A SLA driven VM Auto-Scaling Method in Hybrid Cloud Environment

Hyejeong Kang Jung-in Koh Yoonhee Kim
Dept. of Computer Science
Sookmyung Women's University
Seoul, Korea
{hjkang, jungin, yulan}@sookmyung.ac.kr

Jaegyoon Hahm
Korea Institute of Science and Technology Information
Daejeon, Korea
jaehahm@kisti.re.kr

***Abstract*—The advent of Science Clouds enables scientists to facilitate large-scale scientific computational experiments over cloud environment besides specialized supercomputers in diverse science domains. Cloud computing service elicits efficiency on on-demand resource usage and timely execution at any given time depending on experimental requirements. Hybrid clouds, composing of private and public clouds, even extend research opportunities on resource selection for further complicated experiments but increase the needs of dynamic resource management to maximize its utilization. At existing public cloud providers for commercial use, rule-based and schedule-based mechanisms have been tried for automatic resource allocation to provide resources for processing dynamic workload of modern applications. However, most of the auto-scaling methods just simply support performance metric such as CPU utilization but rarely are aware of Service Level Agreements (SLA) including execution deadline or cost. In this paper, we propose an auto-scaling method that automatically allocates resources depending on variable resource requirements in hybrid clouds satisfying a user's requirements on SLA. We present experimental results which show that the proposed auto-scaling can minimize SLA violations and acceptable cost if needed.**

Keywords—auto-scaling, hybrid cloud computing, SLA, multi-policies;

I. INTRODUCTION

The advent of Science Clouds enables scientists to facilitate large-scale scientific computational experiments over cloud environment besides specialized supercomputers in diverse science domains. Cloud computing enables applications to exploit on-demand and scalable resources. Computational scientific applications (e.g. scientific workflow) and big data processes can be classified as many task computing (MTC) requiring high performance computing (HPC) or high throughput computing (HTC). It is essential for such applications to supply high performance resources in a long term and guarantee stable executions of applications. In these days, as the needs of high performance computing increase, it leads the advent of exa-scale computing or peta-scale computing. Hence, it is getting important to study computational problem solving environment which provides the management of job executions or resources in the large scale of computation. Auto-scaling with virtualization enables efficient utilization of cloud resources for the computational problem solving environment. Technically, auto-scaling can dynamically increase or decrease the number of VMs (Virtual Machine) during execution of an application and a notable example is "Auto-scaling" of AWS (Amazon Web Service) [1]. Auto-scaling services like AWS or Scalr [2] are based on user-defined rules (e.g. when the utilization of CPU is more than 60%, add 2 VMs) when deciding scale of resources. Although the rule-based auto-scaling methods are simple, it is difficult for them to satisfy dynamic patterns of resource requirements caused by variable workloads of modern applications. Especially, if resource requirements of an application cannot be fulfilled, this would result in SLA (Service Level Agreement) [3] violation: deadline violation, growth of execution failure and so on.

In this paper, we propose an auto-scaling method to provide efficient resource utilization in a hybrid cloud computing environment. The proposed auto-scaling meets SLAs such as deadline, cost-oriented or performance-oriented policies. It is also available to auto-scale in economic way by considering both characteristics of applications (computation-intensive, data-intensive or I/O-intensive) and instance types of cloud resources for commercial use (public cloud). Based on Amazon EC2 instance types, in case of that two different instances (e.g. High-CPU type and Standard type) have same CPU resources, CPU-intensive type has lower price with the same execution time on computation-intensive jobs. The main assumptions that are implied in this paper are that (1) jobs of an application have no dependencies; these jobs are able to do parallel process for grand scale job from their Bag-of-Tasks type [4], and that (2) jobs, once started, should not be moved to a different virtual resource, so only queued jobs are adaptively scheduled during the auto-scaling phases.

The continuation of the paper is structured with the sections as follows; we introduce some related works in Section 2, and then present architecture of our auto-scaling framework. Section 4 explains auto-scaling algorithm, and Section 5 contains contents about a scenario of using auto-scaling. After that, experiment and simulation results are discussed in Section 6, while conclusion and future work are in final section.

II. RELATED WORK

Cloud computing offers unlimited resource by virtualization technology and facilitates extension and

reduction of resources. Auto-scaling issues are currently being discussed and studied as effective resource management.

"Auto-scaling" of AWS, as mentioned before, is one of existing representative auto-scaling methods. As a rule-based auto-scaling method which elastically expands or contracts resources to follow user-defined metrics, this service configures metrics related to hardware performance as criteria: CPU utilization, hard disk utilization, etc. Through setting upper bound and lower bound of a metric, VMs can be created or destroyed based upon the values. Similarly, there exist Paraleap [5] for Windows Azure [6], Scalr [2], and RightScale [7]. While rule-based auto-scaling methods are simple enough to allocate resources dynamically, it may not be possible to satisfy an amount of resources actually needed, in the case of applications which have a complex workload, then it could cause a severe problem called SLA violation (deadline, cost, and etc.).

References [8], [9] are the studies of auto-scaling considering deadline of applications or cost for resource usage. Reference [8] guarantees execution of an application within its deadline using their auto-scaling method. They respect cost as well in the auto-scaling method. However, they use only public cloud, which means that the scope of resource is limited to demonstrate performance of their auto-scaling method in a general way. In order to minimize expenses for resource usage, [9] proposes an auto-scaling method mapping optimal physical machines to resource requirements. It is combined with horizontal scaling which adds or removes VMs and vertical scaling which increases or decreases a VM size. In spite of the combination of auto-scaling methods, it is still inadequate for fulfilling resource requirements of dynamic workloads because it fixes the number of VMs created or deleted in the horizontal scaling.

In this paper, we propose an auto-scaling method which enables resource allocation considering SLA such as deadline and expenses in hybrid cloud environment. Using continuous monitoring service, it is possible to calculate the numbers of requisite VMs so that VMs can be properly added or removed whenever the need varies.

III. Service Architecture of Auto-Scaling Framework

In this section, we discuss architecture of services. Fig. 1 related to the whole architecture which supports auto-scaling in hybrid cloud environment. In order to decide how many VMs will be created or destroyed on a dynamic basis, the framework needs a SLA description from a user. An application is submitted with the SLA and queued before they are processed while the SLA goes to 'Job Metadata' module.

Four major services form auto-scaling framework as illustrated in Fig. 1. Each service has sub-modules to perform specific work for auto-scaling.

'Metadata Mgmt. Service' (MMS) maintains information of three different categories and mainly stores static information. One of the categories is 'Job Metadata' module which contains length, execution history, definition (e.g. input, output), and SLA descriptions of applications. Another is 'Resource Metadata' module. This module reserves resource specifications and the number of physical machines. The last module in MMS is 'VM Catalog'. 'VM catalog' manages specifications and templates of VMs in private cloud, with specifications and cost of instance types in public cloud.

'Auto-Scaling Service' (ASS) is core of our framework. Dynamic workload of modern applications leads service providers to have difficulties of predications on execution of jobs and their resource requirements. Because of that, all of processes of 'Auto-Scaling Service' are conducted on a regular basis, so that ASS allocates actually required resources at any given time. This service consists of three modules: 'Scheduling', 'SLA Monitoring' and 'Run-time Scaling'. 'Scheduling' module allocates jobs to VMs which fit resource requirements of jobs under the selected policy. We offer two kinds of policies. One is cost-oriented policy which is aware of resource utilization cost of public cloud environment. Through this policy, users can expense reasonable costs, while executing applications within a deadline. The other is performance-oriented policy which is aiming to finish application as fast as possible (i.e. this policy has a short deadline). If a user chooses the performance policy, a deadline which must be bigger than execution time of Critical Path (CP) and minimum performance requirement, the user requests, will be included in SLA. In the performance policy, jobs are allocated on VMs which have much more resources than minimum performance requirement. 'SLA Monitoring' module is responsible for estimation of performance and cost. Performance can be estimated by comparing the biggest EFT (Estimated Finish Time) of the last job on a VM with the deadline in SLA whether the EFT would exceed the deadline or not. 'Run-time Scaling' module decides the number of VMs to create or destroy for application execution and then gets the decisions across to 'Dynamic Resource Mgmt. Service' and 'Job Execution Service'. Scale-out which expands scale of resources through adding extra VMs is fulfilled when a resource allocation is executing, whereas scale-in is performed at going after the resource allocation.

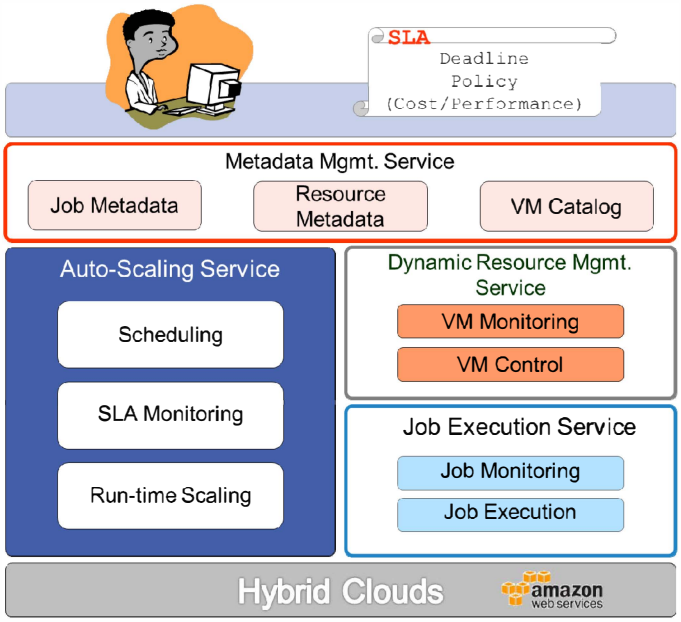

Fig. 1. Service Architecture

'Dynamic Resource Mgmt. Service' (DRMS) and **'Job Execution Service' (JES)** are the same except an object of attention. DRMS covers dynamic resource allocation observing decisions from ASS and VM monitoring. 'VM Control' module practically creates or destroys VMs when ASS commands to do so. ' VM Monitoring' module collects information: the number of VMs, the list of available resources of each VM and the list of VMs of which status is idle. In case of JES, it is interested in jobs, therefore monitors status of jobs (by 'Job Monitoring' module) and submits jobs (by 'Job Execution' module) to VMs created by DRMS.

IV. SLA DRIVEN VM AUTO-SCALING ALGORITHM

In this section, we describe how this 'Auto-Scaling Service' works and what the effect will be.

A. Assumptions

Our approach is based on some background assumptions. Most importantly, about applications, we assume that jobs of an application can be scheduled on resources independently from each other. Jobs have their own types like computation-intensive or data-intensive, and based on these types, our method can allocate cost-effective resources for each job (i.e., computation-intensive jobs to High-CPU instance type of VMs and data-intensive jobs to High-Memory instance type of VMs). Additionally, we assume that jobs, once started, should not be moved to a different virtual resource. Only queued jobs are adaptively scheduled during the auto-scaling phases. Auto-scaling method doesn't detect extension of running jobs but previous performed jobs by comparing actual start time and estimated start time (i.e. estimated finish time of previous job) of running jobs. Furthermore, monitoring is conducted on a regular basis. Monitoring intervals are static and detecting deadline violation is performed at every interval.

B. Algorithms

Our scaling algorithm is classified under three large groups: Run-time Scaling (in Algorithm 1), SLA Monitoring (in Algorithm 2), and Policy-based Scheduling (in Algorithm 3, we only describe performance-oriented scheduling). For better understanding, main notations for the algorithms are explained at the below:

- Application = $\{j \mid j_i, i=1,2,\ldots, n\}$
  : An application is consist of many jobs.
- SLA = {a policy *P*, a deadline *D* [, minimum performance requirement *minPM*]}
  : SLA of an application. It has a policy, a deadline and minimum performance requirement in case of performance-oriented policy. ([*][] means optional)
- *Cost*: The total cost for resource usage.
- IT_j
  : Instance type of public cloud which matches the job *j*.
- *vm*: Running VMs.
- EFT_{vm}: Estimated finish time **of a VM**.
- *EST*: Earliest start time **of a job** on a VM.
- *AST*: Actual start time **of a job** on a VM.
- *ET*: Execution time **of a job** on a VM.
- PM_{vm}: Performance specification of a VM.

Algorithm 1 describes our overall auto-scaling method. If *SCALING* is true (line 3), at first, jobs are sorted as descending order based on their execution time, then, sorted jobs are scheduled with applying the policy of SLA. After scheduling, the algorithm checks if there is any VM that do not have a running job or queued jobs. If so, the VMs are added to *toShutDown* list for deletion (line 13). Scaling decisions and scheduling decisions are sent to DRMS and JES respectively. SLA monitoring is conducted at every certain interval, and the monitoring returns the value *SCALING* that indicates whether scaling is necessary or not.

Algorithm 1 – Run-time Scaling
Input – An application,
SLA={a policy *P*, a deadline *D* [, minimum performance requirement *minPM*]}
Output – Scaling decision *S* = { *toStartUp*, *toShutDown* }
Scheduling decision *S* = { *jobs* → *VMs*}

```
1:  SCALING ← TRUE;
2:  while (true)
3:      if SCALING == TRUE
4:          Sort waiting jobs in decreasing order of execution length;
5:          switch P
6:              case Performance:
7:                  S ← PerformanceOrientedScheduling(
                                    sortedJobs, D, minPM); break;
8:              case Cost:
9:                  S ← CostOrientedScheduling(sortedJobs, D); break;
10:         end switch
11:         for each vm where status is running do
12:             if no running/waiting jobs on vm then
13:                 add vm to toShutDown;          //destroy the vm
14:             end if
15:         end for
            //send scaling decisions to DRMS
16:         send(DRMS, {toStartUp, toShutDown});
            //send scheduling decisions to JES
17:         send(JES, S);
18:         waitForNextInterval();
19:         SCALING ← SLAMonitoring(runningJobs, D);
20:     end if
21: end while
```

Algorithm 2 defines our SLA monitoring mechanism. In each interval, algorithm calculates *delay* of running jobs by subtracting *EST* from *AST*. Then, add *delay* to *EFT* of the VM that job *j* is running. If this value is bigger than the deadline *D* (i.e. Scale-out is required) or -*delay* bigger than threshold τ (i.e. Scale-in is required), algorithm sets *SCALING* as TRUE so that Run-time Scaling algorithm can reconfigure virtual resources pool.

Every time we try to schedule the jobs to VMs (Algorithm 3), private cloud resources (line 4) are allocated for jobs prior to public cloud to save the cost. After that, we find the *EST* of the job in each resource and added execution time to

Algorithm 2 – SLA Monitoring
Input – Running jobs of the application, the deadline D
Output – Whether scaling is required or not, *SCALING*

```
1:  for each job j do
2:      delay ← AST - EST ;
3:      if (EFT_currentVM + delay) > D or -delay > τ then
4:          return TRUE;
5:      end if
6:  end for
7:  return FALSE;
```

Algorithm 3 – Performance-oriented Scheduling
Input –Sorted jobs of the application, deadline D, minimum performance requirement *minPM*
Output –Scheduling decision $S = \{jobs \rightarrow VMs\}$, list *toStartUp*.for the VMs to be newly created

```
1:  VM ← null;
2:  toStartUp ← null;
3:  for each job j do
4:      for each private vm where PM_vm >= minPM do
5:          VM ← find a vm on which j can start the most quickly
                                                    within the D;
6:      end for
7:      if VM is not null then
8:          schedule j to vm;
9:          continue with the next job;
10:     else
11:         for each public vm where PM_vm >= minPM do
12:             VM ← find a vm on which j can start the most quickly
                                                    within the D;
13:         end for
14:         if VM is not null then
15:             VM ← ChooseCheaperVM( VM, IT_j );
16:             schedule j to vm;
17:             if VM is IT_j then
18:                 add IT_j to toStartUp;  //to create a new VM of IT_j type
19:             end if
20:             continue with the next job;
21:         else
22:             add IT_j to toStartUp;      //to create a new VM of IT_j type
23:         end if
24:     end if
25: end for
26: return S;
```

EST, so we could get the *EFT* in all resources and then schedule the job *j* to the resource which has the earliest *EFT* (line 4-9 (private), line11-20 (public)). If this allocation decision has any chance to violate deadline, public cloud resources are allocated to accelerate overall execution time by offering extra resources (line 22). When our scaling method tries to allocate public resources for jobs, it considers running VMs first. Providers have certain billing policy that users have to pay per billing time unit [10] which is relatively large. In case of Amazon EC2, billing time unit is one hour, so user must pay as much as price of one hour, even if he/she did use just 10 minutes. This is the reason why we consider running VMs before starting new one. If running VM has remaining time for billing and this VM's instance type suits for a job, it is more cost-effective than starting new one. Another effort to achieve cost-effectiveness is to take instance types of public cloud into account. Based on Amazon EC2 instance types, in case of that two different instances (e.g. High-CPU type and Standard type), have same CPU resources, CPU-intensive type has lower price with the same execution time on computation-intensive jobs. In line 15, *ChooseCheaperVM* compares the prices between allocating a task on a running VM on which the task can finish earlier than on any other running VM and allocating a task on a New VM of which type is the optimal time of the task; the cost for using VMs is charged based on billing time unit. If both costs are same, *IT* is selected for the type of newly created VM and added to *toStartUp* List.

V. SLA DRIVEN VM AUTO-SCALING SCENARIO

We explain simple scenarios for better understanding of proposed algorithms in the previous section.

Fig. 2 shows initial resource allocation of auto-scaling proposed in this paper. In this figure, job #15 which is a computation-intensive job can be scheduled to the second High-CPU type VM, but it incurs additional expenses by exceeding billing time unit. Instead of allocating matching instance type of the job, allocating unfavorable type of instance for the job can prevent extra charge and execute the job within deadline, nonetheless actual execution time of the job is extended.

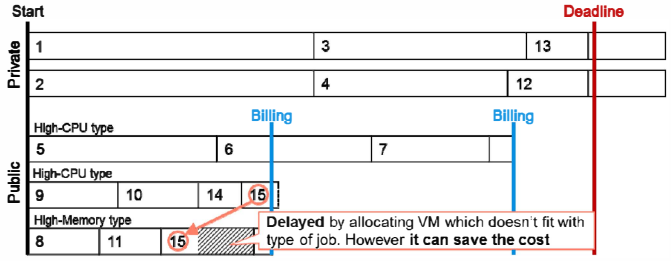

Fig. 2. Result of Initial Resource Allocation

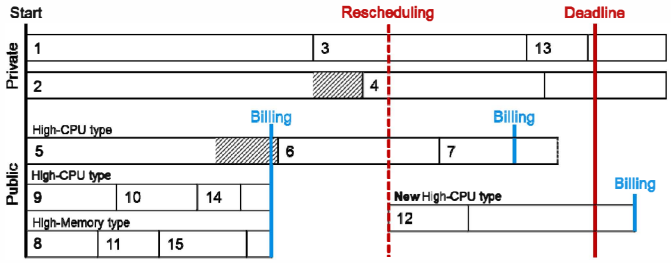

Fig. 3. Result of Monitoring & Auto-Scaling

Fig. 3 is depicted a result of auto-scaling. Our algorithm monitors execution of jobs regularly to avoid violation of SLA and to allocate resources on which an application "Really" needs to execute their jobs in time. Our algorithm compares expected finish time which calculated last monitoring interval with real finish time on current resources. If there are delays (Jobs #2, #6), the algorithm measures whether this application would violate deadline or not. As a result of the measurement,

we know that job #12 would exceed deadline. Then queued jobs (Jobs #7, #12, and #13) are newly scheduled on other VMs. Waiting jobs are sorted by execution length (i.e. #7, #12, and #13) and resources are allocated for jobs following this order. As mentioned before, we consider private cloud resources first among all VMs. Jobs #7 and #12 would violate if these are scheduled on private resource. Thus, these jobs should be executed on public resource. Job #7 is scheduled on same resource which is allocated to job #7 at last monitoring interval and job #12 is scheduled on new High-CPU type VM (i.e. Scale-out) because of violation of deadline when this job is allocated on same resource with #7. Finally, job #13 is allocated to the previous allocated VM in private cloud. After resource allocation, idle VMs are released and doing so scale-in is accomplished.

VI. EXPERIMENTS

A. Workload

Scientific applications with the HTC (High Throughput Computing) jobs are suitable for the resource allocation in this paper. These scientific applications, which are finding patterns at massive data analysis, have two features: massive job and independent execution of jobs. These applications are able to do parallel process for grand scale job from their Bag-of-Tasks type [4]. The execution time of an application usually takes from 3~4 hours to several months because the execution time of data normally takes several hours and there are more than one data that must be handled. Therefore, when using increased computing power you can get high rate of data process.

To demonstrate our auto-scaling method, we use workload of e-AIRS 2.0 [11] system for CFD [12] applications of aerodynamics running over PRAGMA [13] grid environment. The job traces have been collected for approximately 7 months. Using this traces, we recreated new job traces, which are based on both grid and hybrid cloud via CloudSim [14].

B. Simulation

We simulated with CloudSim. As we mentioned before, we used result values of the CFD application in hybrid cloud: private cloud (OpenNebula [15]) and public cloud (Amazon EC2) for preliminary values. We assumed that an application has 5,000 tasks. The metric for evaluation is the number of failed jobs. Through the simulation, it shows that our scaling method reduces the number of failed jobs in comparison with initial scheduling.

Fig. 4 shows the performance of the proposed auto-scaling method comparing with initial scheduling by changing the deadline of an application. Initial scheduling estimates execution of jobs of the application and schedules those jobs applying a user-defined scheduling policy. Whereas, the auto-scaling method, periodically monitoring the execution of the application, performs re-scheduling by dynamically allocating resources when there is a big difference between estimated execution time and actual execution time. The proposed auto-scaling method prevents more jobs from violating than initial scheduling does.

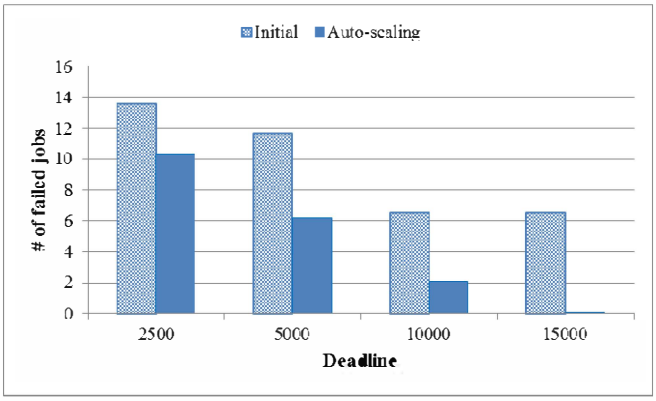

Fig. 4. Comparison result of SLA violation with initial scheduling

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose auto-scaling method to effectively manage huge resources in hybrid cloud computing. Through this auto-scaling users can use actually required resources for their applications in online manner. It can minimize their resource usage cost. Also, by choosing performance-oriented policy of two policies (i.e. the other one is cost-oriented policy) users can derive benefit of execution time of applications.

We planned to add a policy named 'Eco-scaling' to our auto-scaling method. The eco-scaling is aware of power consumption which can lead to reduction of maintenance expenses of datacenter. We will conduct an experiment on a real hybrid cloud system.

REFERENCES

[1] Amazon Web Service, http://aws.amazon.com/

[2] Scalr, http://scalr.com/

[3] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef, "Web Services on Demand: WSLA-Driven Automated Management," IBM Systems Journal, vol. 43, no. 1, pp. 136–158, 2004.

[4] W. Cirne, F. Brasileiro, J. Sauvé, Na. Andrade, D. Paranhos, E. Santos-Neto, R. Medeiros, "Grid Computing for Bag of Jobs Applications," Proceedings of the 3rd IFIP Conference on E-Commerce, E-Business and E-Government, September 2003.

[5] Paraleap, https://www.paraleap.com/

[6] Windows Azure, http://www.windowsazure.com/

[7] RightScale, http://www.rightscale.com/

[8] Ming Mao and Marty Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 12-18, 2011.

[9] Dutta, S., Gera, S., Verma, A., and Viswanathan, B. "SmartScale: Automatic application scaling in enterprise clouds," in 5th IEEE International Conference on Cloud Computing (CLOUD 2012), pp. 221-228, June. 2012.

[10] S. Genaud and J. Gossa, "Cost-wait Trade-offs in Client-side Resource Provisioning with Elastic Clouds," in 4th IEEE International Conference on Cloud Computing (CLOUD 2011), 2011.

[11] .Kim, Y.; Kim, E.-k.; Kim, J. Y.; Cho, J.-h.; Kim, C. & Cho, K. W., "e-AIRS: An e-Science Collaboration Portal for Aerospace Applications," HPCC LNCS (Lecture Note in Computer Science), Vol.4208, pp. 813-822, 0302-9743, 2006.

[12] Computational Fluid Dynamics, http://www.cfd-online.com.

[13] PRAGMA Grid and Cloud Monitoring homepage, http://pragma-goc.rockscluster.org

[14] Rodrigo N., Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," Software: Practice and Experience, 41(1):23–50, 2011.

[15] OpenNebula, http://opennebula.org/