Parallel Processing Letters

BUDGET ESTIMATION AND CONTROL FOR BAG-OF-TASKS SCHEDULING IN CLOUDS

ANA-MARIA OPRESCU

Department of Computer Science, Vrije Universiteit, De Boelelaan 1081a
1081 HV Amsterdam, The Netherlands

and

THILO KIELMANN

Department of Computer Science, Vrije Universiteit, De Boelelaan 1081a
1081 HV Amsterdam, The Netherlands

and

HARALAMBIE LEAHU

Department of Mathematics and Computer Science, Technical University Eindhoven (TU/e),
Den Dolech 2
5600 MB Eindhoven, The Netherlands

Received January 2011
Revised March 2011
Communicated by Guest Editors

ABSTRACT

Commercial cloud offerings, such as Amazon's EC2, let users allocate compute resources on demand, charging based on reserved time intervals. While this gives great flexibility to elastic applications, users lack guidance for choosing between multiple offerings, in order to complete their computations within given budget constraints. In this work, we present *BaTS*, our budget-constrained scheduler. Using a small task sample, BaTS can estimate costs and makespan for a given bag on different cloud offerings. It provides the user with a choice of options before execution and then schedules the bag according to the user's preferences. BaTS requires no a-priori information about task completion times. We evaluate BaTS by emulating different cloud environments on the DAS-3 multi-cluster system. Our results show that BaTS correctly estimates budget and makespan for the scenarios investigated; the user-selected schedule is then executed within the given budget limitations.

Keywords: runtime estimation, sampling, linear regression, cloud computing

1. Introduction

In *computational science*, *parameter sweep* or *bag of tasks* applications are as dominant as computationally demanding. The classic Condor [1] system is in widespread

use to deploy as many application tasks as possible on otherwise under utilized computers, coining the term of *High Throughput Computing*, utilizing networks of idle workstations, cluster computers, and computational grids.

Common to such computing platforms is the model of sharing on a best-effort basis, without any performance guarantees, and commonly also free of charge. This cost-free, best-effort model has had a strong influence on the way bag-of-tasks applications have been deployed. Scientists simply grab as many machines as possible, trying to improve their computational throughput, while neglecting how quickly certain machines can perform the given tasks.

When Amazon announced EC2, its *Elastic Computing Cloud* [2], the era of cloud computing started to offer a different computing paradigm. EC2 and other commercial cloud offerings provide compute resources with defined quality of service (CPU type and clock speed, size of main memory, etc.) These computers can be allocated, and are charged, for given time intervals, typically per hour.

The various commercial offerings differ not only in price, but also in the types of machines that can be allocated. Within EC2 alone, there are several types of machines. While all machine offerings are described in terms of CPU clock frequency and memory size, it is not clear at all which machine type would execute a given user application faster than others, let alone predicting which machine type would provide the best price-performance ratio. The problem of allocating the right number of machines, of the right type, for the right time frame, strongly depends on the application program, and is left to the user. As a consequence, users lack a sense of the range of budgets needed for a given bag's execution.

In this work, we extend *BaTS*, our budget-constrained scheduler [3]. BaTS requires no a-priori information about task completion times, instead BaTS learns them at runtime. The contribution of this paper provides the user with a range of reasonable budget and runtime estimations before the bulk of the execution starts. We extend BaTS by a mechanism to estimate makespans and costs of computing a bag of tasks on different combinations of cloud machines using a single and small initial sample of tasks.

As the result of the initial sampling phase, the user is presented with several choices, either preferring lower cost or faster execution. Based on the user's choice, BaTS schedules such that the bag of tasks will be executed within the given budget (if possible), while minimizing the completion time.

For evaluation purposes, we have emulated different types of clouds on the DAS-3 multi-cluster system. Here, our evaluation shows that BaTS finds different schedules and is able to comply to its own predicted makespans and budget limits.

This paper is structured as follows. In Section 2, we describe the estimation mechanism for bag runtimes and costs on different cloud combinations. In Section 3, we present the BaTS scheduling algorithm and how it enforces the estimated makespan and budget limitations. In Section 4, we evaluate BaTS' performance and compliance to its own estimations. Section 5 discusses related approaches before we draw conclusions and outline directions of ongoing work in Section 6.

2. Budget Estimation for Bags of Tasks

BaTS is scheduling large bags of tasks onto multiple cloud platforms. The individual tasks are scheduled in a self-scheduling manner onto the allocated machines. The core functionality (execution phase) is to allocate a number of machines from different clouds, and to adapt the allocation regularly by acquiring and/or releasing machines in order to minimize the overall makespan while respecting the given budget limitation. We enrich the core functionality with an initial sampling phase that computes a list of budget estimates providing the user with flexible control of the execution phase.

We assume that the tasks of a bag are independent of each other, so they are ready to be scheduled immediately. We also assume that the tasks can be preempted and rescheduled later, if needed by a reconfiguration of the cloud environment. Our task model incurs no prior knowledge about the task execution times. We assume that there is some completion time distribution among the tasks of a bag, but, a-priori, it is unknown to both the user and to the BaTS scheduling algorithm. The only information we require is the size of the bag (the total number of tasks that need to be executed).

About the machines, we assume that they belong to certain categories (like EC2's "Standard Large" or "High-Memory Double Extra Large") and that all machines within a category are homogeneous. The only information BaTS uses about the machines is their price, like "$0.1 per hour". Also, BaTS uses a list of machine categories (cloud offerings) and the maximum number of machines in each category, to which the user has access to. We use the term *cluster* for the machines of a category (we do not, however, assume any kind of hardware clustering or co-location).

Our cost model assumes that machines can be allocated (reserved and charged) for given machine reservation cycles, called *accountable time unit* (ATU), expressed in minutes. For simplicity, we currently assume that the ATU is the same for all clusters, e.g., sixty minutes. Each cluster, however, has its own cost per ATU per machine, expressed in some currency.

Our very general task model does not allow any hard budget guarantees for the execution of the entire bag. Since BaTS has no a-priori information about the individual execution time of each task, we cannot guarantee that a certain budget will be definitely sufficient. The case might always occur that one or more outlier tasks, with exceptionally high completion time, might be scheduled only towards the end of the overall execution. With sufficiently large bags, however, this denotes only a corner case. We can, however, guarantee that a certain threshold budget (specified by a user guided by our estimates) will not be exceeded at the penalty of an incomplete bag execution.

Figure 1 sketches BaTS' overall system architecture. BaTS itself runs on a *master* machine, likely outside a cloud environment. Here, the bag of tasks is available. On the left side, Figure 1 sketches the sample phase module architecture. Here, BaTS learns the bag's stochastic properties and generates a list of budget estimates

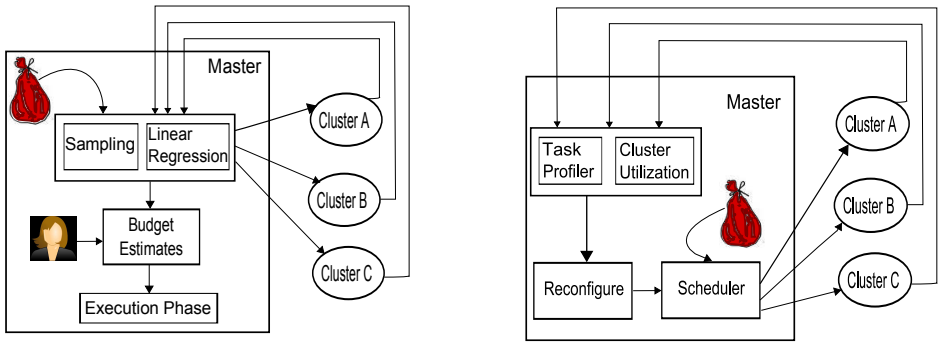

Fig. 1. BaTS system architecture: sampling phase (left) and execution phase (right).

accordingly. The user is then asked to select one of the budgets corresponding to a desired schedule. The user-selected schedule then determines the machines allocated by BaTS for the execution phase, shown on the right side of Figure 1. Here, BaTS allocates machines from various clusters and lets the scheduler dispatch the tasks to the cluster machines. Feedback, both about task completion times and cluster utilization, is used to reconfigure the clusters periodically.

In this section we explain the statistical and algorithmic mechanisms behind BaTS. Section 2.1 describes how BaTS learns and maintains the completion time distribution characteristic for each machine category. In Section 2.2 we describe how the efforts of the sampling phase can be minimized using linear regression across clusters. We explain how we compute budget estimates from the completion time distributions in Section 2.3.

2.1. *Profiling task execution time*

BaTS learns execution time-related information by constantly observing the runtimes of submitted tasks. The basic idea is to estimate an average of the task execution time for each cluster. For this purpose, BaTS uses a cumulative moving average mechanism. Based on these estimates, BaTS decides which combination of machines would satisfy the budget constraint and optimize the makespan. During the execution phase, we also provide the decision loop with feedback from monitoring the actual progress made, as will be described in Section 3.1.

We profile the task execution time on a per-cluster level. When the sampling phase module is absent, we perform an implicit sampling phase within the execution phase. The theoretical foundations are the same for this implicit sampling and for the separate sampling phase. (The latter is a significantly optimized variant of what is described here.)

We use a small sample set as an initial subset of data points; one set per cluster. The size n_s of the sample set can be computed with respect to a certain confidence level, based on the canonical statistical formula for sampling with replacement [4]:

$$n_s = \left\lceil \frac{N * z_\alpha^2}{z_\alpha^2 + 2 * (N-1)\Delta^2} \right\rceil,$$

where N is the size of the bag, $\Delta \in \{0.10, 0.15, 0.20, 0.25\}$ are typical values for the error level, $\alpha \in (0, 1)$ is the confidence level and z_α is related to the Gaussian cumulative distribution function, usual values being: $z_{0.90} = 1.65$, $z_{0.95} = 1.96$ and $z_{0.99} = 2.58$. To correctly approximate the sampling with replacement model, N must be much larger than n_s; in practice, $n_s \leq 0.05 * N$ provides the desired property. However, n_s has an upper bound given by $\left\lceil \frac{z_\alpha^2}{2*\Delta^2} \right\rceil$, as shown in Figure 2. This means that relatively small samples are sufficient to estimate task runtimes reasonably well.

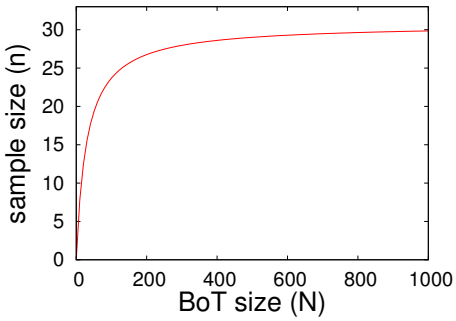

Fig. 2. Sample size variation w.r.t. bag size; $\alpha = 0.95$, $z_{0.95} = 1.96$, $\Delta = 0.25$

From the moment at which all sample tasks of a cluster are finished, we derive an average task execution time per cluster (T_i, expressed in minutes), computed as a modified cumulative moving average [5] of task execution times seen so far:

$$T_i = \frac{\sum_{k=1}^{tasks_{running}} \tau_k + rt_{done}}{tasks_{running} + tasks_{done}}$$

We use the execution times of the sample tasks as indicators for all tasks from the bag. For this purpose, we maintain an ordered list of the execution times from the sample. Whenever T_i is (re-)computed later during the run, a task j submitted on cluster i which has not yet finished execution at this time is estimated by (τ_{j_e}) as the average of sample set tasks runtimes higher than τ_j, the time elapsed since its submission:

$$\tau_{j_e} = \frac{\sum_k^n \tau_k}{n - k + 1}, \quad \tau_k > \tau_j$$

This estimate is used when calculating the new T_i, representing task j as one of those from the tail of the sample's distribution.

The average estimated task execution time for cluster i represents the mapping between the bag-of-tasks and a machine of type i. Therefore any such machine can execute $\frac{1}{T_i}$ tasks from the bag per minute and this quantity is the theoretical average estimated speed of a machine of type i. The T_i values provide information about the quality of the machines in cluster i with respect to the bag-of-tasks currently under execution.

T_i is initialized when the sample set tasks sent to a cluster i finish. After the initialization step we update T_i at given monitoring intervals. We chose the monitoring interval small enough to enable timely detections of possible constraint violations.

2.2. *Minimizing the sampling-phase efforts*

We aim to enable users with a choice of schedules for their bag, having different makespans and budget requirements. We achieve this by decoupling the sampling phase from the execution phase, as shown by Figure 1. In the sampling phase, BaTS learns the stochastic properties (e.g. the runtime distribution, average execution time) of the user's bag of tasks w.r.t. each participating cluster. Based on these estimates, BaTS computes relevant schedules and their respective required budget estimates, prompting the user for selection. The execution phase controls the actual cost incurred by the bag to comply with the user selection.

In order to estimate stochastic properties of the bag-of-tasks w.r.t. each participating cluster, we need to execute n_s randomly selected samples on each cluster. However, there are two problems with this approach: (a) it incurs the cost of executing $n_s \times n_{\mathsf{clusters}}$ tasks, while actually executing only n_s distinct tasks; and (b), it uses all machine types, no matter how un-profitable. We can increase the efficiency by executing different full sample sets on each cluster. This approach remains valid from a stochastic point of view, but does not address the second problem.

We propose a model in which task execution times on different machine types (clusters) exhibit a linear dependency:

$$t_{i,k} = \beta_{0_{k,j}} + \beta_{1_{k,j}} \times t_{i,j}$$

where $t_{i,k}$ is the expected execution time of a task i on a machine of type k and it is calculated based on the execution time of that task on a machine of type j. Based on this assumption, we use linear regression to estimate task runtimes across clusters. Results from statistics [6] show that for linear regression a sample set $n_{lr} = 7$ suffices to accurately compute $\beta_{0_{k,j}}$, $\beta_{1_{k,j}}$. We randomly select from the bag n_s tasks. We replicate 7 tasks from this set on all machine types. The remaining $n_s - n_{lr}$ tasks are distributed among all the machines in a self-scheduling manner. When we collect all the runtimes of the 7 replicated tasks, we arbitrarily select one cluster (base) and we compute all pairs $(\beta_{0_{k,\mathsf{base}}}, \beta_{1_{k,\mathsf{base}}})$, with $1 \leq k \leq n_{\mathsf{clusters}}, k \neq \mathsf{base}$.

We use these pairs to estimate the runtimes on cluster base for those tasks part of the $n_s - n_{lr}$ set which where executed on other machine types:

$$t_{i,\mathsf{base}} = \frac{t_{i,k} - \beta_{0_{k,\mathsf{base}}}}{\beta_{1_{k,\mathsf{base}}}}$$

We now have a complete sample set and we can proceed to derive the stochastic properties of the bag w.r.t. the base cluster. Furthermore, based on ($\beta_{0_{k,\mathsf{base}}}$, $\beta_{1_{k,\mathsf{base}}}$) and ($\beta_{0_{j,\mathsf{base}}}$, $\beta_{1_{j,\mathsf{base}}}$) we can derive the linear dependency parameters for any two machine types k, j:

$$\beta_{0_{k,j}} = \beta_{0_{k,\mathsf{base}}} - \beta_{1_{k,\mathsf{base}}} \times \frac{\beta_{0_{j,\mathsf{base}}}}{\beta_{1_{j,\mathsf{base}}}} \quad ; \quad \beta_{1_{k,j}} = \frac{\beta_{1_{k,\mathsf{base}}}}{\beta_{1_{j,\mathsf{base}}}}$$

Based on the complete sample set of the base cluster, we generate complete sample sets for all other participating clusters. We then use these sample sets to populate the sample sets required by the task profiler presented in Section 2.1.

2.3. *Computing budget estimates from the completion time distributions*

We use the average task execution time of each cluster $C_i, i \in \{1, ..., C_{nc}\}$ participating in the bag-of-tasks execution to compute estimates of the makespan (T_e) and budget(B_e) needed for the bag-of-tasks execution:

$$T_e = \frac{N}{\sum_{i=1}^{C_{nc}} \frac{a_i}{T_i}} \quad ; \quad B_e = \left\lceil \frac{T_e}{ATU} \right\rceil * \sum_{i=1}^{C_{nc}} a_i * c_i,$$

where N is the number of tasks in the bag, $c_i, i \in \{1, ..., C_{nc}\}$ is the cost per ATU for a machine of type C_i and $a_i, i \in \{1, ..., C_{nc}\}$ are the numbers of machines allocated from each cluster. Our current approach finds the best makespan affordable given the user-specified budget (B): We minimize T_e by maximizing the number of executed tasks per minute, and therefore per ATU, while the cost of executing all N tasks at this speed stays within the user specified budget:

$$\begin{aligned} &maximize \ \sum_{i=1}^{C_{nc}} a_i * \frac{1}{T_i} \\ &subject\ to \ \left\lceil \frac{N}{ATU * \sum_{i=1}^{C_{nc}} \frac{a_i}{T_i}} \right\rceil * \sum_{i=1}^{C_{nc}} a_i * c_i \leq B \end{aligned}$$

where $A_i, i \in \{1, ..., C_{nc}\}$ is the maximum number of machines of type i.

Modified Bounded Knapsack Problem (BKP) solver. We solve the above non-linear integer programming problem by modifying the Bounded Knapsack Problem (BKP). In general, the BKP takes a set of item types, where each type j is

characterized by a profit p_j, a weight w_j and a maximum number of items b_j, and determines the number of items (x_j) from each type to be packed such that it maximizes the total profit while the total weight is less than a certain limit (W):

$$\begin{aligned} &maximize \ \sum_{j=1}^{m} p_j * x_j \\ &subject\ to \ \sum_{j=1}^{m} w_j * x_j \leq W, \quad x_j \in \{0, 1, ..., b_j\} \end{aligned}$$

We reformulate the Bounded Knapsack Problem (BKP) in the following way: maximize the total speed per ATU (profit), while maintaining the total cost within the budget. A cluster becomes an item type with profit $\frac{1}{T_i}$ and cost c_i and the bounds for each item type are given by the maximum number (A_i) of available machines (including the ones already acquired by BaTS) in the respective cluster. The solution to this problem represents a machine configuration, where the number of machines from a cluster i is the number of items of the corresponding type. Intuitively, our modified BKP looks for the fastest combination of machines whose total cost per ATU is within a given limit.

Though BKP is NP-complete, it can be solved in pseudo-polynomial time either by expressing it as a 0-1 Knapsack Problem [7] or by using a specialized algorithm [8]. Since both the number of machine types as well as the number of machines of each type are small (compared to the number of tasks), the input of the reformulated BKP can be considered small in its *length*, not only in its *value*, which greatly reduces the time needed to find an exact solution. We chose to solve our modified BKP as a 0-1 Knapsack Problem, using dynamic programming. We define recursively P(i,w):

$$\begin{aligned} P(0, w) &= 0 \\ P(i, 0) &= 0 \\ P(i, w) &= max\{P(i-1, w), p_i + P(i-1, w-c_i)\}, \\ &\quad\ if\ c_i \leq w \\ P(i, w) &= P(i-1, w),\ otherwise \end{aligned}$$

where $P(i, w)$ is the profit obtained by using i items with an average cost per ATU of w. In order to find the candidate solutions, we compute $P(\sum_{i=1}^{C_{nc}} A_i, \sum_{i=1}^{C_{nc}} c_i * A_i)$. We filter the solution set using the constraint

$$price * \left\lceil \frac{N}{ATU * P_{cand}(m, price)} \right\rceil \leq B \ ,$$

where N is the number of tasks to be executed and P_{cand} is the candidate solution representing m machines that cost *price* per ATU. The final solution is processed

to obtain the number of machines from each type.

The complexity of our modified BKP is dominated by $(\sum_{i=1}^{C_{nc}} A_i) * (\sum_{i=1}^{C_{nc}} c_i * A_i)$, while an algorithm that searches exhaustively for all possible solutions is dominated by $\prod_{i=1}^{C_{nc}} A_i$. This allows BaTS to re-compute machine configurations at runtime.

Adjusting BKP's results to indivisible tasks. In previous work [3] we have identified a rounding problem. The rounding problem arises from BKP considering a fluid computation time across reserved machines, whereas individual tasks can not be split across multiple machines and thus expose discrete runtime requirements. As a consequence, though the total computation time of reserved machines covers the total computation time required by the execution of the bag, in some cases the individual computation time of reserved machines did not allow execution of the remaining tasks in the bag, forcing BaTS to either abort execution or reconfigure to a much slower schedule. We have analyzed the particular cases and found that we can predict very early when such a situation would arise.

Let us assume that the user chose a schedule Sched. The schedule is characterized by an input budget B, an output configuration $\{a_k\}$ (the number of machines of type k) and an output makespan M (expressed in ATUs). The makespan is expressed in accountable time units for cost computation reasons. We can also compute an expected average makespan, expressed in minutes. A machine of type k is expected to complete on average n_k tasks during the makespan M, with $n_k \in \mathbb{N}$.

$$n_k = \left\lfloor \frac{M}{T_k} \right\rfloor$$

We identify high risk schedules by comparing the total number of tasks in the bag, N, with the average number of tasks covered by the machines in the schedule's configuration:

$$\Delta N = N - \sum_{k=1}^{k=mt} a_k * n_k$$

Theorem 1. For a given schedule Sched, ΔN has an upper bound equal to $\sum_{k=1}^{k=mt} a_k, a_k \in \mathsf{Sched}$.

Proof. By construction, as a solution to BKP given as input N and B, a schedule Sched satisfies

$$N \leq \sum_{k=1}^{k=mt} a_k * \frac{M}{T_k}$$

Since

$$\Delta N = N - \sum_{k=1}^{k=mt} a_k * n_k$$

follows that

$$\Delta N \leq \sum_{k=1}^{k=mt} a_k * \frac{M}{T_k} - \sum_{k=1}^{k=mt} a_k * n_k$$

Let us remember that

$$n_k = \left\lfloor \frac{M}{T_k} \right\rfloor$$

Follows that

$$\Delta N \leq \sum_{k=1}^{k=mt} a_k * \frac{M}{T_k} - \sum_{k=1}^{k=mt} a_k * \left\lfloor \frac{M}{T_k} \right\rfloor$$

which can be written as

$$\Delta N \leq \sum_{k=1}^{k=mt} a_k * \left(\frac{M}{T_k} - \left\lfloor \frac{M}{T_k} \right\rfloor \right)$$

Since $M, T_k > 0$,

$$\frac{M}{T_k} - \left\lfloor \frac{M}{T_k} \right\rfloor \leq 1$$

and since $a_k \geq 0$,

$$a_k * \left(\frac{M}{T_k} - \left\lfloor \frac{M}{T_k} \right\rfloor \right) \leq a_k$$

Finally

$$\sum_{k=1}^{k=mt} a_k * \left(\frac{M}{T_k} - \left\lfloor \frac{M}{T_k} \right\rfloor \right) \leq \sum_{k=1}^{k=mt} a_k$$

and

$$\Delta N \leq \sum_{k=1}^{k=mt} a_k$$

□

When $\Delta N > 0$, individual computation times are likely to force BaTS to abort execution or reconfigure to a much slower schedule. In such cases, we have to improve the schedule such that we provide enough individual computation time for the ΔN tasks. We identify two cases which require different handling by analyzing the high risk schedule (HRS): (a) the HRS has not reached the upper limit of machines on all participating types and adding a bit of money could either replace cheaper,

but slower machines with better ones, albeit more expensive, or simply allow the acquisition of one more machine for the entire execution phase; (b) the HRS contains the maximum number of machines available from each participating type and cannot be improved by adding more machines of some type(s).

When (a) occurs, we refine the HRS by iteratively increasing the input budget B_i until ΔN becomes zero or negative, or a threshold ("cushion") budget has been reached.

When (b) occurs, simply increasing the input budget B_i does not help, since BKP cannot find a faster configuration for higher B_i. For this case, we compute a ΔB extra ("cushion") budget meant to pay for the execution of ΔN final tasks when their execution would be aborted otherwise. BaTS is aware whether there is an extra budget for this run and how many tasks the extra money is meant for.

In this section, we have explained how BaTS can estimate makespans and budget requirements for a given bag of tasks on a set of cloud clusters. BaTS presents several estimated budget/makespan combinations to the user, ranging from the lowest budget (with high makespan) to the fastest makespan (with higher budget) and some values in between. The user can then make an educated guess about expected runtimes, costs, and choose a combination to his or her likings.

3. Scheduling Bags of Tasks under Budget Control

Using the statistical properties of the bag, we let the user to select a feasible schedule that satisfies her requirements. Based on the chosen budget/makespan combination, BaTS has to implement the chosen option in the execution phase.

Decoupled from the sampling phase, the execution phase is dedicated to observing and enforcing the user's choice. Apart from its high-throughput behavior, the execution phase must also monitor its own compliance with the user selected schedule, and perform corrective actions (cluster reconfigurations) if necessary.

3.1. *Monitoring the plan's execution*

As described so far, BaTS estimates task runtimes and the related costs based on the tasks that have been completed during the initial sampling phase, resulting in an initial machine allocation. At regular monitoring intervals, this initial plan is revisited to accommodate the actual progress of the bag of tasks. BaTS uses a monitoring interval equal to a (small) fraction of the ATU, but at least equal to 5 minutes.

There are two reasons why the initial plan needs continuous refinement. First, the average task completion time gets refined with each completed task. Second, the allocated worker machines are not running in lock-step, such that each machine has its own phase of ATU starting time, and its own ATU utilization given the actual tasks it gets to execute that might leave unused time intervals. If, at a given monitoring interval, the new information indicates a possible budget violation, BaTS has to find another machine allocation, possibly marking certain (expensive)

machines for being preempted at the end of their ATU, and/or other (cheaper) machines to be added instead.

For evaluating the current machine allocation, BaTS checks for a possible discrepancy between the remaining budget and the remaining size of the bag. The estimated number of tasks left in the bag, N_e, describes the utilization of the paid ATUs for all workers. The remaining budget and its current distribution among the active workers is reflected in the potential number of executed tasks, N_p. The comparison between N_e and N_p provides feedback on whether the current scheduling plan fits the (new) information about the tasks of the bag.

For every cluster $C_i, i \in \{1, ..., C_{nc}\}$ we maintain a list of all machines $m_j, j \in \{1, ..., m_{max_i}\}$ that participated at some point in the computation. For every machine m_j we remember the number of executed tasks (nt_{m_j}), the time spent executing tasks (rt_{m_j}) and the total uptime (up_{m_j}).

According to our economical model, the current ATU of each active machine has been paid for once the machine enters it. However, the machine did not run yet for the whole corresponding time. This means that tasks which will be executed by the end of the machine's current ATU are still in the bag. Follows that the current size of the bag is not an accurate indicator of the necessary amount of money to finish the computation. Therefore, we monitor the actual progress of the bag-of-tasks execution by computing an estimate for the number of tasks, N_e, left in the bag after the time for which we already paid elapses on each machine, at regular intervals. The estimate is based on how many tasks each active machine is likely to execute during the remaining span of their respective current ATU.

The remaining span does not include the expected runtime of the task currently executed by the machine (i.e. if the machine is not marked for preemption, the current task could take the machine to the next ATU). If a machine is not marked for preemption we use the estimate of its uptime ($up_{m_{j_e}}$) and its current speed (v_{m_j}) to compute the expected number of tasks executed by it. We compute $up_{m_{j_e}}$ based on the current uptime up_{m_j}, elapsed runtime of task (τ) the task currently executed on m_j and its estimated runtime (τ_e).

$$up_{m_{j_e}} = up_{m_j} + (\tau_e - \tau)$$

We compute the current speed of this machine using the number of executed tasks (nt_{m_j}), the total runtime of executed tasks (rt_{m_j}) and the estimate of the currently running task:

$$v_{m_j} = \frac{nt_{m_j} + 1}{rt_{m_j} + \tau_e}$$

To compute the expected future number of tasks (ft_{m_j}) executed by m_j during this ATU we learn the remaining span (δ_{m_j}) by using $up_{m_{j_e}}$:

$$ft_{m_j} = \lfloor \delta_{m_j} * v_{m_j} \rfloor$$

where

$$\delta_{m_j} = ATU - up_{m_{j_e}} \bmod ATU$$

We can now express N_e as

$$N_e = N_r - \sum_{i}^{C_{nc}} \sum_{j=1}^{m_{max_i}} ft_{m_j}$$

where N_r is the number of remaining tasks in the bag at the current moment (monitoring interval) from which we subtract the number of tasks estimated to be completed in the remaining, already paid-for time.

Workers are not synchronized with each other. Therefore, each worker is at a different stage in the current execution plan. At regular intervals we need to check that the remaining time for each worker according to the current execution plan covers the estimated number of tasks left in the bag. For this purpose, we learn how many tasks each worker is likely to execute in their remaining ATUs of the execution plan, nr_{m_j}. The sum of these tasks is the potential number of executed tasks, N_p:

$$N_p = \sum_{i}^{C_{nc}} \sum_{j=1}^{m_{max_i}} \lfloor (nr_{m_j} * ATU + \eta_{m_j}) * v_{m_j} \rfloor$$

where $\eta_{m_j} = \left\lceil \frac{up_{m_{j_e}}}{ATU} \right\rceil - \frac{ft_{m_j}}{v_{m_j}}$ represents the time left of the previous ATU which could not accommodate the execution of a task, but becomes useful since the machine is not preempted.

We also keep track of the money spent so far by accumulating the current cost of each machine used, including machines no longer active. To estimate the cost of a task k still running we use again its τ_{k_e} value.

Since the remaining budget must accommodate the execution of N_e we check at each monitoring interval that $N_e \leq N_p$ to avoid possible budget violations. When the relationship does not hold, BaTS default behavior is to compute a new configuration which solves the possible budget violation. In turn, this leads to a much slower schedule, as illustrated by results presented in previous work [3].

We extended BaTS by enabling the user to provide a cushion budget that BaTS is allowed to spend dealing with the tasks resulting from the rounding problem. We enhanced BaTS such that, when a possible budget violation is identified and the user had provided a cushion, BaTS checks whether the cause is the rounding problem (Section 2.3) and solves the possible budget violation accordingly.

Intuitively, the user-provided cushion would cover the execution of ΔN tasks which rendered this schedule risky (see Section 2.3), therefore those tasks should be

added to N_p. BaTS identifies possible budget violations triggered by the rounding problem by checking whether $N_e \leq N_p + \Delta N$ holds. If not, BaTS reverts to its default behavior, invoking BKP with the remaining budget (without considering the cushion) and the current estimate of the remaining problem size to find a new machine configuration that satisfies the new budget constraint.

3.2. *The BaTS algorithm*

Based on the mechanisms developed so far, we can formulate the BaTS scheduling algorithm, as shown in Fig. 3 (sampling phase) and Fig. 4 (execution phase). BaTS takes as input a bag-of-tasks with a known size N and the description (cost and maximum number of machines) of a set of available clusters$((c_i, A_i), i \in \{1, .., C_{nc}\})$. Based on N it computes the sample size n_s (see Section 2.1) and acquires a number iw of machines, the *initial workers* on each participating cluster (currently, $iw \in \{1, 4, 7\}$). iw ideally equals 7, but the user may select different values. This set of machines becomes the initial configuration.

```
1: compute n_s = sample size
2: initialize n_lr = 7
3: construct initial configuration C
4: acquire machines according to C
5: while sample phase do
6:     wait for any machine M to ask for work
7:     if M returned result of task T then
8:         update statistics for machine M
9:         if T ∈ replicated set then
10:            update the rt_done for regression point T for c_m
11:        else if T ∈ sample set then
12:            update the rt_done for sample point T for c_m
13:        end if
14:    end if
15:    if replicated set tasks for c_m not finished then
16:        send M a replicated set task T'
17:        place T' in c_m's regression points
18:    else if sample set tasks not sufficient then
19:        send M a randomly selected task T'
20:        place T' in c_m's sample points
21:    end if
22: end while
23: present user with list of possible schedules
```

Fig. 3. Summary of the BaTS algorithm for the sampling phase.

BaTS acts as a master, while the acquired machines act as workers. As workers join the computation, BaTS dispatches randomly selected tasks from the bag in a first-come first-served manner, thus avoiding any bias from the task order within the bag. When a worker running on a machine M from cluster c_m reports back

```
1: if (user selected schedule) then
2:     populate sample sets
3:     load configuration C according to selected schedule
4: else
5:     run implicit sample phase
6:     compute configuration C
7: end if
8: acquire machines according to C
9: while bag has tasks do
10:    wait for any machine M to ask for work
11:    if M returned result of task T then
12:        update statistics for machine M
13:        update the rt_done for c_m
14:    end if
15:    if (monitoring time) then
16:        compute estimates
17:        if constraint violation then
18:            call BKP to compute a new configuration C'
19:            acquire the extra machines required by C'
20:            save C' in C
21:        end if
22:    end if
23:    if number of machines of c_m satisfies C then
24:        send M a randomly selected task T'
25:        remove T' from bag and place it in pending
26:    else if number of machines of c_m should decrease then
27:        release M
28:    end if
29: end while
```

Fig. 4. Summary of the BaTS algorithm for the execution phase.

with a task T's result, BaTS updates the worker-related information (runtime - time spent executing tasks, and the number of tasks executed by M), as well as the total execution time (rt_{done}) for cluster c_m with T's execution time.

BaTS starts with the sampling phase (Figure 3), during which it replicates $n_{lr} = 7$ randomly selected tasks among the iw workers such that each task is executed by one worker from each cluster. BaTS collects these runtimes $(rt_{i1}, \ldots, rt_{iC_{nc}})$ for each task $i, i \in \{1, .., n_{lr}\}$. When a worker running on a machine M from cluster c_m returns a task T_i's result, BaTS updates the rt_{ic_m}, sends the worker another randomly selected task and marks the worker for release at the end of its current ATU. Once all n_{lr} tuples are complete, BaTS computes the linear regression parameters (β_1, β_0) and checks whether $n_s - n_{lr}$ tasks, different from the replicated tasks, are finished. If not, BaTS computes an intermediary most profitable machine type based on runtimes collected so far and continues execution of tasks on one machine of that type until we obtain n_s runtimes of independent tasks. When this stage is done, BaTS is able to compute the list of budget estimates and their respective

schedules. The user selects a schedule, and BaTS resumes execution (Figure 4) after acquiring machines according to the selected schedule's configuration.

During the execution phase (Figure 4), BaTS monitors the actual progress made. The search for a new configuration is triggered when estimates computed at the end of a monitoring interval indicate a budget constraint violation. In this case, the new configuration C' replaces C. BaTS decides whether machine M should continue to be part of the computation or it should be released, based on the current configuration. Note that BaTS computes no schedules until it can derive stochastic properties for all participating clusters (the end of the sampling phase). If there are no tasks finished during a monitoring interval, BaTS updates both the profiling and cluster utilization estimates as described in previous sections.

It is important to note that BaTS execution phase can also function in a standalone mode (lines 4-6 in Figure 4), for those cases where the user has already a clear idea for a desired budget or the linear dependency assumption does not hold.

4. Performance Evaluation

We have implemented a Java-based BaTS prototype using our Ibis platform [9]. The prototype consists of two parts: the master, which can run on any desktop-like machine, and the worker which is deployed on cloud machines. The master component implements the core of BaTS, while the worker is a lightweight wrapper for task execution. All communication between the master and the workers uses Ibis' communication layer, IPL.

We have emulated an environment of two different (cloud) clusters in the DAS-3 multi-cluster system. The physical machines are 2.4 GHz AMD Opteron DP, each has 4 GB of memory and 250 GB of local disk space, running Scientific Linux. Both emulated clusters have 32 machines. Requests for machines that will run the worker component are sent by the master component to the local cluster scheduler (SGE), thus incurring realistic, significant startup times as in real clouds. However, we do not allow queueing delays due to competing requests.

Previous work [3] evaluated extensively the performance of the BaTS execution phase scheduling algorithm in different scenarios and compared it to a budget-oblivious self-scheduling algorithm. Though it showed that BaTS successfully schedules bags of tasks within given, user-defined budget contraints, and that, if the budget allows, BaTS finds schedules comparable in makespan to the those produced by the self-scheduling algorithm, it also identified a serious cost overhead incurred by the implicit sampling phase on each cluster, the latter is being overcome by our new, linear-regression based, initial sampling phase.

Our current performance evaluation consists of two parts. The first part compares the cost of budget estimation (sampling phase) to the cost of the bag's execution itself. It also investigates the accuracy of the linear regression mechanism, which is used for the budget estimates. The second part of the evaluation verifies the execution of the estimated schedules, and their real costs and runtime.

4.1. *Emulation Setup*

For the present performance evaluation we use a workload of medium size, where the assumptions made by the statistical device behind BaTS hold. Recent work [10] on the properties of bags-of-tasks has shown that in many cases the intra-BoT distribution of execution times follows a normal distribution. Accordingly, we have generated a workload of 1000 tasks whose runtimes (expressed in minutes) are drawn from the normal distribution $N(15, \sigma^2), \sigma = \sqrt{5}$. We enforce these runtimes by executing the *sleep* command accordingly.

We assume two different clusters, characterized by their own cost and execution speed, and derived from these, their profitability, a measure for the value offered for the money paid. We define the *profitability* of a machine type (cluster) i with respect to the cheapest machine type available m as the speed increase compared to the cost increase:

$$profitability = \frac{T_m}{T_i} * \frac{c_m}{c_i}$$

where T_m, T_i represent the average theoretical task execution time for clusters m and i, respectively; c_m, c_i represent the cost of using a machine from the respective cluster for one ATU.

One emulated cluster ($\texttt{cluster}_1$) charges \$3 per machine per ATU and executes tasks according to the runtimes drawn from $N(15, \sigma^2), \sigma = \sqrt{5}$. We create 5 different scenarios w.r.t. the price and speed of the other emulated cluster $\texttt{cluster}_2$ compared to $\texttt{cluster}_1$:

- S_{1-1}: (profitability 1) $\texttt{cluster}_2$ charges the same price for a machine and has the same speed;
- S_{1-4}: (profitability 4) $\texttt{cluster}_2$ charges the same price for a machine, but is 4 times as fast;
- S_{4-1}: (profitability 0.25) $\texttt{cluster}_2$ charges 4 times as much (\$12 for a machine) but has the same speed;
- S_{3-4}: (profitability 1.33) $\texttt{cluster}_2$ is 3 times as expensive (charges \$9 for a machine), and is 4 times as fast;
- S_{4-3}: (profitability 0.75) $\texttt{cluster}_2$ is 4 times as expensive (charges \$12 for a machine), and is 3 times as fast.

For emulating variable speeds of $\texttt{cluster}_2$, we modify the parameter to *sleep* accordingly. All prices are per accountable time unit, which we set to 60 minutes (without loss of generality).

4.2. *Estimating runtime distributions (Linear Regression)*

Figure 5 shows how well the runtime distributions of the bags are estimated by the linear regression mechanism. For all 5 scenarios, we compare μ and σ of the

real bag (computed offline from all runtimes) with the values estimated by the LR mechanism, for both clusters c_1 and c_2. It can be clearly seen that LR finds almost perfect estimates based on the rather tiny sample size.

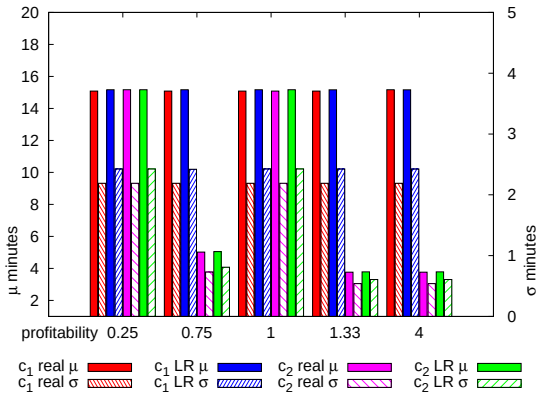

Fig. 5. Comparing bag stochastic properties as estimated by LR with the real values (computed using all generated runtimes).

Table 1 presents the costs of the LR-based sampling phase and the costs of the implicit sampling phase used by BaTS when the LR-based sampling module is absent [3]. We further compare the sampling phase costs to the costs of the cheapest makespan (C.M.) and the fastest makespan (F.M.) possible. In the LR-based case, the latter costs are expressed as the sum of the respective sampling and execution phase costs. For BaTS sampling without LR, the sampling phase is intertwined with the execution phase, and therefore the costs of the cheapest and the fastest makespans are not separate.

Table 1. Costs compared for each profitability case:
for sampling(S), for cheapest makespan(C.M.), and for fastest makespan(F.M.).

	with LR			without LR		
profitability	S	C.M.	F.M.	S	C.M.	F.M.
0.25 (4-1)	105	105 + 732 = 837	105 + 1920 = 2025	450	1026	2034
0.75 (4-3)	105	105 + 696 = 801	105 + 960 = 1065	450	846	1062
1 (1-1)	42	42 + 732 = 774	42 + 768 = 810	180	756	810
1.33 (3-4)	84	84 + 513 = 597	84 + 768 = 852	360	594	768
4 (1-4)	42	42 + 171 = 213	42 + 384 = 426	180	258	384

Our results show that, without prior knowledge of profitability ratios, using too many machines during the sampling phase (as done by BaTS without LR) can incur serious cost penalties. Moreover, the cases without LR where the more expensive machines are also the most profitable show that the gain in the extra number of

tasks executed during the sampling phase is not significant enough to justify the risk of cost penalties.

Overall, the LR-based sampling mechanism has a rather small (but non-negligible) cost. The cost of this sampling phase, however, is not wasted, because the first real tasks are already computed here that need not be repeated in the following execution phase.

4.3. *Budget estimate and control*

The LR-based estimation phase produces several proposed schedules, along with their estimated makespan and cost, ranging from mimimum budget to fastest makespan. The minimum budget (B_{min_1}) is computed as the budget needed for the bag's execution on 1 machine of the most profitable type. This budget is given as input to BKP yielding a schedule (with a configuration possibly consisting of more machines) that gives the cheapest makespan. At the other extreme of the relevant range is the budget required for the fastest makespan,($B_{fastest}$), which is computed as the budget needed for the bag's execution on all available machines. (In either case, BaTS may require a cushion to achieve $\Delta N \leq 0$.) This leads to the following list of proposed schedules:

- Sched_1 corresponds to the budget (B_{min_1}) and represents the cheapest makespan.
- Sched_2 corresponds to a budget of B_{min_1} increased by 20%.
- Sched_3 corresponds to a budget of $B_{fastest}$ decreased by 20%.
- Sched_4 corresponds to the budget ($B_{fastest}$) needed for the bag's execution on all available machines.

We evaluate BaTS' budget estimate and control capabilities using two interesting profitability scenarios, S_{4-1} and S_{3-4}, presented in Section 4.1. S_{4-1} represents a corner case, where all machines have the same speed, but those in $\texttt{cluster}_2$ cost 4 times more ($12 per ATU), while S_{3-4} illustrates a realistic scenario, where machines in $\texttt{cluster}_2$ cost 3 times more ($9 per ATU), but are 4 times faster. For each scenario, we ran (where relevant) BaTS with each budget presented in Table 2. The only exception is that we do not run BaTS in scenario S_{4-1} with schedule Sched_1 since it would run very long while not providing useful insights.

Table 2 summarizes, for both scenarios, the schedules proposed by the LR estimation phase. Each schedule is described by the estimated budget, the necessary budget cushion (in case $\Delta N > 0$), the machine configuration in numbers of machines on ($\mathsf{cluster}_1, \mathsf{cluster}_2$), and the estimated makespan expressed in ATU's.

Results are shown in Figure 6. In scenario S_{3-4}, one interesting case is Sched_1 which uses the B_{min_1} as input to BKP and at first obtains a configuration consisting of 28 $\texttt{cluster}_2$ machines and 1 $\texttt{cluster}_1$ machine. However, BaTS discovers a ΔN=14 and indicates to the user that a 2% increase of B_{min_1} would produce a schedule that renders ΔN negative. The new schedule requires 29 machines from $\texttt{cluster}_2$ and none from $\texttt{cluster}_1$. The new configuration is successful and BaTS

Table 2. Schedules proposed by the LR estimation phase for scenarios S_{3-4} and S_{4-1}.

		budget	cushion	machines	makespan
S_{3-4}	Sched_1	B_{min_1} = \$513	\$11	(0, 29)	2 ATU
	Sched_2	$\lceil 1.2 * B_{\mathsf{min}_1} \rceil$ = \$615		(6, 32)	2 ATU
	Sched_3	$\lceil 0.8 * B_{\mathsf{fastest}} \rceil$ = \$615		(6, 32)	2 ATU
	Sched_4	B_{fastest} = \$768		(32, 32)	2 ATU
S_{4-1}	Sched_1	B_{min_1} = \$732		(4, 0)	61 ATU
	Sched_2	$\lceil 1.2 * B_{\mathsf{min}_1} \rceil$ = \$872		(32, 1)	8 ATU
	Sched_3	$\lceil 0.8 * B_{\mathsf{fastest}} \rceil$ = \$1536	\$97	(32, 19)	5 ATU
	Sched_4	B_{fastest} = \$1920	\$9	(32, 32)	4 ATU

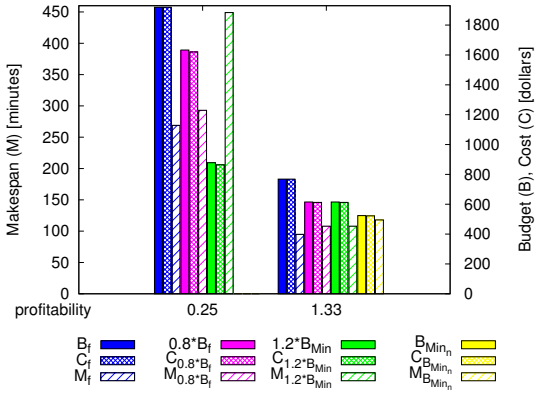

Fig. 6. Budget, cost and makespan for bag after sampling, comparing BaTS in two profitability cases with four different budgets.

finishes the bag execution in 1 hour and 58 minutes.

In the same scenario S_{3-4}, Sched_2 and Sched_3 happen to be the same, albeit obtained in two different ways. The run happens as planned and BaTS finishes the execution in 1 hour and 48 minutes, using a configuration of 32 machines from $\mathtt{cluster}_2$ and 6 machines from $\mathtt{cluster}_1$. When running Sched_4 in scenario S_{3-4}, BaTS finishes the bag in 1 hour and 35 minutes, using all machines from both clusters.

For scenario S_{4-1}, we find several interesting cases. First, when analyzing Sched_3, BaTS finds ΔN=32 on a configuration of 32 machines from $\mathtt{cluster}_1$ and 17 machines from $\mathtt{cluster}_2$. BaTS proceeds to refine the schedule by increasing the budget in 1% increments from 80% of B_{fastest} and finds a new schedule at 0.85*B_{fastest}=\$1633 which has a $\Delta N \leq 0$. The new schedule configuration consists of 32 machines from $\mathtt{cluster}_1$ and 19 machines from $\mathtt{cluster}_2$. The respective run is successful and BaTS finishes the bag execution in 4 hours and 53 minutes within budget.

BaTS computes for the schedule corresponding to B_{fastest}=\$1920 a ΔN=3. In this particular case (all available machines already in use), adding a cushion budget can only lead to a longer makespan. The alternative to using a cushion might be

another longer, but possibly aborted run due to the budget violation. To assess the impact of the user-provided cushion we ran BaTS in this particular setting both with a cushion and without. Results are shown in Figure 7. BaTS pre-computes a cushion of \$9 (3 machines from $\texttt{cluster}_1$ running for at most one more ATU (hour) after the planned number of ATUs expired). When the user agrees to it, BaTS finishes the bag within the initial budget of \$1920 with a makespan of 3 hours and 56 minutes. When the user does not agree to the cushion, BaTS still manages to finish the bag, but it takes 4 hours and 29 minutes. In both runs, BaTS starts with 32 machines from each cluster. However, in the run without cushion, BaTS needs to reconfigure at the first possible budget violation signal. This happens during the first ATU, and BaTS reconfigures to 32 machines from $\texttt{cluster}_1$ and 22 machines from $\texttt{cluster}_2$ starting with the second ATU which leads to the 33 minutes delay compared to the cushioned run. The same possible budget violation arises when BaTS runs with a cushion, but in this case BaTS can ignore it since the difference between the number of tasks left in the bag and the number of tasks that can be accommodated by the remaining budget is less or equal to ΔN; those tasks are covered by the cushion.

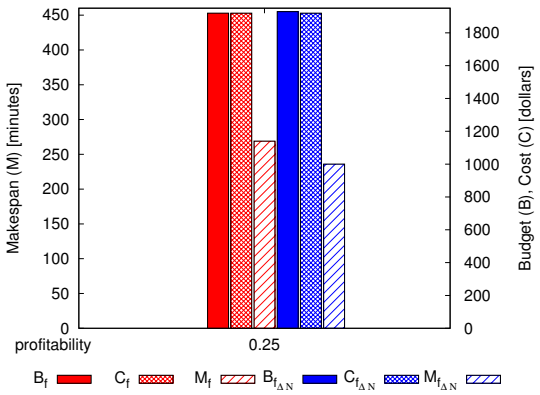

Fig. 7. Budget, cost and makespan for bag with and without cushion.

We run BaTS in the same S_{4-1} scenario with Sched_2, comprising 32 machines from $\texttt{cluster}_1$ and 1 machine from $\texttt{cluster}_2$, and the bag is executed in 7 hours and 29 minutes at a cost of $1.2{*}B_{\mathsf{min}}$=\$872. At a cost of 20% more than B_{min_1}, BaTS delivers a makespan 86% smaller than that corresponding to Sched_1.

5. Related Work

Recent research efforts have addressed different aspects of bag-of-tasks applications. We compare our present proposal to existing research with respect to the assumptions made on task and resource characteristics, the proposed goals, the scheduling

plan characteristics, the performance metrics used to compare against established algorithms, and the respective algorithms.

The assumptions on task characteristics involve the existence of prior knowledge on arrival rate, execution time, or deadline for each task in the bag. Work presented in [11, 12, 13, 14] assumes a-priori known task execution times. One relaxation is found in [15] where all tasks are assumed to be in the same complexity class and there is a calibration step to determine execution time estimates per machine type. The assumptions are further relaxed in [16] to relative complexity classes of tasks, though all the classes are supposed to be known in advance, which is similar to assumptions made by [17] where job classes have known average execution times per machine type. BaTS, in contrast, only assumes that some form of runtime distribution exists, and uses stochastic methods to detect it at runtime.

Work presented in [18] addresses homogenous grid resources and is extended to heterogenous systems in [19]. Completely heterogenous systems are addressed by [13, 14, 16, 20]. Our target platform is composed of several heterogenous sets of homogenous machines, which fits perfectly a composite of scientific grids and cloud systems.

Makespan minimization is the main focus of research done in [11, 20]. This is accompanied by response time minimization at task level in [14]. These scheduling algorithms are compared against traditional algorithms such as Min-Min, Max-Min [21] and Sufferage [22]. A mixture of robustness optimization, while satisfying makespan and price constraints is presented in [13]. It assumes a fixed, one-time cost per machine type. Robustness optimization is the main focus of research conducted in [16]. In contrast, we use an economic model for resource utilization, that matches the current, elastic cloud system offerings.

The mapping between tasks and resources can be done either off-line [13, 19] or on-line. The on-line techniques can be further categorized using the mapping event granularity: fine granularity implies the mapping is performed as soon as a task arrived/is ready [20]. Coarse granularity makes mapping decisions on a batch of tasks. Hybrid approaches are employed in [14, 16]. With BaTS, we consider all tasks to be available for execution when the application starts. However, mapping events are triggered by an (adjustable) timeout.

Recent work [17, 23] shares more similarities with BaTS. The work presented in [17] tackles the mirrored problem: observing a user-specified job response time deadline while minimizing the cost per hour. A more flexible approach is presented in [23], which deals with either minimizing the cost for a user-specified deadline or minimizing the makespan for a user-specified budget. However, both approaches do not consider intermediate solutions which the user could consider best fitted for her needs. In our own previous work [3], we presented our self-scheduling and self-reconfiguration approach for the execution phase and identified the need to guide users by estimating a range of suitable budgets. As presented here, BaTS achieves this by decoupling the sampling phase from the execution phase, while adding a sophisticated, stochastic estimation tool for bag runtimes and budgets.

6. Conclusions

Elastic computing, as offered by Amazon and its competitors, has changed the way compute resources can be accessed. The elasticity of clouds allows users to allocate computers on the fly, according to the application's needs. While each commercial offering has a defined quality of service, users still lack guidance for deciding how many machines of which type and for how long would be necessary for their application to complete within a given budget, and as quickly as possible.

Bags of tasks are an important class of applications that lend themselves well for execution in elastic environments. In this work, we have introduced BaTS, our budget-constrained scheduler for bag-of-tasks applications. BaTS requires no a-priori information about task execution times. It uses statistical methods to execute samples of tasks on all cloud platforms that are available to a user. During an initial sampling phase, BaTS estimates makespans and budget requirements for different combinations of cloud offerings, and presents the user with a choice of either cheaper or faster options. Based on the user's choice, BaTS monitors the progress of the tasks during the bag execution, according to its own predictions. BaTS dynamically reconfigures the set of machines, based on the expected budget consumption and completion time, should this become necessary during execution.

We have evaluated BaTS by emulating different clouds on the DAS-3 multi-cluster system. For each test, we used two clouds with different profitability (price-performance ratio) and let BaTS schedule a bag of 1000 tasks. We have investigated various different profitability ratios.

We have evaluated the quality of the estimates and have shown that, despite the rather tiny sample size, BaTS almost perfectly estimates the bag's properties, at rather low execution costs for the initial sample. We have verified that the actual execution of a bag, performed by BaTS based on the user's choice of budget and makespan, conforms to the predicted times and costs.

Our results are very encouraging, but they also open up new questions. Improving the tail phase of the schedule seems a promising approach for further minimizing BaTS' makespans, without raising the costs incurred. Allowing different accountable time units from cluster to cluster would further enhance BaTS' flexibility. As users are also concerned by other properties, like, for example, energy consumption, more parameters will have to be modelled by BaTS in the future, apart from cost and makespan.

Acknowledgments

This work has partially been funded by the European Commission via grants to the projects XtreemOS and Contrail. We would like to thank Kees Verstoep for enabling us to finish the long-running experiments in time.

References

[1] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[2] Amazon Web Services. http://aws.amazon.com.

[3] A.-M. Oprescu and T. Kielmann. Bag-of-Tasks Scheduling under Budget Constraints. In *2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, Indianapolis, USA, 2010. IEEE.

[4] E. S. Keeping. *Introduction to Statistical Inference*. D. Van Nostrand, Princeton, New Jersey, 1962.

[5] J. F. Kenney and E. S. Keeping. *Mathematics of Statistics*. D. Van Nostrand, Princeton, New Jersey, 1962.

[6] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S. Fourth Edition*. Springer, 2002.

[7] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[8] D. Pisinger. A minimal algorithm for the bounded knapsack problem. Technical report, University of Copenhagen, 1994.

[9] H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, R. Kemp, N. Palmer, G. Wrzesinska, T. Kielmann, F. Seinstra, and C. Jacobs. Real-world distributed computing with Ibis. *Computer*, 43(8):54–62, 2010.

[10] A. Iosup, O. Sonmez, S. Anoep, and D. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 97–108, New York, NY, USA, 2008. ACM.

[11] Y. C. Lee and A. Y. Zomaya. Practical scheduling of bag-of-tasks applications on grids with dynamic resilience. *IEEE Transactions on Computers*, 56(6):815–825, 2007.

[12] P. Sugavanam, H. Siegel, A. Maciejewski, J. Zhang, M. Shestak, M. Raskey, A. Pippin, R. Pichel, M. Oltikar, A. Mehta, P. Lee, Y. Krishnamurthy, A. Horiuchi, K. Guru, M. Aydin, M. Al-Otaibi, and S. Ali. Robust processor allocation for independent tasks when dollar cost for processors is a constraint. In *IEEE International Conference on Cluster Computing*, 2005.

[13] P. Sugavanam, H. J. Siegel, A. A. Maciejewski, M. Oltikar, A. M. Mehta, R. Pichel, A. Horiuchi, V. Shestak, M. Al-Otaibi, Y. G. Krishnamurthy, S. A. Ali, J. Zhang, M. Aydin, P. Lee, K. Guru, M. Raskey, and A. J. Pippin. Robust static allocation of resources for independent tasks under makespan and dollar cost constraints. *J. Parallel Distrib. Comput.*, 67(4):400–416, 2007.

[14] C. Weng and X. Lu. Heuristic scheduling for bag-of-tasks applications in combination with qos in the computational grid. *Future Generation Comp. Syst.*, 21(2):271–280, 2005.

[15] H. González-Vélez. Self-adaptive skeletal task farm for computational grids. *Parallel Comput.*, 32(7):479–490, 2006.

[16] J. Smith, L. D. Briceno, A. A. Maciejewski, H. J. Siegel, T. Renner, V. Shestak, J. Ladd, A. M. Sutton, D. L. Janovy, S. Govindasamy, A. Alqudah, R. Dewri, and P. Prakash. Measuring the robustness of resource allocations in a stochastic dynamic environment. In *IPDPS*, 2007.

[17] Ming Mao, Jie Li, and Marty Humphrey. Cloud Auto-scaling with Deadline and Budget Constraints In *The 11th ACM/IEEE International Conference on Grid Computing (Grid 2010)*, Brussels, Belgium, 2010.

[18] Q. Zhu and G. Agrawal. An adaptive middleware for supporting time-critical event response. *International Conference on Autonomic Computing*, pages 99–108, 2008.

[19] Q. Zhu and G. Agrawal. A resource allocation approach for supporting time-critical applications in grid environments. *International Symposium on Parallel and Distributed Processing*, pages 1–12, 2009.

[20] W. Cirne, D. P. da Silva, L. Costa, E. Santos-Neto, F. V. Brasileiro, J. P. Sauvé, F. A. B. Silva, C. O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. In *ICPP*, 2003.

[21] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, 1977.

[22] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Eighth Heterogeneous Computing Workshop*, pages 30–44. IEEE Computer Society Press, 1999.

[23] K. Liu, H. Jin, J. Chen, X. Liu, D. Yuan, and Y. Yang. A compromised-time-cost scheduling algorithm in swindew-c for instance-intensive cost-constrained workflows on a cloud computing platform. *International Journal of High Performance Computing Applications*, 24(4):445–456, 2010.