Resource Management for Dynamic MapReduce Clusters in Multicluster Systems

Bogdan Ghiţ, Nezih Yigitbasi, Dick Epema
Delft University of Technology, the Netherlands
{*b.i.ghit, m.n.yigitbasi, d.h.j.epema*}*@tudelft.nl*

***Abstract*—State-of-the-art MapReduce frameworks such as Hadoop can easily scale up to thousands of machines and to large numbers of users. Nevertheless, some users may require isolated environments to develop their applications and to process their data, which calls for multiple deployments of MR clusters within the same physical infrastructure. In this paper, we design and implement a resource management system to facilitate the on-demand isolated deployment of MapReduce clusters in multicluster systems. Deploying multiple MapReduce clusters enables four types of isolation, with respect to performance, to data management, to fault tolerance, and to versioning. To efficiently manage the underlying physical resources, we propose three provisioning policies for dynamically resizing MapReduce clusters, and we evaluate the performance of our system through experiments on a real multicluster.**

***Keywords*-MapReduce isolation, dynamic resource management, performance evaluation, multicluster systems.**

I. Introduction

With the considerable growth of data-intensive applications, the MapReduce programming model has become an exponent for large-scale many-task computing applications [1], as it not only eases the management of *big data*, but also simplifies the programming complexity of such applications on large cluster systems. Despite the high scalability of MapReduce frameworks, isolating MapReduce workloads and their data is very attractive for many users. In this paper, we design support for deploying multiple MapReduce clusters (MR clusters) within multicluster environments through extensions to our KOALA grid scheduler [2]. Furthermore, we develop a dynamic resizing mechanism for MR clusters, and three resource provisioning policies. We evaluate the performance of the system through experiments conducted on a multicluster system (DAS-4 [1]) managed by KOALA.

Having multiple MR clusters within a single multicluster system brings four advantages over a single MR cluster deployment, which are related to performance, data management, fault tolerance, and versioning. First, we can achieve *performance isolation* between streams of jobs with different characteristics, for instance, by having separate MR clusters for large and small jobs, or for production and experimental jobs. Secondly, different (groups of) users each working with their own data set may prefer having their own MR clusters to avoid interference, or for privacy and security reasons. We call this type of isolation with different data sets in different MR clusters *data isolation*. A third type of isolation is *failure isolation*, which hides the failures of one MR cluster from the users of the other MR clusters. Fourth, with the multi-MR clusters approach, *version isolation* may be enforced as well, such that different versions of the MapReduce framework can run simultaneously.

Making efficient use of the resources is mandatory in a multicluster environment. Therefore, to improve resource utilization, we provide the MR clusters with a grow/shrink mechanism. The problem of dynamically resizing MR clusters brings two challenges. First, we need to determine under which conditions the size of an MR cluster should be modified. When the data set exceeds the storage capacity available on the nodes of the MR cluster, or the workloads are too heavy, grow decisions should be taken. On the other hand, in case of an underutilized MR cluster, some of its nodes may be released. Secondly, we need to address the issue of rebalancing the data when resizing a cluster. When resizing, we distinguish *core* nodes and *transient* nodes. Both types of nodes are used to execute tasks, but only the core nodes locally store data. Using transient nodes to provision an MR cluster has the advantage of not having to change the distribution of the data when they are released. On the down side of this approach, the data locality principle is broken, as all the tasks executed on transient nodes need to access non-local data.

The contributions of this paper are as follows:

- The architectural design of KOALA that provides isolation between the deployments of multiple MR clusters within a multicluster system.
- The dynamic resizing mechanism for MR clusters with three distinct provisioning policies that avoids high reconfiguration costs and handles the data distribution in a reliable fashion.
- An evaluation of the performance of KOALA with MapReduce support on a real infrastructure (DAS-4).

[1]www.cs.vu.nl/das4/

II. Background

This section presents the main technologies on which our work is based on, the MapReduce model with its open source implementation Hadoop, and our grid scheduler KOALA.

A. Hadoop

The popularity of MapReduce has increased exponentially since its birth in 2004 due to its simplicity, its large applicability to an extensive variety of problems, and its ability to scale to large clusters of thousands of machines.

Hadoop [3] is a MapReduce implementation based on the Google model first introduced in [4]. Hadoop provides an execution framework that splits the job into multiple tasks and the Hadoop Distributed File System (HDFS) that ensures efficient and reliable storage for large volumes of data. Hadoop execution is divided into two main phases. During the map phase, each map task processes an input block of fixed size (e.g., 64 MB or 128 MB), and produces a set of intermediate pairs. Afterwards, in the reduce phase, all intermediate values for a given key are combined, producing in this manner a set of merged output values.

A central entity (the JobTracker) running on a master node is responsible for managing a number of workers (the TaskTrackers). Each TaskTracker can be configured with several map and reduce slots such that multiple tasks may be executed in parallel on the same node. The JobTracker assigns tasks to the workers in FIFO order, while the TaskTrackers report their state via a heartbeat mechanism.

The HDFS is designed in the same master-worker fashion: the NameNode runs on the master node and manages the system metadata, while the DataNodes run on the workers and store the actual data. For the I/O operations, the NameNode provides the locations of the data block to be read or written, and the actual transfer occurs directly between the HDFS client and the DataNodes.

B. Koala Grid Scheduler

KOALA is a grid resource manager developed for multicluster systems such as the DAS-4 with the goal of designing, implementing, and analyzing scheduling strategies for various application types. The scheduler represents the core of the system and is responsible for scheduling jobs submitted by users by placing and executing them on suitable cluster sites according to its scheduling policies. Jobs are submitted to KOALA through specialized runners for certain application types (e.g., cycle scavenging jobs [5], workflows [6], and malleable applications [7]). To monitor the status of the resources, KOALA uses a processor and a network information service.

To develop a MapReduce runner for KOALA we took as a reference the design of the CS-Runner [5], which provides KOALA with mechanisms for the efficient allocation of otherwise idle resources in a multicluster to Cycle-Scavenging (CS) applications (e.g., Parameter Sweep Applications). The CS-Runner initiates *Launchers*, which are a kind of pilot jobs, to execute the required set of parameters. The CS-Runner implements a grow/shrink mechansim that allows increasing or decreasing the number of Launchers when a resource offer or a resource reclaim is received from KOALA.

To schedule jobs, KOALA interfaces with the local resource managers of the clusters in the multicluster grid system. However, KOALA does not fully control the grid resources, as users may submit their jobs directly through the local resource managers deployed on each physical cluster.

III. Design Considerations

MR clusters may be isolated in two different ways in a multicluster system, as illustrated in Figure 1: either *across* different physical clusters (inter-cluster isolation), or *within* single physical clusters (intra-cluster isolation). In both cases, four types of isolation can be identified: performance isolation, data isolation, failure isolation, and version isolation, which we now describe in turn.

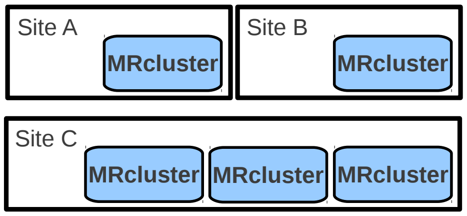


Figure 1. The two types of isolation: inter-cluster isolation (two MR clusters deployed within sites A and B) and intra-cluster isolation (three MR clusters deployed within site C).

A. Performance Isolation

With the standard FIFO scheduling technique, MR clusters executing long running jobs delay the execution of small jobs. To overcome this shortcoming, [8] proposes the FAIR MapReduce scheduler. With its dynamic resource allocation scheme, the small jobs receive their share of resources in a short time by reducing the number of nodes occupied by the large jobs. The execution time of the large jobs is increased, as they are forced to spawn across a smaller number of nodes than the actual MR cluster capacity. In this way, the small jobs gain their fair share of resources without long delays, at the expense of reducing the performance of the large jobs.

The trade-off between performance and fairness of the MapReduce jobs can be avoided by isolating the streams of small and large (or potentially even more classes of) jobs within separate MR clusters. Similarly, deploying multiple MR clusters allows large companies to isolate their production workloads from experimental jobs. Jobs in the development phase may need thorough testing and debugging before being launched on a production cluster. Thus, isolating them within a separate MR cluster first preserves the performance of the production MR cluster, and secondly, may reduce the debugging time for the developer.

B. *Data Isolation*

Users may form groups based on the data sets they want to process, such that the jobs of a certain group are executed within a separate MR cluster. Therefore, several data sets may be stored in different MR clusters, while the stream of jobs is demultiplexed into multiple substreams based on the data set they need to process. Data isolation is important for instance for systems research groups that want to anaylize the performance of single applications under controled conditions.

Furthermore, users may request their own MR cluster for running experiments within an isolated environment, or to guarantee the privacy and the security of their data. In the case of a single MR cluster deployment, the data is uniformly distributed across the nodes of the system, and the distributed file system is visible to all users. Thus, there is no privacy among the users of the system. Also, due to the lack of protection, users may intentionally or unintentionally alter the data of other users. For these reasons, there is a need to isolate the data sets within different MR clusters and to selectively allow access to read and process them.

C. *Failure Isolation*

A third type of isolation is the failure isolation. The MapReduce deployments are prone to both software (implementation or configuration errors) and hardware failures (server or network equipment failures). In both cases, failures of the system may cause loss of data, interruption of the running jobs, and low availability. By deploying multiple MR clusters, only the users of a specific MR cluster suffer the consequences of its failures.

D. *Version Isolation*

With multiple MR clusters we can enable access to different versions of the MapReduce framework at the same time. This is useful when upgrades of the framework are being made, or when new implementations are being developed. Testing, debugging, and benchmarking the frameworks, while having at the same time a running stable production MR cluster is enabled by our approach.

IV. Deploying Dynamic MapReduce Clusters with Koala

In this section, we present our approach for achieving isolation between multiple MR clusters. First, we explain the system model, then we describe the components of the KOALA resource management system, and finally, we propose three dynamic resizing policies.

A. *System Model*

An MR cluster relies on a two-layer architecture: a compute framework to facilitate an execution environment for MapReduce applications, and an underlying distributed file system that manages in a reliable and efficient manner large data volumes. Both layers are distributed across all nodes of an MR cluster, such that each node may execute tasks and also store data. A node of an MR cluster can be configured with multiple task slots such that each slot corresponds to a core of the processor available on the node. Based on the type of tasks to be executed, we distinguish map and reduce slots.

When growing or shrinking the MR cluster, the two layers need to be adjusted accordingly. While the execution framework can be easily resized without significant reconfiguration costs, changing the size of the distributed file system is more complex, because it may require rebalancing the data. As this operation is expensive, and may have to be performed frequently, we propose a hybrid architecture for MR clusters, with two types of nodes:

- The **core nodes** are the nodes that are initially allocated for the MR cluster deployment. They are fully functional nodes that run both the TaskTracker and the DataNode, and so they are used both for their compute power to execute tasks, and for their disk capacity to store blocks of data.
- The **transient nodes** are temporary nodes provisioned after the initial deployment of the MR cluster. They can be used as compute nodes that only run the TaskTracker, but do not store blocks of data and do not run the DataNode. Their removal does not change the distribution of the data.

B. *System Architecture*

This section explains how we have extended the original KOALA architecture to include MapReduce support. Figure 2 illustrates the interaction between the existing KOALA components and the additional components that extend the scheduler with support for deploying MR clusters on a multicluster system. The new components are the following: a specific KOALA runner called the MR-Runner, a specific MR cluster configuration module called the MR-Launcher, and the global manager of all active MR-Runners called the MR-ClusterManager.

KOALA is responsible for scheduling jobs, which in this case are complete MR clusters, received from the MR-Runners. Based on the desired size (number of nodes) of the MR cluster, KOALA schedules the job on the adequate physical cluster by applying one of its placement policies. To reduce the overhead of redistributing the data, we assume that the size of the MR cluster never decreases below the initial number of core nodes. Nevertheless, MR clusters may be resized by adding or removing transient nodes. The grow or shrink requests to the active MR-Runners are initiated by the scheduler itself, which tries to achieve fairness between multiple MR clusters.

KOALA monitors the availability of the resources through the KOALA Information System (KIS) module. When idle nodes are identified, the MR-Runners may receive grow requests. In contrast, in order to open up space for new job submissions, the scheduler may send shrink requests to the active MR-Runners.

After KOALA allocates nodes for the MR cluster deployment on a certain physical cluster, the MR-Runner interfaces with the local resource manager (e.g., Grid Engine) in order to proceed with the deployment of the MR cluster. The MR-Runner distinguishes one of the nodes as the master node, while the others are marked as slaves.

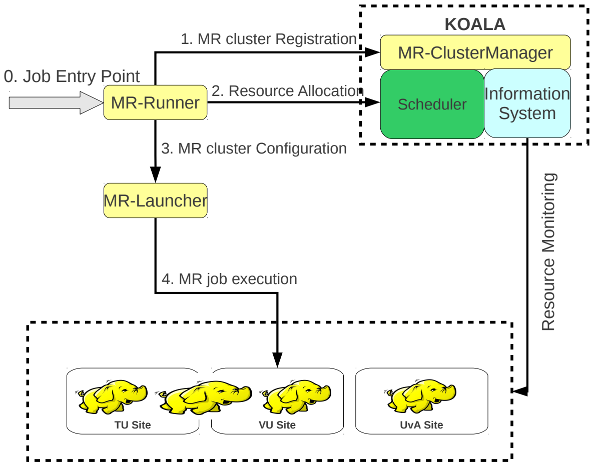


Figure 2. The MapReduce Runner and the Koala Grid Scheduler

From this point, the actual configuration of the MapReduce framework is realized through the MR-Launcher. The MR-Launcher configures the core nodes in two phases: first, it mounts the distributed file system on the local storage of the nodes, and then it installs the compute framework for executing MapReduce applications. Furthermore, the MR-Launcher is also able to configure and remove transient nodes, or to shut down the entire MR cluster. In the current implementation, the MR-Launcher uses the Hadoop daemons to configure the MR cluster: the NameNode and DataNodes for the HDFS, and the JobTracker and TaskTrackers for the compute framework.

Besides the scheduling and deployment functions, the MR-Runner also monitors several parameters of the MR cluster: the total number of (real) MapReduce jobs, the status of each such job, and the total number of map and reduce tasks. The monitoring process feeds a runner-side provisioning mechanism based on which the MR-Runner takes resizing decisions. We propose three provisioning policies, which we describe in detail in the next section.

The MR-ClusterManager is a central entity running on the scheduler side in order to maintain the metadata of each active MR cluster. To submit MapReduce jobs to an MR cluster scheduled through KOALA or to manipulate the data within the distributed file system, the user needs access to the corresponding metadata: the unique cluster identifier, the location of the configuration files, and the IP address of the master node. All the commands to submit MapReduce jobs or to access the distributed file system are executed on the master node of the MR cluster.

C. Resizing Mechanism

KOALA enables a two-level scheduling architecure. On the scheduler side, KOALA allocates resources for the MR cluster deployments based on a fair-share policy, such as the Equipartition-All or Equipartition-PerSite [5]. By monitoring the multicluster system utilization, the scheduler periodically offers additional nodes to the MR-Runners (grow requests), or reclaims previously provisioned nodes (shrink requests). Upon receiving a resource offer or reclaim from KOALA, the MR-Runner grows or shrinks the MR cluster depending on the ratio F of the number of running tasks (map and reduce tasks) and the number of available slots (map and reduce slots) in the MR cluster. The resizing mechanism dynamically tunes the value F between a minimum and a maximum threshold by adding nodes to or removing nodes from the MR cluster according to the following policy:

- **GSP (Grow-Shrink Policy):** The MR-Runner maintains the value of F to be between a minimum and a maximum threshold by accepting grow and shrink requests. On the MR-Runner side the user sets two constants, S_+ and S_-, representing the number of nodes the MR-Runner adds or removes whenever it receives a grow or shrink request. As the transient nodes may frequently join or leave the system, they do not contribute to the storage layer.

We compare the GSP with two basic policies, which accept every resource offer, and shrink only when the workload execution is completed:

- **GGP (Greedy-Grow Policy):** With this policy the MR-Runner accepts every resource offer regardless of the utilization of the MR cluster and ignores all shrink requests from KOALA. Thus, the MR cluster only grows in size, and only shrinks when the workload is finished. The provisioning is supported by transient nodes which do not contribute to the storage layer.
- **GGDP (Greedy-Grow-with-Data Policy):** Similarly to GGP, this policy makes the MR cluster grow in size every time a resource offer is received. As opposed to the previous policy, the GGDP is based on provisioning with core nodes instead of transient nodes. When a resource offer is received, the provisioned nodes are configured as core nodes, running both the TaskTracker and the DataNode. As a consequence, to avoid data redistribution, all shrink requests are declined.

V. Experimental Setup

This section presents the description of the multicluster grid system DAS-4, the Hadoop configuration parameters, and the workloads we generate for our experiments.

A. System Configuration

The infrastructure that supported our experiments is a wide-area computer system dedicated to research in parallel and distributed computing. The Distributed ASCI Supercomputer (DAS-4), currently in its fourth generation, consists of six clusters distributed in institutes and organizations across the Netherlands. As shown in Table I, the compute nodes are equipped with dual-quad-core processors at 2.4 GHz, 24 GB memory, and a local storage of 2 TB. The networks available on DAS-4 are Ethernet at 1 Gbps and the high-speed QDR Infiniband at 10 Gbps. The Grid Engine is configured on each cluster as the local resource manager.

We deploy KOALA as a meta-scheduler that interfaces with the local schedulers on each cluster in order to schedule and execute jobs. The MR-Runner is implemented in Java and currently configures Hadoop 0.20.203 clusters. The actual MR cluster configuration is realized through bash scripts which are executed within Java processes.

We configure the HDFS on a virtual disk device (with RAID0 software) that runs over two physical devices, with 2 TB storage in total. The data is stored in the HDFS in blocks of 64 MB or 128 MB with a default replication factor of 3. With 16 logical cores per node enabled through hyperthreading, we configure the TaskTrackers with 6 up to 8 map slots and 2 reduce slots, respectively. To avoid issues such as network saturation due to the limited bandwidth, the Hadoop daemons use the Infiniband network.

Table I
NODE CONFIGURATION

Processor	Dual quad-core Intel E5620
Memory	24 GB RAM
Physical Disk	2 ATA OCZ Z-Drive R2 with 2 TB (RAID0)
Network	10 Gbps Infiniband
Operating system	Linux CentOS-6
JVM	jdk1.6.0_27
MapReduce framework	Hadoop 0.20.203

B. Workloads

The Wordcount and Sort applications are two common MapReduce applications included in the Hadoop distribution that are used as MapReduce benchmarks in Hibench [9]. Wordcount counts the number of occurences of each word in a given set of input files. The map function simply emits key-value pairs for each word, while the actual counting is performed by the reducers. On the other hand, Sort transforms the input data from one representation to another. With both map and reduce functions implemented as identity functions, which do not modify the input data, the sort operation is performed by the MapReduce framework itself during the shuffling phase. As the framework guarantees that the intermediate key/value pairs are processed in increasing key order, the output generated is sorted.

Both small and large jobs are popular in MapReduce clusters. According to [10], 98% of the jobs at Facebook process 6.9 MB of data in less than a minute. On the other hand, Google reported in 2004 MapReduce computations that process terabytes of data on thousands of machines [4].

Based on the input size to process, we define 8 types of MapReduce jobs with the characteristics given in Table II. For each job, the number of reducers is set between 5% and 25% of the number of map tasks (same setting as in the workloads used in [8]). In addition, for the large jobs (more than 160 map tasks) we also use the common rule of thumb for setting the number of reducers: 90% or 180% of the number of available reduce slots in the MR cluster [3].

In Section VI, we will perform four experiments. For determining the CPU and disk utilization, we generate 100 GB input data using the RandomTextWriter and RandomWriter programs included in the Hadoop distribution. With the data block size set to 128 MB, a Wordcount or a Sort MapReduce job running on the given data set launches 800 map tasks. The same configuration is used to determine the speedup of the applications. To evaluate the performance of transient nodes, we generate data sets of 40 GB for Wordcount and 50 GB for Sort, with the data block size set to 128 MB. Finally, for the performance evaluation of the resizing policies, we generate a stream of 50 MapReduce jobs processing data sets from 1 GB to 40 GB split into data blocks of 64 MB, with an exponential inter-arrival pattern with a mean value of 30 seconds [8].

Table II
WORKLOAD CHARACTERISTICS

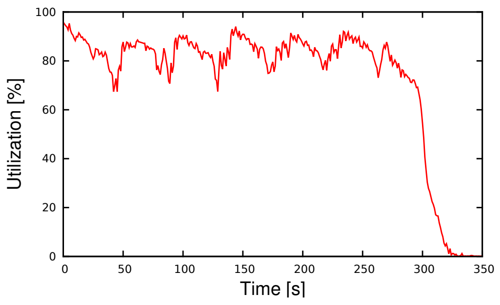
JobType	InputSize (GB)	BlockSize(MB)	Maps
0	100	128	800
1	50	128	400
2	40	128	320
3	40	64	640
4	20	64	320
5	10	64	160
6	5	64	80
7	2.5	64	40
8	1	64	16

VI. EXPERIMENTAL RESULTS

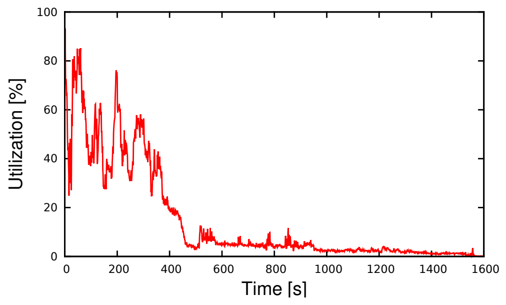
In our performance evaluation we investigate the CPU and disk utilization for each of the two applications, we determine the impact on the job response time of the numbers of core and transient nodes in an MR cluster, and we run benchmarks to compare the resizing policies.

A. CPU and Disk Utilization

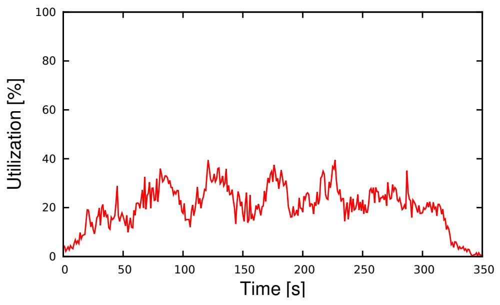
First, we seek to understand the characteristics of our workloads, and to this end, we monitor the resource utilization when running them. Thus, we run the Wordcount and Sort applications on the 100 GB data set (JobType 0) and we gather CPU and disk utilization statistics from every node of the MR cluster at 1-second intervals. The Linux tools we use for monitoring are *top* and *netstat*. The graphs in Figure 3 show the CPU and the disk utilizations as average



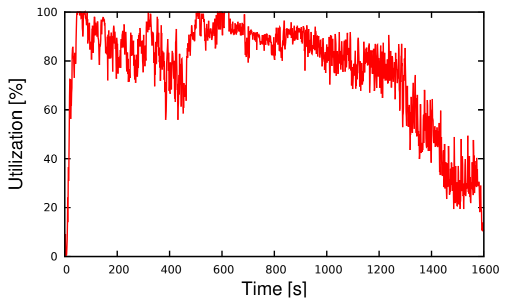
(a) CPU Utilization of Wordcount.



(b) CPU Utilization of Sort.



(c) Disk Utilization of Wordcount.



(d) Disk Utilization of Sort.

Figure 3. The CPU and disk utilization for the Wordcount and Sort applications.

values on all nodes of the MR cluster. The disk utilization of a node is the average of the values measured on the two physical hard disks.

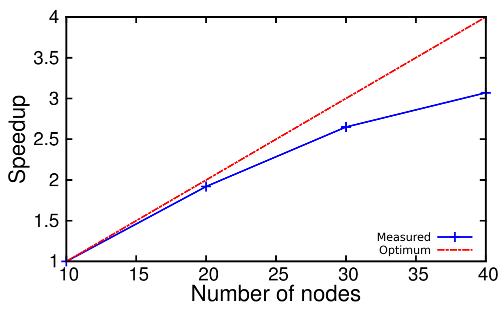
The CPU and disk utilizations illustrated in Figure 3 show that the Wordcount application is CPU-bound, and that the Sort application is IO-bound. For the Wordcount application we observe a long map phase with high CPU utilization, followed by a short reduce phase which has a low CPU utilization. The disk utilization is below 40% during the entire execution of the application. On the other hand, the figures for the Sort application show that it has a short map phase during which the CPU utilization oscillates between 40% and 60%, followed by a long reduce phase which is highly disk intensive and with very low CPU utilization.

B. Execution Time versus Fraction of Transient Nodes

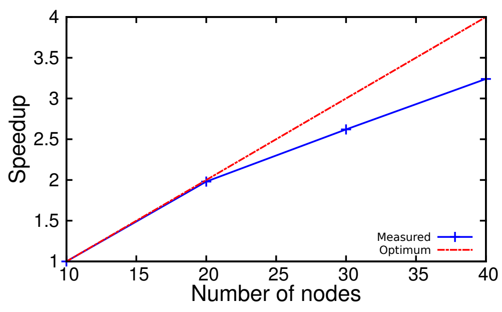
First, we assess the impact of the transient nodes on the exection time of single applications using static MR clusters, without any resizing mechanism. We set up MR clusters of 10 up to 40 nodes with the large data set as input data (JobType 0 for Wordcount and Sort). The Sort application launches 144 reduce tasks, representing 180% of the available reduce slots on the MR cluster with 40 nodes ($1.8 \times 2 \times 40$); for Wordcount we use a single reduce task. Figure 4 shows that the speedup for both applications on MR clusters with only core nodes is close to linear; the speedup is defined here relative to an MR cluster with 10 core nodes. Each data point in 4 was obtained by averaging the measurements over 3 runs.

Secondly, we run each application on MR clusters with a total of 40 nodes with a variable number of transient nodes that do not contain data. In Figure 5(a) we notice that the Wordcount application (JobType 2) has similar response times no matter how many transient nodes the MR cluster has (from 0 up to 30). On the other hand, Sort (JobType 1) shows a significant performance degradation when the number of transient nodes increases: the execution time for Sort doubles when the number of transient nodes increases from 0 to 30. Wordcount scales better on transient nodes than Sort due to the smaller amount of output data. While Wordcount generates less than 20 KB for the input data of 40 GB, in the case of Sort, the size of the output data equals the size of the input data, which is 50 GB.

We will now explain why the Sort application performs poorly on MR clusters with a large number of transient nodes. Let's consider an MR cluster with n_c core nodes and n_t transient nodes. At the end of the reduce phase, all $n_c + n_t$ nodes perform write requests on the local storages of the core nodes. The amount of output data generated by the transient nodes is $D_t = (n_t \times D_{out})/(n_c + n_t)$ Gbits, where D_{out} represents the size of the entire output data for Sort. This data is transferred across the network from the transient nodes to the local storages of the core nodes where the HDFS is mounted. The time to transfer D_t is $T_t = D_t/bw$, where bw represents the bandwidth available on the DAS-4 (10 Gbps on Infiniband). The time spent writing the data on the disks is $T_w = D_t/(2 \times n_c \times s_{write})$ seconds, where s_{write} denotes the write speed on the local disks and $2 \times n_c$ is the total number of disks available on the core nodes. For instance, an MR cluster with $n_c = 10$ core nodes and

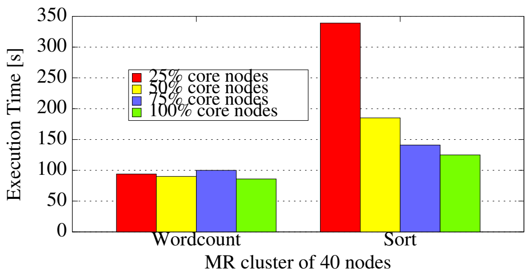


(a) Speedup of Wordcount.

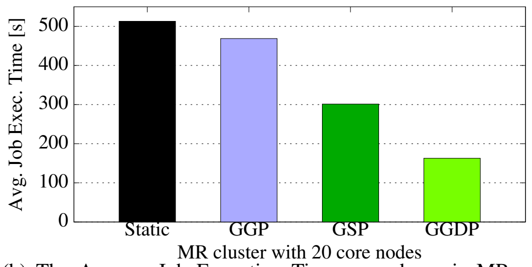


(b) Speedup of Sort.

Figure 4. The speedup for the Wordcount and Sort applications on MR clusters with only core nodes.



(a) Execution time versus the fraction of core nodes.



(b) The Average Job Execution Time on a dynamic MR cluster running 50 jobs.

Figure 5. The MR-Runner performance under single and multiple jobs workloads.

$n_t = 30$ transient nodes generates $D_t = 922$ Gbits. With $bw = 10$ Gbps we obtain $T_t = 92$ seconds. According to the specification of the devices, the read/write speed on the ATA OCZ disks is 1 Gbps and 900 Mbps, respectively. Therefore, T_w is 51 seconds. The smaller the number of core nodes, the higher the contention on the physical disks on which the HDFS is mounted. Therefore, as we can see in Figure 5(a), with 25% core nodes of the MR cluster capacity, the execution time of the workload is 2.7 higher than the ideal case of an MR cluster with 40 core nodes.

C. *Performance of the Resizing Mechanism*

To assess the performance of the MR cluster resizing mechanism, we run four benchmarks that execute successively a stream of jobs on a static MR cluster, and on a dynamic MR cluster with one of our three provisioning policies enabled (GGP, GGDP, or GSP). The MR cluster consists of 20 core nodes, and we assume a pool of 20 transient nodes to be available for provisioning during our experiments. The job stream consists of 50 jobs. For simplicity, we instrument the MR-Runner to initiate resource offers every T seconds, where T is set either to 30 seconds for the GSP, or to 120 seconds for the GGP and GGDP. For the last two policies, the size of the resource offer is 2 nodes. For the GSP, the MR cluster accepts a resource offer when F is greater than 1.5, and releases nodes when F is below 0.25. When the MR cluster frequently changes its size by adding or releasing a large number of nodes, the reconfiguration overhead may impact the performance of the running jobs. Therefore, we set S_+ and S_- to 5 and 2, respectively.

As can be seen in Figure 5(b), GGP shows a small improvement over the static approach, while GSP reduces the execution time by a factor of 1.7. Inuitively, GGP is not effective growing decisions are taken without considering the state of the MR cluster (e.g., if the MR cluster is idle, adding nodes is useless). Also, with a large number of transient nodes, the contention on the HDFS increases considerably. On the other hand, GSP monitors the number of tasks running within the MR cluster and initiates grow and shrink requests based on the throughput. However, the best policy is GGDP, which provides local storage for the provisioned nodes, reducing in this manner the costs of transferring the output data across the network and writing it on the disks of the core nodes.

VII. Related Work

The current state of the art work reveals several approaches for improving the performance of MapReduce clusters by different architectural enhancements that facilitate dynamic resizing or isolation.

Mesos [11] multiplexes a physical cluster between multiple frameworks such that different types of applications (MPI, MapReduce) may share access to large data sets. The scheduling mechanism enabled by Mesos is based on resource offers. The scheduler decides how many resources each framework is entitled to with respect to a fair-share policy, while the frameworks decide on their own which resources should be accepted based on the given framework side policies. Therefore, frameworks have the ability to reject an offer that does not comply with their resource requirements. By delegating the scheduling control to the frameworks, Mesos achieves high scalability, robustness, and stability. While our multi-MR cluster approach enables four types of isolation (performance, data, failure, and version isolation), Mesos achieves high utilization, efficient data sharing and resource isolation through OS container technologies, such as Linux Containers.

MOON [12] applies the hybrid architecture with core and transient nodes in order to deploy MapReduce clusters on volunteer computing systems. Towards this goal, a small number of dedicated nodes with high reliability are deployed behind the voluteer computing system for storage and computing power. To overcome the impact of unexpected interruptions, MOON proactively replicates tasks towards the job completion. In addition, MOON uses the dedicated nodes not only as data servers, but also to execute task replicas. In our design, the transient nodes are used to improve the performance of the MapReduce cluster deployed on the core nodes, as opposed to MOON, which uses the dedicated nodes to supplement the volunteer computing system.

Elastizer [13] estimates the impact of the cluster resource properties on the MapReduce job execution. In order to address a given cluster sizing problem, the Elastizer performs a search through the space of resources and job configuration parameters. This search is driven by a *what-if* engine that explores job profiles of MapReduce job executions to find the optimal set of resources and job configuration parameters. As opposed to the offline reasoning of the Elastizer, our approach uses the throughput as deciding factor for growing or shrinking the MapReduce cluster.

Our Koala scheduler along with the MR-Runner provide different types of isolation that improve the performance, data management, fault tolerance and development of MapReduce frameworks. In addition, the MR-Runner is enriched with a grow/shrink mechanism that dynamically changes the size of the MR cluster.

VIII. Conclusion

In this paper we have presented the design of a MapReduce runner for our KOALA grid scheduler that enables multiple deployments of MapReduce clusters within a multicluster system. The proposed system architecture achieves four types of isolation, with respect to performance, data, failures and versions. We enrich the MapReduce clusters with dynamic resizing capabilities and we incorporate three provisioning policies.

The performance evaluation shows that the CPU-bound applications scale on transient nodes as well as on core nodes, while the IO-bound applications suffer a high performance degradation when the number of transient nodes increases. We improve the performance of a static MR cluster with a grow/shrink mechanism that enables three provisioning policies: GGP, GSP, and GGDP.

As future work, the MR-Runner will be extended to support deployments of single MR clusters across multiple physical clusters. More thorough parameter study to find the optimal values for F, T, S_+, and S_- is desired as well.

IX. Acknowledgment

This publication was supported by the Dutch national program COMMIT.

References

[1] I. Raicu, I. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," *1st Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)*, pp. 1–11, 2008.

[2] H. Mohamed and D. Epema, "Koala: A Co-allocating Grid Scheduler," *Concurrency and Computation: Practice and Experience*, Vol. 20, pp. 1851–1876, 2008.

[3] T. White, *Hadoop: The Definitive Guide*. Yahoo Press, 2010.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Comm. of the ACM*, Vol. 51, no. 1, pp. 107–113, 2008.

[5] O. Sonmez, B. Grundeken, H. Mohamed, A. Iosup, and D. Epema, "Scheduling Strategies for Cycle Scavenging in Multicluster Grid Systems," *9th Int'l. Symp. on Cluster Computing and the Grid (CCGrid)*, pp. 12–19, 2009.

[6] O. Sonmez, N. Yigitbasi, S. Abrishami, A. Iosup, and D. Epema, "Performance Analysis of Dynamic Workflow Scheduling in Multicluster Grids," *19th Int'l. Symp. on High-Performance Distributed Computing (HPDC)*, pp. 49–60, 2010.

[7] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema, "Scheduling Malleable Applications in Multicluster Systems," *9th Int'l. Conference on Cluster Computing*, pp. 372–381, 2007.

[8] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," *5th European conference on Computer systems (EuroSys)*, pp. 265–278, 2010.

[9] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The Hibench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," *26th Int'l Conference on Data Engineering Workshops (ICDEW)*, pp. 41–51, 2010.

[10] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, "Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis," pp. 43–56, 2012.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," *8th Int'l. Symp. on Networked Systems Design and Implementation (NSDI)*, pp. 1–14, 2011.

[12] H. Lin, X. Ma, J. Archuleta, W. Feng, M. Gardner, and Z. Zhang, "Moon: Mapreduce On Opportunistic Environments," *19th Int'l. Symp. on High Performance Distributed Computing (HPDC)*, pp. 95–106, 2010.

[13] H. Herodotou, F. Dong, and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," *2nd Symp. on Cloud Computing*, pp. 18–31, 2011.