2015 International Conference on Cloud and Autonomic Computing

A Job Dispatch Optimization Method on Cluster and Cloud for Large-scale High-Throughput Computing Service

Jieun Choi, Theodora Adufu, Yoonhee Kim
Dept. of Computer Science
Sookmyung Women's University
Seoul, Republic of Korea
{jechoi1205, theodora, yulan}@sookmyung.ac.kr

Seoyoung Kim, Soonwook Hwang
National Institute of Supercomputing and Networking
KISTI
Daejeon, Republic of Korea
{sssyyy77, hwang}@kisti.re.kr

***Abstract*—Cloud technologies, clusters and grids have actively supported large-scale scientific computing over the years. Whereas these technologies provide unlimited computing resources, combining them with the existing infrastructures to effectively support demanding scientific applications is more and more laborious. In this paper, we design a service architecture and propose an algorithm to optimize job distribution on a cluster and a cloud using *HTCaaS*. *HTCaaS* is a pilot job-based multi-level scheduling system for large-scale scientific computing in Korea. In addition, we present a newly added cloud module on HTCaaS which is based on OpenStack. We implement and validate the algorithm in HTCaaS. A preliminary experiment is also conducted to find an optimal distribution ratio for CPU-intensive jobs and I/O-intensive jobs in our cloud and cluster environments. We compare our method to a baseline approach which distributes tasks in proportions of the number of cores each resource has in order to validate the proposed job dispatch optimization method. Experimental results show that the proposed method can improve throughput and match tasks to appropriate resources using adaptive job distribution ratio in cloud and cluster environments.**

***Keywords*-job scheduling optimization; cloud computing; high-throughput computing; distribution ratio;**

I. Introduction

Cloud computing is an innovative technology for dynamic provisioning of shared resources over the Internet. It also presents users with the ability to rent computational resources from its unlimited pool for use in high-throughput computing (HTC), high performance computing (HPC) as well as for many task computing (MTC) in the scientific application fields of study. Meanwhile, in every distributed computing system, the importance of scheduling methods is evident. A variety of studies have been proposed to dynamically and effectively support scientific applications with execution environments that hide the complexities of distributed infrastructures. However, properly scheduling such scientific applications to heterogeneous computing resources in order to achieve good performance or low cost is still challenging because performances vary depending on where the applications are executed.

In this paper, we design a service architecture and propose a job scheduling optimization method to take account of metrics in a distributed environment which involves cluster and cloud resource usage. This method controls a ratio of scheduling distribution for distributed environments according to the characteristics of a given application and its current workload. The ratio of scheduling distribution is adaptable to dynamic computing environments. In addition, our method considers as a policy, two metrics which are performance and QoS. Both metrics are significantly considered for modern large-scale scientific applications, which request high-throughput computing services such as events generation in high-energy physics and automated docking tools. The proposed method is evaluated on HTCaaS [1], a pilot job-based multi-level scheduling system for large-scale scientific computing in Korea. We conducted a preliminary experiment in order to find an optimal distribution ratio for CPU-intensive and I/O-intensive jobs in our cloud and cluster environments. To verify the proposed job dispatch optimization method which considers application types, we compare the method to a core-based method which distributes tasks in proportions of the number of cores each resource has. Experimental results show that it can improve the overall performance by up to 21%.

The rest of this paper is structured as follows: Section II discusses related works and Section III introduces the service architecture model. We discuss our proposed algorithm in detail in Section IV, while Section V presents experiment and its results. Finally, we conclude this paper in Section VI.

II. Related Work

In distributed computing systems, there are many studies that have considered diverse scheduling algorithms for large-scale High Throughput Computing (HTC) applications such as scientific computational applications over distributed middleware systems. Such research on job scheduling algorithms are carried out according to different policies such as performance, priority and QoS. Falkon [2] is an one of the middleware systems that takes on multi-level scheduling mechanisms similar to HTCaaS which used in this paper and described in section V-A. Falkons' scheduler use a data diffusion approach by adding a data-aware scheduler [3].

Ioan et al. [2] consider only the next-available policy, which dispatches each task to the next available resource. It can reduce task dispatch time by using a streamlined dispatcher.

Muthuvelu et al. [4] propose a method to optimally map the tasks to the resources based on the QoS which is referred to as advance QoS planning. The task-resource mapping is performed prior to the task group deployment in order to maximize the processed task count within the QoS using the advance QoS planning. Lee et al. [5] present good performance and fairness simultaneously by adopting progress share as a share metric in a cost-effective manner. Choudhary et al. [6] propose new scheduling strategy that grouped task schedules according to greedy and priority based scheduling. The prioritization determine the tasks priority depending on cost and deadline. However, these papers are unable to provide sufficient methods for specific applications which have large computing tasks since they do not consider current system workload and application specifications. Moschakis et al. [7] use Gang Scheduling strategies, AFCFS and LJFS under the notion of Cloud Computing. It is proved that, both algorithms can be efficiently applied in an environment with a non-static number of VMs. While both algorithms provide similar performance for medium workloads LJFS outperforms AFCFS when workloads get heavier. The papers however, lack application to a real heterogeneous distributed computing system since they used simulation and considered only one platform.

In this paper, we propose a job scheduling optimization method which takes policy into account in our distributed environment. In addition, we take into consideration other metrics such as application types and current workloads. We control a ratio of scheduling distribution for distributed environment according to applications characteristics and current workload in order to optimize scheduling.

III. Service Model

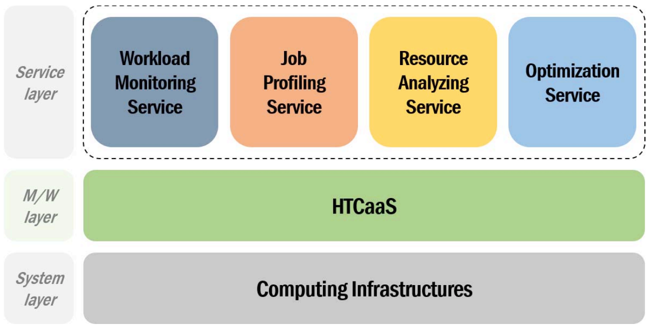


Figure 1: Service Architecture Model

Figure 1 presents an overview of a service architecture model for a hybrid distributed computing infrastructure. It basically consists of three layers and four major service modules which efficiently manage and control the dispatch of jobs above a middleware layer in various computing infrastructure.

The lowest layer of this model is characterized by the physical computing resources on top of which the infrastructure is deployed. Resources can have a distinct nature like cluster, grid and cloud, etc. The infrastructure is controlled and managed by middleware layer which aims to harness the computing resources at best. In this paper, we adopt *HTCaaS* as a middleware solution which is a pilot job-based multi-level scheduling system leveraging various distributed computing resources. We explain *HTCaaS* in more details in Section V-A. In order to support advanced services, the middleware solution needs additional features such as application profiling, optimization for job scheduling and so on. We model our solution according to four major services: Workload Monitoring, Job Profiling, Resource Analyzing and Optimization services.

Workload Monitoring Service continuously monitors overall workloads of the system, and determines an optimized distribution ratio whilst taking into account both dynamically changing workload status and application types. This enables the system to improve utility and capacity of resources. ***Job Profiling Service*** identifies the types of applications which are used for determining the distribution ratio. It also maintains and manages records of tasks the system can refer to later. ***Resource Analyzing Service*** analyzes and maintains information of diverse platforms and resources. The data enables the system to scale resources dynamically during the analysis of the characteristics of resources. In this paper, we exploit two types of computing resources which are cluster and cloud resources. Lastly, ***Optimization Service*** mainly manage where the jobs are carried out according to the distribution ratio set by the *Workload Monitoring Service*. It also controls a degree of over-provisioning metric(*i.e.*, `vCPU/pCPU` ratio) considering a policy such as *performance* or *QoS*, etc.

IV. Algorithm

Usually, a job dispatch process on heterogeneous distributed environments can be generalized into the following four stages in accordance to the above-mentioned service architecture model (Figure 1): *'Monitoring Workloads'*, *'Identifying Applications'*, *'Selecting Resources'* and *'Dispatching Jobs'*.

- *Monitoring Workload:* observes workloads of overall resources and collects status information.
- *Identifying Applications:* identifies the application type; whether it is CPU-intensive, I/O intensive, etc.
- *Selecting Resources:* selects a target resource based on Service-Level-Agreement (SLA) or machines' specification.
- *Dispatching Jobs:* manages job distribution to resources according to various application characteristics in order to optimize overall throughput.

The key notations used in the algorithms are listed in Table I. $\mathbb{M}$ corresponds to a set of metajobs. D refers to a resource distribution ratio and D with subscript(`cpu` or `i/o`) denotes an application-specific ratio. R_{ji} stands for any resource where $task_i$ of metajob *j* will be carried on. In this paper, it represents a location of an agent on which the task should be deployed, as harnessed by HTCaaS which is a pilot-based mechanism. λ represents a threshold of current workload and is calculated by averaging the number of tasks from m number of recent metajobs. $W_{resource}$ indicates the number of waiting tasks in the specified resource. The rest of notations will be explained later.

In all algorithms of this paper, we assume that the computing resources are limited to local cluster and private cloud resources, since we conducted experiments on both infrastructural types. In addition, it is also assumed in this paper that there are only two types of applications which are *CPU* and *I/O* intensive applications.

Table I: Notations

Notation	Description
$\mathbb{M}$	A set of MetaJobs
D_{cpu}	Resource distribution ratio(of cluster) for CPU-intensive job
$D_{i/o}$	Resource distribution ratio(of cluster) for I/O-intensive job
R_{ji}	Resource which schedule $task_i$ of Metajob *j*
λ	Threshold of workload
k	Value to adjust the resource distribution ratio
$W_{resource}$	The number of waiting jobs in *resource*
$r_{v/p}$	Ratio of *vCPU* to *pCPU*
ER	Error counts
ε	Threshold of error count
C	Maximum capacity

A. Job Dispatch Optimization

Algorithm 1 describes an optimization of job dispatch for jobs which come into the system. Each time new requests to submit one or several metajobs are received, Algorithm 1 is executed to compute its optimized distribution ratio. It requires two parameters that are, a set of metajobs and policy. A metajob has properties which include the type of the metajob and the number of subjobs it has. Policy can be either one of performance or QoS. Performance (*e.g,* throughput), which allows the system to exploit the maximum capacity of cores in Cloud via over-provisioning, is a default policy in this paper. QoS, on the other hand, is the policy where a user prefers a stable status(low error rate) rather than performance and so leads to an adjustment of the over-provisioning metric if errors occur.

$$r_{v/p} = \frac{vCPU}{pCPU}, \ 0 < r_{v/p} \leq C \tag{1}$$

Once a set of metajob is submitted to the system, Algorithm 1 identifies the policy(lines 2-6). If *performance*, it allows the system to use all available cores with over-provisioning (namely in the mode of that vCPU/pCPU is maximum). Otherwise, it adjusts vCPU/pCPU rate according to the count of error occurred during running time. The ratio, $r_{v/p}$, is obtained from Eq. 1 which means the number of virtual cores mapping onto a physical core and where C refers to the maximum amount of vCPU that cloud solution serves. For example, if the cloud solution is OpenStack C can be 16, since it provides 16 virtual CPU cores as a default maximum ratio. C can vary relying on which resources are based on, because it is also affected by size of disk or memory as well as available physical CPU.

Algorithm 1 Job Dispatch Optimization Algorithm

```
Input: a set of MetaJob $\mathbb{M}$, policy $p$
 1: MetaJob j= ($jobtype$ T, # $of\ tasks$ N)
 2: if $p$ == Performance then
 3:     $r_{v/p}$ = $C$ ;
 4: else if $p$ == QoS then
 5:     Control $r_{v/p}$
 6: end if
 7: for each $j$ in $\mathbb{M}$ do
 8:     Switch ($T$)
 9:      case CPU-intensive:
10:        $D_{cpu} \leftarrow$ getRatio(T);
11:        $R_{j[0:N*D_{cpu}-1]} = Cluster$;
12:        $R_{j[N*D_{cpu}:N-1]} = Cloud$;
13:      case I/O-intensive:
14:        $D_{i/o} \leftarrow$ getRatio(T);
15:        $R_{j[0:N*D_{i/o}-1]} = Cluster$;
16:        $R_{j[N*D_{i/o}:N-1]} = Cloud$;
17:      default:
18:        T= detectJobType;
19:     end switch
20: end for
21: ($D_{cpu}$,$D_{i/o}$) = MonitorWorkload();
Output: Resource select decision RS
        = {(MetaJob j,$R_{ji}$) | i = 0,1,$\cdots$, N−1,
           $R_{ji} \in (Cluster, Cloud, ...)$ }
```

The whole process to adjust vCPU/pCPU rate is illustrated in Figure 2. If ER, which is the total count of error occurred during runtime such as VM failure, is greater than an error threshold(ε) and the ratio($r_{v/p}$) is not the minimum value, the system decreases the ratio by 1. In contrast, if the condition is false, $r_{v/p}$ increases repeatedly till C.

After, it classifies each metajob in $\mathbb{M}$ into cpu and i/o-intensive jobs(line 7-20). For each case, tasks belonging to the metajob are assigned to the target resource through '***Monitor Workload()***'(line 21), on the basis of D_{cpu} or $D_{i/o}$. That is, if the type of metajob *j* is CPU-intensive, it dispatches all jobs to cluster and cloud according to D_{cpu} by getting the distribution ratio from workload monitor modules(lines 9-12). If it belongs to I/O-intensive, it sends all jobs to cluster or cloud according to $D_{i/o}$ by getting the distribution ratio from workload monitor modules(lines 13-16).

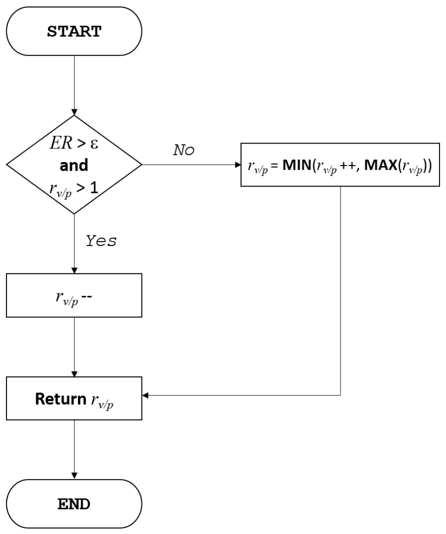

START
$ER > \varepsilon$ and $r_{v/p} > 1$
No
$r_{v/p}$ = MIN($r_{v/p}$++, MAX($r_{v/p}$))
Yes
$r_{v/p}$ --
Return $r_{v/p}$
END

Figure 2: Flow Chart of Over-provisioning ratio adjustment

Suppose that a metajob which has the following parameters, `(cpu, 10000)`, is submitted to the system and that D_{cpu} is `0.4`. The scheduler dispatches the tasks to *Cluster* at a rate of D_{cpu}`(0.4, 40%)` which is given by *getRatio(*`cpu`*)* and 60% tasks to *Cloud*. After that, it monitors and updates ratio **D**(if necessary) depending on current workloads. We deal with further details in the next subsection.

B. *Workload Monitoring*

Algorithm 2 Workload Monitoring Algorithm

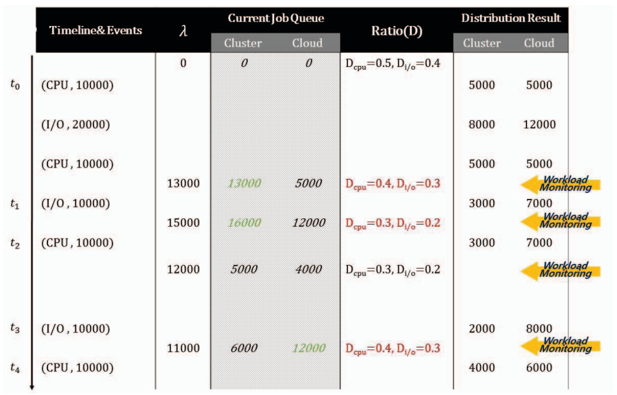
```
 1: $\lambda$ = threshold($m$);
 2: if $W_{cluster} \geq \lambda$ and $W_{cloud} \geq \lambda$ then
 3:     $D_{cpu} = initializeRatio()$;
 4:     $D_{i/o} = initializeRatio()$;
 5: else if $W_{cluster} \geq \lambda$ then
 6:     $D_{cpu} = D_{cpu} - k$;
 7:     $D_{i/o} = D_{i/o} - k$;
 8: else if $W_{cloud} \geq \lambda$ then
 9:     $D_{cpu} = D_{cpu} + k$;
10:     $D_{i/o} = D_{i/o} + k$;
11: end if
Output: $SetRatio(D_{cpu}, D_{i/o})$
```

In this system, a '*Workload Monitoring*' service returns a ratio of scheduling distribution with regards to the job type and current workload threshold(λ). λ is defined as the average number of tasks from m number of recent metajobs as follows (Eq. 2).

$$threshold(m) = \begin{cases} 0 & \text{if } n = 0 \\ \dfrac{\sum_{i=n-m+1}^{n} |MetaJob_i|}{m} & \text{if } n > 0 \end{cases} \tag{2}$$

where m is the number of recent metajobs and n is the last id of metajobs submitted.

If workloads of both cluster and cloud exceed λ, then the distribution ratio will be initialized (lines 2-4). This is because, we consider the current ratio unfeasible. Note that initial ratio is determined through an empirical measurement. If only cluster's workload is greater than the threshold, λ, then both D_{cpu} and $D_{i/o}$ decreases by k(lines 5-7). It is because the D refers to the proportion by which to dispatch the tasks to cluster. Otherwise, if only that of cloud is greater than the value of λ then both D_{cpu} and $D_{i/o}$ increases by k(lines 8-10). Here we set the value of k as `0.1`.



	Timeline& Events	λ	Current Job Queue		Ratio(D)	Distribution Result		
			Cluster	Cloud		Cluster	Cloud	
		0	*0*	*0*	D_{cpu}=0.5, $D_{i/o}$=0.4			
t_0	(CPU , 10000)					5000	5000	
	(I/O , 20000)					8000	12000	
	(CPU , 10000)					5000	5000	
		13000	*13000*	*5000*	D_{cpu}=0.4, $D_{i/o}$=0.3			Workload Monitoring
t_1	(I/O , 10000)					3000	7000	
		15000	*16000*	*12000*	D_{cpu}=0.3, $D_{i/o}$=0.2			Workload Monitoring
t_2	(CPU , 10000)					3000	7000	
		12000	*5000*	*4000*	D_{cpu}=0.3, $D_{i/o}$=0.2			Workload Monitoring
t_3	(I/O , 10000)					2000	8000	
		11000	*6000*	*12000*	D_{cpu}=0.4, $D_{i/o}$=0.3			Workload Monitoring
t_4	(CPU , 10000)					4000	6000	

Figure 3: An Example of the Optimization Algorithm

Figure 3 shows an example where multiple metajobs having various conditions are arriving at different times. In this example, initial value of λ would be zero, since there was no previous metajob before time t_0(Eq. 2). Suppose that ratios D for `cpu` and `i/o` are 0.5 and 0.4, respectively, and three metajobs are submitted sequentially at time t_0. It is assumed that users choose the default policy in this example, as well. The system starts to distribute tasks of the metajobs onto the existing resources which are cluster and cloud, in a five to five ratio for cpu tasks and in a four to six ratio for i/o tasks. Right after submitting three metajobs, 'monitoring workload' service would be performed. At that time, λ results in 13000 by Eq. 2 and ratios are updated to 0.4 and 0.3 (decreased by 0.1), since $W_{cluster}$ is equal to λ. Hence, I/O-intensive job would be scheduled into the resources at a two to eight ratio at t_1. After t_1, in the same manner, D would be updated until that the system is in stable status. If the workload of cloud is over λ, after t_3, the distribution ratios increase by 0.1. In this way, it can adapt to current status of dynamic environments and contribute to making the overall system stable.

V. Experiment

Experiments that validated our scheduling optimization methods are presented in this section. First, we describe the system and target applications, and subsequently present the experimental setting along with results.

A. *HTCaaS*

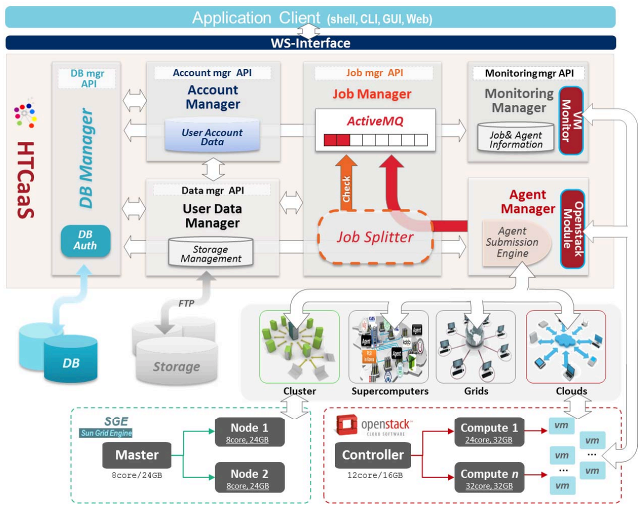


Figure 4: HTCaaS Architecture

HTCaaS[8] aims to facilitate exploring large-scale and complex HTC or MTC problems by leveraging various computing resources such as Supercomputers, Grids, Clusters and Clouds. It can hide heterogeneity of integrating different resources from users, and allow users to efficiently submit a large number of jobs at once. This system also helps researchers to solve large-scale and complex scientific problems efficiently by providing them with hybrid resources managed by many kinds of local resource managers (such as PBS[9], HTCondor[10], LoadLeveler[11], SGE[12] and gLite[13]) systematically. It adopts an agent-based(*or* pilot-based) multi-level scheduling mechanism which supports the decoupling of resource allocation from resource binding.

Since it adopts agent-based multi-level scheduling and streamlined job dispatching (as described in Falkon[2], condor glide-in[14]), a first-level request to a batch scheduler (e.g., Load Leveler and Condor in PLSI [15] Supercomputers, gLite for Grids, PBS for Amazon EC2[16]) reserves resources by submitting agents as batch jobs and then each agent pro-actively pulls the user tasks from the job manager which implements the lightweight and fast job dispatching mechanisms.

By employing this mechanism consistently across heterogeneous computing resources, HTCaaS can effectively form a dedicated resource pool on-the-fly for fast dispatching of many tasks to circumvent the performance bottleneck of traditional batch schedulers. The general agent itself is a regular batch job which is submitted by HTCaaS system and is assigned into the resources by the local batch scheduler. Then it performs '***pulling and executing***' sub-jobs as well as coordinating the launch and monitoring processes. It avoids the necessity to queue each sub-job and as a consequence, leads to the efficient utilization of resources as well as reducing the time of completion.

In order to support private cloud platform, OpenStack module is recently added to HTCaaS as shown on Figure 4. In the OpenStack module, each virtual machine core corresponds to an agent as mentioned above. Thus once a virtual machine is launched, the agent in the VM starts to pull & launch the task it is given. Our proposed algorithm is mainly communicated with *Agent Manager* module.

B. *Empirical Results*

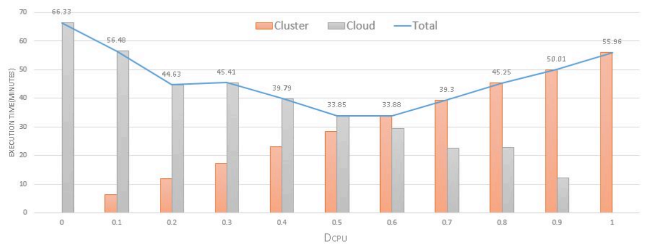
In our proposed algorithms, the initial distribution ratios are requisites and can be decided by empirical finding, as previously mentioned. In this section, we show the empirical findings for the optimized job distribution ratios from our cluster and cloud computing infrastructures. Figure 5 and 6 show the experimental results of the mean elapsed time of tasks which are distributed by varying distribution rates ranging from 0 to 100%(denoted as `0, 0.1, .. 0.9, 1.0` in graph). Figure 5a and Figure 6a are the results for which system has exploited full capacity of processors in order to consider the policy, '*Performance*'. Figure 5b and Figure 6b are the results when system has set $r_{v/p}$ as 1 for the policy '*QoS*'. The rate having the lowest elapsed time through experiment would be the initial distribution ratio. In this case, D_{cpu} can be `0.5` for *performance* and for *QoS*, while $D_{i/o}$ is `0.4`, respectively. For these experiments, we ran multiple thousand(10^3) metajobs having thousands(10^3) of subjobs independently. In these experiments, we set C to 2.

C. *Target Application*

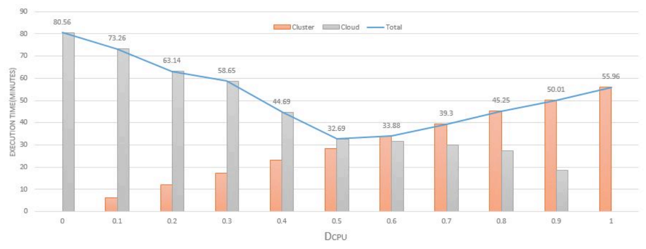
We target High Throughput Computing(HTC) and Many Task Computing(MTC) application which usually have millions or billions of tasks to be processed with relatively *short* per task execution time. This application paradigm embraces a wide range of scientific domains such as high-energy physics, pharmaceutics, chemistry and so on. For our experiment, we use two applications, PYTHIA and Autodock to represent CPU and I/O intensive jobs, respectively.

PYTHIA[18] is a standard tool for Monte Carlo(MC) simulations, written in fortran(and C++), for events generation in high-energy physics. It comprises a coherent set of physics models as a library and also has a set of utilities and interfaces to external programs. PYTHIA is mainly a CPU-intensive application with small size of in/output files.

AutoDock[19] is a suite of automated docking tools to predict how small molecules, such as substrates or drug candidates, bind to a receptor of known 3D structure. The

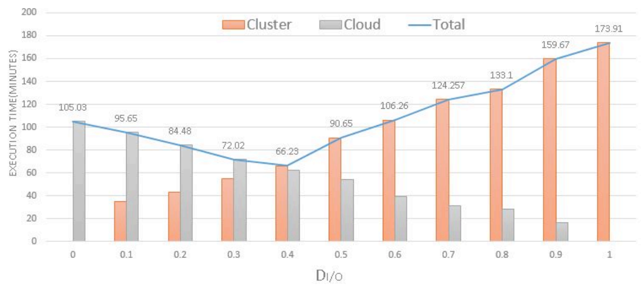


(a) Performance-based policy

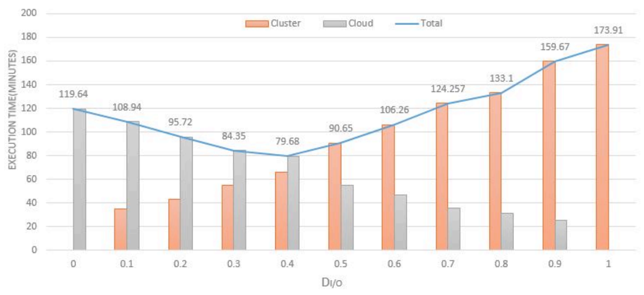


(b) QoS-based policy

Figure 5: Empirical results for CPU-intensive jobs



(a) Performance-based policy



(b) QoS-based policy

Figure 6: Empirical results for I/O-intensive jobs

AutoDock tool is used for performing the docking of ligands to a set of target proteins in order to discover potential new drugs for several serious diseases such as SARS, Malaria. The AutoDock job we use is considered an I/O-intensive application having in/output data files with small sizes of memory usage.

D. Experimental Settings & Results

The experiment environments consist of local cluster and private cloud resources using HTCaaS. The local cluster uses a Sun Grid Engine(SGE) [12] which is a batch-queuing system supported by Sun Microsystems and later Oracle. OpenStack [17] is an open source software that provides large pools of compute, storage and networking resources used for cloud environments. As an infrastructure management system, maximizing resource usage is a basic requirement for OpenStack. Our OpenStack environment consists of 1 Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz Controller with 12 cores of CPU and 16GB of RAM and 1 Inter(R) Core(TM) i7-4930K CPU @ 3.40GHz compute node with 8 CPU cores and 26GB RAM and 2 Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz compute nodes with each 24 and 32 CPU cores and 32GB RAM. Each VM was created identically using Ubuntu 12.04 Server image with 1GB ram and 1 vCPU.

For the performance evaluation, we use a simple pattern of workload consisting of uniform and bursty patterns as shown on Figure 7. It can represent a common environment for large-scale computing service having unexpected and sudden overloads for several times.

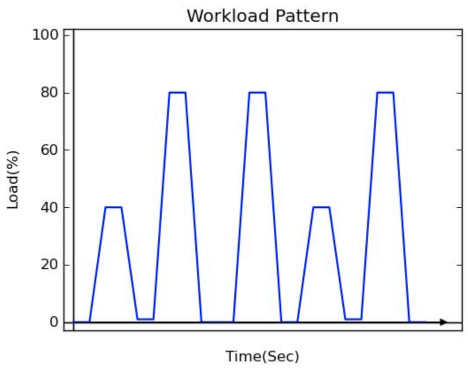
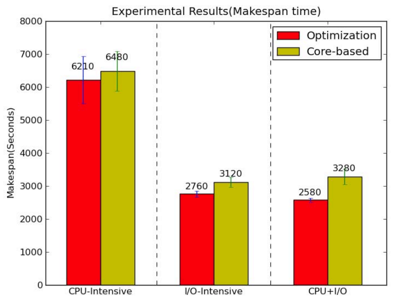
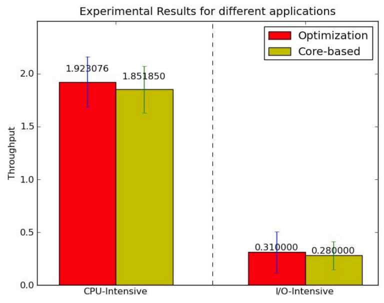


Figure 7: Workload Pattern

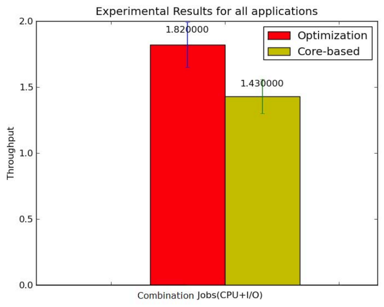
We compare the proposed job dispatch optimization method which is represented as optimization in the graph, to a baseline policy through the evaluations. The baseline policy which is means as core-based in the graph is that system distributes tasks in proportions of the number of cores each resource has. With the addressed conditions, results of the evaluation are as follows: Figure 8 depicts the comparative results in terms of makespan time and throughput. First of all, Figure 8a shows the comparisons of our method to the baseline with respect to average makespan



(a) Makespan time for different jobs



(b) Throughput for different jobs: CPU-intensive(left), I/O-intensive(right)



(c) Throughput for combination jobs

Figure 8: Experimental results

time of metajobs. The results show there are overall improvements in performance by reducing makespan time for each application. In Figure 8b, the proposed methods can improve the throughput by about 3.8 percent for CPU-intensive jobs and about 9.6 percent for I/O-intensive jobs. Therefore, our job dispatch optimization method is a suitable method, when the system dispatches jobs according to job distribution ratio. In the case of a combination of two jobs as shown in Figure 8c, our optimization method is better than the core-based method by about 21.4 percent. Especially, in the case of multiple applications, our method results in more noticeable improvement to each individual application.

VI. Conclusion

We design a system architecture and propose a job scheduling optimization method which takes account of metrics (e.g., performance, QoS) in cluster and cloud environments. This method controls a ratio of scheduling distribution for distributed environment according to characteristics of application and current workload. The ratio of scheduling distribution is adaptable to dynamic computing environments. We conducted a preliminary experiment in order to find an optimal distribution ratio for CPU-intensive job and I/O-intensive job in our cloud and cluster environments. Our approach is compared with baseline approach. According to the experimental results, it can improve the overall performance by approximately 21%.

In the near future, we are going to extend our algorithms by adding an application profiling service which can identify the type of jobs ingeniously. We will perform additional experiments using a variety of realistic workload patterns and diverse kinds of applications as well.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2013R1A1A3007866)

References

[1] HTCaaS Wiki : http://htcaas.kisti.re.kr/wiki/ (2014)

[2] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, Falkon: a Fast and Light-weight tasK executiON framework, in Proceedings of the 2007 ACM/IEEE conference on Supercomputing (2007)

[3] M. Feller, I. Foster, and S. Martin, GT4 GRAM: A Functionality and Performance Study, in Proceedings of the TERAGRID 2007 Conference, Jun. 2007.

[4] Nithiapidary Muthuvelu, Ian Chai, Eswaran Chikkannan and Rajkumar Buyya, QoS-based Task Group Deployment on Grid by Learning the Performance Data, Journal of Grid Computing, Vol. 12, No. 3, pp. 465-483, September (2014)

[5] Gunho Lee, Byung-Gon Chun and Randy H. Katz, Heterogeneity-Aware Resource Allocation and Scheduling in the Cloud, in Proc. of 3rd USENIX Workshop on Hot Topics in Cloud Computing, June (2011)

[6] Monika Choudhary, Sateesh Kumar Peddoju, A Dynamic Optimization Algorithm for Task Scheduling in Cloud Environment, International Journal of Engineering Research and Applications, Vol. 2, No. 3, pp. 2564-2568, May-June (2012)

[7] Ioannis A. Moschakis, Helen D. Karatza, Evaluation of gang scheduling performance and cost in a cloud computing system, The Journal of Supercomputing, Vol. 59, No. 2, pp. 975-992, February (2012)

[8] Jik-Soo Kim, Seungwoo Rho, Seoyoung Kim, Sangwan Kim, Seokkyoo Kim, and Soonwook Hwang, HTCaaS: Leveraging Distributed Supercomputing Infrastructures for Large-Scale Scientific Computing, ACM 6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS'13) held with SC13, November (2013)

[9] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson, The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters, in Proceedings of the Usenix, Proceedings of the 4th Annual Linux Showcase & Conference (2000)

[10] HTCondor, http://research.cs.wisc.edu/htcondor/

[11] IBM Tivoli Workload Scheduler LoadLeveler, http://www-03.ibm.com/systems/software/loadleveler/

[12] Gentzsch, W., Sun Grid Engine: towards creating a compute power grid, Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on , vol., no., pp.35-36 (2001)

[13] gLite - Lightweight Middleware for Grid Computing, http://glite.cern.ch/

[14] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, Condor-G: A Computation Management Agent for Multi-Institutional Grids. Cluster Computing, 5(3):237246, July (2002)

[15] Partnership and Leadership for the nationwide Supercomputing Infrastructure, http://www.plsi.or.kr/

[16] Amazon EC2(Elastic Compute Cloud), http://aws.amazon.com/ec2

[17] OpenStack, http://www.OpenStack.org

[18] T. Sjstrand, S. Mrenna, P.Z. Skands, PYTHIA 6.4 physics and manual, 050262006

[19] Autodock, http://autodock.scripps.edu/