An Adaptive Distributed Simulator for Cloud and MapReduce Algorithms and Architectures

This is an author copy of the paper published at the
IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC 2014)

Pradeeban Kathiravelu
INESC-ID Lisboa
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: pradeeban.kathiravelu@tecnico.ulisboa.pt

Luis Veiga
INESC-ID Lisboa
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: luis.veiga@inesc-id.pt

***Abstract*—Scalability and performance are crucial for simulations as much as accuracy is. Due to the limited availability and access to the variety of resources, cloud and MapReduce solutions are often evaluated on simulator platforms. As the complexity of the architectures and algorithms keep increasing, simulations themselves become large and resource-hungry. Simulators can be designed to be adaptive, exploiting the clusters and data-grid platforms. This paper describes the research for the design, development, and evaluation of a complete fully parallel and distributed cloud and MapReduce simulator ($Cloud^2Sim$), leveraging the Java in-memory data grid platforms. $Cloud^2Sim$ provides a concurrent and distributed cloud simulator, by extending CloudSim cloud simulator, using Hazelcast in-memory key-value store. It also provides an assessment of the MapReduce implementations of Hazelcast and Infinispan, with means of simulating MapReduce executions. $Cloud^2Sim$ scales out the cloud and MapReduce simulations to multiple nodes running Hazelcast and Infinispan, based on load. The distributed execution model and adaptive scaling solution could further be leveraged as a general purpose auto-scaler middleware for a multi-tenanted deployment.**

I. Introduction

Cloud simulations are used in evaluating architectures, algorithms, topologies, and strategies that are under research and development, tackling many issues such as resource management, application scheduling, load balancing, workload execution, and optimizing energy consumption. While the exact environment of the cloud platform may not be accessible to the developers at the early stages of development, simulations give an overall idea on the related parameters, resource requirements, performance, and output.

With the increasing complexity of the systems that are simulated, cloud simulations are getting larger and the larger simulations tend to take longer and longer time to complete being run in a single node. Also, cloud simulation environments require a considerable amount of memory and processing power to simulate a complex cloud scenario.

Processors are increasingly becoming more powerful with multi-core architectures and the computing clusters in the research laboratories themselves can be used to run complicated large simulations in a distributed manner. However, current simulation tools provide very limited support to utilize these resources, as they are mostly written with a sequential execution model targeting to run on a single server.

Java in-memory data grids provide a distributed execution and storage model for problems in the grid-scale. They offer scalability and seamless integration with persistent storage. Hazelcast [1], Infinispan [2], Terracotta BigMemory[1], and Oracle Coherence [3] are some of the currently most used platforms for distributed execution and storage [4]. Using these platforms, users can create data grids and distributed cache, on the utility computers, to execute much larger jobs that cannot be run on any single computer, or that would take a huge time to execute often with a slower response.

Exploiting the existing simulation approaches that are heavily centralized, and the distributed execution platforms, cloud simulations can be made distributed, such that they can be able to utilize the computer clusters in the research labs. Distributed simulations can enable larger simulations to execute in a shorter time with a better response, whilst making it possible to simulate scenarios that may not even be possible on a single instance. Utilizing distributed computers to share the cycles to the simulation, as required by the simulation, would enable simulating bigger and more complex scenarios that cannot be simulated effectively in a single node, or it could be a very time consuming execution. While cycle sharing and volunteer computing are used in scientific research and grid computing projects, these models are not widely utilized to provide computing resources for cloud simulations. Moreover, when the resource providers are inside a trusted private network such as a research lab, security concerns related to cycle sharing can be considered lightly. Hence, the cycle sharing model can be leveraged to operate in a private cluster to provide a scalable middleware platform.

This paper describes $Cloud^2Sim$, an adaptively scaling middleware platform for concurrent and distributed cloud and MapReduce simulations, by leveraging CloudSim [5], [6] as the core simulation module, whilst taking advantage of the distributed shared memory provided by Hazelcast and in-memory key-value data grid of Infinispan. In the upcoming sections, we will further analyze the proposed adaptively scaling middleware platform for the simulations in a distributed and concurrent manner. Section II will address background information on cloud and MapReduce simulators, and distributed execution frameworks. Section III discusses the solution architecture of $Cloud^2Sim$, the proposed middleware platform, and how CloudSim is enhanced and extended as to become a distributed and concurrent cloud simulator. Section IV deals with the implementation details of $Cloud^2Sim$. $Cloud^2Sim$ was benchmarked against CloudSim and was evaluated on multiple nodes, with the results discussed in Section V. Finally, Section VI closes the paper discussing the current state of the research and future enhancements.

II. Related Work

A. Cloud Simulators

Some of the cloud simulators are quite generic, while others tend to be more focused to a narrower domain. Cloud

[1] http://terracotta.org/products/bigmemory

simulators have overlapping features as well as features specific to only a few simulators. CloudSim [5], EmuSim [7], and SimGrid [8], [9] are some of the mostly used general-purpose cloud simulation environments. OverSim [10] and PeerSim [11] are simulators for peer-to-peer and overlay networks. GridSim, a grid simulation tool, was later extended as CloudSim, a cloud simulation environment [6]. CloudSim is capable of simulating application scheduling algorithms, power-aware data centers, and cloud deployment topologies. It has been extended into different simulation tools such as CloudAnalyst [12], WorkflowSim [13], and NetworkCloudSim [14]. Simulation environments have a trade-off of accuracy/speed [15], with faster less-accurate simulators and slower accurate simulators. Researchers focus on enhancing the speed and accuracy of existing simulators. Extensions to CloudSim tend to address its limitations or add more features to it. NetworkCloudSim enables modeling parallel applications such as MPI and workflows in CloudSim [14]. WorkflowSim simulates scientific workflows through a higher level workflow management layer [13].

Optimizing the energy consumption is a major focus in cloud infrastructure, since power consumed by cloud data centers is enormous. Simulating energy-aware solutions has become part of the cloud simulators such as CloudSim [16], as energy-aware simulation is becoming a major research interest for cloud scientists. Simulators are also developed exclusively for power systems. Internet technology based Power System Simulator (InterPSS) is a distributed and parallel simulation tool for optimizing and enhancing the design, analysis, and operation of power systems [17].

B. MapReduce Simulators

As MapReduce applications and systems are developed with an increasing complexity, necessity to simulate MapReduce executions became apparent, in order to study their (and that of underlying algorithms') performance, efficiency, scalability, and resource requirements. Some of the MapReduce simulations were built from scratch, while some were developed on top of the existing simulation frameworks of cloud or network. MapReduce simulators are often built on top of the frameworks of the MapReduce implementation that they try to simulate, such as Hadoop.

SimMR is a MapReduce simulator that can replay the tasks from the logs of the real workloads produced by Hadoop, executing the tasks within 5% of the time the MapReduce task originally takes to execute [18]. MRPerf is a simulator of the MapReduce implementation of Hadoop, built using ns-2 [19]. Job execution time, amount of the data transferred, and time taken for each phase of the job are output from the simulator [20]. HSim, another Hadoop MapReduce simulator following the same design paradigm of MRPerf, claims to improve the accuracy of the MapReduce simulations for the complex Hadoop applications [21].

MRSG is a MapReduce Simulator built on top of SimGrid, providing APIs to prototype MapReduce policies and evaluate the algorithms [22]. Since MapReduce tasks are often run on bigger clusters, energy becomes a very important concern to address. BEEMR (Berkeley Energy Efficient MapReduce) is a MapReduce workload manager that is energy efficient [23].

C. Distributed Execution

Multiple Java in-memory data grids exist, both open source and commercial. Hazelcast and Infinispan are two of the open source in-memory data grids that are widely used in research.

Hazelcast is a distributed in-memory data grid that provides distributed implementations for the java.util.concurrent package [1]. Computer nodes running Hazelcast can join or create a Hazelcast cluster using either multicast or TCP-IP based join mechanisms. Additionally, Amazon web service EC2 instances with a hazelcast instance running, can use the Hazelcast/AWS join mechanism to form a Hazelcast cluster with the other EC2 instances. Multiple Hazelcast instances can also be created from a single node by using different ports, hence providing a distributed execution inside a single machine. As Hazelcast distributes the objects to remote JVMs, the distributed objects must be serializable, or custom serializers must be developed and registered for each of the classes that are distributed. Hazelcast custom serialization requires the classes to be serialized to have public setters and getters for the properties that need be serialized. Hazelcast is partition-aware, and exploiting its partition-awareness, related objects can be stored in the same instance, reducing the data transmission and remote invocations. Hazelcast supports both synchronous and asynchronous backup for fault tolerance. Hazelcast has been already used in research to distribute the storage across multiple instances [24].

Infinispan is a distributed key/value data-grid [2]. As an in-memory data-grid, Infinispan has been used in many researches. Palmieri et al have developed a self-adaptive middleware platform to provide transactional data access services, based on the in-memory data management layer of Infinispan [25]. JBoss RHQ [2] provides an enterprise management solution for Infinispan as well as the other projects from JBoss, which can be used to monitor the state and health of the Infinispan distributed cache instances. Infinispan offers JCache[3] and MemCached[4] APIs. Goal-oriented self-adaptive scalable platforms are researched using Infinispan as an in-memory persistence and cache solution [26].

In-memory data grids and cycle sharing model provide resources for a distributed execution. They can be leveraged to execute larger simulations, to increase the performance of existing simulators, without sacrificing the performance. Currently, no cloud or MapReduce simulator is able to provide scale-out.

III. Solution Architecture

As designed to run top of a cluster, $Cloud^2Sim$ attempts to execute larger and more complicated simulations that would not run on a single node or terminal, or consume huge amount of time. A cluster of shared resources can be built over a cluster of computers, using the in-memory data grid frameworks. Simulations are executed on the cluster, utilizing the resources such as storage, processing power, and memory, provided by the individual nodes, as indicated by Figure 1. Hazelcast and Infinispan are used as the in-memory data grid libraries in $Cloud^2Sim$.

[2] http://rhq.jboss.org/

[3] https://jcp.org/en/jsr/detail?id=107

[4] http://memcached.org/

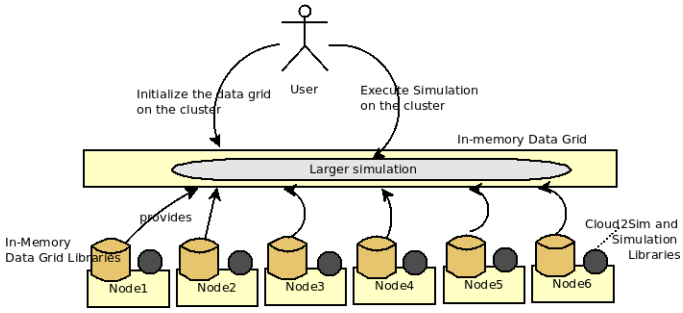

Fig. 1. High Level Use-Case of $Cloud^2Sim$

$Cloud^2Sim$ functions in two basic modes as a concurrent and distributed simulator: cloud and MapReduce. It was decided to extend an existing cloud simulator to be concurrent and distributed, instead of writing a new cloud simulator from the scratch, to be able to take advantage of existing simulations. Developed as a Java open source project, CloudSim can be easily modified by extending the classes, with a few changes to the CloudSim core. Its source code is open and maintained. Hence, CloudSim was picked as the core simulation module to build the distributed simulator, while configuring and leveraging Hazelcast to distribute the storage of Vm, Cloudlet, and Datacenter objects and also to distribute the execution, according to the scheduling, to the instances in the cluster. Users have the freedom to choose Hazelcast based or Infinispan based distributed execution for the cloud and MapReduce simulator, as the simulator is implemented on top of both platforms following the same design. Classes of CloudSim are extended and a few are also modified to be able to extend CloudSim with further functionality. External dependencies such as Hazelcast and Infinispan are used unmodified, for added transparency and portability. The definition of cloud simulations and MapReduce simulations are independent by design. Cloud and MapReduce simulations can be executed independently, though experiments can be run utilizing both cloud and MapReduce simulations.

A. Distributed Middleware Platform for Simulations

As multiple instances execute a single simulation, measures are taken to ensure that the output is consistent as if simulating in a single instance, while having enhanced performance and scalability. Data is partitioned across the instances by leveraging and configuring the in-memory data grid. Each member of the cluster executes part of the logic on the objects that are stored in the local partitions of the respective nodes.

Execution of simulations is improved, by leveraging the multi-core environments, and exploiting the multi-threaded programming. While CloudSim provides some means for a concurrent execution, its support is very limited. Simulations should be executed utilizing the multi-threaded environments, where the simulator itself runs the tasks concurrently, whenever that is possible and efficient. Runnables and callables are used to submit tasks to be run in a separate thread, while the main thread is executing its task. The relevant check points ensure that the threads have finished their execution and the values are returned from the callables, as required.

A cluster can be formed by multiple instances. Multiple clusters can be used to execute parallel cloud or MapReduce simulations, as multiple tenants of the nodes. As each cluster is unaware of the other clusters, tenant-awareness is ensured so that the parallel experiments can be independent and secured from the other parallel simulations.

Pulling data from each of the nodes for execution has a higher communication cost. To overcome this, the data locality features provided for Hazelcast distributed executors are leveraged and used appropriately to send the logic to the data instead. Partition-awareness feature of Hazelcast is exploited in storing the distributed objects, such that the data that are associated with each other are stored in the same partition to decrease the remote invocations. Partitioning of data and execution is done by 3 different strategies, as listed below.

1. Simulator - Initiator based Strategy*:* $Simulator$ is the complete $Cloud^2Sim$ with the simulation running. A Hazelcast instance is started by $Cloud^2Sim\ Initiator$, which keeps the computer node connected to the Hazelcast cluster, offering the resources of the node to the data grid. The $Simulator$ instance is run from the master instance, where an instance of $Initiator$ is spawned from the other instances. Simulator acts as the master, distributing the logic to the Initiator instances. Part of the logic is executed in the master itself, and the execution is partitioned uniformly among all the instances, using the ExecutorService.

2. Simulator - SimulatorSub based Strategy*:* One instance contains the Simulator, which is the master, where others execute SimulatorSub, which are the slave instances. Master coordinates the simulation execution. Execution is started by all the instances and parts of the execution are sent by each instance respectively to the other instances, using the ExecutorService. Hence, the load on the master is reduced. Some of the unparallelizable tasks can be delegated to the primary worker, which is an instance other than the master instance, that is decided upon the cluster formation. This mitigates overloading the master instance.

3. Multiple Simulator Instances Strategy*:* There is no predefined Simulator master in this strategy. The instance that joins first becomes the master at run time, where other instances function as $SimulatorSub$ instances. Logic is partitioned across the instances using the partitioning algorithms defined in $Cloud^2Sim$ distributed data center brokers. $PartitionUtil$ manages the partitioning of the data and execution, manipulating the data structures across the instances. It provides the initial and final IDs of the data structure such as cloudlets and VMs, given the total number of the data structure elements and the initial offset.

The *Simulator - Initiator based Strategy* is chosen for the implementation of the simulations that are effectively scheduled by the single master to all the instances that are joined, such as the MapReduce simulator. The *multiple Simulator instances strategy* is used in the CloudSim simulations such as the simulation of matchmaking-based application scheduling, where the simultaneous instances are more effective, than having a single static master that handles most of the task. The *Simulator - SimulatorSub based strategy* is proposed for the compound simulations involving both Cloud and MapReduce executions, or simulating MPI workflows. The *multiple Simulator instances strategy* is usually preferred over the *Simulator - SimulatorSub based strategy* as it is easier to maintain since it does not

fragment the logic, and also electing the master at run time is more effective in terms of scalability and fault-tolerance.

Existence of the master instance is always ensured in the *multiple Simulator instances strategy*. The instance that joins the cluster as the first instance in the cluster becomes the master, where in the *Simulator - SimulatorSub based strategy*, the instance of Simulator should be manually started before the sub instances, and this may become a bottleneck. Moreover, when backups are available, the *multiple Simulator instances strategy* is resilient to failures as when the assigned master fails, another instance can take over as the master. This is not possible in the other strategies, as the master is chosen statically, and the other nodes do not contain the same code as the master instance.

Cloud Simulations: $Cloud^2Sim$ is designed on top of CloudSim, where cloud2sim-1.0-SNAPSHOT can be built using Maven independently without rebuilding CloudSim. Modifications to CloudSim are very minimal. $Cloud^2Sim$ enables distributed execution of larger CloudSim simulations. The compatibility layer of $Cloud^2Sim$ enables the execution of the CloudSim simulations with minimal code change, on top of either the Hazelcast and Infinispan based implementations, or the pure CloudSim distribution, by abstracting away the dependencies on Hazelcast and Infinispan, and providing a compatible API.

MapReduce Simulations: Design of the MapReduce simulator is based on a real MapReduce implementation. A simple MapReduce application executes as the Simulator is started. The number of times map() and reduce() are invoked can easily be configured. The MapReduce simulator is designed on two different implementations, based on Hazelcast and Infinispan, making it possible to benchmark the two implementations against each other. Multiple simulations are executed in parallel, without influencing others, where an instance of a coordinating class could collect the outputs from the independent parallel MapReduce jobs carried out by different clusters.

B. Scalability and Elasticity

$Cloud^2Sim$ achieves scalability through both static scaling and dynamic scaling. Static scaling is the scenario where $Cloud^2Sim$ uses the storage and resources that are initially made available, when instances are started and joined manually to the execution cluster. Multiple nodes can be started simultaneously at the start-up time for large simulations that require large amount of resources. $Initiator$ instances can also be started manually at a later time, to join the simulation that has already started. Simulations begin when the minimum number of instances specified have joined the simulation cluster. $Cloud^2Sim$ scales smoothly as more Hazelcast instances join the execution.

Scaling can also be achieved by $Cloud^2Sim$ itself dynamically without manual intervention, based on the load and simulation requirements. When the load of the simulation environment goes high, $Cloud^2Sim$ scales itself to handle the increased load. Dynamic scaling of $Cloud^2Sim$ provides a cost-effective solution, instead of having multiple instances being allocated to the simulation even when the resources are under-utilized. Since scaling introduces the possibility of nodes joining and leaving the cluster, as opposed to the static execution or manual joins and exits of instances, scalable simulation mandates availability of synchronous backup replicas, to avoid losing the distributed objects containing the simulation data upon the termination of an instance.

A health monitor was designed to monitor the health of the instances, and trigger scaling accordingly. The health monitoring module runs from the master node and periodically checks the health of the instance by monitoring the system health parameters such as the process CPU utilization, system CPU utilization, and the load average. Based on the policies defined in the configuration file, the health monitor triggers the dynamic scaler. When the current observed value of the monitored health parameter (such as load average or process or system CPU utilization) is higher than the $maxThreshold$ and the number of total instances spawned is less than the $maxInstancesToBeSpawned$, a new instance will be added to the simulation cluster. Similarly, when the current observed value is lower than the $minThreshold$, an instance will be removed from the simulation cluster.

During scale out, more instances are included into the simulation cluster, where scale in removes instances from the simulation cluster, as the opposite of scale out. Dynamic scaling is done in two modes - auto scaling and adaptive scaling, as discussed below.

Auto Scaling: By default, the $Cloud^2Sim$ auto scaler spawns new instances inside the same node/computer. When there is only a limited availability of resources in the local computer clusters that is insufficient to simulate a large scenario, $Cloud^2Sim$ can be run on an actual cloud infrastructure. Hazelcast can be configured to form a cluster on Amazon EC2 instances, with the Hazelcast instances running on the same AWS[5] account. When using AWS join mechanism provided by Hazelcast to form the cluster, Hazelcast uses the access key and secret key to authorize itself into forming the cluster. Ports that are involved in Hazelcast clustering should be open and permitted in the EC2 instances. Scaling can be triggered by the $Cloud^2Sim$ health monitoring or using the scaling policies configured with AWS Auto Scaling and Amazon Cloud Watch.

Adaptive Scaling: Adaptive Scaling is a scenario, where in a clustered environment, more computer nodes will be involved in an application execution based on the load. More instances will be attached to the simulation cluster when the load is high, and instances will be detached or removed from simulation when the load is low. Scaling decisions are made in a separate cluster.

The health monitor in the main instance monitors the load and health status of the main instance with simulation running in $cluster - main$, and shares this information with the $AdaptiveScalerProbe$ thread in $cluster - sub$, using the local objects, as they are from the same JVM. $AdaptiveScalerProbe$ shares this information with $IntelligentAdaptiveScaler$ (IAS) instances, which are threads from all the other nodes that are connected to $cluster - sub$.

When IAS from one instance notices the high load in the master, it spawns an Initiator instance in the $cluster - main$, and sets the flag to false to avoid further scaling outs/ins.

[5] https://aws.amazon.com/

Monitoring for scaling out happens when there is no Initiator instance in the node, and monitoring for scaling in happens when there is an Initiator instance, for each individual node. This ensures 0 or 1 of Initiator instances in each node, and avoids unnecessary hits to the Hazelcast distributed objects holding the health information. Since IAS is in a separate cluster (cluster-sub) from the simulation (cluster-main), the executions are independent.

Adaptive Scaling is used to create prototype deployments with elasticity. When the simulations complete, the Hazelcast instances running in the cluster-main will be terminated, and the distributed objects stored in the cluster-sub will be cleaned. These instances just require Hazelcast and the adaptive scaler thread to keep them connected, providing their CPU, memory, and storage for the simulation work voluntarily, in a BOINC[6]-like cycle sharing model. The entire simulation code can be loaded and kept only on the master and exported transparently to other nodes joining it, and execute from all the nodes, following the *Simulator - Initiator based strategy*. All the member nodes are from the same network, that they have joined by TCP-IP or multicast. Hence the cycle sharing in $Cloud^2Sim$ is not public as in voluntary computing. Due to this nature, the security implications involved in voluntary computing are not applicable to $Cloud^2Sim$.

The scaling decision flag should be get and set in a concurrent and distributed environment atomically, ensuring that exactly one instance takes action of it. Access to the object that is used as the flag must be locked during update from any other instance in the distributed environment.

Multiple Hazelcast clusters can be run from a single computer cluster or even a single machine. By exploiting this feature, multiple experiments can be run on $Cloud^2Sim$ in parallel, as different clusters are used for independent simulations. The adaptive scaler is further extended to have the node cluster providing its resources to different applications or simulations running on different Hazelcast clusters. Figure 2 shows the execution of two independent simulations in a cluster with adaptive scaling. The adaptive scaler functions as a $Coordinator$ instance, coordinating and allocating its resources to multiple tenants. Here, instead of representing the scaling decisions using single keys, distributed hash maps are used, mapping the scaling decisions and health information against the cluster or tenant ID. Similarly, the pointers to the master instances are mapped against the cluster ID, making it possible to refer to and coordinate multiple tenants from the coordinator.

C. Software Architecture and Design

Distributed storage and execution for CloudSim simulations is achieved by exploiting Hazelcast. Infinispan integration with the compatibility layer ensures easy integration of Infinispan to replace Hazelcast as the in-memory data grid for CloudSim simulations. Figure 3 depicts a layered architecture overview of $Cloud^2Sim$, hiding the fine architectural details of CloudSim.

Hazelcast monitoring and heart beats are run on a separate thread, hence not interfering with the main thread that runs the simulations. Simulation objects, cloudlets and VMs were ported from Java lists to Hazelcast distributed maps. This enabled

[6] http://boinc.berkeley.edu/

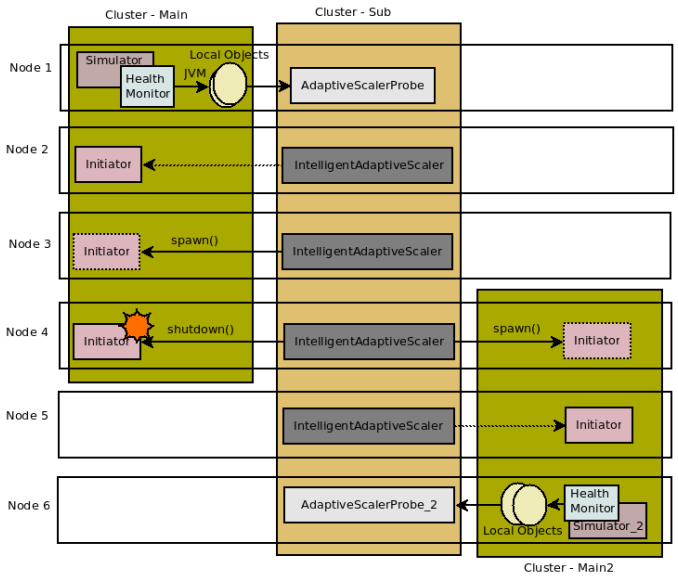

Fig. 2. An Elastic Deployment of $Cloud^2Sim$

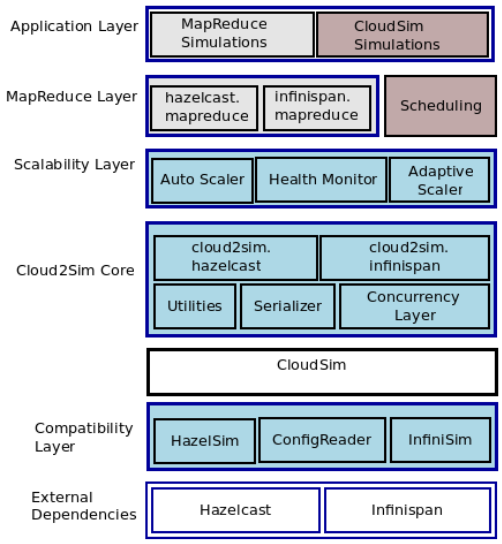

Fig. 3. $Cloud^2Sim$ Architecture

storing these objects in a distributed shared memory provided by Hazelcast spanning across the cluster. Instances of Hazelcast $IMap$ are used as the data structure. The core CloudSim class, $CloudSim$ is extended as $HzCloudSim$ to address the Hazelcast specific initializations. Similarly, $Cloudlet$ and Vm are extended as $HzCloudlet$ and $HzVm$ respectively. This extended class hierarchy enabled modifying the internals of Vm and Cloudlet classes by sub-classing them to use Hazelcast distributed maps as the storage data structure, instead of Java lists. As extending CloudSim, $Cloud^2Sim$ provides an API compatible with CloudSim, for the cloud simulations. Classes of CloudSim are extended as shown by Table I, while preserving the invocation interfaces and preserving code generality.

The scheduling package handles scheduling in complex scenarios that involve searching large maps consisting of VMs,

TABLE I. $Cloud^2Sim$ AND CLOUDSIM

$Cloud^2Sim$ **Class**	**Extended CloudSim class**	**Core Responsibilities**
HzCloudSim	CloudSim	* Core class of the Simulator * Initializes distributed data structures
HzDatacenterBroker	DatacenterBroker	* Implements distributed scheduling
$Cloud^2Sim$Engine	-	* Starts Simulation based on the configuration * Starts supportive threads for scaling and health monitoring
PartitionUtil	-	Calculates the partitions of the data structures
HzCloudlet	Cloudlet	* Extends Cloudlet
HzVm	Vm	* Extends Vm
HazelSim	-	* Singleton of Hazelcast integration
HzObjectCollection	-	* Provides unified access to distributed objects

cloudlets, and the user requirements. Distributed application scheduling is done by the extended data center brokers that are capable of submitting the tasks and resources in a distributed manner. A new package named "compatibility" composed of the core classes such as $HazelSim$ is placed inside CloudSim to integrate Hazelcast, Infinispan, and other new dependencies, and to enable multiple modes of operation (Such as Hazelcast or Infinispan based and regular CloudSim simulations).

The concurrency layer consists of callables and runnables for asynchronous invocations to concurrently execute. As complex objects must be serialized before sending them to other instances over the wire, custom serializers are needed for Vm, $Cloudlet$, $Host$, $Datacenter$, and the other distributed objects to be able to distribute them across the instances, store and access them remotely in a binary format, effectively. The utilities module provides the utility methods used throughout $Cloud^2Sim$.

The MapReduce layer provides MapReduce representation and implementations based on Hazelcast and Infinispan MapReduce modules. MapReduce Simulator can be configured with health monitoring and scaling. Hence, the execution time for varying the number of map() and reduce() invocations as well as the health parameters such as load average and CPU utilization can be measured.

IV. IMPLEMENTATION

Based on the design, $Cloud^2Sim$[7] was implemented as a concurrent and distributed cloud and MapReduce simulator.

A. Concurrent and Distributed Cloud Simulator

CloudSim simulations can run on $Cloud^2Sim$ with minor changes to facilitate distribution. Distributing the simulation environment has been implemented using an incremental approach. The CloudSim trunk version was forked and used in the implementation. Hazelcast version 3.2 and Infinispan version 6.0.2 were used in the implementations and evaluations. A complete distributed cloud simulator was built with Hazelcast, having CloudSim as the core simulation module.

Sample concurrent simulations were implemented, with concurrent data center creation, concurrent initialization of VMs and cloudlets, and submission of them to the brokers. Though the initialization of threads and executor frameworks introduced an overhead for small simulations, it provided a speed-up for the larger simulations. Very small simulations do not require distributed execution, as they perform reasonably well, and were never the target of this work. Simulations that fail to execute or perform poorly due to the processing power requirements on a single thread, perform much better on the concurrent environments utilized by $Cloud^2Sim$. Hence, the overheads imposed by the initializations is not a limitation to usability, as the performance gain is higher. Sample prototype developments with concurrent creation of data centers showed an increased performance, overcoming the overheads.

Hazelcast $IExecutorService$ was utilized to make the execution distributed. While MapReduce executions are effectively executed in the *Simulator - Initiator based strategy*, cloud simulations rather follow a model where all the instances initiate and send logic fractions. Initially implemented as different classes, following *Simulator - SimulatorSub based strategy*, the master and other instances were later unified, following the *multiple Simulator instances Strategy*, such that a same $Simulator$ class can be run from all the instances. The first instance to join the cluster becomes the master and executes the core fractions of the logic which must not be distributed, decentralized, or run in parallel for a correct execution of the simulation. Callables and runnables were made to implement $HazelcastInstanceAware$ interface, to ensure the members of the clusters executed part of the logic on the data partition that is stored in themselves, to minimize remote invocation, by increasing data locality. $Cloud^2Sim$ optimizes the data locality of the distributed objects by storing the related objects together, as they frequently access each other.

Partitioning of data and execution is calculated iteratively for each instance. The number of instances currently in the cluster is tracked by an instance of distributed map, called $deploymentList$. An instance will have an offset value assigned to it, which is the number of instances that have joined previously. Hence the offset of the first instance will be zero and initial ID of the partition will be zero as well. Final ID of the data partition of the instance that joins last, will be same as the last ID of the distributed data structure.

$Cloud2SimEngine$ is started as the initial step of $Cloud^2Sim$ cloud simulations. $Cloud2SimEngine.start()$ starts the timer and calls $HzConfigReader$ to read the configurations. If health checks are enabled, it starts the health monitor thread, to periodically monitor the instance status and report as configured. If adaptive scaling is enabled, it also starts the AdaptiveScalerProbe in a separate thread, to communicate with the $IntelligentAdaptiveScaler$ instances in the other nodes to adaptively scale the simulation. It finally initializes HzCloudSim, where Hazelcast simulation cluster is initialized with the simulation job, and CloudSim simulation is started. Data centers are created concurrently. Brokers extending $HzDatacenterBroker$ create instances of $HzVm$ and $HzCloudlet$ and start scheduling in a distributed manner, using all the instances in the simulation cluster. The core simulation is started using $HzCloudSim.startSimulation()$, and executed by the master instance. When the simulation finishes, the final output is logged by the master instance. Based on the simulation, the instances are either terminated or their distributed objects are cleared and the instances are reset for the next simulation.

Scalable Middleware Platform: The health monitoring module exploits the system

[7] Checkout the source code at https://sourceforge.net/p/cloud2sim/code/ci/master/tree/, with user name, "cloud2sim" and password, "Cloud2Simtest".

health information that can be retrieved using $com.sun.management.OperatingSystemMXBean$. It facilitates scaling based on a few parameters such as CPU load, and also provides an API to extend it further. Scaling policies are defined on the maximum and minimum thresholds of the defined properties, along with the maximum and minimum number of instances that should be present in the simulation cluster. Once a new instance is spawned, the adaptive scaler will wait for a user-defined period, which is usually longer than the time interval for health checks, for the next scaling action. This longer wait between scaling decisions prevents cascaded scaling and jitter where multiple instances are added or removed at once, or within a very short period of time interval, during the time taken for the scaling effect to be reflected on the simulation. The gap between the high and low thresholds is kept reasonably high, to prevent the jitter effect, where instances are added and removed frequently, as the high and low thresholds are frequently met. The health monitor configuration provides means to configure the scaling and monitoring to fit the application requirements and extend the module further to fine tune according to the application requirements.

Scaling decisions are made atomically in the distributed environment, to make the adaptive scaling work without simultaneously starting or shutting down multiple instances. This is implemented using the distributed flags, leveraging Hazelcast IAtomicLong data structure.

B. MapReduce Simulator

MapReduce Simulator has two different implementations, based on Hazelcast and Infinispan. A basic MapReduce word count application was implemented using Hazelcast and Infinispan and incorporated into $Cloud^2Sim$. Complex MapReduce scenarios were simulated using this small application.

Infinispan is integrated using the compatibility layer in CloudSim, to facilitate later migration of $Cloud^2Sim$ to Infinispan. This also enables the same design and architecture for both Hazelcast and Infinispan based distributions. A transactional cache is created from the cache manager. An instance of cache in Infinispan is similar to an instance in Hazelcast. Multiple instances of Cache form a cluster and execute the jobs. Simulator and Initiator instances are created using the same configurations. The cache instance initialized by the master node acts as the supervisor of the MapReduce jobs, and distributes the tasks across the Initiator instances.

V. Evaluation

A computer cluster with 6 identical physical nodes (Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz and 12 GB memory) was used for the evaluation. Multiple simulations were experimented on the system using 1 to 6 nodes. Each node executed one Hazelcast or Infinispan instance, except during the experiments involving multiple instances in a single node.

A. Initial Tests

The master node always completes the last, as the final outcome is printed by the master node in the simulations considered. Time taken by the master node is noted down, as the other nodes finish the execution before the master.

TABLE II. Execution time (sec) for CloudSim Vs. $Cloud^2Sim$

Deployment	Simple Simulation	Simulation with a cloudlet workload
CloudSim	3.678	1247.400
$Cloud^2Sim$ (1 node)	20.914	1259.743
$Cloud^2Sim$ (2 nodes)	16.726	120.009
$Cloud^2Sim$ (3 nodes)	14.432	96.053
$Cloud^2Sim$ (6 nodes)	20.307	104.440

Table II shows the time taken to simulate a round robin application scheduling simulation with 200 users, 15 data centers, with and without a cloudlet workload for a varying number of VMs and cloudlets. CloudSim outperformed $Cloud^2Sim$ in the base execution without a workload, due to the dominant inherent coordination overhead involved in $Cloud^2Sim$. $Cloud^2Sim$ with multiple nodes showed a considerable 10-fold improvement in the execution time when the cloudlets contained a relevant workload to be simulated once scheduled. Time taken (in seconds) for an experiment in $Cloud^2Sim$ with 1, 2, 3, and 6 nodes as well as in CloudSim is depicted by Table II for 200 VMs and 400 cloudlets.

In a simulation where each cloudlet does a complex job, the time taken for the simulation increases with the number of cloudlets. With the number of VMs fixed at 200, simulation time taken on 1 - 6 nodes was measured. Figure 4 depicts how the application scales with varying number of cloudlets. As the size of the simulation is increased, performance is seen increasing with the number of nodes, depicting the suitability of the distributed execution model for larger simulations.

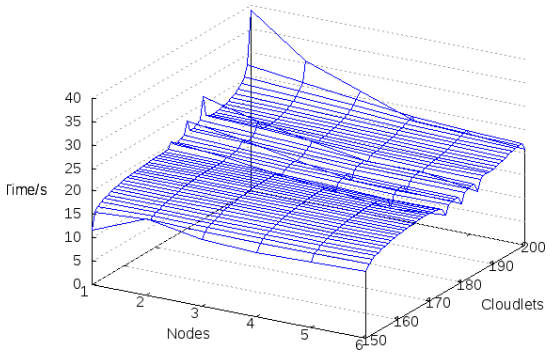

Fig. 4. Simulation Time for Application Scheduling Scenarios

B. Distributed CloudSim Simulations

The experiment was repeated with different combinations of VMs (from 100 - 200) and Cloudlets (from 100 - 400), with and without a complex mathematical operation to be performed for each cloudlet as a load. Distributed objects and distributed execution were monitored by Hazelcast Management Center. Objects were uniformly distributed among the available Hazelcast instances, consuming almost the same amount of entry memory from each instance. Partitions of different instances were equally hit or accessed. This shows an effective partitioning of the distributed objects by Hazelcast. Parameter 'isLoaded' is set to true, for a cloudlet workload. Four distinct cases of scalability were noticed, as described below.

*1) **Success Case (Positive Scalability)**:* Figure 5 depicts the scenarios of (noOfVMs = 200, noOfCloudlets = 400, isLoaded

= true) and (noOfVMs = 100, noOfCloudlets = 200, isLoaded = true), where the time taken for simulation is decreasing with the number of nodes. This is a desired scenario of scaling where the task is so much CPU intensive for each cloudlet to handle in a single node, such that introducing more nodes distribute the tasks, reducing the simulation time.

Dynamic Scaling: With the dynamic scaling enabled, this case introduced more instances into the execution, as the load goes high. Memory used by the application as a percentage of the total memory used was used as the health monitoring measure. With the adaptive scaling, the environment of 200 VMs and 400 cloudlets with load scaled up to 3 instances, for a CPU utilization of 0.20, even when more than 3 instances were included in the sub-cluster. Reducing the maximum threshold made the main-cluster to scale out earlier and faster, involving all the available instances to the simulation. Figure 5 shows the time taken for the simulations with and without adaptive scaling. As shown by Figure 5, the execution time is converging as more nodes are added. Hence, introducing further nodes beyond a certain maximum number of nodes may not be economically feasible, and at a point this may become the case 3, which is explained below as the *common case*.

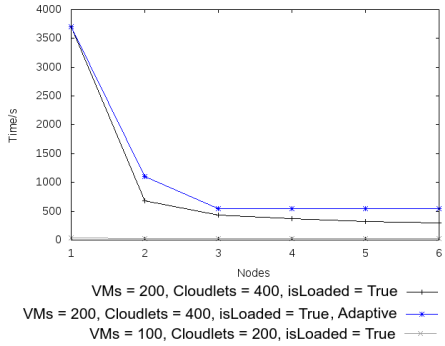

Fig. 5. Distributed Execution - Positive Scalability

Adaptive scaling was not observed in the other cases, except when the maximum process CPU load is reduced below 0.15 from the configurations. This shows that a single instance was sufficient to run the sample simulations of the other 3 cases discussed below. For the scenario of scale ins, synchronous backups should be enabled to prevent the data loss, which eliminates the possibility of a fair comparison, as the simulations with the fixed number of instances are run with no backups. Hence, the low threshold was kept low enough during the experiment, such that there were no scale ins. Scale in was observed, when the minimum process CPU load was increased beyond 0.02.

Load Average: With adaptive scaling configured, load average was logged during the execution. Table III shows the load averages observed during and after the scaling events, for the simulation environment with 6 nodes available. Up to 3 nodes were involved in the simulation by the $IntelligentAdaptiveScaler$. Waiting time acts as a buffer to prevent cascaded scaling events. Health is monitored periodically, except during the buffer time introduced immediately following the scaling events. These intervals are configured to fit the requirements and the nature of the simulation.

TABLE III. LOAD AVERAGES WITH ADAPTIVE SCALING ON 6 NODES

No. of Instances	Master I0	I1	I2	Event
1	0.30	-	-	Spawning - I1
2	0.30	0.24	-	Waiting Time
2	0.25	0.24	-	Spawning - I2
3	0.23	0.23	0.13	Waiting Time
3	0.21	0.19	0.13	Health Monitoring
3	0.09	0.18	0.09	Health Monitoring
3	0.06	0.18	0.08	Health Monitoring

*2) **Other Cases of Scalability:*** Figure 6 depicts 3 distinct cases of scalability, where the execution time changes in different patterns with the increasing number of nodes. The scenarios are analyzed below.

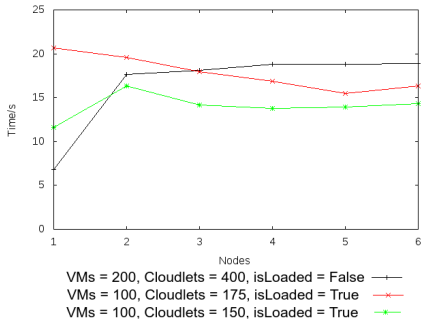

Fig. 6. Distributed Execution - Different Patterns of Scaling

Coordination-Heavy Case (Negative Scalability): Simulation time is increasing with the number of nodes, for the case of (noOfVMs = 200, noOfCloudlets = 400, isLoaded = false). This is because the cloudlets are not doing any task or no load attached to each cloudlet to perform. Hence, Hazelcast integration imposes an overhead consisting of coordination and other fixed costs, for an application for which a sequential and centralized execution is good enough. As in the success case, the time is converging here as well. Introducing further nodes will not increase the time any more, after some number of nodes.

Common Case: Simulation time is decreasing with the number of nodes steadily till a number of nodes, and then it starts to increase steadily, for the case of (noOfVMs = 100, noOfCloudlets = 175, isLoaded = true). This is one of the commonest cases, where a memory-hungry application that can hang (infinitely long time) in a single node, runs faster (10x speedup) in 2 nodes and also in 3 nodes, where further nodes may decrease the performance, due to the coordination and communication costs. In this particular example, 5 nodes was the ideal scenario and introducing the 6th node created a negative scalability. Here the communication and serialization costs start to dominate the benefits of the scalability at latter stages.

Complex Case (Weird Patterns and borderline cases): Scenario (noOfVMs = 100, noOfCloudlets = 150, isLoaded = true) initially shows a negative scalability, followed by a positive scalability and then by a negative scalability again. Through repeating different experiments, a pattern was noticed in this rarely occurring scenario. Initially, introducing Hazelcast causes an overhead over the performance enhancements it provides, hence increasing the execution time. Then, the

application starts to have the advantages of distribution and enhanced scalability, when the speedup due to distribution dominates over the initial overheads of distribution, specially the serialization and initialization costs of Hazelcast. Later, communication costs tend to overtake the advantages of the distribution, causing negative scalability again.

Among all the cases, there was a pattern, and it was possible to predict the changing scalability pattern, based on the curves for the other number of cloudlets and VMs combinations, given that the application remained unchanged.

C. MapReduce Implementations

Hazelcast-based and Infinispan-based MapReduce simulator word count implementations were benchmarked against multiple big files of 6 - 8 MB, each consisting of more than 125,000 lines, having the full size up to 9.4 GB. Both implementations were observed to distribute the job uniformly across all the instances in the execution cluster. Figure 7 represents the time taken for both implementations on a single server with 3 map() invocations, along with the increasing number of reduce invocations with the size. Here the size is measured by the number of lines taken into consideration for the MapReduce task.

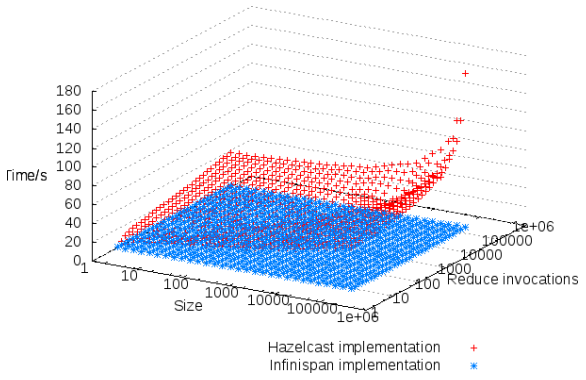

Fig. 7. Reduce invocations and time taken for different sizes of MapReduce tasks

The results showed Infinispan outperforming Hazelcast by 10 to 100 folds. Infinispan based simulator was still fast, even when operating verbose, where each of the execution step is monitored and logged. Infinispan MapReduce implementation is matured. Hazelcast MapReduce implementation is young, and still could be inefficient. Infinispan performs well in a single-node mode, as it operates better as a local cache. Hazelcast is optimized for larger set ups with very high number of real server nodes, and probably Hazelcast could outperform Infinispan, when larger number of nodes (such as 50) are involved. MapReduce executions are easily parallel and distributed by nature, even in a single node. This is not the case for general applications like CloudSim simulations. Infinispan model is perfect for a single node (or even a few node) MapReduce tasks.

1) ***Infinispan MapReduce Implementation****:* Infinispan implementation was tested for its scalability, with the same MapReduce job distributed to different number of nodes. Figure 8 shows the scaling of Infinispan MapReduce implementation to multiple nodes, with the time taken to execute different number of map() invocations. Number of reduce() invocations was kept constant at 159,069. Number of map() invocations is equal to the number of files present in the word count execution used. Hence, the number of files were increased for different scenarios. As the number of instances was increased, the jobs were distributed to the available instances.

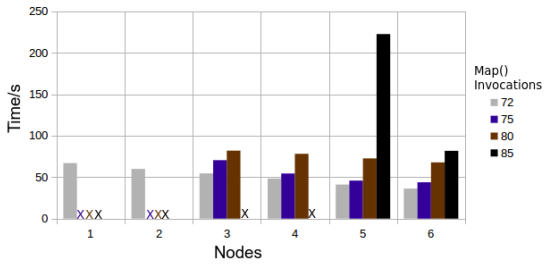

Fig. 8. Distributing the Infinispan MapReduce execution to multiple nodes

When the number of map() invocations was increased, jobs started to fail in single instance, due to the out of memory (java.lang.OutOfMemoryError: Java heap space) issue. Further, garbage collection (GC) overhead limit was exceeded in some scenarios. These issues prevented larger invocations to execute in smaller number of instances. When the number of instances was increased, the jobs that failed started to execute successfully and a positive scalability was observed. These evaluations prove that memory and processing requirements increase as the number of map() and reduce() invocations are increased. Further, distributing the execution enables larger executions, and makes the executions faster.

2) ***Hazelcast MapReduce Implementation****:* As Hazelcast based MapReduce simulator was slow when run in a single mode, it was tested on 1 - 6 nodes in verbose mode to check the improvements in execution time. One node starts the MapReduce simulator, where other nodes start the *Initiator* class, which just connects to the cluster and executes the logic fractions sent by the master.

Time taken for different sizes of the task to run on different number of instances is shown by Figure 9. Number of map() invocations was kept constant at 3, while increasing the number of reduce() invocations. Infinispan with single node was noticed to be still faster than all 6 nodes running MapReduce in Hazelcast.

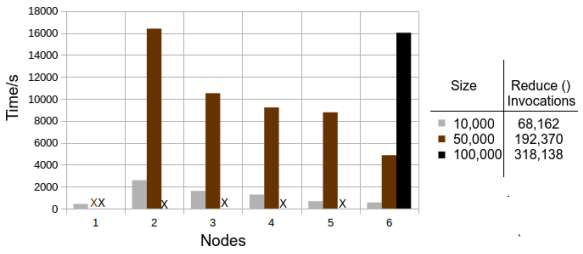

Fig. 9. Distributing the Hazelcast MapReduce execution to multiple nodes

For the size of 10,000 (68,162 reduce() invocations), Hazelcast running on a single instance was fast enough, and distributing the execution to multiple nodes started with a considerable negative scalability. This is because the communication and

TABLE IV. TIME (SEC) TAKEN FOR MULTIPLE HAZELCAST INSTANCES TO EXECUTE THE SAME TASK

No. of Hazelcast Instances	1	2	3	4
Time taken (sec)	416.687	2580.087	1600.655	1275.664
	6	**8**	**10**	**12**
	553.296	432.926	320.055	312.414

coordination costs were higher than the improvements from the distributions. However, positive scalability, though not significant, was achieved when more than 8 instances were used, as shown by Table IV. Up to 2 Hazelcast instances were executed from each of the nodes during this. This shows that even for smaller applications, distribution may be advantageous overtaking the communication and other costs introduced by distributing the execution.

The sample application failed to run on single node for the size of 50,000 (192,370 reduce() invocations), due to the heap space limitations. It ran smoothly on 2 instances, and showed a perfect positive scalability, when the nodes were joined to the cluster up to 6. The application failed to run on a single node for the size of 100,000 (318,138 reduce() invocations), due to the out of memory issue in heap space. The issue persists even when the cluster size was increased up to 5 nodes. The application ran successfully only when 6 nodes were involved. The last two cases clearly show the requirement of distributed MapReduce simulations for larger tasks, as a single or a fewer nodes in the cluster were proven to be insufficient for the higher memory requirements of the MapReduce tasks.

VI. CONCLUSION AND FUTURE WORK

Typically, cloud and MapReduce simulators are sequential, and thus run on a single computer, where computer clusters and in-memory data grids can be leveraged to execute larger simulations that cannot be executed on a single computer. Even the simulations that can run on a single node can take advantage of more resources from the cluster, that it can run faster and more effectively. The cycle sharing model can be utilized to provide means of sharing the resources across the simulation instances, allowing multiple independent simulations to execute in parallel, in a multi-tenanted manner.

$Cloud^2Sim$ presents an architecture that enables the execution of larger simulations in a cluster, that cannot be run on single nodes due to the requirement of huge heap space, and long execution times. $Cloud^2Sim$ has the advantages of CloudSim while being efficient, faster, customizable, and scalable. MapReduce implementations stand as an extension and proof that the same distributed execution model can be extended beyond cloud simulations. By virtue of being elastic and adaptive, it is cloud-ready and can be the basis of a truly concurrent and distributed Simulation-as-a-Service for Cloud and MapReduce simulations.

Acknowledgments: This work was partially supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013.

REFERENCES

[1] Mat Johns. *Getting Started with Hazelcast*. Packt Publishing Ltd, 2013.

[2] Francesco Marchioni. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[3] Seovic, A., Falco, M., & Peralta, P. (2010). Oracle Coherence 3.5. Packt Publishing Ltd.

[4] Marco Ferrante. A java framework for high-level distributed scientific programming. 2003.

[5] Rodrigo N Calheiros, Rajiv Ranjan, César AF De Rose, and Rajkumar Buyya. Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. *arXiv preprint arXiv:0903.2525*, 2009.

[6] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[7] Rodrigo N Calheiros, Marco AS Netto, César AF De Rose, and Rajkumar Buyya. Emusim: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. *Software: Practice and Experience*, 43(5):595–612, 2013.

[8] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 430–437. IEEE, 2001.

[9] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126–131. IEEE, 2008.

[10] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84. IEEE, 2007.

[11] Alberto Montresor and Márk Jelasity. Peersim: A scalable p2p simulator. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 99–100. IEEE, 2009.

[12] Bhathiya Wickremasinghe, Rodrigo N Calheiros, and Rajkumar Buyya. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 446–452. IEEE, 2010.

[13] Weiwei Chen and Ewa Deelman. Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–8. IEEE, 2012.

[14] Saurabh Kumar Garg and Rajkumar Buyya. Networkcloudsim: Modelling parallel applications in cloud simulations. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 105–113. IEEE, 2011.

[15] Pedro Velho and Arnaud Legrand. Accuracy study and improvement of network simulation in the simgrid framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, page 13. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[16] Beloglazov, A., Abawajy, J., & Buyya, R. (2012). Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. Future Generation Computer Systems, 28(5), 755-768.

[17] Siddhartha Kumar Khaitan and Anshul Gupta. *High Performance Computing in Power and Energy Systems*. Springer, 2012.

[18] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Play it again, simmr! In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 253–261. IEEE, 2011.

[19] Guanying Wang, Ali R Butt, Prashant Pandey, and Karan Gupta. Using realistic simulation for performance analysis of mapreduce setups. In *Proceedings of the 1st ACM workshop on Large-Scale system and application performance*, pages 19–26. ACM, 2009.

[20] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[21] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: a mapreduce simulator in enabling cloud computing. *Future Generation Computer Systems*, 29(1):300–308, 2013.

[22] Wagner Kolberg, Pedro De B Marcos, Julio Anjos, Alexandre KS Miyazaki, Claudio R Geyer, and Luciana B Arantes. Mrsg–a mapreduce simulator over simgrid. *Parallel Computing*, 39(4):233–244, 2013.

[23] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 43–56. ACM, 2012.

[24] Vishal Pachori, Gunjan Ansari, and Neha Chaudhary. Improved performance of advance encryption standard using parallel computing. *International Journal of Engineering Research and Applications (IJERA)*, 2(1):967–971, 2012.

[25] Roberto Palmieri, Pierangelo di Sanzo, Francesco Quaglia, Paolo Romano, Sebastiano Peluso, and Diego Didona. Integrated monitoring of infrastructures and applications in cloud environments. In *Euro-Par 2011: Parallel Processing Workshops*, pages 45–53. Springer, 2012.

[26] Liliana Rosa, Luís Rodrigues, and Antónia Lopes. Goal-oriented self-management of in-memory distributed data grid platforms. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 587–591. IEEE, 2011.