Heuristic for resources allocation on utility computing infrastructures

João Nuno Silva, Luís Veiga, Paulo Ferreira
INESC-ID / Technical University of Lisbon, Portugal
{joao.n.silva, luis.veiga, paulo.ferreira}@inesc-id.pt

ABSTRACT

The use of utility on-demand computing infrastructures, such as Amazon's Elastic Clouds [1], is a viable solution to speed lengthy parallel computing problems to those without access to other cluster or grid infrastructures. With a suitable middleware, bag-of-tasks problems could be easily deployed over a pool of virtual computers created on such infrastructures.

In bag-of-tasks problems, as there is no communication between tasks, the number of concurrent tasks is allowed to vary over time. In a utility computing infrastructure, if too many virtual computers are created, the speedups are high but may not be cost effective; if too few computers are created, the cost is low but speedups fall below expectations. Without previous knowledge of the processing time of each task, it is difficult to determine how many machines should be created.

In this paper, we present an heuristic to optimize the number of machines that should be allocated to process tasks so that for a given budget the speedups are maximal. We have simulated the proposed heuristics against real and theoretical workloads and evaluated the ratios between number of allocated hosts, charged times, speedups and processing times. With the proposed heuristics, it is possible to obtain speedups in line with the number of allocated computers, while being charged approximately the same predefined budget.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems

General Terms

Resource allocation on utility computing

Keywords

Scheduling heuristics, Resource allocation, Bag-of-tasks

MGC'08, December 1-5, 2008 Leuven, Belgium

1. INTRODUCTION

Grid or cluster infrastructures are the best way to solve lengthy jobs, but require the user to have membership or institutional relationship with the organization owning the computing power. In the case of scientists with punctual computing needs, or even home users, the use of these systems in impractical.

Utility computing infrastructures such as Amazons Elastic Computing Clouds (E2C)[1], may be used by such users. Such infrastructures provide mechanisms only for users to create virtual computers. If these computers run a suitable middleware, jobs can be executed on them. These virtual computers are easily initiated and managed, allowing the creation of clusters of virtual computers running operating systems and software previously provided by the users. The creation of these machine pools is performed programmatically using a supplied API, while the allocation and management of the physical resources are hidden from the user. The user needs only to state which operating system image should be used.

By using virtual machines with the necessary operating system and software, a computational cluster can be easily created. Another benefit is the easy service subscription, where each user only needs to sign a simple contract and pay for the processing time used.

The on-demand launch of virtual machines can be used to provide computing cycles for the resolution of bag-of-tasks problems. After launching the virtual machines with the remote tasks' execution software, these machines can contribute by solving some of the tasks. In the case of bag-of-tasks problems, where no communication happens between tasks, the orchestration between the participant computers is straightforward.

The middleware launching virtual computers only has to predict the number of necessary computers and create them on the remote infrastructure. This value depends on the time necessary to complete the job and the minimum time unit charged. For the allocated budget, the user will want to obtain the best possible speedups. For instance, if the tasks are much shorter than the minimum time unit charged (usually one hour), and are allocated as many computers as there are tasks, the ratio between processing time and charged time will be low: only one small fraction of the time used the virtual computers while running was actually used to solve the problem. By allocating a large number of computers one gets the maximum possible speedup, but it may not be financial feasible.

The definition of the number of participant computers is

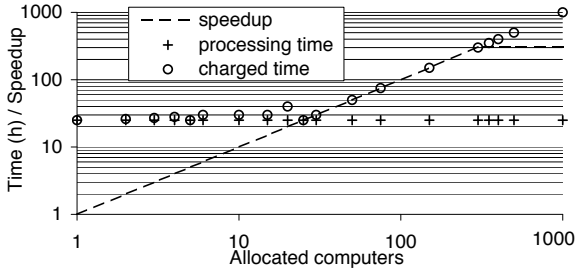

Figure 1: Evaluation of cost and speedups

problematic if the time each task takes to be completed is not precisely known and if processing time is charged in units much larger than the time each task takes to complete. In this paper we present an heuristic algorithm that dynamically allocates resources for computation, guaranteeing that the user pays close to an initially predefined amount, while maximizing the speedups.

If, in order to solve a job composed of 300 tasks (each one taking about 5 minutes to complete) 300 computers are allocated, then the speedup will be maximal but the charged time will also be high. If one hour is the minimum charge time unit, for a total processing time of 25 hours, the user will be charged 300 hours. In Fig. 1 we show for this example the charged time and speedups for different numbers of allocated remote computers.

In the example presented in Fig. 1 we can easily conclude that while paying the minimum possible (25 hours), the best speedup is accomplished with 25 computers; in this case every computer spent one hour solving tasks. Using fewer computers, the speedups are lower but without any actual decrease in the charged time. If more computers are allocated, the cost increases proportionally to the speedup.

Domestic users or even scientists with low resources should prefer to execute their jobs with substantial speedups within constrained budgets, instead of paying for the maximum speedup possible.

The definition of the necessary number of hosts is trivial if the time each tasks takes to complete is known before starting the job. If the processing time is known, with small runtime adjustments it is possible to obtain good speedups while paying the minimum amount possible. If the time to complete each task is not previously known, the definition of how many hosts should participate should be made during runtime. In this case after the completion of each task, its execution time should be used to calculate the necessary number of hosts.

A user estimation for task executing times may be used, but most of user estimations are incorrect [10] and the less knowledgeable the user is, the more prone to error his estimates are [8]. If the provided estimates are below the real time, a runtime adjustment should be made, if the estimation are high more hosts than the necessary are allocated.

Furthermore, the automation of task running time prediction removes this burden from the user. The user no longer needs to try to discover the average running time for the tasks.

In the next section we present other distributed computing infrastructures and why their requirements and scheduling strategies do not apply to our target environment. The following sections present the model of our resources and target applications and the proposed heuristic algorithm. Finally, we evaluate our algorithm against real traces and present some conclusions.

2. RELATED WORK

The scheduling of computational resources is a fundamental problem in order to optimize program execution and usage of necessary infrastructures. Scheduling algorithms and heuristics ensure that a given request is handled with a specified quality of service, and that underlying resources usage is optimized.

Typically, MPI [9] applications require a fixed predetermined number of host to cooperate in order to solve a problem, thus simplifying the scheduling algorithms. The processors allocated to each process can be used exclusively or shared with other requests. In *gang scheduling*, only one application is executing on each allocated computer, while in *co-scheduling* different applications execute simultaneously on a computational node. Some hybrid techniques such as presented by Bouteiller [3] try to conciliate the best of the these approaches.

The access to grid infrastructures usually requires the user to define the characteristics of the application to execute. These characteristics must state how many processors (or hosts) are necessary, their architecture, the operating system and the maximum duration of each task. In order to reduce the timespan of parallel applications, grid schedulers employ heuristics that try to take into account the expected task duration and the speed and availability of the selected hosts [4]. In the case of workflow applications, besides host selection heuristics, tasks are also ordered in order to reduce job's total timespan [7].

In the case of bag-of-tasks problems, the number of concurrent processes is not previously known and may vary. Current cycle-sharing systems, such as BOINC [2], use a greedy approach to allocate remote computers: all available computers are used to solve part of the problem. Some improvements have been made, such as in CCOF [14], in order to add some resource efficiency to remote host selection algorithms.

Due to their nature (independence and restartability), bag-of-tasks jobs can be scheduled to Grids and executed at idle hosts. For instance, Transparent Allocation Strategy [11] allows the allocation of processing power using requests parameterized with the number o processors (p), and duration (tr). Smaller p and tr allow a better fit of the requests, while larger tr accommodates a wider range of tasks. The cluster resource manager tries to satisfy the request, but when processors are necessary to other jobs, such tasks are killed. As a solution to the difficulty of determining p and tr, another strategy is proposed. This Explicit Allocation Strategy [13] presents an adaptive heuristic allowing, during runtime, the definition of both p and tr for each request, using information gathered by space-shared resource scheduler. This heuristic takes into account free slots available on the cluster and estimated task duration time to generate the first request. If the tasks included in such request are successfully executed, the execution time of the longest task will be used in subsequent requests; if the requested time is not enough, the estimated task runtime will be multiplied by an integer factor. Even though some estimation is performed w.r.t. task execution time, this solution tries neither to obtain average task processing times, nor to reduce the

unused idle time by the requests.

Existing utility computing infrastructures, (e.g. Amazon EC2 [1], Enomalism [6], or Eucalyptus [5]) provide means for the management of pools of computers, by deployment and execution of virtual machines. Such machines are created from disk images containing an operating system and necessary applications. Images are provided by the users, employing an API to launch and terminate the various instances of the machines.

In available utility computing infrastructures, resource allocation and scheduling problems are hidden at a lower level. When a user creates a virtual machine, the middleware is responsible for assigning a physical computer that can deliver the contracted quality of service. There is no need to know the total execution time for each virtual machine beforehand, as it is only used, after termination, to calculate the amount to charge. Furthermore, in commercial infrastructures, the charged time unit is large, usually one hour, which requires guarantees that machines are idle a minimum amount of time.

Taking these characteristics into account, current scheduling algorithms do not solve the problem of optimizing the number of hosts to allocate. Available solutions either require information about the execution time of all tasks, or employ solely a fixed number of computers, or take a greedy approach.

3. RESOURCE / APPLICATION MODEL

Recently, hardware vendors started offering solutions to create truly on-demand providers of utility computing resources. Such providers own pools of computers and offer computing power in the form of virtual machines through simple launching mechanisms. Users first register their virtual machines' images, with suitable operating systems and software, and then launch them. These new computing infrastructures also provide means to allow virtual machines within the same pool to discover each other and communicate among them. Moreover, the creation of machines is also possible from a running virtual machine.

Load distribution between available physical computers is not user's responsibility. The virtual machine management software handles the creation of virtual machines and schedules them to suitable computers in order to guarantee a minimum of quality of service (e.g., virtual machine processing speed).

Users need not reserve processing time slots. As with any other utility provider, accounting of chargeable time is performed after termination of the virtual machines using some predefined time unit (e.g., one hour per instance). The user will then pay for the time used.

We propose an heuristic we propose suitable for bag-of-tasks applications, where no communication occurs between tasks and where there is no need for all tasks to run simultaneously. Furthermore, tasks should be short lived, with execution times smaller than the charging time unit. The number of tasks should be high, on the order of hundreds, requiring several concurrent computers to process them all within an acceptable time frame.

An heuristic to determine the optimal number of hosts to solve a bag-of-tasks becomes required when task execution times are neither constant nor known in advance. This category of problems includes some scientific simulations, data analysis, or even the parallel rendering of large images,

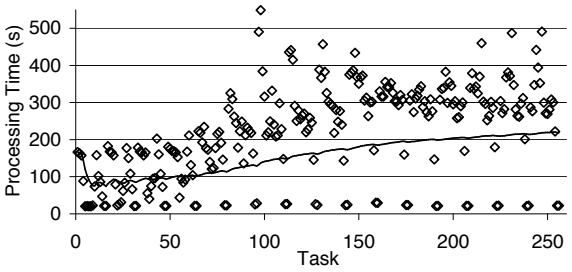

Figure 2: Image rendering tasks executing time

where each task renders a small view-port of the final image, or a signal frame int the rendering of movies.

As an example, in Fig. 2 we present the processing time of each of the 256 tasks necessary to concurrently render a 512x512 pixel image containing 2587 objects, using POV-Ray [12], measured on a 3.2GHz Pentium 4 PC.

From Fig. 2 we can conclude that it is not possible to predict how long will the next task take to execute. Periodically some tasks take about 25 seconds to complete, but this behavior depends on the image being rendered; other images or a different problem may exhibit a different pattern that a generic system is incapable to predict. Another unpredictable characteristic of jobs is the amplitude of the variation of task executing time. In the presented example this amplitude is about 9 minutes, while on other problems this variation may be on the order of just seconds.

In this example, the solid line represents the average duration of the tasks that were executed up to that moment. The average time increases slowly and only when executing the last tasks, does the average time reach its final value. The opposite can also happen if the first tasks are much slower that the average. This variations on temporal distribution of task completion times should be handled by an heuristic.

4. HEURISTIC

The proposed heuristic allows the definition of the number of machines to allocate on a pool of computers in order to ensure that the charged time fits in a predefined budget, while obtaining the maximum speedup possible with the allocated machines.

If a users wants to pay the minimum possible, each machine should execute and solve tasks during the whole of the minimum charging unit (one hour, for instance). If too many machines are created, each machine will have some idle time that will be charged anyway; if too few machines are allocated the job timespan will increase, with no extra savings

In order to define how many machines are needed, it is necessary to know how long a task will take to be executed. Only after knowing the average time to run a few tasks it is possible to calculate an estimate of the number of necessary computers, while taking into account the charging time-unit and how much the user is willing to pay.

Initially, one host is allocated and starts executing randomly selected tasks; whenever a task finishes, the time it took to be executed is used to calculate task average processing time. After allocating more hosts, the processing time of solved tasks is also used to calculate task average processing time. With every finished task, the calculated average

```
1     remainingTasks --
2     finishedTasks ++
3     totalProcessingTime += concludedTask.processingTime
4     tasksAverageTime = totalProcessingTime/finishedTasks
5
6     possibleTasks = 0
7     for each runningComputer:
8        possibleTasks += runningComputer.possibleTasks()
9     necessaryComputers = round((remainingTasks-possibleTasks)* tasksAverageTime/hostProcessingTime)
10
11    if (necessaryComputers>=0):
12       if (creationRatio == 1.0):
13          computersToCreate = necessaryComputers
14       else:
15          computersToCreate = int(necessaryComputers*creationRatio)+1
16       for i in range(computersToCreate):
17          allocateNewComputer()
18       creationRatio = creationRatio+(1-creationRatio)*increaseRatio
```

a)

```
1     for each runningComputer:
2        runningTasksProcessingTime += runningComputer.currentTask.processingTime
3     runningTasksAverageTime = runningTasksProcessingTime/runningComputers
4     for each finishedTask:
5        finishedTasksProcessingTime += finishedTask.processingTime
6     finishedTasksAverageTime = finishedTasksProcessingTime/finishedTasks
7     if runningTasksAverageTime > finishedTasksAverageTime:
8        tasksAverageTime =
9           (runningTasksAverageTime + finishedTasksAverageTime)/(finishedTasks + runningComputers)
```

b)

Figure 3: Heuristic pseudocode: a) executed when a task concludes, b) executed periodically (partial)

processing time is used to predict the number of necessary hosts to conclude the job with minimum time charging.

In Fig. 3.a) we present the pseudocode of the heuristic that calculates the necessary number of remote hosts, in order to increase speedups while guaranteeing that charged time on the utility computing infrastructure is close to the previously defined budget. This is executed whenever a task is concluded and the result returned.

Lines 1-4 calculate the average time that was necessary to process the finished tasks. This value will later be used to find the number of necessary hosts. In the following lines of code (lines 6-8), for each host it is predicted how many tasks it will be able to process until the end of the `hostProcessingTime` interval. Both the remaining time each host still has and task average time are used in this prediction.

If the user wants to pay the minimum amount possible, every host should be processing tasks for the most of the time, so `hostProcessingTime` should have the same value as the infrastructure charged unit (e.g. 60 minutes). For instance, if the user is willing to pay twice the minimum, every host can be idle half of the time, processing tasks during 30 minutes (if the charged time unit was 60 minutes).

The number of possibly processed tasks (`possibleTasks`) is then used to find how many tasks can not be executed by the current hosts. The average time previously calculated and the difference `remainingTasks-possibleTasks` are used to infer the number of additional necessary hosts, which will be corrected with the `creationRatio` factor, as will be described in the following paragraphs. The described process is applied whenever a task processing is finished and its result returned.

In the example in Fig. 2, if the tasks were executed in the presented order, our heuristic could never produce good results. The average time to process the tasks increases and only close to the last tasks the final average is obtained. In this case, near the end of the job, new hosts would be allocated, because the previously allocated hosts were not sufficient. The number of allocated hosts would be higher than the necessary and the job would take longer than required to conclude. To solve this problem it is necessary to guarantee that a close to final average value is obtained right from the first completed tasks. By randomly selecting the tasks, the average tasks processing time will converge more rapidly.

Even with a random task selection, if the processing times of the first tasks are higher than the final average processing time, too many hosts will be initially allocated. To solve this second problem the `creationRatio` factor is used. Initially this value is lower than 1.0, so that the number of initially allocated computers is lower than the calculated value. Later, as new tasks are solved and the average time converges to the final value, `creationRatio` may also converge to 1.0.

After receiving each processed task result, the value of `creationRatio` is updated, as shown in Fig. 3.a), line 18, `creationRatio` will increase and converge to 1. The influence of the initial `creationRatio` and `increaseRatio` will be evaluated in the next section.

If the first processed task takes too long to terminate, it takes a long time before new hosts are allocated to the computation. To solve this problem, periodically, the average processing time of the executing tasks is calculated, as shown on Fig. 3.b). If this value is higher than the average execution time of the previously finished tasks (lines 4-6), the `tasksAverageTime` is updated (lines 7-9), the number of necessary hosts is calculated and the necessary hosts created as shown on lines 6-9 and 11-17 in Fig. 3.a).

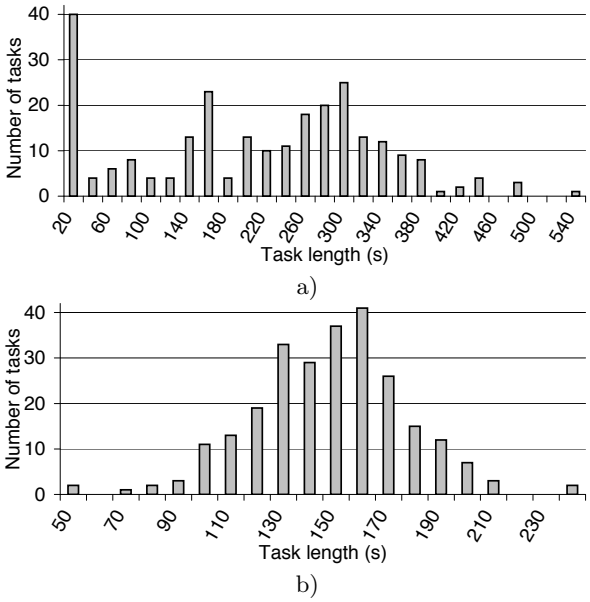

Figure 4: Tasks processing time distribution: a) image rendering, b) normal

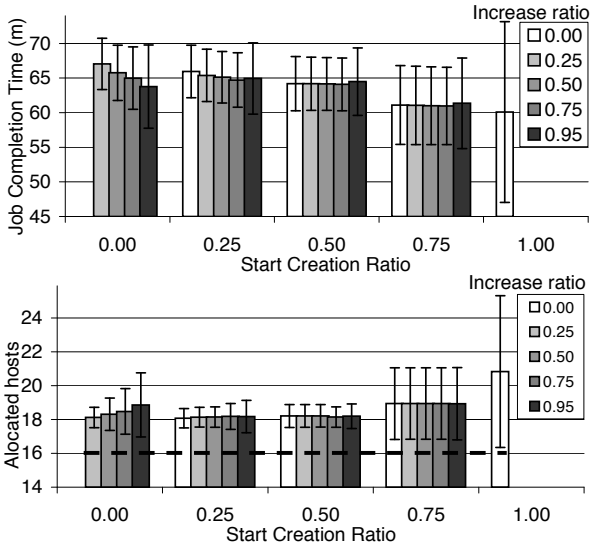

Figure 5: Parallel image rendering

All this code is executed locally on the computer responsible for the coordination of all virtual machines. As this machine has all the information regarding every virtual computer creation times and all running tasks, no communication is necessary.

In order to guarantee that the time each host is running is close to the minimum charging unit, to reduce wasted time, every host is terminated just before reaching the minimum charge time, usually one hour. In Fig. 3.a) line 8, the number of tasks each host is expected to execute can be calculated because it is well known for how long the virtual machine is supposed to execute.

5. EVALUATION

In order to evaluate the proposed heuristic we used two different jobs: i) traces of an image rendering, and ii) a set of tasks whose processing times have a normal distribution, as presented in Fig. 4.

The times presented on Fig. 4.a) are the measurement of the real partitioned image rendering, described in Section 3, Fig. 2. In this evaluation, for each experiment, we tested our heuristic against 200 different combinations of these values. In the case of the normal distribution (example on Fig. 4.b), each test was also performed against 200 different task execution time populations, each with a mean time of 150 seconds, a deviation of 30 seconds and 256 elements.

The tests were performed using a discrete-event simulator with the heuristic implemented in Python. Figures 5 and 6 present the measured timespan and the number of allocated hosts for different initial `creationRatio` and `increaseRatio` values.

In these graphics we can observe that our heuristic, with suitable selection of `creationRatio` and `increaseRatio`, delivers a good prediction on the number of hosts to allocate, at most our results differ by 12% from the optimal number of hosts to allocate (dashed line).

With a `creationRatio` of 1.0, where the deviation of the timespan and the number of allocated computers is high, it is highly probable that the number of allocated hosts exceeds the user's budget. This is due to the fact that during job execution, any variation on the calculated average time may increase the number of allocated hosts. A lower than 1.0 `creationRatio` is necessary in order to reduce the influence of these variations, which incorrectly increase the number of necessary hosts.

A too low initial `creationRatio` does not yield good results either. Only after too many completed tasks, enough hosts are allocated to speed the computation. With a initial `creationRatio` of 0.0 it is possible to observe the influence of the `increaseRatio`: a higher `increaseRatio` reduces the timespan, but increases the number of allocated hosts and the deviation of both the timespan and allocated hosts. With higher initial `creationRatio` these observations are not so evident due to the reduced number of allocated host.

In order to evaluate the precision of our heuristic, it is necessary to compare the number of allocated hosts with the obtained speedups. In Fig. 7 we present, for two different combinations of `creationRatio` and `increaseRation`, the number of allocated hosts and the speedup obtained for each execution. In the conservative behavior, both the `creationRatio` and `increaseRation` used where 0.5; in the aggressive behavior, these values where 0.75.

In both scenarios, even though the number of allocated hosts and the speedups do not coincide, they are close. For instance, executions that used 13 hosts had speedups between 11 and 12. This difference is due to the delay there is between the start of the first task and the final computation of the number of necessary hosts.

The main difference between both behaviors does not lie on the ratio between allocated hosts and speedups, but on the total number of allocated hosts. This difference is only noticeable in the image rendering example. With the conservative behavior, the interval of the number of allocated hosts is narrow, while with the aggressive behavior more hosts may be allocated. With an aggressive selection of `creationRatio` and `increaseRation`, it is possible that the number of

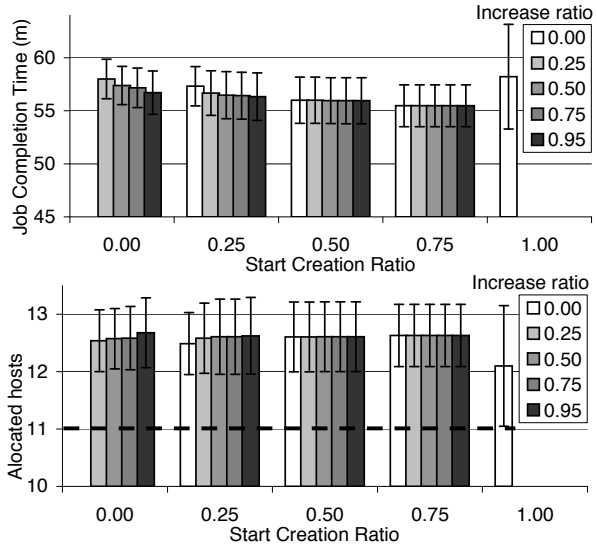

Figure 6: Parallel normal distribution job

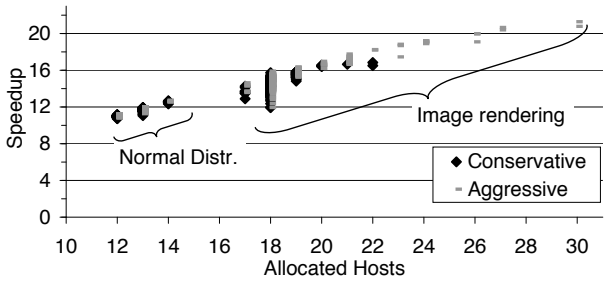

Figure 7: Speedup comparison

allocated host is close to twice the minimum number of hosts that would deliver a optimal cost per precessing time. The minimum number of allocated hosts and speedups are the same for both behaviors.

Varying the values of `creationRatio` and `increase Ratio` between 0.5 and 0.75, the user can obtain different allocated hosts (and charged values) variations, but always obtaining speedups close to optimal.

6. CONCLUSIONS

In this paper we presented a heuristic that efficiently defines the number of hosts to allocate on a utility computing infrastructure in order to solve bag-of-tasks problems. In the target environment, hosts may be allocated on demand, the user will later be charged for the time each host was used and jobs are composed of tasks whose execution times are not known before their execution.

The results show that our heuristic determines the number of necessary hosts to guarantee that the charged time is close to desired value. The number of allocated hosts are close to the optimal value that would be found if task duration were previously known. The speedups accomplished are close to the number of allocated hosts. The presented heuristic can provide both a conservative as well as a more aggressive behavior. Varying both the `creationRatio` and the `increaseRatio` it is possible to lower the charged time (with higher timespan) or lower the job timespan with an increase in payment.

If the user has a guess on the tasks processing time, this information can be used to initially launch several computers. The number of computers to launch should be corrected with the `creationRatio`, in order to avoid the allocation of too much machines.

7. REFERENCES

[1] Amazon.com, Inc. Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2.

[2] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2006.

[3] A. Bouteiller, H. L. Bouziane, T. Hérault, P. Lemarinier, and F. Cappello. Hybrid preemptive scheduling of mpi applications on the grids. *In Int. Journal of High Performance Computing Special issue*, 20:77–90, 2006.

[4] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 349–363, 2000.

[5] CS Department of Computer Science - University of California, Santa Barbara. Eucalyptus. http://eucalyptus.cs.ucsb.edu/.

[6] Enomaly Inc. Enomalism : Elastic computing platform - virtual server managemen. http://enomalism.com.

[7] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[8] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004*. Springer, 2005.

[9] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[10] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[11] M. Netto, R. Calheiros, R. Silva, C. De Rose, C. Northfleet, and W. Cirne. Transparent resource allocation to exploit idle cluster nodes in computational grids. *e-Science and Grid Computing, 2005. First International Conference on*, December 2005.

[12] Persistence of Vision Raytracer Pty. Ltd. Persistence of vision raytracer. http://www.povray.org/.

[13] C. A. F. D. Rose, T. Ferreto, R. N. Calheiros, W. Cirne, L. B. Costa, and D. Fireman. Allocation strategies for utilization of space-shared resources in bag of tasks grids. *Future Gener. Comput. Syst.*, 24(5):331–341, 2008.

[14] D. Zhou and V. Lo. Cluster computing on the fly: resource discovery in a cycle sharing peer-to-peer system. *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 66–73, April 2004.