Scalable Execution of LIGO Astrophysics Application in a Cloud Computing Environment

Presented By

Dong Leng

Submitted in total fulfilment of the requirements of the degree of

Master of Engineering in Distributed Computing

The Cloud Computing and Distributed Systems (CLOUDS) Laboratory

The Department of Computer Science and Software Engineering

The University of Melbourne, Australia

Abstract

LIGO gravitational wave search application requires a system that can provide resources on demand. Platform-as-a-Service (PaaS) is such a solution that helps manage user applications in a scalable manner by using resources delivered through a virtualized Cloud data centre. It is a necessary characteristics for the PaaS layer to dynamically adjust the load on computing platforms based on the agreed quality of service (QoS) parameters with Cloud users. In order to make the platform services scalable, there is a need for a decentralized service design and autonomous provisioning algorithms in contrast to a centralized PaaS service design. In this report, we present a software design for LIGO application, which enables multiple PaaS services to run on decentralized PaaS layer that can manage dynamic load arising from application services running on top of it. There are two core algorithms in this system design which automate the load-balancing at multiple layers of the Cloud software stack. 1) a dynamic resource provisioning algorithm that facilitates the ‘scaling out’ and ‘scaling in’ of the compute resources needed for hosting multiple PaaS services at the middleware level; 2) a provisioning algorithm that instantiates compute resources at the infrastructure level required for running user applications. We also describe the detailed functionality of different components of LIGO Gravitational Wave search system. Through large amount of experiments, we analyse the challenges when executing the application and demonstrate the feasibility of our scalable platform design using Amazon Cloud services.

Acknowledgements

A large project such as this is impossible with only the power of individual. It requires the coordination across different group with different people. Many great ideas are generated within the discussion among researchers and staffs. I am only a member of professional research team. There are many people who gave me valuable suggestions and help.

Firstly, I would like to thank my coordinator Professor Rajkumar Buyya, who is the Professor of Computer Science and Software Engineering and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He provided me the opportunity to work on such important project and offered me access to the necessary resources to conduct the research and experiments. Also, he provided supervision and guidance from the top level.

Secondly, I would like to thank my supervisor Dr Suraj Pandey, who is a Research Fellow at the Cloud Computing and Distributed Systems Laboratory (CLOUDS) of the Computer Science & Software Engineering department, at the University of Melbourne, Australia. He guided me on the project from very beginning design to the finalization of the paper. Not only when drawing the big picture of the project, but also implementing the detailed part of program, he is keeping helping me all the way through. He is an expert in both academic research and software development. The discussion with him enlightens me on many perspectives of the project. I highly appreciate all the help he provide.

Lastly, I would like to thank all the researchers of the Cloud Computing and Distributed Systems Laboratory, the Department of Computer Science and Software Engineering and Physics Department of Melbourne University. We thank Amazon for providing us an educational grant that enabled us to experiment using EC2 and S3 Cloud resources.

Table of Contents

Figures

1. Introduction

1.1 Project Background

Gravitational waves (GW) are ripples thought to occur in the fabric of space-time that result from interstellar collisions, explosions, or movement of large and extremely dense objects such as neutron stars. Those ripples can then pass through the space-time that Earth occupies, causing a distortion which Advanced LIGO is meant to pick up. Currently, several interferometric Gravitational Wave detectors around the world such as LIGO, VIRGO, GEO600, TAMA300 have been collecting data that could then be used by scientists for searching GWs [3].

There are numerous search technologies for GW data known to the community. There are many potential sources of GW, of three main types (stochastic, burst and continuous). Each search for a specific type of source will need to be performed over the certain parameter space, with an optimal balance required between parameter space mismatch and computational resources. Most of these searches can be represented as a workflow consisting of tasks linked through data dependencies. Each workflow can then be replicated using different parameter set. Numerous scientists could then use these multiple workflows for analyzing and searching GWs [3]. Without support for scheduling and management of data and tasks, in the worst case, the parallel execution of these multiple workflows will be reduced to sequential execution due to contention of common Cloud resource.

Currently, users of LIGO data, such as Ms. Sammut, are bound to submit their search procedures using Condor DAG scripts to compute resources located primarily in Germany and the USA that are managed by Condor. Once the Australian Data Center is established, data generated locally resides in the Australian Data Center, where the computations can be carried out. The main impact of the solution (scalable executions of multiple workflows) will be to design from the ground up a user- centric, optimally configured Data Center, in preparation for the day when the data is accessible to a wider community of users (e.g. traditional electromagnetic astronomers).

A demonstration of a working prototype that dynamically provisions middleware components for load balancing purposes and uses virtualized memory for efficient I/O will show the extensibility and usefulness of virtualized Data Centers (e.g. Cloud data centers provided by Amazon, GoGrid, etc) in terms of cost savings and resource utilizations. From Physics point of view, a successful detection of a source like Sco X-1 will help to determine the origin of the GW quadrupole. A nondetection will yield upper or lower limits on many of the quantities still under investigation [6]. Making this possible with ease, efficiency, and scale will help reach a long awaited breakthrough earlier than most people anticipate.

1.2 Aim

Leverage the strengths of cloud computing for computation intensive and storage intensive applications, use computer resources in a scalable manner, balance the load of resources dynamically according to their capabilities, maximise the performance and efficiency.

1.3 Objectives

The problem of executing multiple workflows for multiple users is challenging. Parallel executions of workflows could lead to resource contention, as each workflow instance requires same set of data as input, specific number of compute resources and are bound by deadlines set by users. Thus, the challenging tasks are:

1. Allocate Cloud resources to tasks, workflows and users effectively to avoid resource contention dynamic resource provisioning problem

2. Minimize delay in executing workflows task/workflow/users scheduling problem

3. Dynamically balance the load of different resources

In this project, we accomplish all the goals above.

1.4 Motivation

The main motivation of this project is to solve the issues with huge computation of Gravitational Wave search, which are encountered by the Physicist who use LIGO applicaion. It provide well structured solution for the coming LIGO data centre in Australia. We also demonstrate it at the Fourth IEEE International Scalable Computing Challenge (SCALE 2011), held in conjunction with CCGrid 2011 conference in Newport Beach, CA.

2. Technology Review

2.1 Middleware - Workflow Engine

We describe a process of constructing and experimenting with the workflow on the Grid and cloud computing environment. We then present the components of Cloudbus Workflow Management System (GWMS).

2.1.1 Workflow Deployment Cycle

In the input phase, scientists provide the input data, batch scripts, sample output files, and application requirements. In order to run the application on the Grid with heterogeneous resources, the executables are required to be compiled and installed (can be submitted at runtime) at both remote and local sites. For quality assurance or initial results verification, this step involves the testing of conformance of our execution with that of the sample results provided by the scientists. Once the initial results are verified the workflow structure and its details need to be composed in the designing phase. In this phase, the automated generation of the workflow in terms of the workflow language used can also be done by taking into account the run-time parameters that users might want to change during execution [7].

In the Execution phase the resources, where the application can be executed, need to be setup. The resource list, its credentials and the services provided by each resource need to be inserted into the catalogue. When experiments are conducted repeatedly and in time, resource availability and conditions will have changed. This requires services and credentials list to be generated for each execution with the help of the catalogue. The application is then executed on the Grid. Usually debugging and testing is done while the application is being executed, but this depends on the software development process being used. Depending on the analysis of the results from the output phase, the workflow design can be further optimized according to the requirements of the user. These are the basic steps involved in constructing most of the scientific workflows to be executed on the Grid, but doesn't generalize all applications. The process can get complicated when more user and system requirements are added [7].

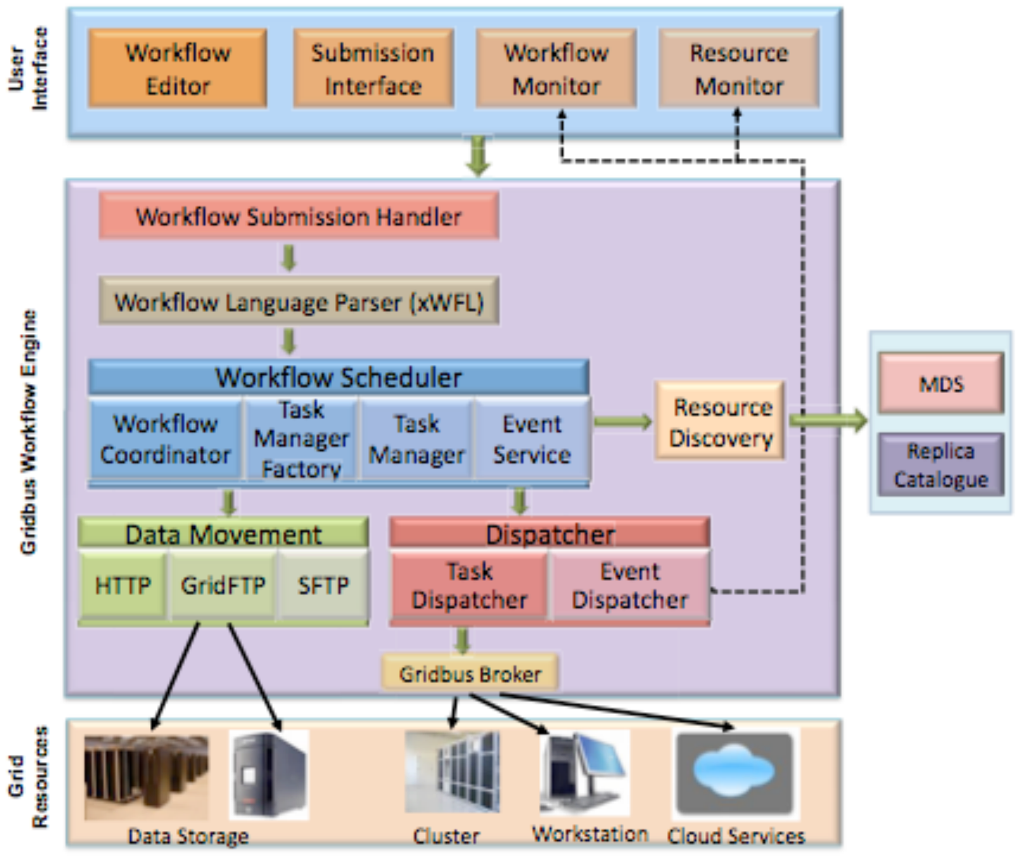


Figure 1 Components of Workflow Engine

2.2 Infrastructure - Amazon EC2 Cloud

Amazon Elastic Compute Cloud (Amazon EC2) 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's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Amazon EC2 provides developers the tools to build failure resilient applications and isolate themselves from common failure scenarios. [9]

2.2.1 Amazon EC2 Functionality

Amazon EC2 presents a true virtual computing environment, allowing you to use web service interfaces to launch instances with a variety of operating systems, load them with your custom application environment, manage your network's access permissions, and run your image using as many or few systems as you desire [9].

To use Amazon EC2, you simply:

• Select a pre-configured, templated image to get up and running immediately. Or create an Amazon Machine Image (AMI) containing your applications, libraries, data, and associated configuration settings.

• Configure security and network access on your Amazon EC2 instance.

• Choose which instance type(s) and operating system you want, then start, terminate, and monitor as many instances of your AMI as needed, using the web service APIs or the variety of management tools provided.

• Determine whether you want to run in multiple locations, utilize static IP endpoints, or attach persistent block storage to your instances.

• Pay only for the resources that you actually consume, like instance-hours or data transfer.

2.3 Application – LIGO (Laser Interferometer Gravitational-Wave Observatory)

2.3.1 Gravitational Waves: Ripples in the Fabric of Space-Time

Albert Einstein predicted the existence of gravitational waves in 1916 as part of the theory of general relativity. He described space and time as different aspects of reality in which matter and energy are ultimately the same. Space-time can be thought of as a "fabric" defined by the measuring of distances by rulers and the measuring of time by clocks. The presence of large amounts of mass or energy distorts space-time -- in essence causing the fabric to "warp" -- and we observe this as gravity. Freely falling objects -- whether a soccer ball, a satellite, or a beam of starlight -- simply follow the most direct path in this curved space-time [8].

When large masses move suddenly, some of this space-time curvature ripples outward, spreading in much the way ripples do the surface of an agitated pond. Imagine two neutron stars orbiting each other. A neutron star is the burned-out core often left behind after a star explodes. It is an incredibly dense object that can carry as much mass as a star like our sun, in a sphere only a few miles wide. When two such dense objects orbit each other, space-time is stirred by their motion, and gravitational energy ripples throughout the universe.

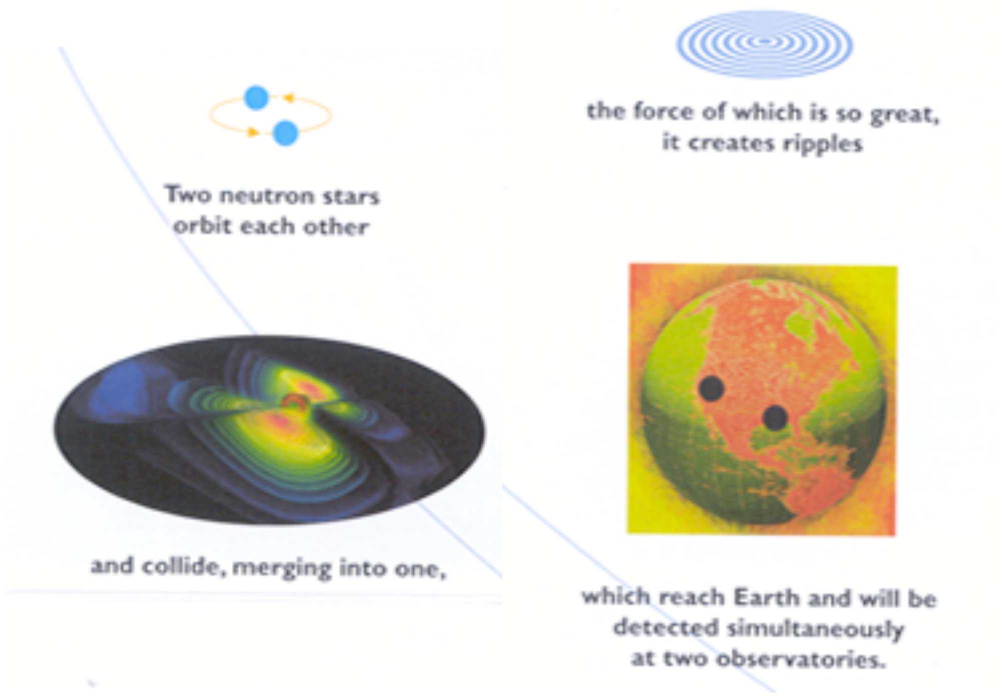


Figure 2 Gravitational Waves and LIGO

In 1974 Joseph Taylor and Russell Hulse found such a pair of neutron stars in our own galaxy. One of the stars is a pulsar, meaning it beams regular pulses of radio waves toward Earth. Taylor and his colleagues were able to use these radio pulses, like the ticks of a very precise clock, to study the orbiting of neutron stars. Over two decades, these scientists watched for and found the tell-tale shift in timing of these pulses, which indicated a loss of energy from the orbiting stars -- energy that had been carried away as gravitational waves. The result was just as Einstein's theory predicted.

2.3.2 How LIGO Works

LIGO will detect the ripples in space-time by using a device called a laser interferometer, in which the time it takes light to travel between suspended mirrors is measured with high precision using controlled laser light. Two mirrors hang far apart, forming one "arm" of the interferometer, and two more mirrors make a second arm perpendicular to the first. Viewed from above, the two arms form an L shape. Laser light enters the arms through a beam splitter located at the corner of the L, dividing the light between the arms. The light is allowed to bounce between the mirrors repeatedly before it returns to the beam splitter. If the two arms have identical lengths, then interference between the light beams returning to the beam splitter will direct all of the light back toward the laser. But if there is any difference between the lengths of the two arms, some light will travel to where it can be recorded by a photo detector [8].

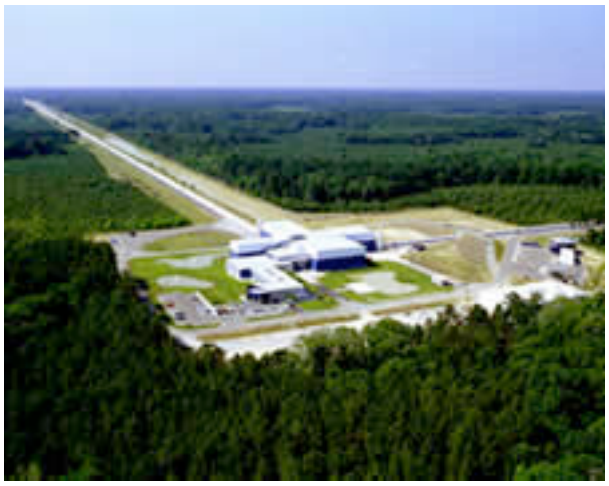


Figure 3 Livingston Observatory Livingston, Louisiana

The space-time ripples cause the distance measured by a light beam to change as the gravitational wave passes by, and the amount of light falling on the photo detector to vary. The photo detector then produces a signal defining how the light falling on it changes over time. The laser interferometer is like a microphone that converts gravitational waves into electrical signals. Three interferometers of this kind were built for LIGO -- two near Richland, Washington, and the other near Baton Rouge. Louisiana. LIGO requires at least two widely separated detectors, operated in unison, to rule out false signals and confirm that a gravitational wave has passed through the earth.

3. Design

3.1 System Architecture

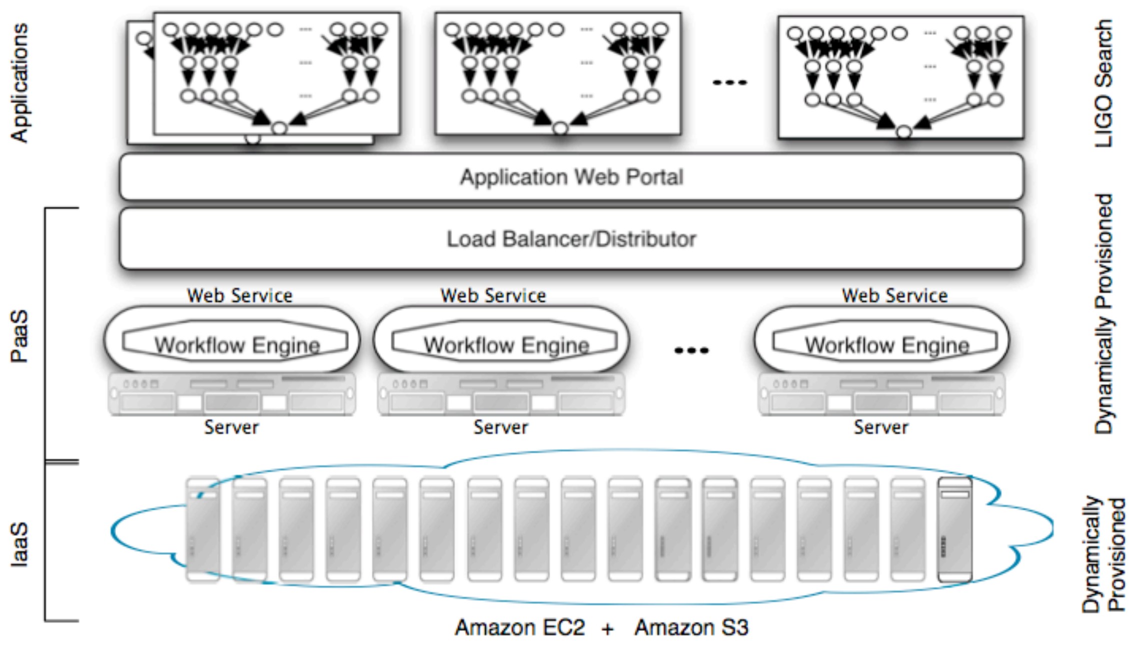


Figure 4 System Architecture for Autonomic Scaling of PaaS Services

We propose a layered design that can process multiple users and their workflows in a scalable manner, as depicted in Figure 4. At the bottom most layer, we use virtualized resources provided by various Cloud service providers (e.g. Amazon EC2, GoGrid, Azure etc). We explore a novel way of managing commonly used data across all VM instances by using application level Virtualized Memory, which allows applications on multiple servers to share data without replication, decreasing total memory need. At the middleware level, we use our existing middleware solutions: Workflow Engine [5] for managing application workflow at task level, and Aneka [5] for managing jobs (one workflow block as depicted in Figure 3, and/or tasks) that are submitted by the workflow engine

In order to scale-out/scale-in the middleware components, we will design a load balancer/distributor algorithm that dynamically instantiates workflow engines (each running on a separate VM) based on: 1) User priority, 2) Number of waiting jobs (user requests coming to the server request level parallelism), and 3) Average completion time of workflows submitted to the virtualized resources. With the coordination of middleware component provisioning and compute resource provisioning at that infrastructure level, we propose to scale out and scale in the resources to manage multiple workflows submitted by large number of users across Cloud data centres.

3.2 LIGO Gravitational Wave Search System Design

Based on the system architecture described in the Section 3.1, we design the system for LIGO Gravitational Wave Search. Figure 5 shows the entities of LIGO Gravitational Wave Search System and their interactions. We select Amazon EC2 as cloud resource provider. The whole system is deployed to Amazon EC2 to gain better performance and lower latency. There are four main components in this system, web portal, load balancer, engine and worker. Web portal takes the input of users and provides the monitor of running tasks and results. Load balancer takes the responsibility of distributing tasks and instantiating computer resources dynamically. This is done in an autonomic manner based on the work load of the system. Engine is an middleware that used to schedule and assign tasks to different workers. Worker is the actual resource that executes the LIGO application. We will discuss each of these components with details in the following sections.

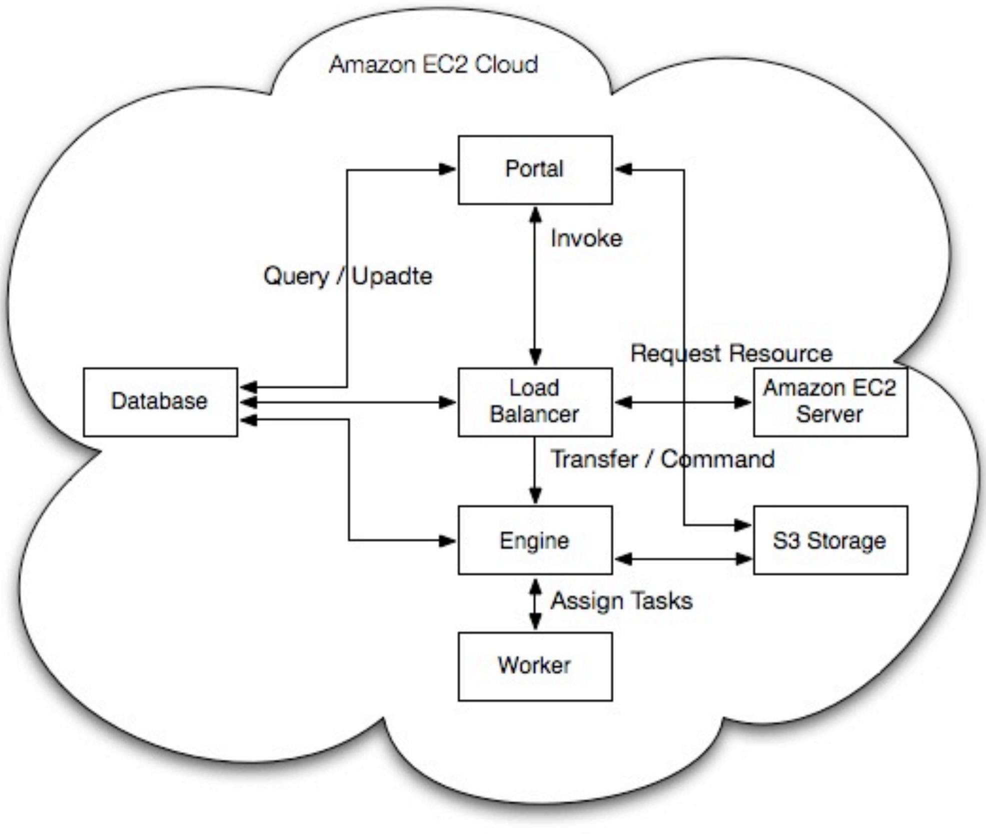


Figure 5 System Entities Relationship

3.3 Load Balancer

Load balancer is the core of the whole system. It manages all the computer resources from cloud data centre. The activities of load balancer show in Figure 6. Based on the amount of request, load balancer calculates the number of computer resources in need and sends the resource requests to cloud administrate server. Load balancer adjusts the capability of system in a scalable manner. Once more requests come in, load balancer increases the capability of system by starting more instances. On the other hand, once less tasks left, it decreases the capability of system by terminating instances. As all the tasks have been done, all the instances will be shut down.

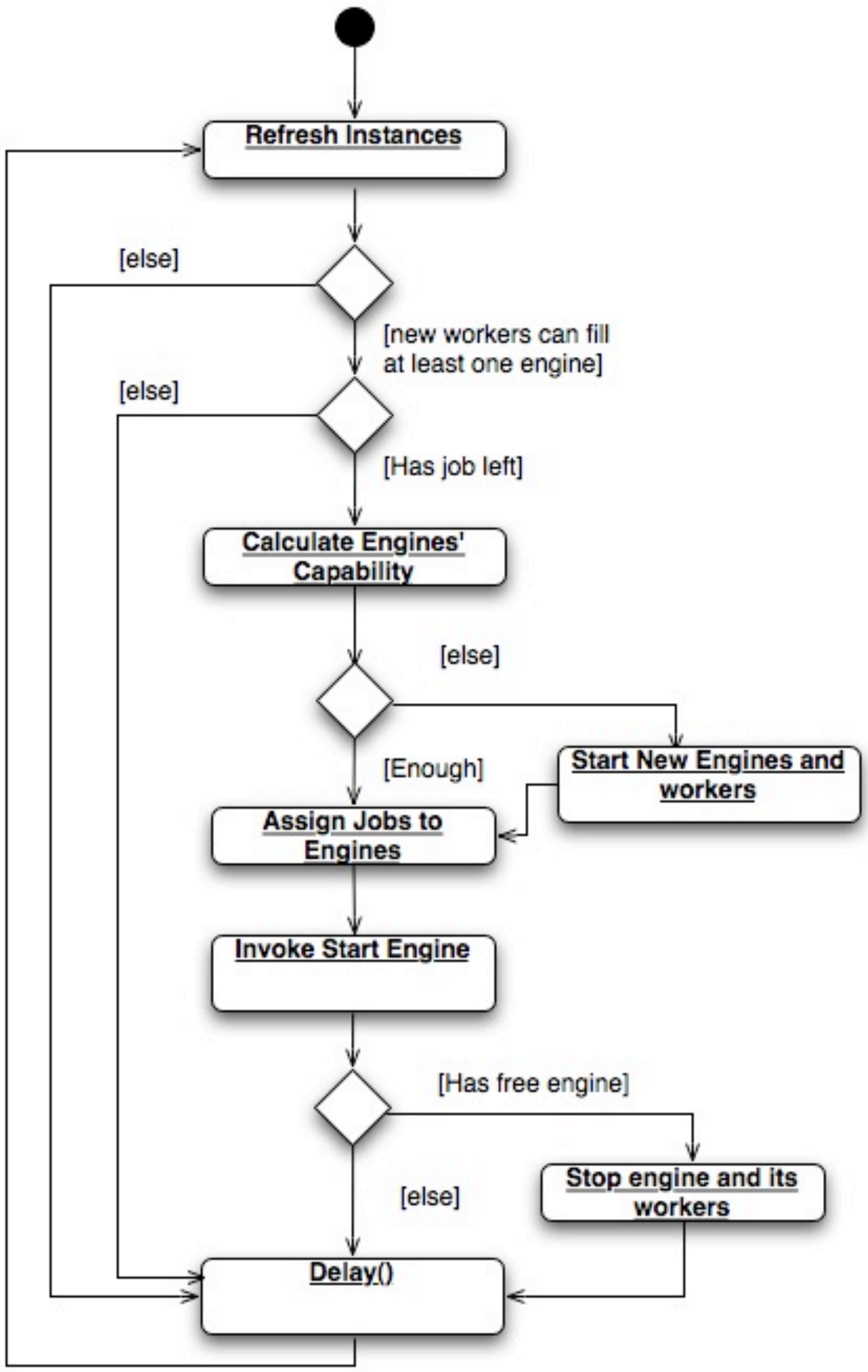
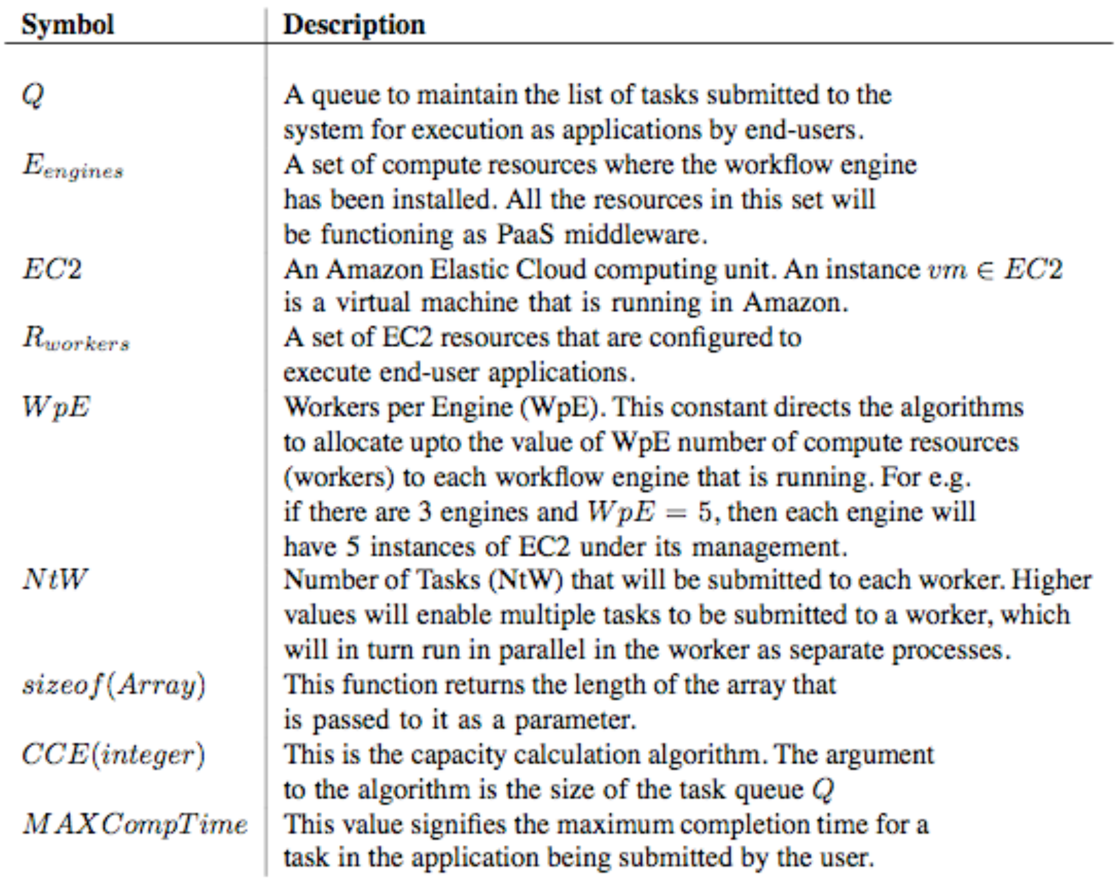


Figure 6 Load balancer Activity Diagram

3.3.1 Load Balancer Algorithm



Symbol	Description
Q	A queue to maintain the list of tasks submitted to the system for execution as applications by end-users.
$E_{engines}$	A set of compute resources where the workflow engine has been installed. All the resources in this set will be functioning as PaaS middleware.
$EC2$	An Amazon Elastic Cloud computing unit. An instance $vm \in EC2$ is a virtual machine that is running in Amazon.
$R_{workers}$	A set of EC2 resources that are configured to execute end-user applications.
WpE	Workers per Engine (WpE). This constant directs the algorithms to allocate upto the value of WpE number of compute resources (workers) to each workflow engine that is running. For e.g. if there are 3 engines and $WpE = 5$, then each engine will have 5 instances of EC2 under its management.
NtW	Number of Tasks (NtW) that will be submitted to each worker. Higher values will enable multiple tasks to be submitted to a worker, which will in turn run in parallel in the worker as separate processes.
$sizeof(Array)$	This function returns the length of the array that is passed to it as a parameter.
$CCE(integer)$	This is the capacity calculation algorithm. The argument to the algorithm is the size of the task queue Q
$MAXCompTime$	This value signifies the maximum completion time for a task in the application being submitted by the user.

Figure 7 Description of symbols used in the work

First put all the tasks need to be done into a queue called Q, refresh running computer resources, put new resources into array of engine E engines and queue of worker to assign R workers, respectively, according to the type and property of instances. Because it takes time to start computer resource required by the last loop, we wait until there are enough workers in W to fully filly empty space of one engine of E or waiting list of computer resources R is empty. Then we compute the capability of each engine of E with algorithm CCE. In other words, we measure how many new tasks the existed engines can handle during a reasonable time span. If existed engines is not enough to handle all the tasks in T, we compute how many more engines and workers we still need and start these computer resources. Next poll task from T and assign it to engines, each time we decrease the capability of that engine by 1. Repeat this step until there is no capability left for that engines, then trigger start application function of that engine to run broker for newly added tasks. Retrieval all the engines, till there is no capability left. After these, we go back to the beginning and repeat all the steps until all tasks have been finished.

Algorithm 1: PaaS load balancing algorithm

```
1  begin
2      Maintain a list of submitted tasks in queue $Q$
3      repeat
4          Refresh list of running computing resources
5          Divide newly added instances $vm \in EC2$ to set of engines $\{E_{engines}\}$ and workers
           $\{R_{workers}\}$
6          Associate up to $WpE$ amount of workers in $\{R_{workers}\}$ to each engine $e_i \in E_{engines}$
7          if ( (size of $R_{workers} >= 0$) OR all waiting compute resources available) $AND$
           (sizeof(Q) $> 0$) then
8              Number of tasks remaining to be submitted for execution
               $nPending = CCE(sizeof(Q))$
9          if $nPending > 0$ then
10             Workers to run $wr = nPending/NtW$
11             $wPending = wr$
12             for each Engine $e_i \in E_{engines}$ do
13                 Free slots for engine $e_i : es = WpE - (current\_number\_of\_workers\_in\_e_i)$
14                 $wPending = wPending - es$
15                 if $wPending > 0$ then
16                     Engines to run $er = wPending/WpE$ //Integer division
17                 Provision $er$ engines in the Cloud
18                 Provision $wr$ workers in the Cloud
19         for each Engine $e_i \in E_{engines}$ do
20             repeat
21                 Assign tasks in $Q$ to $e_i$
22             until current_engine reaches it maximum load
23             Start execution of tasks assigned to engine $e_i$
24     until all the submitted tasks have been finalized
25 end
```

Figure 8 PaaS load balancing algorithm

3.3.2 CCE Algorithm

First set the capability of each free worker to be cw, which means the maximum number of tasks can be allocated to the worker to ensure they will be done in a reasonable time. Set the threshold of completion time of a single task to be tt, this threshold is the standard to check whether the engine is overloaded or not. If it is overloaded, we stop assigning tasks to it. Otherwise, we assign more tasks to it. Then, for each engine of E, we compute the average completion time ti during the last n minutes, then we compare the average completion time ti with threshold tt to gain the percentage of availability of engine ei. The capability ci of the whole engine ei is its availability multiple by the number of worker of ei multiple by maximum number of tasks per worker cw. ci means how many more tasks can be assign to the engine ei. Then we assign the worker in the queue W to the free spaces of engine ei of zE, because these workers are newly joined free worker, we increase the capability of that engine ei by cw. Finally, we calculate how many tasks are still left after assigning to ei E according to the capability of ei and return it. Tasks left is useful for decide the number of new engines and new workers that are still needed to run.

```
Algorithm 2: CCE(number of tasks): Engine capacity/load calculating algorithm
1  begin
2      Set the capability of each free worker to be $NtW$
3      Set the threshold of completion time of a single task to be $MAXCompTime$
4      for each Engine $e_i \in E_{engines}$ do
5          Get average task completion time $ct_{ei}$ of $e_i$ during the last $n$ minutes
6          Compute the availability $a_i$ of $e_i$, where
           $a_i = (ct_{ei} - MAXCompTime)/MAXCompTime$
7          Compute the capability $c_i$ of $e_i$ according to percentage of availability, where
           $c_i = a_i * (numberofworkersofe_i) * (maxtasksperworker)$
8      Refresh compute resource status
9      if New workers are ready then
10         Increase the capability of its engine by $NtW$
11     $not\_scheduled\_tasks$ = Calculate tasks remaining to be assigned to a engine
12     return $not\_scheduled\_tasks$
13 end
```

Figure 9 CCE(number of tasks): Engine capacity/load calculating algorithm

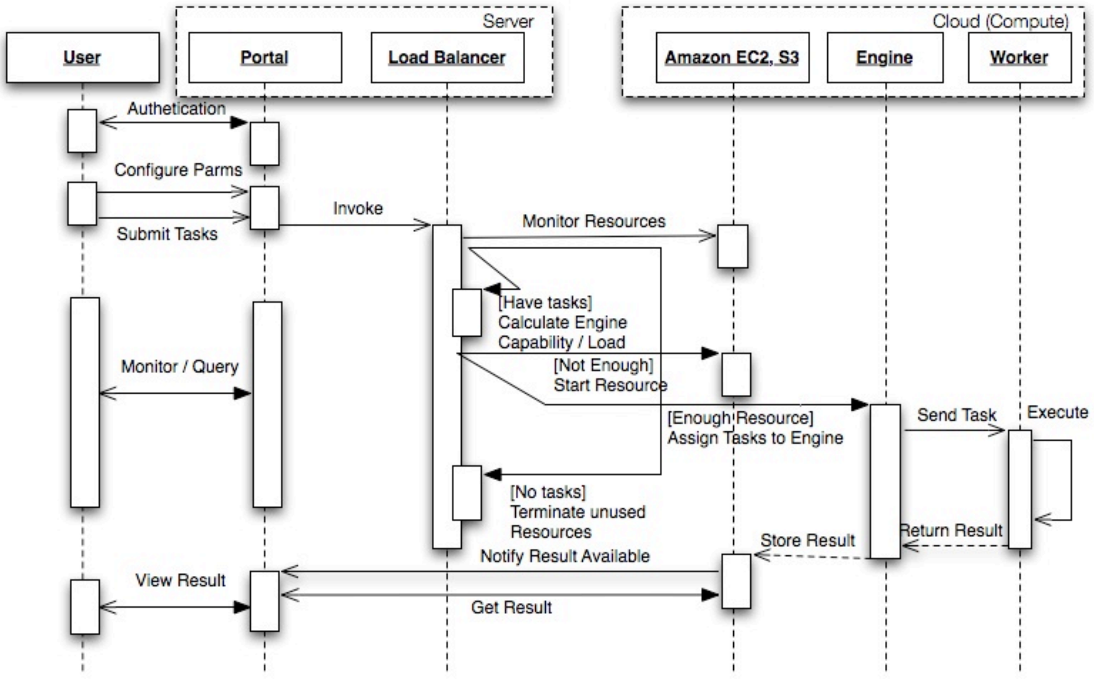


Figure 10 Load Balancing Sequence Diagram

3.4 Engine

There are two components in each engine. One is workflow engine, which is responsible for scheduling tasks and allocates tasks to worker. The other one is engine web service, which receives tasks and command from load balancing, based on SOAP web service.

3.4.1 Workflow Engine

Workflow engine manages the workers that belong to it and tasks been assign to it. It can monitor the status of workers and the execution of tasks dynamically. It was an workflow.

3.4.2 Engine Web Service

Once a engine starts running, first it receives the configuration files from load balancer, which contains necessary configuration parameters of workflow engine. Then task files including application file, service file and credential file will be received for computation. After all task files transfer successfully, a command of starting broker will be triggered. The broker retrieves all the received tasks and pushes them to the task queue, all the tasks will be assigned to different workers for execution. At the mean time, engines keep checking the status of workers and tasks.

3.5 Worker

The worker is the actual computer resource that executes LIGO Gravitational Search application. Once worker have receive task file from the engine, it start executing. After it finish the task, it will send result back to engine.

3.6 Portal

Users can manage all their tasks and cloud resources through web portal. All the operations are Ajax-driven. The container of portal is Tomcat server. JavaServer Faces and Richfaces are used for font-end implementation.

3.6.1 Property Setup

All the configuration of workflow, database and load balancing can be accessed from configuration section of portal.

3.6.2 Task Generator

Users first submit a signal source file with regular parameters, then set up the customized parameters. After set up all the parameters properly, use java internal xml property to generate task xml files, the number of files depends on the search band parameter.

3.6.3 Task Submission

For each source of each user, the task files are saved in a separate folder. When submitting tasks, the files of folder will be retrieved and push the tasks in a task queue. Tasks will be polled out from queue when load balancer can assign it to engine for execution.

3.6.4 Monitor

Portal enable users monitor all the tasks, computer resources and the relationship between them. Tasks are categorized by application, when user submit the application ID, all the tasks of that application will be listed in a table, users can monitor the progress of each task, such like the submission time, time consumption and the number of times task fails. Also, there is another table for describing tasks is executed by which worker.

4. Experiment

In order to fine tune our system, we conduct a large amount of experiment from different perspectives. Even through the process of experiment evolves many difficulties, such like the failure of application and Amazon cloud resources, we finally get most of expected data for analysing. Generally, there are two groups of experiment, one is a serious of experiment of different combination of engines, worker and sources. The other is a large scale of experiment for each source with 500 computing workers. In this group, we get gravitational wave search result which contains all the points.

4.1 Data Collection and Analysis

4.1.1 Different combination with the fixed number of sources

First run a series of experiment with the fixed number of sources and different combination of engines and workers. Limit the total number of tasks to be 40. To gain better understanding of practical environment, we try to use different source for each experiment. Set the number of engines to be 1, 2 and 4 respectively. Also, limit the number of workers to be 4, 8 and 16. Run through all possible 9 combinations altogether.

4.1.1.1 Engine 1

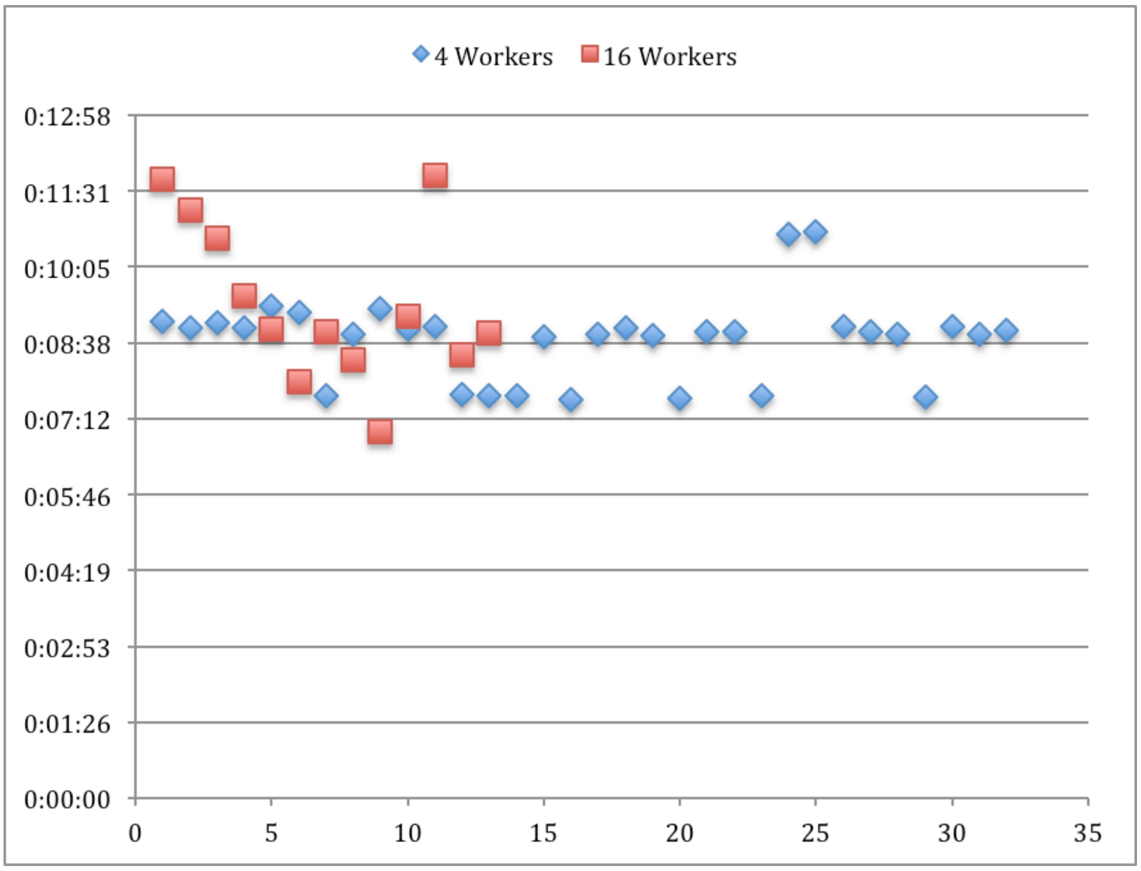


Figure 11 Engine 1 Worker 4, 16 Tasks vs Execution Time

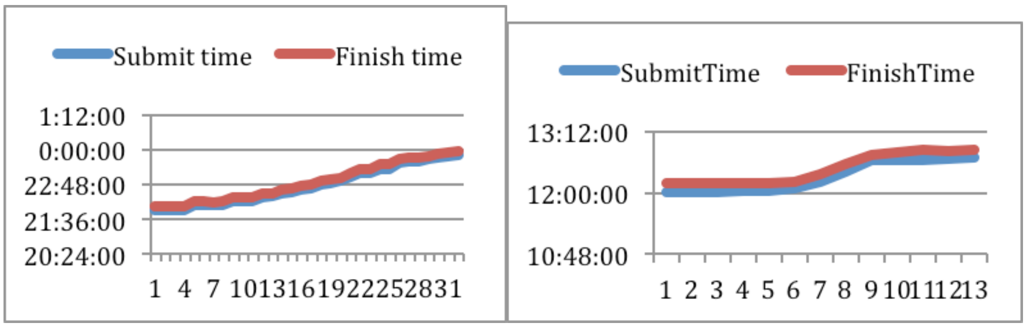


Figure 12 Engine 1 Worker 4, 16 Tasks vs Time

We start the experiment by one engine and limit the number of workers to be 4, 8, and 16. Unfortunately, the application still have some bugs, most jobs of 8 workers do not record done time successfully. Therefore, we could not collect the result for workers 8 group. From the Figure 9, we can see most of tasks are done around 8 mins, when the worker number goes up to 16, it evolves a little higher failure, less tasks are been done. In Figure 10, there are two charts. The left side is for 4 workers, when the number of task reach the multiple of 4, there are sharp upwards. It is reasonable that all the workers are busy. The right one is for 16 workers, the trend is flat as most of tasks could be handled at once.

4.1.1.2 Engines 2

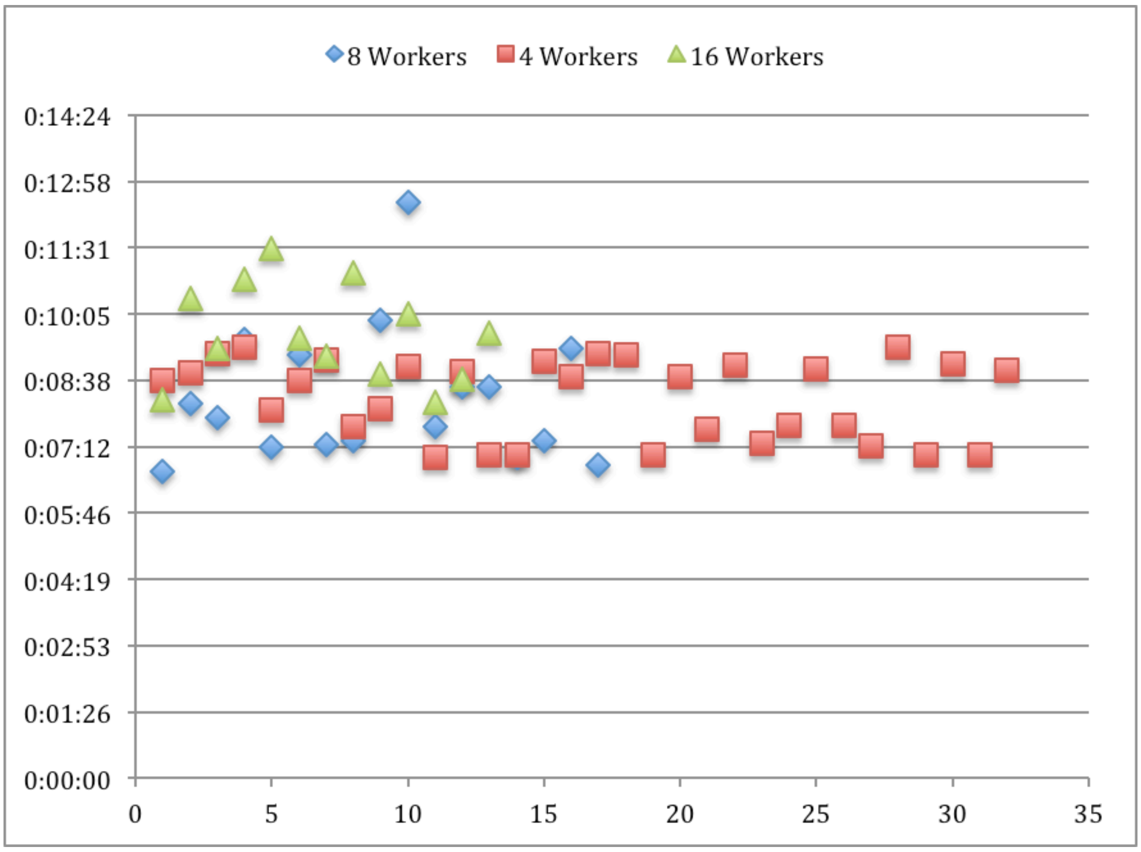


Figure 13 Engine 2 Worker 4, 8, 16 Tasks vs Execution Time

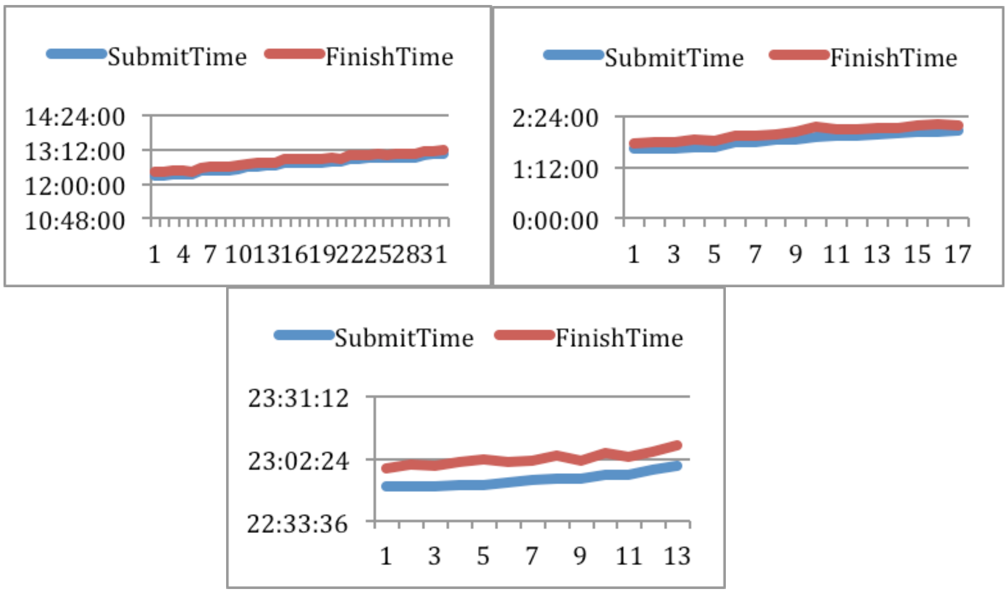


Figure 14 Engine 2 Worker 4, 8, 16 Tasks vs Time

At the second time, we run experiments with 2 engines. The workers are still limit to 4, 8 and 16 respectively. This time we get the done times for all 3 groups. From the first chart, we can see engine with 4 and 8 workers have similar execution time, the execution time of engine with 16 is slightly higher. Through comparing the rest 2 charts, we can see that once all the workers are busy, the tasks left have to wait until a worker is available.

4.1.1.3 Engines 4

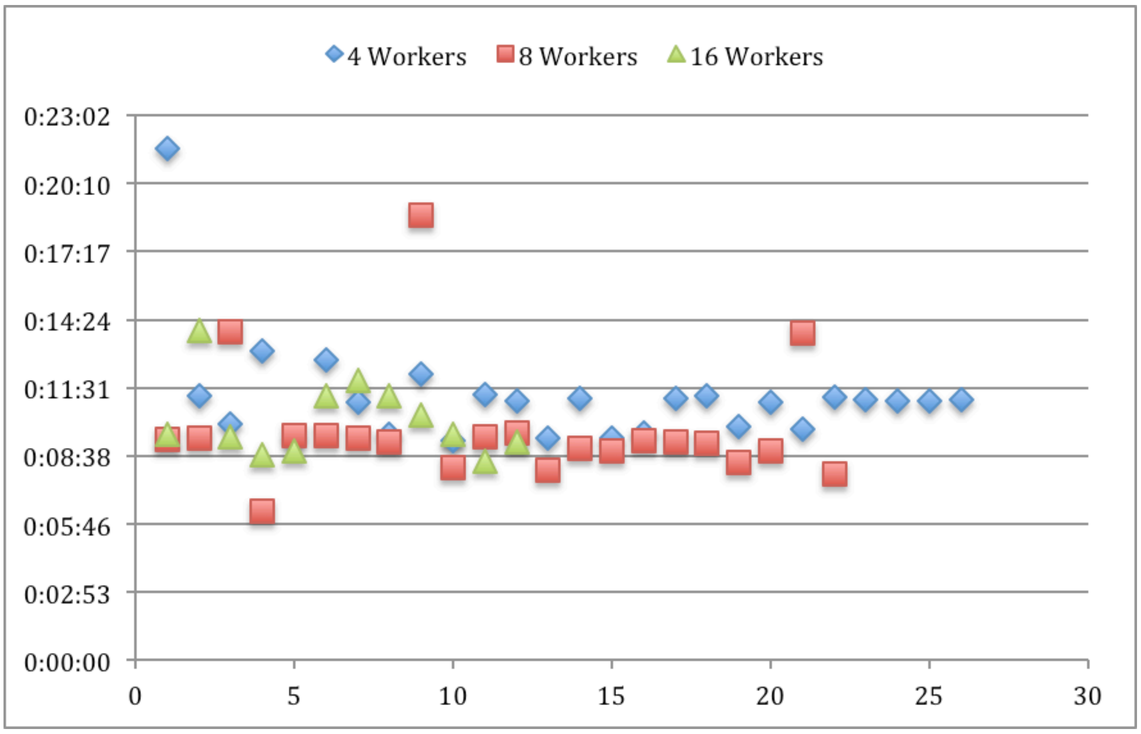


Figure 15 Engine 4 Worker 4, 8, 16 Tasks vs Execution Time

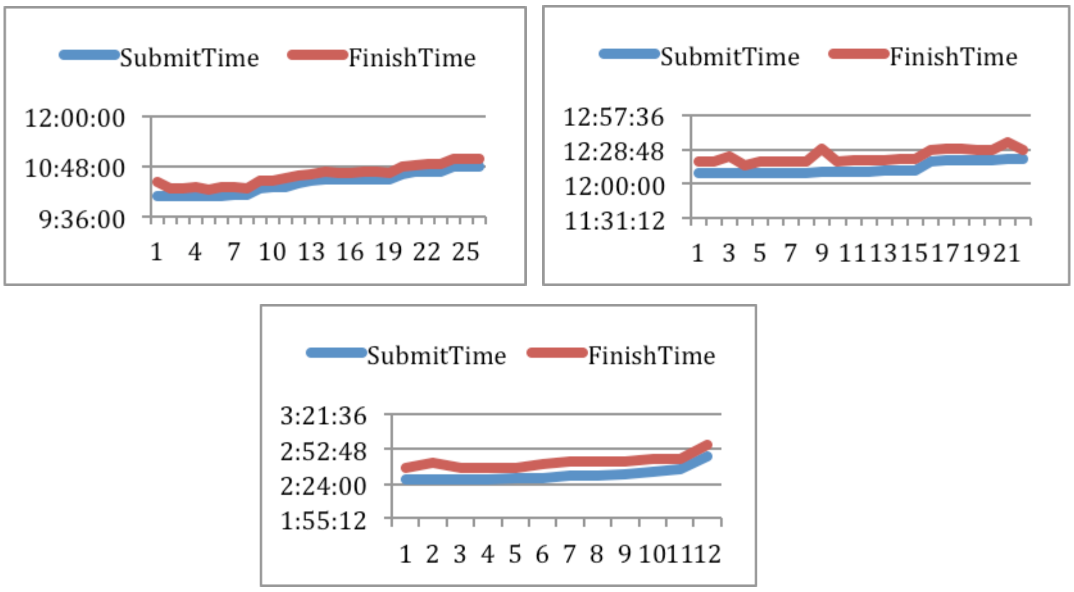


Figure 16 Engine 4 Worker 4, 8, 16 Tasks vs Time

At the third time, we run experiment with 4 engines, and we allocate 4, 8 and 16 workers for each engine. The first graph is similar to the last group, which means the number of engine do not affect the performance of individual task. From graphs of tasks vs time, we have a clear idea that the time of submission is decided by the availability of workers. The last graph is almost flat as the number of workers is greater than the number of tasks. Therefore, nearly all task are submitted once the workers are ready.

4.1.1.4 Combined Data

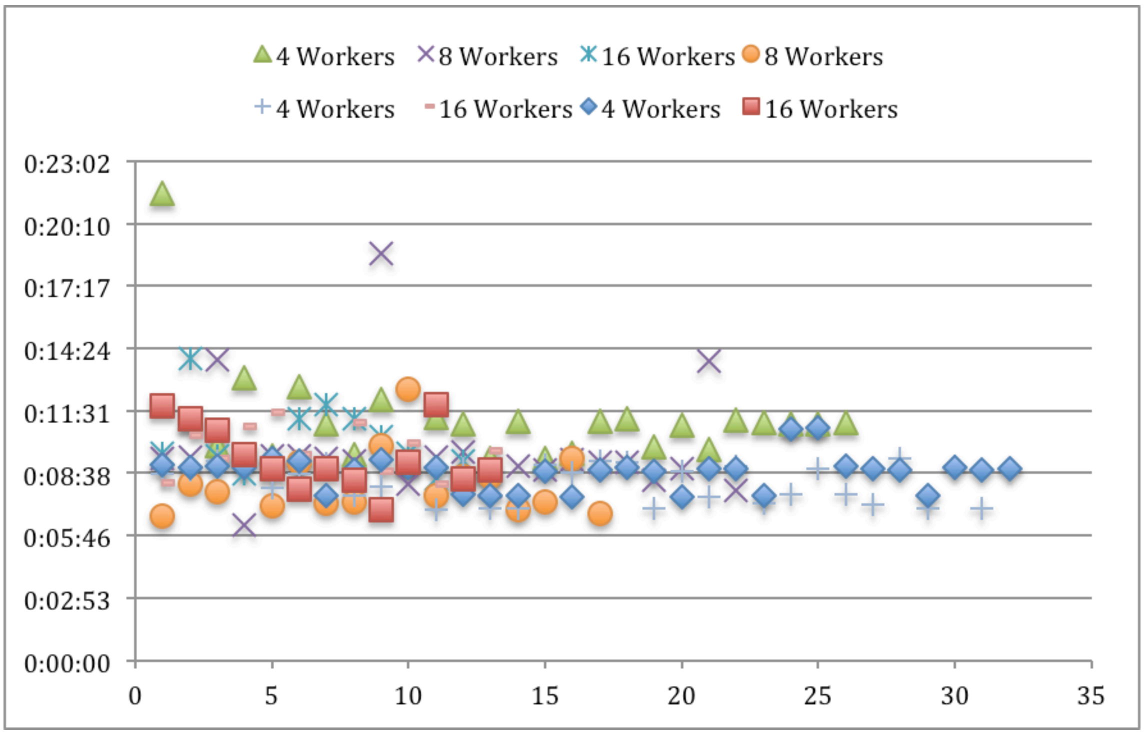


Figure 17 Combined Result

When combining all the data of this series experiment together, we plot a graph as Figure 16. It is not hard to know the average execution time of LIGO application is 9 minutes, and most of tasks have been done around 9 minutes. Only a few of them takes much longer time as the failure of execution. Less workers generally have better performance and incur less failure. 8 workers for each engine is appropriate combination in terms of resource consumption and performance.

4.1.2 Large scale experiment with 500 computing workers

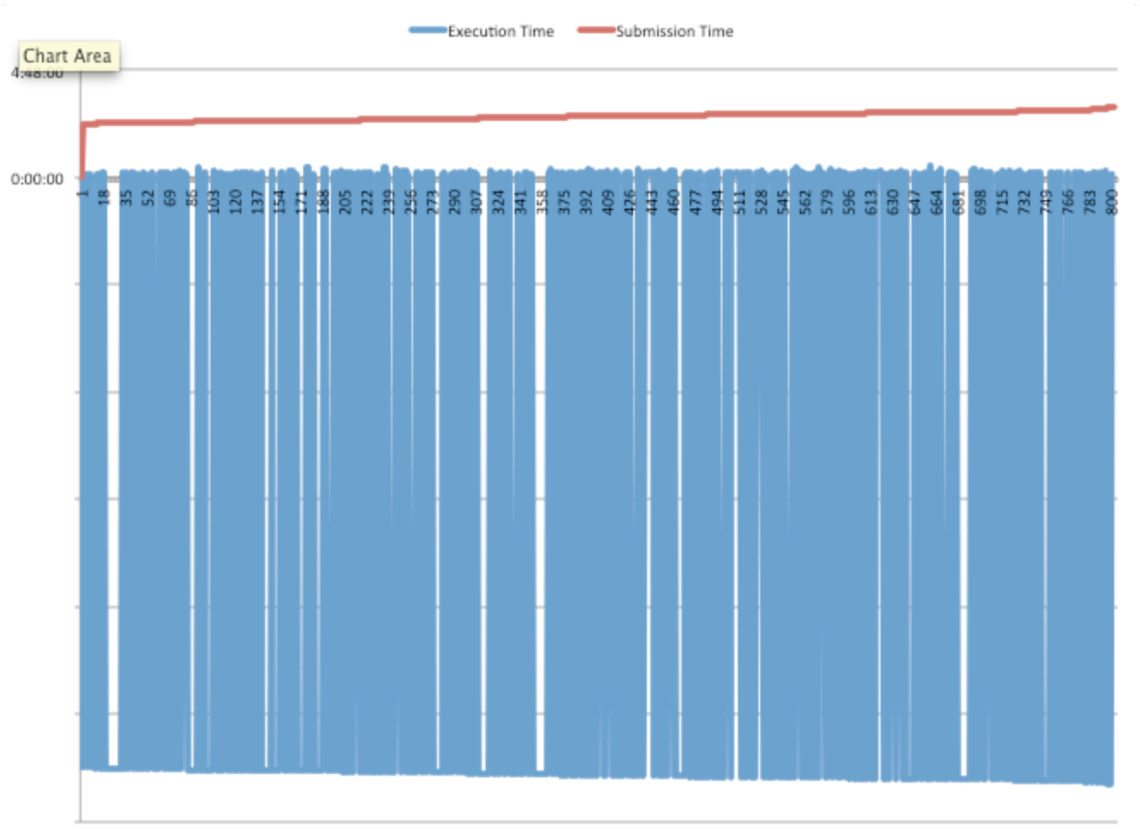


Figure 18 1000 tasks with 500 workers Tasks vs Time

We conduct many large scale experiment with up to 500 workers. Figure 18 shows the submission time and execution time for each task. There are intervals for submission time periodically, because we set a thread pool for submitting tasks to prevent exhausting the resource. When the thread pool is full, submission of tasks will wait. We can see there are white gaps between blue lines, because some tasks are failed or their done time could not be recorded successfully. This is an issue with workflow engine that we should address in the future.

4.2 Application Output

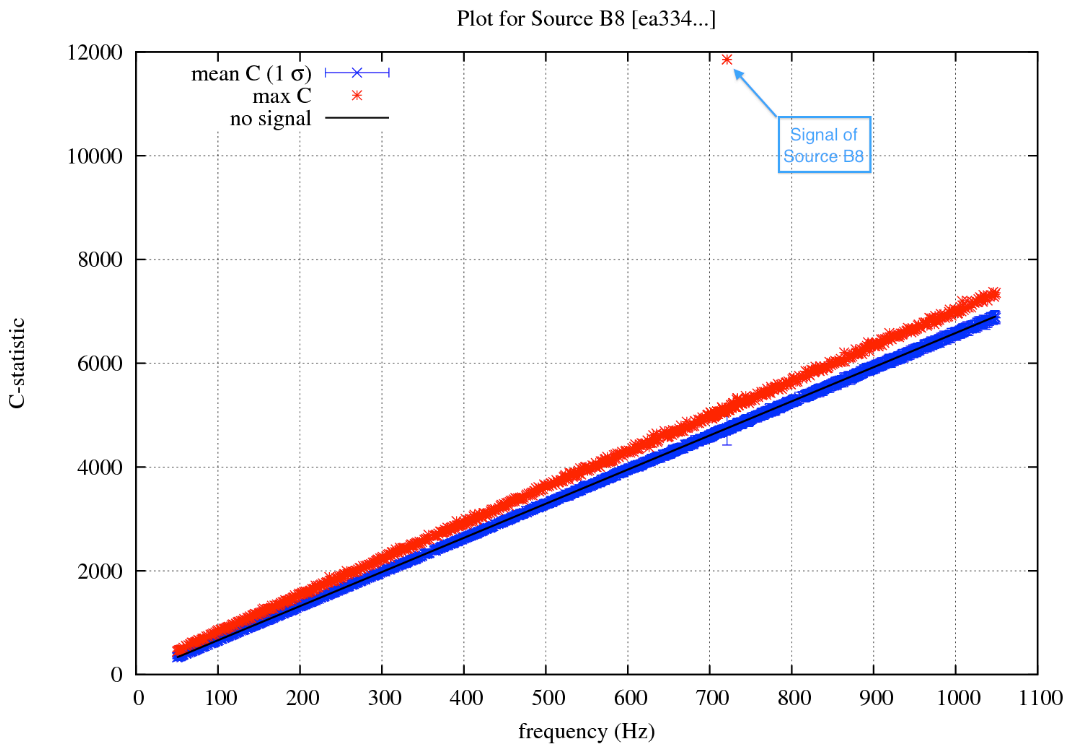


Figure 19 Source 8 Search Result

We gain Figure 16 after running through all 1000 tasks for source B8, each single point represents a result of one task. We plot the graph with 1000 result which been stored on Amzon S3 dynamically. 1000 dots link together and become the line above. There is one dot which goes far away from the line is the signal. We have highlighted with description. From the point view of Physics, this is the signal of gravitational wave.

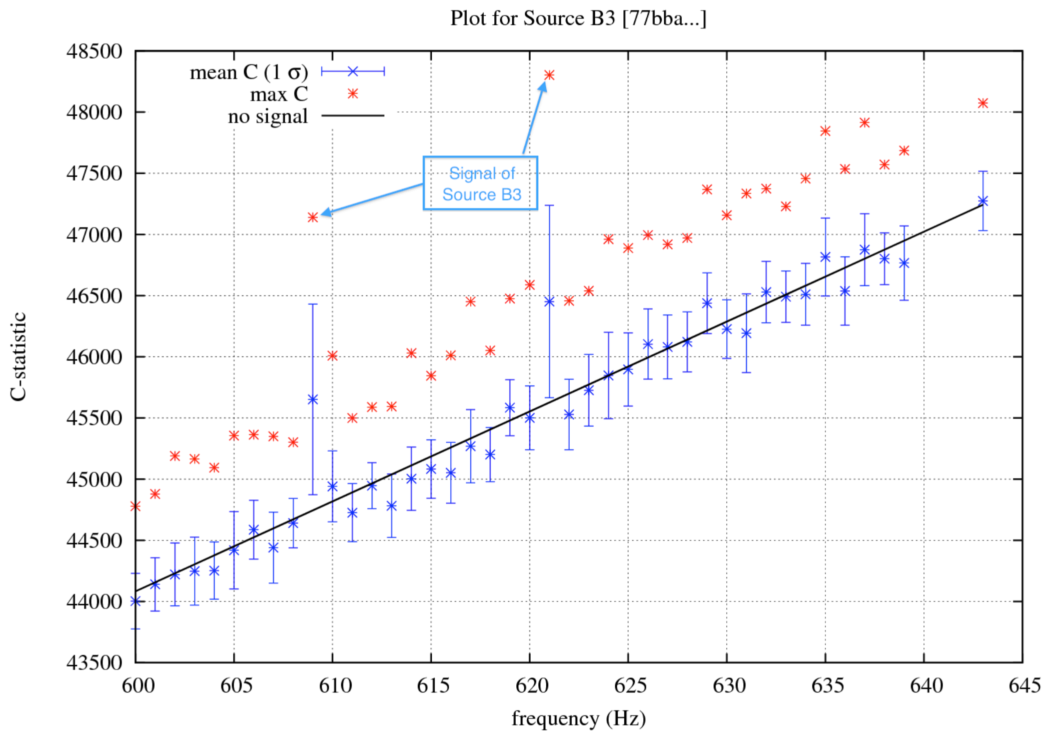


Figure 20 Search Result of Source B3

The graph above Figure 17 shows a small scale search with 40 tasks, some points are missing as the failure of tasks, but it is easy to identify the signal from this graph, there are two signals in the graph, which goes out of normal distribution of points. I have also highlight them with blue description.

5. Conclusion and Future Work

Cloud data centres are prevalent nowadays, there are a large number of IT enterprises selling their computer resource in the form of computation. However, the cooperation of computer resources in a scale manner is still a big challenge. In this work, we provide the whole system solution in order to maximum the usage of each computer resource and scale the size of system based on the amount of request from users. We design four layered system, which includes web portal for controlling and monitoring front end users, load balancer for scaling system and assigning tasks, engine for scheduling tasks and worker for executing tasks. This system have successfully applied to search gravitational waves, which evolves a large amount of computation. In our experiment, thousands of tasks can be executed in parallel. The computer resources scale in and out according to the amount of tasks. The load of each resource is balanced dynamically.

Even such system design is well proved, there are still many improvements could be made in the future. We have solved the bottleneck of engine with traditional 3 layers system. However, the web portal is still facing the pressure when the request increases. A solution is to set up several tomcat server to support multiple web portal. During our experiment, we actually have implemented it and it is easy to configure according to our experience. Another improvement which could be done is to change the protocol of web service from SOAP to REST, SOAP is a heavy and complex protocol compared with REST. When conducting the experiment and test, we do have trouble with getting the response message with SOAP. REST is more appropriate for this project for its light weight feature. At this stage, we only support Amzon EC2 cloud, later system could extend to multiple cloud provider, even mix cloud data centres with clusters. In addition, the middleware, Workflow Engine, assign tasks based on the number of cores of resource. In the case of searching gravitational wave, the application consume a great amount of memory, it is unstable to assign several tasks. Optimization could be made to middleware based on the nature of application.

6. Bibliography

1. Broberg, J., R. Buyya and Z. Tari, MetaCDN: Harnessing 'storage clouds' for high performance content delivery, Journal of Network and Computer Applications 32 (2009), pp. 1012–1022.

2. Collaboration, L. S., Lal/lalapps: Freesoftware (gpl) tools for data-analysis, http://www.lsc-group.phys.uwm.edu/daswg/, Online.

3. Deelman,E.,G.Singh,M. H.Su,J.Blythe, Y.Gil,C.Kesselman, G.Mehta,K. Vahi,G.B. Berriman,J.Good, A.Laity,J.C. Jacob and D. S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (2005), pp. 219–237.

4. Izal, M., U. G. Keller, E. Biersack, P. Felber, A. Hamra and G. L. Erice, Dissecting bittorrent: Five months in a torrent's lifetime (2004).

5. Pandey, S., D. Karunamoorthy and R. Buyya, "Cloud Computing: Principles and Paradigms," Wiley Press, 2011 pp. 299– 319.

6. Sammut, L., A. Melatos, C. Messenger and B. Owen, A frequency Comb Search for Sco X-1, in: American Astronomical Society Meeting Abstracts #214, American Astronomical Society Meeting Abstracts 214, 2009, pp. 600.06–+

7. S Pandey, WVoorsluys1 , M Rahman1, R Buyya1, J Dobson2, K Chiu, A Grid Workflow Environment for Brain Imaging Analysis on Distributed Systems, 2009 John Wiley & Sons, Ltd

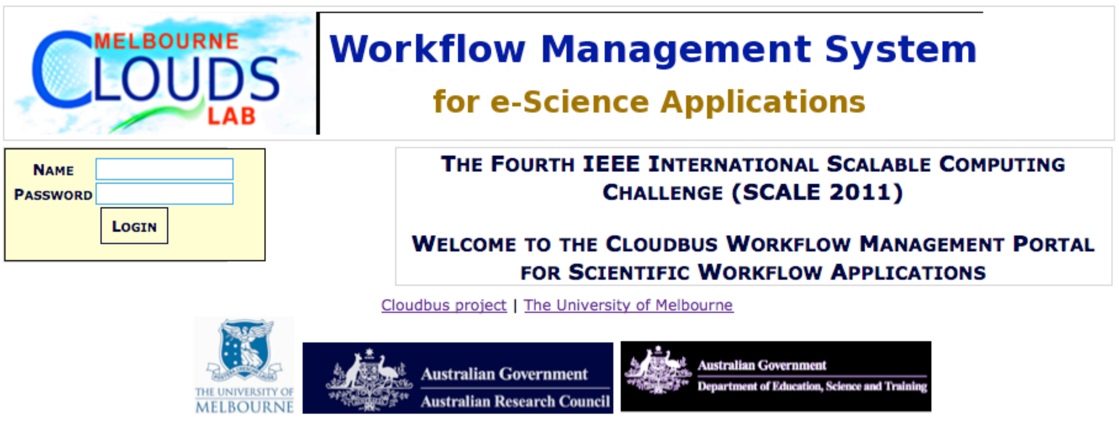
8. LSC - LIGO Scientific Collaboration, http://www.ligo.org/, Online.

9. Amazon Web Services (AWS), What is AWS? http://aws.amazon.com/what-is-aws, Online.

7. Appendices

Appendix A User Manual

1. Type the address of web portal in web browser and log in with username and password.



Workflow Management System

for e-Science Applications

NAME

PASSWORD

LOGIN

THE FOURTH IEEE INTERNATIONAL SCALABLE COMPUTING CHALLENGE (SCALE 2011)

WELCOME TO THE CLOUDBUS WORKFLOW MANAGEMENT PORTAL FOR SCIENTIFIC WORKFLOW APPLICATIONS

Cloudbus project | The University of Melbourne

Figure 21 Web Portal Screen Shot 1

2. After log in to the user panel, select Workflow Property Configuration from left navigation.

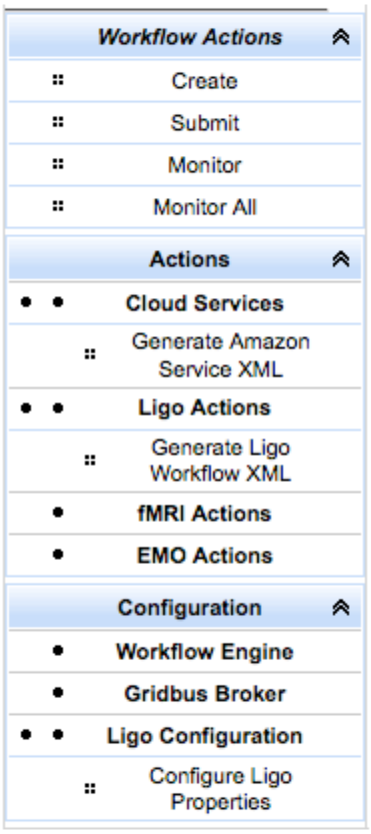
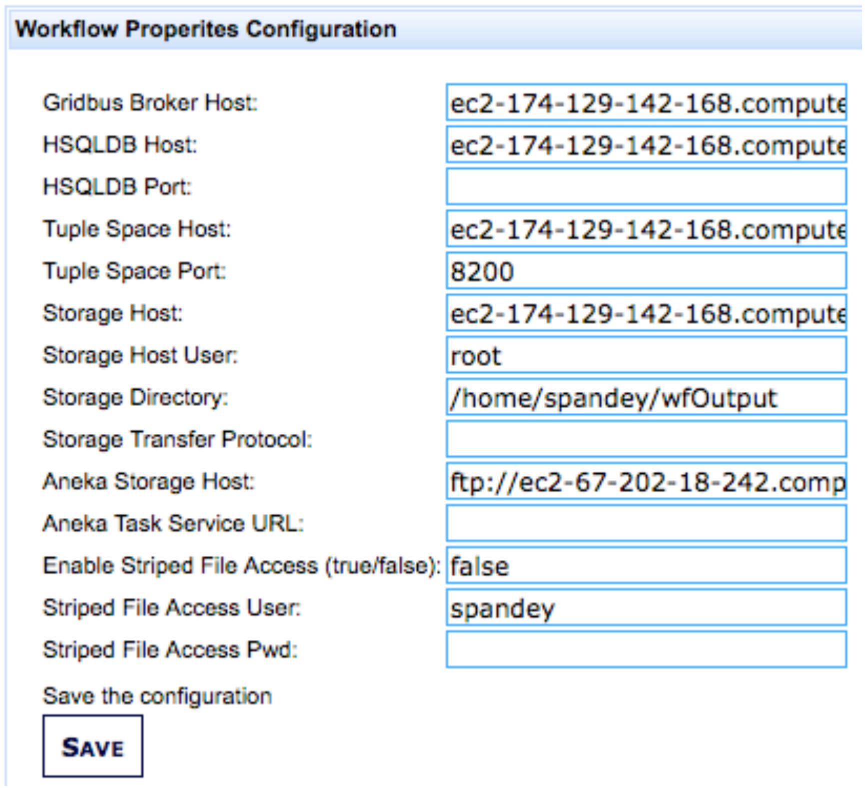


Figure 22 Web Portal Screen Shot 2

3. Setup the parameters for Workflow engine and save it.



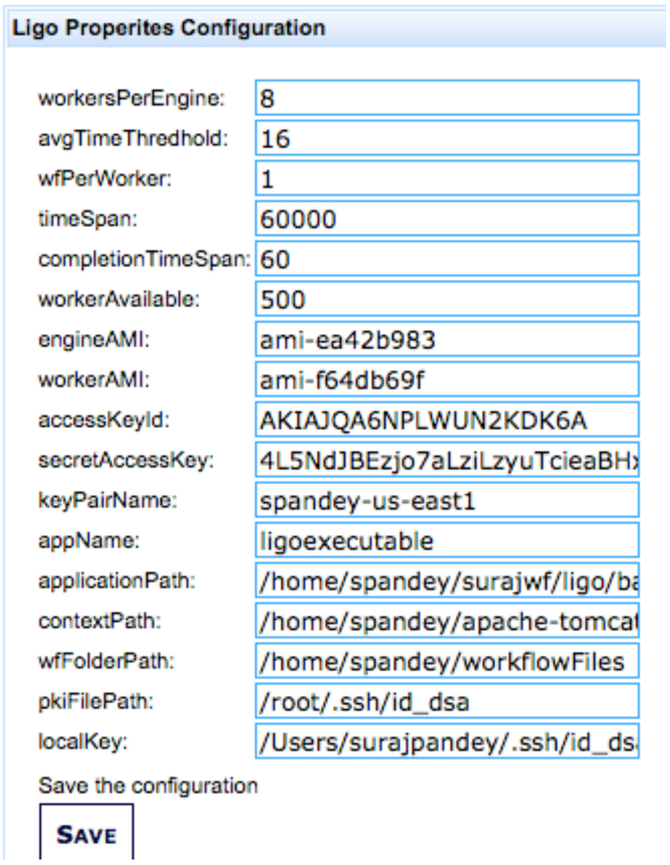
Workflow Properites Configuration

Gridbus Broker Host:	ec2-174-129-142-168.compute
HSQLDB Host:	ec2-174-129-142-168.compute
HSQLDB Port:	
Tuple Space Host:	ec2-174-129-142-168.compute
Tuple Space Port:	8200
Storage Host:	ec2-174-129-142-168.compute
Storage Host User:	root
Storage Directory:	/home/spandey/wfOutput
Storage Transfer Protocol:	
Aneka Storage Host:	ftp://ec2-67-202-18-242.comp
Aneka Task Service URL:	
Enable Striped File Access (true/false):	false
Striped File Access User:	spandey
Striped File Access Pwd:	

Save the configuration

Figure 23 Web Portal Screen Shot 3

4. Navigate to Ligo property configuration. Setup the parameters for Load balancer and save it.

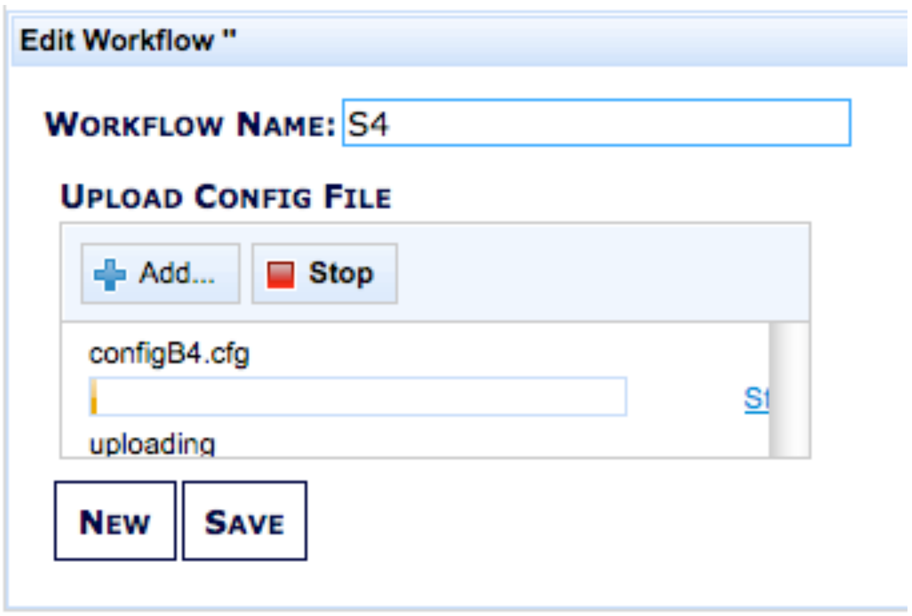

Ligo Properites Configuration

workersPerEngine:	8
avgTimeThredhold:	16
wfPerWorker:	1
timeSpan:	60000
completionTimeSpan:	60
workerAvailable:	500
engineAMI:	ami-ea42b983
workerAMI:	ami-f64db69f
accessKeyId:	AKIAJQA6NPLWUN2KDK6A
secretAccessKey:	4L5NdJBEzjo7aLziLzyuTcieaBH
keyPairName:	spandey-us-east1
appName:	ligoexecutable
applicationPath:	/home/spandey/surajwf/ligo/b
contextPath:	/home/spandey/apache-tomca
wfFolderPath:	/home/spandey/workflowFiles
pkiFilePath:	/root/.ssh/id_dsa
localKey:	/Users/surajpandey/.ssh/id_ds

Save the configuration

Figure 24 Web Portal Screen Shot 4

5. Select Ligo action > Generate Ligo Workflow XML from left navigation, go to the bottom section, type the name for workflow application and upload source configuration file. Then click refresh button, the description and operation of workflow application will come up in the table.


Edit Workflow ''

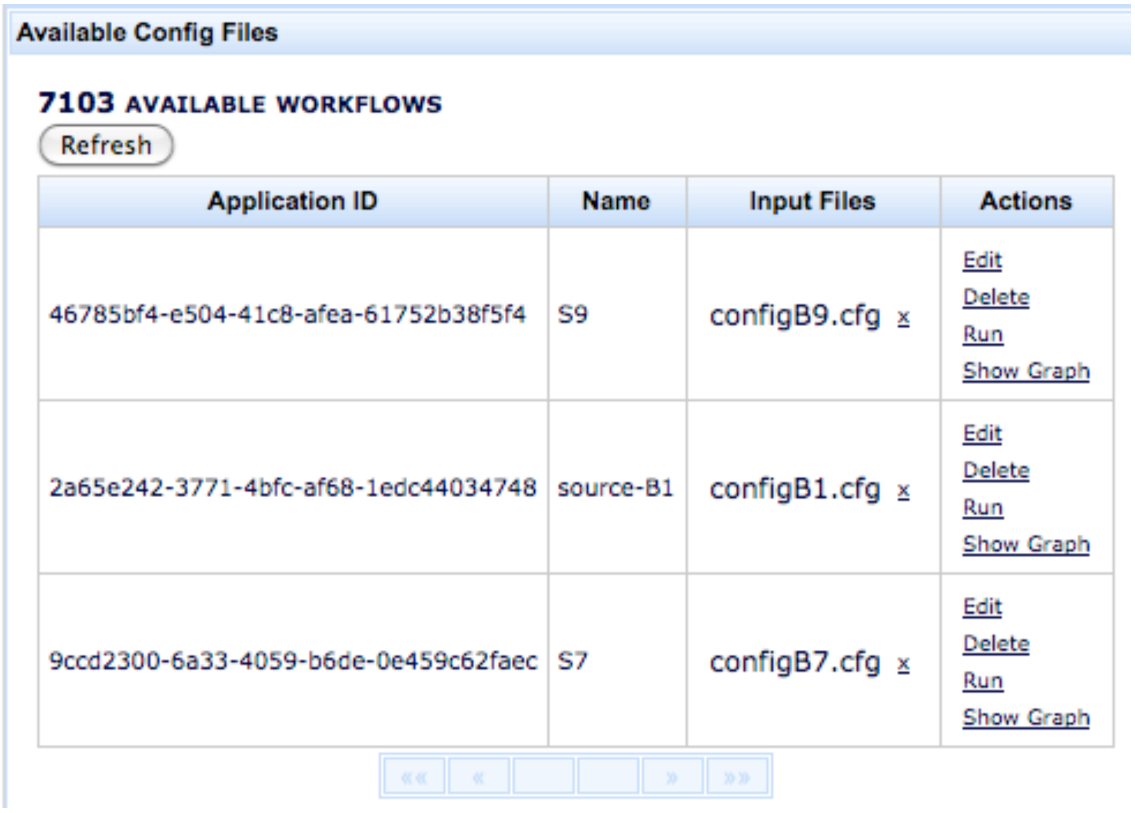
WORKFLOW NAME: S4

UPLOAD CONFIG FILE

configB4.cfg

uploading

Figure 25 Web Portal Screen Shot 5

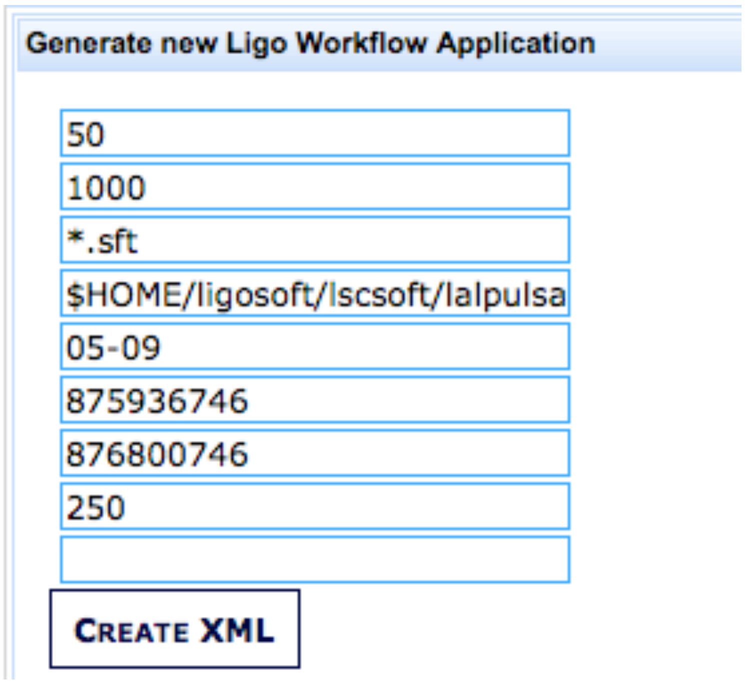

Available Config Files

7103 AVAILABLE WORKFLOWS

Application ID	Name	Input Files	Actions
46785bf4-e504-41c8-afea-61752b38f5f4	S9	configB9.cfg x	Edit Delete Run Show Graph
2a65e242-3771-4bfc-af68-1edc44034748	source-B1	configB1.cfg x	Edit Delete Run Show Graph
9ccd2300-6a33-4059-b6de-0e459c62faec	S7	configB7.cfg x	Edit Delete Run Show Graph

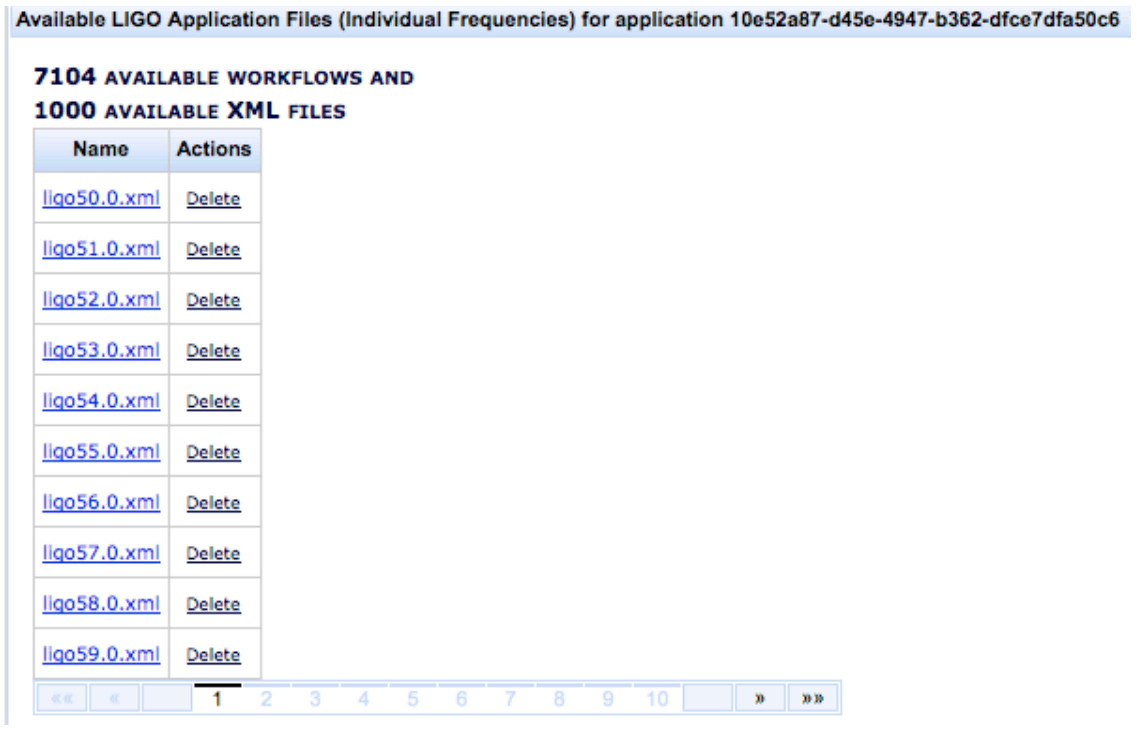
Figure 26 Web Portal Screen Shot 6

6. Setup the application parameters, input the name of source that we just uploaded. Then click "CREATE XML" button. The application files of task will generate immediately. We can browser these files as shown in Figure 27.


Generate new Ligo Workflow Application

50
1000
*.sft
$HOME/ligosoft/lscsoft/lalpulsa
05-09
875936746
876800746
250

Figure 27 Web Portal Screen Shot 7



Available LIGO Application Files (Individual Frequencies) for application 10e52a87-d45e-4947-b362-dfce7dfa50c6

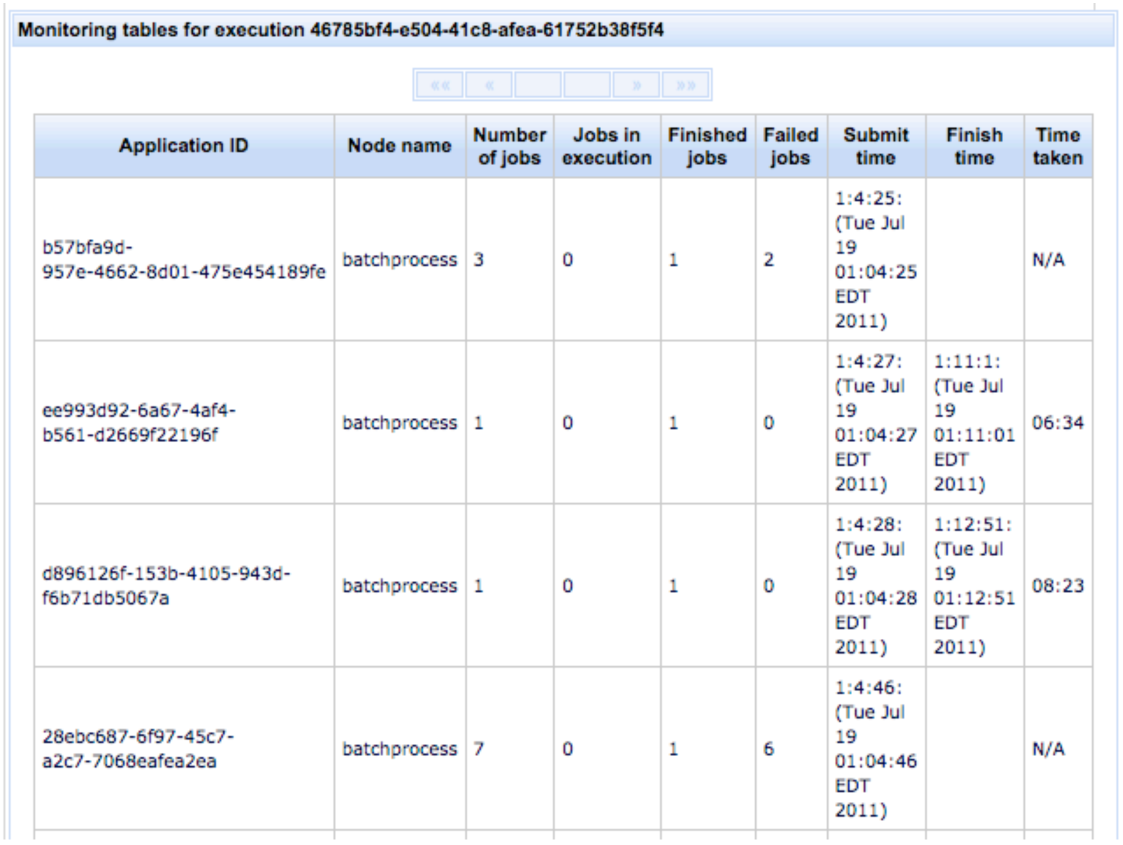
7104 AVAILABLE WORKFLOWS AND
1000 AVAILABLE XML FILES

Name	Actions
ligo50.0.xml	Delete
ligo51.0.xml	Delete
ligo52.0.xml	Delete
ligo53.0.xml	Delete
ligo54.0.xml	Delete
ligo55.0.xml	Delete
ligo56.0.xml	Delete
ligo57.0.xml	Delete
ligo58.0.xml	Delete
ligo59.0.xml	Delete

Figure 28 Web Portal Screen Shot 8

7. As we have already gotten application files ready. Now we may run the application with single click, it is shown in Figure 25. After a short time, the running tasks will be recorded in the database. We can monitor the details of running tasks dynamically by clicking “Monitor All” from left navigation as shown in Figure 28.

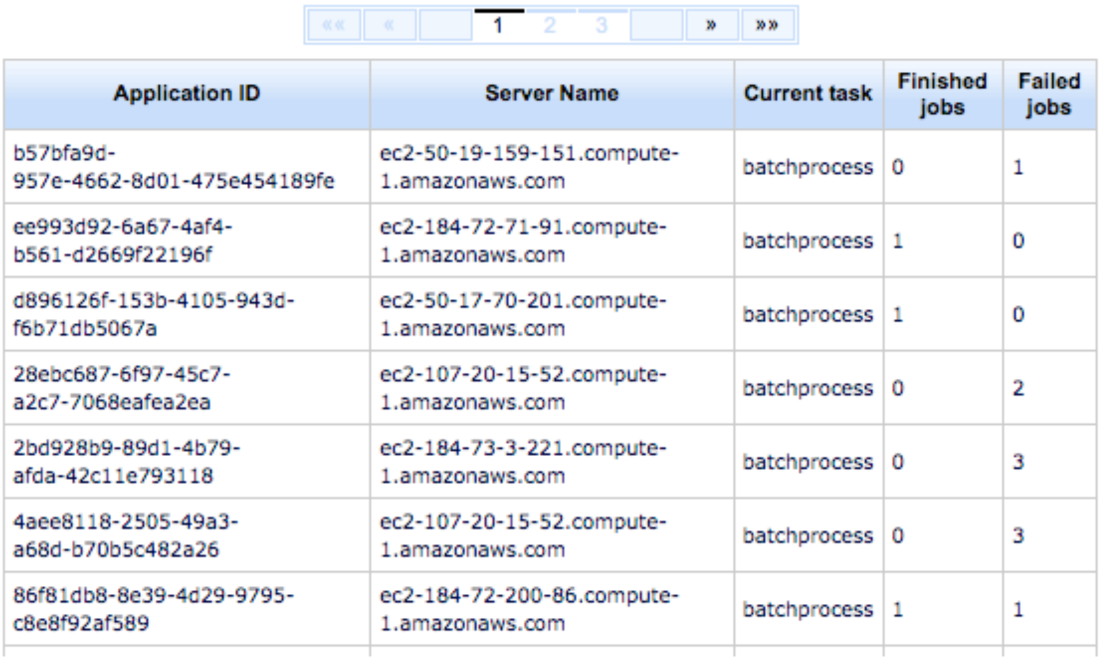
Also, users can see the running worker for each task (Figure 29)



Monitoring tables for execution 46785bf4-e504-41c8-afea-61752b38f5f4

Application ID	Node name	Number of jobs	Jobs in execution	Finished jobs	Failed jobs	Submit time	Finish time	Time taken
b57bfa9d-957e-4662-8d01-475e454189fe	batchprocess	3	0	1	2	1:4:25: (Tue Jul 19 01:04:25 EDT 2011)		N/A
ee993d92-6a67-4af4-b561-d2669f22196f	batchprocess	1	0	1	0	1:4:27: (Tue Jul 19 01:04:27 EDT 2011)	1:11:1: (Tue Jul 19 01:11:01 EDT 2011)	06:34
d896126f-153b-4105-943d-f6b71db5067a	batchprocess	1	0	1	0	1:4:28: (Tue Jul 19 01:04:28 EDT 2011)	1:12:51: (Tue Jul 19 01:12:51 EDT 2011)	08:23
28ebc687-6f97-45c7-a2c7-7068eafea2ea	batchprocess	7	0	1	6	1:4:46: (Tue Jul 19 01:04:46 EDT 2011)		N/A

Figure 29 Web Portal Screen Shot 9



Application ID	Server Name	Current task	Finished jobs	Failed jobs
b57bfa9d-957e-4662-8d01-475e454189fe	ec2-50-19-159-151.compute-1.amazonaws.com	batchprocess	0	1
ee993d92-6a67-4af4-b561-d2669f22196f	ec2-184-72-71-91.compute-1.amazonaws.com	batchprocess	1	0
d896126f-153b-4105-943d-f6b71db5067a	ec2-50-17-70-201.compute-1.amazonaws.com	batchprocess	1	0
28ebc687-6f97-45c7-a2c7-7068eafea2ea	ec2-107-20-15-52.compute-1.amazonaws.com	batchprocess	0	2
2bd928b9-89d1-4b79-afda-42c11e793118	ec2-184-73-3-221.compute-1.amazonaws.com	batchprocess	0	3
4aee8118-2505-49a3-a68d-b70b5c482a26	ec2-107-20-15-52.compute-1.amazonaws.com	batchprocess	0	3
86f81db8-8e39-4d29-9795-c8e8f92af589	ec2-184-72-200-86.compute-1.amazonaws.com	batchprocess	1	1

Figure 30 Web Portal Screen Shot 10