Job Dispatching and Scheduling under Heterogeneous Clusters

Ting-Chou Lin, Pangfeng Liu
The Graduate Inst. of Networking and Multimedia
National Taiwan University
{,pangfeng}@csie.ntu.edu.tw

Ching-Chi Lin
Department of C.S.I.E
National Taiwan University
d00922019@csie.ntu.edu.tw

Chia-Chun Shih,Chao-Wen Huang
Chunghwa Telecom Laboratories
{ccshih, huangcw}@cht.com.tw

***Abstract*—Many enterprises or institutes are building private clouds by establishing their own data center. In such data centers, the physical machines can be different due to annual upgrades, but the amount of machine are fixed for most of the time. In such heterogeneous environment, scheduling jobs with different resource requirements and characteristic in order to meet different timing constraints is important.**

In this paper, we proposed a cloud resource management framework to dynamically adjust the number of computation nodes for every job in the system.

***Keywords*-Job/Task scheduling; Heterogenous; Resource allocation;**

I. Introduction

Cloud computing has become a popular issue in recent year. Many enterprises or institutes are building private clouds by establishing their own data center. In such data centers, the maximum amount of computing resources, or number of physical machines, are fixed for most of the time. It only changes during adding newly purchased servers, or remove obsolete ones during equipment upgrade.

Jobs in a data center vary from one another. Jobs may have different characteristic, resource requirements, time constraints, and priority. For example, scientific computation requires large computing resources, while a billing subsystem need to generate the credit-card bill for each user every month. Also, newly arrival emergency jobs may require large amount of resource in a short time. How to allocate jobs in a heterogeneous environment in order to meet different requirements becomes an important issue.

In this paper, we proposed a cloud resource management framework to dynamically adjust the number of computation nodes for every job in the system. This framework make decisions according to some specified policies – *priority-based*, *proportion-based*, *workload-based* and *deadline-based*. The policies take the remaining workload, priority or deadline of each job into consideration, and generate a scheduling plan that satisfies the job requirements. This framework can work as an individual cloud computing system, or as extension components of an existent cloud system.

The rest of the paper is organized as followed. The system architecture is presented in Section II. Section III introduce the four policies we implement for different types of jobs. The experimental results are in Section IV. Section V is the conclusion.

II. System Architecture

In this section, first we will introduce the job and task model in our framework. Then we give an overview of the proposed framework, followed by the description and implementation of each component. The system flow is presented in the last section.

A. Job and Task Model

We define our job and task models as the following. A *job* submitted by a user consists of several *tasks*, which is the minimum scheduling unit of our system. *Tasks* in the same job are independent to each other, so that they can be processed in parallel. A *job* is finished if all the *tasks* from this *job* have been processed.

Here is an example of *job* and *task*. In a telephone company, they have to send bills to users every month. However, the settle days of users are not the same. Therefore the system will batch user data with the same settle day into a *job*, and calculate the amount of money each user has to pay. Since user data are independent to each other, they can be processed in parallel. We called the computation of each user data as *task*.

We make some assumptions about our models. We assume that the number of *tasks* in a *job* is known. *Tasks* can have different workload in a *job*. The remaining workload of a *job* can be roughly estimated by counting the number of unprocessed *tasks* or summing the workloads of these *tasks*. A *job* may have "deadline", which means all the *tasks* of this *job* must be finished before this time constraint.

B. System Overview

Our cloud management framework consists of three parts, the *management system*, *clients*, and *workers*. The *management system* is the major part of our framework. Users submit *jobs* from *clients*. *Workers* are the computation nodes that process *tasks* in *jobs*. Figure 1 gives the idea of the proposed framework. The proposed framework can work as an individual cloud computing system, or as extension components of an existent cloud system.

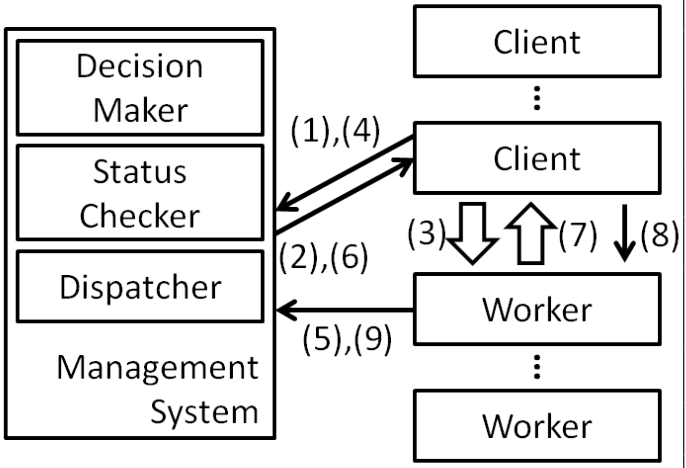

Figure 1. System Architecture Overview

We implemented the components in the *management system* and the *worker* as separate RPC (remote procedural call) [1] servers. Separate RPC server implementation makes each component *pluggable*. The pluggability gives the system administrator flexibility to choose the most suitable component implementation to fit different needs. Moreover, without shutting the whole system down, it is possible to change system configurations — or even upgrade the system — by substituting target components with feasible ones. Besides, this design allows us to easily integrate the management system with other cluster management frameworks like JPPF [2] or cloud operating system like Roystonea [3]. Roystonea also benefits from the RPC server implementation.

1) Client: The *client* is the programming interface for users to submit jobs to our system. In our framework, a user will submit *jobs* to the system via a client instance created using the provided library.

Users specify several attributes to jobs before submission, such as deadline, priority and execution profile from previous experience. These attributes are later passed to decision maker for reference so that it can schedule the jobs.

The client library also provides several features to end users to satisfy their varying programming needs. First, we support background task execution, which means users can specify the synchronization point that waits all the submitted task to be done as they like, without blocking the code before it. Besides, user can submit multiple jobs at a time. Combining these two features, user can easily program the synchronization model they want, just like those traditional work flow description languages can do [4]. Here's an example on how to program with our client library.

```
j1 = Job.new('Job1')
j1.add_task Task.new(...)
... # Add more tasks
j2 = Job.new('Job2')
j2.add_task Task.new(...)
... # Add more tasks
# 200-second deadline
j1.deadline = j2.deadline = Time.now + 200.0
# Submit j1 and j2 together
c12 = Client.new([j1,j2])
# Background execution
c12.start
# Do other time consuming computation
...
j3 = Job.new('Job3')
j3.add_task Task.new(...)
... # Add more tasks
c3 = Client.new(j3)
c3.start
# Remaining part can't run until j3 is done
c3.wait_all
# Some more things to do
...
# Wait until j1 and j2 is done.
c12.wait_all
# Combning j1, j2 and j3
...
```

Example Code 1: Sample code of client usage

Assume that the user has 3 jobs — j_1, j_2 and j_3 — to be done separately and their result to be aggregated: j_1 and j_2 takes long time to run on the cluster can be submitted directly; j_3 takes shorter time on remote clusters if separated into parallel tasks but requires time consuming local pre-processing and post-processing. Since j_1, j_2 and pre-processing of j_3 is costive, it's better to do them in parallel. As a result, we submit j_1 and j_2 at first and wait for their results in the background then start pre-processing j_3. Not needing post-processing, results of j_1 and j_2 are synchronized after j_3. Finally, as all of the results come back, we can do the final aggregation.

2) Worker: Workers are the computation nodes that consume cloud resources to process *tasks*. They execute *tasks* assigned by the management system. A *worker* instance executes at most one *task* at a time. In other words, a *worker* is the minimum scheduling slot of our system. However, it does not imply that a physical machine can run only one *task* at a time. A physical machine can serve multiple *worker* instances, therefore executing more than one task simultaneously.

Deploying multiple *worker* instances on a powerful machine (e.g., with large number of cores) can gain better performance from multiprogramming. But in contrary, it might however cause resource contention on low-end machines. We leave the choice to system administrators.

If the task execution performance of a machine running multiple worker instances is far worse than expected, it is very likely due to resource contention. In that case, the system administrator should consider reducing the number of worker instances on that machine. The pluggable implementation enable the reduction to be done *online*, i.e. without stopping any of other components.

3) Management System: The core part is the management system, which consists of three components: *status checker*, *decision maker* and *dispatcher*. *Status checker periodically collects the information about* worker*, and sends these information to* Decision Maker. *Decision maker is in charge of adjusting resource allocation plans.* Dispatcher is the component that deal with the physical resource allocation adjustment according to the allocation plan made by the *Decision Maker.*

Status Checker: Status checker periodically collects the information about worker *instances and physical servers. Leveraging on the information collected by status checker, the decision maker can make resource allocation plans according to a policy specified by the system administration.*

The most essential information for the system is probably the status of workers. A worker can be in status of either available*,* occupied*,* busy *or* down. Available means the worker is idle and ready to accept task assignment; *occupied indicates that the worker is assigned to a job but executing any task (e.g., waiting for necessary data);* busy is the state that a worker is really executing a task; *down shows that the worker instance is currently nonfunctional.*

Aside from essential worker status, system administrators can also specify other kinds information to collect if it's essential to their customized policy. For example, if someone deployed the management system on a cluster which focuses on utilizing its I/O bandwidth, they would probably specify a schedule policy in regard to I/O utilization of the physical server. In this situation, status checker must collect the information about not only the worker instances but the physical server as well.

Necessary information will be written to a database for persistent storage. This allows status checker to recover its state and work normally after being plugged off. Besides, writing to a external database makes integration with other framework easier since it's more general to access a database.

Decision Maker: Decision maker is in charge of adjusting resource allocation plans. It makes allocation plans according to a specified policy *that takes different parameters, for example, job deadline and priority, into consideration.*

Instead of letting decision maker schedule resources actively, we decided to have it work in passive mode*, which means it is invoked only under certain circumstances. Normally, it is only invoked by dispatcher whenever a job is submitted or finished. By this design, decision maker is only responsible for making allocation plans. In other words, the only job this component do is to perform the schedule algorithm and give results.*

Doing seemingly little work only, decision maker is however the most critical part of the management system. This is not only because the whole system acts according to the plan the decision maker makes, but also because it provides the flexibility that other frameworks can easily obtain the result of the scheduling algorithm by invoking decision maker.

Dispatcher: Dispatcher is the component that deal with the physical resource allocation adjustment according to the allocation plan made by the *Decision Maker.* Dispatcher receives job requests from clients, and sends the job information to the *Decision Maker. After the decision is made, the* Dispatcher response to the client which *workers are assigned to the run the* job.

4) System Flow: Figure 1 shows our system flow. 1) The client sends a job request to the Dispatcher in the management system. 2) According to the pre-defined policy and current worker status, the Decision maker generates a resource allocation plan. The dispatcher responds to the client about which workers the client should executes its job on. 3) The client transfer the required data to workers. 4) The client sends a pending request to the management system, waiting for response after the management system receives the "job done" signal from workers. 5) After finishing the job, the workers send "job done" signal to the management system. 6) On receiving the "job done" signal, the management system informs the corresponding client. 7) The client collects the job results from workers, and 8) send an acknowledgement to workers after the transmission is completed. 9) Last, the worker send a "clear" signal to management system.

III. Scheduling Policies

In this section, we introduce the four policies in our system — *priority-based*, *proportion-based*, *workload-based* and *deadline-based*. Each policy requires different parameters such as job priority, execution efficiency per different server and execution deadline. *Decision maker makes dispatching plans to according to a specified policy.*

A. Priority-based Policy

Priority-based *distributes workloads according to job priority. We first sort all jobs in the system according to their priority in descending order. Starts from the job with the highest priority, we assign x workers to the job, where $x = max(number of idle worker, number of tasks in the job)$. The job is removed from the list. The process is repeated until there are no idle workers or unassigned jobs. The idea behind this policy is to make the job with highest priority runs as fast as possible. However, this assignment might cause* starvation *to jobs with low priority*

To avoid starvation*, we preserve a portion for low priority jobs. We set a reservation rate $r \in [0, 1]$ that the portion r of workers will be reserved for low priority jobs. The job with highest priorities can use at most the $1 - r$ of all workers* *and the remaining workers, including those are not reserved but still not assigned to the highest priority job because of the task amount limit, will be assigned to jobs with lower priority, in the manner that one job receives one worker. The complete algorithm is shown as algorithm 1.*

```
Algorithm 1: Priority-based policy
  Input: $workerSet = \{w_1, w_2, \ldots, w_m\}$,
         $jobSet = \{j_1, j_2, \ldots, j_n\}$, preserving rate
         $r \in [0, 1]$
  Output: A mapping of each job to scheduled workers
1 $result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key $\in jobSet$);
2 $jobSet \leftarrow sortByPriority(jobSet)$;
3 $c \leftarrow min(jobSet[0]$.totalTask, $workerSet$.size$-floor((1-r) * workerSet$.size);
4 $result[jobSet[0]] \leftarrow \{w_1, ..., w_c\}$;
5 for $i = 1$ to $min(m-c, n)$ do
6     $result[jobSet[i]] \leftarrow \{w_{c+i}\}$;
7 end
8 return result;
```

B. Proportion-based Policy

The main concern of Proportion-based policy is the workload proportion of a job to all the others. In contrast to *priority-based, this policy does not take job priorities into consideration unless the priority of a task is strongly related to its workload. The* Decision maker first summing all the workloads from each job, then compute the ratio of the individual job. The number of workers a job is assigned equals to the number of all workers multiplies it ratio. This algorithm, as shown in algorithm 2, is also known as *fair share scheduling* [5]. Workers a job should be allocated is in proportion to its workload proportion to all jobs.

Proportion-based is a suitable solution for streaming jobs or jobs with dependencies. For streaming jobs with vary input rate, the number of workers needed is highly related to its input rate, thus workload. As for jobs with dependencies, a job may need the output from other jobs as input. Therefore, we should assign more workers to the predecessor in the beginning, and reduce the number of workers according to time. On the other hand, the number of worker of a successor will be increasing since the workload from the predecessor increased.

C. Workload-based Policy

Similar to the proportion-based policy, the *workload-based policy takes the workload of jobs as scheduling metric. However,* workload-based takes worker heterogeneity, i.e. each worker may have different computing ability, into consideration. This policy tries to finish every job by assigning jobs to the most suitable workers, i.e. workers with higher throughput.

```
Algorithm 2: Proportion-based policy
  Input: $workerSet = \{w_1, w_2, \ldots, w_m\}$,
         $jobSet = \{j_1, j_2, \ldots, j_n\}$,
  Output: A mapping of each job to scheduled workers
1 $result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key $\in jobSet$);
2 $jobSet \leftarrow sortByPriority(jobSet)$;
3 for $i = 1$ to $m$ do
4     break if $workerSet$.size equals to 0 ;
5     $c \leftarrow min(ceil(\frac{jobSet[i].\text{workload}}{\text{total workload}})), workerSet$.size) ;
6     $result[jobSet[i]] \leftarrow$ {first $c$ element of $workerSet$};
7     $workerSet \setminus result[jobSet[i]]$;
8 end
9 return result;
```

1) *Model Definition:* We represent each job j_i submitted as $j_i = (w_i, p_i, n_i)$, where w_i is the estimated workload, p_i is the job priority, and n_i is the number of tasks in j_i. As for the workers, we model each worker s_k as a vector of estimated *throughput* of each submitted job; we can write is as $s_k = (th_1^k, th_2^k, \ldots)$.

2) *Algorithm:* We apply greedy approach in this policy. First, we sort all the jobs according to their priority in descending order. Starts from the job with the highest priority J_m, sort the remaining workers according to their throughput th_m to J_m. Then assign the top k workers that can satisfy the workload requirement of J_m, and remove assigned workers from the list. The job is also removed from the list waiting for assignment. Repeat this process until there are no idle worker or un-assigned job. The complete algorithm is shown as algorithm 3.

D. Deadline-based Policy

Deadline-based policy tries to finish every jobs before their deadline. If a feasible scheduling plan does not exist, this policy tends to meet the deadlines for jobs with higher priority. There are varies kinds of jobs that must be finished before a given time constraint. For example, as a Internet service provider, settling monthly bills of millions of users within few days after the monthly charge-off day is very critical to their business. It it obvious that computing resource allocated to this kind of job should increase as their deadline approaches — the closer the deadline is, the more worker a job should get. Unfortunately, policies introduced above does not directly adapt to this need; hence, we introduce this policy designed for deadline-aware *scenarios.*

1) *Model Definition: For this policy, we assume that each job is provided with priority and deadline. The execution time for each job is assumed to be known, which can be estimated by historical data or profiling. A job j_i can thus be represented as $j_i = (d_i, p_i, n_i)$, where d_i, p_i, n_i*

Algorithm 3: Workload-based policy

```
Input: $workerSet = \{w_1, w_2, \ldots, w_m\}$,
       $jobSet = \{j_1, j_2, \ldots, j_n\}$,
Output: A mapping of each job to scheduled workers
$result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key $\in jobSet$);
$jobSet \leftarrow sortByPriority(jobSet)$;
for each job $j_i = (w_i, p_i, n_i) \in jobSet$ do
    if $workerSet$ is empty then
        break;
    end
    $cmp \leftarrow function(w)\{$return $w.th[j_i]\}$;
    $workerSet \leftarrow sort(workerSet, cmp, DECSENDING)$;
    $th \leftarrow 0$;
    for ( $k = 0$ ;
    $k < workerSet.size$ AND $k < n_i$ AND $th < w_i$;
    $k \leftarrow k + 1$) do
        $th$ += $workerSet[k].th[j_i]$;
    end
    $result[j_i] \leftarrow workerSet[0...i]$;
    $workerSet$.remove(0,$k$)
end
return $result$;
```

Algorithm 4: Deadline-based policy

```
Input: $workerSet = \{w_1, w_2, \ldots, w_m\}$,
       $jobSet = \{j_1, j_2, \ldots, j_n\}$,
Output: A mapping of each job to scheduled workers
$result \leftarrow$ KeyValueMap(key $\rightarrow \{\}$ for all key $\in jobSet$);
$jobSet \leftarrow sortByPriority(jobSet)$;
for each job $j_i = (d_i, p_i, n_i) \in jobSet$ do
    if $workerSet$ is empty then
        break;
    end
    $cmp \leftarrow function(w)\{$return $1/w.execTime[j_i]\}$;
    $workerSet \leftarrow sort(workerSet, cmp, DECSENDING)$;
    $th \leftarrow 0$;
    for ( $k = 0$ ;
    $k < workerSet.size$ AND $k < n_i$ AND
    $th < 1/w_i.execTime[j_i]$; $k \leftarrow k + 1$) do
        $th$ += $1/workerSet[k].execTime[j_i]$;
    end
    $result[j_i] \leftarrow workerSet[0...i]$;
    $workerSet$.remove(0,$k$)
end
return $result$;
```

refers to the job deadline, job priority, and the number of tasks in j_i, respectively The workers are modeled as a vector of estimated execution time of each submitted job: $s_k = (T_1^k, T_2^k, \ldots)$, similar to the previous policy.

2) Algorithm: A job should be finished before its deadline. Intuitively, for the same job, if the deadline is twice tighter, it requires twice as many workers to meet that deadline. More generally speaking, the deadline and the workload are in reciprocal relationship. This implies that an worker allocation S to job j_i meeting the deadline is equivalent to

$$\sum_{s_k \in S} \frac{1}{T_{j_i}^k} \geq \frac{1}{d_i}$$

With this observation, only a little modification on the workload-based policy is needed to apply to this model: simply use the inverse of deadline as throughput.

First, we sort all the jobs according to their priority in descending order. Starts from the job with the highest priority J_m, sort the remaining workers according to their throughput th_m to J_m. Then assign the top k workers that can finish J_m before it deadline, and remove assigned workers from the list. The job is also removed from the list waiting for assignment. Repeat this process until there are no idle worker or un-assigned job. The complete algorithm is shown as algorithm 4.

IV. Experiment

A. Experimental Settings

Our experimental environment consists of one master node and ten working nodes. The master node is a physical machine with two quad-core CPU. The CPU model is Intel(R) Xeon(R) CPU X5570 @ 2.93GHz. On the other hand, each working node is a single-core physical machine. The memory sizes are 1.5GB and 768MB, respectively. Our management system works on the master node. The master node also serves as a client which generate the workloads. There are two workers on each working node.

We conduct a trace-based simulation to demonstrate the efficiency of our system. The trace we use is the *CERIT-SC workload log*, which is provided by the CERIT-SC and the Czech National Grid Infrastructure MetaCentrum. The data sets, which contains 17,900 jobs, are generated from TORQUE traces during the first 3 months of the year 2013. We take samples from these eighteen-thousand jobs, and scale the waiting and execution time of the sampled jobs. The sample rate is XXX.

During simulation, the client starts new jobs according to the arrival time and execution time from the trace. Since the actual workloads is not available, a worker will be set to “sleep mode” after receiving a task from the client. The sleeping duration is equal to the task execution length. After resumed from the sleeping mode, the worker sends a message to the management system indicating it has finished a task.

B. Experimental Results

V. Conclusion

Many enterprises or institutes are building private clouds by establishing their own data center. In such data centers, the physical machines can be different due to annual upgrades, but the amount of machine are fixed for most of the time. In such heterogeneous environment, scheduling jobs with different resource requirements and characteristic in order to meet different timing constraints is important.

In this paper, we proposed a cloud resource management framework to dynamically adjust the number of computation nodes for every job in the system. This framework make decisions according to some specified policies. The proposed framework can work as an individual cloud computing system, or as extension components of an existent cloud system. Our experiment results demonstrate that ...

References

[1] B. J. Nelson, "Remote procedure call," Ph.D. dissertation, Pittsburgh, PA, USA, 1981, aAI8204168.

[2] "Java parallel processing framework," http://www.jppf.org/.

[3] C.-E. Yen, J.-S. Yang, P. Liu, and J.-J. Wu, "Roystonea: A cloud computing system with pluggable component architecture," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec 2011, pp. 80–87.

[4] "Workflow management coalition," http://en.wikipedia.org/wiki/Workflow_Management_Coalition.

[5] G. Henry, "The unix system: The fair share scheduler," *AT T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1845–1857, Oct 1984.