Cloud-aware Development of Scientific Applications

Alessandra De Benedictis
Università di Napoli Federico II, DIETI,
Napoli, Italy
alessandra.debenedictis@unina.it

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

Mauro Turtur, Umberto Villano
Università del Sannio, DING,
Benevento, Italy
{mauro.turtur, 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 economic 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

I. 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 [1] 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 obsolescing 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 [4], Dryad [5]);
- the use of native cloud APIs and programming artifacts: applications are written by exploiting vendor or cloud-neutral API and resources [6], [7];
- the use of a cloud-enabled programming platform: cloud-aware application are developed on the top of
a cloud-aware programming framework [8], [9].

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 [10]. 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 this paper, we introduce a light-weight framework for the development of cloud bag-of-task applications. 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 [11]. The solution proposed here is different, as it relies on a vendor-neutral programming platform (the mentioned mOSAIC [8]), 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 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 III deals with the mOSAIC programming platform. Then Section IV presents the rationale and the architecture of the framework we have implemented for the development of bag-of-tasks applications in the cloud. A preliminary performance analysis, involving overhead and scalability assessment of a synthetic application, is proposed in Section V. The paper closes with our conclusions and plans for future research.

II. 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 economicity and to on-demand provision is discussed in [12]. The performance disadvantages of clouds for scientific computing workloads are presented in [13].

In [14], 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 [11], 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 [15].

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 [16], the Authors present Cloudine, a platform for the development of generic scientific applications able to exploit at best cloud elasticity. The paper [17] 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 [18], where CMPI, a novel MPI library based on the cloud-oriented optimization proposed in [19], is presented.

III. 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 multiple cloud providers [8]. 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
Queue-based programming. The default configuration allows applications to send messages through it. All applications can communicate with each other after the queue has been created. A queue component offers an API to create message queues, which can be used by applications to communicate with each other. When a message is published, the registered consumers receive 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 applications 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 one offered by Azure Queue storage service, which uses explicit timeouts to assume a consumer dead [11].

The KVstore component is based on Riak [21]. 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 elaborations, in order to make them accessible to the external interface.

IV. 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. They 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 [4] 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.

A. 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 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 on top of mOSAIC cloudlets, taking advantages of all the cloudlet features pointed out in section III.

In practice, the user has to supply a jar 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.

Two main strategies have been implemented to achieve fault tolerance: (a) if a component crashes during a compu-
tation, 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 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.

V. ANALYSIS OF THE FRAMEWORK OVERHEAD AND SCALABILITY

The goal of this paper is not to provide ultimate performance measurements for real-world applications, but just to evaluate the feasibility of the approach proposed, i.e., the development of scientific code (for the time being, based on the bag-of-tasks pattern) on the top of a cloud-aware programming framework. To achieve this we will limit ourselves to perform a preliminary set of measurements to assess the overhead introduced by the cloud/mOSAIC execution environment.

For our tests, we have developed a skeletal BOT application, where the work method in the Worker class is able to process a given parameterized load, expressed in MFLOPS and obtained as a parameter from the task description. The Splitter and Merger do not actually perform splitting/merging, but just activate workers and collect response times, respectively.

We submit to the above application a stream of equal jobs, which are splitted into the same number of tasks for the workers to process. We submit the jobs at fixed rate, measuring the completion time for the whole set of jobs submitted. This experiment is representative of an application processing a set of consecutive jobs, each involving equal processing load for the system.

The workload submitted to the application is described by the following parameters:

- **number of jobs**: total number of jobs submitted;
- **period**: time between two successive job submissions;
- **tasks**: number of tasks each job is splitted into;
- **workload**: the computational load (in MFLOPS) for each job.

The two workloads used in our tests are shown in Tables I and II. The first is representative of an almost null load for the workers, and is useful to find out the overhead introduced (the overhead dwarfs “real” work). The second is a more realistic example where the workers perform some processing. A cluster of seven physical nodes running Eucalyptus has been used for our measurements. Each node was equipped with: 2x Intel Xeon quad-core 2.00 Ghz, 10 GB RAM, 200 GB of local storage. mOSAIC instances with 2 and 4 virtual CPUs have been started on two virtual machines on the top of Eucalyptus.

As a first test, we started the full application with 1 to 5 workers under workload I. In this scenario, each worker just starts and terminates almost immediately. Hence the completion time is a rough but realistic measurement of the cloud-based overhead. The measured completion time is about 30s, independently of the number of workers, and does not change significantly as the number of virtual CPUs (VCs) is varied. In other words, it is mainly a fixed overhead due to the orchestration and the execution of the components making up the BOT framework.

Then we executed the application submitting the workload II, varying again the number of workers and of VCs assigned to the system. As shown in Table III, the application scales in an acceptable way (i.e., completion times decrease as the number of workers is increased) until the number of workers is equal to the number of VCs available. Higher number of workers are inevitably allocated to the same VCs and hence there is no beneficial effect on response times. In fact, for 4 VCs the performance saturation is reached with just 3 workers. This is not strange, because the overhead has reached the absolute minimum (the 30s due to overhead).

Even if the figures reported above have been obtained through a limited set of tests, we think that the result is encouraging. An almost fixed overhead of 30s is acceptable for long-running jobs, and these are the target of our framework implementation. Potential users will never find in the cloud response times comparable to those of an HPC system, but low-cost processing power with the possibility to avoid any investment in hardware. Our framework will provide automatically the possibility to use variable amount of resources with reasonable scalability of response times, and fault-tolerance without any programming hassle.

VI. 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 platform. We have devel-

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of jobs</td>
<td>10</td>
<td>number of jobs</td>
<td>10</td>
</tr>
<tr>
<td>period</td>
<td>1s</td>
<td>period</td>
<td>1s</td>
</tr>
<tr>
<td>tasks</td>
<td>30</td>
<td>tasks</td>
<td>30</td>
</tr>
<tr>
<td>workload</td>
<td>1 MFLOP</td>
<td>workload</td>
<td>1000 MFLOPS</td>
</tr>
</tbody>
</table>

Table I
WORKLOAD I

<table>
<thead>
<tr>
<th>Number of Workers</th>
<th>2 VCs</th>
<th>4 VCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>49</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>47</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>48</td>
<td>30</td>
</tr>
</tbody>
</table>

Table II
WORKLOAD II

<table>
<thead>
<tr>
<th>Number of Workers</th>
<th>2 VCs</th>
<th>4 VCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>47</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>48</td>
<td>30</td>
</tr>
</tbody>
</table>

Table III
COMPLETION TIMES FOR WORKLOAD 2 (s)
oped 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 run applications on the cloud with scalability and fault-tolerance provided out-of-the box by the programming framework.

We have also presented preliminary performance tests 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 can be tolerable for long-running jobs and performance scales linearly, at least in the ranges of VC values examined. 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. This research is partially supported by HOST Project FP7-REGPOT-CT-2011-284595-HOST.

REFERENCES


