Disclaimer

This copy is a preprint of the article self-produced by the authors for personal archiviation. Use of this material is subject to the following copyright notice.

IEEE Copyright notice

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.
Early Prediction of the Cost of HPC Application Execution in the Cloud

Massimiliano Rak
Department of Industrial and Information Engineering
Second University of Naples, Aversa, Italia
massiliano.rak@unina2.it

Mauro Turtur, Umberto Villano
Department of Engineering
University of Sannio, Benevento, Italia
mauro.turtur@unisannio.it, villano@unisannio.it

Abstract—Even if clouds 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 both simple development environments, able to exploit cloud scalability, and the capability to easily predict the cost of HPC application runs.

This paper presents a framework built on the top of a cloud-aware programming platform (mOSAIC) for the development of bag-of-tasks scientific applications. The framework integrates a cloud-based simulation environment able to predict the behavior of the developed applications. Simulations enable the developer to predict at an early development stage performance and cloud resource usage, and so the infrastructure lease cost on a public cloud.

The paper sketches the framework organization and discusses the approach followed for application development. Moreover, some validation tests of prediction results are presented.

I. INTRODUCTION

At least in theory, clouds could be profitably used to bring the power of economic and scalable parallel computing to the masses. The main obstacles to this process are the substantial differences between the “traditional” and the cloud-based paradigm, and the lack of adequate development tools to support the porting of legacy application to the cloud.

Furthermore, 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 [1]. 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 an almost ideal candidate for execution on the cloud. The major advantage is economic: the cost for leasing 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 provider offerings often allows to acquire the computing resources needed at very low cost (see for example the EC2 Spot Instances offer [2]). This enables any organization to run parallel code whenever needed, at a low cost, without investing the capital in rapidly obsolescing parallel hardware. The second important issue is 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.

We think that the final step to make clouds fully advantageous for sporadic scientific users is providing simple tools to predict the performance behavior of their application, allowing them to make a tradeoff between performance and leasing costs. In a previous paper [3] we proposed the use of a cloud-enabled programming platform. This platform makes it possible to develop cloud applications on the top of a cloud-aware programming framework (mOSAIC [4], [5]) by exploiting the bag-of-tasks programming paradigm. The bag-of-tasks (BOT) paradigm, also known as master-worker, processor farm, ..., is widely understood, and ubiquitous in small and medium-scale scientific computing.

Moreover, in the past we worked on the performance prediction of cloud applications developed on the top of the mOSAIC framework [6]. Already-available tools enable us to predict the performance of a cloud application without running it on (payed) cloud resources. In this paper, we discuss the enrichment of the bag-of-task framework with performance prediction capabilities, allowing the automatic generation of the application simulation models.

The remainder of this paper is structured as follows. In the next section we will examine related work. Section III illustrates the rationale and the architecture of the framework we have implemented for the development of bag-of-tasks applications in the cloud. Section IV presents our approach to early performance prediction and Section V shows some of our validation tests. The paper closes with our conclusions and plans for future research.
II. RELATED WORK

A. HPC in the Cloud

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 economic to and on-demand provision is discussed in [7]. The performance disadvantages of clouds for scientific computing workloads are presented in [8].

In [9], 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 implementation of BOT similar to the one presented in this paper, although with a few significant differences, is presented in [10]. A Java framework for the development of fault-tolerant applications is proposed in [11].

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 Cloudine, a platform for the development of generic scientific applications able to exploit at best cloud elasticity. The paper [13] 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 [14], where CMPI, a novel MPI library based on the cloud-oriented optimization proposed in [15], is presented.

B. Simulation and Performance Prediction

The core of our proposal is the use of a simulation-based approach for application performance prediction. The use of simulation in the HPC context is widely discussed in a number of papers, as [16], [17]. Recent efforts in this field are documented in [18] and [19].

For Component-Based Software Systems (CBSS), most approaches available in the literature focus on integrating performance prediction and evaluation techniques at design time, i.e., when the system implementation is not available. An exception, which has some similarities with our approach, is the COMPAS system [20]. The paper [21] presents a complete survey of performance strategies for CBSS.

CloudCMP [22] is one of few examples of simulation-based performance predictions of cloud applications, where the goal is to compare different cloud providers and to select a suitable one. Another notable work in this area is CloudSim [23], which targets the simulation of the entire stack of software/hardware components in a cloud infrastructure.

III. THE FRAMEWORK FOR SCIENCE APPLICATIONS

The solution we proposed in [3] for BOT cloud application development 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 job to the application, which performs always the same actions over different data.

Our BOT development framework implements the simple and common split-work-merge solution pattern. A problem to be solved is split in sub-problems (tasks), and handed out to task solvers (workers), whose partial results are finally merged (see Figure 1). The resulting workflow could be applied to 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. Bag-of-tasks applications can be developed by extending the above described components with problem-specific algorithms. The details of the development framework are presented in [3].

The BOT development framework provides all the needed components (splitter, merger, worker and orchestrator) and an API that can be used to integrate the user-supplied application code. In fact, all the supplied components are mOSAIC components. 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 [4]. The mOSAIC platform offers an easy way to package and to deploy automatically in the cloud software components. Through the platform interface it is possible to deploy multiple instances of the same component and to restart them, in the case of a failure.

mOSAIC offers a set of already-developed basic components, which can be easily extended by the developer (as we have done in the case of the BOT framework). Among the standard components, Queues and KVstores play the most important roles.

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.

The KVstore component is based on Riak [26]. 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; “computing” mOSAIC components (cloudlets) can store in a KVstore the results of their elaboration, in order to make them be accessible to the external interface.
IV. PERFORMANCE PREDICTION OF BAG-OF-TASKS APPLICATIONS

In the previous section we have described our bag-of-tasks framework for the development of scientific applications in the cloud. A key point is that such applications are fully defined by the number of instances of the mOSAIC components mentioned above and by their interconnections. Moving from these premises, we have devised a performance model of the application that can be used to predict its behavior and to tune its performance.

Our performance model is process-based [27], [28], [29], in that it is described through a set of discrete-event simulation components whose temporal behavior is described as a process. Event management and discrete-event actions/reactions are modeled in terms of process synchronization primitives. The simulated components have been developed by exploiting the JADES simulation library [29], which allows the description of process-oriented simulations in Java.

A noteworthy feature of the solution devised is that the simulation models expressed through the JADES library can be easily evaluated through the mJADES platform [30]. mJADES is a recently-developed system that supports the distribution of multiple JADES simulations on cloud resources. The mJADES simulation system is based on a Java-based modular architecture. The mJADES simulation manager produces simulation tasks from simulation jobs, and schedules them to be executed concurrently on multiple instances of the simulation core. This is a process-oriented discrete-event simulation engine based on the JADES simulation library, which allows the description of process-oriented simulations in Java.

A. Bag-of-tasks Simulation Model

To model a bag-of-tasks application, we developed a simulation component for each of the core components making up the bag-of-task framework (Splitter, Worker and Merger), and for each component devoted to manage the cloud application execution (HTTPgw, Orchestrator), communication and storage (Queue and KVstore).

The management components of the bag-of-task framework (HTTPgw and Orchestrator) have the role of managing the cloud application execution, offering an interface to end users (HTTPgw) and orchestrating the execution, forwarding the messages to the right splitter when multiple bag-of-tasks are executed on the same resources and starting/stopping the triple Splitter-Worker-Merger.

Our simulation model is driven by a workload generator, which is in charge to generate the sequence of requests the users issue in time. At the start of simulation, it begins sending out the messages to the Orchestrator according to the chosen workload, described in a configuration file. Currently two simple workload models are available: a set of requests with fixed inter-arrival time, and a Poisson arrival model that generates messages with random exponential inter-arrival time. Moreover, it is also possible to start a multiple number of concurrent workloads miming the load generated by multiple users.

The Orchestrator model is fairly straightforward: it continuously receives Jobs from a queue (that mimics a HTTP channel) and forwards them to the Splitter. At the state of the art, our model is very simple, since it is based on the assumption of a fixed application configuration, which cannot be dynamically altered (i.e., we cannot start a new set of Split-Work-Merge components during the simulation). So the Orchestrator has just the role of routing the message to the Splitter instances. We aim at improving the Orchestrator model in the future.

The Splitter, Merger and Worker models behave in a similar way, receiving and forwarding the messages from/to the internal queues accordingly to the described bag-of-tasks pattern. The Merger process, after collecting all the intermediate job Result messages, sends a job Result message to a special simulator component, the report generator, which gathers the results and produces the final simulation reports.

The computational resources consumed by our core components (i.e., Orchestrator, Splitter, Merger and Worker processes) are taken into account by means of a component simulating CPU resource sharing. At simulation
Listing 1. The Queue Simulation Process

```java
public class Queue extends it.unisannio.ing.perflab.jades.core.Process {
    private Mailbox inputMailbox;
    private Mailbox outputMailbox;
    public Queue(String name, double beta0, double beta1, double beta2) {
        super(name);
        inputMailbox = new Mailbox(name + "inputMailbox");
        outputMailbox = new Mailbox(name + "outputMailbox");
    }
    public void send(Object m) {
        inputMailbox.send(m);
    }
    public void run() {
        while (true) {
            int msgSize = (Integer) inputMailbox.receive();
            int msgInQueue = inputMailbox.msg_in_queue() + 1;
            hold(beta2 * msgSize + beta1 * msgInQueue + beta0);
            outputMailbox.send(msgSize);
        }
    }
    public Object receive() {
        return outputMailbox.receive();
    }
}
```

start-up it is necessary to provide the actual number of available virtual CPUs (vCPUs) and the allocation to vCPUs of the framework components involved in a run.

The response time of the queues (i.e., the time needed to notify a message to a process, once it has been published on the queue) is modeled as a function of the number of queued messages and of the dimension of the messages (according to the iLDS model [31]). Listing 1 shows the code of the process simulating the behavior of the communication queues.

Figure 2 sketches the proposed simulation model. It should be noted how close it resembles the structure of the bag-of-tasks application.

As previously pointed out, even if the framework actual behavior strictly depends on the specific algorithm to be implemented, in any case the core bag-of-tasks components receive messages from queues and forward them to other queues, consuming a suitable amount of CPU time. Starting from this consideration, we can fully describe a bag-of-tasks instance by means of two sets of parameters.

The application instance parameters represent the values under the control of the framework user. They describe the application and the BOT framework configuration to be simulated, as follows:

- **send job frequency** (SEND_FREQ): job send rate;
- **tasks** (TASKS): number of tasks generated by the Splitter;
- **worker overhead** (WORK_TIME): estimate of the vCPU time required by a Worker to process a Task;
- **allocation map**: allocation matrix describing the allocation of the framework processes to the available vCPUs.

The framework tuning parameters are used to model a specific framework instance, taking into account the overhead introduced by the framework itself, by the platform and by any underlying software layer. After they have been estimated for a framework instance, they will not vary between different simulation runs. In the following section we will describe a methodology for the evaluation of such values. The parameters are:

- **HTTP overhead** (HTTP_OH): the communication delay introduced by a HTTP communication channel;
- **Orchestrator overhead** (ORCH_OH): the orchestrator overhead (vCPU time) introduced to process each submitted job and to forward the descriptor to the Splitter;
- **Splitter overhead** (SPLIT_OH): overhead introduced (vCPU time) to execute a single split operation, to create and to forward a Task;
- **Merger overhead** (MERGE_OH): overhead (vCPU time) introduced to execute a single merge operation;
- **Job Descriptor message** (JOB_MSG_SIZE): size of the Job Descriptor messages;
made out of the following steps:

- **Result message** (RESULT\_MSG\_SIZE): size of the Result messages;
- **Task message** (TASK\_MSG\_SIZE): size of the Task messages;
- **HTTP channel**: \(\beta_0, \beta_1, \beta_2\) parameters (see Listing 1) to model the HTTP channel;
- **bag-of-tasks queues**: \(\beta_0, \beta_1, \beta_2\) parameters to model the framework internal queues.

### C. Early Prediction: Bag-of-tasks Development Methodology

Program performance simulation is an interesting matter, but it becomes a fundamental technique if it can be adopted at the very early development stages, before the complete development and deployment of the actual application. We believe that performance prediction can be integrated in the BOT application life cycle, adopting a development methodology made out of the following steps:

1. **Identify the Split/Work/Merge Algorithms**: the developer has to rethink of its algorithm in order to identify the role of the core bag-of-tasks components. This is not a complex task, because the BOT paradigm is usually close to the behavior of most common applications. In this phase the developer starts writing and rethinking its own code.

2. **Prepare the set of data to be compared**: concurrently with algorithm development, the developer identifies the way in which the application will be used in the future, i.e., how many requests will be served and how the work will be split among workers. This activity can be conducted concurrently with development, so that it is possible to adapt splitting and merging algorithms in order to reduce the execution costs in future, on the basis of the simulation predictions.

3. **Execute the benchmarks and collect the results**: the benchmark applications are launched in the target mOSAIC cloud, obtaining performance figures for every component on the actual execution platform. It should be noted that, if the application is not fully developed, the developer will only run the benchmarks for the core components. The CPU time requirements of not fully-developed code can be estimated by means of static software analysis. Of course, this is likely to affect adversely the prediction accuracy.

4. **Obtain a performance prediction by executing the simulation model**: the simulation model can be executed with multiple synthetic workloads, using the parameters estimated as described in the previous steps, obtaining performance predictions for different scenarios of interest.

Following such approach, the developer is able to predict the performance and resource usage (and related costs) from the early development stages of his cloud application.

### V. RESULTS

The goal of this paper is not to provide ultimate performance measurements for real-world applications. Our objective is just to evaluate the feasibility of the proposed approach, i.e., the development of scientific code on the top of a cloud-
Fig. 2. Bag-Of-Tasks simulation model

aware programming framework, exploiting early performance prediction techniques for making cost/performance trade-offs.

As outlined in the previous sections, the framework we propose is composed of (i) the cloud BOT development framework, which enables the developer to run his applications on cloud environment, (ii) the simulation framework, which enables the developer to predict application performance even in the early development stages and (iii) the benchmark applications, which are used to evaluate the timing parameters for simulation.

In order to validate the approach, we run benchmark application on the target VMs to gather the timing parameters for the simulation environment, using a minimal set of resources. Then we compared a real application with its simulation, varying both the workload and the amount of resources assigned to the application run.

A. Measurement of Timing Parameters

Our BOT framework makes it possible to obtain applications that are completely independent of the real environment on which it will run. The number and the characteristics of cloud resources leased for execution only affect performance. We capture the characteristics of the actual execution environment through a set of benchmarks, whose results produce the timing parameters successively used for simulation.

For our tests, we have deployed the mOSAIC platform on Virtual Machines (VM) leased from the Amazon Web Services (AWS) infrastructure. The characteristics of the acquired VMs are shown in Table I. Running our benchmark suite, we obtained the values in Table II.

<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 I. AWS VM INSTANCE DETAILS

B. Simulation Validation Varying the Workload

We have developed a skeletal BOT application, where the work method in the Worker class is able to process a given 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 submitted the same workload both to the skeletal application running in the real execution environment and to the simulator, comparing the measured and predicted completion times. For our tests, we used the workloads briefly described in Table III, which are representative of light (WORK_TIME=15 ms) and medium-heavy (WORK_TIME=200 ms) load for the workers. To vary dynamically the number of tasks, we used a pseudo-random uniform distribution.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>vCPU</td>
<td>2</td>
</tr>
<tr>
<td>JOBS</td>
<td>30</td>
</tr>
<tr>
<td>SEND_FREQ</td>
<td>1 s</td>
</tr>
<tr>
<td>TASKS</td>
<td>uniform(100,400)</td>
</tr>
<tr>
<td>WORK_TIME</td>
<td>15 ms; 200 ms</td>
</tr>
<tr>
<td>WORKERS</td>
<td>4</td>
</tr>
<tr>
<td>SPLITTERS</td>
<td>2</td>
</tr>
<tr>
<td>TASK_MSG_SIZE</td>
<td>400 B</td>
</tr>
<tr>
<td>JOB_MSG_SIZE</td>
<td>20 B</td>
</tr>
<tr>
<td>RESULT_MSG_SIZE</td>
<td>20 B</td>
</tr>
</tbody>
</table>

TABLE II. TESTBED TIMING PARAMETERS

Figure 4 shows on the x-axis the job number (30 jobs are submitted, according to Table III) and on the y-axis its completion time. The completion time associated to job #30 is the total completion time for the whole burst of 30 jobs. The simulation results are summarized in Table IV, where is also reported the estimated vCPU usage.

Summarizing the results shown in Figure 4, for Test 1 (WORK_TIME=15 ms) we measured a completion time of
512 ms versus a predicted one of 551 ms, with a relative error of 7.61% (considering also intermediate jobs completion times, the error ranges from a minimum of 0.77% to a maximum of 28.57%). For Test 2 (WORK\_TIME=200 ms) we measured a completion time of 1383 ms against a predicted of 1134 ms, with a relative error of about 18% (considering intermediate jobs, the error ranges from 4.37% to 18.62%).

In both cases simulation offers good predictive capacities. Moreover, even at this stage (no actual application developed) it offers interesting information about resource usage. The vCPU usage in Test 1 (the one with lower WORK\_TIME) is very low, as most of execution time is spent in communications. This is a relevant hint for the developer of real code, who can tune its implementation so as to obtain a coarser task granularity and hence a more efficient balance of computing and communication.

### C. Simulation Validation Varying the Resources Used

In the following we compare the performance of the skeletal application executed on real resources to the one obtained by simulation, using for the real and the simulated run a variable number of workers. A higher number of vCPUs is clearly necessary for these tests; leasing additional vCPUs on AWS of the same type used in the previous test, we can use once again the timing parameters in Table II. Only the allocation matrix in the simulator configuration file has to be updated.

We submitted the workload described in Table V to the synthetic application, varying the number of workers and comparing it to the real measurements on the skeletal application. The test outcome matches the results proposed in [3], where we evaluated the framework overhead.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4</td>
</tr>
<tr>
<td>JOBS</td>
<td>10</td>
</tr>
<tr>
<td>SEND_FREQ</td>
<td>1 s</td>
</tr>
<tr>
<td>TASKS</td>
<td>30</td>
</tr>
<tr>
<td>WORK_TIME</td>
<td>100 ms</td>
</tr>
<tr>
<td>WORKERS</td>
<td>1 to 5</td>
</tr>
<tr>
<td>SPLITTERS</td>
<td>1</td>
</tr>
<tr>
<td>TASK_MSG_SIZE</td>
<td>20 B</td>
</tr>
<tr>
<td>JOB_MSG_SIZE</td>
<td>20 B</td>
</tr>
<tr>
<td>RESULT_MSG_SIZE</td>
<td>20 B</td>
</tr>
</tbody>
</table>

### V. Conclusions and Future Work

The aim of the work described in this paper is to propose an approach that integrates the development of scientific application on top of a cloud platform and its performance prediction through a dedicated simulation environment.

To obtain this integration, on one hand we have built a development framework, currently specialized for the bag-of-task paradigm, which exploits the API and the components
provided by the mOSAIC platform. On the other, we have developed a set of simulation components for JADES. These simulation components correspond one-to-one to the mOSAIC components for the BOT framework. Besides presenting the approach, we have discussed the outcome of our preliminary performance tests used to evaluate the simulation accuracy.

In our tests, we offered some examples of usage under different workloads and using different amount of resources, in order to show how it will be possible for a developer to predict the behavior of an application, without running it on (paid) cloud resources, but simply using the associated simulation environment.

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

REFERENCES


