A Framework for Cloud-aware Development of Bag-of-Tasks Scientific Applications

Alessandra de Benedictis
Università di Napoli Federico II,
DIETI,
Napoli, Italy
E-mail: alessandra.debenedictis@unina.it

Massimiliano Rak
Seconda Università di Napoli,
DII,
Aversa, Italy
E-mail: massimiliano.rak@unina2.it

Mauro Turtur
Università del Sannio,
DING,
Benevento, Italy
E-mail: mauro.turtur@unisannio.it

Umberto Villano
Università del Sannio,
DING,
Benevento, Italy
E-mail: villano@unisannio.it

Abstract: The potential of cloud computing is still underutilized in the scientific computing field. Even if clouds probably are not fit for high-end HPC applications, they could be profitably used to bring the power of low-cost and scalable parallel computing to the masses. But this requires simple and friendly development environments, able to exploit cloud scalability and to provide fault tolerance. This paper presents a framework built on the top of a cloud-aware platform (mOSAIC) for the development of bag-of-tasks scientific applications.

Keywords: Cloud Computing; Cloud-aware programming; Scientific Computing; Bag-of-tasks.


Biographical notes: fuyuyfu itiut uyuoiyiuo in iut ui t
uyt8t 8yt8 tit it iut ui tiuyiu in ti

1 Introduction

Notwithstanding the explosive diffusion of cloud computing observed in the last few years, it seems that the great potential of this paradigm is still underutilized. In most cases, this is due to security issues (see for example [23] for a survey of the state of the art in cloud security). But, even when security is just an option, sometimes users are reluctant to move their applications to the cloud due to a perceived inadequacy of the cloud paradigm to fit their computing needs. This may be due to substantial differences between the “traditional” and the cloud-based paradigm, to the lack of adequate development tools to support the porting legacy application to the cloud, and
to a partly misplaced belief that the move to a multitenant virtualized environment should adversely affect application performance.

A field where the transition to the cloud has turned out to be particularly difficult is high-performance and scientific computing. First of all, the users/developers of scientific codes are not prone to tolerate the moderate performance losses due to the systematic use of virtualization and, above all, to the use in cloud data centers of networks designed mainly for scalability, and not for performance [2]. The high variance of response times due to multitenancy and to loads hidden from the user view and control, along with always-possible transient failures of the cloud infrastructure, do the rest. As a matter of fact, cloud computing is inherently unfit for high-end scientific applications, which in the near future are likely to be still executed in purposely-designed and dedicated HPC systems.

However, there is a wide range of applications widely used in science, engineering and for commercial purposes that have highly variable response times, are moderately CPU intensive, are not immediately suitable for GPU computing and are made up of loosely coupled tasks, so that computation easily dwarfs communication times. We think that this class of “para-scientific” applications is very close to be an ideal candidate for execution on the cloud. The major advantage is economic: the cost for a small set of virtual cores can be very low, especially if there are relaxed time constraints for obtaining the results. A wise choice among the provider offerings often allows to acquire the computing resources needed at very low cost (see for example the EC2 Spot Instances offer [3]). This enables any organization to run parallel code whenever needed, at a low and well-known cost, without investing the capital in rapidly obsoleting parallel hardware. The second important point is linked to the cloud elasticity, which allows to scale in/out the number of virtual cores on-the-fly (i.e., while the application is running), based on the particular job requirements, paying just for the resources actually used. In other words, cloud computing is also a great opportunity for everyone to experiment and to exploit parallel computing at low cost, using a comfortable pay-as-you-go model.

However, wealth of computer power at low cost it is not sufficient to bring scientific users to the cloud. It is also necessary to provide an easy and widely-understood programming paradigm along with ready-to use tools. The options to run parallel applications in the cloud are essentially the following:

- the use of MPI in a virtual cluster: existing MPI applications are executed as-they-are on a virtual cluster leased from a cloud provider;
- the use of structured paradigms: applications are written using structured paradigms oriented to distributed computing (e.g., Map-Reduce [11], Dryad [16]);
- the use of native cloud APIs and programming artifacts: applications are written by exploiting vendor or cloud-neutral API and resources [5], [4];
- the use of a cloud-enabled programming platform: cloud-aware application are developed on the top of a cloud-aware programming framework [21], [22].

In our opinion, none of these solutions seems particularly fit to bring cloud parallel computing to the masses. The use of virtual clusters and MPI is straightforward, but it fails to exploit cloud elasticity, as the amount of computing resources is leased statically (the size of the virtual cluster). Furthermore, MPI codes have typically no provision for fault-tolerance. Even a transient failure of a computing node, always possible in a cloud, is likely to induce the failure of long-running computations. Structured distributed computation paradigms are intrinsically fault-tolerant, but require at least code restructuring. Moreover, their performance on the cloud is far from optimal at the state of the art [18]. Native cloud programming is performance-optimal and suitable to exploit at best elasticity, but it is complex and not widely understood. Cloud-aware programming frameworks try to mitigate the problem, but currently they are not enough mature for everyday use.

In [10], we introduced a light-weight framework for the development of cloud bag-of-tasks applications. This paper is an extension of the mentioned one, with a focus on performance issues, dealt with by considering a real-world application. The bag-of-tasks programming paradigm (also known as master-worker, processor farm, just to mention a few) is widely understood and ubiquitous in small and medium-scale scientific computing. Even if it is not fit to solve any kind of scientific problem, a simple cloud framework is likely to be an interesting opportunity for the not-particularly-specialized parallel computing practitioners. A cloud bag-of-tasks framework has already been developed for the Azure cloud platform, exploiting Azure native APIs and components [1]. The solution proposed here is different, as it relies on a vendor-neutral programming platform (the mentioned mOSAIC [21]), and hence can be used on the resources of any provider, or even on a federated provider resource broker.

It should be noted that the development of a cloud-aware scientific application (or of a framework for scientific application development, which is the essentially the same) on the top of native cloud artifacts and API is a challenging task. Distributed-memory scientific programming aims at high performance. This is often obtained at the expense of dynamicity in the number of processes/tasks, which is at odds with cloud elasticity, and at the expense of fault-tolerance, which should be suitably taken into account in clouds, where transient faults are always possible, if not frequent. Furthermore, the interaction mechanisms of scientific codes (message exchange, collective communication primitives, . . . )
involve synchronization among processes. The cloud native programming artifacts (queues, blobs, key-value stores, . . . ) are mainly oriented to asynchronous interactions and are designed for scalability and fault-tolerance, not for low latency. Trying to make the two programming styles coexist is an interesting field for research, and one that would lend to interesting opportunities for scientists. In our intentions, this paper should be a first step in that direction.

The remainder of this paper is structured as follows. In the next section we will examine related work. Section 3 deals with the mOSAIC programming platform. Then Section 4 presents the rationale and the architecture of the framework we have implemented for the development of bag-of-tasks applications in the cloud. The performance analysis of a combinatorial optimization application and its comparison with an equivalent MPI code is proposed in Section 5. The paper closes with our conclusions and plans for future research.

2 Related work

For the reasons mentioned in the introduction, the literature on the use of clouds to execute scientific applications is not too wide. The potential of clouds for scientific computing linked to economity and to on-demand provision is discussed in [17]. The performance disadvantages of clouds for scientific computing workloads are presented in [15].

In [19], the applicability of cloud platforms, and in particular of Microsoft Azure, to scientific computing is studied by implementing a well-known bioinformatics algorithm (BLAST). An approach similar to the one presented in this paper, although with a few significant differences, as discussed in the introduction, is presented in [1], which shows the implementation of a framework for exploiting the bag-of-tasks pattern in the Azure cloud platform. A Java framework for the development of fault-tolerant applications is proposed in [20].

A few papers discuss how to exploit the intrinsic elasticity of clouds, i.e., the ability to increase or decrease the amount of computing resources used for application execution. In [12], the Authors present Cloudline, a platform for the development of generic scientific applications able to exploit at best cloud elasticity. The paper [24] tackles the problem of adding elasticity to existing MPI codes. This is obtained by terminating the execution and restarting the program on a different amount of resources, scaling up/down the number of computing nodes used. The execution of MPI codes over a cloud-aware communication library is discussed in [13], where CMPI, a novel MPI library based on the cloud-oriented optimization proposed in [14], is presented.

3 mOSAIC

mOSAIC is a cloudware that builds up a Platform-as-a-Service on the top of computing resources leased in Infrastructure-as-a-Service mode from a single or even from multiple cloud providers [21]. The mOSAIC platform offers a set of components that can be used in mOSAIC-enabled programs in order to build cloud applications that are intrinsically distributed, elastic and scalable.

Figure 1 shows the typical programming artifacts of the mOSAIC cloud platform; further ones could be developed, as the platform is extensible. All artifacts are managed by the enabling platform as components, which can be launched in multiple instances. mOSAIC applications are composed of collections of such components. In the figure, the artifacts are divided into three main groups:

- **Computation** components, whose main role is to execute specific algorithms (consuming CPU resources);
- **Communication** components, dedicated to communication among components of the application and/or communication with customers or other applications (i.e., internal and/or external interfaces);
- **Storage** components, dedicated to provide reliable storage in the cloud.

mOSAIC cloudlets are components, developed using the mOSAIC API, able to run on the top of the platform and dedicated to computational purposes. Cloudlets can be started multiple times (thus creating multiple cloudlet instances) and they automatically balance the load and share their status. Cloudlets perform specific tasks and interact with other components through communication components. In a typical setup, a communication component providing an external interface (HTTPgw) receives service requests from a customer, and forwards them through one or more queue to the cloudlets that perform the computational tasks required.

The communication components offer solutions that enable communication among different components
(internal interfaces) or with customers (external interfaces). An example of component of the first class is the Queue, which can be used to exchange messages. HTTPgw is an example of external interface, as it offers an HTTP interface to the customer. Each HTTP request is forwarded to the cloud application through queues (internal interface).

The mOSAIC Queue component is a customized version of the RabbitMQ queue server [25]. It is a software component that offers an API to create messages queues, which can be used by the applications to communicate each other. After that a queue has been created, all connected applications can send a message through it. All applications registered as consumers will be able to receive the message.

Queue servers offer many different possible configurations for queue-based programming. In mOSAIC, the default configuration has been devised to implement a publish/notify approach with acknowledgement. However, the developer may reconfigure the queue server in order to match his specific requirements. The default configuration works as follows: an application can register itself to the queue as publisher and/or as consumer. When a message is published, one of the registered consumers receives a notification and can read the message. The message will not be deleted from the queue until an acknowledgement has been received. If the consumer that got the message fails without sending an acknowledgement, the queue server understands from the death of the connection that the message has not been fully processed, and delivers it to another consumer.

This setup helps developing fault-tolerant application using the mOSAIC cloud platform. If a cloudlet does not confirm the successful processing of the message, e.g., if the instance crashes mid-processing, the message reappears in the queue and can be consumed by another cloudlet. This behavior is similar to the one offered by Azure Queue storage service, which uses explicit timeouts to assume a consumer dead [1].

The KVstore component is based on Riak [6]. It offers a persistent NoSQL storage service to cloud applications. Computation and communication components can store data in the shared KVstore components, and retrieve them by a key. For example, the HTTPgw can use the KVstore to store HTTP messages that have a large body; cloudlets can store in a KVstore the results of their elaboration, in order to make them be accessible to the external interface.

The mOSAIC platform can be deployed on several nodes, on which its components are actually executed. Each node is usually a Virtual Machine (but it could also be a bare-metal machine) that runs a customized Unix-based operating system (mOS). Further nodes can be added at run-time; the software platform, which runs on top of the mOS, is in charge to discover new nodes, as well to perform synchronization and communication between them. The platform hides the actual node on which a component executes. The component allocation policies are configurable, and span from simple round-robin to resource available-based ones.

4 A Framework for Science Applications

The objective of the mOSAIC platform is to support applications with a statically-defined behavior, whose elaborations have to be repeated many times over different data sets. While this is the most common case for web applications, this pattern can also be applied to a wide range of scientific applications.

The solution we propose requires several assumptions that span the various steps of the scientific application life-cycle:

- **Development**: the developer of the application provides only the basic sequential code blocks implementing the chosen algorithm. He should not care about communication/synchronization details, but only take into account how data are organized and elaborated.
- **Deployment**: the developer/user should be able to start the application over the cloud, choosing the amount of resources to be used and possibly scaling dynamically them up/down at run time. Fault tolerance is guaranteed by the development framework, and is completely hidden at code level.
- **Execution**: the developer/user submits multiple jobs to the application, which performs always the same actions over different data.

In order to test how well scientific bag-of-tasks applications can run in a cloud exploiting the mOSAIC API, we have set up a framework implementing a very simple and fairly general solution pattern. This is based on the use of the most common split-work-merge paradigm. A problem to be solved is split in sub-problems (tasks), and handed out to task solvers (workers), whose partial results are finally merged. It is worth pointing out that the resulting workflow can be applied both to a bag-of-tasks pattern (which is the object of this paper) and to map-reduce [11] as well. In fact, both patterns involve a split-work-merge sequence. The difference is in the timing, as the workers in a bag-of-tasks are not constrained to proceed in lock-step, and can work on sub-jobs asynchronously among them.

4.1 Development of a bag-of-tasks application

In order to simplify the development of bag-of-tasks (BOT) cloud applications over mOSAIC, we have implemented a simple development framework providing all the needed mOSAIC components and an API that can be used to integrate the user-supplied application code. The mOSAIC platform is multi-language (it supports Java, Python, GO, NodeJS, Erlang). For the time being, the BOT framework API is provided only
Figure 2 Architecture of the BOT framework

for the Java language, as the majority of mOSAIC applications is currently written in Java. The relatively poor performance of Java code for scientific applications is not an issue, as in this context performance is dominated by cloud overhead. In any case, the support of additional languages is planned for the near future.

The architecture of the BOT framework is presented in Figure 2, which shows the mOSAIC components involved, communicating by means of queue and KVstores. In order to implement a BOT application, the developer has to supply the three classes Splitter, Worker and Merger, which implicitly define the format of the messages exchanged among components. Each supplied class should override two methods defined in the interfaces provided. These methods, called by the framework under specified conditions, are: (1) initialization ones, called once at the submission of every new job; (2) “core” ones, called repeatedly until the activity (i.e., splitting, working, merging) is not accomplished. The Splitter class “core” method (split) fetches input and produces, for each call, a Task object for the bag. For each Task, the work method of the Worker class performs the actual computation and creates a Result. The merge method of the Merger class is called every time a Result is available, to store it in a container holding partial results. Furthermore, after the last Result has been processed, finalize_merge method (belonging to the Merger class) performs the final merging of the partial results and stores the application output data.

The interaction between user-supplied code and the BOT framework is managed by orchestrator component. The orchestrator is responsible for (a) the instantiation of the user code; (b) the preparation of the runtime for the new job execution; (c) the delivery of the job descriptor to the splitter; (d) making the initialization calls.

During the job execution, the orchestrator can adjust the actual number of instances, automatically or on user request, by creating/destroying appropriate components. In fact, each worker component can be instantiated multiple times without affecting the overall application behavior. Each described component (Splitter, Merger, Worker, Orchestrator) is implemented through mOSAIC cloudlets, taking advantage of all the cloudlet features pointed out in section 3.

In practice, the user has to supply a jar archive containing his code (i.e., implemented classes, libraries), making it available on the network (i.e., published on a web server, with a known url). When the application is instantiated, the orchestrator retrieves the jar and embeds suitably the code into the mOSAIC components. Interaction of the user with the framework (e.g., job submission, monitoring, ...) is obtained through a REST/Web interface by exploiting an HTTPgw mOSAIC component.

The component interaction is presented graphically in the sequence diagram of Figure 3. The sequence of steps is the following:

- after the orchestrator receives the create_bag command, it fetches the user code and instantiates
one Splitter, one Merger and (a single or) multiple Workers;

- the start_job command leads to the execution, in parallel, of the initialization methods on each created instance;
- the Splitter starts to create new Tasks and puts them into the bag (i.e., into the Workers’ queue);
- each Worker fetches a Task from the bag (i.e., from the Workers’ queue), starts processing it and eventually sends back a Result to the Merger;
- the Merger fetches a Result, and performs its processing;
- when the last Result has been processed by the Merger, the finalize_merge method is called, to collect the outputs and to store suitably them.

Two main strategies have been implemented to achieve fault tolerance: (a) if a component crashes during a computation, the message with the object (i.e., JobDescriptor, Task, Result) it was working on will be automatically restored in the proper queue; (b) between two successive split (or merge) method call, the status is automatically check-pointed to avoid, in the case of faults, the restart of the activity from the beginning. However, it should be explicitly pointed out that fault tolerance, as well as communication, synchronization and elasticity, is provided by the framework and completely hidden from the user/developer view.

5 Experiments with the BOT framework: the binary knapsack problem

In order to perform performance tests on a real-world example, we have developed a BOT application to solve a classical combinatorial optimization problem: the binary knapsack problem (also known as the 0-1 knapsack problem). In its usual metaphoric formulation (see Figure 4), the problem consists of the selection of a set of items to fill a knapsack. Each item is characterized by a given value and weight. The knapsack can be loaded until a maximum weight is reached. The objective is to maximize the total value of the items in the knapsack. In the binary (or 0-1) formulation, an item can only be selected or discarded; it is not permitted to select an item more than one time, or to load the knapsack with

More formally, the problem is defined by:

- $N \in \mathbb{N}$: the given number of items;
- $v_i \in \mathbb{R}^+; \forall i \in [1, 2, ..., N]$; the value of each item;
- $w_i \in \mathbb{R}^+; \forall i \in [1, 2, ..., N]$; the weight of each item;
- $W \in \mathbb{R}^+$: the maximum weight the knapsack can hold.

The objective is

$$\max \sum_{i=1}^{N} x_i v_i ; \quad x_i \in \{0, 1\}. \quad (1)$$

subject to the constraint

$$\sum_{i=1}^{N} x_i w_i \leq W ; \quad x_i \in \{0, 1\}. \quad (2)$$

The binary vector $X (x_1, x_2, ..., x_N)$ represents a solution to the given problem, since it states if the $i$-th item has to be added to the knapsack ($x_i = 1$) or not ($x_i = 0$). A solution is considered valid only if the constraint in Equation 2 is respected, not valid otherwise.

As is well known [7], the problem can be solved by several different techniques, spanning from dynamic programming to classical branch and bound. In order to find a parallel solution based on the BOT pattern, a brute-force solution technique has been chosen. For this problem, the brute-force approach consists of generating all the possible solutions and (i) to discard the ones that are not valid as do not respect the constraint in Equation 2, (ii) to select the one that maximizes the objective function (Equation 1). Since an item can simply be put into the knapsack or not, the solution space can be viewed as all the possible configurations of the vector $X$. As every $x_i$ can have only two values (0 or 1), the solution space is $2^N$ wide. The brute-force approach is expensive, but fits perfectly a bag-of-tasks pattern, since it is possible to evaluate each candidate solution independently of the others.

The pseudo-code of the sequential algorithm is the following:

**Algorithm 1:**

1: function Knap($N, MAX_SIZE, W[N], V[N]$)
2: bestValue $\leftarrow 0$
3: solutions $\leftarrow$ POWER(2, $N$)
4: while solutions > 0 do
5:   validSol $\leftarrow$ TRUE
6:   value $\leftarrow 0$
7:   for $i = 1$ to $N$ do
8:     if $v_i > bestValue$ then
9:       bestValue $\leftarrow v_i$
10:      validSol $\leftarrow$ FALSE
11:     end if
12:   end for
13:   if validSol then
14:     return bestValue
15:   end if
16:   solutions $\leftarrow$ solutions - 1
17: end while
18: return bestValue.
measurements have been made varying the number of our tests using platforms with up to four nodes. The acquired VMs are shown in Table 1. We performed (AWS) infrastructure. The characteristics of the Machines (VM) leased from the Amazon Web Services mOSAIC platform has been deployed on Virtual a sequential C language code would be unfair). The are based on Java code, and so a comparison with 5.2 Experiments and performance results merge(...) is referred to with the term "chunk". The worker (see Appendix, Listing 2) takes a task from its queue, holding just the one with the higher value (see Figure 5). In order to tune the computation/communication ratio so as to reduce the communication overhead, each task is in charge of considering a configurable number of candidate solution vectors. In the following, this number is referred to with the term “chunk”.

The proposed code does not take explicitly into account any synchronization issue: it is just a slightly different version of the proposed Algorithm 1, distributing the work among the split(...), work(...) and merge(...) methods. Most of the newly added sections are only used to deal with “chunk” management.

5.2 Experiments and performance results

To evaluate the application performance, we have also developed a Java sequential version of the sequential algorithm to use as a reference (mOSAIC cloudlets are based on Java code, and so a comparison with a sequential C language code would be unfair). The mOSAIC platform has been deployed on Virtual Machines (VM) leased from the Amazon Web Services (AWS) infrastructure. The characteristics of the acquired VMs are shown in Table 1. We performed our tests using platforms with up to four nodes. The measurements have been made varying the number of Worker instances, in a range from 1 to 8. The experiment has been performed using a knapsack of 27 items, and submitting one job per each measurement (see details in Table 2). The measured execution times can be found in Table 3.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance type</td>
<td>c1.medium</td>
</tr>
<tr>
<td>vCPU</td>
<td>2</td>
</tr>
<tr>
<td>RAM</td>
<td>1.7 GB</td>
</tr>
<tr>
<td>storage</td>
<td>350 GB</td>
</tr>
<tr>
<td>network performance</td>
<td>moderate</td>
</tr>
</tbody>
</table>

Table 1: AWS VM instance details

In order to evaluate the obtained performance, speedup \( (S(n) = \frac{T_1}{T_n}) \) and efficiency \( (E(n) = \frac{S(n)}{n}) \) have been calculated. The speedup has been obtained using as sequential time the Java sequential version of the algorithm \( (T_S = 542 \text{ s}) \). The execution time of the Java sequential solution has been measured on an AWS Virtual Machine with the characteristics in Table 1 and the workload in Table 2. The efficiency has been computed using the number of Worker instances as the \( n \) parameter. The related plots are shown in Figure 6 and 7. The speedup obtained is coherent with the number of available vCPUs per platform (remember that, according to Table 1, there are two vCPU per node) showing horizontal asymptotes near the values 2, 4 and 8. The efficiency curves show a super-linear behavior for the lower values of \( n \); this behavior can be explained considering that the Splitter and Merger work in parallel, while we use as values for the \( n \) parameter the number of Worker instances. For instance, even with only one Worker instance \( (n = 1) \) we have a parallel degree of 3.

The proposed measurements show that the overheads introduced by the overall framework are acceptable, as they allow to obtain a linear performance gain as the number of the employed resources rises.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>knapsack items (N)</td>
<td>27</td>
</tr>
<tr>
<td>submitted jobs</td>
<td>1</td>
</tr>
<tr>
<td>chunk</td>
<td>100,000</td>
</tr>
<tr>
<td>solutions</td>
<td>134,217,728</td>
</tr>
<tr>
<td>generated tasks</td>
<td>1,343</td>
</tr>
</tbody>
</table>

Table 2: Experiment details

In order to take into account the actual penalties introduced by the Java language and the overall framework, a C-MPI version of the algorithm has been developed. The developed C-MPI code is much longer and more complex than the one shown in the Appendix. We had to take explicitly into account the Workers management, in order to send each task only to the not-busy ones, mimicking the behavior of a BOT pattern.

In this case, the speedup has been calculated using, as the sequential time, a C sequential version of
Figure 5  The Splitter adds a Task with a chunk 4 to the bag. The Worker discards 2 solutions and sends the Result to the Merger.

Table 3  BOT framework execution times (s)

<table>
<thead>
<tr>
<th>VMs / Workers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 node</td>
<td>570</td>
<td>207</td>
<td>210</td>
<td>231</td>
<td>242</td>
<td>245</td>
<td>251</td>
<td>245</td>
</tr>
<tr>
<td>2 nodes</td>
<td>407</td>
<td>236</td>
<td>149</td>
<td>152</td>
<td>121</td>
<td>121</td>
<td>122</td>
<td>121</td>
</tr>
<tr>
<td>4 nodes</td>
<td>409</td>
<td>224</td>
<td>135</td>
<td>98</td>
<td>79</td>
<td>70</td>
<td>70</td>
<td>70</td>
</tr>
</tbody>
</table>

the presented sequential algorithm ($T_s = 78\, s$). The execution time of the C sequential solution has been measured on an Amazon Virtual Machine with the characteristics in Table 1 and the workload in Table 2. The execution time measurements are presented in Tab. 4. The efficiency has been calculated using the number of Worker instances as the $n$ parameter. It should be explicitly pointed out that in any case there is one additional process running (the coordinator). The related plots are shown in Figure 8 and 9. The sequential C version is about 7 times faster than the Java sequential one, while, as shown in Table 5, the MPI parallel runs are faster, on the average, just by a factor of 5 than the BOT framework ones. As can be observed on the presented diagrams, the curve trends are similar for the mOSAIC and BOT versions. This means that the relative performance gains are comparable. It is possible to conclude that the higher overhead of the mOSAIC solution is mainly imputable to the inefficiency of the Java language, and not to the proposed framework.

6 Conclusions and Future Work

The aim of the work described in this paper was to understand the issues linked to the development of a scientific application on the top of a cloud.
Figure 7  Efficiency of the mOSAIC BOT solution

Figure 8  Speedup of the MPI BOT solution (reference: $T_r = 78$ s)

<table>
<thead>
<tr>
<th>VMs / Workers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 host</td>
<td>88</td>
<td>61</td>
<td>54</td>
<td>51</td>
<td>49</td>
<td>48</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>2 hosts</td>
<td>81</td>
<td>41</td>
<td>27</td>
<td>32</td>
<td>27</td>
<td>24</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>4 hosts</td>
<td>81</td>
<td>42</td>
<td>28</td>
<td>22</td>
<td>18</td>
<td>16</td>
<td>14</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 4  MPI execution times (s)

<table>
<thead>
<tr>
<th>VMs / Workers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 host/node</td>
<td>4.20</td>
<td>3.39</td>
<td>3.89</td>
<td>4.53</td>
<td>4.94</td>
<td>5.10</td>
<td>5.23</td>
<td>5.10</td>
</tr>
<tr>
<td>2 hosts/nodes</td>
<td>5.02</td>
<td>5.76</td>
<td>5.52</td>
<td>4.75</td>
<td>4.48</td>
<td>4.32</td>
<td>5.08</td>
<td>4.65</td>
</tr>
<tr>
<td>4 hosts/nodes</td>
<td>5.05</td>
<td>5.33</td>
<td>4.82</td>
<td>4.45</td>
<td>4.39</td>
<td>4.38</td>
<td>5.00</td>
<td>4.12</td>
</tr>
</tbody>
</table>

Table 5  Ratio between the execution times of mOSAIC and MPI solutions
platform. We have developed a simple development framework, currently specialized for the bag-of-task paradigm, that exploits the API and the components provided by the mOSAIC platform. This makes our contribution different from other work implementing BOT frameworks on top of a vendor-locked native cloud programming interface. The resulting solution makes it fairly easy for a scientist to develop and to run applications on the cloud with scalability and fault-tolerance provided out-of-the box by the programming framework.

We have also presented performance tests on a combinatorial optimization application to evaluate the overhead introduced by the platform and by the cloud environment, and to assess the scalability of the developed applications. The presented results show satisfactory performance figures, in that the overhead is tolerable for long-running jobs and performance scales linearly, at least in the ranges of VM values considered. Once again, it should be pointed out that communication, synchronization, scalability and fault-tolerance are provided by the framework and by the underlying platform, so they do not present an issue to the developer.

Our future research work will focus on the extensive performance testing of the framework, by measurements on real-world scientific codes running in private and commercial cloud environments. We also plan to implement alternative frameworks for additional programming paradigms.

Acknowledgment

The authors would like to thank Ciprian Craciun, Dana Petcu and the IeAT Team for their support, and Luigi Pino who contributed to the development of the MPI example code. This research is partially supported by HOST Project FP7-REGPOT-CT-2011-284595-HOST.
References


[8] Creative Commons. Creative Commons License. http://creativecommons.org/licenses/by-sa/2.5.


Listing 1  Knapsack splitter

```java
public class KnapsackSplitter implements Splitter<String, KnapsackTask, SplitContext> {
    @Override
    public SplitContext initSplit (Properties conf, String input, SplitterUtils utils) throws Exception {
        KnapDescriptor descr = new KnapDescriptor(input);
        SplitContext context = new SplitContext();
        context.iterations= descr.items * descr.items;
        context.descriptor = descr;
        utils.setGlobalContext("descr", descr);
        return context;
    }

    @Override
    public KnapsackTask split(SplitContext context) {
        KnapsackTask task = new KnapsackTask();
        if (context.iterations > 0 ){
            long range = context.iterations - context.descriptor.workerChunk;
            if (range >=0){
                context.iterations = range;
                task.from = range;
            } else {
                task.from = 0;
                context.iterations = 0;
            }
        } else {
            //end of the split
            return null;
        }
        return task;
    }
}
```

Listing 2  Knapsack worker code

```java
public class KnapsackWorker implements Worker<KnapsackTask, KnapsackResult, KnapDescriptor> {
    @Override
    public KnapDescriptor initWorker(Properties conf, WorkerUtils utils) throws Exception {
        return (KnapDescriptor)utils.getGlobalContext("descr");
    }

    @Override
    public KnapsackResult work(KnapDescriptor context, KnapsackTask payload) {
        KnapsackResult validSolutions = new KnapsackResult();
        for (long j = payload.from ; j < payload.to; j++){
            boolean solutionValid = true;
            KnapSolution solution = new KnapSolution();
            solution.items = createSolution(j,context.items);
            for (int k = 0 ; k < context.items ; k++){
                if (solution.items[k]==1){
                    solution.totalValue += context.values[k];
                    solution.totalWeight += context.weights[k];
                    if (solution.totalWeight > context.knapSize){
                        solutionValid = false;
                        break;
                    }
                }
            }
            if (solutionValid){
                validSolutions.sols.add(solution);
            }
        }
        return validSolutions;
    }

    private int [] createSolution ( long solution, int items) {
        //...implementation follows...
    }
```
Listing 3  Knapsack merger code

```java
public class Knapsack01_merger implements Merger<String, KnapsackResult, KnapSolution> {
    @Override
    public KnapSolution initMerge(Properties conf) throws Exception {
        KnapSolution ks = new KnapSolution();
        return ks;
    }

    @Override
    public void merge(KnapSolution context, KnapsackResult validSolutions) {
        for (int i = 0; i < validSolutions.sols.size(); i++) {
            if (context.totalValue <= validSolutions.sols.get(i).totalValue) {
                context.items = validSolutions.sols.get(i).items;
                context.totalValue = validSolutions.sols.get(i).totalValue;
                context.totalWeight = validSolutions.sols.get(i).totalWeight;
            }
        }
    }

    @Override
    public String endMerge(KnapSolution context) {
        return "Total weight: " + context.totalWeight + "\nTotal value: " + context.totalValue;
    }
}
```