Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–17
© The Author(s) 2015
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342015594678
hpc.sagepub.com
SAGE

PyCOMPSs: Parallel computational workflows in Python

Enric Tejedor[1], Yolanda Becerra[1], Guillem Alomar[1], Anna Queralt[1], Rosa M Badia[1,2], Jordi Torres[1], Toni Cortes[1] and Jesús Labarta[1]

Abstract

The use of the Python programming language for scientific computing has been gaining momentum in the last years. The fact that it is compact and readable and its complete set of scientific libraries are two important characteristics that favour its adoption. Nevertheless, Python still lacks a solution for easily parallelizing generic scripts on distributed infrastructures, since the current alternatives mostly require the use of APIs for message passing or are restricted to embarrassingly parallel computations. In that sense, this paper presents PyCOMPSs, a framework that facilitates the development of parallel computational workflows in Python. In this approach, the user programs her script in a sequential fashion and decorates the functions to be run as asynchronous parallel tasks. A runtime system is in charge of exploiting the inherent concurrency of the script, detecting the data dependencies between tasks and spawning them to the available resources. Furthermore, we show how this programming model can be built on top of a Big Data storage architecture, where the data stored in the backend is abstracted and accessed from the application in the form of persistent objects.

Keywords

Scientic computing, parallel programming models, Python, Big Data storage

1 Introduction

The use of the Python programming language for scientific computing has been gaining momentum in recent years, sometimes replacing traditional tools such as Matlab.[1] The fact that Python is free software makes it available to anyone at no cost, and its portability enables execution on a variety of platforms. The language itself is compact, readable and very suited for rapid prototyping, while being powerful enough for writing large applications. Even though some people do not consider it efficient enough, Python integrates very well with C/C++, allowing us to easily invoke external modules programmed in those languages for the sake of performance or code reuse. Furthermore, it has a rich set of scientific libraries, e.g. for numeric computation,[2] data processing and analysis,[3] plotting[4] and graphical interfaces.[5]

All of these advantages make Python appealing to the scientific community but, for the language to be used in big projects, it needs to be parallelizable. Its default and most widely-used implementation, CPython,[6] cannot run multiple threads at once because of a global lock in the interpreter. In response, a number of alternatives for spawning multiple Python processes have appeared,[7] both for shared-memory environments and for distributed infrastructures such as grids, clusters and clouds, as will be discussed in Section 2. However, most of these solutions are MPI wrappers, only permit to launch embarrassingly parallel computations or require the user to include infrastructure-related details in the application.

In that sense, this paper presents PyCOMPSs, a parallel programming framework for Python applications that overcomes the aforementioned limitations of other approaches. PyCOMPSs is built on top of COMPSs (Lordan et al., 2014; Tejedor and Badia, 2008) and it aims to facilitate the development of computational

[1]Department of Computer Sciences, Barcelona Supercomputing Center (BSC-CNS), Barcelona, Spain
[2]Artificial Intelligence Research Institute (IIIA), Spanish Council for Scientific Research (CSIC), Barcelona, Spain

Corresponding author:
Rosa M Badia, Artificial Intelligence Research Institute (IIIA), Centro Nacional de Supercomputación, Nexus II Building, c/ Jordi Girona, 29, Barcelona 08034, Spain.
Email: rosa.m.badia@bsc.es

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

workflows in Python for distributed infrastructures. For that purpose, it offers a programming model based on sequential development, the application is a plain sequential Python script, where the user annotates the functions to be run as asynchronous parallel tasks. A runtime system is in charge of automatically exploiting the inherent concurrency of the script, detecting the data dependencies between tasks and spawning them to the available resources.

Furthermore, for PyCOMPSs scripts to easily access and compute on huge data sets, the programming model has been implemented on top of a Big Data storage platform. In our proposal, the application sees such Big Data in the form of regular Python objects that can be made persistent. The objective here is twofold: first, make possible for a Python script to handle big objects (too big to fit in the memory of a single node) and, second, offer a simple mechanism to access those objects and manage their persistency. Thus, Python programs can keep and share the modifications made to objects, with no need to explicitly read/write the data to persistent storage (e.g. a file); the actual access to the data is kept transparent to the programmer, who uses the references to the objects to operate with them in a normal way. By placing PyCOMPSs on top of such a storage platform, we intend to integrate the automatic detection and exploitation of parallelism in Python scripts with a data model that targets both Big Data and persistency.

The paper is structured as follows. Section 2 discusses the related work. Section 3 provides an overview of the PyCOMPSs programming model. Section 4 describes the basic runtime implementation of the model. Section 5 explains how PyCOMPSs has been integrated in a Big Data storage platform. Section 6 presents the results of the experiments. Finally, the conclusions and future work can be found in Section 7.

2 Related work

Although Python distributions include a threading module, there is not a simple way for parallelizing applications. Although several threads can be used, the most popular implementation (CPython) does not support thread concurrency and the Global Interpreter Lock (GIL) does not enable more than one thread to execute Python bytecode at a time. The Wiki page at https://wiki.python.org/moin/parallelprocessing classified the alternatives available for parallelizing Python applications according to the platform that is tackled: symmetric multiprocessing (SMP), cluster computing, cloud computing and grid computing.

The Python multiprocessing module[8] supports the spawning of processes in SMP machines using an API similar to the threading module, with explicit calls to create processes, argument passing, execution, synchronization, result collection, etc. The multiprocessing module avoids the GIL issue by using subprocesses initiated with a fork system call instead of threads.

Parallel Python (PP)[9] is a Python module which provides mechanisms for parallel execution of Python code on SMP and clusters. It is based on an API which provides explicit functions to specify the number of workers to be used, submit the jobs for execution, get the results from the workers, etc. Similar to the multiprocessing module, all of the management of the parallelism is delegated to the programmer, mixing the actual algorithm with the parallelism management.

Singh et al. (2013) described the Pool/Map approach, as well as the Process/Queue approach. The Pool/Map approach spawns a pool of worker processes and returns a list of results. The Python map function is extended to the multiprocessing module and can be used with the Pool class to instantiate with a single operation a set of worker processes that will work in parallel and collect the results, using the regular Python syntax. The Process/Queue approach again extends the multiprocessing module to enable more than one input parameter to the concurrent function. This approach creates two first in first out (FIFO) queues, one for the input data and one for receiving output data. The parallel worker processes are started using the Process class. In the same paper, the Pool/Map, Process/Queue approach and Parallel Python are compared when solving parallel astronomical data processing applications, the Process/Queue approach showing better performance. Although in terms of features, Parallel Python will also enable execution in clusters, while the approaches based in multiprocessing can only be executed in a single node.

Other approaches to implement parallelism with Python are based on MPI wrappers. MPI for Python (mpi4py) (Dalcín et al., 2008) provides bindings of the message passing interface (MPI) standard for the Python programming language, allowing any Python program to exploit multiple processors. It supports point-to-point and collective communications of any pickable Python object, as well as optimized communications of Python object exposing the single-segment buffer interface. A similar approach is followed by pyMPI.[10]

dispy[11] is a Python framework for parallel execution of computations by distributing them across multiple processors in a single machine (SMP), among many machines in a cluster, grid or cloud. It is based on an API that enables to explicitly create jobs, submit them for execution, execute callbacks on job completion, wait for finalization, etc.

Alternatives to flow-based programming in Python are described at https://wiki.python.org/moin/FlowBasedProgramming. Ruffus (Goodstadt, 2010) offers a syntax to explicitly define computational pipelines in Python using decorators. Cosmos (Gafni et al.,

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

2014) is another alternative to implement workflows in Python using the MapReduce paradigm.

Celery[12] also uses decorators to implement an asynchronous invocation of tasks in a server. The execution units are tasks, that are executed concurrently on a single or more worker web servers using multiprocessing. Tasks can be executed asynchronously or synchronously as invocations to a web service.

With regard these previous alternatives, PyCOMPSs provides an approach to parallelize codes without the need of expressing explicitly the concurrency and to execute them in distributed platforms, including SMPs, clusters and clouds. Ruffus, as PyCOMPSs, is based on the use of decorators, although the decorators in Ruffus explicitly tell how the task functions are connected between them and the data exchanged between the nodes of the pipelines are always files, while PyCOMPSs is also more flexible in this aspect, giving support to regular Python objects. Ruffus uses the Python multiprocessing module to create one process per task in the pipeline. While Ruffus also provides a parallel construct, it is totally explicit, while in PyCOMPSs the parallelism is implicit, automatically detected by the runtime.

Another approach to develop programmatic workflows is Swift, which is based in its own scripting language and finds the opportunities for parallel execution as a combination of parallel loop constructs and an implicit data-flow programming model (Zhao et al., 2007). As data types that are handled, Swift tasks only can manage files. Previously to this work, GRIDSs (Badia et al., 2003) (the predecessor of COMPSs) was proposed for workflows described in C/C++ also being able to unveil task parallelism through the creation at runtime of a data-dependence task graph defined by files as data objects.

When comparing PyCOMPSs with previous work, it provides a generic (non ad-hoc solution) with a simple and flexible interface. The codes are programmed in native Python, with decorators used to indicate which parts of the code should be instantiated as tasks and to give hints about the arguments' directionality which will be used by the PyCOMPSs runtime to automatically find data dependences between tasks. The API of PyCOMPSs is composed of a single function that it is used to synchronize the completion of tasks at the end of parallel sections.

PyCOMPSs enables programming in parallel in the closest way to sequential code with the minimum number of references in the code to computational and data resources and concurrency. Also, the concurrency that it is supported is not only fork-join or MapReduce, but more flexible parallelism described in tasks' graphs that are dynamically built at execution time.

In the same way that PyCOMPSs abstracts concurrency from applications, access to distributed persistent objects is also transparent to the programmer thanks to our persistent storage layer. Current solutions that support persistent objects in Python provide an intuitive way to manage them from applications. Some examples are object databases such as ZODB,[13] or the ORM libraries SQLAlchemy[14] or SQLObject.[15] However, they do not implement the functionality that allows PyCOMPSs to exploit data locality, which will be detailed in Section 5. In particular, they do not provide the information that PyCOMPSs requires to schedule tasks next to the objects they manipulate. In addition, our storage layer also provides the ability to iterate on persistent dictionaries both exploiting data locality and parallelism.

3 PyCOMPSs programming model

PyCOMPSs is a programming framework that aims to facilitate the parallelization of Python scripts. For that purpose, it offers a simple programming model based on sequential development: a PyCOMPSs application is a plain sequential Python script. In the model, the user is mainly responsible for (i) identifying the functions to be executed as asynchronous parallel tasks and (ii) annotating them with a standard Python decorator. A runtime system is in charge of exploiting the inherent concurrency of the script, automatically detecting and enforcing the data dependencies between tasks and spawning these tasks to the available resources, which can be nodes in a cluster, cloud or grid.

Section 3.1 will present a first example of a Python script parallelized with PyCOMPSs, while Section 3.2 will provide a more comprehensive specification of the PyCOMPSs programming model syntax and options.

3.1 First example

The first step when programming with PyCOMPSs consists in defining the tasks of the application. A Python script may be composed by calls to multiple functions, and some of them may be computationally intensive. Such functions may be good candidates to be defined as tasks, so that they can be executed in parallel on a set of distributed resources.

As an example, let us consider the code in Figure 1(a). This script performs some computation for a number of steps (line 5) and merges the partial results, of type dictionary, into a final dictionary (line 6, variable `result`). All of the computations receive a configuration parameter initialized in line 3. The script can be executed as a sequential Python program but, in order to parallelize it with PyCOMPSs, we will define as tasks three functions called by the script: `init_conf`, `compute_step` and `merge`.

Task definition in PyCOMPSs is done by means of Python decorators,[16] which are part of the standard

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

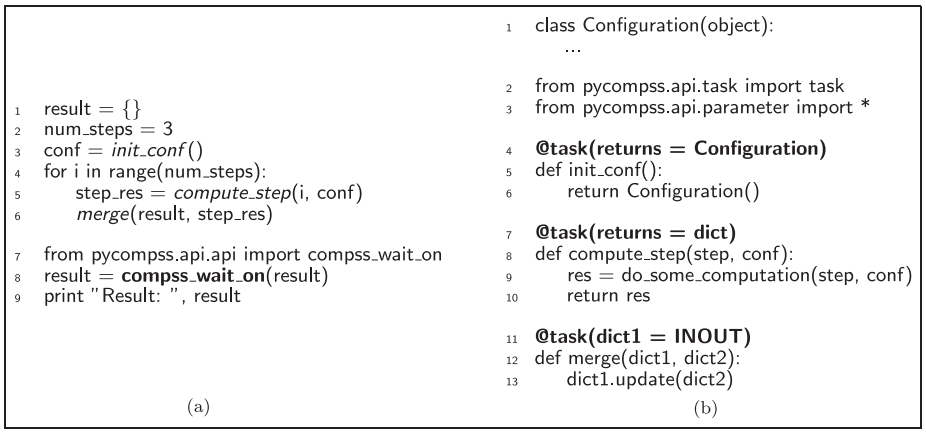

Figure 1. Example of a sequential Python script parallelized with PyCOMPSs: (a) main program of the script; (b) task definition.

Python syntax and permit to wrap calls to functions and with some additional behaviour. In particular, the user needs to add, before the definition of the function, a `@task` decorator that describes the task. Continuing with the example, Figure 1(b) shows the code of the aforementioned functions together with their `@task` decorators. Function `init_conf` returns an object of class `Configuration` (defined in line 1), as stated by its decorator (line 4). Similarly, `compute_step` returns a dictionary (as specified in the decorator of line 7) and receives two parameters: an integer and a `Configuration` object. Finally, `merge` receives two dictionary parameters and merges them into the first one (line 13); in order to state that the first dictionary will be modified inside the task, the decorator defines it as an input–output parameter (line 11).

Once the functions intended to be tasks are properly decorated, a Python script is ready to be executed with PyCOMPSs. When running the script, PyCOMPSs creates an asynchronous task for each invocation to a decorated function, adding these tasks to a dependency graph. The nodes of such graph are the tasks, and the edges represent their data dependencies. In order to detect data dependencies between tasks, PyCOMPSs uses the information about parameter direction specified in the `@task` decorator (e.g. `INOUT` for the `dict1` parameter of function `merge` in Figure 1(b)). For instance, if a task writes some data and a subsequent task reads the same data, there is a data dependency between these tasks. Data dependencies are automatically enforced by PyCOMPSs to ensure the correct execution of the application.

In that sense, Figure 2 depicts the task-dependency graph built on the fly by PyCOMPSs for the example in Figure 1(a). The first asynchronous task that is created corresponds to function `init_conf`, and after that the main program proceeds immediately to execute the loop of computation and merge tasks. Inside the loop, a total of three `compute_step` tasks are generated, and they all depend on the previous `init_conf` task

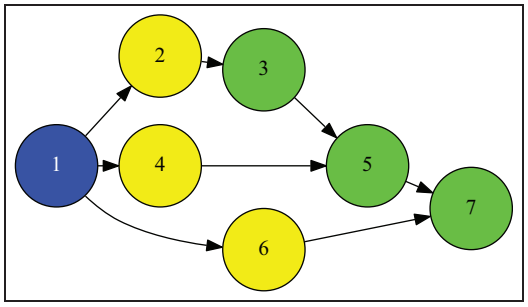

Figure 2. Task dependency graph corresponding to the example script in Figure 1.

because they receive the configuration object `conf` as input parameter: if no direction is specified for a parameter, it defaults to `IN`. The loop also generates three `merge` tasks, each depending on their corresponding `compute_step` for the partial result of the iteration (variable `step_res`); moreover, each `merge` task depends on the result produced by the previous iteration (stored in `result`) and, consequently, they are arranged in a chain of tasks.

Once the loop is completely unrolled, the program reaches lines 7–9, where the final result in variable `result` is printed. Before printing it, though, the script needs to synchronize for the last value of `result`, produced by the last `merge` task. In order to do that, PyCOMPSs provides an API function, `compss_wait_on`, which stalls the main control flow until the last value of `result` is obtained. Hence, the call to `compss_wait_on` in line 8 will wait for the last `merge` task to finish before getting and returning the final `result`, so that it can be printed in line 9.

It is important to note how PyCOMPSs is able to detect and exploit the inherent concurrency of the script, as exhibited by the graph in Figure 2: the `compute_step` tasks can be executed in parallel, while the merge tasks cannot. The programmer does not control such parallelization explicitly; instead, she programs

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

Table 1. Arguments of the `@task` decorator.

Argument	Value
Formal parameter name	- `INOUT`: read–write object. - `OUT`: write-only object. - `FILE`: read-only file. - `FILE_INOUT`: read–write file. - `FILE_OUT`: write-only file.
`returns`	`int` (for integer and boolean), `long`, `float`, `str`, `dict`, `list`, `tuple`, user-defined classes.
`isModifier`	`True` (default) or `False`.
`priority`	`True` or `False` (default).

sequentially and provides information about the task parameters and their direction. Actually, the program in Figure 1 can be executed sequentially by using a sequential version of the PyCOMPSs libraries, imported in line 7 of Figure 1(a) and lines 2–3 of Figure 1(b).

3.2 Syntax

After discussing a complete PyCOMPSs example in Section 3.1, this section specifies in a more comprehensive way the syntax of the PyCOMPSs programming model.

3.2.1 Task definition. In PyCOMPSs, the user can define as a task:

- *functions*;
- *instance methods* (methods invoked on objects);
- *class methods* (static methods belonging to a class).

PyCOMPSs tasks can be defined by means of the `@task` decorator, which provides information about the parameters of the function/method and about the task itself. Table 1 summarizes all of the arguments supported by the decorator.

On the one hand, the metadata corresponding to a parameter is specified as an argument of the decorator, whose name is the formal parameter's name and whose value defines the type and direction of the parameter. Thus, when including parameter metadata in the `@task` decorator, the user has the options shown in the first row of Table 1. The parameter types and directions can be as follows.

- *Types*: primitive types (integer, long, float, boolean), strings, objects (instances of user-defined classes, dictionaries, lists, tuples, complex numbers) and files are supported. PyCOMPSs is able to automatically infer the parameter type for primitive types, strings and objects, but not for files, which need to be defined as `FILE`.
- *Direction*: it can be read-only (`IN` - default), read-write (`INOUT`) or write-only (`OUT`).

```
my_file = 'sample_file.txt'
func(my_file)
```
(a)

```
from pycompss.api.task import task
from pycompss.api.parameter import *

@task(f = FILE_INOUT)
def func(f):
    fd = open(f, 'r+')
    ...
```
(b)

Figure 3. Example of a task receiving a file parameter: (a) call call to function from the application; (b) decorated function.

Consequently, in the following cases there is no need to include an argument in the `@task` decorator for a given task parameter.

- Parameters of primitive types (integer, long, float, boolean) and strings: the type of these parameters can be automatically inferred, and their direction is always `IN`.
- Read-only object parameters: the type of the parameter is automatically inferred, and the direction defaults to `IN`.

On the other hand, there are three reserved arguments of the `@task` decorator, also presented in Table 1. First, if the function or method returns a value, the programmer must specify the type of that value using the `returns` argument. Return values can be seen as parameters with `OUT` direction. Second, for tasks corresponding to instance methods, by default the task is assumed to modify the callee object (the object on which the method is invoked). The programmer can tell otherwise by setting the `isModifier` argument to `False`. Finally, the programmer can also mark a task as a high-priority task by setting the `priority` argument to `True`. This way, when the task is free of dependencies, it will be scheduled before any of the available low-priority (regular) tasks. This functionality is useful for tasks that are in the critical path of the application's task dependency graph.

Figure 3 shows an example of the usage of file parameters in PyCOMPSs tasks, which can be useful to program computational workflows. The script in Figure 3(a)call calls a function `func`, which receives a string parameter containing a file name. Note that, for PyCOMPSs to know that `my_file` is actually a file path that must be treated as such, and not as any other string, the user needs to state that the parameter is of type `FILE`. In addition, since the code of `func` updates the file, the `INOUT` direction also needs to be specified.

Regarding the other arguments of the `@task` decorator: (i) Figure 4 defines a task that returns an

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

```
@task(returns = int)
def ret_func():
    return 1
```

Figure 4. Definition of a task returning an integer.

```
class MyClass(object):
    ...
    @task(isModifier = False)
    def instance_method(self):
        ... # self is NOT modified here
```

Figure 5. Example of usage of the `isModifier` flag.

```
@task(priority = True)
def func():
    ...
```

Figure 6. Example of usage of the `priority` flag.

Table 2. PyCOMPSs API functions.

Function	Use
`compss_wait_on (obj, to_write = True)`	Synchronizes for the last version of an object and returns it.
`compss_open (file_name, mode = 'r' )`	Synchronizes for the last version of a file and returns its file descriptor.

integer value, (ii) Figure 5 shows a usage example of the `isModifier` argument and (iii) Figure 6 defines a task with priority.

3.2.2 Main program. The main program of the application is a sequential Python script (or scripts) that contains calls to tasks. Tasks can modify or generate data (e.g. a file or object), and these data can eventually be accessed from the main program. Before doing so, however, the programmer needs to synchronize that data (i.e. stall the main control flow until obtaining the last version produced by the task, which can imply waiting for the task to finish). As a result, the main program can work with the correct version of the data.

Depending on the data type that is being synchronized, two API functions may need to be invoked (summarized in Table 2).

- `compss_wait_on(obj, to_write = True)`: synchronizes for the last version of object `obj` and returns the synchronized object. It can receive an optional boolean parameter `to_write`, which defaults to `True`, that indicates whether the main program will modify the returned object.
- `compss_open(file_name, mode ='r' )`: similar to the Python `open()` call. It synchronizes for the last version of file `file_name` and returns the file descriptor for that synchronized file. It can receive an optional parameter mode, which defaults to'r', containing the mode in which the file will be opened (the open modes are analogous to those of Python `open()`).

To illustrate the use of the aforementioned API functions, the example in Figure 7 first creates an object of class `MyClass` and invokes a task method called `method` that modifies the object; the object is then synchronized with `compss_wait_on()`, so that it can be used in the main program from that point on. After that, the script invokes a task `func` that writes a file, which is later synchronized by calling `compss_open()`.

If the programmer defines as a task a function or method that returns a value, that value is not generated until the task executes. However, in order to keep the asynchrony of the task invocation, PyCOMPSs manages *future objects* (Walker et al., 1990): a representant object is immediately returned to the main program when a task is invoked. The future object mechanism is applied to primitive types, strings and objects (including the Python built-in types list, dictionary, tuple and complex).

As shown in Figure 8, a future object returned by a task (object `o`, line 4) can be involved in subsequent asynchronous task calls (lines 5 and 6), and PyCOMPSs will automatically find the corresponding data dependencies. On the other hand, in order to synchronize the future object from the main program, the programmer proceeds in the same way as with any object updated by a task (line 7).

4 Runtime system

The programming model described in Section 3 is enabled by a runtime system that parallelizes the application on behalf of the user.

This section describes the basic implementation of such a runtime system, i.e. how it generates and manages tasks and how it processes the data accessed by those tasks.

4.1 Architecture

PyCOMPSs was designed to operate on top of the COMPSs (Lordan et al., 2014; Tejedor and Badia, 2008) Java runtime system, acting as a language binding for Python applications. This design decision allowed

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

```
from pycompss.api.api import compss_wait_on,compss_open

my_obj = MyClass()
my_obj.method()
my_obj = compss_wait_on(my_obj)
...

my_file = 'file.txt'
func(my_file)
fd = compss_open(my_file)
...
```

(a)

```
@task(f = FILE_OUT)
def func(f):
    ...

class MyClass(object):
    ...

    @task()
    def method(self):
        ... # self is modified here
```

(b)

Figure 7. Example of synchronization: (a) main program containing the synchronization calls; (b) task definition task definition.

```
1  @task(returns = MyClass)
2  def ret_func():
3      return MyClass(...)

   # o is a future object
4  o = ret_func()
   ...
5  another_task(o)
   ...
6  o.yet_another_task()
   ...
7  o = compss_wait_on(o)
```

Figure 8. Future objects in PyCOMPSs.

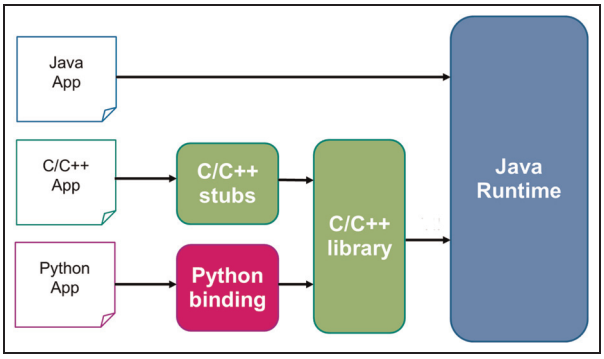

Figure 9. Architecture of the PyCOMPSs runtime system.

PyCOMPSs to leverage COMPSs' functionalities and, as a result, require a shorter development time.

Among the functionalities implemented by COMPSs there are data dependency analysis, task scheduling, data transfer and fault tolerance. In addition, it can execute applications on different kinds of distributed infrastructures such as clouds, clusters or grids. On top of the Java runtime, the Python binding handles the computations and data of the application and interacts (through a C++ library) with the Java libraries underneath. Figure 9 shows the architecture of the whole framework.

4.2 Task generation

Calls to functions decorated with `@task` are wrapped by a function of the Python binding, which forwards the function name and parameters to the Java runtime. With that information, the Java runtime creates a task and adds it to the data dependency graph, immediately returning the control to Python. At this point, the main program can continue executing right after the task invocation, possibly invoking more tasks.

Therefore, the Java runtime executes concurrently with the main program of the application, and as the latter issues new task creation requests, the former dynamically builds a task dependency graph. Such graph represents the inherent concurrency of the program, and determines what can be run in parallel. When all predecessor tasks' of a given task have completed, the task is ready to be scheduled on a resource. The available resources are specified in XML configuration files, and for each resource the runtime obtains the information of the number of simultaneous tasks that can be run on that resource. The runtime also maintains information about the number of tasks in execution in each resource, and therefore is also aware if any free slot is available to run a task.

The default scheduling policy of the runtime is locality-aware. When scheduling a task, the runtime system computes a score for all the available resources and chooses the one with the highest score. This score is the number of task input parameters that are already present on that resource, and consequently they do not need to be transferred. In such a way, the delays between tasks' executions are reduced.

4.3 Task offloading and data management

Once a task has been scheduled, and before submitting the task to the target resource for execution, the Java runtime transfers the missing task input data to that resource.

Tasks created by a PyCOMPSs application can receive different types of data as a parameter, as explained in Section 3.2. Specifically, tasks work with data that is either in memory (Python objects, primitive

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

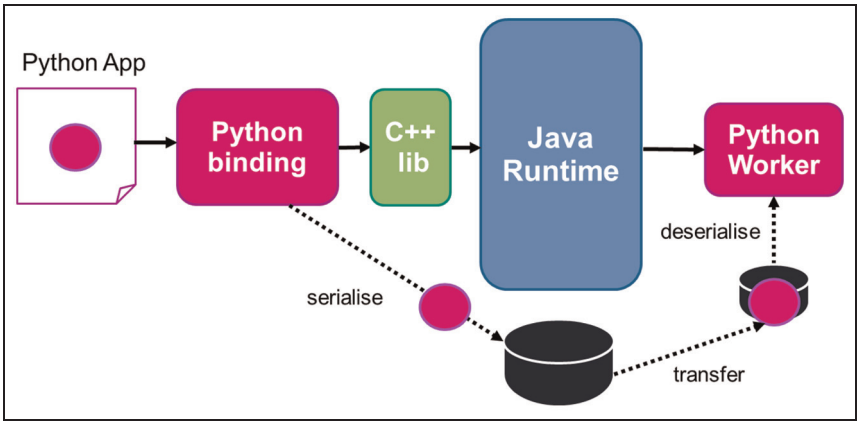

Figure 10. Management of objects in the basic implementation of the PyCOMPSs framework.

types) or disk (files). Such data needs to be forwarded to the Java runtime, which will be in charge of transferring it to the destination resource where the task will run. Primitive types in Python are mapped by the Python binding to the equivalent types in Java, and files are processed as such on the Java side.

Regarding Python objects, their management is a bit more complex. Figure 10 shows how objects are handled by PyCOMPSs. Objects can be created by a Python application in Python space and later be accessed by a task. Since neither these objects nor their associated classes exist in the Java space, they are serialized to files by the Python binding. Hence, when informing the Java runtime of the creation of a new task, the Python binding will define all object parameters as files so that the Java runtime can process and transfer them.

Once the input data for a task is transferred, the Java runtime will trigger the execution of a Python worker script on the worker resource. This script will process the request of running a task, check which data it needs (deserializing the input objects if necessary) and finally run the task.

The runtime processing described in this section obviously comes with an overhead. Therefore, for the remote execution of tasks to be worthwhile, the duration of these tasks should be in the order of tens or hundreds of milliseconds, depending on the number of resources to be used.

5 Support for Big Data

Section 4 described the basic implementation of the PyCOMPSs runtime, where tasks operate on data that is either in memory (regular Python objects and primitive types) or disk (files). In order to make PyCOMPSs capable of orchestrating applications that process large amounts of data, the data management of PyCOMPSs was extended both at programming model and runtime system levels.

In that sense, this section presents the work on integrating PyCOMPSs in a persistent object storage platform. Such platform provides PyCOMPSs scripts, as well as the tasks they generate, with a distributed, fault-tolerant and efficient storage layer. Furthermore, multiple applications can share data in a concurrent way through this layer.

The main objective of the aforementioned platform is facilitate the access to data as much as possible. Any function in a Python script, even if it has not been developed with PyCOMPSs in mind, should be able to seamlessly work with objects in memory or with persistent objects, with no need to be adapted for each case. It is the responsibility of the main program, which invokes the function, to decide whether the object instance is made persistent or not depending on the purpose of such object, but this decision is transparent to the function.

Figure 11 depicts the overall structure of the proposed storage platform. Python scripts programmed with the PyCOMPSs model are located at the top of the stack, and the tasks they generate are processed by the PyCOMPSs runtime system. These scripts rely on a Storage API in order to create, delete, insert, retrieve and iterate over persistent data. At its turn, PyCOMPSs also invokes the Storage API, mainly to obtain locality information about persistent data.

The Storage API can be implemented by multiple Storage Backends. A Storage Backend is responsible for storing data in a set of distributed resources, managing data format and organization and optimizing data queries. Currently, the Storage API does not provide any functions to manage transactions, so the consistency guarantees offered to applications are inherited from the Storage Backend used.

The next subsections present the extensions to PyCOMPSs that were required to operate on top of

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

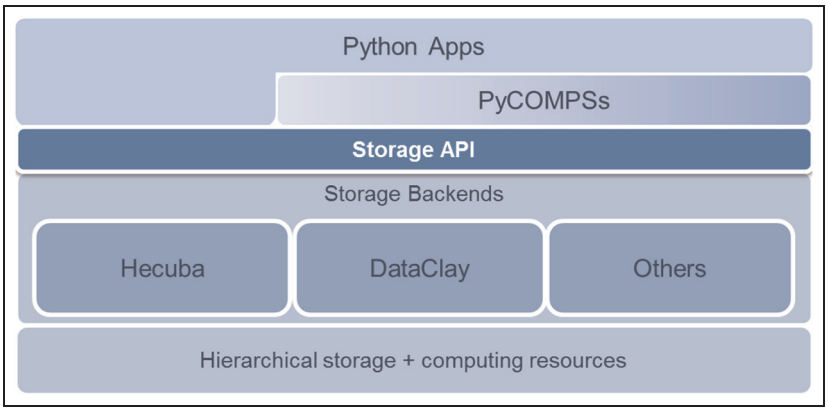

Figure 11. PyCOMPSs on top of a persistent storage layer.

```
1  class Foo(object):
2      """ ClassField bar int """
3      def __init__(self, val):
4          self.bar = val
```
(a)

```
1  @task()
2  def another_func(foo):
       ...
3  o = Foo('MyFooObject')
   ...
4  another_func(o)
```
(c)

```
1  @task()
2  def my_func(foo1, foo2):
3      sum = foo1.bar + foo2.bar
       print 'Sum:', sum
4  o1 = Foo(1)
5  o2 = Foo(2)
   ...
6  o1.make_persistent('MyFooObject')
   ...
7  my_func(o1, o2)
```
(b)

Figure 12. Managing persistent objects: (a) class definition; (b) producer script that makes an object persistent; (c) consumer script that loads a persistent object.

this storage platform, and they also describe a backend that implements the Storage API.

5.1 Programming with persistent objects

The data that resides in the storage platform can have multiple formats, depending on the backend that stores it. Nevertheless, from the point of view of the application, such data is abstracted and it is always accessed as Python objects, no matter how it is stored underneath.

Consequently, the programming model proposed here is based on the use of objects. Such objects can be made persistent by an application, that is, objects initially allocated in memory can be backed by the persistent storage layer. From that point on, changes to the object will be forwarded to the backend as well. On the other hand, objects that were made persistent can be retrieved by other applications. This enables interactive sharing of data between applications that run concurrently.

5.1.1 Managing persistency. In order to deal with persistent objects, the storage platform requires some knowledge about the classes that will be stored. In particular, since Python is a dynamically typed programming language, the names and types of the attributes are not included in the class definition, and they should be specified to correctly map the class representation to the backend. This can be done by means of docstrings, the standard Python documentation mechanism, as shown in Figure 12(a), line 2. This would be then used to generate the corresponding code that implements the Storage API.

To make an object persistent, the programmer needs to invoke the `make_persistent method`, which is part of the Storage API. Figure 12(b) shows a simple example where two regular Python objects `o1` and `o2` of class `Foo` are created (lines 4–5). After that, `make_persistent` is invoked on `o1` in order to be stored as persistent (line 6). The `make_persistent` method receives a parameter of type string that specifies the alias, a name or identifier, that the user wants to give to that object. Such alias must be unique in the storage namespace, and it can be used afterwards to retrieve the object.

A persistent object can be involved in a task call like any regular (non-persistent) object, as exemplified in

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

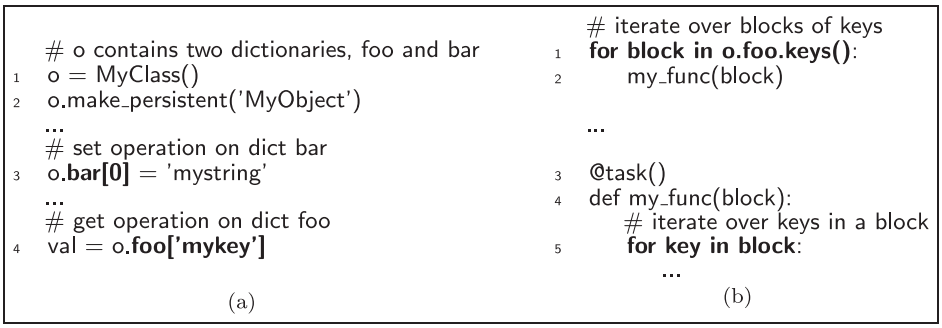

Figure 13. Managing persistent dictionaries: (a) getting and setting elements; (b) iterating over keys.

Table 3. Storage API for applications.

Method	Use
`make_persistent`	Stores an object
`delete_persistent`	Deletes an object from persistent storage
Constructor('alias')	Retrieves the persistent object with the alias provided

line 7 of Figure 12(b), and the programmer does not need to provide any additional information in the `@task` decorator (line 1). `my_func` can be a pre-existing function, completely unaware of object persistency. Thus, it manipulates the persistent object `o1` and the regular object `o2` in the same way (line 3). However, in the case of `o1`, any changes to the object are transparently persisted in the distributed storage. Also importantly, the PyCOMPSs runtime will try to schedule the task on the resource where the object is stored to favour locality.

Figure 12(c) depicts a small consumer script that loads the same object that was stored in Figure 12(b). In order to do that, the script invokes a constructor of class `Foo` that receives the alias of the object as parameter (line 3). This constructor is provided by the implementation of the model, and it returns a reference to the desired persistent object. Once the reference is obtained, the object can also participate in a task call (line 4).

Table 3 details the methods included in the Storage API that provide applications with the ability to manage persistent objects.

5.1.2 Persistent dictionaries. A particular case when working with persistent objects is that of dictionaries. The Python dictionary built-in type is an unordered set of key-value pairs. Keys can be of any immutable type, including tuples. Dictionaries (or objects containing dictionaries) can be made persistent and, when that happens, the data they contain can be easily mapped to a key-value store in a Storage Backend.

In that sense, a PyCOMPSs application can manipulate a persistent dictionary as it would do with any regular Python dictionary, e.g. adding a new key-value pair or requesting the value associated to a given key. Since the dictionary is persistent, any operation will be translated into an access to the persistent data. In Figure 13(a), an object `o` of class `MyClass` is instantiated in line 1 and made persistent in line 2. Let us assume that object `o` has two attributes of type dictionary, `foo` and `bar`. Even if `o` is persistent, the programmer can add new entries to its dictionaries (line 3) or get the value for a given key (line 4) in a normal way. In the case of a persistent object, though, such operations will be forwarded to the Storage Backend, e.g. a distributed key-value store.

On the other hand, Python allows to retrieve the set of keys in a dictionary by invoking the `keys()` method on it. In addition, these keys can be iterated to execute some operation for each key-value pair. In a persistent dictionary, obtaining the keys and iterating over them can be done like with a regular dictionary, but the semantics of these operations differs from the original one.

As can be seen in Figure 13(b), when iterating over the keys of a persistent dictionary (line 1), the iterator will return blocks of keys instead of individual keys. Those blocks correspond to the partitions of key-value pairs in the underlying Storage Backend. This feature is leveraged to exploit data locality: PyCOMPSs tasks that receive one or more blocks as parameters can be created (line 2), and the PyCOMPSs runtime will try to schedule them in the resource where those blocks reside. If that happens, any access to the keys in that block will be local. For instance, if such block is iterated inside the task to obtain the keys it holds (line 5), the keys (and their associated values) will already be on the resource that runs the task.

5.2 Storage API and Backends

The Storage API lies underneath the application and the PyCOMPSs runtime (Figure 11) and offers

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

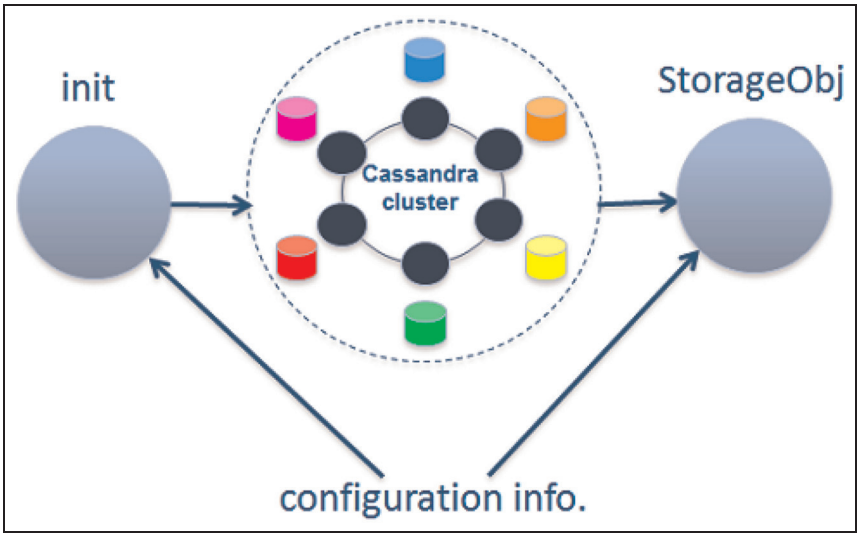

Figure 14. Configuration of StorageObj.

methods to access persistent objects and obtain information about them.

Section 5.1 introduced those methods that the Storage API provides to manage objects from applications, i.e. to make objects persistent, to load them or to modify their attributes. In addition to these methods, the Storage API also provides a set of methods that enable PyCOMPSs to take into account data locality when scheduling tasks with persistent objects as parameters. In order to do that, the PyCOMPSs runtime invokes the getLocations method of the Storage API, which returns the resources where a given persistent object or block is stored. With this information, PyCOMPSs can try to submit a task to a resource where (at least part of) its input data is already present, thus preventing remote accesses to the data from the task. This method receives as a parameter the identifier of an object, which is assigned by the storage platform. This identifier can be obtained by means of the method `getID`. Finally, the method `getByID` provides a reference to the persistent object with the identifier specified as a parameter.

The methods of the Storage API can be implemented by one or more Storage Backends, which transform the generic operations on persistent objects into specific operations on their associated backend data. The next subsection describes the Storage Backend that has been developed so far for its use in the Storage Platform.

5.2.1 Hecuba. Hecuba is a set of tools and interfaces developed in our research group, that aims to facilitate programmers an efficient and easy interaction with non-relational databases. In particular, we have added to Hecuba the implementation of the interface necessary to provide PyCOMPSs with a Storage Backend suitable to support Big Data applications. Currently, this interface is implemented on Apache Cassandra database, although it is easy to port this implementation to any non-relational key-value data store.

Cassandra[17] is a distributed and highly scalable key-value database. In order to guarantee availability and partitioning properties, Cassandra offers eventual consistency, also known as BASE (Basically Available, Soft state, Eventual consistency) as opposed to ACID. Cassandra implements a non-centralized architecture, based on peer-to-peer communication, in which all nodes of the cluster are able to receive and serve queries. Each node of the cluster is assigned a token. A partitioner function uses this token to decide how data is distributed among the nodes in the shape of a ring.

Data in Cassandra is stored on tables by rows, which are identified by a key chosen by the user. This key can be composed by one or several attributes. In each row the user can add additional attributes also identified by a name. In order to enhance data locality when accessing data stored in Cassandra, it is necessary to understand how data is organized. Cassandra stores data by rows, and one node is responsible for hosting a specific row. The target node is chosen based on the key of the row and of the token of each node by the partitioner algorithm. In order to guarantee data availability, Cassandra can be configured to keep several replicas for each data. In a setup with N replicas it is also necessary to choose the nodes that will hold each replica. The default replication algorithm selects the $N - 1$ nodes that come after the target node in the ring's order to store those replicas.

The mapping of a Python dictionary on a data model in Cassandra is straightforward as both consist on values indexed by keys. Thus, the implementation of a Persistent Dictionary backed up by Hecuba is straightforward too. We have decided to map each

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

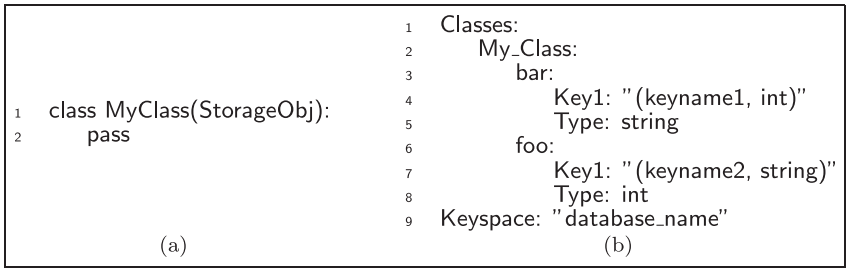

Figure 15. Defining a user class backed by Hecuba: (a) minimum class code; (b) data scheme definition.

class containing one or more Persistent Dictionary on a Cassandra table. This table is indexed by the same key attributes than the Persistent Dictionaries in the class, and contains as many non-key attributes as Persistent Dictionaries. In the current implementation Hecuba only supports the backing of Persistent Dictionaries but as part of our future work we plan to extend it to support other type of attributes.

Following with the example described in Section 5.1.2, we will show how to implement a class backed up by Hecuba. For the sake of rapid prototyping, our current implementation has some variations regarding the definition of classes explained in Section 5.1.1. In particular, Figure 15 shows the minimum code required to implement a class containing two Persistent Dictionaries. It is just necessary to indicate that the class is a subclass of `StorageObj`, which is a class implemented inside Hecuba, and that contains all of the methods that are necessary to access and manipulate data backed up by Cassandra. Thus, the code of the application is mostly independent of the data backend used. Note that programmers can add their own methods to the class definition, the only requirement is that the constructor of the `StorageObj` class has to be executed each time a new object of the class is instantiated.

In addition to this simple class definition, users have to provide the information necessary to connect to the Cassandra cluster (IP addresses and ports) and to configure the database. Configuration of the database requires information about the database identifier (keyspace name) and the data scheme (for each table, name and type of each attribute and which attribute is part of the key). Information about the Cassandra cluster can be set through environment variables and it is common to all applications executed on this cluster. Information about the database configuration can be codified in a YAML file. Figure 15 represents the YAML file that the user should provide to complete the configuration of the application. During the initialization phase of the application, Hecuba code reads and parses this YAML file and configures itself to be able to adapt at runtime the generic code of the `StorageObj` methods to the particular data schemes defined (see Figure 14).

In the following we describe the implementation of the interface implemented by Hecuba as part of the StorageObj class. The methods exported to programmers allow us to perform the following tasks.

- Make an object persistent: this operation is performed through the `make_persistent` method of the `StorageObj`. The implementation of this method creates the Cassandra table that it is necessary to hold the object and populates it with the data that the object had in memory.
- Object instantiation: it is implemented by the constructor of the `StorageObj class`. If the constructor receives a parameter, the constructor binds the instantiated object with the corresponding Cassandra table. In this way, all of the following accesses to that object will be translated into accesses to that table. In contrast, if the constructor does not receive a parameter, all data associated with that new object instance will be kept in memory until the makePersistent method is executed.
- Query/Update data: these operations are implemented through the `get_item` and the `set_item` methods of the Persistent Dictionaries. There are two different scenarios: if the object is in memory, the implementations of these methods are translated into the usual methods of a Python dictionary; if the object is backed by a Cassandra table, then the implementations of these methods consist on queries performed on the Cassandra table.
- Data iteration: as we have explained in Section 5.1.2, the implementation of the `keys` method of a Persistent Dictionary returns a block of keys that will be the input parameter of a task. In order to enhance data locality, we decided to create the blocks of keys based on their node location. This way, PyCOMPSs can use the information about the data location to schedule tasks local to the data. Thus, we have implemented two different iterators: one of them to create the different blocks of keys and the other one to traverse all of the keys in a block (see figure 16).
- Delete a persistent object: this operation is implemented by the `delete_persistent` method of

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

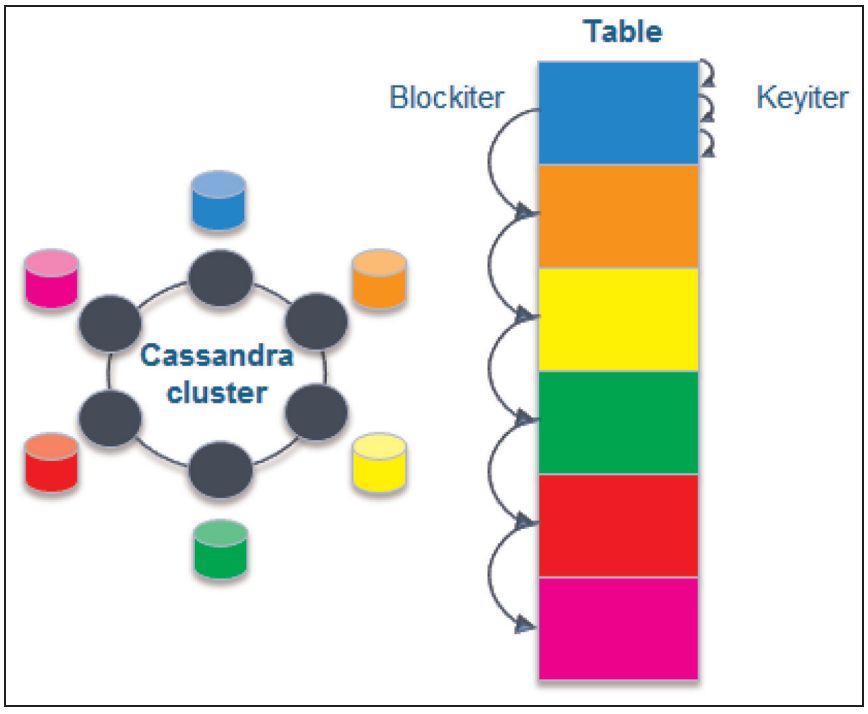

Figure 16. Implementation of iterators.

the `StorageObj` and it just deletes from the database the table that was backing the object.

6 Evaluation

This paper presents the evaluation of PyCOMPSs in two different cases: evaluation of PyCOMPSs standalone and evaluation of PyCOMPSs with the support for Big Data with the Hecuba backend.

6.1 Test environment

The evaluation has been performed in two different environments. The standalone PyCOMPSs tests have been executed in the MareNostrum III supercomputer. MareNostrum III is a cluster with 3056 compute nodes, each of them 2 × Intel SandyBridge-EP E5-2670/1600 20M with 8 cores at 2.6 GHz connected with a Infiniband FDR10 network.

The tests of PyCOMPSs with the support for Big Data have been performed in a cluster of five nodes. Each node is equipped with 2 Intel Xeon Quad-Core L5630 at 2.13 GHz, 24 GB RAM and 6 TB HDD, and they are interconnected with Gigabit Ethernet. Out of the five nodes, one executes the main program of the application and the PyCOMPSs master runtime, while the rest are PyCOMPSs worker nodes that execute tasks. The 4 worker nodes also form a Cassandra ring and have the Hecuba backend installed. The master node has an Hecuba client and a Cassandra client.

In all cases we used PyCOMPSs version 1.2.[18]

6.2 Standalone tests

PyCOMPSs has been evaluated in MareNostrum III with two different applications: DimSweep and NeuronCorr. DimSweep performs a cluster architecture exploration using the Dimemas simulator performing a parameter sweep of several configuration values: Fabric Interconnection Network latency and bandwidth, number of nodes, CPU speed, and intranode latency and bandwidth.

Dimemas (Labarta et al., 1996) is a simulator of the behaviour of MPI applications under different network and architecture conditions by means of replaying the applications' executions recorded in traces. This PyCOMPSs example is codified with two different tasks' types: one that executes the Dimemas simulation and another one that accumulates the results. While the simulation tasks are independent between them, the accumulation tasks create a chain of dependencies that serialize all them. To avoid a long chain of these tasks at the end of the execution, these tasks are prioritized in such a way that are inserted between the simulation tasks.

For the experiment a case with a real execution tracefile and a combination of parameters' configurations that generated 2304 tasks was used. Figure 17 shows the results obtained when varying the number of cores used for PyCOMPSs workers from 16 to 512. The time is the total elapsed time and the speedup is computed

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

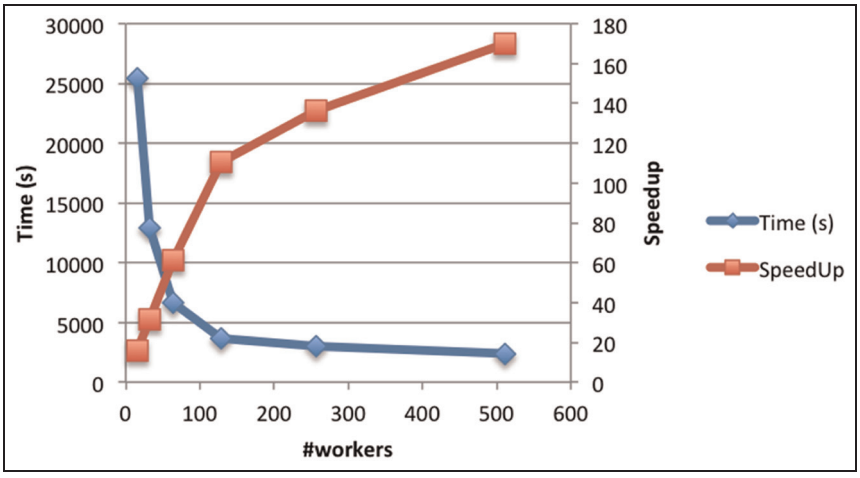

Figure 17. Performance of DimSweep. The chart shows elapsed time and speed-up.

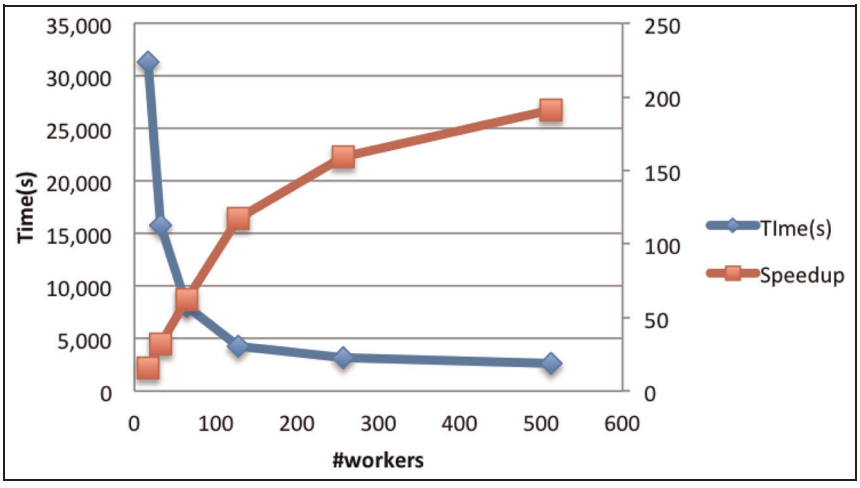

Figure 18. Performance of NeuronCorr. The chart shows elapsed time and speed-up.

using as baseline the case with 16 workers but scaled to 16 (therefore, optimal speedup is the number of workers). The results show very good scaling up to 128 workers, reducing a bit for larger processor counts.

NeuronCorr is a neuroscience data processing example that computes all mutual cross-correlations between all pairs of a set of spike data (Denker et al., 2010). The original example was written in Parallel Python and has been translated to PyCOMPSs. The example has two tasks' types: one that compute the cross-correlations for a block of data and a second one that gathers the results in a data structure. Since the gathers create a chain of tasks, this type of task is prioritized to avoid a long serial chain at the end of the execution. The evaluation was performed with a data set that generates 2048 tasks.

Figure 18 shows the results obtained for the NeuronCorr example, with a speed-up similar to the DimSweep example.

6.3 Tests on persistent storage

For this section, PyCOMPSs has been evaluated with NeuronCorr application, in the Minerva cluster (Figure 19). All simulation tasks are independent, and input data is obtained from a Cassandra table previously created. Once the execution starts, input data is structured in blocks, and this information is used by tasks to know which Cassandra node contains its information. Once each task has finished executing, resulting data is saved in Cassandra.

Table 4 presents results using different configurations of Cassandra nodes and PyCOMPSs workers for a given input data. The column in the left shows the mapping of the Cassandra nodes to Minerva cluster nodes. The column in the middle shows the mapping of PyCOMPSs workers to the Minerva cluster nodes, using multithreading in some cases (marked with *). The same total count of Cassandra nodes and

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

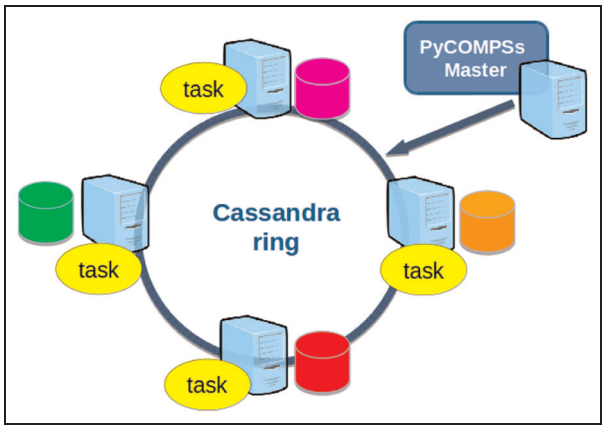

Figure 19. Hecuba topology for persistent storage.

Table 4. Performance results for the NeuronCorr application with persistent storage for some configurations. The column on the left describes the Cassandra configuration, indicating how the Cassandra nodes are mapped in the Minerva cluster nodes. The column in the middle describes the number of PyCOMPSs workers mapped in each of the Minerva cluster nodes. In the cases marked with *, multithreading is activated.

Cassandra topology	PyCOMPSs workers	Time (s)
4, 4, 0, 0	4, 4, 0, 0	19,228
8, 8, 0, 0	8, 8, 0, 0	10,273
16, 16, 0, 0	16, 16, 0, 0 *	12,867
16, 16, 16, 16	16, 16, 16, 16 *	6644

PyCOMPSs workers is used to balance workload. These results show good scaling between the cases of the same nature (with multithreading or without multithreading). The results also reflect the impact of sharing resources between threads when multithreading is used.

7 Conclusions and future work

This paper has presented PyCOMPSs, a programming framework that facilitates the development of parallel computational workflows in Python. With PyCOMPSs, the user programs her script in a sequential fashion and decorates the functions to be run as asynchronous parallel tasks. A runtime system is in charge of exploiting the inherent concurrency of the script, detecting the data dependencies between tasks and spawning them to a set of distributed resources. Those resources can be physical nodes in a cluster or grid or virtual machines in a cloud.

We have also described an extension to the PyCOMPSs programming model and runtime system that builds on top of a persistent object storage layer. In this approach, the application data is stored in a distributed storage backend (e.g. a key-value store like Cassandra). From the application point of view, such data is abstracted and accessed as Python objects, no matter the particular backend being used underneath. PyCOMPSs tasks can receive persistent objects as parameters, and these tasks will be scheduled taking into account data locality, launching them to resources that store the data they need. Furthermore, multiple applications can share data in a concurrent way through this persistent object layer.

The future work includes incorporating a new backend for persistent objects, based on the Scalable Key-Value Store[19] being developed at IBM. On the other hand, we will offer a web portal for programming, deploying and executing PyCOMPSs scripts on the cloud, which will leverage the IPython notebook tool as the main interface for interactive script development and execution.

Funding

This work has been supported by the Spanish Government (grant number SEV-2011-00067 of Severo Ochoa Program and contract number TIN2012-34557, Computación de Altas Prestaciones VI), by the SGR programme of the Catalan Government (grant number 2014-SGR-1051), by The Human Brain Project funded by the European Commission (contract number 604102) and by the Intel-BSC Exascale Lab collaboration.

Notes

1. See http://www.mathworks.com/products/matlab/
2. See http://www.numpy.org/
3. See http://www.scipy.org/ and http://pandas.pydata.org/
4. See http://matplotlib.org/
5. See http://ipython.org/notebook.html
6. See https://www.python.org/
7. See https://wiki.python.org/moin/parallelprocessing
8. See https://docs.python.org/2/library/multiprocessing.html
9. See http://www.parallelpython.com
10. See http://pympi.sourceforge.net
11. See http://dispy.sourceforge.net
12. See http://www.celeryproject.org
13. See http://www.zodb.org
14. See http://www.sqlalchemy.org/
15. See http://sqlobject.org/
16. See https://www.python.org/dev/peps/pep-0318/
17. See http://cassandra.apache.org/
18. Available at http://www.bsc.es/compss
19. See https://github.com/Scalable-Key-Value

References

Badia RM, Labarta J, Sirvent R, Pérez JM, Cela JM and Grima R (2003) Programming grid applications with grid superscalar. *Journal of Grid Computing* 1(2): 151–170.

Dalcín L, Paz R, Storti M and D'Elía J (2008) {MPI} for python: Performance improvements and MPI-2

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

extensions. *Journal of Parallel and Distributed Computing* 68(5): 655–662. DOI:10.1016/j.jpdc.2007.09.005.

Denker M, Wiebelt B, Fliegner D, Diesmann M and Morrison A (2010) Practically trivial parallel data processing in a neuroscience laboratory. In: Grün S and Rotter S (eds.) *Analysis of Parallel Spike Trains* (Springer Series in Computational Neuroscience, vol. 7). New York: Springer, pp. 413–436. DOI:10.1007/978-1-4419-5675-0_20.

Gafni E, Luquette LJ, Lancaster AK, Hawkins JB, Jung JY, Souilmi Y, Wall DP and Tonellato PJ (2014) COSMOS: Python library for massively parallel workflows. *Bioinformatics* 30(20): 2956–2958. DOI: 10.1093/bioinformatics/btu385.

Goodstadt L (2010) Ruffus: a lightweight python library for computational pipelines. *Bioinformatics* 26(21): 2778–2779.

Labarta J, Girona S, Pillet V, Cortes T and Gregoris L (1996) DiP: A parallel program development environment. In: Bouge L, Fraigniaud P, Mignotte A and Robert Y (eds.) *Euro-Par'96 Parallel Processing* (Lecture Notes in Computer Science, vol. 1124). Berlin: Springer, pp. 665–674. DOI:10.1007/BFb0024763.

Lordan F, Tejedor E, Ejarque J, Rafanell R, Álvarez J, Marozzo F, Lezzi D, Sirvent R, Talia D and Badia R (2014) ServiceSs: An interoperable programming framework for the cloud. *Journal of Grid Computing* 12(1): 67–91. DOI: 10.1007/s10723-013-9272-5.

Singh N, Browne LM and Butler R (2013) Parallel astronomical data processing with python: Recipes for multicore machines. *Astronomy and Computing* 2: 1–10.

Tejedor E and Badia RM (2008) COMP Superscalar: Bringing GRID Superscalar and GCM Together. In: *Eighth IEEE international symposium on cluster computing and the grid (CCGrid '08)*, Lyon, France, pp. 185–193. DOI:10.1109/CCGRID.2008.104.

Walker EF, Floyd R and Neves P (1990) Asynchronous remote operation execution in distributed systems. In: *10th international conference on distributed computing systems (ICDCS-10)*, pp. 253–259.

Zhao Y, Hategan M, Clifford B, Foster I, Von Laszewski G, Nefedova V, Raicu I, Stef-Praun T and Wilde M (2007) Swift: Fast, reliable, loosely coupled parallel computation. In: *2007 IEEE congress on services*. IEEE, pp. 199–206.

Author biographies

Enric Tejedor received his PhD from the Technical University of Catalonia (UPC, Spain) in 2013. He conducted his doctorate research as a member of the Grid Computing and Clusters group of the Barcelona Supercomputing Center, where he participated in several EU research projects. As part of his PhD, he also carried out two internships at the IBM TJ Watson Research Center (NY, USA). In 2015 he started working at CERN (Switzerland) as a senior software engineer

Yolanda Becerra is an associate professor at the Computer Architecture Department of the UPC and an associate researcher in the Barcelona Supercomputing Center (BSC) where she is conducting research about resource management strategies for BigData applications in the Cloud. Since 2008 she is involved in a research collaboration project with IBM Research, focused on the development of non-centralized resource management strategies and mechanisms. Since 1998, she lectures courses on Operating Systems in several schools of the UPC and in the Masters program of the Computer Architecture Department of the UPC.

Guillem Alomar is a Computer Engineer currently employed in the Barcelona Supercomputing Center as junior developer in the Autonomic Systems and eBusiness Platform group. He obtained a Bachelors Degree in Computer Science at the Universitat Pompeu Fabra in 2012 and a Masters Degree in High Performance Computing, Information Theory and Security (specialization in HPC) at the Universitat Autònoma de Barcelona in 2013.

Anna Queralt is a senior researcher at the Storage-system research group at the Barcelona Supercomputing Center (BSC). She holds a PhD on Computer Science (2009) from the Technical University of Catalonia (UPC-BarcelonaTech). She was involved in teaching and research activities at the UPC from 2003 to 2012, and was also a part-time lecturer at the Open University of Catalonia, in both cases in the area of object-oriented software engineering. She was a visiting researcher at the KRDB Research Centre at the Free University of Bozen-Bolzano in 2010. Her research interests are data sharing, storage systems and programming models for complex platforms, and is currently participating in several projects related to these areas.

Rosa M Badia holds a PhD on Computer Science (1994) from the Technical University of Catalonia (UPC). She is a Scientific Researcher from the Consejo Superior de Investigaciones Científicas (CSIC) and team leader of the Workflows and Distributed Computing research group at the Barcelona Supercomputing Center (BSC). She was involved in teaching and research activities at the UPC from 1989 to 2008, where she was an Associated Professor since year 1997. From 1999 to 2005 she was involved in research and development activities at the European Center of Parallelism of Barcelona (CEPBA). Her current research interest are programming models for complex platforms (from multicore, GPUs to Grid/Cloud). The group lead by Dr Badia has been developing StarSs programming model for more than 10 years, with a high success in adoption by application developers. Currently the group focuses its efforts in two instances of StarSs: OmpSs for heterogeneous platforms and COMPSs/PyCOMPSs for distributed computing including Cloud. For this last case, the group

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016

has been doing efforts on interoperability through standards, for example using OCCI to enable COMPSs to interact with several Cloud providers at a time. She has published more than 150 papers in international conferences and journals in the topics of her research. She has participated in several European projects, for example BEinGRID, Brein, CoreGRID, OGF-Europe, SIENA, TEXT and VENUS-C, and currently she is participating in the project Severo Ochoa (at the Spanish level), ASCETIC, Euroserver, The Human Brain Project, EU–Brazil CloudConnect, and transPLANT and it is a member of HiPEAC2 NoE.

Jordi Torres is a full professor at UPC Barcelona Tech and research manager at Barcelona Supercomputing Center (BSC) working in EU and industrial research and development projects. His current principal interest as a researcher is Processing and Analyzing Big Data in a Sustainable Cloud. He acts as an expert and disseminator on these topics for various organizations, companies and mentoring entrepreneurs.

Toni Cortes is the manager of the storage-system group at the BSC (since 2006) and is also an associate professor at Universitat Politècnica de Catalunya (since 1998). He received his MS in computer science in 1992 and his PhD also in computer science in 1997 (both at Universitat Politècnica de Catalunya). Since 1992, he has been teaching operating system and computer architecture courses at the Barcelona School of Informatics (UPC) and from 2000 to 2004 he also served as vicedean for international affairs at the same school. His research concentrates on storage systems, programming models for scalable distributed systems and operating systems. He has published 98 technical papers (23 journal papers and 75 international conferences and workshops), 2 book chapters, and has coedited one book on mass storage systems. In addition, he has also advised 10 PhD theses since 1997. He has been involved in several EU projects (Paros, Nanos, POP, XtreemOS, Scalus, IOlanes, PRACE, MontBlanc, EUDAT, Big Storage, IOStack, Rethinkbig, and Severo Ochoa) and has also participated in cooperation with IBM (TJW research lab) on scalability issues both for MPI and UPC.

Jesús Labarta is full professor on Computer Architecture at the Technical University of Catalonia (UPC) since 1990. Since 1981 he has been lecturing on computer architecture, operating systems, computer networks and performance evaluation. His research interest has been centered on parallel computing, covering areas from multiprocessor architecture, memory hierarchy, programming models, parallelizing compilers, operating systems, parallelization of numerical kernels, performance analysis and prediction tools. Since 2005 he has been responsible for the Computer Science Research Department within the Barcelona Supercomputing Center (BSC). He has been involved in research cooperation with many leading companies on HPC-related topics. His major directions of current work relate to performance analysis tools, programming models and resource management. His team distributes the Open Source BSC tools (Paraver and Dimemas) and performs research on increasing the intelligence embedded in the performance analysis tools. He is involved in the development of the OmpSs programming model and its different implementations for SMP, GPUs and cluster platforms. He has been involved in Exascale activities such as IESP and EESI where he has been responsible of the Runtime and Programming model sections of the respective Roadmaps. He leads the programming models and resource management activities in the HPC subproject of the Human Brain Project.

Downloaded from hpc.sagepub.com at Middle East Technical Univ on January 13, 2016