CloudKon DTS

Distributed Task Scheduling in Cloud

Rajagopal Parthasarathi

Department of Computer Science

Illinois Institute of Technology

Chicago, IL USA

rparthas@hawk.iit.edu

Kamal Nath N.G. Shenoi

Department of Computer Science

Illinois Institute of Technology

Chicago, IL USA

kninthas@hawk.iit.edu

**ABSTRACT**

Predictions are that by the end of this decade, we will have exascale system with millions of nodes and billions of threads of execution [1]. Task Scheduling and execution of tasks over these large scale, distributed systems plays an important role in achieving good performance and high system utilization. Many Task computing paradigm[2] aims to bridge the gap between High Performane Computing and High Throughput Computing . Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or [data-intensive](http://en.wikipedia.org/wiki/Data-intensive_computing) but MTC tasks include loosely coupled communication intensive tasks. Today’s jobschedulers have centralized Master/Slaves architecture (e.g. Slurm, Condor, PBS,SGE), where a centralized server is in charge of the resource provisioning and job execution. This architecture has worked well in modest scales and coarse granular workloads, but it has poor scalability at the extreme scales of petascale systems with fine granular MTC workloads. The goal of this project is to provide a efficient light weight and scalable distributed execution framework built on built on open source stack[HazelCast,Cassandra] to address MTC workloads deployed over Amazon Ec2 instance in cloud environment

**Index Terms**— Cloud Computing, Many Task Computing, Distributed Scheduling, Task Execution

# INTRODUCTION

The goal of an execution fabric is to effectively utilize the execution system aiming towards high throughput and provide efficient results for executed tasks. Today’s workload involves many fine granular workloads with execution times in seconds. Centralized schedulers are optimized towards high computational massive tasks where the complex decision policy and architecture of the schedulers play a major role. However, they tend to add considerable overhead while scheduling these lots of small tasks. Moreover, the centralized architecture tends to be a bottleneck in scheduling and execution. The solution to this problem is to have a decentralized and simple architecture. A decentralized architecture avoids the single point of failure, while a simple architecture reduces the considerable overhead involved in decision making for scheduling.

An execution fabric requires lot of computing resources to address the ever-growing workload of today’s world. Clouds seem to be a viable solution to this problem. Our solution is to build a loosely coupled compact and distributed execution fabric over public cloud (Amazon Ec2 instance) with distributed building blocks such as Cassandra [3] and HazelCast [4]. The motivation behind using open source stack in favor of extensive use of AWS [Amazon Web Services] is to decouple the fabric from AWS and provide easier transition to private cloud environment.

Recent studies suggest that clouds were not suitable candidates for scientific HPC computing [5].The problems listed were largely because of following the same approach involved in traditional clusters and grids. Clouds differ a lot from HPC applications as they are based on virtualization and shared resources.Our work involves running applications optimized for cloud environment. Traditional workloads can also be run on our execution fabric but with suitable decomposition of the workload at the client side.

In this project we implement a scalable distributed task execution framework. We have made extensive use of HazelCast , a highly scalable data distribution platform which acts as the reliable storage for the executable tasks . Our next building block is Cassandra, a distributed NoSQL store offering scalability and high availability for monitoring of the entire system. We also leverage the Amazon Elastic Compute Cloud (EC2) to manage virtual resources.

Today’s data analytics are moving towards shorter jobs with higher throughput and shorter latency. More applications are moving towards running higher number of jobs in order to improve the application throughput and performance. The focus is shifting towards Many Task Computing paradigm. Many task computing includes loosely coupled applications that are generally communication intensive.

We propose CloudKon as a job management system that achieves good load balancing and high system utilization. The heart of the CloudKon is the distributed queuing service. We have used HazelCast to facilitate this purpose. HazelCast performs the role of a highly available and reliable distributed pool of tasks to perfection. Worker Nodes are not adminstered by a centralized dispatcher or scheduler. Our work proposes an efficient pull architecture i.e worker nodes pull the tasks from the pool if they are idle. The system is loosely coupled and each component can be scaled based on the needs.

The driving factors behind our implementation of CloudKon are:

1. Design and architect a light-weight task execution framework for MTC workloads
2. Design a simple execution framework with a no frills user interface
3. Design a robust framework which can easily switch between public and private cloud environments
4. Design an extremely scalable execution framework
5. Design a loosely coupled framework to support future enhancements
6. Evaluate CloudKon with other state-of-the-art task execution systems
7. Deliver excellent throughput with <5% codebase of the job management systems

The remaining sections of this paper are as follows. Section 2 provides more background about the systems and the concepts that are related to this project and are necessary to know about. Section 3 discusses about the design and implementation details of CloudKon. Section 4 evaluates the performance of the CloudKon in different aspects using differentmetrics. Section 5 studies the related work in the area of task execution systems. Finally section 6 discusses about the limitations of the current work, and covers the future directions of this work.

# Background

## Amazon Elastic Compute Cloud (EC2)

Cloud computing services are broadly categorized into three layers

* Infrastructure-as-a-Service (IaaS)
* Platform-as-a-Service (PaaS)
* Software-as-a-Service (SaaS)

The focus of this project is on IaaS since the scientific computing community mostly focuses on IaaS because of the need for compatibility with legacy applications and systems. Amazon Elastic Compute Cloud (Amazon EC2) [6] is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. Amazon EC2 presents a true cloud hosting service for users. It allows users to use web service interfaces to launch instances with a variety of operating systems, load them with custom application environment, manage network’s access permissions, and run the image using as many systems as desired.

Amazon uses XEN hypervisor [7] as a middleware to run multiple Virtual Machines on their physical infrastructure. EC2 provides a web service that allows anyone to run their own applications on Amazon’s computing infrastructure, by letting customers “rent” computing resources by the hour. Clients are given access to an “unlimited” source of compute capacity, which is delivered through what is known as EC2 instance. An instance is a running virtual machine on Amazon’s cloud platform. Each of these instances are deployed with an Amazon Machine Image (AMI), which is just a pre-configured operating system and some bundled application software.

There exist several types of instances, each of them with different compute capacities, memory size, I/O performance and storage. Users launch one or more instances by specifying the instance type. Then the instances will be deployed on the server and user can connect to them via SSH using their public IP address. Amazon guarantees the availability rate of 99.95% in its Service Level Agreement. That means the instances are guaranteed to be available 99.95% of the time.

Considering the ways we can have access to these instances, we can categorize them in three different types:

* **Reserved instances:** Amazon allows us to pay upfront per each instance that we want to use during a given period, and in exchange, they give us a lower hourly cost for each of them. Along with the savings, with these instances we make sure that we will have availability through all the period that we paid for.
* **On demand instances:** these are the most common type of instances. You only pay for what you use, allowing easy allocation and deallocation of resources, depending on your capacity requirements. Customers are billed at the end of each month.
* **Spot instances:** This is a very interesting concept. In order to achieve a better utilization of their infrastructure, Amazon allows us to bid on unused EC2 capacity and run instances until the current spot instance price exceeds our bid. Amazon based on the available capacity sets the spot price and load of their systems and it is updated in a 5-minute period. The prices of these instances are much lower than what you pay for On-demand instances. As a drawback, the availability of you instances is only assured while the spot price is under bid. As previously stated, Amazon automatically terminates those instances whose bid is exceeded by the spot price. Besides, one cannot stop a spot instance and use it later as it happens with on-demand or reserved instances. Spot instances can only be terminated or rebooted.

Among these types, the spot instances seem to be the most appropriate for running short-term applications under certain conditions, since they provide the same capacity and features as the other instances at a lower rate. These include scientific applications, which usually run for a predictable amount of time, lowering the costs per experiment.

## Many Task Computing (MTC)

Many-Task Computing (MTC) was introduced by Raicu et al. [14][15] in 2008 to describe a class of applications that did not fit easily into the categories of traditional high-performance computing (HPC) or high-throughput computing (HTC). Many MTC applications are structured as graphs of discrete tasks, with explicit input and output dependencies forming the graph edges. In many cases, the data dependencies will be files that are written to and read from a file system shared between the compute resources; however, MTC does not exclude applications in which tasks communicate in other manners.

MTC applications often demand a short time to solution, may be communication intensive or data intensive, and may comprise of a large number of short tasks. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. For many applications, a graph of distinct tasks is a natural way to conceptualize the computation. Structuring an application in this way also gives increased flexibility. For example, it allows tasks to be run on multiple different supercomputers simultaneously; it simplifies failure recovery and allows the application to continue when nodes fail, if tasks write their results to persistent storage as they finish; and it permits the application to be tested and run on varying numbers of nodes without any rewriting or modification.

The hardware of current and future large-scale HPC systems, with their high degree of parallelism and support for intensive communication, is well suited for achieving low turnaround times with large, intensive MTC applications. The MTC paradigm has been defined and built with the scalability of tomorrow’s systems as a priority and can address many of the HPC shortcomings at extreme scales.

## HazelCast

HazelCast is a distributed in-memory data-grid that provides fast access to large amounts of data distributed across a cluster of machines. HazelCast allows you to easily share and partition your application data across your cluster. HazelCast is a peer-to-peer solution (there is no master node, every node is a peer) so there is no single point of failure. HazelCast is pure Java. JVMs that are running HazelCast will dynamically cluster. Although by default HazelCast will use multicast for discovery, it can also be configured to only use TCP/IP for environments where multicast is not available or preferred. Communication among cluster members is always TCP/IP with Java NIO beauty. Default configuration comes with a single backup so if one node fails, no data will be lost. It is as simple as using java.util.{Queue, Set, List, Map}

HazelCast’s Queue service is used as the queuing component in our implementation of CloudKon. It provides reliable and persistent data storage. HazelCast distributed queue is an implementation of java.util.concurrent.BlockingQueue HazelCast allows you to load and store the distributed queue entries from/to a persistent datastore such as relational database via a queue-store. If queue store is enabled then each entry added to queue will also be stored to configured queue store. When the number of items in queue exceed the memory limit, items will only persisted to queue store, they will not stored in queue memory. Here the queue store configuration options:

* Binary: By default, HazelCast stores queue items in serialized form in memory and before inserting into datastore deserializes them.
* Memory Limit: This is the number of items after which HazelCast will just store items to datastore. For example if memory limit is 1000, then 1001st item will be just put into datastore. This feature is useful when you want to avoid out-of-memory conditions.
* Bulk Load: At initialization of queue, items are loaded from QueueStore in bulks. Bulk load is the size these bulks

HazelCast is highly fault tolerant. It handles failure of nodes gracefully and the data can be replicated across the nodes by specifying the replication factor. HazelCast provides a flexible configuration model providing fine-grained control over each aspect.

After continuous evaluation for its fault tolerance and reliability, we chose HazelCast as the distributed queuing component for our execution fabric. The intent behind choosing HazelCast over Amazon’s distributed queue SQS are

* Fine Grained configuration control
* Support for complex types – object store[In SQS only string support]
* Control on resource allocation and pricing model[Abstraction in case of SQS]
* Flexibility to deploy on private clouds and clusters
* Follows FIFO policy – Message delivery is guaranteed to be in insertion order.
* In Memory Processing delivering efficient speeds
* Highly fault tolerant even in case of multiple node failures

## Cassandra

Apache Cassandra is a No SQL data Store with scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra's support for replicating across multiple datacenters is best in class, providing lower latency for users and the ability to withstand failures.

Cassandra's data model offers the convenience of column indexes with the performance of log-structured updates, strong support for denormalization and materialized views, and powerful built-in caching

Cassandra incorporates a number of architectural best practices that affect performance

* **Fully distributed**- Every Cassandra machine handles a proportionate share of every activity in the system. With each node the same, Cassandra is far simpler to install and operate. There are also no single points of failure and network bottlenecks.
* **Log-structured storage engine**- A log-structured engine that avoids overwrites to turn updates into sequential I/O is essential both on hard disks (HDD) and solid-state disks (SSD). On HDD, because the seek penalty is so high; on SSD, to avoid write amplification and disk failure.
* **Database Level Locking** - Cassandra uses advanced concurrent structures to provide row-level isolation without locking. Cassandra even eliminated the need for row-level locks for index updates.
* **Data Modification** - Cassandra’s storage engine only appends updated data, it never has to re-write or re-read existing data. Thus, updates to a Cassandra row or partition stay fast as your dataset grows.
* **Replication** – Control over synchronous or asynchronous replication for each update favoring performance over consistency. Highly available asynchronous operations are optimized with features like Hinted Handoff and Read Repair.
* **Scalability** - Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.
* **Fault** **Tolerant** - Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime

Cassandra provides an efficient storage mechanism to store the snapshot of the CloudKon execution environment. It provides a simplified interface for persistence of frequent snapshots of the system.

# Design and Implementation of CloudKon

## Normal or Body Text

Please use a 9-point Times Roman font, or other Roman font with serifs, as close as possible in appearance to Times Roman in which these guidelines have been set. The goal is to have a 9-point text, as you see here. Please use sans-serif or non-proportional fonts only for special purposes, such as distinguishing source code text. If Times Roman is not available, try the font named Computer Modern Roman. On a Macintosh, use the font named Times. Right margins should be justified, not ragged.

## Title and Authors

The title (Helvetica 18-point bold), authors' names (Helvetica 12-point) and affiliations (Helvetica 10-point) run across the full width of the page – one column wide. We also recommend phone number (Helvetica 10-point) and e-mail address (Helvetica 12-point). See the top of this page for three addresses. If only one address is needed, center all address text. For two addresses, use two centered tabs, and so on. For more than three authors, you may have to improvise.[[1]](#footnote-1)

## First Page Copyright Notice

Please leave 3.81 cm (1.5") of blank text box at the bottom of the left column of the first page for the copyright notice.

## Subsequent Pages

For pages other than the first page, start at the top of the page, and continue in double-column format. The two columns on the last page should be as close to equal length as possible.

Table . Table captions should be placed above the table

|  |  |  |  |
| --- | --- | --- | --- |
| **Graphics** | **Top** | **In-between** | **Bottom** |
| Tables | End | Last | First |
| Figures | Good | Similar | Very well |

# SECTIONS

The heading of a section should be in Times New Roman 12-point bold in all-capitals flush left with an additional 6-points of white space above the section head. Sections and subsequent sub- sections should be numbered and flush left. For a section head and a subsection head together (such as Section 3 and subsection 3.1), use no additional space above the subsection head.

## Subsections

The heading of subsections should be in Times New Roman 12-point bold with only the initial letters capitalized. (Note: For subsections and subsubsections, a word like *the* or *a* is not capitalized unless it is the first word of the header.)

### Subsubsections

The heading for subsubsections should be in Times New Roman 11-point italic with initial letters capitalized and 6-points of white space above the subsubsection head.

#### Subsubsections

The heading for subsubsections should be in Times New Roman 11-point italic with initial letters capitalized.

#### Subsubsections

The heading for subsubsections should be in Times New Roman 11-point italic with initial letters capitalized.

# REFERENCES

1. P. Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” 2008
2. IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)
3. HazelCast <http://www.hazelcast.com/>
4. Cassandra <http://cassandra.apache.org/>
5. Q. He, S. Zhou, B. Kobler, D. Duffy, and T.McGlynn. “Case study for running HPC applications in public clouds,” In Proc. of ACM Symposium on High Performance Distributed Computing, 2010.
6. Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, (<http://aws.amazon.com/ec2/>)
7. Xen Hypervisor, xen.org, [online] 2013, (<http://www.xen.org/products/xenhyp.html>)

1. [↑](#footnote-ref-1)