COMP Superscalar

COMPSs Sample Applications

Version: 1.3

October 19, 2015

This manual only provides Java, Python and C/C++ sample applications that can be used as a reference. Specifically, it details the applications' code and different ways to execute and analyse them (by using different runcompss flags).

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

For further information about the application development please refer to the *COMPSs User Manual: Application development 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 Sample applications

The first two examples in this section are simple applications developed in COMPSs to easily illustrate how to code, compile and run COMPSs applications. These applications are executed locally and show different ways to take advantge of all the COMPSs features.

The rest of the examples are more elaborated and consider the execution in a cloud platform where the VMs mount a common storage on **/sharedDisk** directory. This is useful in the case of applications that require working with big files, allowing to transfer data only once, at the beginning of the execution, and to enable the application to access the data directly during the rest of the execution.

The Virtual Machine available at our webpage (`http://compss.bsc.es/`) provides a development environment with all the applications listed in the following sections. The codes of all the applications can be found under the */home/compss/workspace_java/* folder.

2.1 Hello World

The Hello Wolrd is a Java application that creates a task and prints a Hello World! message. Its purpose is to clarify that the COMPSs tasks output is redirected to the job files and it is **not** available at the standard output.

Next we provide the important parts of the application's code.

```
// hello.Hello

public static void main(String[] args) throws Exception {
  // Check and get parameters
  if (args.length != 0) {
    usage();
    throw new Exception("[ERROR] Incorrect number of parameters");
  }

  // Hello World from main application
  System.out.println("Hello World! (from main application)");

  // Hello World from a task
  HelloImpl.sayHello();
}
```

As shown in the main code, this application has no input arguments.

```
// hello.HelloImpl

public static void sayHello() {
  System.out.println("Hello World! (from a task)");
}
```

Remember that, to run with COMPSs, java applications must provide an interface. For simplicity, in this example, the content of the interface only declares the task which has no parameters:

```
// hello.HelloItf
```

```
  @Method(declaringClass = "hello.HelloImpl")
  void sayHello(
  );
```

Notice that there is a first Hello World message printed from the main code and, a second one, printed inside a task. When executing sequentially this application users will be able to see both messages at the standard output. However, when executing this application with COMPSs, users will only see the message from the main code at the standard output. The message printed from the task will be stored inside the job log files.

Let's try it. First we proceed to compile the code by running the following instructions:

```
compss@bsc:~$ cd ~/workspace_java/hello/src/
compss@bsc:~$ javac *.java
```

Once done, we can sequentially execute the application by directly invoking the *jar* file.

```
compss@bsc:~$ java -jar hello.jar
```

And we can also execute the application with COMPSs:

```
compss@bsc:~$ cd ~/workspace_java/hello/jar/
compss@bsc:~/workspace_java/hello/jar$ runcompss -d hello.Hello
```

Notice that the COMPSs execution is using the *-d* option to allow the job logging. Thus, we can check out the application jobs folder to look for the task output.

```
compss@bsc:~$ cd ~/.COMPSs/hello.Hello_01/jobs/
compss@bsc:~/.COMPSs/hello.Hello_01/jobs$ ls -1

compss@bsc:~/.COMPSs/hello.Hello_01/jobs$ cat job1_NEW.out
```

2.2 Simple

The Simple application is a Java application that increases a counter by means of a task. The counter is stored inside a file.

2.3 Increment

The Increment application is a Java application that increases N time three different counters. Each increase step is developed by a separated task. The purpose of this application is to show parallelism between the three counters.

Next we provide the main code of this application. The code inside the *increment* task is the same than the previous example.

```
// increment.Increment
```

As shown in the main code, this application has 4 parameters that stand for:

1. **N:** Number of times to increase a counter
2. **InitialValue1:** Initial value for counter 1
3. **InitialValue2:** Initial value for counter 2
4. **InitialValue3:** Initial value for counter 3

Next we will compile and run the Increment application with the *-g* option to be able to generate the final graph at the end of the execution.

```
compss@bsc:~$ cd ~/workspace_java/increment/jar/
compss@bsc:~/workspace_java/increment/jar$ runcompss -g increment.Increment 10 1 2 3

compss@bsc:~/workspace_java/increment/jar$ cd ~/.COMPSs/increment.Increment_01/monitor/
compss@bsc:~/.COMPSs/increment.Increment_01/monitor$ gengraph complete_graph.dot
compss@bsc:~/.COMPSs/increment.Increment_01/monitor$ evince complete_graph.dot.pdf
```

2.4 Matrix multiplication

The Matrix Multiplication (Matmul) is a pure Java application that multiplies two matrices in a direct way. The application creates 2 matrices of N x N size initialized with values, and multiply the matrices by blocks.

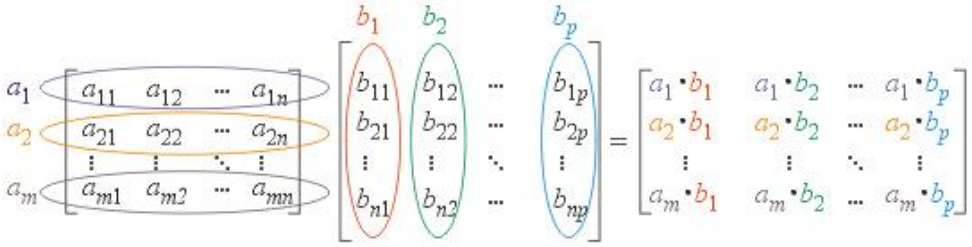

Figure 1: Matrix multiplication

In this application the multiplication is implemented in the multiplyAccumulative that is thus selected as the task that will be executed remotely. In order to run the application the matrix dimension (number of blocks) and the dimension of each block have to be supplied.

```
compss@bsc:~$ cp ~/workspace_java/matmul/package/Matmul.tar.gz /home/compss/
compss@bsc:~$ tar xzf Matmul.tar.gz
```

The command line to execute the application:

```
compss@bsc:~$ runcompss matmul.Matmul <matrix_dim> <block_dim>
```

2.5 Sparse LU decomposition

SparseLU multiplies two matrices using the factorization method of LU decomposition, which factorizes a matrix as a product of a lower triangular matrix and an upper one.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Figure 2: Sparse LU decomposition

The matrix is divided into N x N blocks on where 4 types of operations will be applied modifying the blocks: **lu0**, **fwd**, **bdiv** and **bmod**. These four operations are implemented in four methods that are selecetd as the tasks that will be executed remotely. In order to run the application the matrix dimension has to be provided.

```
compss@bsc:~$ cp ~/workspace/sparselu/package/SparseLU.tar.gz /home/compss/
compss@bsc:~$ tar xzf SparseLU.tar.gz
```

The command line to execute the application:

```
compss@bsc:~$ runcompss sparselu.SparseLU <matrix_dim>
```

2.6 KMeans

2.7 BLAST Workflow

BLAST is a widely-used bioinformatics tool for comparing primary biological sequence information, such as the amino-acid sequences of different proteins or the nucleotides of DNA sequences with sequence databases, identifying sequences that resemble the query sequence above a certain threshold. The work performed by the COMPSs Blast workflow is computationally intensive and embarrassingly parallel.

The workflow describes the three blocks of the workflow implemented in the **Split**, **Align** and **Assembly** methods. The second one is the only method that is chosen to be executed remotely, so it is the unique method defined in the interface file. The **Split** method chops the query sequences file in N fragments, **Align** compares each sequence fragment against the database by means of the Blast binary, and **Assembly** combines all intermediate files into a single result file.

This application uses a database that will be on the shared disk space avoiding transferring the entire database (which can be large) between the virtual machines.

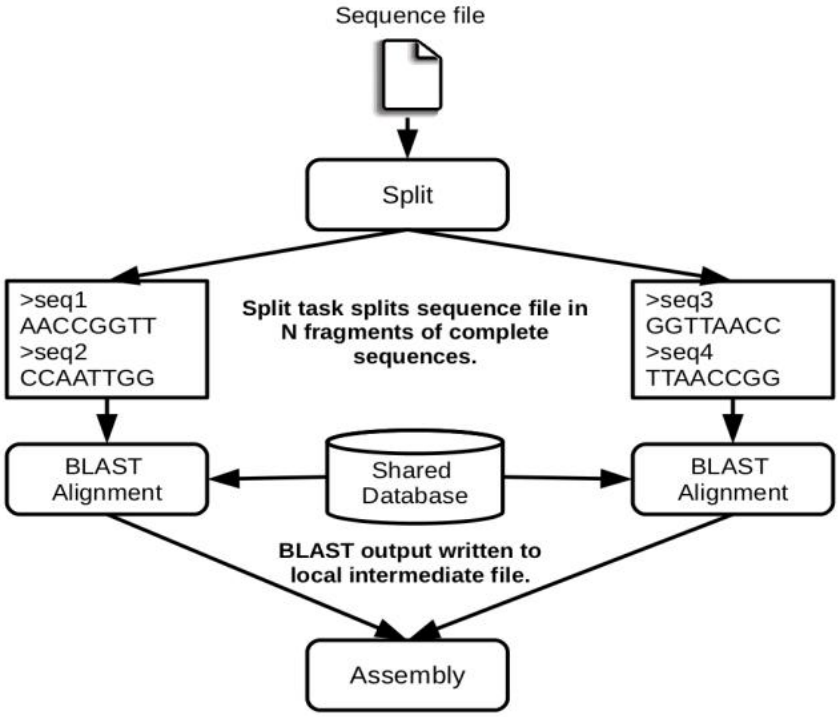

Figure 3: The COMPSs Blast workflow

```
compss@bsc:~$ cp ~/workspace/blast/package/Blast.tar.gz /home/compss/
compss@bsc:~$ tar xzf Blast.tar.gz
```

The command line to execute the workflow:

```
compss@bsc:~$ runcompss blast.Blast <debug>
                                    <bin_location>
                                    <database_file>
                                    <sequences_file>
                                    <frag_number>
                                    <tmpdir>
                                    <output_file>
```

Where:

- **debug**: The debug flag of the application (true or false).
- **bin_location**: Path of the Blast binary.
- **database_file**: Path of database file; the shared disk **/sharedDisk/** is suggested to avoid big data transfers.
- **sequences_file**: Path of sequences file.
- **frag_number**: Number of fragments of the original sequence file, this number determines the number of parallel Align tasks.

- **tmpdir**: Temporary directory (**/home/compss/tmp/**).
- **output_file**: Path of the result file.

Example:

```
compss@bsc:~$ runcompss blast.Blast true
                        /home/compss/workspace_java/blast/binary/blastall
                        /sharedDisk/Blast/databases/swissprot/swissprot
                        /sharedDisk/Blast/sequences/sargasso_test.fasta
                        4
                        /tmp/
                        /home/compss/out.txt
```

3 Python Sample applications

The first two examples in this section are simple applications developed in COMPSs to easily illustrate how to code, compile and run COMPSs applications. These applications are executed locally and show different ways to take advantge of all the COMPSs features.

The rest of the examples are more elaborated and consider the execution in a cloud platform where the VMs mount a common storage on **/sharedDisk** directory. This is useful in the case of applications that require working with big files, allowing to transfer data only once, at the beginning of the execution, and to enable the application to access the data directly during the rest of the execution.

The Virtual Machine available at our webpage (`http://compss.bsc.es/`) provides a development environment with all the applications listed in the following sections. The codes of all the applications can be found under the */home/compss/workspace_python/* folder.

3.1 Hello World

3.2 Simple

3.3 WordCount

3.4 KMeans

4 C/C++ Sample applications

The first two examples in this section are simple applications developed in COMPSs to easily illustrate how to code, compile and run COMPSs applications. These applications are executed locally and show different ways to take advantge of all the COMPSs features.

The rest of the examples are more elaborated and consider the execution in a cloud platform where the VMs mount a common storage on **/sharedDisk** directory. This is useful in the case of applications that require working with big files, allowing to transfer data only once, at the beginning of the execution, and to enable the application to access the data directly during the rest of the execution.

The Virtual Machine available at our webpage (`http://compss.bsc.es/`) provides a development environment with all the applications listed in the following sections. The codes of all the applications can be found under the */home/compss/workspace_c/* folder.

4.1 Simple

4.2 Increment

4.3 Matmul

Please find more details on the COMPSs framework at

http://compss.bsc.es