This is a preprint of the article "C. Mateos, E. Pacini and C. GarcÃ-a Garino: "An ACO-inspired Algorithm for Minimizing Weighted Flowtime in Cloud-based Parameter Sweep Experiments". Advances in Engineering Software. Vol. 56, pp. 38-50. Elsevier. 2013. ISSN 0965-9978."

The final publication is available at http://dx.doi.org/10.1016/j.advengsoft.2012.11.011

An ACO-inspired Algorithm for Minimizing Weighted Flowtime in Cloud-based Parameter Sweep Experiments

Cristian Mateos[a,*], Elina Pacini[b], Carlos García Garino[b]

[a]*ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (249) 4439682 ext. 35. Fax.: +54 (249) 4439681 - Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)*

[b]*ITIC - UNCuyo University. Mendoza, Mendoza, Argentina.*

Abstract

Parameter Sweep Experiments (PSEs) allow scientists and engineers to conduct experiments by running the same program code against different input data. This usually results in many jobs with high computational requirements. Thus, distributed environments, particularly Clouds, can be employed to fulfill these demands. However, job scheduling is challenging as it is an NP-complete problem. Recently, Cloud schedulers based on bio-inspired techniques –which work well in approximating problems with little input information– have been proposed. Unfortunately, existing proposals ignore job priorities, which is a very important aspect in PSEs since it allows accelerating PSE results processing and visualization in scientific Clouds. We present a new Cloud scheduler based on Ant Colony Optimization, the most popular bio-inspired technique, which also exploits well-known notions from operating systems theory. Simulated experiments performed with real PSE job data and other Cloud scheduling policies indicate that our proposal allows for a more agile job handling while reducing PSE completion time.

Keywords: Parameter Sweep Experiments, Cloud Computing, job scheduling, Swarm Intelligence, Ant Colony Optimization, weighted flowtime

1. Introduction

Parameter Sweep Experiments (PSEs) is a very popular way of conducting simulation-based experiments, used by scientists and engineers, through which the same application code is run several times with different input parameters resulting in different output data [48]. Running PSEs involves managing many independent jobs [40], since the experiments are executed under multiple initial configurations (input parameter values) many times, to locate a particular point in the parameter space that satisfies certain user criteria. In addition, different PSEs have different number of parameters, because they model different scientific or engineering problems. Cur-

*Corresponding author.

Email addresses: cmateos@conicet.gov.ar (Cristian Mateos), epacini@itu.uncu.edu.ar (Elina Pacini), cgarcia@itu.uncu.edu.ar (Carlos García Garino)

rently, PSEs find their application in diverse scientific areas such as Bioinformatics [42], Earth Sciences [16], High-Energy Physics [3] and Molecular Science [46].

A concrete example of a PSE is the one presented by Careglio et al. [7], which consists in analyzing the influence of size and type of geometric imperfections in the response of a simple tensile test on steel bars subject to large deformations. To conduct the study, the authors numerically simulate the test by varying some parameters of interest, namely using different sizes and types of geometric imperfections. By varying these parameters, several study cases were obtained, which was necessary to analyze and run on different machines in parallel.

Users relying on PSEs need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called High-Throughput Computing (HTC) environment. In HTC, jobs are dispatched to run independently on multiple machines at the same time. A distributed paradigm that is growing is Cloud Computing [5], which offers the means for building the next generation parallel computing infrastructures along with easy of use. Although the use of Clouds finds its roots in IT environments, the idea is gradually entering scientific and academic ones [32].

Executing PSEs on Cloud Computing environments (or Clouds for short) is not free from the well-known scheduling problem, i.e., it is necessary to develop efficient scheduling strategies to appropriately allocate the workload and reduce the associated computation times. Scheduling refers to the way jobs are assigned to run on the available CPUs of a distributed environment, since there are typically many more jobs running than available CPUs. However, scheduling is an NP-hard problem and therefore it is not trivial from an algorithmic complexity standpoint.

In the last ten years or so, Swarm Intelligence (SI) has received increasing attention in the research community. SI refers to the collective behavior that emerges from social insects swarms [4]. Social insect swarms solve complex problems collectively through intelligent methods. These problems are beyond the capabilities of each individual insect, and the cooperation among them is largely self-organized without any central supervision. After studying social insect swarms behaviors such as ant colonies, researchers have proposed some algorithms or theories for solving combinatorial optimization problems. Moreover, job scheduling in Clouds is also a combinatorial optimization problem, and several schedulers in this line exploiting SI have been proposed.

As discussed in [31, 30], existing SI schedulers completely ignore job priorities. Particularly, for running PSEs, this is a very important aspect. When designing a PSE as a set of N jobs, where every job in the set is associated a particular value for the i_{th} model variable being varied and studied by the PSE. In this case job running times between jobs can be very different. This is due to the fact that running the same code or solver (i.e., job) against many input values usually yields very dissimilar execution times as well. This situation is very undesirable since, unless the scheduler knows some job information, the user can not process/visualize the outputs of the whole PSE until all jobs have finished. Thus, in principle, giving higher (or lower) priority to jobs that are supposed to take longer to finish may help in improving output processing.

In this paper, we propose a new scheduler that is based on Ant Colony Optimization (ACO), the most popular SI technique, for executing PSEs in Clouds by taking into account job priorities. Specifically, we formulate our problem as minimizing the *weighted flowtime* of a set of jobs, i.e., the weighted sum of job finish times minus job start times, while also minimizing makespan,

i.e., the total execution time of all jobs. Our scheduler essentially includes a Cloud-wide VM (Virtual Machine) allocation policy based on ACO to map VMs to physical hosts, plus a VM-level policy for taking into account individual job priorities that bases on the inverse of the well-known "convoy effect" from operating systems theory. Experiments performed using the CloudSim simulation toolkit [6], together with job data extracted from real-world PSEs and alternative scheduling policies for Clouds show that our scheduler effectively reduces weighted flowtime and achieves better makespan levels than other existing methods.

The rest of the paper is structured follows. Section 2 gives some background necessary to understand the concepts of the approach presented in this paper. Then, Section 3 surveys relevant related works. Section 4 presents our proposal. Section 5 presents detailed experiments that show the viability of the approach. Finally, Section 6 concludes the paper and delineates future research opportunities.

2. Background

Certainly, Cloud Computing [5, 44] is the current emerging trend in delivering IT services. By means of virtualization technologies, Cloud Computing offers to end-users a variety of services covering the entire computing stack, from the hardware to the application level. This makes the spectrum of configuration options available to scientists, and particularly PSEs users, wide enough to cover any specific need from their research. Another important feature, from which scientists can benefit, is the ability to scale up and down the computing infrastructure according to PSE resource requirements. By using Clouds scientists can have easy access to large distributed infrastructures and are allowed to completely customize their execution environment, thus deploying the most appropriate setup for their experiments.

2.1. Cloud Computing basics

The concept of *virtualization* is central to Cloud Computing, i.e., the capability of a software system of emulating various operating systems. By means of this support, users exploit Clouds by requesting from them *machine images*, or virtual machines that emulate any operating system on top of several physical machines, which in turn run a host operating system. Usually, Clouds are established using the machines of a datacenter for executing user applications while they are idle. In other words, a scientific user application can co-allocate machine images, upload input data, execute, and download output (result) data for further analysis. Finally, to offer on demand, shared access to their underlying physical machines, Clouds have the ability to dynamically allocate and deallocate machines images. Besides, Clouds can co-allocate N machines images on M physical machines, with $N \geq M$, thus concurrent user-wide resource sharing is ensured. These relationships are depicted in Figure 1.

With everything mentioned so far, there is a great consensus on the fact that from the perspective of domain scientists the complexity of traditional distributed and parallel computing environments such as clusters and particularly Grids should be hidden, so that domain scientists can focus on their main concern, which is performing their experiments [45, 27, 26]. The value of Cloud Computing as a tool to execute complex scientific applications in general [45, 44] and parametric studies in particular [25] has been already recognized within the scientific community.

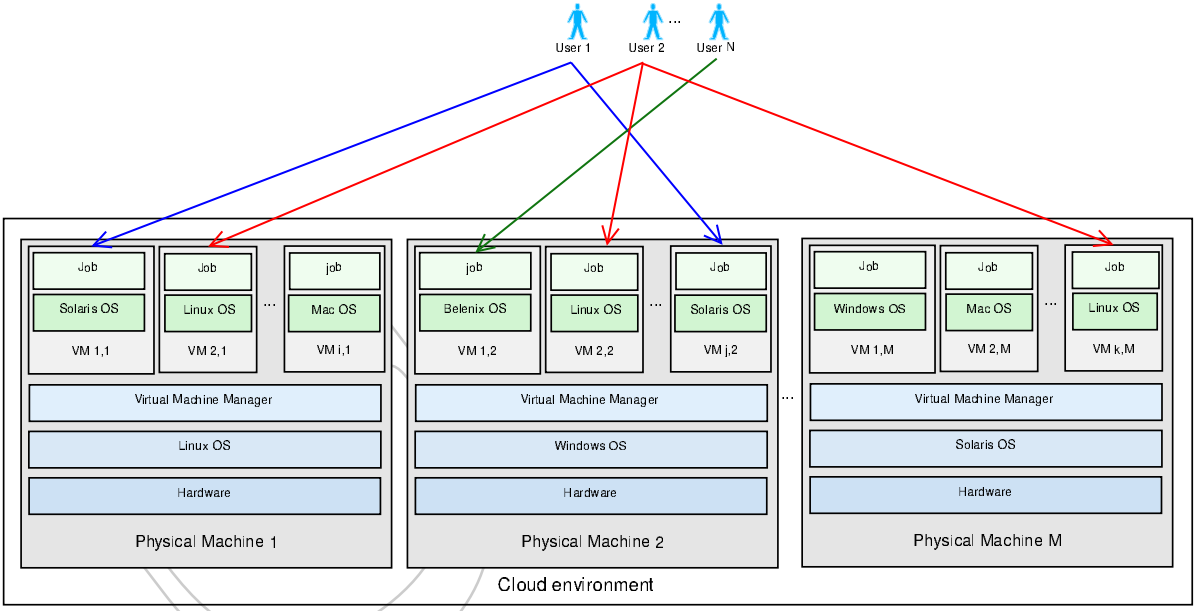


Figure 1: Cloud Computing: High-level view

While Cloud Computing helps scientific users to run complex applications, job management and particularly scheduling is a key concern that must be addressed. Broadly, job scheduling is a mechanism that maps jobs to appropriate executing machines, and the delivered efficiency directly affects the performance of the whole distributed environment. Particularly, distributed scheduling algorithms have the goal of processing a single application that is composed of several jobs by submitting these latter to many machines, while maximizing resource utilization and minimizing the total execution time (i.e., makespan) of the jobs.

2.2. *Job and Cloud scheduling basics*

According to the well-known taxonomy of scheduling in distributed computing systems by Casavant & Kuhl [8], from the point of view of the quality of the solutions a scheduler is able to build, any scheduling algorithm can be classified into *optimal* or *sub-optimal*. The former characterizes scheduling algorithms that, based on complete information regarding the state of the distributed environment (e.g., hardware capabilities and load) as well as resource needs (e.g., time required by each job on each computing resource), carry out optimal job-resource mappings. When this information is not available, or the time to compute a solution is unfeasible, sub-optimal algorithms are used instead.

Sub-optimal algorithms are further classified into *heuristic* or *approximate*. First, heuristic algorithms are those that make as few assumptions as possible (or even no assumptions) about resource load or job duration prior to perform scheduling. Approximate schedulers, on the other hand, are based on the same input information and formal computational model as optimal schedulers but they try to reduce the solution space so as to cope with the NP-completeness of optimal scheduling algorithms. However, having this information again presents problems in practice. Most of the time, for example, it is difficult to estimate job duration accurately since the runtime

behavior of a job is unpredictable beforehand because of conditional code constructs or network congestion.

In this sense, Clouds, as any other distributed computing environment, is not free from the problem of accurately estimate aspects such as job duration. Another aspect that makes this problem more difficult is *multi-tenancy*, a distinguishing feature of Clouds by which several users and hence their (potentially heterogeneous) jobs are served at the same time via the illusion of serveral logic infrastructures that run on the same physical hardware. This also poses challenges when estimating resource load. Indeed, optimal and sub-optimal-approximate algorithms such as those based on graph or queue theory need accurate information beforehand to perform correctly, and therefore in general heuristic algorithms are preferred. In the context of our work, we are dealing with highly multi-tenant Clouds where jobs come from different Cloud users (people performing multi-domain PSE experiments) and their duration cannot be predicted.

It is true that, however, several heuristic techniques for distributed scheduling have been developed [18]. One of the aspects that particularly makes SI techniques interesting for distributed scheduling is that they perform well in approximating optimization problems without requiring too much information on the problem beforehand. From the scheduling perspective, SI-based job schedulers can be conceptually viewed as hybrid scheduling algorithms, i.e., heuristic schedulers that partially behave as approximate ones. Precisely, this characteristic has raised a lot of interest in the scientific community, particularly for ACO [11].

2.3. Bio-inspired techniques for Cloud scheduling

Broadly, bio-inspired techniques have shown to be useful in optimization problems. The advantage of these techniques derives from their ability to explore solutions in large search spaces in a very efficient way along with little initial information. Therefore, the use of this kind of heuristics is an interesting approach to cope in practice with the NP-completeness of job scheduling, and handling application scenarios in which for example job execution time cannot be accurately predicted. Particularly, existing literature show that they are good candidates to optimize job execution time and load balancing in both Grids and Clouds [31, 30]. In particular, the great performance of ant algorithms for scheduling problems was first shown in [29]. Interestingly, several authors [22, 28, 36, 12, 17, 20, 39, 23, 41] have complementary shown in their experimental results that by using ACO-based techniques jobs can be allocated more efficiently and more effectively than using other traditional heuristic scheduling algorithms such as GA (Genetic Algorithms) [36], Min-Min [20, 36] and FCFS [22].

One of the most popular and versatile bio-inspired technique is Ant Colony Optimization (ACO), which was introduced by Marco Dorigo in his doctoral thesis [10]. ACO was inspired by the observation of real ant colonies. An interesting behavior is how ants can find the shortest paths between food sources and their nest.

Real ants initially wander randomly, and upon finding food they return to their colony while laying down pheromone trails. If other ants find the same “lucky” path, they are likely not to keep traveling at random, but to instead follow the trail, returning and reinforcing it if they eventually find more food. When one ant finds a good (i.e., short) path from the colony to a food source, other ants are more likely to follow that path, and positive feedback eventually leaves all the

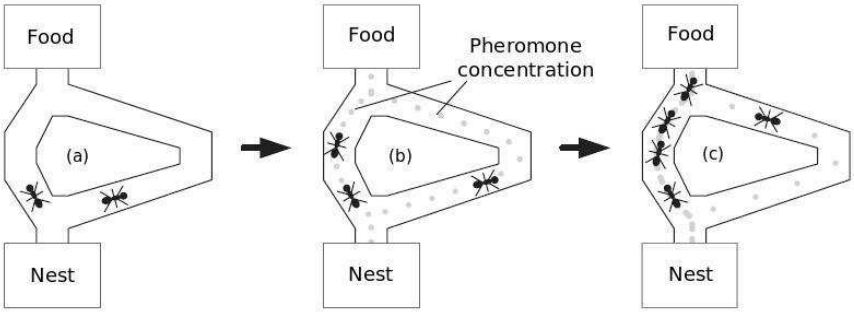


Figure 2: Adaptive behavior of ants

ants following a single path. Conceptually, ACO then mimics this behavior with *simulated ants* walking around the graph representing the problem to solve.

Over time, however, pheromone trails start to evaporate, thus reducing their attractive strength. The more the time it takes for an ant to travel down the path and back again, the less the frequency with which pheromone trails are reinforced. A short path, by comparison, gets marched over faster, and thus the pheromone density remains high as it is laid on the path as fast as it can evaporate. From an algorithmic standpoint, the pheromone evaporation process has also the advantage of avoiding the convergence to a locally good solution. If there were no evaporation at all, the paths chosen by the first ants would tend to be excessively attractive to the following ones. In that case, the exploration of the solution space would be constrained.

Figure 2 shows two possible paths from the nest to the food source, but one of them is longer than the other. Ants will start moving randomly to explore the ground and choose one of the two ways as can be seen in (a). The ants taking the shorter path will reach the food source before the others and leave behind them a trail of pheromones. After reaching the food, the ants will turn back and try to find the nest. The ants that go and return faster will strengthen the pheromone amount in the shorter path more quickly, as shown in (b). The ants that took the long way will have more probability to come back using the shortest way, and after some time, they will converge toward using it. Consequently, the ants will find the shortest path by themselves, without having a global view of the ground. In time, most ants will choose the left path as shown in (c).

When applied to optimization problems, ACO uses a colony of artificial ants that behave as cooperative agents in a solution space were they are allowed to search and reinforce paths (solutions) in order to find the feasible ones. A solution that satisfies the problem constraint is *feasible*. After initialization of pheromone trails, ants construct incomplete feasible solutions, starting from random nodes, and then pheromone trails are updated. A node is an abstraction for the location of an ant, i.e., a nest or a food source. At each execution step, ants compute a set of possible moves and select the best one (according to some probabilistic rules) to carry out the rest of the tour. The transition probability is based on the heuristic information and pheromone trail level of the move. The higher the value of the pheromone and the heuristic information, the more profitable it is to select this move and resume the search. In the beginning, the initial

Algorithm 1 Pseudo-code of the canonical ACO algorithm

```
Procedure ACO
Begin
  Initialize the pheromone
  While (stopping criterion not satisfied) do
    Position each ant in a starting node
    Repeat
      For each ant do
        Chose next node by applying the state transition rate
      End for
    Until every ant has built a solution
    Update the pheromone
  End while
End
```

pheromone level is set to a small positive constant value τ_0 and then ants update this value after completing the construction stage. All ACO algorithms adapt the specific algorithm scheme shown in Algorithm 1.

After initializing the pheromone trails and control parameters, a main loop is repeated until the stopping criterion is met. The stopping criterion can be for example a certain number of iterations or a given time limit without improving the result. In the main loop, ants construct feasible solutions and update the associated pheromone trails. More precisely, partial problem solutions are seen as nodes: each ant starts from a random node and moves from a node i to another node j of the partial solution. At each step, the ant k computes a set of feasible solutions to its current node and moves to one of these expansions, according to a probability distribution. For an ant k the probability p_{ij}^k to move from a node i to a node j depends on the combination of two values:

$$\begin{cases} p_{ij}^k = \frac{\tau_{ij} \cdot \eta_{ij}}{\sum_{q \in allowed_k} \tau_{iq} \eta_{iq}} & if\ j \in allowed_k \\ p_{ij}^k = 0 & otherwise \end{cases}$$

where:

- η_{ij} is the attractiveness of the move and is computed by some heuristic information indicating a prior desirability of that move.
- τ_{ij} is the pheromone trail level of the move, indicating how profitable it has been in the past to make that particular move. The variable represents therefore a posterior indication of the desirability of that move.
- $allowed_k$ is the set of remaining feasible nodes.

Thus, the higher the pheromone value and the heuristic information, the more profitable it is to include state j in the partial solution. The initial pheromone level is a positive integer τ_0. In

nature, there is not any pheromone on the ground at the beginning (i.e., $\tau_0 = 0$). However, the ACO algorithm requires $\tau_0 > 0$, otherwise the probability to chose the next state would be $p_{ij}^k = 0$ and the search process would stop from the beginning. Furthermore, the pheromone level of the solution elements is changed by applying an update rule $\tau_{ij} \leftarrow \rho.\tau_{ij} + \Delta\tau_{ij}$, where $0 < \rho < 1$ models pheromone evaporation and $\Delta\tau_{ij}$ represents additional added pheromone. Usually, the quantity of the added pheromone depends on the desired quality for the solution.

In practice, to solve distributed job scheduling problems, the ACO algorithm assigns jobs to each available physical machine. Here, each job can be carried out by an ant. Ants then cooperatively search for example the less loaded machines with sufficient available computing power and transfer the jobs to these machines.

3. Related work

Indeed, the last decade has witnessed an astonishingly amount of research in improving bio-inspired techniques, specially ACO [34]. As shown in recent surveys [47, 24], the enhanced techniques have been increasingly applied to solve the problem of distributed job scheduling. However, with regard to job scheduling in Cloud environments, very few works can be found to date [31].

Concretely, the works in [2, 49] propose ACO-based Cloud schedulers for minimizing makespan and maximizing load balancing, respectively. An interesting aspect of [2] is that it was evaluated in a real Cloud using the Google App Engine [9] and Microsoft Live Mesh[1], whereas the other effort [49] was evaluated through simulations. However, during the experiments, [2] used only 25 jobs and a Cloud comprising 5 machines. Moreover, as our proposal, these two efforts support true *dynamic* resource allocation, i.e., the scheduler does not need to initially know the running time of the jobs to allocate or the details of the available resources. On the downside, the works ignore flowtime, rendering difficult their applicability to execute PSEs in semi-interactive scientific Cloud environments.

Another relevant approach based on Particle Swarm Optimization (PSO) is proposed in [33]. PSO is a bio-inspired technique that mimics the behavior of bird flocks, bee swarms and fish schools. Contrary to [2, 49], the approach is based on static resource allocation, which forces users to feed the scheduler with the estimated running times of jobs on the set of Cloud resources to be used. As we will show in Section 4, even when some user-supplied information is required by our algorithm, it only needs as input a *qualitative* indication of which jobs may take longer than others when run in any physical machine. Besides, [33] is targeted at paid Clouds, i.e., those that bill users for CPU, storage and network usage. As a consequence, the work only minimizes monetary cost, and does not consider either makespan or flowtime minimization.

Finally, the work in [19] address the problem of job scheduling in Clouds by employing PSO, while reducing energy consumption. Indeed, energy consumption has become a crucial problem [21], on one hand because it has started to limit further performance growth due to expensive electricity bills, and on the other hand, by the environmental impact in terms of carbon

[1] http://explore.live.com/windows-live-mesh-devices-sync-upgrade-ui/

dioxide (CO2) emissions caused by high energy consumption. This problem has in fact gave birth to a new field called Green Computing [21]. As such, [19] does not paid attention to flowtime either but interestingly it also achieves competitive makespan as evidenced by experiments performed via CloudSim [6], which is also used in this paper.

The next Section focuses on explaining our approach to bio-inspired Cloud scheduling in detail.

4. Approach overview

The goal of our scheduler is to minimize the weighted flowtime of a set of PSE jobs, while also minimizing makespan when using a Cloud. One novelty of the proposed scheduler is the association of a *qualitative* priority represented as an integer value for each one of the jobs of a PSE. Priorities are assigned in terms of the relative estimated execution time required by each PSE job. Moreover, priorities are provided by the user, who because of his experience, has arguably extensive knowledge about the problem to be solved from a modeling perspective, and therefore can estimate the individual time requirements of each job with respect to the rest of the jobs of his/her PSE. In other words, the user can identify which are the individual experiments within a PSE requiring more time to obtain a solution.

The estimated execution times of each experiment depends on the particular problem at hand and chosen PSE variable values. Once the user has identified the experiments that may require more time to be executed, a simple tagging strategy is applied to assign a "category" (number) to each job. These categories represent the priority degree that will have a job with respect to the others in the same PSE, e.g., high priority, medium priority or low priority, but other categories could be defined. Despite jobs can demand different times to execute and the user can identify those that take longer, the number of categories should be limited for usability reasons.

Conceptually, the scheduling problem to tackle down can be formulated as follows. A PSE is formally defined as a set of $N = 1, 2, ..., n$ independent jobs, where each job corresponds to a particular value for a variable of the model being studied by the PSE. The jobs are executed on m Cloud machines. Each job j has an associated priority value, i.e., a degree of importance, which is represented by a weight w_j to each job. This priority value is taken into account by the scheduler to determine the order in which jobs will be executed at the VM level. The scheduler processes the jobs with higher priority (or heavier) first and then the remaining jobs. The larger the estimated size of a job in terms of execution time, the higher priority weight the user should associate to the job. This is the opposite criterion to the well-known Shortest Job First (SFJ) scheduler from operating systems, whose performance is ideal. SFJ deals with the "convoy effect" precisely by prioritizing shorter jobs over heavier ones. Finally, in our scheduler, jobs having similar estimated execution times should be assigned the same priority.

Our scheduler is designed to deliver the shortest total weighted flowtime (i.e., the sum of the weighted completion time of all jobs) along with minimum makespan (i.e., the completion time of the last job finished). The flowtime of a job –also known as the response time– is the total time that a job spends in the system, thus it is the sum of times the job is waiting in queues plus its effective processing time. When jobs have different degrees of importance, indicated by the weight of the job, the total weighted flowtime is one of the simplest and natural metrics that

measures the throughput offered by a schedule S and is calculated as $\sum_j^n (C_j(S) - A_j(S)).w_j$, where C_j is the completion time of job j, A_j is the arrival time of job j and w_j is the weight associated to job j. Furthermore, the completion time of job j in schedule S can be denoted by $C_j(S)$ and hence the makespan is $C_{max}(S) = max_j C_j(S)$.

Figure 3 conceptually illustrates the sequence of actions from the time of creating VMs for executing a PSE within a Cloud until the jobs are processed and executed. The *User* entity represents both the creator of a virtual Cloud (i.e., VMs on top of physical hosts) and the disciplinary user that submit their experiments for execution. The *Datacenter* entity manages a number of *Hosts* entities, i.e., a number of physical resources. The *VMs* are allocated to hosts through an *AllocationPolicy*, which implements the SI-based part of the scheduler proposed in this paper. Finally, after setting up the virtual infrastructure, the user can send his/her experiments to be executed. The jobs will be handled through a *JobPriorityPolicy* that will take into account the priorities of jobs at the moment they are sent to an available VM already issued by the user and allocated to a host. As such, our scheduler operates at two levels: Cloud-wide or Datacenter level, where SI techniques (currently ACO) are used to allocate user VMs to resources, and VM-level, where the jobs assigned to a VM are handled according to their priority.

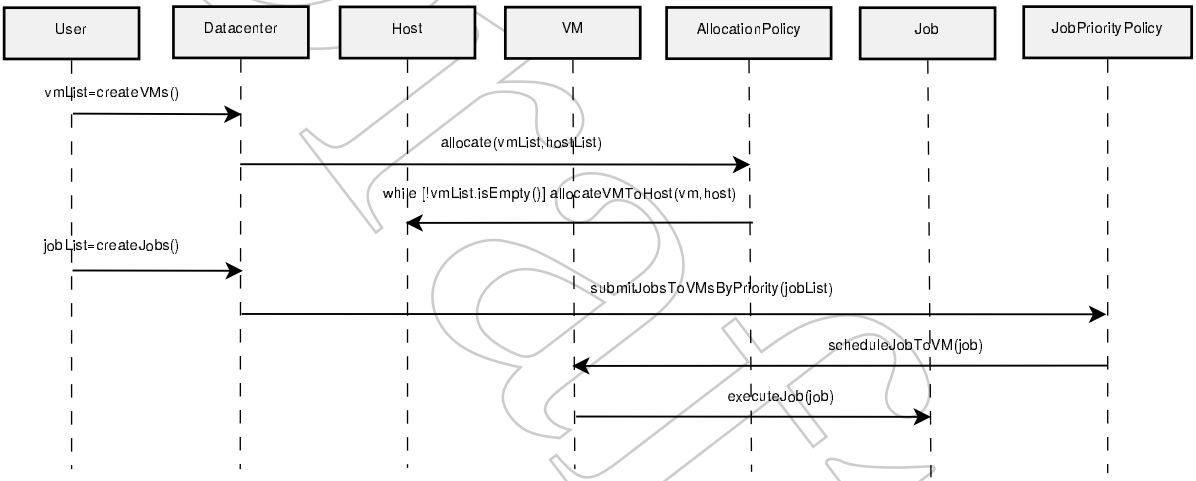


Figure 3: Sequence diagram of scheduling actions within a Private Cloud

4.1. Algorithm implementation

To implement the Cloud-level logic of the scheduler, AntZ, the algorithm proposed in [23] to solve the problem of load balancing in Grid environments has been adapted to be employed in Clouds (see Algorithm 2). AntZ combines the idea of how ants cluster objects with their ability to leave pheromone trails on their paths so that it can be a guide for other ants passing their way.

In our adapted algorithm, each ant works independently and represents a VM "looking" for the best host to which it can be allocated. The main procedure performed by an ant is shown in Algorithm 2. When a VM is created, an ant is initialized and starts to work. A master table containing information on the load of each host is initialized (`initializeLoadTable()`). Subsequently, if an ant associated to the VM that is executing the algorithm already exists, the

ant is obtained from a pool of ants through `getAntPool(vm)` method. If the VM does not exist in the ant pool, then a new ant is created. To do this, first a list of all suitable hosts in which can be allocated the VM is obtained. A host is suitable if it has an amount of processing power, memory and bandwidth greater than or equal to that of required by the wandering VM.

Then, the working ant with the associated VM is added to the ant pool (`antPool.add(vm,ant)`) and the ACO-specific logic starts to operate (see Algorithm 3). In each iteration of the subalgorithm, the ant collects the load information of the host that is visiting –through the `getHostLoadInformation()` operation– and adds this information to its private load history. The ant then updates a load information table of visited hosts (`localLoadTable.update()`), which is maintained in each host. This table contains information of the own load of an ant, as well as load information of other hosts, which were added to the table when other ants visited the host. Here, load refers to the total CPU utilization within a host.

When an ant moves from one host to another has two choices. One choice is to move to a random host using a constant probability or *mutation rate*. The other choice is to use the load table information of the current host (`chooseNextStep()`). The mutation rate decreases with a *decay rate* factor as time passes, thus, the ant will be more dependent on load information than to random choice. This process is repeated until the finishing criterion is met. The completion criterion is equal to a predefined number of steps (*maxSteps*). Finally, the ant delivers its VM to the current host and finishes its task. When the ant has not completed its work, i.e., the ant can not allocate its associated VM to a host, then Algorithm 2 is repeated with the same ant until the ant finally achieves the "finished" state. Prior to repetition, an exponential back-off strategy to wait for a while is performed.

Every time an ant visits a host, it updates the host load information table with the information of other hosts, but at the same time the ant collects the information already provided by the table of that host, if any. The load information table acts as a pheromone trail that an ant leaves while it is moving, in order to guide other ants to choose better paths rather than wandering randomly in the Cloud. Entries of each local table are the hosts that ants have visited on their way to deliver their VMs together with their load information.

When an ant reads the information in the load table in each host and chooses a direction via Algorithm 4, the ant chooses the lightest loaded host in the table, i.e., each entry of the load information table is evaluated and compared with the current load of the visited host. If the load of the visited host is smaller than any other host provided in the load information table, the ant chooses the host with the smallest load, and in case of a tie the ant chooses one with an equal probability.

To calculate the load, the original AntZ algorithm receives the number of jobs that are executing in the resource in which the load is being calculated. In the proposed algorithm, the load is calculated on each host taking into account the CPU utilization made by all the VMs that are executing on each host. This metric is useful for an ant to choose the least loaded host to allocate its VM.

Once the VMs have been created and allocated in physical resources, the scheduler proceeds to assign the jobs to user VMs. To do this, a strategy where the jobs with higher priority value are assigned first to the VMs is used (see Algorithm 5). This represents the second scheduling level of the scheduler proposed as a whole.

Algorithm 2 ACO-based allocation algorithm for individual VMs

```
Procedure ACOallocationPolicy(vm, hostList)
Begin
  initializeLoadTable()
  ant=getAntPool(vm)
  if (ant==null) then
    suitableHosts=getSuitableHostsForVm(hostList,vm)
    ant=new Ant(vm, suitableHosts)
    antPool.add(vm, ant)
  end if
  repeat
    ant.AntAlgorithm()
  until ant.isFinish()
  allocatedHost=hostList.get(ant.getHost())
  allocatedHost.allocateVM(ant.getVM())
End
```

Algorithm 3 ACO-specific logic

```
Procedure AntAlgorithm()
Begin
  step=1
  initialize()
  While (step < maxSteps) do
    currentLoad=getHostLoadInformation()
    AntHistory.add(currentLoad)
    localLoadTable.update()
    if (random() < mutationRate) then
      nextHost=randomlyChooseNextStep()
    else
      nextHost=chooseNextStep()
    end if
    mutationRate=mutationRate-decayRate
    step=step+1
    moveTo(nextHost)
  end while
  deliverVMtoHost()
End
```

Algorithm 4 ACO-specific logic: The ChooseNextStep procedure

```
Procedure ChooseNextStep()
Begin
  bestHost=currentHost
  bestLoad=currentLoad
  for each entry in hostList
    if (entry.load < bestLoad) then
      bestHost=entry.host
    else if (entry.load = bestLoad) then
      if (random.next < probability) then
        bestHost=entry.host
      end if
    end if
  end for
End
```

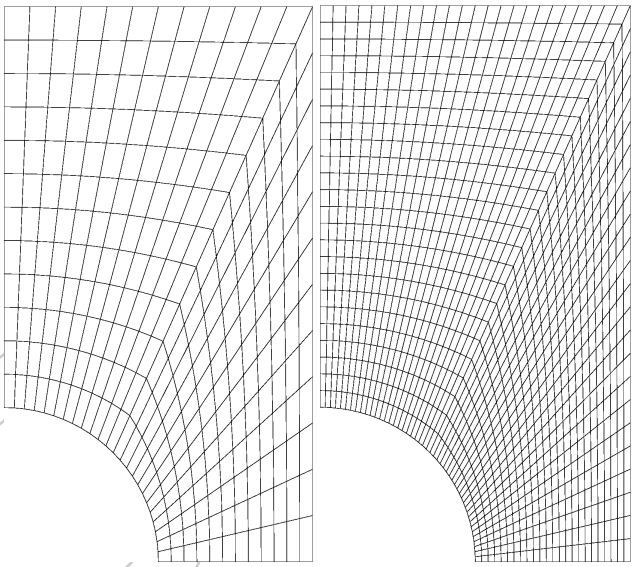
Algorithm 5 The SubmitJobsToVMsByPriority procedure

```
Procedure SubmitJobsToVMsByPriority(jobList)
Begin
  vmIndex=0
  while (jobList.size() > 0)
    job=jobList.getJobByPriority()
    vm=getVMsList(vmIndex)
    vm.scheduleJobToVM(job)
    vmIndex=Mod(vmIndex+1,getVMsList().size())
    jobList.remove(job)
  end while
End
```

To carry out the assignment of jobs to VMs, this subalgorithm uses two lists, one containing the jobs that have been sent by the user, and the other list contains all user VMs that are already allocated to a physical resource and hence are ready to execute jobs. The procedure starts iterating the list of all jobs –`jobsList`– and then, through `getJobByPriority()` method retrieves jobs according to their priority value, this means, jobs with the highest priority first, then jobs with medium priority value, and finally jobs with low priority. Each time a job is obtained from the `jobList` it is submitted to be executed in a VM by a round robin method. The VM where the job is executed is obtained through the method `getVmsList(vmIndex)`. Internally, the algorithm maintains a queue for each VM that contains a list of jobs that have been assigned to be executed. The procedure is repeated until all jobs have been submitted for execution, i.e., when the `jobList` is empty.

5. Evaluation

In order to assess the effectiveness of our proposal for executing PSEs on Clouds, we have processed a real case study for solving a very well-known benchmark problem proposed in the



(a) 288 elements (b) 1,152 elements

Figure 4: Considered input data meshes

literature, see [1] for instance. Broadly, the experimental methodology involved two steps. First, we executed the problem in a single machine by varying an individual problem parameter by using a finite element software, which allowed us to gather real job data, i.e., processing times and input/output file data sizes (see Section 5.1). By means of the generated job data, we instantiated the CloudSim simulation toolkit, which is explained in Section 5.2. Lastly, the obtained results regarding the performance of our proposal compared with some Cloud scheduling alternatives are reported in Section 5.3.

5.1. Real job data gathering

The problem explained in [1] involves studying a plane strain plate with a central circular hole. The dimensions of the plate were 18 x 10 m, with $R = 5$ m. On the other hand, material constants considered were $E = 2.1\ 10^5$ Mpa, $\nu = 0.3$, $\sigma_y = 240$ Mpa and $H = 0$. A linear Perzyna viscoplastic model with $m = 1$ and $n = \infty$ was considered. Unlike previous studies of our own [7], in which a geometry parameter –particularly imperfection– was chosen to generate the PSE jobs, in this case a material parameter was selected as the variation parameter. Then, 25 different viscosity values for the η parameter were considered, namely 1.10^4, 2.10^4, 3.10^4, 4.10^4, 5.10^4, 7.10^4, 1.10^5, 2.10^5, 3.10^5, 4.10^5, 5.10^5, 7.10^5, 1.10^6, 2.10^6, 3.10^6, 4.10^6, 5.10^6, 7.10^6, 1.10^7, 2.10^7, 3.10^7, 4.10^7, 5.10^7, 7.10^7 and 1.10^8 Mpa. Useful and introductory details on viscoplastic theory and numerical implementation can be found in [35, 14].

The two different finite element meshes displayed in Figure 4 were tested. In both cases, Q1/P0 elements were chosen. Imposed displacements (at y=18m) were applied until a final displacement of 2,000 mm was reached in 400 equal time steps of 0.05 mm each. It is worth noting that $\delta = 1$ has been set for all the time steps.

After establishing the problem parameters, we employed a single real machine to run the parameter sweep experiment by varying the viscosity parameter η as indicated above and measuring the execution time for the 25 different experiments, which resulted in 25 input files with different input configurations and 25 output files for either meshes. The tests were solved using the SOGDE finite element solver software [13]. Furthermore, the characteristics of the real machine on which the executions were carried out are shown in Table 1. The machine model was AMD Athlon(tm) 64 X2 Dual Core Processor 3600+, equipped with the Linux operating system (specifically the Ubuntu 11.04 distribution) running the generic kernel version 2.6.38-8. It is worth noting that only one core was used during the experiments since individual jobs did not exploit multicore capabilities.

Feature	Value
CPU power	4,008.64 bogoMIPS
Number of CPUs	2
RAM memory	2 GBytes
Storage size	400 GBytes
Bandwidth	100 Mbps

Table 1: Machine used to execute the real PSE: Characteristics

The information regarding machine processing power was obtained from the native benchmarking support of Linux and as such is expressed in bogoMIPS [43]. BogoMIPS (from *bogus* and MIPS) is a metric that indicates how fast a machine processor runs. Since the real tests were performed on a machine running the Linux operating system, we have considered to use the bogoMIPS measure which is as we mentioned the one used by this operating system to approximate CPU power.

Then, the simulations were carried out by taking into account the bogoMIPS metric for measuring simulated jobs CPU instructions. Once the execution times were obtained from the real machine, we approximated for each experiment the number of executed instructions by the following formula:

$$NI_j = bogomipsCPU * T_j$$

where,

- NI_j is the number of million instructions to be executed by, or associated to, a job j.
- $bogomipsCPU$ is the processing power of the CPU of our real machine measured in bogoMIPS.
- T_j is the time that took to run the job j on the real machine.

Next is an example of how to calculate the number of instructions of a job corresponding to the mesh of 1,152 elements that took 117 seconds to be executed. The machine where the experiment was executed had a processing power of 4,008.64 bogoMIPS. Then, the approximated number of instructions for the job was 469,011 MI (Million Instructions). Details about all jobs execution times and lengths can be seen in Table 2.

5.2. CloudSim instantiation

First, the CloudSim simulator [6] was configured with a datacenter composed of a single machine –or "host" in CloudSim terminology– with the same characteristics as the real machine where the experiments were performed. The characteristics of the configured host are shown in Table 3 (left). Here, processing power is expressed in MIPS (Million Instructions Per Second), RAM memory and Storage capacity are in MBytes, bandwidth is expressed in Mbps, and finally, PEs is the number of processing elements (cores) of the host. Each PE has the same processing power.

Host Parameters	Value
Processing Power	4,008 bogoMIPS
RAM	4 Gbytes
Storage	409,600
Bandwidth	100 Mbps
PEs	2

VM Parameters	Value
MIPS	4,008 bogoMIPS
RAM	1 Gbyte
Machine Image Size	102,400
Bandwidth	25 Mbps
PEs	1
VMM (Virtual Machine Monitor)	Xen

Table 3: Simulated Cloud machines characteristics. Host parameters (left) and VM parameters (right)

Once configured, we checked that the execution times obtained by the simulation coincided or were close to real times for each independent job performed on the real machine. The results were successful in the sense that one experiment (i.e., a variation in the value of η) took 117 seconds to be solved in the real machine, while in the simulated machine the elapsed time was 117.02 seconds. Once the execution times have been validated for a single machine on CloudSim, a new simulation scenario was set. This new scenario consisted of a datacenter with 10 hosts, where each host has the same hardware capabilities as the real single machine, and 40 VMs, each with the characteristics specified in Table 3 (right). This is a moderately-sized, homogeneous datacenter that can be found in many real scenarios.

For each mesh, we evaluated the performance of their associated jobs in the simulated Cloud as we increased the number of jobs to be performed, i.e., $25 * i$ jobs with $i = 10, 20, ..., 100$. This is, the base job set comprising 25 jobs obtained by varying the value of η was cloned to obtain more sets.

Parameter η	Mesh of 288 elements		Mesh of 1,152 elements	
	Execution Time (secs.-mins.)	Length (MIPS)	Execution Time (secs.-mins.)	Length (MIPS)
1.10^4	24 - 0.40	96,207	90 - 1.50	360,778
2.10^4	25 - 0.41	100,216	88 - 1.47	352,760
3.10^4	24 - 0.40	96,207	87 - 1.45	348,752
4.10^4	24 - 0.40	96,207	87 - 1.45	348,752
5.10^4	25 - 0.41	100,216	89 - 1.48	356,769
7.10^4	25 - 0.41	100,216	88 - 1.47	352,760
1.10^5	25 - 0.41	100,216	88 - 1.47	352,760
2.10^5	25 - 0.41	100,216	85 - 1.42	340,734
3.10^5	26 - 0.43	104,225	102 - 1.70	408,881
4.10^5	22 - 0.36	88,190	117 - 1.95	469,011
5.10^5	20 - 0.33	80,173	70 - 1.17	280,605
7.10^5	20 - 0.33	80,173	73 - 1.22	292,631
1.10^6	20 - 0.33	80,173	74 - 1.23	296,639
2.10^6	20 - 0.33	80,173	73 - 1.22	292,631
3.10^6	20 - 0.33	80,173	72 - 1.20	288,622
4.10^6	20 - 0.33	80,173	70 - 1.17	280,605
5.10^6	20 - 0.33	80,173	62 - 1.03	248,536
7.10^6	20 - 0.33	80,173	61 - 1.02	244,527
1.10^7	20 - 0.33	80,173	62 - 1.03	248,536
2.10^7	17 - 0.28	68,147	65 - 1.08	260,562
3.10^7	13 - 0.21	52,112	65 - 1.08	260,562
4.10^7	13 - 0.21	52,112	68 - 1.13	272,588
5.10^7	13 - 0.21	52,112	68 - 1.13	272,588
7.10^7	13 - 0.21	52,112	70 - 1.17	280,605
1.10^8	13 - 0.21	52,112	71 - 1.18	280,613

Table 2: Real jobs execution times and lengths: Meshes of 288 and 1,152 elements

Each job, called *cloudlet* by CloudSim, had the characteristics shown in Table 4 (left), where Length parameter is the number of instructions to be executed by a cloudlet, which varied between 52,112 and 104,225 MIPS for the mesh of 288 elements and between 244,527 and 469,011 for the mesh of 1,152 elements (see Table 2). Moreover, PEs is the number of processing elements (cores) required to perform each individual job. Input size and Output size are the input file size and output file size in bytes, respectively. As shown in Table 4 (left) the experiments corresponding to the mesh of 288 elements had input files of 40,038 bytes, and the experiments corresponding to the mesh of 1,152 elements had input files of 93,082 bytes. A similar distinction applies to the output file sizes. Finally, Table 4 (right) shows the priorities assigned to Cloudlets. For the sake of realism, the base job set comprised 7, 11 and 7 job with low, medium and high priority, respectively. This is, usually most PSE jobs take similar execution times, except for less cases, where smaller and higher execution times are registered.

Cloudlet parameters	Value	
	Mesh of 288 elems.	Mesh of 1,152 elems.
Length (MIPS)	52,112-104,225	244,527-469,011
PEs	1	1
Input size (bytes)	40,038	93,082
Output size (bytes)	722,432	2,202,010

Cloudlet priority	Value	
	Mesh of 288 elems.	Mesh of 1,152 elems.
Low ($w_j = 1$)	52,112-68,147	244,527-272,588
Medium ($w_j = 2$)	80,173-96,207	280,605-348,752
High ($w_j = 3$)	100,216-104,225	352,760-469,011

Table 4: Cloudlet configuration used in the experiments. CloudSim built-in parameters (left) and job priorities (right)

In CloudSim, the amount of available hardware resources to each VM is constrained by the total processing power and available system bandwidth within the associated host. Thus, scheduling policies must be applied in order to appropriately assign Cloudlets (and hence VMs) to hosts and achieve efficient use of resources. On the other hand, Cloudlets can be configured to be scheduled according to some scheduling policy that can both determine how to indirectly assign them to hosts and in which order to process the list of Cloudlets within a host. This allow us to experiment with custom Cloud schedulers such as the one proposed in this paper, which was in fact compared against CloudSim built-in schedulers. The next section explains the associated obtained results in detail.

5.3. Experiments

In this subsection we report on the obtained results when executing the PSE in the simulated Cloud by using our two-level scheduler and two classical Cloud scheduling policies for assigning VMs to hosts and handling jobs. Due to their high CPU requirements, and the fact that each VM requires only one PSE, we assumed a 1-1 job-VM execution model, i.e., although each VM

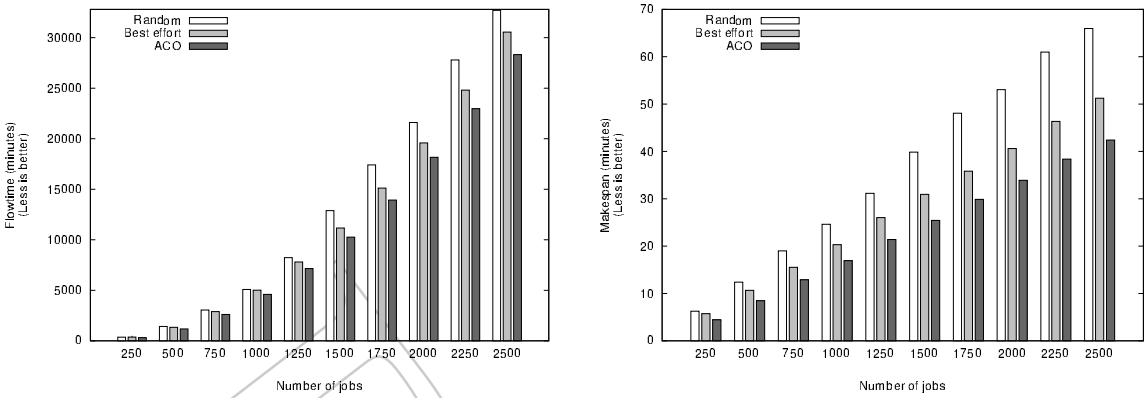
Scheduler	Mesh of 288 elements		Mesh of 1,152 elements	
	Flowtime (mins.)	Makespan (mins.)	Flowtime (mins.)	Makespan (mins.)
Random	153,707.56	366.20	569,335.78	1,373.44
Random (priority)	130,539.85	361.33	482,562.31	1,373.19
Gain (0-100%)	15.07	1.32	15.24	0.02
Best effort	137,378.79	283.89	513,783.00	1,084.60
Best effort (priority)	118,646.36	283.20	435,985.39	1,086.24
Gain (0-100%)	13.63	0.24	15.14	-0.15
ACO	127,270.74	233.50	476,230.77	894.69
ACO (priority)	109,477.35	234.13	401,806.35	896.54
Gain (0-100%)	13.98	-0.27	15.62	-0.20

Table 5: Using the VM-level priority policy: Effects on flowtime and makespan

can have in its waiting queue many jobs, they are executed one at a time. Concretely, apart from our proposal, we employed a random policy, and a "best effort" (CloudSim built-in) policy, which upon executing a job assigns the corresponding VMs to the Cloud machine which has the highest number of free PEs. In our scheduler, we set the ACO-specific parameters –i.e., mutation rate, decay rate and maximum steps– as suggested in [23]. For simplicity, from now on, we will refer to the term "weighted flowtime" as "flowtime". In addition, although our scheduler is independent from the SI technique exploited at the Cloud-level, it will be referred to as "ACO".

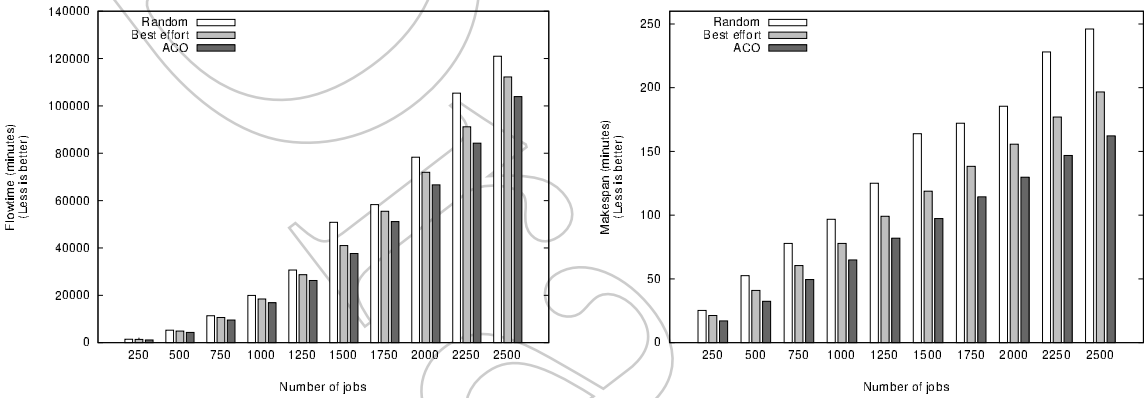
In all cases, the competing policies were also complemented with the VM-level policy for handling jobs within VMs, or the VMs allocated to a single host. As shown in Table 5, regardless the VM allocation policy used or the experiment size (i.e., mesh), considering job priority information yielded as a result important gains with respect to accumulated flowtime, i.e., the weighted flowtime resulted from simulating the execution of $25 * i$ jobs with $i = 10, 20, ..., 100$ of various priorities. For example, for the mesh of 288 elements, gains in the range of 13.63-15.07% compared to not considering priorities were obtained, whereas for the mesh of 1,152, the gains were in the range of 15.14-15.62%. Interestingly, except for few cases were some overhead was obtained (the cells with negative percentage values), the accumulated makespan was not significantly affected. This means that taking into account job priority information at the VM level for scheduling jobs improves weighted flowtime without compromising makespan. Therefore, the rest of the experiments were performed by using the variants exploiting job priority information.

Furthermore, Figures 5 and 6 illustrate the obtained flowtime and makespan by the three schedulers using the VM-level priority policy for both meshes. Graphically, it can be seen that, irrespective of the mesh, flowtime and makespan presented exponential and linear tendencies, re-



(a) Flowtime (b) Makespan

Figure 5: Results as the number of jobs increases: Mesh of 288 elements



(a) Flowtime (b) Makespan

Figure 6: Results as the number of jobs increases: Mesh of 1,152 elements

spectively. All in all, our algorithm performed rather well compared to its competitors regarding the two performance metrics taken. Table 6 shows the reductions or gains obtained by ACO with respect to "best effort". An observation is that, for the mesh of 288 elements, the highest gains in terms of flowtime of ACO compared to the "best effort" policy were achieved for 250-750 jobs, with gains of up to 16%. For 1,000 jobs and beyond, the flowtime gains converged around 7-8%. This is sound since adding more jobs to the environment under test ends up saturating its execution capacity, and thus scheduling decisions have less impact on the outcome. For the mesh of 1,152 elements, on the other hand, a similar behavior was observed when executing 1,000 or more jobs. For 750 or less jobs, flowtime gains were 10-17 %. Despite having heavier jobs (from a computational standpoint) makes computations to stay longer within the Cloud and may impact on flowtime, ACO maintained the gain levels.

Although our aim is to reduce flowtime while being competitive in terms of makespan, from Table 6 it can be seen that important makespan gains were obtained as well. However, compared

#jobs	Mesh of 288 elements		Mesh of 1,152 elements	
	Flowtime	Makespan	Flowtime	Makespan
250	16.43	22.45	17.00	19.87
500	12.03	20.55	11.94	20.76
750	9.98	16.92	9.96	18.15
1,000	8.37	16.60	8.45	16.64
1,250	8.42	17.76	8.63	17.33
1,500	8.11	17.81	8.19	18.06
1,750	7.84	16.63	7.86	17.28
2,000	7.26	16.48	7.37	16.65
2,250	7.41	17.17	7.58	17.03
2,500	7.30	17.27	7.42	17.50

Table 6: Results as the number of jobs increases: Percentage gains of ACO with respect to "best effort"

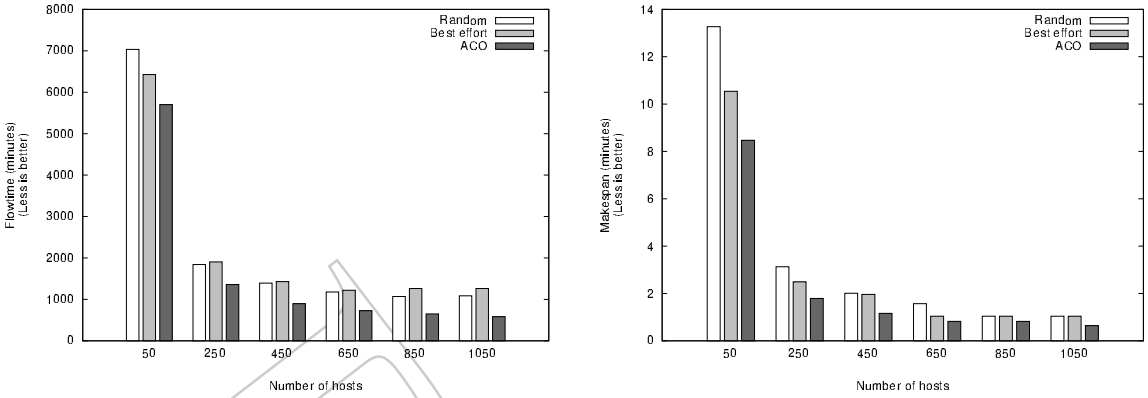
to flowtime, gains were more uniform. Concretely, for the mesh of 288 elements, ACO outperformed "best effort" by 22.45% and 20.55% when executing 250 and 500 jobs, respectively. When running more jobs, the gains were in the range of 16-17%. For the mesh of 1,152, the gains were approximately in the range of 17-19% irrespective of the number of jobs used.

In a second round of experiments, we measured the effects of varying the number of hosts while keeping the number of jobs to execute fixed. The aim of this experiment is to assess the effects of increasing hosts, which is commonly known as *horizontal scalability* or *scaling out*, in the performance of the algorithms. Particularly, we evaluated the three policy-aware scheduling alternatives by using $50 * i$ hosts with $i = 1, 5, 9, 13, 17, 21$. These values were chosen to be consistent with the ones employed in the horizontal scalability analysis performed on the original AntZ algorithm [23]. The hosts and VMs characteristics were again the ones shown in Table 3. Besides, the number of VMs in each case increased accordingly. On the other hand, the number of jobs were set at the maximum used in the previous experiment (2,500), which means 100 PSEs of 25 jobs each. Again, within each PSE, there were 7, 11 and 7 jobs tagged with low, medium and high priority.

Figure 7 shows the obtained results for the mesh of 288 elements. As illustrated, ACO performed very well. In terms of flowtime, it can be seen that "best effort" (gray bar) could not exploit resources for 650 hosts onwards, as the delivered flowtime is around the same value. Moreover, a similar situation occurs with makespan. As one might expect, arbitrarily growing the size of the used Cloud benefited the random policy, which obtained gains in terms of the tested performance metrics up to 850 hosts. However, the best curves were obtained with ACO,

This is a preprint of the article: "C. Mateos, E. Pacini and C. GarcÃ-a Garino: "An ACO-inspired Algorithm for Minimizing Weighted Flowtime in Cloud-based Parameter Sweep Experiments". Advances in Engineering Software. Vol. 56, pp. 38-50. Elsevier. 2013. ISSN 0965-9978."

The final publication is available at http://dx.doi.org/10.1016/j.advengsoft.2012.11.011



(a) Flowtime (b) Makespan

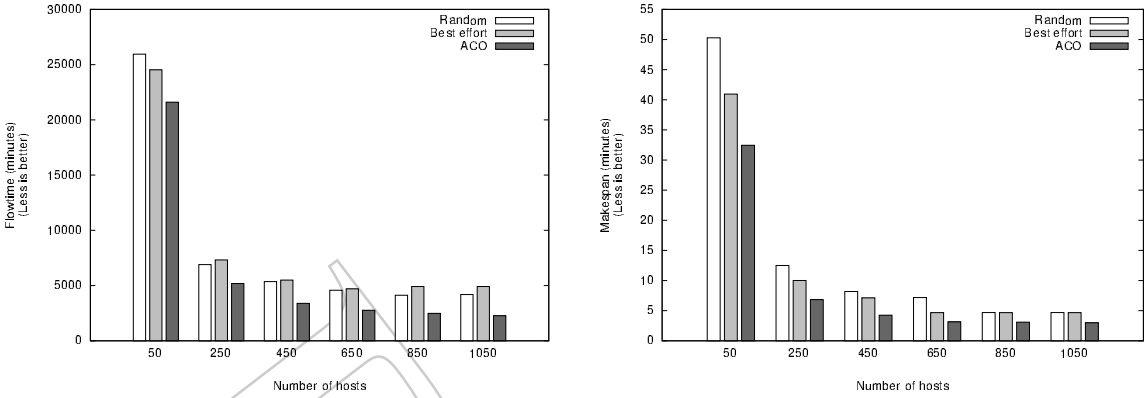
Figure 7: Results as the number of hosts increases: Mesh of 288 elements

which although marginal, also obtained performance gains when using 1,050 hosts. It is worth noting that the goal of this experiment is not to study the number of hosts to which the flowtime and makespan curves converge, which is in fact not useful as it would depend on the parameters established for the jobs and the simulated hardware used. In contrast, the experiment shows that ACO cleanly supports scaling out compared to the analyzed alternatives.

Complementary, Figure 8 shows the resulting flowtimes and makespan for the mesh of 1,152 elements. Although the curves are very similar to the case of the mesh of 288 elements, some differences in the average makespan were obtained. Concretely, the average gain of ACO with respect to "best effort", taken as $\left[\sum_{j=50,250,450,650,850,1050} \frac{(makespan_j(bestEffort) - makespan_j(ACO)}{(makespan_j(bestEffort))}\right]/6$, yielded as a result 30.43% and 26.79% for the mesh of 1,152 and 288 elements, respectively. The same applies to ACO versus Random (41.33% and 37.90%). This is since our support considers host CPU utilization and not just hardware capabilities to make scheduling decisions, it is able to better exploit computational power or Cloud PEs. It is worth noting that CPU utilization is different from regular CPU load [27]. Within a single host, the former metric provides trend information of CPU usage, but not just the length of the queue maintaining the jobs (or VMs) waiting for taking possession of the PEs as the latter metric does. For larger jobs, more extensive use of resources is done, and thus CPU utilization in resources is close to 100% most of the time. Then, ACO is able to quickly schedule VMs to hosts whose CPU utilization is below these levels, thus increasing efficiency.

6. Conclusions

Parameter Sweep Experiments (PSE) is a type of numerical simulation that involves running a large number of independent jobs and typically requires a lot of computing power. These jobs must be efficiently processed in the different computing resources of a distributed environment such as the ones provided by Cloud. Consequently, job scheduling in this context indeed plays a fundamental role.



(a) Flowtime (b) Makespan

Figure 8: Results as the number of hosts increases: Mesh of 1,152 elements

In the last ten years or so, bio-inspired computing has been received increasing attention in the research community. Bio-inspired computing (also known as Swarm Intelligence) refers to the collective behavior that emerges from a swarm of social insects. Social insect colonies solve complex problems collectively by intelligent methods. These problems are beyond the capabilities of each individual insect, and the cooperation among them is largely self-organized without any supervision. Through studying social insect colonies behaviors such as ant colonies, researchers have proposed some algorithms or theories for combinatorial optimization problems. Moreover, job scheduling in Clouds is also a combinatorial optimization problem, and some bio-inspired schedulers have been proposed. Basically, researchers have introduced changes to the traditional bio-inspired techniques to achieve different Cloud scheduling goals, i.e., minimize makespan, maximize load balancing, minimize monetary cost or minimize energy consumption.

However, existing efforts fail at effectively handling PSEs in scientific Cloud environments since, to the best of our knowledge, no effort aimed at minimizing flowtime exists. Achieving low flowtime is important since it means a more agile human processing of PSE job results. Therefore, we have presented a new Cloud scheduler based on SI and particularly Ant Colony Optimization that pays special attention to this aspect. Simulated experiments performed with the help of the well-established CloudSim toolkit and real PSE job data show that our scheduler can handle a large number of jobs effectively, achieving interesting gains in terms of flowtime and makespan compared to classical Cloud scheduling policies.

We are extending this work in several directions. We will explore the ideas exposed in this paper in the context of other bio-inspired techniques, particularly Particle Swarm Optimization, which is also extensively used to solve combinatorial optimization problems. As a starting point, we will implement the first scheduling level based on an adaptation of the ParticleZ [23] Grid scheduling algorithm so as to target Clouds. Eventually, we will materialize the resulting job schedulers on top of a real (but not simulated) Cloud platform, such as Emotive Cloud [15], which is designed for extensibility.

Another issue concerns energy consumption. Clearly, simpler scheduling policies (e.g., random or “best effort”) require fairly less CPU usage, memory accesses and network transfers

compared to more complex policies such as our algorithm. For example, we need to maintain local and remote host load information for ants, which requires those resources. Therefore, when running many jobs, the accumulated resource usage overhead may be arguably significant, resulting in higher demands for energy. Then, we will study the flowtime/makespan vs energy consumption tradeoff in order to consider this problem.

Finally, there is an outstanding and increasing number of available mobile devices such as smartphones. Nowadays, mobile devices have a remarkable amount of computational resources that allows them to execute complex applications, such as 3D games, and to store large amounts of data. Recent work has experimentally shown the feasibility of using smartphones for running CPU-bound scientific codes [37]. Due to these advances, emergent research lines have aimed at integrating smartphones and other kind of mobile devices into traditional distributed computational environments, like clusters and Grids [38], to play the role of job executing "machines". It is not surprising that this research could also span scientific Clouds as well. However, intuitively, job scheduling in these highly heterogeneous environments will be more challenging since mobile devices rely on unreliable wireless connections and batteries, which is necessary to consider at the scheduling level. This will open the door to excellent research opportunities for new schedulers based both on traditional techniques and particularly SI-based algorithms.

Acknowledgments

We thank the anonymous referees for their helpful comments to improve the theoretical background and quality of this paper. We also acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312. The second author acknowledges her Ph.D. fellowship granted by the PRH-UNCuyo Project.

References

[1] G. Alfano, F. D. Angelis, L. Rosati, General solution procedures in elasto-viscoplasticity, Computer Methods in Applied Mechanics and Engineering 190 (39) (2001) 5123–5147.

[2] S. Banerjee, I. Mukherjee, P. Mahanti, Cloud Computing initiative using modified ant colony framework, World Academy of Science, Engineering and Technology (2009) 221–224.

[3] J. Basney, M. Livny, P. Mazzanti, Harnessing the capacity of Computational Grids for high energy physics, in: International Conference on Computing in High Energy and Nuclear Physics (CHEP 2000), 2000.

[4] E. Bonabeau, M. Dorigo, G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems, Oxford University Press, 1999.

[5] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616.

[6] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, 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) (2011) 23–50.

[7] C. Careglio, D. Monge, E. Pacini, C. Mateos, A. Mirasso, C. García Garino, Sensibilidad de resultados del ensayo de tracción simple frente a diferentes tamaños y tipos de imperfecciones, Mecánica Computacional XXIX (41) (2010) 4181–4197.

[8] T. L. Casavant, J. G. Kuhl, A taxonomy of scheduling in general-purpose distributed computing systems, IEEE Transactions on Software Engineering 14 (2) (1988) 141–154.

[9] A. de Jonge, Essential App Engine: Building High-Performance Java Apps with Google App Engine, Addison-Wesley Professional, 2011.

[10] M. Dorigo, Optimization, Learning and Natural Algorithms, Ph.D. thesis, Politecnico di Milano, Italy (1992).

[11] M. Dorigo, T. Stützle, The ant colony optimization metaheuristic: Algorithms, applications, and advances, in: F. Glover, G. Kochenberger (eds.), Handbook of Metaheuristics, vol. 57 of International Series in Operations Research & Management Science, Springer New York, 2003, pp. 250–285.

[12] S. Fidanova, M. Durchova, Ant algorithm for Grid scheduling problem, in: 5th International Conference on Large-Scale Scientific Computing, Springer, 2005, pp. 405–412.

[13] C. García Garino, F. Gabaldón, J. M. Goicolea, Finite element simulation of the simple tension test in metals, Finite Elements in Analysis and Design 42 (13) (2006) 1187–1197.

[14] C. García Garino, M. Ribero Vairo, S. Andía Fagés, A. Mirasso, J.-P. Ponthot, Numerical simulation of finite strain viscoplastic problems, in: M. Hogge et al. (ed.), Fifth International Conference on Advanced Computational Methods in ENgineering (ACOMEN 2011), University of Liege, 2011, pp. 1–10.

[15] I. Goiri, J. Guitart, J. Torres, Elastic management of tasks in virtualized environments, in: XX Jornadas de Paralelismo (JP 2009), 2009, pp. 671–676.

[16] M. Gulamali, A. Mcgough, S. Newhouse, J. Darlington, Using ICENI to run parameter sweep applications across multiple Grid resources, in: Global Grid Forum 10, Case Studies on Grid Applications Workshop, GGF10, 2004.

[17] Y. Hui, S. Xue-Qin, L. Xing, W. Ming-Hui, An improved ant algorithm for job scheduling in Grid computing, in: International Conference on Machine Learning and Cybernetics, vol. 5, IEEE Computer Society, 2005, pp. 2957–2961.

[18] H. Izakian, A. Abraham, V. Snasel, Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments, in: 2009 International Joint Conference on Computational Sciences and Optimization (CSO '09), vol. 1, IEEE Computer Society, Washington, DC, USA, 2009, pp. 8–12.

[19] R. Jeyarani, N. Nagaveni, R. Vasanth Ram, Design and implementation of adaptive power-aware virtual machine provisioner (APA-VMP) using swarm intelligence, Future Generation Computer Systems 28 (5) (2012) 811–821.

[20] K. Kousalya, P. Balasubramanie, To Improve Ant Algorithm's Grid Scheduling Using Local Search, International Journal of Intelligent Information Technology Application 2 (2) (2009) 71–79.

[21] Y. Liu, H. Zhu, A survey of the research on power management techniques for high-performance systems, Software: Practice and Experience 40 (11) (2010) 943–964.

[22] S. Lorpunmanee, M. Sap, A. Abdullah, C. Chompooinwai, An ant colony optimization for dynamic job scheduling in Grid environment, in: World Academy of Science, Engineering & Technology, 2007, pp. 314–321.

[23] S. Ludwig, A. Moallem, Swarm intelligence approaches for grid load balancing, Journal of Grid Computing 9 (3) (2011) 279–301.

[24] T. Ma, W. L. Qiaoqiao Yan, D. Guan, S. Lee, Grid task scheduling: Algorithm review, IETE Technical Review 28 (2) (2011) 158–167.

[25] M. Malawski, M. Kuzniar, P. Wojcik, M. Bubak, How to use Google app engine for free computing, IEEE Internet Computing, to appear.

[26] C. Mateos, A. Zunino, M. Hirsch, M. Fernández, Enhancing the BYG gridification tool with state-of-the-art Grid scheduling mechanisms and explicit tuning support, Advances in Engineering Software 43 (1) (2012) 27–43.

[27] C. Mateos, A. Zunino, M. Hirsch, M. Fernández, M. Campo, A software tool for semi-automatic gridification of resource-intensive java bytecodes and its application to ray tracing and sequence alignment, Advances in Engineering Software 42 (4) (2011) 172–186.

[28] P. Mathiyalagan, S. Suriya, S. Sivan, Modified ant colony algorithm for Grid scheduling, International Journal on Computer Science and Engineering 2 (2010) 132–139.

[29] D. Merkle, M. Middendorf, H. Schmeck, Ant colony optimization for resource-constrained project scheduling, Evolutionary Computation 6 (4) (2002) 333–346.

[30] E. Pacini, C. Mateos, C. García Garino, Planificadores basados en inteligencia colectiva para experimentos de simulación numérica en entornos distribuidos, in: Sexta Edición del Encuentro de Investigadores y Docentes de Ingeniería, 2011.

[31] E. Pacini, C. Mateos, C. García Garino, Schedulers based on Ant Colony Optimization for Parameter Sweep Experiments in Distributed Environments, IGI Global, 2012, to appear.

[32] E. Pacini, M. Ribero, C. Mateos, A. Mirasso, C. García Garino, Simulation on cloud computing infrastructures of parametric studies of nonlinear solids problems, in: F. V. Cipolla-Ficarra et al. (ed.), Advances in New Technologies, Interactive Interfaces and Communicability (ADNTIIC 2011), Lecture Notes in Computer Science, Springer, 2011, pp. 56–68, to appear.

[33] S. Pandey, L. Wu, S. Guru, R. Buyya, A particle swarm optimization-based heuristic for scheduling workflow applications in Cloud Computing environments, in: International Conference on Advanced Information Networking and Applications (AINA 2010), IEEE Computer Society, 2010, pp. 400–407.

[34] M. Pedemonte, S. Nesmachnow, H. Cancela, A survey on parallel ant colony optimization, Applied Soft Computing 11 (8) (2011) 5181–5197.

[35] J.-P. Ponthot, C. García Garino, A. Mirasso, Large strain viscoplastic constitutive model. theory and numerical scheme, Mecánica Computacional XXIV (2005) 441–454.

[36] G. Ritchie, J. Levine, A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments, in: Proceedings of the 23rd Workshop of the UK Planning and Scheduling Special Interest Group, 2004.

[37] J. M. Rodriguez, C. Mateos, A. Zunino, Are smartphones really useful for scientific computing?, in: F. V. Cipolla-Ficarra et al. (ed.), Advances in New Technologies, Interactive Interfaces and Communicability (ADNTIIC 2011), Lecture Notes in Computer Science, Springer, Huerta Grande, Córdoba, Argentina, 2011, pp. 35–44.

[38] J. M. Rodriguez, A. Zunino, M. Campo, Introducing mobile devices into Grid systems: A survey, International Journal of Web and Grid Services 7 (1) (2011) 1–40.

[39] C. Ruay-Shiung, C. Jih-Sheng, L. Po-Sheng, An ant algorithm for balanced job scheduling in grids, Future Generation Computer Systems 25 (2009) 20–27.

[40] M. Samples, J. Daida, M. Byom, M. Pizzimenti, Parameter sweeps for exploring GP parameters, in: Conference on Genetic and Evolutionary Computation (GECCO '05), ACM Press, New York, NY, USA, 2005, pp. 212–219.

[41] K. Sathish, A. R. M. Reddy, Enhanced ant algorithm based load balanced task scheduling in grid computing, IJCSNS International Journal of Computer Science and Network Security 8 (10) (2008) 219–223.

[42] C. Sun, B. Kim, G. Yi, H. Park, A model of problem solving environment for integrated bioinformatics solution on Grid by using Condor, in: Grid and Cooperative Computing - GCC 2004, Lecture Notes in Computer Science, Springer, 2004, pp. 935–938.

[43] W. Van Dorst, Bogomips mini-howto, `http://www.clifton.nl/bogomips.html` (2006).

[44] L. Wang, M. Kunze, J. Tao, G. von Laszewski, Towards building a Cloud for scientific applications, Advances in Engineering Software 42 (9) (2011) 714–722.

[45] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, W. Karl, Scientific cloud computing: Early definition and experience, in: 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008), IEEE Computer Society, Washington, DC, USA, 2008, pp. 825–830.

[46] J. Wozniak, A. Striegel, D. Salyers, J. Izaguirre, GIPSE: Streamlining the management of simulation on the Grid, in: 38th Annual Simulation Symposium (ANSS '05), IEEE Computer Society, 2005, pp. 130–137.

[47] F. Xhafa, A. Abraham, Computational models and heuristic methods for Grid scheduling problems, Future Generation Computer Systems 26 (4) (2010) 608–621.

[48] C. Youn, T. Kaiser, Management of a parameter sweep for scientific applications on cluster environments, Concurrency and Computation: Practice and Experience 22 (18) (2010) 2381–2400.

[49] Z. Zehua, Z. Xuejie, A load balancing mechanism based on ant colony and complex network theory in open Cloud Computing federation, in: 2nd International Conference on Industrial Mechatronics and Automation (ICIMA 2010), IEEE Computer Society, 2010, pp. 240–243.