COMP Superscalar

User Manual

Application development guide

Version: 2.0.rc1704

May 10, 2017

This manual only provides information about the development of COMPSs applications. Specifically, it details the programming model features available in Java, Python and C/C++ languages.

For an extensive list of COMPSs application examples (codes, execution commands, results, logs, etc.) please refer to the *COMPSs Sample Applications* guide at `http://compss.bsc.es/` .

For information about the installation process please refer to the *COMPSs Installation Guide* available at `http://compss.bsc.es/` .

For further information about the application execution please refer to the *COMPSs User Manual: Application execution guide* available at `http://compss.bsc.es/` .

Contents

1 COMP Superscalar (COMPSs) 1

2 Java 2
2.1 Programming Model . . . 2
2.1.1 Main application code . . . 2
2.1.2 Remote methods code . . . 3
2.1.3 Java annotated interface . . . 4
2.1.4 Alternative method implementations . . . 6
2.1.5 Java API calls . . . 7
2.2 Application Compilation . . . 7
2.3 Application Execution . . . 8

3 Python Binding 9
3.1 Programming Model . . . 9
3.1.1 Task Selection . . . 9
3.1.2 Constraints . . . 11
3.1.3 Main Program . . . 12
3.1.4 Important Notes . . . 14
3.2 Application Execution . . . 15
3.2.1 Environment . . . 15
3.2.2 Command . . . 15

4 C/C++ Binding 16
4.1 Programming Model . . . 16
4.1.1 Task Selection . . . 16
4.1.2 Value and Object return . . . 17
4.1.3 Main Program . . . 17
4.1.4 Binding API . . . 18
4.1.5 Functions file . . . 19
4.1.6 Additional source Files . . . 19
4.1.7 Class Serialization . . . 19
4.1.8 Method - Task . . . 20
4.1.9 Task Constraints . . . 21
4.1.10 Task Versions . . . 21
4.2 Application Compilation . . . 22
4.3 Application Execution . . . 23
4.4 Execution Graph . . . 23

5 Constraints 25

6 Known Limitations 28

List of Figures

1 Matmul Execution Graph. 24

List of Tables

1 Arguments of the @*task* decorator. 11
2 COMPSs Python API functions. 13
3 Arguments of the *@constraint* decorator 26
4 Arguments of the *@Processor* decorator 27

COMP Superscalar (COMPSs)

COMP Superscalar (COMPSs) is a programming model which aims to ease the development of applications for distributed infrastructures, such as Clusters, Grids and Clouds. COMP Superscalar also features a runtime system that exploits the inherent parallelism of applications at execution time.

For the sake of programming productivity, the COMPSs model has four key characteristics:

- **Sequential programming:** COMPSs programmers do not need to deal with the typical duties of parallelization and distribution, such as thread creation and synchronization, data distribution, messaging or fault tolerance. thus eliminating most of the difficulties of concurrent/distributed programming. A task is a method or a service called from the application code that is intended to be spawned asynchronously and possibly run in parallel with other tasks on a set of resources, instead of locally and sequentially.

- **Infrastructure unaware:** COMPSs offers a model that abstracts the application from the underlying infrastructure. Hence, COMPSs programs do not include any detail that could tie them to a particular platform, like deployment or resource management. This makes applications portable between infrastructures with diverse characteristics.

- **Standard programming languages:** COMPSs natively supports Java applications, but also offers language bindings for Python and C/C++ applications.

- **No APIs:** In the case of COMPSs applications in Java, the model does not require to use any special API call, pragma or construct in the application; everything is standard Java syntax and libraries. As regards the Python and C/C++ bindings, a small set of API calls should be used on the COMPSs applications.

Java

This section illustrates the steps to develop a Java COMPSs application, to compile and to execute it. The *Simple* application will be used as reference code. The user is required to select a set of methods, invoked in the sequential application, that will be run as remote tasks on the available resources.

Programming Model

A COMPSs application is composed of three parts:

- **Main application code:** the code that is executed sequentially and contains the calls to the user-selected methods that will be executed by the COMPSs runtime as asynchronous parallel tasks.
- **Remote methods code:** the implementation of the tasks.
- **Java annotated interface:** It declares the methods to be run as remote tasks along with metadata information needed by the runtime to properly schedule the tasks.

The main application file name has to be the same of the main class and starts with capital letter, in this case it is **Simple.java**. The Java annotated interface filename is *application name+Itf.java*, in this case it is **SimpleItf.java**. And the code that implements the remote tasks is defined in the *application name + Impl.java* file, in this case it is **SimpleImpl.java**.

All code examples are in the */home/compss/workspace_java/* folder of the development environment.

Main application code

In COMPSs the user's application code is kept unchanged, no API calls need to be included in the main application code in order to run the selected tasks on the nodes.

The COMPSs runtime is in charge of replacing the invocations to the user-selected methods with the creation of remote tasks also taking care of the access to files where required. Let's consider the Simple application example that takes an integer as input parameter and increases it by one unit.

The main application code of Simple app (**Simple.java**) is executed sequentially until the call to the **increment()** method. COMPSs, as mentioned above, replaces the call to this method with the generation of a remote task that will be executed on an available node.

```
package simple;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import simple.SimpleImpl;

public class Simple {
```

```
  public static void main(String[] args) {
    String counterName = "counter";
    int initialValue = args[0];

    //-------------------------------------------------------//
    //Creation of the file which will contain the counter variable//
    //-------------------------------------------------------//
    try {
       FileOutputStream fos = new FileOutputStream(counterName);
       fos.write(initialValue);
       System.out.println("Initial counter value is "
                          +initialValue);
       fos.close();
    }catch(IOException ioe) {
       ioe.printStackTrace();
    }

    //-----------------------------------------------//
    //           Execution of the program            //
    //-----------------------------------------------//

    SimpleImpl.increment(counterName);

    //-----------------------------------------------//
    //    Reading from an object stored in a File    //
    //-----------------------------------------------//
    try {
       FileInputStream fis = new FileInputStream(counterName);
       System.out.println("Final counter value is "+fis.read());
       fis.close();
    }catch(IOException ioe) {
       ioe.printStackTrace();
    }
  }
}
```

Remote methods code

The following code contains the implementation of the remote method of the *Simple* application (**SimpleImpl.java**) that will be executed remotely by COMPSs.

```
package simple;

import  java.io.FileInputStream;
import  java.io.FileOutputStream;
import  java.io.IOException;
import  java.io.FileNotFoundException;

public class SimpleImpl {
  public static void increment(String counterFile) {
    try{
      FileInputStream fis = new FileInputStream(counterFile);
      int count = fis.read();
      fis.close();

      FileOutputStream fos = new FileOutputStream(counterFile);
      fos.write(++count);
      fos.close();
    }catch(FileNotFoundException fnfe){
      fnfe.printStackTrace();
    }catch(IOException ioe){
      ioe.printStackTrace();
    }
  }
```

```
}
```

Java annotated interface

The Java interface is used to declare the methods to be executed remotely along with Java annotations that specify the necessary metadata about the tasks. The metadata can be of three different types:

1. For each parameter of a method, the data type (currently *File* type, primitive types and the *String* type are supported) and its directions (IN, OUT or INOUT).
2. The Java class that contains the code of the method.
3. The constraints that a given resource must fulfill to execute the method, such as the number of processors or main memory size.

A complete and detailed explanation of the usage of the metadata includes:

- **Method-level Metadata:** for each selected method, the following metadata has to be defined:
  - **@Method:** Mandatory. It specifies the class that implements the method.
    - **isModifier** "true" if the method modifies the implicit object, "false" otherwise (it is a string not a java boolean).
    - **isReplicated** "true" if the method must be executed in all the worker nodes when invoked from the main application. (it is a string not a java boolean)
    - **priority** "true" if the task takes priority and "false" otherwise. This parameter is used by the COMPSs scheduler. (it is a string not a java boolean)
  - **@Constraints:** The user can specify the capabilities that a resource must have in order to run a method. For example, in a cloud execution the COMPSs runtime creates a VM that fulfils the specified requirements in order to perform the execution. A full description of the supported constraints can be found in Table 3 in Section 5.
- **Parameter-level Metadata (@Parameter):** for each parameter and method, the user must define:
  - **Direction:** *Direction.IN, Direction.INOUT or Direction.OUT*
  - **Type:** COMPSs supports the following types for task parameters:
    - **Basic types:** *Type.BOOLEAN, Type.CHAR, Type.BYTE, Type.SHORT, Type.INT, Type.LONG, Type.FLOAT, Type.DOUBLE*. They can only have **IN** direction, since primitive types in Java are always passed by value.
    - **String:** *Type.STRING*. It can only have **IN** direction, since Java Strings are immutable.

        * **File:** *Type.FILE*. It can have any direction (IN, OUT or INOUT). The real Java type associated with a FILE parameter is a String that contains the path to the file. However, if the user specifies a parameter as a FILE, COMPSs will treat it as such.
        * **Object:** *Type.Object*. It can have any direction (IN, OUT or INOUT).
    - **Return type:** Any object or a generic class object. In this case the direction is always OUT.

Basic types are also supported as return types. However, we do not recommend to use them because they cause an implicit synchronization

    - **Method modifiers:** the method has to be **STATIC**.

- **Service-level Metadata:** for each selected service, the following metadata has to be defined:
    - **@Service:** Mandatory. It specifies the service properties.
        * **namespace** Mandatory. Service namespace
        * **name** Mandatory. Service name.
        * **port** Mandatory. Service port.
        * **operation** Operation type.
        * **priority** True if the service takes priority, false otherwise. This parameter is used by the COMPSs scheduler.

The Java annotated interface of the Simple app example (SimpleItf.java) includes the description of the *Increment()* method metadata. The method interface contains a single input parameter, a string containing a path to the file counterFile. In this example there are constraints on the minimum number of processors and minimum memory size needed to run the method.

```
package simple;

import  integratedtoolkit.types.annotations.Constraints;
import  integratedtoolkit.types.annotations.task.Method;
import  integratedtoolkit.types.annotations.Parameter;
import  integratedtoolkit.types.annotations.parameter.Direction;
import  integratedtoolkit.types.annotations.parameter.Type;

public interface SimpleItf {

  @Constraints(computingUnits = "1", memorySize = "0.3")
  @Method(declaringClass = "simple.SimpleImpl")
  void increment(
      @Parameter(type = Type.FILE, direction = Direction.INOUT)
      String file
  );

}
```

Alternative method implementations

Since version 1.2, the COMPSs programming model allows developers to define sets of alternative implementations of the same method in the Java annotated interface. The following code depicts an example where the developer sorts an integer array using two different methods: merge sort and quick sort that are respectively hosted in the *packagepath.Mergesort* and *packagepath.Quicksort* classes.

```
@Method(declaringClass = "packagepath.Mergesort")
@Method(declaringClass = "packagepath.Quicksort")
void sort(
    @Parameter(type = Type.OBJECT, direction = Direction.INOUT)
    int[] array
);
```

As depicted in the example, the name and parameters of all the implementations must coincide; the only difference is the class where the method is implemented. This is reflected in the attribute declaringClass of the @Method annotation. Instead of stating that the method is implemented in a single class, the programmer can define several instances of the @Method annotation with different declaring classes.

As independent remote methods, the sets of equivalent methods might have common restrictions to be fulfilled by the resource hosting the execution. Or even, each implementation can have specific constraints. Through the @Constraints annotation, developers can specify the common constraints for a whole set of methods. In the following example only one core is required to run the method of both sorting algorithms.

```
@Constraints(computingUnits = "1")
@Method(declaringClass = "packagepath.Mergesort")
@Method(declaringClass = "packagepath.Quicksort")
void sort(
    @Parameter(type = Type.OBJECT, direction = Direction.INOUT)
    int[] array
);
```

However, these sorting algorithms have different memory consumption, thus each algorithm might require a specific amount of memory and that should be stated in the implementation constraints. For this purpose, the developer can add a @Constraints annotation inside each @Method annotation containing the specific constraints for that implementation. Since the Mergesort has a higher memory consumption than the quicksort, the following example sets a requirement of 1 core and 2GB of memory for the mergesort implementation and 1 core and 500MB of memory for the quicksort.

```
@Constraints(computingUnits = "1")
@Method(declaringClass = "packagepath.Mergesort", constraints = @Constraints(memorySize = "2.0"))
@Method(declaringClass = "packagepath.Quicksort", constraints = @Constraints(memorySize = "0.5"))
void sort(
    @Parameter(type = Type.OBJECT, direction = Direction.INOUT)
    int[] array
);
```

Java API calls

COMPSs also provides a explicit synchronization call, namely *barrier*, which can be used through the COMPSs Java API. The use of *barrier* forces to wait for all tasks that have been submitted before the barrier is called. When all tasks submitted before the *barrier* have finished, the execution continues.

```
import integratedtoolkit.api.COMPSs;

public class Main {
    public static void main(String[] args) {
        // Setup counterName1 and counterName2 files
        // Execute task increment 1
        SimpleImpl.increment(counterName1);
        // API Call to wait for all tasks
        COMPSs.barrier();
        // Execute task increment 2
        SimpleImpl.increment(counterName2);
    }
}
```

Application Compilation

A COMPSs Java application needs to be packaged in a *jar* file containing the class files of the main code, of the methods implementations and of the *Itf* annotation. Next we provide a set of commands to compile the Java Simple application detailed at the *COMPSs Sample Applications* available at our website `http://compss.bsc.es` .

```
compss@bsc:~$ cd workspace_java/simple/src/main/java/simple/
compss@bsc:~/workspace_java/simple/src/main/java/simple$ javac *.java
compss@bsc:~/workspace_java/simple/src/main/java/simple$ cd ..
compss@bsc:~/workspace_java/simple/src/main/java$ jar cf simple.jar simple/
compss@bsc:~/workspace_java/simple/src/main/java$ mv ./simple.jar ../../../jar/
```

In order to properly compile the code, the CLASSPATH variable has to contain the path of the *compss-engine.jar* package. The default COMPSs installation automatically add this package to the CLASSPATH; please check that your environment variable CLASSPATH contains the *compss-engine.jar* location by running the following command:

```
$ echo $CLASSPATH | grep compss-engine
```

If the result of the previous command is empty it means that you are missing the *compss-engine.jar* package in your classpath. We recommend to automatically load the variable by editing the *.bashrc* file:

```
$ echo "# COMPSs variables for Java compilation" >> ~/.bashrc
$ echo "export CLASSPATH=$CLASSPATH:/opt/COMPSs/Runtime/compss-engine.jar" >> ~/.bashrc
```

If you are using an IDE (such as Eclipse or NetBeans) we recommend you to add the *compss-engine.jar* file as an external file to the project. The *compss-engine.jar* file is available at your current COMPSs installation under the path */opt/COMPSs/Runtime/compss-engine.jar*.

Please notice that if you have performed a custom installation, the location of the package can be different.

An Integrated Development Environment for Eclipse is also available to simplify the development, compilation, deployment and execution COMPSs applications. For further information about the *COMPSs IDE* please refer to the *COMPSs IDE User Guide* available at `http://compss.bsc.es` .

Application Execution

A Java COMPSs application is executed through the *runcompss* script. An example of an invocation of the script is:

```
compss@bsc:~$ runcompss --classpath=/home/compss/workspace_java/simple/jar/simple.jar
                        simple.Simple 1
```

A comprehensive description of the *runcompss* command is available in the *COMPSs User Manual: Application Execution* document available at `http://compss.bsc.es` .

In addition to Java, COMPSs supports the execution of applications written in other languages by means of bindings. A binding manages the interaction of the no-Java application with the COMPSs Java runtime, providing the necessary language translation.

The next sections describe the Python and C/C++ language bindings offered by COMPSs.

Python Binding

COMPSs features a binding for Python 2.x applications. The next subsections explain how to program a Python application for COMPSs and how to configure the binding library.

Programming Model

Task Selection

As in the case of Java, a COMPSs Python application is a sequential program that contains calls to tasks. In particular, the user can select as a task:

- Functions
- Instance methods: methods invoked on objects.
- Class methods: static methods belonging to a class.

The task definition in Python is done by means of Python decorators instead of an annotated interface. In particular, the user needs to add, before the definition of the function/method, a @task decorator that describes the task.

As an example, let us assume that the application calls a function *func*, which receives a string parameter containing a file name and an integer parameter. The code of *func* updates the file.

```
my_file = 'sample_file.txt'
func(my_file, 1)
```

In order to select *func* as a task, the corresponding @*task* decorator needs to be placed right before the definition of the function, providing some metadata about the parameters of that function. 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. The parameter types and directions can be:

- Types: *primitive types* (integer, long, float, boolean), *strings*, *objects* (instances of user-defined classes, dictionaries, lists, tuples, complex numbers) and *files* are supported.
- Direction: it can be read-only (*IN* - default), read-write (*INOUT*) or write-only (*OUT*).

COMPSs is able to automatically infer the parameter type for primitive types, strings and objects, while the user needs to specify it for files. On the other hand, the direction is only mandatory for *INOUT* and *OUT* parameters. Thus, when defining the parameter metadata in the @*task* decorator, the user has the following options:

- *INOUT*: the parameter is read-write. The type will be inferred.

- *OUT*: the parameter is write-only. The type will be inferred.
- *FILE*: the parameter is a file. The direction is assumed to be *IN*.
- *FILE_INOUT*: the parameter is a read-write file.
- *FILE_OUT*: the parameter is a write-only file.

Consequently, please note that 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 by COMPSs, and their direction is always *IN*.
- Read-only object parameters: the type of the parameter is automatically inferred, and the direction defaults to *IN*.

Continuing with the example, in the following code snippet the decorator specifies that *func* has a parameter called *f*, of type *FILE* and *INOUT* direction. Note how the second parameter, *i*, does not need to be specified, since its type (integer) and direction (*IN*) are automatically inferred by COMPSs.

```
from pycompss.api.task import task
from pycompss.api.parameter import *
@task (f = FILE_INOUT)
def func(f, i):
    fd = open(f, 'r+')
    ...
```

If the function or method returns a value, the programmer must specify the type of that value using the *returns* argument of the @*task* decorator:

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

Moreover, if the function or method returns more than one value, the programmer must specify how many and their type in the *returns* argument. The next next code snippet shows how to specify that two values (an integer and a list) are returned:

```
@task(returns  = (int, list))
def ret_func():
    return 1, [2, 3]
```

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 of the @*task* decorator to *False*.

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

The programmer can also mark a task as a high-priority task with the *priority* argument of the @*task* decorator. In 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.

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

Table 1 summarizes the arguments that can be found in the @task decorator.

Argument	Value
Formal parameter name	- INOUT: read-write parameter, all types except file (primitives, strings, objects). - OUT: read-write parameter, all types except file (primitives, strings, objects). - FILE: read-only file parameter. - FILE_INOUT: read-write file parameter. - FILE_OUT: write-only file parameter.
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)

Table 1: Arguments of the @*task* decorator.

Constraints

As in Java COMPSs applications, it is possible to define constraints for each task. To this end, the decorator @*Constraint* followed by the desired constraints needs to be placed over the @Task decorator.

```
from pycompss.api.task import task
from pycompss.api.constraint import constraint
from pycompss.api.parameter import INOUT

@constraint (ComputingUnits="4")
@task (c = INOUT)
```

```
def func(a, b, c):
    c += a*b
    ...
```

This decorator enables the user to set the particular constraints for each task, such as the amount of Cores required explicitly. Alternatively, it is also possible to indicate that the value of a constraint is specified in a environment variable. A full description of the supported constraints can be found in Table 3 in Section 5.

For example:

```
from pycompss.api.task import task
from pycompss.api.constraint import constraint
from pycompss.api.parameter import INOUT

@constraint (ComputingUnits="4", AppSoftware="numpy,scipy,gnuplot", memorySize="$MIN_MEM_REQ")
@task (c = INOUT)
def func(a, b, c):
    c += a*b
    ...
```

Please, take into account that in order to respect the constraints, the peculiarities of the infrastructure must be defined in the *resources.xml* file.

Main Program

The main program of the application is a sequential code that contains calls to the selected tasks. In addition, when synchronizing for task data from the main program, there exist four API functions that can to be invoked:

- *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 have 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()*).

- *compss_wait_on(obj, to_write = True)*: synchronizes for the last version of object *obj* and returns the synchronized object. It can have an optional boolean parameter *to_write*, which defaults to *True*, that indicates whether the main program will modify the returned object. It is possible to wait on a list of objects. In this particular case, it will synchronize all future objects contained in the list.

- *barrier()*: performs a explicit synchronization, but does not return any object. The use of *barrier()* forces to wait for all tasks that have been submitted before the *barrier()* is called. When all tasks submitted before the *barrier()* have finished, the execution continues.

To illustrate the use of the aforementioned API functions, the following example first invokes a task *func* that writes a file, which is later synchronized by calling *compss_open()*. Later in the program, an object of class *MyClass* is created and a task method *method* that

modifies the object is invoked on it; the object is then synchronized with *compss_wait_on()*, so that it can be used in the main program from that point on.

Then, a loop calls again ten times to *func* task. Afterwards, the barrier performs a synchronization, and the execution of the main user code will not continue until the ten *func* tasks have finished.

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

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

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

for i in range(10):
    func(str(i) + my_file)
barrier()
...
```

The corresponding task selection for the example above would be:

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

class MyClass(object):
    ...

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

Table 2 summarizes the API functions to be used in the main program of a COMPSs Python application.

Function	Use
compss_open(file_name, mode = 'r')	Synchronizes for the last version of a file and returns its file descriptor.
compss_wait_on(obj, to_write = True)	Synchronizes for the last version of an object (or a list of objects) and returns it.
barrier()	Wait for all tasks submitted before the barrier.

Table 2: COMPSs Python API functions.

Important Notes

If the programmer selects as a task a function or method that returns a value, that value is not generated until the task executes.

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

...

# o  is a future object
o = ret_func()
```

The object returned can be involved in a subsequent task call, and the COMPSs runtime will automatically find the corresponding data dependency. In the following example, the object *o* is passed as a parameter and callee of two subsequent (asynchronous) tasks, respectively:

```
# o  is a future object
o = ret_func()

...

another_task(o)

...

o.yet_another_task()
```

In order to synchronize the object from the main program, the programmer has to synchronize (using the *compss_wait_on* function) in the same way as with any object updated by a task:

```
# o  is a future object
o = ret_func()

...

o = compss_wait_on(o)
```

Moreover, it is possible to synchronize a list of objects. This is particularly useful when the programmer expect to synchronize more than one elements (using the *compss_wait_on* function):

```
# l  is a list of objects where some/all of them may be future objects
l = []
for i in range(10):
    l.append(ret_func())

...

l = compss_wait_on(l)
```

For instances of user-defined classes, the classes of these objects should have an empty constructor, otherwise the programmer will not be able to invoke task instance methods on those objects:

```
class MyClass(object):
    def __init__(self): # empty constructor
        ...

    ...

o = ret_func()

# invoking a task instance method on a future object can only
# be done when an empty constructor is defined in the object's
# class
o.yet_another_task()
```

In order to make the COMPSs Python binding function correctly, the programmer should not use relative imports in the code. Relative imports can lead to ambiguous code and they are discouraged in Python, as explained in:

```
http://docs.python.org/2/faq/programming.html#what-are-the-best-practices-for-using-import-in-a-module
```

Application Execution

The next subsections describe how to execute applications with the COMPSs Python binding.

Environment

The following environment variables must be defined before executing a COMPSs Python application:

JAVA_HOME: Java JDK installation directory (e.g. */usr/lib/jvm/java-7-openjdk/*)

Command

In order to run a Python application with COMPSs, the runcompss script can be used, like for Java and C/C++ applications. An example of an invocation of the script is:

```
compss@bsc:~$ runcompss \
                --lang=python \
                --pythonpath=$TEST_DIR \
                --library_path=/home/user/libdir \
                $TEST_DIR/test.py arg1 arg2
```

For full description about the options available for the runcompss command please check the *COMPSs User Manual: Application Execution* available at `http://compss.bsc.es` .

C/C++ Binding

COMPSs provides a binding for C and C++ applications. The new C++ version in the current release comes with support for objects as task parameters and the use of class methods as tasks.

Programming Model

Task Selection

As in Java the user has to provide a task selection by means of an interface. In this case the interface file has the same name as the main application file plus the suffix "idl", i.e. Matmul.idl, where the main file is called Matmul.cc.

```
interface Matmul
{
      // C functions
      void initMatrix(inout Matrix matrix,
                      in int mSize,
                      in int nSize,
                      in double val);

      void multiplyBlocks(inout Block block1,
                          inout Block block2,
                          inout Block block3);

      // C++ class methods
      void Block::multiply(in Block block1,
                           in Block block2);

      static Matrix Matrix::init(in int mSize,
                                 in int bSize,
                                 in double val);
};
```

The syntax of the interface file is shown in the previous code. Tasks can be declared as classic C function prototypes, this allow to keep the compatibility with standard C applications. In the example, initMatrix and multiplyBlocks are functions declared using its prototype, like in a C header file, but this code is C++ as they have objects as parameters (objects of type Matrix, or Block).

A class method can be also a task, and it is declared using its signature. In the example, Block::multiply and Matrix::init are class methods. In this example, C functions encapsulates object method calls, as we will see later.

The grammar for the interface file is:

```
["static"] return-type task-name ( parameter {, parameter }* );

return-type = "void" | type

ask-name = <qualified name of the function or method>

parameter = direction type parameter-name

direction = "in" | "out" | "inout"

type = "char" | "string" | "int" | "short" | "long"
```

```
    | "float" | "double" | "boolean" | "File" | class-name

class-name = <qualified name of the class>
```

Value and Object return

The binding allows returning a value (void, int, long, float, etc.) or an object from a function or method. In C/C++ the default policy is to make a copy of the value or object when it is returned [A = foo();], and this copy (A) is a new position in memory whom reference or address is not possible to know before the return statement. As the COMPSs runtime cannot know such reference before returning from the task execution (foo) it must do a synchronization before the return statement for the correct value to be copied when returning. This is called an explicit synchronization.

Alternatively, the return of a value or an object can be done also by mean of an out or inout parameter, and no explicit synchronization is needed because the reference is passed to the binding in this case using the & operator [foo(&A);].

Main Program

The next listing includes an example of matrix multiplication written in C++.

```
#define DEBUG_BINDING
#include "Matmul.h"
#include "Matrix.h"
#include ""Block.h"
int N; //MSIZE
int M; //BSIZE
double val;
int main(int argc, char **argv)
{
      Matrix A;
      Matrix B;
      Matrix C;

       N = atoi(argv[1]);
       M = atoi(argv[2]);
      val = atof(argv[3]);

      compss_on();

      A = Matrix::init(N,M,val);

      initMatrix(&B,N,M,val);
      initMatrix(&C,N,M,0.0);

      cout << "Waiting for initialization...\n";

      compss_wait_on(B);
      compss_wait_on(C);

      cout << "Initialization ends...\n";

      C.multiply(A, B);

      compss_off();
      return 0;
}
```

The developer has to take into account the following rules:

1. The directive **DEBUG_BINDING** can be defined if we need debug information from the binding.
2. A header file with the same name as the main file must be included, in this case **Matmul.h**. This header file is automatically generated by the binding and it contains other includes and type-definitions that are required.
3. A call to the **compss_on** binding function is required to turn on the COMPSs runtime.
4. As in C language, out or inout parameters should be passed by reference by means of the "**&**" operator before the parameter name.
5. Synchronization on a parameter can be done calling the **compss_wait_on** binding function. The argument of this function must be the variable or object we want to synchronize.
6. There is an **implicit synchronization** in the init method of Matrix. It is not possible to know the address of "A" before exiting the method call and due to this it is necessary to synchronize before for the copy of the returned value into "A" for it to be correct.
7. A call to the **compss_off** binding function is required to turn off the COMPSs runtime.

Binding API

Besides the aforementioned **compss_on**, **compss_off** and **compss_wait_on** functions, the C/C++ main program can make use of a variety of other API calls to better manage the synchronization of data generated by tasks. These calls are as follows:

- *void compss_ifstream(char * filename, ifstream & ifs)*: given an uninitialized input stream *ifs* and a file *filename*, this function will synchronize the content of the file and initialize *ifs* to read from it.
- *void compss_ofstream(char * filename, ofstream & ofs)*: behaves the same way as *compss_ifstream*, but in this case the opened stream is an output stream, meaning it will be used to write to the file.
- *FILE* compss_fopen(char * file_name, char * mode)*: similar to the C/C++ *fopen* call. Synchronizes with the last version of file *file_name* and returns the FILE* pointer to further reference it. As the mode parameter it takes the same that can be used in *fopen* (*r, w, a, r+, w+* and *a+*).
- *void compss_wait_on(T & obj)*: synchronizes for the last version of object obj, meaning that the execution will stop until the value of *obj* up to that point of the code is received (and thus all tasks that can modify it have ended).

- *void compss_delete_file(char * file_name)*: makes a synchronized delete of file *filename*. When all previous tasks have finished updating the file, it is deleted.
- *void compss_barrier()*: similarly to the Python binding, performs an explicit synchronization without a return. When a *compss_barrier* is encountered, the execution will not continue until all the tasks submitted before the *compss_barrier* have finished.

Functions file

The implementation of the tasks in a C or C++ program has to be provided in a functions file. Its name must be the same as the main file followed by the suffix "-functions". In our case Matmul-functions.cc.

```
#include "Matmul.h"
#include "Matrix.h"
#include "Block.h"

void initMatrix(Matrix *matrix,int mSize,int nSize,double val){
     *matrix = Matrix::init(mSize, nSize, val);
}

void multiplyBlocks(Block *block1,Block *block2,Block *block3){
     block1->multiply(*block2, *block3);
}
```

In the previous code, class methods have been encapsulated inside a function. This is useful when the class method returns an object or a value and we want to avoid the explicit synchronization when returning from the method.

Additional source Files

Other source files needed by the user application must be placed under the directory "**src**". In this directory the programmer must provide a **Makefile** that compiles such source files in the proper way. When the binding compiles the whole application it will enter into the src directory and execute the Makefile.

It generates two libraries, one for the master application and another for the worker application. The directive COMPSS_MASTER or COMPSS_WORKER must be used in order to compile the source files for each type of library. Both libraries will be copied into the lib directory where the binding will look for them when generating the master and worker applications.

Class Serialization

In case of using an object as method parameter, as callee or as return of a call to a function, the object has to be serialized. The serialization method has to be provided inline in the header file of the object's class by means of the "**boost**" library. The next listing contains an example of serialization for two objects of the Block class.

```
#ifndef BLOCK_H
```

```
#define BLOCK_H

#include    <vector>
#include    <boost/archive/text_iarchive.hpp>
#include    <boost/archive/text_oarchive.hpp>
#include    <boost/serialization/serialization.hpp>
#include    <boost/serialization/access.hpp>
#include    <boost/serialization/vector.hpp>

using namespace std;
using namespace boost;
using namespace serialization;

class Block {
public:
    Block(){};

    Block(int bSize);

    static Block *init(int bSize, double initVal);

    void multiply(Block block1, Block block2);

    void print();

private:
    int M;
    std::vector< std::vector< double > > data;

    friend class::serialization::access;
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version) {
        ar & M;
        ar & data;
    }
};

#endif
```

For more information about serialization using "boost" visit the related documentation at `www.boost.org`.

Method - Task

A task can be a C++ class method. A method can return a value, modify the *this* object, or modify a parameter.

If the method has a return value there will be an implicit synchronization before exit the method, but for the *this* object and parameters the synchronization can be done later after the method has finished.

This is because the *this* object and the parameters can be accessed inside and outside the method, but for the variable where the returned value is copied to, it can't be known inside the method.

```
#include "Block.h"

Block::Block(int bSize) {
       M = bSize;
       data.resize(M);
       for (int i=0; i<M; i++) {
              data[i].resize(M);
       }
```

```
}

Block *Block::init(int bSize, double initVal) {
       Block *block = new Block(bSize);
       for (int i=0; i<bSize; i++) {
              for (int j=0; j<bSize; j++) {
                     block->data[i][j] = initVal;
              }
       }
       return block;
}

#ifdef COMPSS_WORKER

void Block::multiply(Block block1, Block block2) {
       for (int i=0; i<M; i++) {
              for (int j=0; j<M; j++) {
                     for (int k=0; k<M; k++) {
                            data[i][j] += block1.data[i][k] * block2.data[k][j];
                     }
              }
       }
       this->print();
}

#endif

void Block::print() {
       for (int i=0; i<M; i++) {
              for (int j=0; j<M; j++) {
                     cout << data[i][j] << " ";
              }
              cout << "\r\n";
       }
}
```

Task Constraints

The C/C++ binding also supports the definition of task constraints. The task definition specified in the IDL file must be decorated/annotated with the *@Constraints*. Below, you can find and example of how to define a task with a constraint of using 4 cores. The list of constraints which can be defined for a task can be found in Section 5

```
interface Matmul
{
      @Constraints(ComputingUnits = 4
      void multiplyBlocks(inout Block block1,
                          in Block block2,
                          in Block block3);

};
```

Task Versions

Another COMPSs functionality supported in the C/C++ binding is the definition of different versions for a tasks. The following code shows an IDL file where a function has two implementations, with their corresponding constraints. It show an example where

the *multiplyBlocks_GPU* is defined as a implementation of *multiplyBlocks* using the annotation/decoration *@Implements*. It also shows how to set a processor constraint which requires a GPU processor and a CPU core for managing the offloading of the computation to the GPU.

```
interface Matmul
{
        @Constraints(ComputingUnits=4);
        void multiplyBlocks(inout Block block1,
                            in Block block2,
                            in Block block3);

        // GPU implementation
        @Constraints(processors={
               @Processor(ProcessorType=CPU, ComputingUnits=1)});
               @Processor(ProcessorType=GPU, ComputingUnits=1)});
        @Implements(multiplyBlocks);
        void multiplyBlocks_GPU(inout Block block1,
                                in Block block2,
                                in Block block3);

};
```

Application Compilation

To compile the user application with the C/C++ binding the "**buildapp**" command the user has to be executed in the directory of the main application code; the name of the application has to be passed as argument to this script, in this case *Matmul*.

```
user@localhost:~/matmul_objects$ buildapp Matmul

Building application...

g++ -DCOMPSS_MASTER -g -I. -I/opt/COMPSs/Runtime/bindings/c/include -I/opt/COMPSs/Runtime/bindings/bindings
     -common/include -c Block.cc Matrix.cc ar rvs libmaster.a Block.o Matrix.o

g++ -DCOMPSS_WORKER -g -I. -I/opt/COMPSs/Runtime/bindings/c/include -I/opt/COMPSs/Runtime/bindings/bindings
     -common/include -c Block.cc Matrix.cc ar rvs libworker.a Block.o Matrix.o

Building all:

Building Master...

g++ -g -O2 -o Matmul Matmul-empty.o Matmul-stubs.o Matmul.o -L../../lib -lmaster -L/usr/lib/jvm/java-6-
     openjdk-amd64/jre/lib/amd64/server -ljvm -ldl -L/opt/COMPSs/Runtime/bindings/c/../bindings-common/lib
     -lbindings_common -L/opt/COMPSs/Runtime/bindings/c/lib -lcbindings -lboost_iostreams -
     lboost_serialization

Building Worker...

g++ -g -O2 -o Matmul-worker Matmul-worker.o Matmul-functions.o -L../../lib -lworker -ldl -lboost_iostreams
     -lboost_serialization -L/opt/COMPSs/Runtime/bindings/c/lib

Command succesful.
```

[The previous output has been cut for simplicity]

Application Execution

The following environment variables must be defined before executing a COMPSs C/C++ application:

JAVA_HOME: Java JDK installation directory (e.g. /usr/lib/jvm/java-8-openjdk/)

After compiling the application, two directories, master and worker, are generated. The master directory contains a binary called as the main file, which is the master application, in our example is called Matmul. The worker directory contains another binary called as the main file followed by the suffix "-worker", which is the worker application, in our example is called Matmul-worker.

The *runcompss* script has to be used to run the application:

```
compss@bsc:~$ runcompss \
                --lang=c \
                -g \
                /home/compss/workspace_c/matmul_objects/master/Matmul 3 4 2.0
```

The completelist of options of the runcompss command is available in the *COMPSs User Manual: Application Execution* at `http://compss.bsc.es` .

Execution Graph

Figure 1 depicts the execution graph for the Matmul application in its object version with 3x3 blocks matrices, each one containing a 4x4 matrix of doubles. Each block in the result matrix accumulates three block multiplications, i.e. three multiplications of 4x4 matrices of doubles.

The light blue circle corresponds to the initialization of matrix "A" by means of a method-task and it has an implicit synchronization inside. The dark blue circles correspond to the other two initializations by means of function-tasks; in this case the synchronizations are explicit and must be provided by the developer after the task call. Both implicit and explicit synchronizations are represented as red circles.

Each green circle is a partial matrix multiplication of a set of 3. One block from matrix "A" and the correspondent one from matrix "B". The result is written in the right block in "C" that accumulates the partial block multiplications. Each multiplication set has an explicit synchronization. All green tasks are method-tasks and they are executed in parallel.

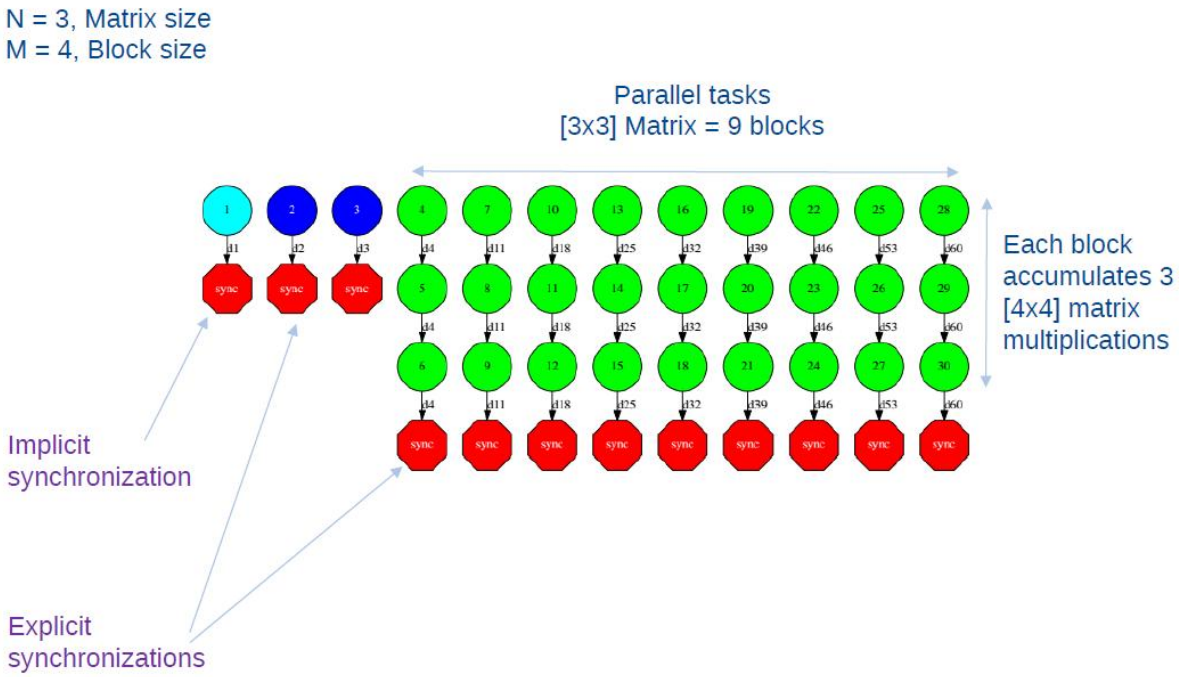

Figure 1: Matmul Execution Graph.

Constraints

This section provides a detailed information about all the supported constraints by the COMPSs runtime for **java**, **python** and **C/C++** languages. The constraints are defined as key-value pairs, where the key is the name of the constraint. Table 3 details the available constraints names for *Java*, *Python* and *C/C++*, its value type, its default value and a brief description.

Java	Python / C / C++	Value type	Default value	Description
computingUnits	ComputingUnits	<string>	¨1¨	Required number of computing units
processorName	ProcessorName	<string>	¨[unassigned]¨	Required processor name
processorSpeed	ProcessorSpeed	<string>	¨[unassigned]¨	Required processor speed
processorArchitecture	ProcessorArchitecture	<string>	¨[unassigned]¨	Required processor architecture
processorType	ProcessorType	<string>	¨[unassigned]¨	Required processor type
processorPropertyName	ProcessorPropertyName	<string>	¨[unassigned]¨	Required processor property
processorPropertyValue	ProcessorPropertyValue	<string>	¨[unassigned]¨	Required processor property value
processorInternalMemorySize	ProcessorInternalMemorySize	<string>	¨[unassigned]¨	Required internal device memory
processors	-	List<@Processor>	¨{}¨	Required processors (check table 4 for Processor details)
memorySize	MemorySize	<string>	¨[unassigned]¨	Required memory size in GBs
memoryType	MemoryType	<string>	¨[unassigned]¨	Required memory type (SRAM, DRAM, etc.)
storageSize	StorageSize	<string>	¨[unassigned]¨	Required storage size in GBs
storageType	StorageType	<string>	¨[unassigned]¨	Required storage type (HDD, SSD, etc.)
operatingSystemType	OperatingSystemType	<string>	¨[unassigned]¨	Required operating system type (Windows, MacOS, Linux, etc.)
operatingSystemDistribution	OperatingSystemDistribution	<string>	¨[unassigned]¨	Required operating system distribution (XP, Sierra, openSUSE, etc.)
operatingSystemVersion	OperatingSystemVersion	<string>	¨[unassigned]¨	Required operating system version
wallClockLimit	WallClockLimit	<string>	¨[unassigned]¨	Maximum wall clock time
hostQueues	HostQueues	<string>	¨[unassigned]¨	Required queues
appSoftware	AppSoftware	<string>	¨[unassigned]¨	Required applications that must be available within the remote node for the task

Table 3: Arguments of the *@constraint* decorator

All constraints are defined with a simple value except the *HostQueue* and *AppSoftware* constraints, which allow multiple values.

The *processors* constraint allows the users to define multiple processors for a task execution. This constraint is specified as a list of @Processor annotations that must be defined as shown in table 4

Annotation	**Value type**	**Default value**	**Description**
computingUnits	<string>	¨1¨	Required number of computing units
name	<string>	¨[unassigned]¨	Required processor name
speed	<string>	¨[unassigned]¨	Required processor speed
architecture	<string>	¨[unassigned]¨	Required processor architecture
type	<string>	¨[unassigned]¨	Required processor type
propertyName	<string>	¨[unassigned]¨	Required processor property
propertyValue	<string>	¨[unassigned]¨	Required processor property value
internalMemorySize	<string>	¨[unassigned]¨	Required internal device memory

Table 4: Arguments of the *@Processor* decorator

Known Limitations

The current COMPSs version (2.0.rc1704) has the following limitations:

- **Exceptions:**
  The current COMPSs version is not able to propagate exceptions raised from a task to the master. However, the runtime catches any exception and sets the task as failed.

- **Java tasks:**
  Java tasks **must** be declared as **public**. Despite the fact that tasks can be defined in the main class or in other ones, we recommend to define the tasks in a separated class from the main method to force its public declaration.

- **Java objects:**
  Objects used by tasks must follow the *java beans* model (implementing an empty constructor and getters and setters for each attribute) or implement the *serializable* interface. This is due to the fact that objects will be transferred to remote machines to execute the tasks.

- **Java object aliasing:**
  If a task has an object parameter and returns an object, the returned value must be a new object (or a cloned one) to prevent any aliasing with the task parameters.

```
// @Method(declaringClass = "...")
// DummyObject incorrectTask (
//    @Parameter(type = Type.OBJECT, direction = Direction.IN) DummyObject a,
//    @Parameter(type = Type.OBJECT, direction = Direction.IN) DummyObject b
// );
public DummyObject incorrectTask (DummyObject a, DummyObject b) {
    if (a.getValue() > b.getValue()) {
        return a;
    }
    return b;
}

// @Method(declaringClass = "...")
// DummyObject correctTask (
//    @Parameter(type = Type.OBJECT, direction = Direction.IN) DummyObject a,
//    @Parameter(type = Type.OBJECT, direction = Direction.IN) DummyObject b
// );
public DummyObject correctTask (DummyObject a, DummyObject b) {
    if (a.getValue() > b.getValue()) {
        return a.clone();
    }
    return b.clone();
}

public static void main() {
    DummyObject a1 = new DummyObject();
    DummyObject b1 = new DummyObject();
    DummyObject c1 = new DummyObject();
    c1 = incorrectTask(a1, b1);
    System.out.println("Initial value: " + c1.getValue());
    a1.modify();
    b1.modify();
    System.out.println("Aliased value: " + c1.getValue());
```

```
    DummyObject a2 = new DummyObject();
    DummyObject b2 = new DummyObject();
    DummyObject c2 = new DummyObject();
    c2 = incorrectTask(a2, b2);
    System.out.println("Initial value: " + c2.getValue());
    a2.modify();
    b2.modify();
    System.out.println("Non-aliased value: " + c2.getValue());
}
```

- **Services types:**
  The current COMPSs version only supports SOAP based services that implement the WS interoperability standard. REST services are not supported.

- **Use of file paths:**
  The persistent workers implementation has a unique *Working Directory* per worker. That means that tasks should not use hardcoded file names to avoid file collisions and tasks misbehaviours. We recommend to use files declared as task parameters, or to manually create a sandbox inside each task execution and/or to generate temporary random file names.

- **Python constraints in the cloud:**
  When using python applications with constraints in the cloud the minimum number of VMs must be set to 0 because the initial VM creation doesn't respect the tasks contraints. Notice that if no contraints are defined the initial VMs are still usable.

- **Intermediate files**:
  Some applications may generate intermediate files that are only used among tasks and are never needed inside the master's code. However, COMPSs will transfer back these files to the master node at the end of the execution. Currently, the only way to avoid transferring these intermediate files is to manually erase them at the end of the master's code. Users must take into account that this only applies for files declared as task parameters and **not** for files created and/or erased inside a task.

- **Python object hierarchy dependency detection**:
  Dependencies are detected only on the objects that are task parameters or outputs. Consider the following code:

```
# a.py
class A:
  def __init__(self, b):
    self.b  = b
# main.py
from a import A
from pycompss.api.task import task
from pycompss.api.parameter import *

@task(obj = IN, returns = int)
def get_b(obj):
  return obj.b

@task(obj = INOUT)
def inc(obj):
  obj += [1]
```

```
def main():
  from pycompss.api.api import compss_wait_on
  my_a = A([5])
  inc(my_a.b)
  obj = get_b(my_a)
  obj = compss_wait_on(obj)
  print obj

if __name__ == '__main__':
  main()
```

Note that there should exist a dependency between `A` and `A.b`. However, PyCOMPSs is not capable to detect dependencies of that kind because it doesn't do any kind of introspection. This kind of dependencies must be handled (and avoided) manually.

- **Python static methods**:
  PyCOMPSs is not capable to resolve static methods (i.e: those that have the decorator `@staticmethod`). It is recommended to use module functions instead.

- **Python modules with global states**:
  Some modules (for example `logging`) have internal variables apart from functions. These modules are not guaranteed to work in PyCOMPSs due to the fact that master and worker code are executed in different interpreters. For instance, if a `logging` configuration is set on some worker, it will not be visible from the master interpreter instance.

- **Python global variables**:
  This issue is very similar to the previous one. PyCOMPSs does not guarantee that applications that create or modify global variables while worker code is executed will work. In particular, this issue (and the previous one) is due to Python's Global Interpreter Lock (GIL).

- **Python application directory as a module**:
  If the Python application root folder is a python module (i.e: it contains an `__init__.py` file) then `runcompss` must be called from the parent folder. For example, if the Python application is in a folder with an `__init__.py` file named `my_folder` then PyCOMPSs will resolve all functions, classes and variables as `my_folder.object_name` instead of `object_name`. For example, consider the following file tree:

```
my_apps/
|- kmeans/
    |- __init__.py
    |- kmeans.py
```

Then the correct command to call this app is `runcompss kmeans/kmeans.py` from the `my_apps` directory.

- **Python early program exit**:
  All intentional, premature exit operations must be done with `sys.exit`. PyCOMPSs

needs to perform some cleanup tasks before exiting and, if an early exit is performed with `sys.exit`, the event will be captured, allowing PyCOMPSs to perform these tasks. If the exit operation is done in a different way then there is no guarantee that the application will end properly.

Please find more details on the COMPSs framework at

http://compss.bsc.es