Multi-Criteria Malleable Task Management for Hybrid-Cloud Platforms

Eddy Caron, Marcos Dias de Assunção

University of Lyon - LIP Laboratory
UMR CNRS - ENS de Lyon - Inria - UCB Lyon 5668
Email: {eddy.caron, marcos.dias.de.assuncao}@ens-lyon.fr

***Abstract*—The use of large distributed computing infrastructure is a means to address the ever increasing resource demands of scientific and commercial applications. The scale of current large-scale computing infrastructures and their heterogeneity make scheduling applications an increasingly complex task. Cloud computing minimises the heterogeneity by using virtualisation mechanisms, but poses new challenges to middleware developers, such as the management of virtualisation, elasticity and economic models. In this context, this work proposes algorithms for efficient scheduling and execution of malleable computing tasks with high granularity while taking into account multiple optimisation criteria such as resource cost and computation time. We focus on hybrid platforms that comprise both clusters and cloud providers. We define and formalise the main aspects of the problem, introduce the difference between local and global scheduling algorithms and evaluate their efficiency using discrete-event simulation.**

I. Introduction

Managing and providing compute resources to user applications is one of the main challenges tackled by the distributed computing community. To manage available resources, existing solutions rely on abstractions where users submit application requests to a management system or middleware responsible for task scheduling and resource allocation. Over the years, with varying levels of heterogeneity and success, several types of distributed infrastructure have been established to provide users with the resources they need to execute their applications. Cloud computing, by making use of resource virtualisation, has enabled the provision of more standard resource units (*i.e.* virtual machines) that are allocated atop underlying physical infrastructure.

From the point of view of allocating cloud resources to be managed by a middleware for running application tasks, the allocation can be either *static*, where a fixed number of resources is allocated; or *dynamic*, a scenario in which resources are allocated or released on demand to match the applications workload. Among the possible types of deployment, one can identify *centralised*, where the middleware is deployed in a single cloud; a *federation*, in which the middleware manages resources obtained from multiple clouds; and *hybrid*, a case where cloud resources are used together with clusters, data centres and workstations.

For reasons of cost, availability, control, and avoidance of provider lock-in, many organisations prefer to maintain and manage local infrastructure (*e.g.*, their own clusters and data centres) while using a cloud for certain workloads, and sometimes they exploit multiple cloud offerings simultaneously. Such hybrid deployment scenarios considered in the present work, are hereafter termed as *hybrid-cloud platforms*. Although the problem of scheduling requests onto available resources has been extensively investigated [1], [2], [3], the number of criteria taken into account when scheduling application tasks is generally limited. Moreover, cloud computing and the increasing availability of commercial offerings force middleware designers to deal with several additional issues related to managing virtualisation, exploiting elasticity, and multiple economical models.

In this work we investigate the problem of multi-criteria management and scheduling of malleable tasks [4]. These tasks provide a high level of granularity and parallelisation and have numerous practical applications in scientific and real-life computations [5], [6], [7]. In contrast to previous work that restricted the optimisation criteria for scheduling, the present work investigates algorithms to schedule and efficiently execute malleable computing tasks with high granularity while taking into account multiple optimisation criteria such as resource cost and computation time.

The rest of this paper is organised as follows. Section II presents background on hybrid platforms with multiple types of resources. Section III defines and formalises the main aspects of the considered problem, introduces the difference between local and global scheduling, and describes the main steps of the scheduling algorithms. Detailed explanation of local scheduling algorithms and realisations for the different types of platforms are found in Section IV. Section V presents a global scheduling algorithm and discusses means to optimise it. Description of simulation settings and results on performance evaluation of the algorithms are presented in Section VI. A discussion of existing work is given in Section VII, whereas the last section concludes the paper.

II. Platform Framework

This work focuses on scheduling of malleable tasks onto heterogeneous resources, considering several types of computing infrastructures for middleware deployment. We use the terms job and task interchangeably. Our goal is to leverage the heterogeneity of underlying available resources to devise efficient scheduling strategies. The tasks are submitted to a hybrid-cloud platform through a single interface provided by the global scheduler and are scheduled on various types of systems detailed next. Each system has its local scheduler that strives to respect tasks requirements. The acceptance of a job by a local scheduler can trigger rescheduling activities that can in turn result in migration of previously scheduled tasks to other systems. The rest of this section describes the three types of resource infrastructures considered, whereas the next section details the (re)scheduling activities that schedulers perform.

A. Dedicated Cluster

A dedicated cluster is entirely available to a user or client. By default, it is highly available and is viewed by a user as a computing resource with negligible cost. Its performance, however, is bounded and static, a reason why many organisations also use clusters along with other types of on-demand infrastructure.

B. Computing Centre

A computing centre is a fee-based resource, where the computation price P_i per hour of a job i can be estimated as:

$$P_i = t_i \times n_i \times k$$

where t_i is computation time, n_i the number of used processors, and k is the cost of using a processor over an hour. Here k accounts for the fixed and variable costs incurred by providers when offering and managing infrastructure, such as energy consumption, management personnel and so forth. This formula can in fact be used for all fee-based resources whose price is taken into account during scheduling.

C. Cloud Infrastructure

A cloud is a more recent model of distributed computing infrastructure under which a resource provider acquires a certain number of servers that are in turn rented out to users. Considering Amazon EC2, a widely used Infrastructure as a Service (IaaS) provider, virtual machine instances can be leased by customers under various models: reserved instances, on-demand instances and spot instances.

This work considers the use of spot instances as they are usually cheaper than their on-demand counterparts. Switching to on-demand instances is possible when Amazon EC2 cannot provide the required number of spot instances. The use of reserved instances may be more appealing when the cloud usage grows large and yearly reservation ends up justifying the pre-allocation of multiple reserved instances.

The choice of the platform types described above is motivated by the fact that they are frequently used by enterprises. Dedicated clusters correspond to computing resources owned by an enterprise. It should be used as much as possible to maximise return of investment. Cluster resources shared with or rented from partners are referred here as *allocated clusters*. Use of the cloud depends on the charge of the resources mentioned above as well as on cloud provider prices.

III. Problem and Scheduling Steps

The problem of scheduling malleable tasks onto hybrid clouds is split into two levels, namely:

- **Local scheduling:** which handles tasks at a locally controlled resource.
- **Global scheduling:** responsible for selecting a resource to which a new task is assigned. Algorithms at this level schedule tasks considering the overall distributed infrastructure.

The optimisation criteria provided for task scheduling can be conflicting. The introduced algorithms consider three criteria: deadline, budget and task priority. The goal is to assign a task to a computing resource so that the price paid for its use and the task's makespan respect, respectively, the task's budget and deadline constraints. A local scheduler sorts tasks by decreasing order of priority when selecting a task to execute.

The task parameters are explained as follows. The input of scheduling consists of a tuple (T, R), where T comprises computing tasks and information about their computational complexity, priority, deadline, maximum budget for execution, or other parameters that a user might find relevant. An individual task is represented as T_{task_id}:

$$T_{task\_id} = (task\_id, size, priority, deadline, budget)$$

R, on the other hand, describes a set of hosts with parameters such as type, performance (MFlops/CPU), usage cost (*e.g.* cost per CPU hour as P_i previously introduced in Section II), and number of CPUs:

$$R_{rsc\_id} = (rsc\_id, type, performance, cost, nbCPU)$$

While T_{task_id} is information provided by a user, R_{rsc_id} describes a computing resource managed by the middleware. The scheduling objective is to provide mappings of (T, R) such that a candidate resource R_{rsc_id} is as closer a match to a task T_{task_id} requirements as possible.

The scheduling **Step 1** defines two parameter sets for a given tuple $(T_{task\_id}, R_{rsc\_id})$ namely optimising cost and computation time (*i.e.* makespan):

$$\{P_{cost}, P_{perf}\} = create\_schedules(T_{task\_id}, R_{rsc\_id})$$

The building of such schedules depends on the type of resources considered for task execution – *e.g.* for a cloud we can vary the number and type of machines. Although task execution on a low-power machine will be cheaper, it may take longer than allocating several powerful instances. For clusters the balance of makespan and price depends on the reservation strategy in place. The speed-up of using multiple processor cores is rarely linear, so usually the more processors we use the smaller is the performance/price ratio. The number of reserved time slots also correlates with the ratio because of deployment and start-time overhead; *i.e.*, an application needs time to start running. Although this execution time is usually paid for, it is not actually used for task execution. Specific optimisation schemes to handle this step are given in Section IV which describes the resource-specific local algorithms.

A submission strategy $P_{strategy}$ defines the actual execution cost, the list of processors used for computation, the intervals over which processors will be allocated as specified by start (st) and end time (et), and the list of tasks to be executed with these parameters. More formally, a $P_{strategy}$ can be viewed as:

$$P_{strategy} = (cost, \{(cpu\_id_1, st_1, et_1), (cpu\_id_2, st_2, et_2), \ldots, (cpu\_id_n, st_n, et_n)\}, \{tasks\_to\_migrate\})$$

where $tasks\_to\_migrate$ refers to tasks that will be rescheduled if the new task is accepted. A *candidate* – a tuple composed of task, resource, submission parameters and the candidate type – hence refers to a computing resource that can suit a given job execution scenario; *i.e.*, the expected job makespan, execution cost, and list of tasks to be rescheduled. Let us define a *good candidate* as a resource respecting both the deadline and budget of a task, and a *bad candidate* as one that does not respect at least one of the two constraints.

For each possible set of submission parameters, there exists a type of candidate (good or bad) that is initially set to *null*:

$$\forall\ strategy\ \forall rsc\_id\ \forall task\_id\ \exists\ C_{strategy,rsc\_id,task\_id} \mid C_{strategy,rsc\_id,task\_id} = (T_{task\_id}, R_{rsc\_id}, P_{strategy}, null)$$

At scheduling **Step 2**, a candidate status is represented by boolean whose value is true if the candidate parameters respect all the task requirements and false otherwise. A job makespan, when choosing a given candidate, is represented by:

$$P.makespan = \max_i P.cpu\_usage_i.et$$

Moreover, a candidate is considered *good* when the following inequality is respected:

$$IsGood(C_i) = C_i.T.deadline \geq C_i.P.makespan\ \&\ C_i.T.budget \geq C_i.P.cost\ \&\ C_i.T.size \geq \sum_j C_i.P.tasks\_to\_migrate_j$$

Next, the *null* value used on the previous step is replaced by the actual type of the candidate, *true* for a good candidate or *false* for a bad one. Hence, we redefine the types of all candidates in the previously described list as it is shown in the equation:

$$C_{strategy,rsc\_id,task\_id} = (T_{task\_id}, R_{rsc\_id}, P_{strategy}, IsGood(C_{strategy,rsc\_id,task\_id}))$$

Step 3 splits the candidate set into two subsets namely containing good and bad candidates as represented by:

$$C_{good} = \{C_i \mid IsGood(C_i) = true\}$$
$$C_{bad} = \{C_i \mid IsGood(C_i) = false\}$$

After that, **Step 4** picks a candidate among the set of all the potential candidates $\{C_i\}$ to which a task is then submitted. For simplicity, we use a Heaviside step function to take into account the budget or deadline overhead:

$$H(x) = \begin{cases} 0 & \text{if} \quad x \leq 0 \\ 1 & \text{if} \quad x > 0. \end{cases}$$

Using this function we define the operator $(\cdot)_+$ as:

$$(x)_+ = H(x) \cdot x$$

If there exist *good* candidates, we pick the cheapest one. Otherwise, two simple strategies are possible for selecting a candidate C_{fetch}:

1) Pick the least expensive candidate among those with smallest delay, where:

$$\{C_{fetch}\} = \left\{ C \in \{C_i\} \mid (C.P.makespan - C.T.deadline)_+ = \min_{(C_i.P.makespan - C_i.T.deadline)_+} C_i \right\}$$

such that:

$$C_{opt} = \underset{C_{fetch}.cost}{\text{argmin}}\ C_{fetch}$$

2) Select the fastest candidate among those with smallest spent budget, where:

$$\{C_{fetch}\} = \left\{ C \in \{C_i\} \mid (C.P.budget - C.T.cost)_+ = \min_{(C_i.P.budget - C_i.T.cost)_+} C_i \right\}$$

such that:

$$C_{opt} = \underset{C_{fetch}.deadline}{\text{argmin}}\ C_{fetch}$$

IV. Local Scheduling Algorithms

As each type of platform has its own characteristics, we provide a specific local scheduling algorithm for each one. The proposed algorithms, however, provide the same interface and expect the same parameters in order to take a decision on task scheduling. The common task parameters required by all algorithms in order to determine the scheduling of new task are namely: size, priority, deadline and budget.

A local scheduling algorithm can offer more than one possible candidate; for example, it gives the options of either respecting deadline or budget constraints, when both cannot be simultaneously respected. Moreover, the algorithm may not return any candidates at all, such as when the sum of the sizes of tasks to be rescheduled exceeds the new task's size. Therefore, all described algorithms provide as output a list of candidates even if it contains only one element.

The algorithms described next compute the candidate parameters. When a task submission candidate is placed in the queue, the preempted jobs should be rescheduled by the submitter of the new job.

A. Dedicated Cluster

Algorithm 1 presents the placement of tasks on a dedicated cluster according to their priority while striving to respect users requirements. The higher a task's priority, the earlier it is executed. Once a task becomes the first in the queue, it is executed using all available processors, as we target applications with high parallelisation levels.

As described earlier, when a new task is admitted by the scheduler the deadlines of previously scheduled tasks may no longer be respected. The possible decisions under such a case are the following:

- Reschedule tasks whose deadlines are no longer respected;
- Do not schedule the new job on the current host.

As illustrated by Algorithm 1, when a new job is received, the scheduler computes two lists namely with jobs whose priorities are lower than that of the received job (line 2), and one with jobs whose priorities are higher than the incoming job's (line 3). Then the algorithm checks whether the new job's deadline is respected (lines 5 to 10). After that, it determines the jobs whose deadlines can still be respected if scheduled on the same host, and then builds a list with jobs whose deadlines can no longer be satisfied (lines 12 to 18). Jobs whose deadlines cannot be respected become candidates to be rescheduled (see the $host$ data structure in lines 20 to 24). As the position of a job in a queue is determined by its priority, it is not worth rescheduling a job on the same resource because it will take the same place in the queue. The rescheduling of a job ends if it reaches a resource that does not support rescheduling (*e.g.* Cloud) or after postponing another job's deadline under lack of resources. To prevent excessive rescheduling, the total size of rescheduled jobs should not exceed the size of the new job being scheduled.

B. Allocated Cluster

Algorithm 2, which illustrates the scheduling for an allocated cluster, is similar to scheduling for a dedicated cluster, except for the fact that the local scheduler does not control all the underlying cluster resources.

In order to execute jobs, a reservation must be made through the local scheduler. A reservation consists in a set of processors which the scheduler can dispose of during a fixed time period. As we focus on malleable jobs, one job can be executed during several reservations even if the number of reserved processors is not the same across reservations.

To make reservations correctly, we first simulate the placement of a new task and calculate the amount of

Algorithm 1: Dedicated Cluster Algorithm (DCA)

```
schedule_received_job(job)
    scheduled_after={ j $\in$ jobs_queue : j.priority < job.priority }
    scheduled_before=jobs_queue $\setminus$ scheduled_after
    host.id = self.id
    job_finished=job.length + $\sum_{j \in scheduled\_before}$ j.length
    host.end_time = job_finished
    if host.end_time $\leq$ job.deadline then
        host.good_candidate = true
    else
        host.good_candidate = false
    end
    host.rescheduled_jobs=[ ]
    host.cost=0
    for j $\in$ scheduled_after do
        if job_finished + j.length < j.deadline then
            job_finished = job_finished + j.length
        else
            host.rescheduled.append(j)
        end
    end
    if $\sum_{j \in host.rescheduled}$ j.length < job.length then
        // list of one host
        parent.send([host])
    else
        // avoid using this host
        parent.send([ ]) // empty list
    end
```

processor time to reserve in order to fulfil the resource requirements of all jobs (lines 1 to 12 of the algorithm). Then, as done for the dedicated cluster, we sort jobs by decreasing order of priority and check whether deadlines of all tasks are respected. After that, the algorithm copies one for dedicated cluster only adding the budget sufficiency step.

C. Cloud Scheduler

For clouds, more detailed information is required to determine the type of allocated machines. The rationale of Algorithm 3 is to verify whether the cloud is able to execute a received job respecting its budget and deadline. If that is the case, the algorithm returns a *good candidate*. Otherwise, it returns a list of two candidates; the first minimising the budget overspend and calculating the execution time needed with this budget, and the second calculating the amount of money needed to finish the job in time.

However, clouds have their intricacies that demand a technique to optimise expenses slightly. Certain clouds (*e.g.* Amazon EC2) charge per-hour fees for resource usage. Hence, the same money is spent for 30 minutes and 59 minutes of use of a given computing resource. To exploit this particularity one should only shut down free machines at the end of their payment hour. All the jobs scheduled on the cloud should use as much resources as possible from the already running machines. In Algorithm 3, the time left for a machine is denoted by the variable $free\_time\_left$.

Algorithm 2: Allocated Cluster Algorithm (ACA)

```
schedule_received_job(job)
    // Copy scheduling table to simulate job
       placement
    simsc = schedule_table.copy()
    simsc.reserve(job.size)
    scheduled_after={ j $\in$ jobs_queue : j.priority < job.priority }
    scheduled_before=jobs_queue $\setminus$ scheduled_after
    host.id = self.id
    // Simulate scheduling of jobs with high
       priority, not affected by the new job
    for j $\in$ scheduled_before do
        simsc.schedule(j)
    end
    sched_params = simsc.schedule(job)
    criterion1 = (sched_params.end_time $\leq$ job.deadline)
    criterion2 = (sched_params.cost $\leq$ job.budget)
    if criterion1 and criterion2 then
        host.good_candidate = true
    else
        host.good_candidate = false
    end
    host.end_time = sched_params.end_time
    host.cost = sched_params.cost
    // Calculate list of rescheduled jobs
    host.rescheduled_jobs=[ ]
    for j $\in$ scheduled_after do
        if simsc.schedule(j).end_time > j.deadline then
            simsc.remove(j)
            host.rescheduled_jobs.append(j)
        end
    end
    if $\sum_{j \in host.rescheduled}$ j.length < job.length then
        parent.send([host]) // list of one host
    else
        // avoid use of this host
        parent.send([ ]) // empty list
    end
```

Algorithm 3: Cloud Cluster Algorithm (CCA)

```
schedule_received_job(job)
    host.id = self.id
    number_of_vm = $\lceil$ job.size/(vm_productivity $\times$ job.deadline) $\rceil$
    price = number_of_vm $\times$($\lceil$ job.deadline $\rceil\times$ cost_vm_hour -
    $free\_time\_left$)
    host.cost = price
    host.end_time = job.deadline
    host.reschedule = [ ]
    if host.cost $\leq$ job.budget then
        host.good_candidate = true
        parent.send([ host ])
    else
        host.good_candidate = false
        // One more option for scheduler
           minimizing budget overspend
        host2.id = self.id
        host2.end_time = cloud.get_time(job.budget)
        host2.cost = job.budget
        host2.reschedule = [ ]
        host2.good_candidate = false
        parent.send([host, host2])
    end
```

V. Global scheduling model

The goal of the global scheduling algorithm is to optimise the overall money spent while respecting task deadlines and budgets.

A basic global scheduler would broadcast job re-

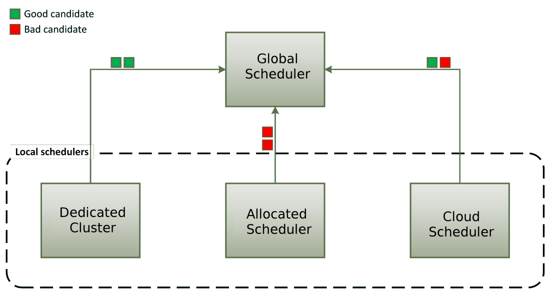

Fig. 1: Example of distributed heterogeneous platform structure.

quirements to all computing resources and choose the best candidates. If *good candidates* exist, we pick one with the lowest price to which a job is submitted. This approach is efficient when the number of computing resources is small, but as the number grows the global scheduler may become a bottleneck. Hence, we propose a hierarchical approach for the global scheduler. The principle of this approach is to pass down the hierarchy details of a task execution, aggregate answers from local schedulers using a multi-level system and choose a convenient host. The approach considers a multi-level system comprising scheduling agents structured as a tree with computing hosts as leaves.

The rationale of Algorithm 4, used for global scheduling, is simple. Each scheduling agent requests from its children a list of candidates. If there exists a good candidate, the agent forwards it one level up in the hierarchy. Otherwise it returns a candidate with the best price with the least deadline violation and another and another with the earliest makespan among those that least exceed the budget. The top level node of the hierarchy then returns one good, or two bad, candidates to the caller which chooses one onto which the job is scheduled.

One can see that the simple approach is a particular case of hierarchical algorithm with just one root with all computation nodes connected directly to it. The hierarchical algorithm is distributed and its complexity can be expressed as $O(mn)$, where m is maximum leaf depth, n is maximum number of children per scheduling agent.

VI. Performance Evaluation

A Python simulator was developed in order to validate and evaluate the efficiency of the proposed algorithms. We resorted to discrete-event simulation as it enables controlled and repeatable experiments, and because the use of real resources such as those allocated from a cloud, could make the cost of evaluation prohibitive. The choice was hence motivated by the need to test all the types of previously described resources,

Algorithm 4: Algorithm used by scheduling agent.

```
agent_on_job_receive (job)
    children.broadcast(job)
    hosts = children.gather()
    good_candidates =
    $\{h : h \in hosts, h.good\_candidate = true\}$
    if $good\_candidates.length \neq 0$ then
        best_candidate= $\underset{h \in good\_candidates}{\text{argmin}}$ (h.cost)
        parent.send(best_candidate)
    else
        // No good candidates
        // Minimise budget overspend
        min1 = $\min_{h \in hosts} (h.cost - j.budget)_+$
        list1 =
        $\{h : h \in hosts, (h.cost - j.budget)_+ = min1\}$
        candidate1 = $\underset{h \in list1}{\text{argmin}}$ h.end_time
        // Minimize deadline overpass
        min2 = $\min_{h \in hosts} (h.end\_time - j.deadline)_+$
        list2 = $\{h : h \in$
        $hosts, (h.end\_time - j.deadline)_+ = min2\}$
        candidate2 = $\underset{h \in list2}{\text{argmin}}$ (h.cost)
        parent.send([candidate1, candidate2])
    end
```

namely a dedicated cluster with negligible computing price, a paid allocated cluster, and cloud resources.

For the clusters the granularity at which resources can be allocated is one minute, and the performance of all processors is homogeneous. For the paid clusters, the cost per processor minute is taken as a cost unit. The price of an hour on an 8 processor machine on the cloud is equal to 960 units (2 units/minute per processor).

The jobs were generated randomly. The time between job arrivals was exponentially distributed whose random variable has a mean of 1 minute. The priority is an integer uniformly chosen from 1 to 20. The deadline for all tasks is uniformly distributed between minimal possible execution time (using maximal number of processors) and time of execution on a single processor. To evaluate the behaviour of the algorithms, three sets of jobs were generated: small jobs (60-120 processor minutes), big jobs (1000-2000 processor minutes) and mixed sets with equal number of tasks of both sizes.

We considered three evaluation metrics:

- Total job execution time, which is sum of the time between job arrival and end of execution for all jobs.
- Total execution cost.
- Delay of jobs against their initial deadlines.

A round-robin algorithm which submits a job to a uniformly chosen resource was taken as baseline and tested against the same job sets. Algorithms were run on sets of different sizes, gradually increasing the number of jobs. Each algorithm was run twice for all the job sets considering namely: the first run where in the absence of “good” candidates the algorithm chooses the least expensive “bad” candidate (money saving strategy, policy A); and the second run in which the fastest candidate is picked (time saving strategy, policy B). The rest of this section presents the simulation plots for all the described variations of parameters.

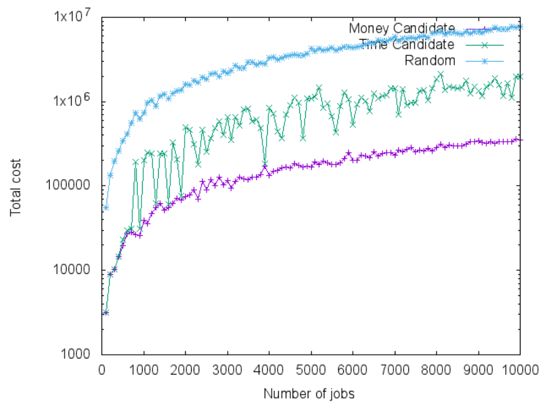

Fig. 2: Total cost for mixed jobs.

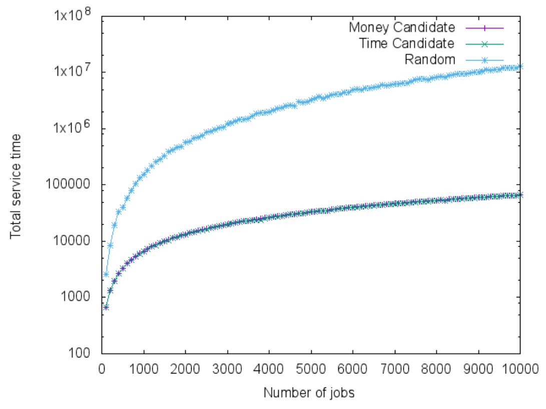

Fig. 3: Total execution time of mixed jobs.

The graphs in figures 2 to 4 demonstrate that in general the proposed algorithms outperform the round-robin approach except under small tasks. However, even in this case the proposed algorithms give much better savings of budget (Figure 6) and significantly lower delay (Figure 7). In the case of big jobs, our algorithm shows better performance considering all metrics.

The difference between the policies of budget overspend and delay minimisation is also shown. It corresponds to the nature of policies minimising the amount of money spent on computations and delay respectively.

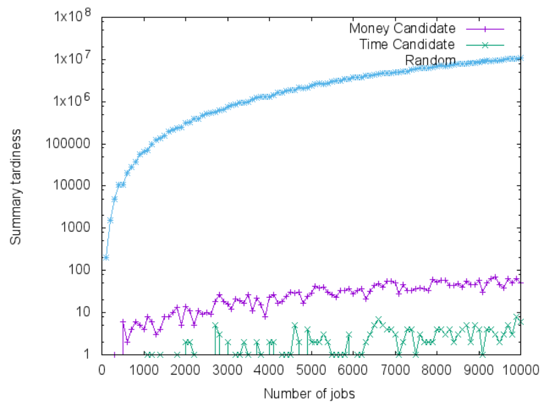

Fig. 4: Total delay of mixed jobs.

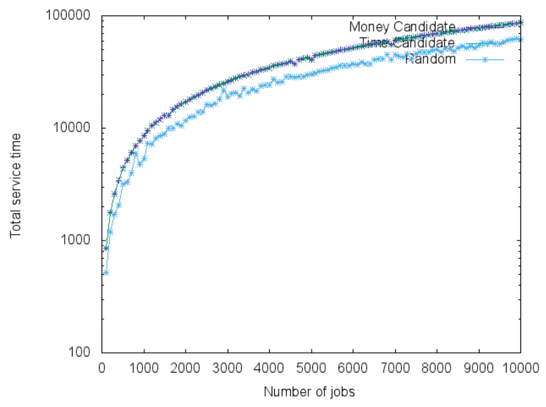

Fig. 5: Total execution time of small jobs.

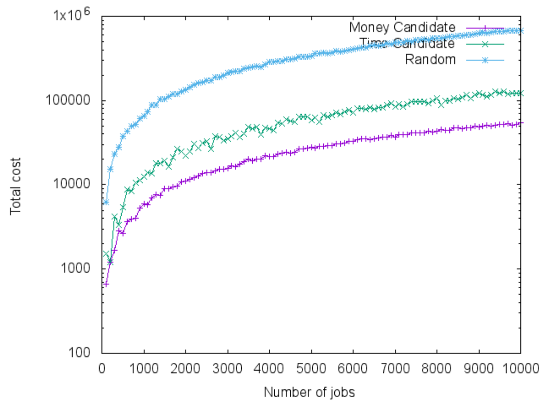

Fig. 6: Total cost of small jobs.

VII. Related Work

There exists a vast literature on task scheduling on distributed systems, part of which considers scheduling onto heterogeneous resources [4], [8]. The use of clouds

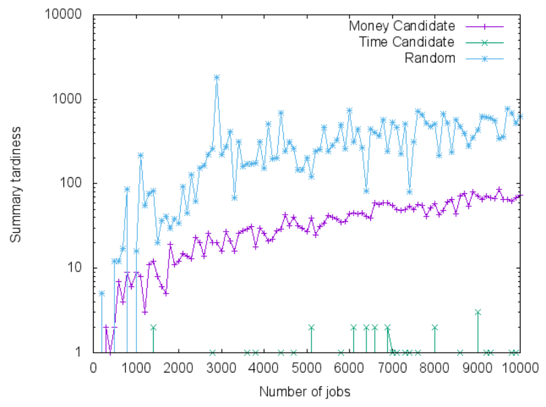

Fig. 7: Total delay of small jobs.

to augment the capabilities of local infrastructure has also been considered in previous work [9].

The case of multi-criteria optimisation has also been considered [10], even though the proposed algorithms are often *off-line*, and require prior information about tasks in order to execute. *On-line* multi-criteria scheduling algorithms with no a-priori information about tasks has been considered in previous work, where computing resources are represented by clusters composed of ordinary computers of heterogeneous performance [11], and where the choice of platforms is much richer and includes Internet desktop Grids, best effort Grids and Clouds [12]. Stating the same problem, this previous work offers absolutely different heuristic solutions tailored to desktop Grid environments and hence less efficient when considering hybrid clouds.

The algorithms proposed in our paper have advantages over the precited existing work, as it uses rescheduling mechanisms for moving tasks across resources. In order to maximise efficiency we focused on malleable tasks.

VIII. Conclusions

This paper presented algorithms that optimise resource consumption. Though the work considered a large list of target platforms, it can further be extended. The algorithms can also be customised to suit the needs of a user. Additional criteria (*e.g.* energy efficiency) could be included with minor modifications of local algorithms and the function $IsGood$.

The efficiency of algorithms was demonstrated by simulation and is being evaluated with real computational tasks. The results may slightly differ because of different distributions of the task sizes and of periods between task arrivals.

IX. Acknowledgement

We thank Semen Marchuk for his help in this work especially for his work on the simulator.

References

[1] Pulin Agrawal and Smitha Rao. Energy-aware scheduling of distributed systems. *Automation Science and Engineering, IEEE Transactions on*, 11(4):1163–1175, 2014.

[2] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[3] Krithi Ramamritham and John A Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE software*, (3):65–75, 1984.

[4] Denis Trystram. Scheduling parallel applications using malleable tasks on clusters. In *International Parallel and Distributed Processing Symposium (IPDPS 2001)*, pages 199–, Washington, USA, 2001.

[5] Ricky A Kendall, Edoardo Aprà, David E Bernholdt, Eric J Bylaska, Michel Dupuis, George I Fann, Robert J Harrison, Jialin Ju, Jeffrey A Nichols, Jarek Nieplocha, et al. High performance computational chemistry: An overview of nwchem a distributed parallel application. *Computer Physics Communications*, 128(1):260–283, 2000.

[6] S. Rampersaud, L. Mashayekhy, and D. Grosu. Computing nash equilibria in bimatrix games: Gpu-based parallel support enumeration. *Parallel and Distributed Systems, IEEE Transactions on*, 25(12):3111–3123, Dec 2014.

[7] Stavros A Zenios. High-performance computing in finance: The last 10 years and the next. *Parallel Computing*, 25(13):2149–2175, 1999.

[8] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L.V. Kale. A batch system with efficient adaptive scheduling for malleable and evolving applications. In *International Parallel and Distributed Processing Symposium (IPDPS 2015)*, pages 429–438, May 2015.

[9] Marcos Dias de Assunção, Alexandre di Costanzo, and Rajkumar Buyya. Evaluating the cost-benefit of using Cloud computing to extend the capacity of clusters. In *18th ACM International Symposium on High performance Distributed Computing (HPDC 2009)*, pages 141–150, New York, USA, 2009. ACM.

[10] Veronika Rehn-Sonigo. *Multi-criteria mapping and scheduling of workflow applications onto heterogeneous platforms.* PhD thesis, University of Passau, 2009. http://d-nb.info/100937074X.

[11] Ranieri Baraglia, Patrizio Dazzi, Gabriele Capannini, and Giancarlo Pagano. A multi-criteria job scheduling framework for large computing farms. In *Proc. of the 10th IEEE International Conference on Computer and Information Technology*, pages 187–194, 2010.

[12] Mircea Moca, Cristian Litan, Gheorghe Cosmin Silaghi, and Gilles Fedak. Multi-criteria and satisfaction oriented scheduling for hybrid distributed computing infrastructures. *Future Generation Computer Systems*, (0):–, 2015.