J Supercomput (2012) 59:975–992
DOI 10.1007/s11227-010-0481-4

Evaluation of gang scheduling performance and cost in a cloud computing system

Ioannis A. Moschakis · Helen D. Karatza

Published online: 16 September 2010

Abstract Cloud Computing refers to the notion of outsourcing on-site available services, computational facilities, or data storage to an off-site, location-transparent centralized facility or "Cloud." Gang Scheduling is an efficient job scheduling algorithm for time sharing, already applied in parallel and distributed systems. This paper studies the performance of a distributed Cloud Computing model, based on the Amazon Elastic Compute Cloud (EC2) architecture that implements a Gang Scheduling scheme. Our model utilizes the concept of Virtual Machines (or VMs) which act as the computational units of the system. Initially, the system includes no VMs, but depending on the computational needs of the jobs being serviced new VMs can be leased and later released dynamically. A simulation of the aforementioned model is used to study, analyze, and evaluate both the performance and the overall cost of two major gang scheduling algorithms. Results reveal that Gang Scheduling can be effectively applied in a Cloud Computing environment both performance-wise and cost-wise.

Keywords Cloud computing · Gang scheduling · HPC · Virtual machines

1 Introduction

Cloud Computing is a *revolutionary* way of providing shared resources over the Internet. Through the use of low level *virtualization software*, such as Xen [6], the *Cloud* provides virtualized computing hardware infrastructure in a manner similar to the public utilities, thus it is also termed as *Infrastructure-as-a-Service (IaaS)* or *Utility Computing*. Since all hardware is virtualized, the *Cloud* gives the illusion of limitless resources which can be made available to the user on-demand and can be dynamically

I.A. Moschakis (✉) · H.D. Karatza
Department of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
e-mail: imoschak@csd.auth.gr

scaled up or down. On the other hand *Computing* refers to the applications and software platforms being offered through the *Cloud* usually under the notion of a service model, hence called *Software-as-a-Service (SaaS)* [2].

The importance of Cloud Computing arises in the opportunity that it provides for the development of application services without the requirement of a prior to deployment *Capital Expenditure (CapEx)*. This allows for startup Internet companies with tight budgets to use their profits for *Operational Expenditure (OpEx)* alone. Furthermore, in the scientific field of study, CC presents us with the ability to lease computational resources from its *virtually* infinite pool for use in *High Performance Computing (HPC)*. In this way, even small institutions or individuals can have access to a large number of computational resources at a fraction of the cost of maintaining a supercomputer center. Since the *Cloud* is cost-associative, we pay only for the computing time that we spent running each VM and for data transfers in and out of the Cloud. One, of course, could argue that this problem is already addressed by the *Grid*, but the Grid posses certain restrictions on the availability of software while Cloud VMs can be custom built with *virtually* any software a user needs.

In order to take advantage of computational resources that span one server, or in our case a virtual machine a *parallel or distributed* computing scheme must be applied. Although Cloud Computing infrastructure is virtualized, and thus provides no direct access to the underlying hardware, the Amazon EC2 specification provides multicore VMs, hence parallelization even on a single VM is possible. Moreover, one of the main features of Cloud Computing is its ability to adapt, so a user can expand or contract his system dynamically. Conclusively, if CC is going to be used for HPC, whose market share comprises a third of the server market [2, 5], appropriate methods must be considered for both parallel job scheduling and VM scalability.

The importance of scheduling methods is apparent in every distributed system. The scheduling algorithm must seek a way to maximize the performance of the system by avoiding unnecessary delays [18] and also in our case maintain a good response time to leasing cost ratio. The main task of the scheduler is to allocate processors to parallel jobs that have entered the system [25]. In the system modeled, parallel jobs consist of tasks that are in very frequent communication and, therefore, must execute both simultaneously and concurrently. Gang scheduling is a special case of job scheduling that allows the scheduling of such jobs. A system that applies this kind of scheduling must guarantee that every task of a given job will be allocated on a different processor so that it will begin and finish its execution at the same time as the other tasks. In this way, the system can avoid cases where a task is blocked while waiting for input from another task that is not currently executing.

This type of scheduling has been extensively studied in the past in the area of distributed and Grid systems [9, 11, 13, 23, 24]. In [11–14] Karatza has studied the performance of Adaptive First Come First Serve (AFCFS) and Largest Job First Served (LJFS) gang scheduling policies. Gang Scheduling has also been examined in situations involving more than one clusters of processors [7, 18, 19]. Also, [18] considers task migration strategies with the inclusion of high priority jobs in the process. In the aforementioned publications, the number of processors available to the system was always static during the simulation and the workload consisted of jobs with a degree of parallelism in the range $[1..P]$, P being the total number of available processors, regardless of distribution.

Scheduling strategies have been studied before under the notion of Cloud Computing. In [3], Assunção et al. studied the use of CC as an extension to private clusters. In their model, tasks were separate from each other and did not communicate. Virtual Machine usage and leasing has also been studied in [21, 22] through the use of Haizea[1] VM-based lease management architecture.

In this paper, the simulation model consists of one distributed and dynamically scaling Cloud Computing cluster of VMs. The workload consists of parallel jobs (*gangs*) that are either small or large based on a pre-simulation specified job size coefficient. We compare AFCFS and LJFS under this model in order to study their performance and cost efficiency in a Cloud Computing environment. Additionally, we implement a complex system for adding and removing virtual machines from the system depending on the system's load at any specific time. To the best of our knowledge, there have not been any other publications that have addressed this specific subject.

The structure of this paper is as follows. Section 2 presents an in-depth description of the system and workload models. Section 3 describes the Dispatching and the Scheduling strategies utilized in the simulation. In Sect. 4, we discuss the VM handling system that we have implemented. Section 5 presents the metrics used to measure performance and cost, the parameters of the simulation, and the results along with an analysis of them. Finally, Sect. 6 provides some conclusive remarks along with our thoughts about future work on the subject.

2 System and workload models

The simulation model consists of a single cluster of Virtual Machines connected with a *Dispatcher Virtual Machine (DVM)*. Initially, the system leases no VMs so the cluster is empty. Depending on the workload at any specific moment, the system has the ability to lease new VMs up to a total number of $P_{\max} = 120$. This is a limitation posed by *Amazon EC2* which allows up to *20 "Regular"* and up to *100 "Spot"* VMs which can be leased under certain conditions [1], hence virtually up to 120 VMs. The user can request even more VMs through electronic request, but the approval of the request is not certain nor an answer is guaranteed within a specified time limit. Therefore, for the time being, such a feature is excluded from the model.

Each Virtual Machine incorporates its own task waiting queue where the tasks of parallel jobs are dispatched by the *Dispatcher Virtual Machine (DVM)*. The DVM also includes a waiting queue for jobs that where unable to be dispatched at the moment of their arrival due to either *inadequacy of VMs* at that moment or due to *overloaded VMs*. For the sake of simplicity DVM is not counted within the overall limit of VMs, $P_{\max}$.

In this paper, we assume that the communication between the virtual machines is contention-free. Therefore, we consider that the communication latencies are included *implicitly* in the jobs' execution time. However, we do consider *explicit* delays

[1] Haizea: http://haizea.cs.uchicago.edu/.

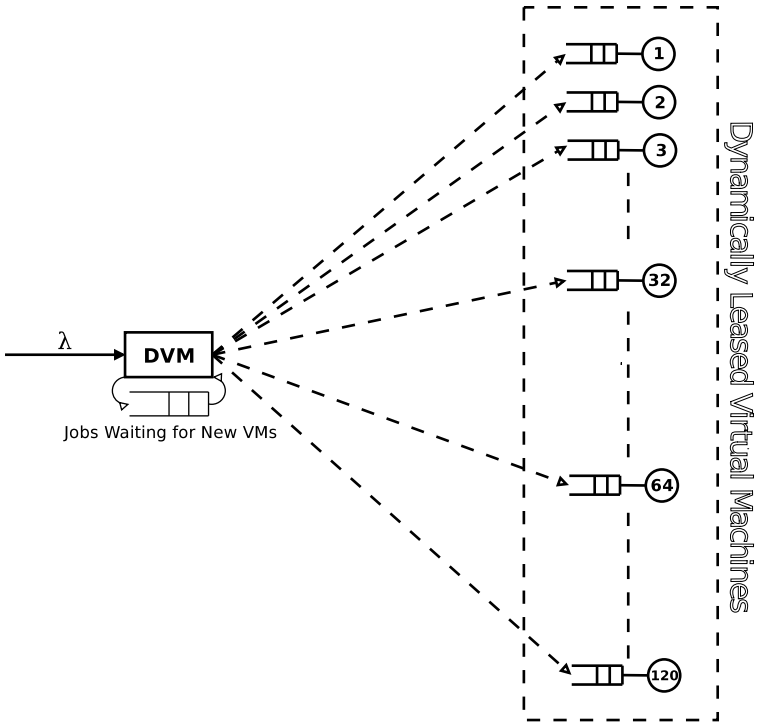

Fig. 1 The system model

when jobs are not immediately dispatched for the reasons discussed in the previous paragraph.

We also assume that all virtual machines are identical, that is, they all belong to the same class of EC2 virtual machines. As is true with *nonvirtualized* systems, VMs can suffer from *inequalities* in their performance depending on the state of the underlying hardware at any specific moment. However, studies [2, 16, 17] have shown that VMs are able to provide near homogeneous performance as long as no I/O takes place. Even this problem is expected to be resolved in the near future through the use of newer types of flash memory such as solid state drives (SSD). For these reasons, we consider that any overhead that may exist due to temporal performance difference between VMs is implicitly included in the execution time of jobs.

Gang scheduling is a special case of scheduling parallel jobs in which tasks of jobs need to communicate very frequently [18]. Thus, each job requires a number of processors equal to its degree of parallelism, the number of tasks that it consists of, in order to be dispatched and executed. In the model under study, degrees of parallelism are random numbers following the discrete uniform distribution. Furthermore, jobs fall in two different categories of size:

– **Lowly Parallel Jobs**, that have job sizes in the range [1..16] with a probability of q.

– **Highly Parallel Jobs**, that have job sizes in the range [17..32] with a probability of $1 - q$.

where q is the job size coefficient which determines the amount of jobs that belong to the first or the second category.

So we can compute the average number of tasks per job or *Average Job Size (AJS)* in the following way:

$$AJS = q\frac{1+16}{2} + (1-q)\frac{17+32}{2}. \quad (1)$$

The mean interarrival time of jobs is exponentially distributed with a mean of $1/\lambda$. The mean task service time is exponentially distributed with a mean of $1/\mu$. There exists no correlation between service times and job size, for example, it is not necessary for a large job to have a long service time.

We must emphasize here that jobs always execute to completion and that no preemption takes place. This happens because context switching in the case of Gang Scheduling involves high overhead since network status must be saved and then be restored when switching between tasks [10]. Also, as noted in the same reference, there is a possibility that some messages that should have been received by a process before it was switched may be received by another process after the context switch. For this reason, it is impractical and possibly dangerous to either preempt or migrate gang tasks when they are already running.

3 Dispatching and scheduling strategies

3.1 Job routing

The job entry point for the system is the Dispatcher VM. If the degree of parallelism of any arriving job is less than or equal to the number of the available VMs, the job is immediately dispatched. The allocation of VMs to tasks is handled by the DVM which employs the *Shortest Queue First (SQF)* algorithm for this. SQF dispatches tasks to VMs with the shortest, least loaded, queues. Tasks that belong to the same job, also called sibling tasks, cannot occupy the same queue since gang scheduling requires that there exists a one-to-one mapping of tasks to server VMs. An abstracted view of *SQF* is provided by the following listing 1.

Algorithm 1 Shortest Queue First

```
vmsByQueueLength := getVMsByQueueLengthIncremental();
for i = 0 to numberOfTasks do
  vmsByQueueLength[i].InsertInQueue(Task[i]);
end for
```

3.2 Job scheduling

Two of the most commonly used strategies in the bibliography around gang scheduling are applied. Both *Adaptive First Come Fist Served (AFCFS)* and *Largest Job Fist Served (LJFS)* have been extensively studied in the past in the area of *Grid Computing*. Our system studies their application under the notion of *Cloud Computing*. It must be noted that both algorithms are called repetitively each time a job departs since a large job may leave enough VMs idle to fit more than one smaller jobs. In this way, job scheduling is more efficient.

3.2.1 Adaptive first come first serve

The *AFCFS* algorithm tries to schedule a job every time a server VM becomes idle following a departure. If there is a job whose tasks are in front of their respective queues and all server VMs for those tasks are idle, then that job is scheduled for execution. Should there not be such a job AFCFS tries to schedule jobs whose tasks are further back in their queues in the same manner until it finds such a job or not. Due to this way of scheduling jobs, *AFCFS* tends to favor jobs that require a smaller number of server VMs. Naturally this usually results at an increase of the response time of larger jobs. In Algorithm 2, we provide a brief view of the implementation of *AFCFS*.

Algorithm 2 Adaptive First Come First Serve

```
freeVMs := getFreeVMs();
for each vm in freeVMs do
    tasksWaiting := vm.getWaitingTasks();
    for each task in tasksWaiting do
        job := task.belongsToJob();
        check := checkIfJobCanBeExecuted(job);
        if check == true then
            return job;
        else
            continue;
        end if
    end for
end for
```

3.2.2 Largest job first serve

With *LJFS* jobs are scheduled depending on their size, with priority given to the jobs with larger sizes. Every time the system tries to schedule a job it searches all jobs ordered by size and the first whose tasks are allocated to server VMs that are available is scheduled for execution. This method improves the response time of highly parallel jobs at the expense of smaller jobs taking longer to be scheduled for execution. In

the real world, this does not necessarily constitute a problem since such a discrimination is acceptable for large computer centers whose job is to handle parallel jobs that cannot be executed elsewhere. In Algorithm 3, an abstracted view of *LJFS* is provided.

Algorithm 3 Largest Job First Served

```
jobsBySize := getJobsBySizeDecremental();
for each job in jobsBySize do
  check := checkIfJobCanBeExecuted(job);
  if check == true then
    return job;
  else
    continue;
  end if
end for
```

In Sect. 5.4, we also provide results for the *First-Come-First-Serve* algorithm, which is the nonadaptive version of *AFCFS*.

4 Virtual machine handling

This section provides a thorough description of the VMs *lease/release* system. The *Cloud* provides users with the ability to quickly up-scale or sub-scale their available resources. Particularly EC2 provides service calls that request more instances of a VM. The addition of more VMs is accomplished through a virtual machine cloning process which involves the replication of a single initial state that all new virtual machines share [15]. This initial state is stored in a VM image file which contains a pre-configured system ready to be booted in a new VM instance [1].

As previously stated in Sect. 2 our system introduces a delay when a job's dispatch is delayed for the reasons previously mentioned. This delay refers to the time that the VM cloning process will take to create a stated number of new VMs. Under Amazon EC2, this delay is usually less than 10 minutes [1, 15]. In our simulation model, provisioning and release delays are random numbers following a continuous uniform distribution with a mean delay of 0.1, which is comparable to one-tenth of our mean service time of job tasks ($1/\mu$), which is 1.

4.1 Provisioning for additional virtual machines

The system implements a complex system for the *lease/release* cycle of VMs which happens dynamically while the system is in operation. In order for the system to lease new VMs, certain conditions must first be met:

– **Inadequate VMs**, This condition happens when a large job arrives and the system has an inadequate amount of VMs, at that time, to serve the job. If this happens, the newly arrived job enters the waiting queue of the DVM and waits there while

the system provisions for new virtual machines. This procedure obviously involves a certain delay that refers to the real world delay of cloning a virtual machine and inserting it in the VM cluster. The subject of cloning VMs under Amazon EC2 has been studied in [15]. In this case, the system only provisions for enough VMs in order to serve the job.

– **Overloaded VMs**, Every time a new job arrives the system checks the *Average Load Factor (ALF)* of the available VMs which is equal to

$$ALF = \frac{\sum_{i=1}^{P_l} t_i}{P_l}, \tag{2}$$

where t_i is the number of tasks currently assigned to VM i and P_l is the number of VMs leased by the system at that moment. The *ALF* threshold is set to 10 tasks per VM in our model. Should *ALF* surpass this threshold, the system provisions for new VMs equal to the degree of parallelism of the arriving job putting it on hold until they are available. After this type of VM provisioning takes place, the release mechanism is paused for 10 arrival cycles allowing the waiting queues of new VMs to fill up. While already dispatched tasks do not migrate to the new VMs the *Shortest Queue First (SQF)* dispatching strategy that is applied guarantees that new VMs will be prioritized for the next dispatches.

In any case, the system will never lease more than the aforementioned limit of $P_{\max} = 120$ VMs.

4.2 Releasing existing virtual machines

Server VMs can also be released when they are not needed. This operation is quite important since idle servers in the Cloud are costly for the user. The process of releasing a VM involves a certain delay and is considered irreversible in our model. For this reason, VMs are released only if certain criteria are met:

– The VM is currently idle, meaning that it is not servicing any task.
– The VM's task waiting queue must also be empty.
– The removal of the VM from the system will not cause a new shortage, of VMs, for jobs that are waiting in the DVM's queue for new VMs to be leased, and thus introducing further delays for those jobs.

5 Simulation metrics & results

As we have previously mentioned, the use of the *Cloud* is "cost- associative," hence one pays only for the computing time which is equivalent to the total lease time of virtual machines. Therefore, this study evaluates the proposed model not only from the *performance* perspective but also from a *cost to performance efficiency* view.

Table 1 Notations used in performance metrics

N	The total number of jobs served
q	The percentage of jobs that belong in the first size group [1..16]
$1-q$	The percentage of jobs that belong in the second size group [17..32]
μ	Mean service rate of job tasks
$1/\mu$	Mean service time of job tasks
λ	Mean arrival rate of jobs
$1/\lambda$	Mean inter-arrival time of jobs
ART	Average Response Time
AWRT	Average Weighted Response Time
AWT	Average Waiting Time
AWWT	Average Weighted Waiting Time
ABSLD	Average Bounded Slowdown
APPSLD	Average Per-Processor Slowdown

5.1 Performance metrics

In order to evaluate the system's performance, we apply the following performance metrics, summarized in Table 1:

Reponse Time r_j of a parallel job j refers to the time that it takes for the job to be processed by the system from the moment that the job arrives to the dispatcher VM to the time of the job's service completion. This metric also includes any delay that the job may suffer while waiting at the DVM's queue for new server VMs to be added to the system. The *Average Response Time (ART)* is defined as follows [13]:

$$ART = \frac{\sum_{j=1}^{N} r_j}{N}. \tag{3}$$

We also include *Weighted Response Time* which takes into account the size of each job which represents an important factor since highly parallel jobs are the prime clients of HPC implementations. The response time of each job r_j is weighted to its number of tasks $p(x_j)$ so *Average Weighted Response Time (AWRT)* is defined as follows:

$$AWRT = \frac{\sum_{j=1}^{N} p(x_j) r_j}{\sum_{j=1}^{N} p(x_j)}. \tag{4}$$

Waiting Time w_j of a parallel job j refers to time that passes between the dispatching of the job's tasks to the queues of the server VMs and the job's scheduling for execution. The *Average Waiting Time (AWT)* and *Average Weighted Waiting Time (AWWT)* are defined accordingly:

$$AWT = \frac{\sum_{j=1}^{N} w_j}{N}, \tag{5}$$

$$AWWT = \frac{\sum_{j=1}^{N} p(x_j) w_j}{\sum_{j=1}^{N} p(x_j)}. \tag{6}$$

Table 2 Notations used in cost metrics

$T_{\text{lease}(i)}$	Lease Time of VM i
P_{tot}	Total number of VMs leased
LT	Total Lease Time of VMs
CPE	Relative (%) increase in Cost-Performance when LJFS is applied instead of AFCFS
D_{LT}	Relative (%) decrease in LT when LJFS is applied instead of AFCFS
D_{ART}	Relative (%) decrease in ART when LJFS is applied instead of AFCFS

Slowdown s_j of a job j is the response time of the job divided by its service time. This metric is used to measure the delay suffered by a job against its actual run-time. If e_j is the execution time of job j. then slowdown is defined as follows:

$$S_j = \frac{r_j}{e_j}. \tag{7}$$

This metric overemphasizes the importance of very short jobs since an extremely small time at the denominator can have an immediate effect at the average slowdown of the entire system. For this reason we apply the "Bounded-Slowdown" metric [8]:

$$S_j^{\text{bounded}} = \max\left\{\frac{r_j}{\max\{e_j, \tau\}}, 1\right\}. \tag{8}$$

For *Average Bounded Slowdown (ABSLD)*:

$$ABSLD = \frac{\sum_{j=1}^{N} S_j^{\text{bounded}}}{N}. \tag{9}$$

And the "Per-Processor Slowdown" metric [26]:

$$S_j^{pp} = \max\left\{\frac{r_j}{p(x_j)\max\{e_j, \tau\}}, 1\right\}. \tag{10}$$

For the *Average Per-Processor Slowdown (APPSLD)*:

$$APPSLD = \frac{\sum_{j=1}^{N} S_j^{pp}}{N}, \tag{11}$$

where τ is the threshold we set so that low execution times do not affect slowdown significantly. During the simulation of this model, this threshold was set to $\tau = 0.001$.

5.2 Cost metrics

Cost metrics, as summarized in Table 2, are considered in this study since the use of the *EC2 Cloud* is not free. The cost of Cloud usage derives from the lease time of VMs. Therefore, when evaluating the performance of a scheduling algorithm, we must also take into account the total lease time (LT) of virtual machines while the system is in operation:

$$LT = \sum_{i=1}^{P_{\text{tot}}} T_{\text{lease}(i)}, \tag{12}$$

where $T_{\text{lease}(i)}$ is the lease time of VM i and P_{tot} is the total number VMs leased by the system.

Thus, in order to compare LJFS and AFCFS cost-performance wise, we devise the following metric:

Cost-Performance Efficiency (CPE) which is evaluated by combining *LT* with the *ART* performance metric. *CPE* provides the relative increase in cost-performance when *LJFS* is applied instead of *AFCFS* and is defined as follows:

$$CPE = (-D_{LT}) + (-D_{ART}), \tag{13}$$

where D_{LT} is the relative decrease in lease time when LJFS is applied instead of AFCFS and D_{ART} is the relative decrease in ART when LJFS is applied instead of AFCFS. D_{LT} & D_{ART} are computed as such:

$$D_{LT} = \frac{LT_{LJFS} - LT_{AFCFS}}{LT_{AFCFS}}, \tag{14}$$

$$D_{ART} = \frac{ART_{LJFS} - ART_{AFCFS}}{ART_{AFCFS}}. \tag{15}$$

It must be made clear that a negative value in D_{LT} means that *LJFS* acts better than *AFCFS*. The same is true for D_{ART}, hence *CPE* is the computed by summing the negated values of both D_{LT} and D_{ART}. For this reason, a positive *CPE* denotes that *LJFS* is better than *AFCFS* in terms of cost-performance while a negative *CPE* denotes that *AFCFS* is better.

5.3 Simulation parameters

The queuing network model previously described was implemented using discrete event system simulation [4]. Each result presented here is the average of 30 differently instantiated replications of the simulation experiment for each arrival rate (λ) and each algorithm per job size coefficient. Each simulation run was terminated upon successful completion of 64,000 jobs.

Three different job size coefficients where applied in order to study their effects on performance while maintaining the same workload:

- $q = 0.25$, in order to study how the system behaves when larger jobs are significantly more than smaller ones.
- $q = 0.5$, in order to study a balanced system.
- $q = 0.75$, in order to study the behavior of the system under a multitude of smaller jobs as workload.

According to (1):

- For $q = 0.25$ jobs require 20.5 VMs on average.
- For $q = 0.50$ jobs require 16.5 VMs on average.
- For $q = 0.75$ jobs require 12.5 VMs on average.

The simulation model is bounded at 120 VMs which means that for $q = 0.75$, $\lambda < \frac{120}{12.5} = 9.6$, for $q = 0.50$, $\lambda < \frac{120}{16.5} \approx 7.27$ and for $q = 0.25$, $\lambda < \frac{120}{20.5} \approx 5.85$. Since the

system uses gang scheduling, where in many occasions jobs are not executed while idle VMs are available, the arrival rates should be much smaller than those computed above. Furthermore, this paper aims to study a dynamically scalable system, thus through empirical study we have chosen arrival rates that do not lead to a degenerate system. The term "degenerate" refers to a system where almost no releases take place so the system ends up having close to 120 VMs permanently leased throughout its operation.

In order to further clarify our choice of arrival rates, we should note that for each simulation experiment we computed P_{avg} which is the average number of processors in the system for each experiment and is given by the following formula:

$$P_{\text{avg}} = \frac{LT}{T}, \tag{16}$$

where LT is, as stated above, the total lease time of VMs and T is the total simulation time.

This number reveals if the arrival rate used at the experiment led to system degeneration. If $P_{\text{avg}} \simeq P_{\text{max}}$ the system is considered "degenerate." Consequently, arrival rates that led to system degeneration where excluded from experimentation. Different arrival rates were evaluated for each q since the differentiation in job sizes greatly affects the system's performance and its ability to scale.

The following arrival rates were used in our experiments:

- For $q = 0.25$, $\lambda = 1.75, 2.0, 2.25, 2.5, 2.75$.
- For $q = 0.50$, $\lambda = 2.25, 2.5, 2.75, 3.0, 3.25$.
- For $q = 0.75$, $\lambda = 2.5, 3.0, 3.5, 4.0, 4.5$.

As previously mentioned, the mean service time of job tasks $(1/\mu)$ used in all experiments was 1.

For every mean value, a 95% confidence interval was evaluated. The half-widths of all confidence intervals were less than 5% of their respective mean values.

5.4 Results

The following results depict the difference between both the performance and the cost of the two aforementioned algorithms under various workloads and different job size coefficients. Performance results regarding time, ergo ART, AWRT, AWT, and AWWT, are counted in theoretical generic Time Units (TUs) since the model was simulated with discrete event simulation.

Figure 2a–f depicts the *Average Response Time* and *Average Weighted Response Time* versus λ. Figure 3a–f shows the *Average Waiting Time* and *Average Weighted Waiting Time* versus λ. Figure 4a–f depicts the *Average Bounded Slowdown* and *Average Per-Processor Slowdown* versus λ. Finally, Table 3 lists the Cost-to-Performance-Efficiency for all arrival rates and job size coefficients.

5.5 Average response time and average weighted response time

Figure 2a, c, e depicts the comparison of the response times given by *AFCFS, LJFS,* and *FCFS* for $q = 0.25$, $q = 0.5$, and $q = 0.75$, respectively.

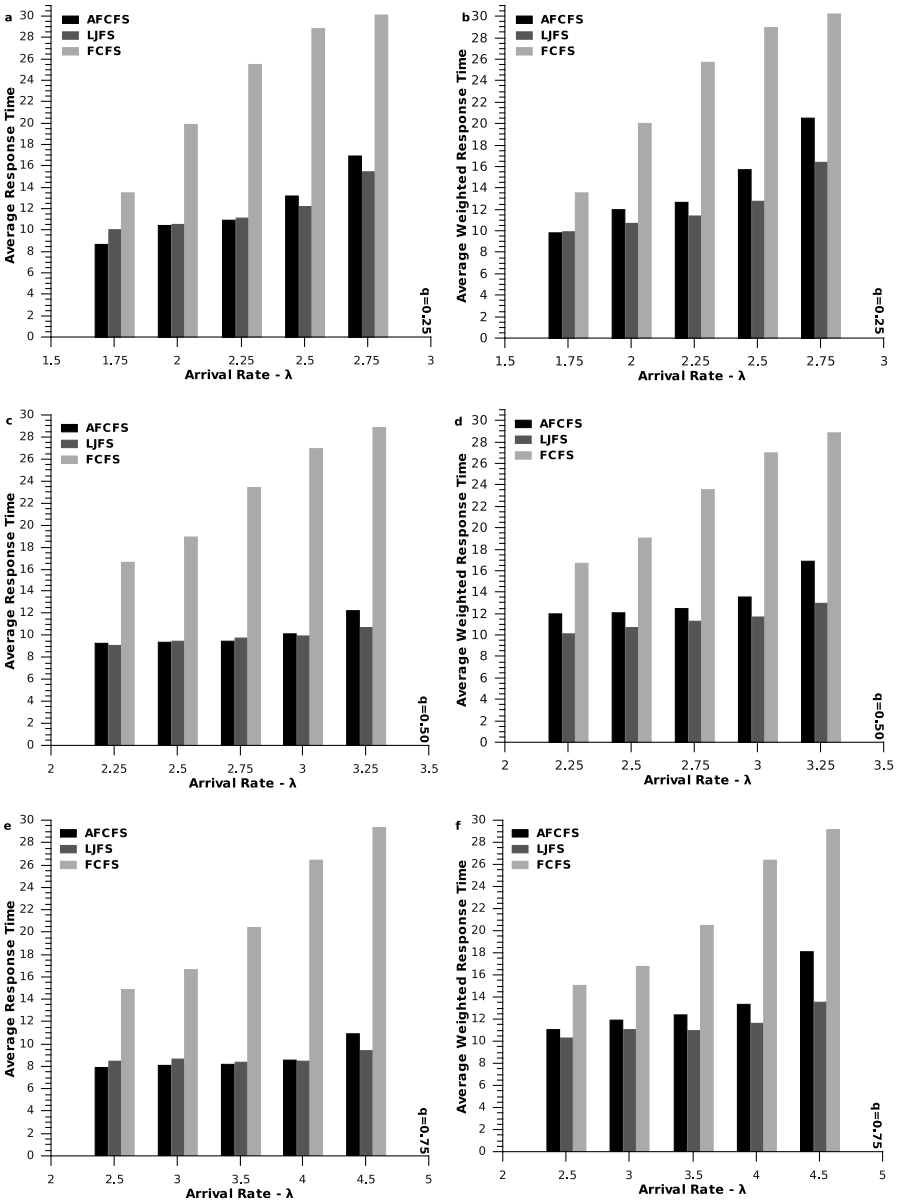

Fig. 2 (**a**) *ART* vs λ $q = 0.25$, (**b**) *AWRT* vs λ $q = 0.25$, (**c**) *ART* vs λ $q = 0.50$, (**d**) *AWRT* vs λ $q = 0.50$, (**e**) *ART* vs λ $q = 0.75$, (**f**) *AWRT* vs. λ $q = 0.75$

As is apparent throughout simulation results *FCFS*, as a nonadaptive method, quickly degenerates the system under all situations. This happens because *FCFS* tries to schedule jobs whose tasks are in the first slot of their waiting queues. If those tasks cannot be scheduled at that time *FCFS* looks no further down the queues. As a result,

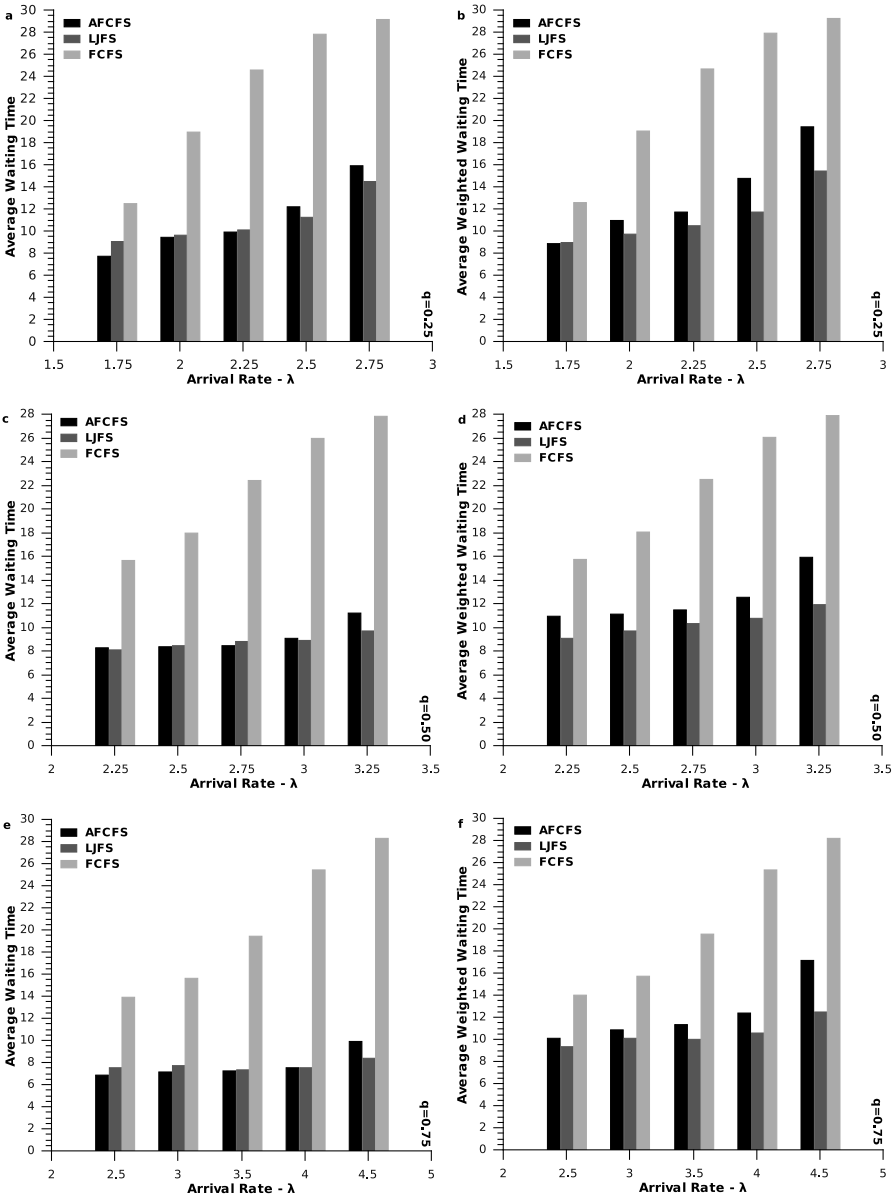

Fig. 3 (**a**) *AWT* vs λ $q = 0.25$, (**b**) *AWWT* vs λ $q = 0.25$, (**c**) *AWT* vs λ $q = 0.50$, (**d**) *AWWT* vs λ $q = 0.50$, (**e**) *AWT* vs λ $q = 0.75$, (**f**) *AWWT* vs. λ $q = 0.75$

on many occasions jobs that could have been scheduled, were not because of the way *FCFS* works.

AFCFS seems to offer better results than *LJFS* while arrival rates are low regardless of the job size coefficient. When the workload gets heavier its performance drops

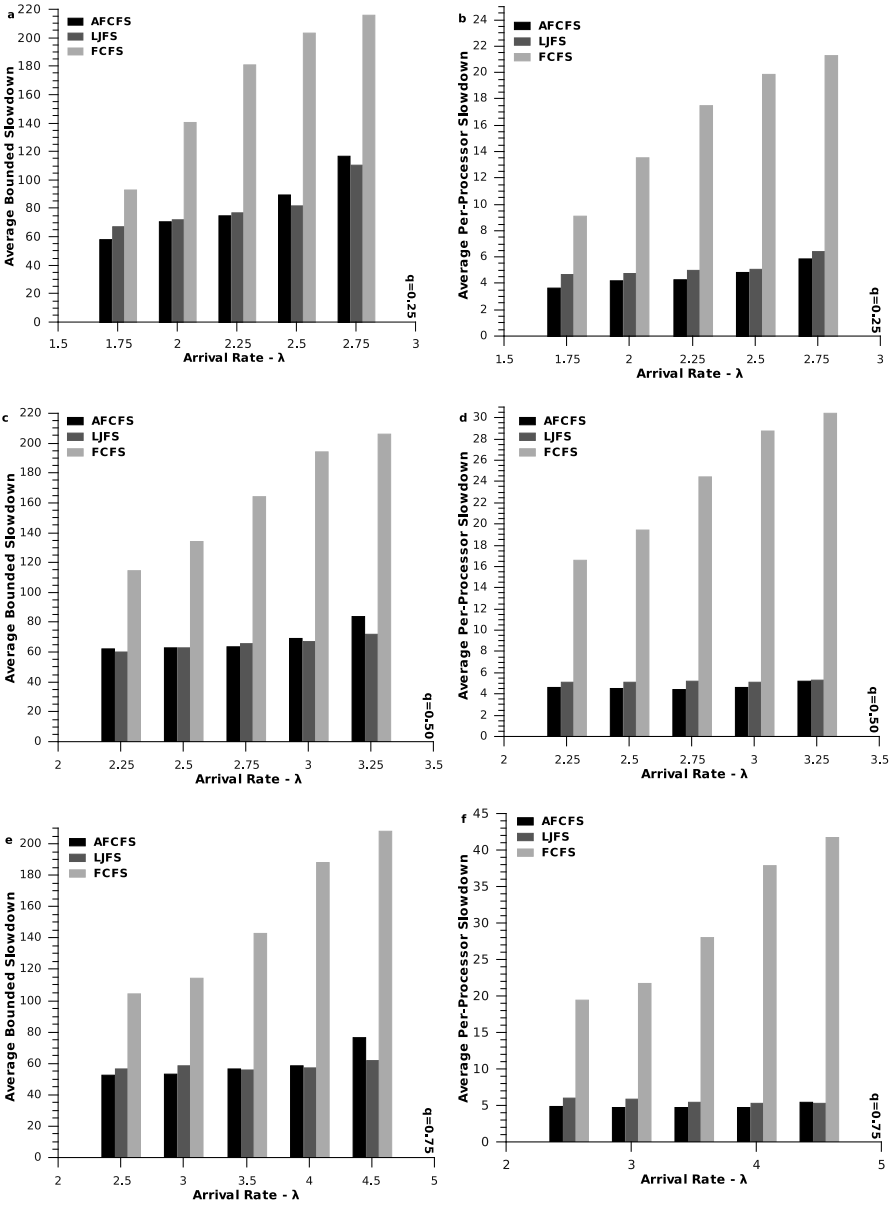

Fig. 4 (**a**) *ABSLD* vs λ $q = 0.25$, (**b**) *APPSLD* vs λ $q = 0.25$, (**c**) *ABSLD* vs λ $q = 0.50$, (**d**) *APPSLD* vs λ $q = 0.50$, (**e**) *ABSLD* vs λ $q = 0.75$, (**f**) *APPSLD* vs λ $q = 0.75$

to the same level as *LJFS*. High workloads which push the system close to the point of degeneration show that *LJFS* is slightly more effective at handling them compared to *AFCFS*. In fact for $q = 0.25$ *AFCFS* degrades quicker even at smaller workloads. Such results are expected since *AFCFS* tends to favor smaller jobs.

In Fig. 2b, d, f, the weighted response time of all algorithms is depicted. Again, *FCFS* offers worse results as is expected. This metric takes into consideration job sizes as shown in (4). Therefore, it illustrates even more the weakness of *AFCFS* which has a tendency to over-schedule smaller jobs in the expense of postponing execution of larger jobs, which are usually more important. This figure also depicts that *LJFS* offers superior performance for larger jobs since they are given priority over smaller ones. On the other hand, smaller jobs do not necessarily suffer starvation since larger jobs often leave large enough numbers of VMs idle for smaller jobs to be effectively scheduled.

5.6 Average waiting time & average weighted waiting time

Waiting Time and *Weighted Waiting Time* follow a similar pattern to that of the response time. This is expected as there exists no correlation between size and execution times. As is depicted in Fig. 3a, c, e, *AFCFS* tends to offer smaller waiting times than *LJFS* for low arrival rates. That changes though when the workloads get heavier. Furthermore Fig. 3b, d, f show that waiting times of larger jobs are significantly higher when *AFCFS* is applied.

5.7 Average bounded slowdown and average per-processor slowdown

Slowdown measures the delay that a job suffers in relation to its execution time. Figure 4a, c, e illustrate again the Slowdown posed by *AFCFS* is larger for heavier workloads. Figure 4b, d, f reveal that Per-Processor Slowdown of *AFCFS* is consistently lower than that of *LJFS* by a small margin. This is not unexpected though since Slowdown is a metric that is easily affected [8] by multiple factors. This is even more true for Per-Processor Slowdown which adds the number of VMs required by each job to the equation.

5.8 Cost-to-performance efficiency

Table 3 shows the *CPE* for all arrival rates and job size coefficients tested. As previously mentioned, positive values denote that *LJFS* is more efficient cost-performance wise while negative values denote the opposite. As is depicted, *LJFS* steadily provides better efficiency. Even at situations where *AFCFS* provides better response time, the busy time to lease time ratio of *LJFS* is significantly better than that of *AFCFS*. Since the usage of the "Cloud" is directly associated with lease time, it becomes obvious that the use of *LJFS* can provide better overall performance at a significantly lower cost even more so when workloads are heavier.

6 Conclusions and future work

This study explored the application of the two most commonly used Gang Scheduling strategies *AFCFS* and *LJFS* under the notion of Cloud Computing. The simulation model implemented was based on the already existing *Amazon EC2* cloud computing

Table 3 Cost-to-performance efficiency AFCFS-LJFS

λ	$q = 0.25$	$q = 0.5$	$q = 0.75$
1.75	−10.1232	–	–
2	1.3171	–	–
2.25	4.2493	6.3589	–
2.5	14.8915	4.2130	0.1413
2.75	17.3365	2.2873	–
3	–	8.1537	0.4645
3.25	–	21.5644	–
3.5	–	–	4.7581
4	–	–	8.4878
4.5	–	–	22.9386

implementation which to the best of our knowledge is the only currently available CC infrastructure that can implement our model.[2] Virtual machines were considered as the computational unit of our distributed system instead of classic servers. Multiple job sizes where studied in our implementation along with dynamic availability of VMs which were leased and released by the system in real time. Experiments were conducted through the use of a simulation model under various workloads and job size characteristics. Apart from the performance metrics that are usually utilized in bibliography new metrics were devised in order to assess the performance in relation to the cost that the application of each algorithm imposed on the *Cloud* user.

Both algorithms proved that they can be efficiently applied in an environment with a nonstatic number of VMs. While both algorithms provide similar performance for medium workloads *LJFS* outperforms *AFCFS* when workloads get heavier. That is even more apparent in weighted metrics which take into account job sizes. Cost-wise *LJFS* provides superior cost-performance efficiency than *AFCFS* and is far-superior in situations of heavy workload.

We must note here that as this field of study is relatively new and, though the present model is complex, in the future different scenarios involving the use of job migration along with variable workloads and job sizes and types must be considered to better fit a real *HPC* cloud computing implementation. Furthermore, the introduction of classes of VMs can also be added to the model since CC can offer computation units with nonuniform performance at will.

References

1. Amazon Web Services LLC (2009) Amazon elastic compute cloud (EC2). http://aws.amazon.com/ec2/
2. Armbrust M, Fox A, Griffith R et al (2009) Above the clouds: a Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Berkeley Reliable Adaptive Distributed Systems Laboratory, USA, February 2009

[2] Haizea as stated by Sotomayor in [20] is another platform that can possibly implement our model.

3. Assunção MD, Costanzo A, Buyya R (2009) Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. High perform distrib comp. ACM, Munich
4. Averill L (2007) Simulation modelling & analysis, 4th edn. McGraw-Hill, New York
5. Bechtolsheim A (2008) Cloud computing and cloud networking. Talk at UC Berkeley
6. Citrix Systems Inc (2010) Xen Hypervisor. http://www.xen.org/
7. Dimitriadou S, Karatza H (2009) Multi-site allocation policies on a grid and local level. In: 4th Int workshop pract appl stoch model, Mascots 2009 Workshop. Imperial College, London
8. Feitelson DG (2001) Metrics for parallel job scheduling and their convergence. In: Feitelson DG, Rudolph L (eds) Job sched strateg for parallel process. Springer, London, pp 188–206
9. Feitelson DG, MA Jette (1997) Improved utilization and responsiveness with gang scheduling. In: Job sched strateg for parallel process. Lect Notes in Comp Sci, vol 1291. Springer, London, pp 238–261
10. Hori A, Tezuka H, Ishikawa Y (1998) Overhead analysis of preemptive gang scheduling. In: Proc of the workshop on job sched strateg for parallel process. Lect notes in comp sci, vol 1459. Springer, Berlin, pp 217–230
11. Karatza H (2006) Scheduling gangs in a distributed system. Int J Simul: Syst Sci Technol 7:15–22
12. Karatza H (2006) Performance analysis of gang scheduling in a partitionable parallel system. In: Proc 20th Eur conf model simul, Bonn, Germany, 2006
13. Karatza H (2007) Performance of gang scheduling policies in the presence of critical sporadic jobs in distributed systems. In: Proc symp perform evaluation of comp telecommun syst 2007, San Diego, CA, pp 547–554
14. Karatza H (2009) Performance of gang scheduling strategies in a parallel system. Simul Model Pract Theory 17:430–441. doi:10.1016/j.simpat.2008.10.001
15. Lagar-Cavilla A, Whitney J, Scannell A, Patchin P, Rumble S, Lara E, Brudno M, Satyanarayanan M (2009) SnowFlock: Rapid virtual machine cloning for cloud computing. EuroSys '09, Nuremberg, Germany
16. McCalpin J (1995) Memory bandwidth and machine balance in current high performance computers. IEEE Tech Comm on Comput Archit Newsl, pp 19–25
17. Murphy M, Abraham L, Fenn M, Goasguen S (2010) Autonomic clouds on the grid. J Grid Comput 8(1):1–18. doi:10.1007/s10723-009-9142-3
18. Papazachos Z, Karatza H (2009) The impact of task service time variability on gang scheduling performance in a two-cluster system. Simul Model Pract Theory 17:1276–1289. doi:10.1016/j.simpat.2009.05.002
19. Papazachos Z, Karatza H (2009) Performance evaluation of gang scheduling in a two-cluster system with migrations. In: 8th int workshop perform model, evaluation optim of ubiquitous comp and netw syst, Rome, Italy 2009. doi:10.1109/IPDPS.2009.5161172
20. Sotomayor B (2009) Haizea and private clouds, "blog.dsa-research.org" blog, http://blog.dsa-research.org/?p=138, February 19, 2009
21. Sotomayor B, Keahey K, Foster I (2008) Combining batch execution and leasing using virtual machines. In: ACM/IEEE int symp on high perform distrib comp 2008, Boston
22. Sotomayor B, Montero S, Llorente M (2009) Foster I resource leasing and the art of suspending virtual machines. In: The 11th IEEE int conf on high perform comp and commun. Seoul, Korea
23. Wiseman Y, Feitelson DG (2003) Paired gang scheduling. IEEE Trans Parallel Distrib Syst 14:581–592
24. Zhang Y, Franke H, Moreira JH, Sivasubramaniam A (2001) An integrated approach to parallel scheduling using gang-scheduling, backfilling and migration. In: 7th Workshop on job sched strateg for parallel process. Lect notes in comp sci, vol 2221. Springer, Berlin, pp 133–158
25. Zomaya A, Chan F (2005) Efficient clustering for parallel task execution in distributed systems. J Found Comput Sci 16:281–299. doi:10.1109/IPDPS.2004.1303164
26. Zotkin D, Keleher PJ (1999) Job-length estimation and performance in backfilling schedulers. In: 8th Intl symp high perform distrib comput. IEEE, Los Alamitos