce input phase while the former is specific to the map input phase. They complement each other to increase the locality of the entire process of MapReduce jobs and then minimize the total cost. Additionally, the data placement we talk about later is also helpful in reducing the contentions among jobs to some extent.

Through an extensive experimental evaluation, we demonstrate that Morpho significantly improves the system performance, including a 62% reduction in job execution time and reducing up to 70% of the cross-rack network traffic in some scenarios as compared with the Amazon cloud mode, which uses two separated clusters as the computing cluster and persistent storage cluster—the users' virtual clusters for MapReduce computing are hosted on the computing cluster, while the input (output) data have to be read from (written to) the storage cluster into (from) the local HDFS before (after) the job begins (finishes). On the other hand, we compare our quantitative and user-oriented data placement with the originally random placement of HDFS [9] with consideration of the contentions. It significantly reduces the probability of VM placement contentions of the data-location-aware VM scheduling without forcibly splitting users' data into fixed physical machine partitions.

Section 2 of this paper presents the problem analysis and algorithms of data and VM placement; Section 3 describes the design and implementation of the Morpho system; Section 4 covers performance measurements of our implementation from a variety of aspects. Related work is covered in Section 5, while Section 6 concludes.

2. Problem analysis and Morpho algorithms

Typically, cloud service providers use two distinct infrastructures for storage and computing (e.g. Amazon S3 for storage and Amazon EC2 for computing). Executing a MapReduce job in such infrastructures requires an additional loading step, in which data is loaded from the storage cloud into the HDFS of the MapReduce VMs running in the computing cloud. Such additional loading adversely impacts performance, as while the job is running there is a copy in the compute cloud for MapReduce processing along with the storage cloud original, leading to high costs for the provider.

In contrast, we propose a decoupled MapReduce in a cloud in which data is directly stored on the same physical machines that run MapReduce VMs. There are only computing processes in the MapReduce VMs and they directly access the HDFS on the physical machines without any explicit or implicit data-loading step. This prevents any waste of resources for data loading before executing a MapReduce job.

2.1. Problem analysis

There are several critical factors to large distributed systems, e.g. scalability, response time, execution time and throughput. We focus on the reduction of execution time and cost of the system. In this section, we will give a formal analysis concerning our data and VM placement strategy in the decoupled data processing context. We start with some notation.

A MapReduce job J is defined as a 3-tuple: (D, P, V) [10], where D is the set of input data chunks; $P = \{M_k : 1 \leq k \leq |P|\}$ is the set of physical machines that store dataset D; $V = \{V_k : 1 \leq k \leq |P|\}$ is the set of VMs which executing the map and reduce tasks of job J, and $|V|$ represents the number of VMs requested by the user. To simplify analysis of the model, we make the following assumptions: the dataset D consists uniform sized blocks $B_i : 1 \leq |D|$, where $|D|$ represents the number of blocks for D and each map has the same size of output. We denote the mean size of the expected map output of each block of D by $mapoutput(D)$. Let $dist(M_l, M_k)$ denote the network distance between the physical machines M_l and M_k and $velo(V_l, M_k)$ denote the data transfer rate between V_l and M_k.

2.1.1. Execution time

As we can see from Fig. 2, the execution time T of a job consists of many parts, but some of them are relatively stable among different executions under the same CPU and memory configurations of machines, such as the process time of map and reduce functions and result output. We just focus on the critical and

potentially optimizable parts: namely, the map input phase and reduce input phase. So we can simplify T to $T(J) = MT(J) + RT(J)$, where $MT(J)$ is the map phase execution time of job j, and $RT(J)$ is the reduce phase execution time of job j. The mean time of one map input loading can be calculated by

$$\overline{mt}(B) = \frac{\sum_{1\le i\le |D|} \frac{B_i}{velo(M(B_i),V(B_i))}}{|D|} \quad (1)$$

where $M(B_i)$ is the physical machine storing block B_i, and $V(B_i)$ is the virtual machine which is assigned the corresponding map task of block B_i. According to the assumption, all the map tasks have the same running time, in which case the map tasks complete in rounds. If there are at most m concurrent mappers and r concurrent reducers in the system, one job consists of $\lceil |D|/m \rceil$ rounds of map tasks; therefore $MT(J) = \lceil |D|/m \rceil \times \overline{mt}(B)$. Since each reduce task needs to read the input data from all map tasks and the reduce phase begins at the same time as the first map round ends, it means the time of the two phases overlap. So we just consider the time of data transfer from the last round map tasks to the reduce tasks. Let the number of last round map tasks $lrm = |D|\%m$,

$$\overline{rt} = \frac{\sum_{1\le j\le r} \frac{\sum_{1\le j\le r} \frac{mapoutput(D)}{|D|\times r\times velo(V(m_i),V(r_j))}}{lrm}}{r} \quad (2)$$

where $V(m_i)$ is the virtual machine executing the map i.

Ideally, in the formula for $\overline{mt}(B)$, $velo$ is approximately divided into three orders of magnitude when map tasks read blocks from different physical machines – PM-local, rack-local and none-local – whose values increase gradually. To minimize $\overline{mt}(B)$ we expect all the blocks can be read PM-locally. So by the data placement we try to control the number of physical machines storing the input data blocks of the job. Ideally, when we get $|P| = |V|$, we can achieve 100% PM-locality, which will be decreased by the competition among users in a real environment. Similarly, $velo$ in the formula for $\overline{rt}$ could be divided into inter-rack and outer-rack rate values. So a set of compactly placed VMs can take advantage of this well and get minimal $\overline{rt}$.

2.1.2. Data transfer cost

It is easy to find that the data transfer cost of map input $Mcost(J) = \sum_{1\le i\le |D|} B_i \times dist(M(B_i), V(B_i))$ and reduce input $Rcost(J) = \sum_{1\le i\le |D|} B(i,j) \times dist(V(B_i), V(r_j))$, where $B(i,j)$ is the processing output of block B_i which is transferred to reduce r_j. Our placement strategy can minimize both execution time and data transfer cost.

In our system, when users upload their data onto the platform, the data will be broken up into blocks corresponding to MapReduce splits and stored on a distributed file system deployed on physical machines. Our data placement strategy decides on which machine to store each block and the VM placement strategy attempts to place VMs on the physical machines that contain the input data blocks for the map phase while the former is for the reduce phase. Because data and VM co-placement involves a bin-packing problem, giving an optimal solution for this problem is NP-Hard [11].

2.2. Data placement

Before presenting the data placement strategy, we make some statements about usage of our system. In our system, when a user submits a job and input dataset, he also needs to provide the number of VMs $|V|$ and some extra parameters, such as the number of map and reduce tasks and configuration parameters of the VMs. A user may want to upload the data only and submit the corresponding job later. In this case we take an estimated value according to the size of his dataset instead of the submitted parameter $|V|$ when applying the data placement. Next, we will demonstrate our data placement in the first case.

The goal of the data placement is minimizing the transfer cost of reduce input. As each reduce task needs to read the output of all map tasks, a sudden explosion of network traffic can degrade the system performance significantly, with the data randomly spreading across the entire system. So we expect data stored compactly, thus the VMs can be placed compactly by applying the VM placement described in the next section, therefore reducing the network traffic. Furthermore, we also take the storage utilization of racks into our consideration. The data placement algorithm in Morpho is shown in Algorithm 1.

Algorithm 1 Data placement algorithm

Require: $F = \{b_1, \ldots, b_n\}$ the set of data blocks of the input file f
Ensure: assign all replicas to machines for data blocks in F

```
find the rack rack_min with the maximum available storage capacity
T ← φ
add P machines randomly chosen from rack_min into the target machine set T, |P| ≈ |V|
for all block b_i ∈ F do
    for all replica r_j(b_i) of block b_i, j ≦ f_replicas do
        randomly choose target machine m_target from T
        assign replica r_j(b_i) to machine m_target
    end for
end for
```

In Algorithm 1, for every uploading, we check the residual storage capacity of each rack and select the maximum one. Then we distribute the data blocks on $|P|$ physical machines, where $|P| \approx |V|$; in the scope of the rack we just select to ensure that the physical machines containing the dataset are close to each other. We prefer $|P| \approx |V|$ to $|P| = |V|$ because, as shown in Fig. 3, dataset A is placed on physical machines in rack 1 and B is placed in rack 2. When the dataset C arrives and three machines are requested, it will be placed on two physical machines in rack 1 rather than three machines across two racks, for which a few data transfers of map input can avoid remarkable cross-rack network traffic.

On the other hand, the principles make the distribution of all the data relatively average and can weaken competition among users to some extent. For the most part, we just need to select one rack, but sometimes when the residual storage capacity is not enough, we should select the two or three largest ones. The number (of racks we select) should not be larger, as it indicates that the system is loaded to full capacity.

Additionally, we set two block replicas, where the first one is on the same rack as the primary replica and the second one is on another rack. So we can use the former as a candidate in case of contention and the latter for fault tolerance and failure recovery.

2.3. VM placement

Once data blocks are placed in a set of closely connected nodes, VM placement ensures that VMs get placed on the physical machines compactly either containing the input data or the close-by ones. As each reduce task needs to read the output of all map tasks [1], a sudden explosion of network traffic can significantly degrade cloud performance. This is especially true when data has to traverse a greater number of network hops while going across racks of servers in the data center [12]. For this placement, mappers of the jobs can read data locally, while reducers closer to each other

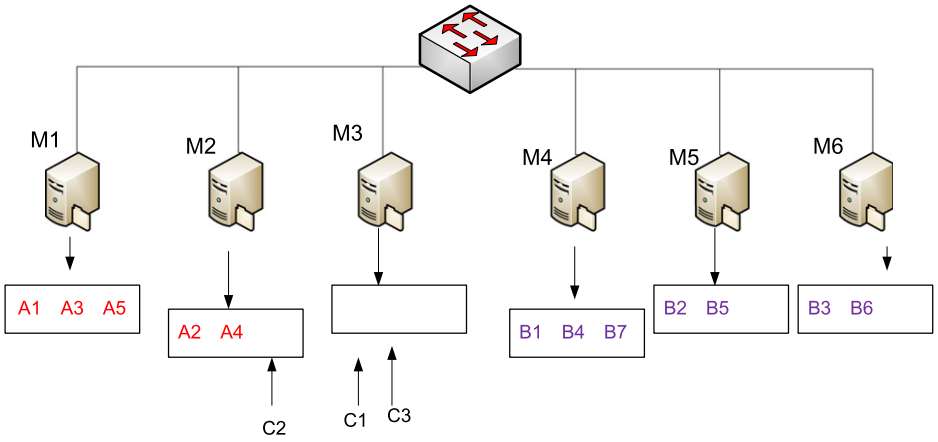


Fig. 3. Example of data placement.

Algorithm 2 Data placement algorithm

Require: $V = \{v_1, ..., v_n\}$ the set of virtual machines to be scheduled
Require: $P = \{p_1, ..., p_n\}$ the set of physical machines that stores data blocks to be processed
Ensure: assign all virtual machines to physical machines
1: **for all** virtual machine $v_i \in V$ **do**
2: **if** physical machine p_i can host v_i, $\exists p_i \in P$ **then**
3: randomly choose the target machine m_{target} that can host v_i from P
4: **else**
5: randomly choose the target machine m_{target} that can host v_i close-by P
6: **end if**
7: assign virtual machines v_i to physical machine m_{target}
8: **end for**

have a short network distance. This can significantly speed up job execution and reduce cumulative data center network traffic. The VM placement algorithm in Morpho is shown in Algorithm 2.

$|V| = |P|$. This is a perfect condition. We just need to place VMs on the physical machines that contain the input blocks, and this is the basic principle.

$|V| < |P|$. In this situation, we prefer migrating VMs to transferring data apart from the basic principle. When data processing is finished we shut down the VM on that physical machine and start a VM on another physical machine while maintaining the total number of VMs.

$|V| > |P|$. We place VMs on physical machines either storing the input data or the close-by machines.

All of the three circumstances above may happen such that some machine containing the input data blocks does not have sufficient capacity. For such a case, the VM may be placed on the node that stores the first replica on the same rack. If the required resources of the machine are not available, the VM will be placed on a node close-by which has enough capacity.

3. System design and implementation

Our Morpho system is developed, based and deployed on a private cloud platform prototype in our lab, which is built on a customized version of OpenNebula. Fig. 4 shows the general architecture of Morpho. The Morpho client and server are independent components written in Java, and the data perception modular is implemented by modifying Hadoop *Jobtracker* and *NameNode* plus a mapping plugin with the *DNSToSwitchMapping* interface. We implement a virtual machine scheduling framework with two pluggable algorithms (round-robin and data-location-aware) within the Morpho server. The entire VM placement information is also maintained in the Morpho server. We just use *deploy* and *cancel* functions of OpenNebula to startup and shutdown virtual machine instances on specified physical machines. Therefore, OpenNebula can be replaced with any other virtualized cloud software that has similar VM *deploy* and *cancel* functions.

3.1. Morpho client

In our system, users can use the client-side tool to upload thedata that will be processed and the job-specific Hadoop MapReduce jar files or other types of compiled executable files written in different languages. This step is similar to any typical cloud service which requires setting up of the application stack and data. Then the client-side submits the jar file and some parameters to the server-side through calling XML-RPC, it will return a web or ssh address for users to view the status of their jobs at any time. The command line interface (CLI) is also provided for the system administrator to configure and manage the system, such as the number of vCPUs of each VM, memory size, changing the number of slave nodes and so on.

3.2. Morpho server

The Morpho server consists of several core components: Job Queue, VM Scheduler, Job Runner, a three-layer mapping module, OpenNebula front-end and the NameNode daemon. Just like Fig. 5 shows, first the submitted job goes into a queue, then when the required system resources are available the VM Scheduler creates VMs according to the VM placement, which we talk about later, and configures the virtual cluster. Then the Job Runner deploys Hadoop on all nodes of the virtual cluster and runs Hadoop MapReduce instances without HDFS. When the job finishes, Job Runner informs VM Scheduler to destroy the virtual cluster in order that the resources can be allocated to other users in a timely fashion. More details of the operational process will be described in the next part.

3.3. Data perception after decoupling

In this section, we will explain how the map tasks in VMs can intelligently get the data they will process on physical machines.

In contrast to the traditional MapReduce architecture, which only has a two-layer mapping relation, known as rack–PM(physical machine), the MapReduce architecture in the cloud changes a lot as

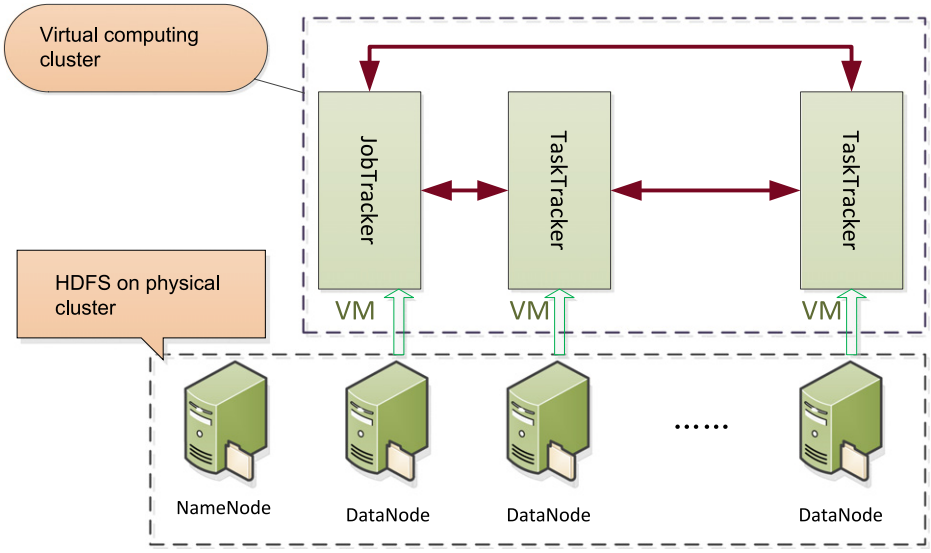


Fig. 4. General architecture of MORPHO.

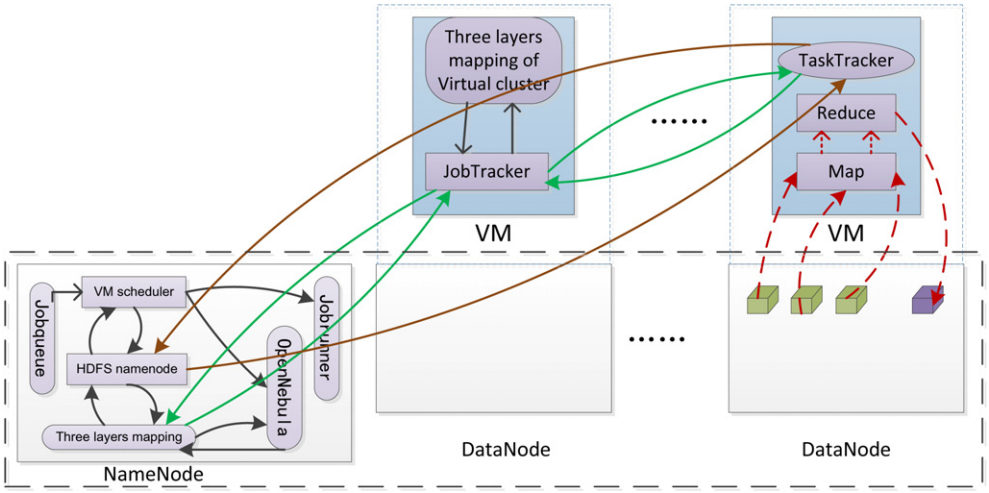


Fig. 5. The concrete procedure of decoupled computing.

the VM is used. First of all, we build a three-layer mapping module which stores the relation of rack–PM–VM of the system for the specific purpose of searching. This mapping relationship changes dynamically as virtual clusters are created and destroyed, while the mapping of rack–PM is relatively constant unless the administrator refreshes it. So we let this module store the mapping relation of rack–PM in advance and then access the OpenNebula front-end repeatedly every several seconds to get the mapping relation of PM–VM and finally get the whole latest mapping relationship of the system. Furthermore, the relationship is stored in the structure of a tree, similar to that shown in Fig. 6.

When a virtual cluster is created, every TaskTracker reports the name of its host VM to the JobTracker while sending a heartbeat according to the IP information of JobTracker. Then the JobTracker accesses the mapping module to search the three-layer mapping tree for the names just reported and gets the PM and rack information of every VM in that virtual cluster. For example, the JobTracker searches the mapping module for Vm1 and gets its host PM, named M2 and rack1, so it gets the topology information /rack1/PM2/Vm1 of Vm1. Actually, in this course the JobTracker also builds a three-layer tree, but the difference is that this is only a part of the mapping tree in the mapping module. Moreover, when the JobTracker receives the job submitted, it accesses the NameNode to figure out where the data blocks to be processed are located and creates the tasks. Ultimately, when a TaskTracker requests a task, the JobTracker can intelligently assign a task to it based on the mapping relation of the virtual cluster it caches.

This paragraph discusses how the TaskTracker gets the data it will process. When a TaskTracker is allocated a task, it will ask the NameNode where it can get the closest data. For simplicity, we do not change the topology tree the NameNode original stores, which only includes rack–PM mapping relationship. We take the method of inserting a node. The TaskTracker sends a request as well as the name of its host VM to the NameNode, we just let NameNode search the mapping module to get the information of its host PM and then temporarily insert the VM node into the corresponding PM node of the tree it stores. Thus, the NameNode can search its own tree to find the closest PM node and return its information to the TaskTracker, then the TaskTracker can directly send the request to get data from that PM node. After processing, the data can upload into physical machines directly while applying the data placement strategy we discuss subsequently.

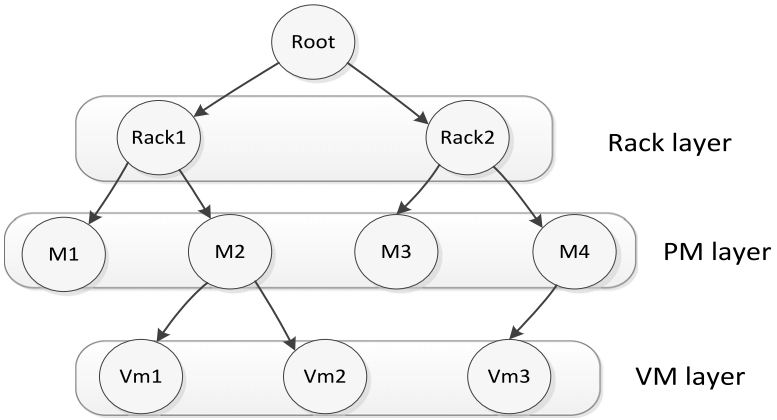


Fig. 6. The three-layer mapping relationship.

4. Evaluation

4.1. Performance evaluation

We perform all our experiments in a 22-node testbed from a 322-node educational HPC cloud in our school which has two clusters: a 17-node compute cluster (cluster A, one master and 16 workers) and a 5-node external storage cluster (cluster B, one master and four workers). Each node in cluster A is equipped with two 6-core 2.67 GHz Intel Xeno CPU, 16 GB RAM and one 1 TB SAS disk, while two 4-core 2.4 GHz Intel Xeno CPU, 24 GB RAM and one 1 TB SATA disk in cluster B respectively. All the nodes are interconnected by a gigabit Ethernet network, and the rack-level switches are linked by a backbone switch. Cluster A is managed by a IaaS system called Crane, which was developed in our lab. The core component of Crane is a modified version of OpenNebula with customized virtual machine scheduling policies. To bypass Crane's scheduling policies, Morpho communicates directly with the OpenNebula master through XML-RPC to launch and destroy virtual machines. We compare Morpho with the native Hadoop 1.0.0 using the Amazon Elastic MapReduce workflow. The Morpho Map/Reduce tasks running in the virtual machines read/write directly from/to the internal HDFS deployed on the cluster A; while the Amazon Elastic MapReduce workflow first loads the input files from the external HDFS deployed on cluster B, and then transfers the output files in the temporary HDFS launched inside the virtual cluster into the external HDFS after the job finishes. In the Amazon mode, all the virtual machines will be placed on physical machines by the VM scheduler in a round-robin manner. Each worker node of cluster A could allocate at most 8 CPU cores and 8 GB memory to the virtual machines, and all the masters of virtual clusters will be placed on the master node of cluster A. We use two VM types: each type-a VM is allocated one VCPU core and 1 GB memory and configured with 1 mapper slot and 1 reducer slot; the type-b VM has two VCPU cores and 2 GB memory with 2 map/reduce slots. We pick three built-in example applications from Hadoop: *Grep* (searching for a single word), *WordCount* and *Sort*, which generate 10–30 bytes, less-than-input and equal-to-input sized intermediate/output data respectively. The input files of *Grep*, *WordCount* are 48 GB randomly generated words written by the built-in *RandomTextWriter*, and the input files of *Sort* are 48 GB randomly generated integers written by *RandomWriter*. To validate the data-aware VM scheduling, all the input data blocks in the internal HDFS are distributed on only 8 nodes of cluster A.

We find in Fig. 7 that Morpho has up to 150% faster total execution time when compared to native Hadoop. The speedup increases along with the increase of the number of the virtual machines because the additional load/store transfer overhead is almost constantly high, even ramping up the capacity of the virtual cluster, due to the network topology of our testbed. The pure job execution times of Morpho are slightly longer than the execution times of the Amazon mode jobs, since the coupled MapReduce framework and HDFS deployed inside the virtual cluster can fully utilize the data locality of map tasks, while Morpho jobs cannot achieve the same level of data locality under our setup even using the data-aware VM placement. We also notice that the per-VM performance decreases along with the increase of the number of co-located VMs on a same PM and the intermediate/output data size, because the co-located VMs compete for the I/O bandwidth of their host's single disk. In the simulation evaluation, we assume all virtual machines have the same disk I/O bandwidth irrespective of the co-located VM number.

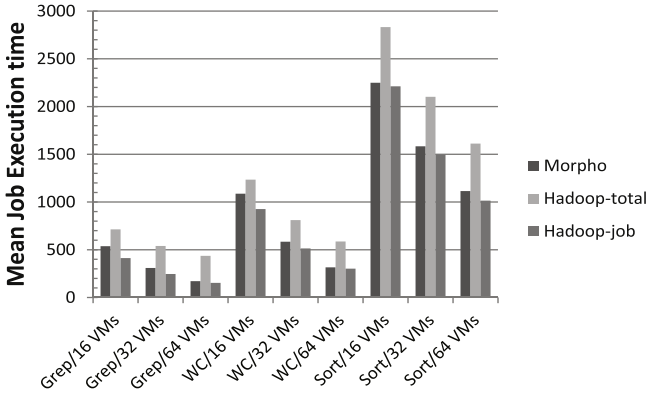


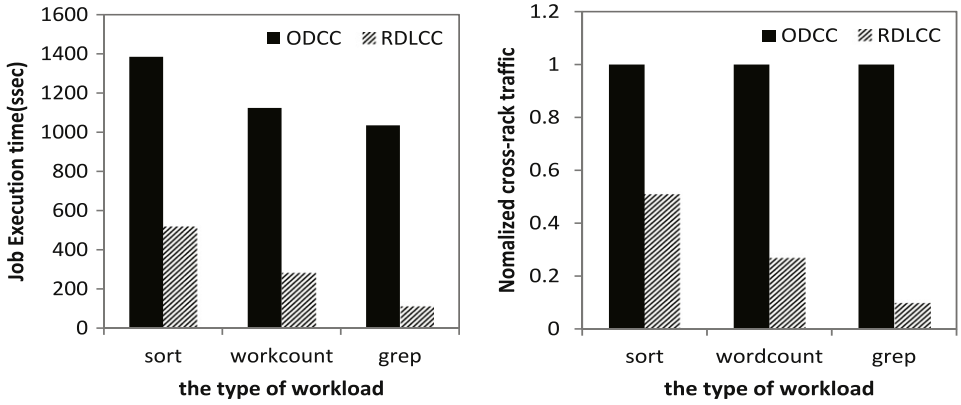
Fig. 7. Performance evaluation.

4.2. Simulation evaluation

4.2.1. Simulation setup

Metrics: We evaluate our strategy using two metrics: (1) job execution time: techniques that allow jobs to read data directly and locally result in faster execution; (2) Cross-rack data transfer amount: techniques that read a lot of data across racks result in poorer throughput.

Simulation setup: Since MapReduce is widely used in modern production data centers for large-scale data intensive jobs, we choose several typical MapReduce jobs in Table 1 as the workloads and conducted measurements in the Hadoop 1.0.0 platform by a implemented simulator, similar to the existing NS-2 [13] based



(a) Job execution time.

(b) Cross-rack traffic.

Fig. 8. The technique of data perception.

Table 1
Different types of jobs.

Job	Sort	Grep	WordCount
Input data (GB)	40	80	80

MRPerf simulator [14]. The simulation framework simulates a data center of 400 nodes with a 1 Gbps network configured in the typical tree topology. The 400 nodes are uniformly organized in 10 racks. In the experiments, each server has two 4-core CPUs and 32 GB memory, running KVM-based virtual machines and is connected to a 1 Gbps switch. By default, each VM is allocated a core and 4 GB memory and has 2 mapper slots and 2 reducer slots. Because each physical server reserves 2 CPU cores for the VM hypervisor and the HDFS DataNode, it can host up to 6 VMs, which is 240 VMs per rack with non-overprovisioning. All the jobs are generated by GridMix (a Hadoop built-in workload generator) with default parameters. Each job applies 20–50 VMs, the actual number being randomly chosen and independent of the job size.

To bootstrap the simulator, we used measurements obtained from the performance experiments to configure the simulator parameters, e.g. map and reduce execution time and data transfer rate under different locality levels.

Simulation scenarios:

- Decoupled but Locality-aware Cloud Computing (DLCC): this is what we present in this paper.
- Traditional Cloud Computing (TCC): this model is widely adopted in commercial clouds today, as Fig. 1 shows in the introduction section. It has frequent data transfer between the HDFS in the virtual cluster and the physical device.
- Only Decoupled Cloud Computing (ODCC): in this system we just apply our decoupled strategy without the data perception or data placement.
- DLCC with random data placement (RDLCC): RDLCC uses random data placement in the traditional HDFS instead of the data placement we propose.

4.2.2. Results analysis

The technique of data perception. We conduct this experiment in ODCC and RDLCC. In Fig. 8, we compare the job execution time and cross-rack traffic of the two schemes for the three workloads in Table 1.

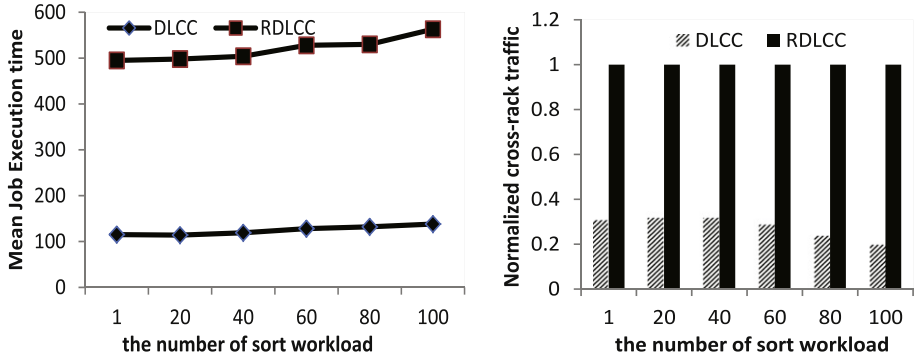
To reduce the overhead of transferring data between the HDFS in the virtual cluster and the physical device we decouple the storage and computation, but after decoupling the two modules become relatively independent. That is why ODCC has a poor performance. In RDLC, we apply our data perception technique to address the problem of independency, which brings intelligent data reading because of the fundamental VM placement. So RDLC gives a 30% reduction in job execution times and reduces up to 40% of cross-rack network traffic.

The technique of data placement. In this experiment, we compare two data placement schemes: our proposed quantitative and compact data placement in DLCC and the random data placement in RDLCC. In contrast, the random data placement scheme places data blocks in a set of randomly chosen physical machines that have available storage capacity without any provisions about the VM placement. We also study the impact of varying the number of jobs in Fig. 9 using the same workload *Sort*. In Fig. 9(a), the number of jobs is varied from 1 to 50 and the average job execution time and cross-rack traffic are compared.

Firstly we compare the two data placements. For RDLCC the data blocks have been distributed all over the network randomly, so the VMs are spread across the system and hence the reduce input phase obtains poor locality, leading to longer execution times in Fig. 9(a). The normalized cross-rack traffic in Fig. 9(b) is also indicative of the same trend. On the other hand, in DLCC the mean job execution time keeps stable with an initial increase jobs. When the number of jobs is up to 30 it shows a tiny increase, which means some contentions occur among jobs. Importantly, even the maximum number of jobs brings only a 5% increase of mean execution time, which is totally within the acceptable limits.

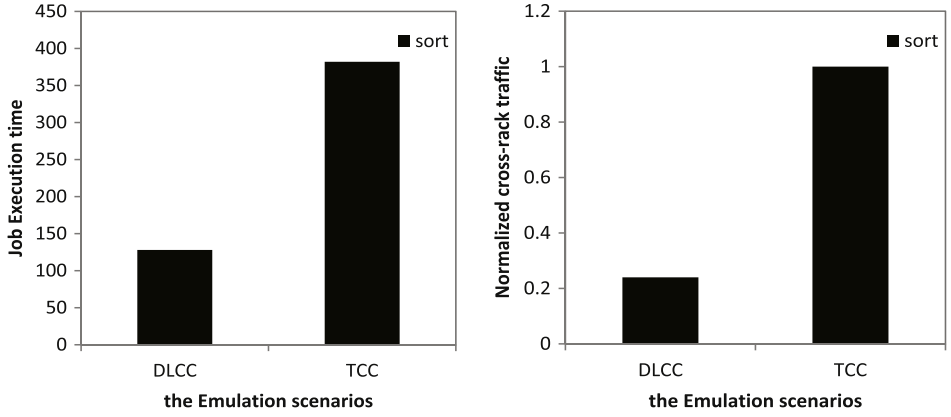
The Comparison of DLCC and TCC. We conduct this experiment in both one-job and multi-job environments, because a one-job test can demonstrate the ideal difference of performance of our system and a multi-job test can show how the system performs while contentions occur. For the multi-job test we use a mixed workload of jobs, consisting of equal proportions of all workloads in Table 1, and the arrival rate of the jobs on the datasets is uniformly distributed from 200 to 2000 s. We use a 80 GB dataset for both the *WordCount* and *Grep* workloads and a 40 GB dataset for *Sort* workload. A total of 60 datasets were used, 20 for each of the workload types.

We find in Fig. 10(a) that DCLL has up to 62% faster execution time when compared to TCC. TCC performs poorly since it has an extra data transfer, as Fig. 1 shows, and it uses random data placement. Our quantitative and compact data placement with locality awareness in VM placement ensures that the mappers are placed on the physical machines containing the input data and the reducers are packed close to each other so that reduce traffic does not traverse a long distance on the network. So Fig. 10(b) shows the cross-rack traffic in DLCC is only 30% of that in TCC.



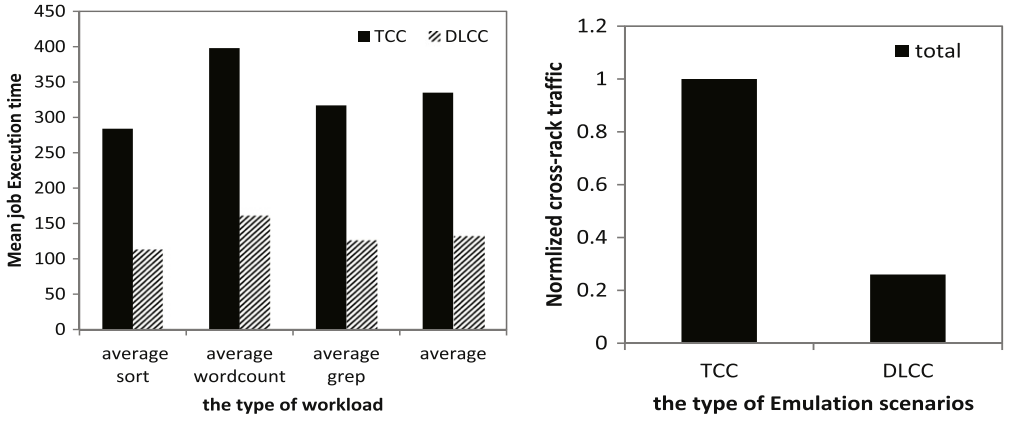
(a) Job execution time. (b) Cross-rack traffic.

Fig. 9. The technique of data placement.



(a) One-job execution time. (b) One-job cross-rack traffic.

Fig. 10. The comparison of DLCC and TCC (one-job).



(a) Job execution time. (b) Cross-rack traffic.

Fig. 11. The comparison of DLCC and TCC (multi-job).

Though we do not explicitly design an exclusive mechanism or module to address the resource contentions among jobs, every step and principle we take is a precaution against them. From the multi-job measurement in Fig. 11 we can tell it does well. When the total number of jobs reaches 60, the mean job execution time becomes longer, but still 58% faster compared to TCC. We find an 8% increase in execution time by comparing the one-job and the multi-job experiments, while the result in TCC turns out to 13%. Importantly, the cross-rack traffic increases by 10 times as the jobs increase by 60 times. In contrast, this number in TCC is 40 times. This accounts for our system being more efficient and energy-saving.

5. Related work

To the best of our knowledge, Morpho, with its decoupled MapReduce mechanism and complementary data and VM placement, is unique in exploiting MapReduce in cloud computing. We briefly review some of the current situation in this area. Zaharia et al. [15] have proposed a new speculative task scheduling algorithm called Longest Approximate Time to End (LATE) to improve the performance of Hadoop in a heterogeneous (especially virtualized) environment. Sangwon et al. [16] have proposed pre-fetching and pre-shuffling schemes for a shared MapReduce

computation environment. While the pre-fetching scheme works by assigning the map tasks to the nearest node to the blocks and then pre-fetching the blocks considering a list generated by a prediction model, the pre-shuffling scheme works by looking over the input split before the map phase begins and predicting the target reducer where the intermediate output is partitioned into local nodes. Starfish [17] improves MapReduce performance by automatically tuning Hadoop configuration parameters. Sandholm et al. [18] present a system for allocating resources in shared data that uses regulated and user-assigned priorities to offer different service levels to jobs and users over time. Zaharia et al. [19] have proposed a simple scheduling algorithm called delay scheduling to achieve locality and fairness in multi-user MapReduce job scheduling. BitDew-MapReduce [20] adopted optimizations include aggressive task backup, intermediate result backup, task re-execution mitigation and network failure hiding for the Internet Desktop Grid.

Much work has explored the placement of applications (VMs) in a virtualized data center to reduce resource consumption and achieve high performance. Tashi [21] proposes that location awareness is important, without providing a complete solution. Quincy [22] is a resource allocation system for scheduling concurrent jobs on clusters while considering input data locality. In recent work, Mantri [1] identifies that cross-rack traffic during the reduce phase is a crucial factor for MapReduce performance. However, it has limited performance improvements during task placement without a locality-aware data placement. Purlieus [8] is different from those, as it considers both input and intermediate data locality for MapReduce. However, it has some assumptions that we think are not feasible, such as the expected load on each dataset is known and dividing all the jobs into three distinct classes according to the amount of data accessed in the map and reduce phases. So we learn from it and improve it without any unreasonable assumptions, as discussed in Section 4, for our decoupled MapReduce framework to achieve high performance. Moreover, we agree that without an efficient underlying data placement, even a sophisticated locality-aware compute placement may not be able to achieve the high data locality presented in Purlieus. Last, we think our decoupled framework without data transfer is much better than its seamless data-transfer manner with techniques of loopback mounts and VM disk-attach [8].

6. Conclusion

This paper presents Morpho, a decoupled but locality-aware MapReduce framework for cloud computing. We describe how existing cloud MapReduce services lead to longer job execution times and large amounts of cross-rack network traffic in the data center. To address the problems of frequently loading and running HDFS in virtual clusters and downloading and uploading data between virtual clusters and physical machines, Morpho uniquely proposes a decoupled MapReduce mechanism that decouples the HDFS from computation in a virtual cluster and loads it onto physical machines permanently. Moreover, we apply complementary data and VM placement in this decoupled system to give relative independence after decoupling and achieve high performance at the same time. Our detailed evaluation shows significant performance gains, with a close to 60% reduction in execution time, an up to 70% reduction in the cross-rack network traffic, and an acceptable increase when contentions occur.

Acknowledgments

This paper is partly supported by the NSFC under grant No. 61370104 and No. 61133008, National Science and Technology Pillar Program of China under grant No. 2012BAH14F02, MOE-Intel Special Research Fund of Information Technology under grant MOE-INTEL-2012-01, and Chinese Universities Scientific Fund under grant No. 2012TS046.

References

[1] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Proceedings of the 6th Symposium on Operating System Design and Implementation, OSDI 2004, San Francisco, California, USA, 2004.
[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the Clouds: a Berkeley View of Cloud Computing, Tech. Rep., University of California at Berkeley, 2009.
[3] Amazon Elastic MapReduce. URL http://aws.amazon.com/elasticmapreduce/.
[4] Amazon Elastic Compute Cloud. URL http://aws.amazon.com/ec2/.
[5] Amazon Simple Storage Service. URL http://aws.amazon.com/s3/.
[6] Amazon Elastic MapReduce Developer Guide. URL http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/.
[7] D. Shue, M.J. Freedman, A. Shaikh, Performance isolation and fairness for multi-tenant cloud storage, in: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, 2012.
[8] B. Palanisamy, A. Singh, L. Liu, B. Jain, Purlieus: locality-aware resource allocation for MapReduce in a cloud, in: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, 2011.
[9] Scheduling in Hadoop. URL http://www.cloudera.com/blog/tag/scheduling/.
[10] X. Yang, J. Sun, Reliable estimation of execution time of MapReduce program, China Commun. 8 (6) (2011) 11.
[11] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.
[12] G. Ananthanarayanan, S. Kandula, A.G. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in Map-Reduce clusters using Mantri, in: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, Vancouver, BC, Canada, 2010.
[13] Network Simulator (NS2). URL http://www.isi.edu/nsnam/ns/.
[14] G. Wang, A.R. Butt, P. Pandey, K. Gupta, A simulation approach to evaluating design decisions in MapReduce setups, in: Proceedings of the 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2009, South Kensington Campus, Imperial College London, 2009.
[15] M. Zaharia, A. Konwinski, A.D. Joseph, R.H. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, in: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, San Diego, California, USA, 2008.
[16] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, S. Maeng, HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment, in: Proceedings of the 2009 IEEE International Conference on Cluster Computing, CLUSTER 2009, New Orleans, Louisiana, USA, 2009.
[17] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F.B. Cetin, S. Babu, Starfish: a self-tuning system for big data analytics, in: Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, 2011.
[18] T. Sandholm, K. Lai, MapReduce optimization using regulated dynamic prioritization, in: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2009, Seattle, WA, USA, 2009.
[19] M. Zaharia, D. Borthakur, J.S. Sarma, K. Elmeleegy, S. Shenker, 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 2010, Paris, France, 2010.
[20] L. Lu, H. Jin, X. Shi, G. Fedak, Assessing MapReduce for Internet computing: a comparison of Hadoop and BitDew-MapReduce, in: Proceedings of the 13th ACM/IEEE International Conference on Grid Computing, GRID 2012, Beijing, China, 2012.
[21] M.A. Kozuch, M.P. Ryan, R. Gass, S.W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. López, M. Stroucken, G.R. Ganger, Tashi: location-aware cluster management, in: Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds, ACDC'09, ACM, New York, NY, USA, 2009, pp. 43–48.
[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: fair scheduling for distributed computing clusters, in: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, 2009.
[23] Windows Azure HDInsight. URL https://www.hadooponazure.com.
[24] Hadoop. URL http://hadoop.apache.org/.
[25] Microsoft HDInsight Tutorial. URL http://www.windowsazure.com/en-us/manage/services/hdinsight/.
[26] Hadoop DFS User Guide. URL http://hadoop.apache.org/.

Lu Lu received his Bachelor degree in Computer Science and Technology from the Naval University of Engineering (China) in 2008. He is now a Ph.D. candidate student in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL) at Huazhong University of Science and Technology (China). His research interests focus on large scale distributed data processing. Contact him at llu@hust.edu.cn.

Xuanhua Shi received his Ph.D. degree in Computer Engineering from Huazhong University of Science and Technology (China) in 2005. From 2006, he worked as an INRIA Post-Doc in PARIS team at Rennes for one year. Currently he is an associate professor in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) at Huazhong University of Science and Technology (China). His research interests include cluster and grid computing, fault-tolerance, web services, network security and grid security, virtualization technology, data-intensive computing. He is a member of the IEEE. Contact him at xhshi@hust.edu.cn.

Hai Jin is a Cheung Kung Scholars Chair Professor of Computer Science and Engineering at the Huazhong University of Science and Technology (HUST) in China. He is now the Dean of School of Computer Science and Technology at HUST. He received his Ph.D. in Computer Engineering from HUST in 1994. In 1996, he was awarded German Academic Exchange Service (DAAD) fellowship for visiting the Technical University of Chemnitz in Germany. He worked for the University of Hong Kong between 1998 and 2000 and participated in the HKU Cluster project. He worked as a visiting scholar at the University of Southern California between 1999 and 2000. He is the chief scientist of the largest grid computing project, ChinaGrid, in China, and he is the director of Key Lab of Service Computing Technology and System, MOE (Ministry of Education). Also, he is the chief scientist of National 973 Basic Research Program, Basic Theory and Methodology of Virtualization Technology for Computing System. Jin is a senior member of IEEE and a member of ACM. Jin is the member of Grid Forum Steering Group (GFSG). He has co-authored 15 books and published over 400 research papers. His research interests include computer architecture, cluster computing and grid computing, peer-to-peer computing, network storage, network security, and virtualization technology. Contact him at hjin@hust.edu.cn.

Qiuyue Wang is a masters student in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) at Huazhong University of Science and Technology (China). She is now doing some research on cluster and virtualization technology. Contact her at qywang@hust.edu.cn.

Daxing Yuan is a masters student in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) at Huazhong University of Science and Technology (China). He is now doing some research on cluster and virtualization technology. Contact him at dxyuan@hust.edu.cn.

Song Wu is a Professor of Computer Science and Engineering at the Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. from HUST in 2003. He is now the vice head of Computer Engineering Department at HUST. He is also served as the vice director of Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) of HUST now. He has worked for ChinaGrid project and ChinaGrid Support Platform (CGSP) for almost four years. In China Nation Grid (CNGrid) project, he is responsible for the construction of the Grid Center in Central China. In 2007, he was awarded New Century Excellent Talents in University (NCET). He has published more than sixty papers and obtained two patents (more than ten patent applications in the process) and almost ten software copyrights. Now he is in charge of a National High-tech project (863) and a NSF project on Grid Computing. He is also involved in a National Fundamental Research project (973) about virtual machine and interested in virtualization technology. Contact him at wusong@hust.edu.cn.