COMP Superscalar

User Manual

Application development guide

Version: 1.3

October 19, 2015

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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

For full COMPSs application examples (codes, execution commands, results, logs, etc.) please refer to the *COMPSs Sample Applications* available 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

List of Figures

List of Tables

1 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. Instead, the model is based on sequential programming, which makes it appealing to users that either lack parallel programming expertise or are looking for better programmability.

- **Infrastructure unaware:** COMPSs offers a model that abstracts the application from the underlying distributed 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 is based on the popular programming language Java, but also offers language bindings for Python and C/C++ applications. This facilitates the learning of the model, since programmers can reuse most of their previous knowledge.

- **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 pure standard Java syntax and libraries. With regard the Python and C/C++ bindings, a small set of API calls should be used on the COMPSs applications.

2 Java

In this section the steps to develop a Java COMPSs application will be illustrated. The sequential *Simple application* will be used to explain an application porting to COMPSs. The user is required to select a set of methods, invoked in a sequential application, to be run as remote tasks on the available resources.

2.1 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 on the Cloud.
- **Remote methods code:** the implementation of the remote tasks.
- **Java annotated interface:** It declares the selected methods to be run as remote tasks and metadata used to schedule the tasks.

The main application code (sequential) will have the name of the application, always starting with capital letter, in this case will be **Simple.java**. The Java annotated interface will be named as *application name+Itf.java* in this case will be **SimpleItf.java**. And the code that implements the remote tasks will be called as *application name + Impl.java*, in this case will be **SimpleImpl.java**.

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

2.1.1 Main application code

In COMPSs the application is kept completely unchanged, i.e. 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 from the main application code.

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**) will be executed in a sequential way except the **increment()** method. COMPSs, as mentioned above, will replace at execution time the call to this method generating a remote task on the remote 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();
    }
  }
}
```

2.1.2 Remote methods code

The following code is 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();
    }
  }
}
```

2.1.3 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 fulfil to execute the method, such as the number of processors or main memory size.

Here follows a complete and detailed explanation of the usage of the metadata:

- **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.
    - **priority** True if the task is prioritary and false otherwise. This parameter is applyed on the schedule policies.
  - **@Constraints:** Mandatory. The user can specify the capabilities that a resource must have in order to run a method. The COMPSs runtime will create a VM (in a cloud environment), that fits the specified requirements in order to perform the execution.
    - Processor:
      - **processorArchitecture** Required processor architecture
Default: "[unassigned]"
      - **processorCPUCount** Required number of CPUs for the host
Default: 0
      - **processorCoreCount** Required number of Cores
Default: 1
      - **processorSpeed** Required processor frequency
Default: 0.0f
    - Memory:
      - **memoryPhysicalSize** Required physical memory size in GBs
Default: 0.0f
      - **memoryVirtualSize** Required virtual memory size in GBs
Default: 0.0f
      - **memoryAccessTime** Top memory access time in nanoseconds
Default: 0.0f
      - **memorySTR Minimal** Memory bandwith in GB/s
Default: 0.0f

* Storage:
  * **storageElemSize** Amount of required storage space in GB
    Default: 0.0f
  * **storageElemAccessTime** Top access time to the storage system in milliseconds
    Default: 0.0f
  * **storageElemSTR Minimal** Storage bandwith in MB/s
    Default: 0.0f
* Miscellaneous:
  * **operatingSystemType** Required operative system
    Default: "[unassigned]"
  * **hostQueue** Required queues
    Default: "[unassigned]"
  * **appSoftware** Required applications
    Default: "[unassigned]"

- **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, a basic type or a generic class object.
  - **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 is prioritary and false otherwise. This parameter is applyed on the schedule policies.

The Java annotated interface of the Simple app example (SimpleItf.java) declares the *Increment()* method that will be executed remotely. The method implementation can be found in simple.SimpleImpl class and needs a single input parameter, a string containing a path to the file counterFile. Besides, 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.Method;
import  integratedtoolkit.types.annotations.Parameter;
import  integratedtoolkit.types.annotations.Parameter.Direction;
import  integratedtoolkit.types.annotations.Parameter.Type;

public interface SimpleItf {

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

}
```

2.1.4 Equivalent remote methods

Since version 1.2, the COMPSs programming model allows developers to define sets of equivalent remote methods. Thus, an invocation to any of the methods in the set might produce the execution of another method of the set in the remote resource.

The coding of the application does not change, the remote methods are implemented as regular Java methods and the main code of the application is a sequential code that contains calls to these methods. The only component of the application that changes for defining a set of equivalent methods is the Java annotated interface.

The programming model considers all the equivalent methods of a set as different implementations of the same method. Therefore, 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 an array of declaring classes for the method.

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",
                           "packagepath.Quicksort" })

void sort(
    @Parameter(type = Type.OBJECT, direction = Direction.INOUT)
    int[] array
```

```
);
```

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 might have specific constraints. Through the @Constraints annotation, developers can specify the common constraints for a whole set of methods. The following example states that for both sorting algorithms only one core is required to run the method.

```
@Constraints(processorCoreCount = 1)
@Method(declaringClass = { "packagepath.Mergesort",
                           "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 @Multiconstraints annotation containing the specific constraints for each 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(processorCoreCount = 1)
@MultiConstraints({
        @Constraints(memoryPhysicalSize= (float)2.0),
        @Constraints(memoryPhysicalSize= (float)0.5)})
@Method(declaringClass = { "packagepath.Mergesort",
                           "packagepath.Quicksort"})
void sort(
    @Parameter(type = Type.OBJECT, direction = Direction.INOUT)
    int[] array
);
```

2.2 Application Compilation

A java application needs to be packaged in a *jar* file containing the source files and 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/simple/
compss@bsc:~/workspace_java/simple/src/simple$ javac *.java
compss@bsc:~/workspace_java/simple/src/simple$ jar cf simple.jar simple
```

For a successfull compilation the *compss-engine.jar* package must be provided by the CLASSPATH variable. The default COMPSs installation automatically inserts this package into the CLASSPATH but some users may have overwriten or deleted it. Please check that your environment variable CLASSPATH containts 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.

The easiest solution comes up by manually exporting the CLASSPATH variable into the user session:

```
$ export CLASSPATH=$CLASSPATH:/opt/COMPSs/Runtime/compss-engine.jar
```

However, you will need to remember to export this variable every time you log out and back in again. Consequently, we recomend to add this export to the *.bashrc* file:

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

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

Attention: The *compss-engine.jar* is installed inside the COMPSs installation directory. If you have performed a custom installation, the path of the package can be different.

To make things even easier, we have developed a COMPSs IDE which is an Integrated Development Environment to develop, compile, deploy and execute COMPSs applications. For further information about the *COMPSs IDE* please check the *COMPSs IDE User Guide* available at our webpage: `http://compss.bsc.es` .

2.3 Application Execution

In order to run a Java application with COMPSs, the script runcompss can be used. An example of an invocation of the script is:

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

For full description about the options available for the runcompss command please check the *COMPSs User Manual: Application Execution* available at our webpage `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 not-Java application with the COMPSs Java runtime, providing the necessary language translation.

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

3 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.

3.1 Programming Model

3.1.1 Task Selection

Like in the case of the Java language, 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.

Regarding task selection, in Python it is not done by means of an annotated interface but with the use of Python decorators. 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 *fi*, 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
```

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. 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.

```python
@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.

3.1.2 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 two API functions that need 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 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()*).
- *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.

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.

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

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)
...
```

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 and returns it.

Table 2: COMPSs Python API functions.

3.1.2.1 Future Objects

If the programmer selects 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, COMPSs manages future objects: a representant object is immediately returned to the main program when a task is invoked.

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

...

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

The future 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 future 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 future object from the main program, the programmer proceeds 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)
```

The future object mechanism is applied to primitive types, strings and objects (including the Python built-in types list, dictionary and tuple).

It is important to note that, 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()
```

3.1.3 Important Notes

For the COMPSs Python binding to function correctly, the programmer should not use relative imports in her 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
```

3.2 Application Execution

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

3.2.1 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/*)

3.2.2 Command

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

```
compss@bsc:~$ runcompss \
                --lang=python \
                --classpath=$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 our webpage `http://compss.bsc.es` .

4 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.

4.1 Programming Model

4.1.1 Task Selection

As in Java language the user must write the task selection like 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 as follows:

```
["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>
```

4.1.2 Value and Object return

Notice that returning a value or an object is now supported, this means a "void" value, a value of a primitive type (an int, long, float, etc.), or an object of a class, can be returned from a function or method.

IMPORTANT:

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 binding can't know such reference before leaving 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 de & operator [foo(&A);].

4.1.3 Main Program

The main program is a sequential code written in C++ that launches tasks to be executed in parallel and may have several data-synchronization points or none.

```
#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 main points when programming the main code are:

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 certain type-definitions that are needed.
3. A call to the **compss_on** binding function to turn on the COMPSs runtime.
4. As in C language, when passing and out or inout parameter, the memory address should be passed in order the parameter can be modified. For that, in a task-function call, we use 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 to turn off the COMPSs runtime.

4.1.4 Functions file

The function file is where the programmer writes the implementation of the tasks in a C or C++ function style. 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);
}
```

There is no special consideration when writing the functions (or tasks), only to include the Matmul.h header file is needed.

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 that we have mentioned before.

4.1.5 Other Source Files

The user application for sure will have other source files than the main-file and the functions-file seen in the previous sections. This other files 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 executes the Makefile.

The previous code shows an example of a 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.

4.1.6 Class Serialization

As known, the C++ classes can be written in separated files, a header file for the declaration and a source file for the implementation.

Here we show the class Block. The important thing here is that the class must provide the method for the object serialization. This serialization is done using the "**boost**" library. The "**serialize**" method is implemented inline in the header file.

```
#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`.

4.1.7 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 due to the “this” object and parameters can be accessed inside the method and outside, 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";
        }
}
```

4.2 Application Compilation

In order to compile the user application against the binding libraries in a proper way a script is accessible in the system path after the binding installation.

In the same directory where the main file resides, just execute the command "**buildapp**" and pass the name of the application 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]

4.3 Application Execution

4.3.1 Environment

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-7-openjdk/)

4.3.2 Command

After compiling the application, two directories are generated, the master and the worker directories. 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.

In order to run the whole application, master and worker applications, use the *runcompss* script that can be found in the system path after the runtime installation. Here is the example of the command execution for the Matmul application.

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

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

4.4 Matmul Execution Graph

This is the execution graph for the matmul application in its object version with 3x3 matrices of blocks. That means a total of 9 blocks where each block is another 4x4 matrix of doubles.

Each block in the result matrix accumulates three block multiplications, in other words, three multiplications of 4x4 matrices of doubles.

The light blue circle corresponds to the initialization of matrix "A" by mean of a method-task and it has an implicit synchronization inside. The dark blue circles correspond to the other two initializations by mean of function-tasks, the synchronizations are explicit in this case, the user has written them after the task call. Both implicit and explicit synchronizations appear in a red circle.

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.

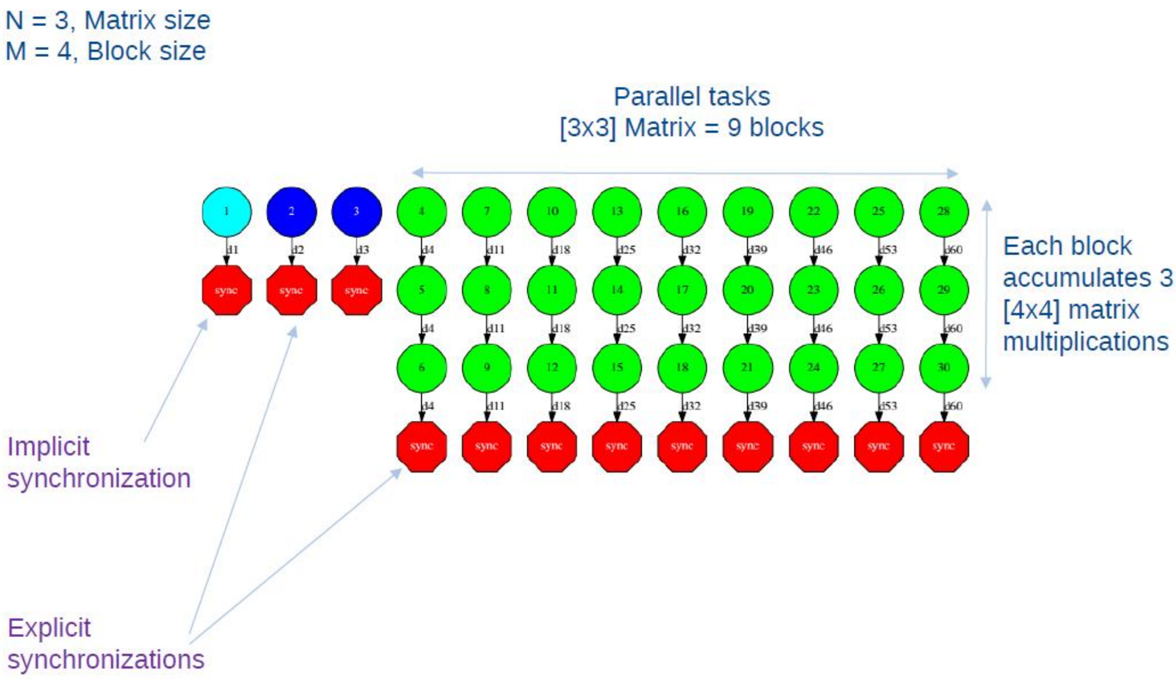

Figure 1: Matmul Execution Graph.

5 Known Limitations

Users must take into account that COMPSs is an on-going project. The *Workflows and Distributed Computing* group at the *BSC-CNS* is doing its best to maintain and enhance COMPSs; granting all its users a better support in each new COMPSs release.

Consequently, COMPSs is beeing continously developed and has several on-going projects. We are aware that the current COMPSs version (1.3) has some limitations and we are working to solve them as soon as possible. Next, we provide a list of the major known limitations:

- **Exceptions:**
  The current COMPSs version is not able to catch exceptions raised from a task.

- **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 transfered to remote machines to execute the tasks.

- **Services types:**
  The current COMPSs version only supports SOAP based services that implement the WS interoperability standard. Supporting Big Services is a feature we are **not** planning to implment in the near future.

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

- **Tracing:**
  The current version of the COMPSs tracing system slows down the application execution. Users running huge applications may experience a non-negligible overhead when using this feature.

- **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 transfering 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.

- **NIO Workers cache**:
  Persistent workers have a cache to avoid transfering files and objects repeatedly. However, this cache is **not** working for INOUT parameters (only works for IN and OUT parameters). For most applications, if users are willing to exploit this cache, we recommend to convert INOUT parameters in **two** separated parameters: one declared as IN parameter and the other declared as OUT parameter.

If you are willing to know the COMPSs development state you can check out our webpage `http://compss.bsc.es` or contact us at `support-compss@bsc.es` .

Please find more details on the COMPSs framework at

http://compss.bsc.es