COMP Superscalar

User Manual

Application execution guide

Version: 1.4

May 30, 2016

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

This manual provides information about how to execute COMPSs applications, how to retrieve the results and the logs of an execution and it provides an overview of the COMPSs tools usage. It is highly recommended to test the examples described in this manual with a working COMPSs installation. For this purpose we provide a *COMPSs Virtual Machine* 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 development please refer to the *COMPSs User Manual: Application development guide* available at `http://compss.bsc.es/` .

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/` .

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. 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 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 Executing COMPSs applications

2.1 Prerequisites

Prerequisites vary depending on the application's code language: for Java applications the users need to have a **jar archive** containing all the application classes, for Python applications there are no requirements and for C/C++ applications the code must have been previously compiled by using the *buildapp* command.

For further information about how to develop COMPSs applications please refer to the *COMPSs User Manual: Application development guide* available at the `http://compss.bsc.es/` webpage.

2.2 Runcompss command

COMPSs applications are executed using the **runcompss** command:

```
compss@bsc:~$ runcompss [options] application_name [application_arguments]
```

The application name must be the fully qualified name of the application in Java, the path to the *.py* file containing the main program in Python and the path to the master binary in C/C++.

The application arguments are the ones passed as command line to main application. This parameter can be empty.

The *runcompss* command allows the users to customize a COMPSs execution by specifying different options. For clarity purposes, parameters are grouped in *Runtime configuration*, *Tools enablers* and *Advanced options*.

```
compss@bsc:~$ runcompss -h
  Runtime configuration options:
    --project=<path>                     Path to the project XML file
                                          Default: /opt/COMPSs/Runtime/configuration/
                                          xml/projects/project.xml

    --resources=<path>                   Path to the resources XML file
                                          Default: /opt/COMPSs/Runtime/configuration/
                                          xml/resources/resources.xml

    --lang=<name>                        Language of the application (java/c/python)
                                          Default: java

    --log_level=<level>, --debug, -d     Set the debug level: off | info | debug
                                          Default: off

  Tools enablers:
    --graph=<bool>, --graph, -g          Generation of the complete graph (true/false)
                                            When no value is provided it is set to true
                                            Default: false

    --tracing=<level>, --tracing, -t     Set generation of traces and/or tracing level ( [ true | basic ]
      | advanced | false)
                                            True and basic levels will produce the same traces.
                                            When no value is provided it is set to true
                                            Default: false
```

```
    --monitoring=<int>, --monitoring, -m  Period between monitoring samples
                                          (milliseconds)
                                          When no value is provided it is set to 2000
                                          Default: 0

 Advanced options:
   --comm=<path>                          Class that implements the adaptor for
                                          communications
                                          Default: integratedtoolkit.nio.master.
                                          NIOAdaptor

   --library_path=<path>                  Non-standard directories to search for
                                          libraries (e.g. Java JVM library, Python
                                          library, C binding library)
                                          Default: .

   --classpath=<path>                     Path for the application classes / modules
                                          Default: current working directory.

   --task_count=<int>                     Only for C/Python Bindings.
                                          Maximum number of different functions/methods
                                          invoked from the application that have been
                                          selected as tasks
                                          Default: 50

   --uuid=<int>                           Preset an application UUID
                                          Default: Automatic random generation

   --PyObject_serialize=<bool>            Only for Python Binding.
                                          Enable the object serialization to string
                                          when possible (true/false).
                                          Default: false
```

2.3   Running a COMPSs application

Before running COMPSs applications the application files **must** be in the **CLASSPATH**. Thus, when launching a COMPSs application, users can manually pre-set the **CLASS-PATH** environment variable or can add the *–classpath* option to the *runcompss* command.

The next three sections provide specific information for launching COMPSs applications developed in different code languages (Java, Python and C/C++). For clarity purposes we will use the *Simple* application (developed in Java, Python and C++) available in the COMPSs Virtual Machine or at `https://compss.bsc.es/projects/bar` webpage. This application takes an integer as input parameter and increases it by one unit using a task. For further details about the codes please refer to the *Sample Applications* document available at `http://compss.bsc.es` .

2.3.1   Running Java applications

A Java COMPSs application can be launched through the following command:

```
compss@bsc:~$ cd workspace_java/simple/jar/
compss@bsc:~/workspace_java/simple/jar$ runcompss simple.Simple <initial_number>
```

In this first execution we use the default value of the *–classpath* option to automatically add the jar file to the classpath (by executing runcompss in the directory which contains the jar file). However, we can explicitly do this by exporting the **CLASSPATH** variable

```
compss@bsc:~/workspace_java/simple/jar$ runcompss simple.Simple 1
Using default location for project file: /opt/COMPSs/Runtime/scripts/user/../../configuration/xml/projects/project.xml
Using default location for resources file: /opt/COMPSs/Runtime/scripts/user/../../configuration/xml/resources/resources.xml

---------------- Executing simple.Simple -------------------------

WARNING: IT Properties file is null. Setting default values
[   API]  -  Starting COMPSs Runtime v1.3 (build 20150820-1617.rnull)
Initial counter value is 1
Final counter value is 2
[   API]  -  No more tasks for app 1
[   API]  -  Getting Result Files 1
[   API]  -  Execution Finished

----------------------------------------------------------
```

Figure 1: Execution of a Java COMPSs application.

or by providing the –classpath value. Next, we provide two more ways to perform the same execution:

```
compss@bsc:~$ export CLASSPATH=$CLASSPATH:
                            /home/compss/workspace_java/simple/jar/simple.jar
compss@bsc:~$ runcompss simple.Simple <initial_number>
```

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

2.3.2 Running Python applications

To launch a COMPSs Python application users have to provide the *–lang=python* option to the runcompss command.

```
compss@bsc:~$ cd workspace_python/simple/
compss@bsc:~/workspace_python/simple$ runcompss --lang=python simple.py <initial_number>
```

```
compss@bsc:~/workspace_python/simple$ runcompss --lang=python simple.py 1
Using default location for project file: /opt/COMPSs/Runtime/scripts/user/../../configuration/xml/projects/project.xml
Using default location for resources file: /opt/COMPSs/Runtime/scripts/user/../../configuration/xml/resources/resources.xml

---------------- Executing simple.py -------------------------

WARNING: IT Properties file is null. Setting default values
[   API]  -  Starting COMPSs Runtime v1.3 (build 20150820-1617.rnull)
Initial counter value is 1
Final counter value is 2
[   API]  -  No more tasks for app 0
[   API]  -  Getting Result Files 0
[   API]  -  Execution Finished

----------------------------------------------------------
```

Figure 2: Execution of a Python COMPSs application.

2.3.3 Running C/C++ applications

To launch a COMPSs C/C++ application users have to compile the C/C++ application by means of the *buildapp* command. For further information please refer to the *COMPSs*

User Manual: Application development guide document available at `http://compss.bsc.es` . Once complied, the *–lang=c* option must be provided to the runcompss command.

```
compss@bsc:~$ cd workspace_c/simple/
compss@bsc:~/workspace_c/simple$ runcompss --lang=c simple <initial_number>
```

```
compss@bsc:~$ runcompss --lang=c /home/compss/workspace_c/simple/master/simple 1
---------------- Executing simple -------------------------

JVM_OPTION_FILE: /tmp/tmp.EIQA3NX3Wr

IT_HOME: /opt/COMPSs//Runtime/scripts/user/../..

Args: 1

WARNING: IT Properties file is null. Setting default values
[   API]  -  Starting COMPSs Runtime v1.3 (build 20150820-1617.rnull)
Initial counter value is 1
Final counter value is 2
[   API]  -  No more tasks for app 0
[   API]  -  Getting Result Files 0
[   API]  -  Execution Finished
-----------------------------------------------------------
```

Figure 3: Execution of a C++ COMPSs application.

2.4 Additional configurations

The COMPSs runtime has two configuration files: *resources.xml* and *project.xml* . These files contain information about the execution environment and are completely independent from the application.

For each execution users can load the default configuration files or specify their custom configurations by using, respectively, the $--resources =< absolute\_path\_to\_resources.xml >$ and the $--project =< absolute\_path\_to\_project.xml >$ in the *runcompss* command. The default files are located in the */opt/COMPSs/Runtime/configuration/xml/* path. Users can manually edit these files or can use the *Eclipse IDE* tool developed for COMPSs. For further information about the *Eclipse IDE* please refer to Section 4.4.

Next sections describe in detail the *resources.xml* and the *project.xml* files, explaining the available options.

2.4.1 Resources file

The *resources* file provides information about all the available resources that can be used for an execution. This file should normally be managed by the system administrators. Its full definition schema can be found at
/opt/COMPSs/Runtime/configuration/xml/resources/resource_schema.xsd.

It contains one entry per available resource defining its name, its capabilities and its requirements. Administrators can define several resource capabilities (see example in the next listing) but we would like to underline the importance of **Processor CoreCount**. This capability represents the number of available cores in the described resource and it is used to schedule the correct number of tasks. Thus, it becomes essential to define it accordingly to the number of cores in the physical resource.

```
compss@bsc:~$ cat /opt/COMPSs/Runtime/configuration/xml/resources/resources.xml
<?xml version="1.0" encoding="UTF-8"?>
<ResourceList>
        <Resource Name="localhost">
                <Capabilities>
                        <Host>
                                <TaskCount>0</TaskCount>
                                <Queue>short</Queue>
                                <Queue/>
                        </Host>
                        <Processor>
                                <Architecture>IA32</Architecture>
                                <Speed>3.0</Speed>
                                < CoreCount > 4 < /CoreCount >
                        </Processor>
                        <OS>
                                <OSType>Linux</OSType>
                                <MaxProcessesPerUser>32</MaxProcessesPerUser>
                        </OS>
                        <StorageElement>
                                <Size>8</Size>
                        </StorageElement>
                        <Memory>
                                <PhysicalSize>4</PhysicalSize>
                                <VirtualSize>8</VirtualSize>
                        </Memory>
                        <ApplicationSoftware>
                                <Software>Java</Software>
                        </ApplicationSoftware>
                        <Service/>
                        <VO/>
                        <Cluster/>
                        <FileSystem/>
                        <NetworkAdaptor/>
                        <JobPolicy/>
                        <AccessControlPolicy/>
                </Capabilities>
                <Requirements/>
                <Adaptors>
                        <Adaptor name="integratedtoolkit.nio.master.NIOAdaptor">
                                <MinPort>43001</MinPort>
                                <MaxPort>43002</MaxPort>
                        </Adaptor>
                </Adaptors>
        </Resource>
</ResourceList>
```

2.4.2 Project file

The project file provides information about the resources used in a specific execution. Consequently, the resources that appear in this file are a subset of the resources described in the *resources.xml* file. This file, that contains one entry per worker, is usually edited by the users and changes from execution to execution. Its full definition schema can be

found at
/opt/COMPSs/Runtime/configuration/xml/projects/project_schema.xsd.
We emphasize the importance of correctly defining the following entries:

installDir Indicates the path of the COMPSs installation **inside the resource** (not necessarily the same than in the local machine).

User Indicates the username used to connect via ssh to the resource. This user **must** have passwordless access to the resource (for more information check the *COMPSs Installation Manual* available at our website `http://compss.bsc.es`). If left empty COMPSs will automatically try to access the resource with the **same username than the one that lauches the COMPSs main application**.

LimitOfTasks The maximum number of tasks that can be simultaneously scheduled to a resource. Considering that a task can use more than one core of a node, this value must be lower or equal to the number of available cores in the resource.

```
compss@bsc:~$ cat /opt/COMPSs/Runtime/configuration/xml/projects/project.xml
<?xml version="1.0" encoding="UTF-8"?>
<Project>
  <!--Description for any physical node-->
  <Worker Name="localhost">
    < InstallDir > /opt/COMPSs/Runtime/scripts/system/ < /InstallDir >
    <WorkingDir>/tmp/</WorkingDir>
    <!-- < User > user < /User > -- >
    < LimitOfTasks > 4 < /LimitOfTasks >
  </Worker>
</Project>
```

2.5 Configuration examples

In the next subsections we provide specific information about the services, shared disks, cluster and cloud configurations and several *project.xml* and *resources.xml* examples.

2.5.1 Services configuration

To allow COMPSs applications to use WebServices as tasks, the *resources.xml* can include a special type of resource called *Service*. For each WebService it is necessary to specify its wsdl, its name, its namespace and its port.

```
<?xml version="1.0" encoding="UTF-8"?>
<ResourceList>
        <Resource Name="localhost">
                ...
        </Resource>

        <Service wsdl="http://bscgrid05.bsc.es:20390/hmmerobj/hmmerobj?wsdl">
                <Name>HmmerObjects</Name>
                <Namespace>http://hmmerobj.worker</Namespace>
                <Port>HmmerObjectsPort</Port>
        </Service>
</ResourceList>
```

When configuring the *project.xml* file it is necessary to include the service as a worker by adding an special entry indicating only the name and the limit of tasks as shown in the following example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Project>
        <!--Description for any physical node-->
        <Worker Name="localhost">
                ...
        </Worker>

        <Worker Name="http://bscgrid05.bsc.es:20390/hmmerobj/hmmerobj?wsdl">
                <LimitOfTasks>3</LimitOfTasks>
        </Worker>
</Project>
```

2.5.2 Cluster and grid configuration (static resources)

In order to use external resources to execute the applications, the following steps have to be followed:

1. Install the *COMPSs Worker* package (or the full *COMPSs Framework* package) on all the new resources following the *Installation manual* available at `http://compss.bsc.es` .

2. Set SSH passwordless access to the rest of the remote resources.

3. Create the *WorkingDir* directory in the resource (remember this path because it is needed for the *project.xml* configuration).

4. Manually deploy the application on each node.

The *resources.xml* and the *project.xml* files must be configured accordingly. Here we provide examples about configuration files for Grid and Cluster environments.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ResourceList>
    <Resource Name="hostname1.domain.es">
        <Capabilities>
            <Host>
                <TaskCount>0</TaskCount>
                <Queue>Short</Queue>
            </Host>
            <Processor>
                <Architecture>x86_64</Architecture>
                <Speed>2.5</Speed>
                <CoreCount>4</CoreCount>
            </Processor>
            <OS>
                <OSType>Linux</OSType>
            </OS>
            <StorageElement>
                <Size>250.0</Size>
            </StorageElement>
            <Memory>
                <PhysicalSize>4.0</PhysicalSize>
            </Memory>
```

```xml
                <ApplicationSoftware>
                    <Software>BLAST</Software>
                </ApplicationSoftware>
            </Capabilities>
            <Requirements/>
            <Adaptors>
                <Adaptor name="integratedtoolkit.nio.master.NIOAdaptor">
                    <MinPort>43001</MinPort>
                    <MaxPort>43002</MaxPort>
                </Adaptor>
            </Adaptors>
        </Resource>
        <Resource Name="hostname2.domain.es">
    ...
        </Resource>
</ResourceList>
```

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Project>
    <Worker Name="hostname1.domain.es">
        <InstallDir>/opt/COMPSs/Runtime/scripts/system/</InstallDir>
        <WorkingDir>/home/user/</WorkingDir>
        <User>user</User>
        <LimitOfTasks>2</LimitOfTasks>
    </Worker>
    <Worker Name="hostname2.domain.es">
        ...
    </Worker>
</Project>
```

2.5.3 Shared Disks configuration example

Configuring shared disks might reduce the amount of data transfers improving the application performance. To configure a shared disk the users must edit the *resources.xml* indicating how the shared disk is hosted in the master node and how the shared disk is mounted in each worker.

To indicate a shared disk hosted in the master node the *resources.xml* file must include a *Disk* tag describing the disk and the mount point. The following example states that in the master node there is a shared disk labelled *sharedDisk0* mounted on the */sharedDisk* directory.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ResourceList>
    <Disk Name="sharedDisk0">
        <MountPoint>/sharedDisk</MountPoint>
    </Disk>
<ResourceList>
```

On the other side, to declare that a worker has a shared disk mounted the *resources.xml* file must include a *Disk* tag inside the specific worker indicating its name (defined in the master *Disk* tag) and its mount point inside the worker. In the following example the *sharedDisk0* is mounted on the resource *hostname1.domain.es* under the path */home-/user/mySharedDisk/* .

```
<Resource Name="hostname1.domain.es">
    <Capabilities>
    ...
    </Capabilities>
    <Requirements/>
    <Adaptors>
    ...
    </Adaptors>
    <Disks>
        <Disk Name="sharedDisk0">
            <MountPoint>/home/user/mySharedDisk</MountPoint>
        </Disk>
    </Disks>
</Resource>
```

The *Disks* tag can have multiple disk children nodes.

2.5.4 Cloud configuration (dynamic resources)

In order to use cloud resources to execute the applications, the following steps have to be followed:

1. Prepare cloud images with the *COMPSs Worker* package or the full *COMPSs Framework* package installed.

2. The application will be deployed automatically during execution but the users need to set up the configuration files to specify the application files that must be deployed.

The COMPSs runtime communicates with a cloud manager by means of connectors. Each connector implements the interaction of the runtime with a given provider's API, supporting four basic operations: ask for the price of a certain VM in the provider, get the time needed to create a VM, create a new VM and terminate a VM. This design allows connectors to abstract the runtime from the particular API of each provider and facilitates the addition of new connectors for other providers.

The *resources.xml* file must contain one or more $< CloudProvider >$ tags that include the information about a particular provider, associated to a given connector. The tag **must** have an attribute **name** to uniquely identify the provider. Table 1 summarizes the information to be specified by the user inside this tag.

Server	Endpoint of the provider's server
Connector	Class that implements the connector

ImageList	Multiple entries of VM templates
• **Image**	• VM image
– **Architecture**	– Architeture of the VM image
– **OSType**	– Operative System installed in the VM image
– **ApplicationSoftware**	– Multiple entries of software installed in the VM image
* **Software**	* Software installed in the VM image
– **SharedDisks**	– Multiple entries of shared disks mounted in the VM image
* **Disk**	* Disk description
– **Adaptors**	– Multiple entries of adaptors available at the VM image
* **Adaptor**	* Adaptor name and specific properties
InstanceTypes	Multiple entries of resource templates
• **Resource**	• Instance type offered by the provider
– **Capabilities**	– Hardware details of instance type
* **Processor**	* Architecture and number of available cores
* **StorageElement**	* Size in GB of the storage
* **Memory**	* PhysicalSize, in GB of the available RAM

Table 1: Configuration of resources.xml file, tag $< CloudProvider >$

The *project.xml* complements the information about a provider listed in the *resources.xml* file. This file can contain a $< Cloud >$ tag where to specify a list of providers, each with a $< Provider >$ tag, whose **name** attribute must match one of the providers in the *resources.xml* file. Thus, the *project.xml* file **must** contain a subset of the providers specified in the *resources.xml* file. Table 3 summarizes the information that users need to specify inside the $< Cloud >$ tag and Table 2 summarizes the information that users need to specify inside the $< Provider >$ tag of the *project.xml* file.

InitialVMs	Number of VM to be created at the beginning of the application
minVMCount	Minimum number of VMs available in the computation
maxVMCount	Maximum number of VMs available in the computation
Provider	Multiple entries of Cloud providers

Table 2: Configuration of project.xml file, tag $< Cloud >$

LimitOfVMs	Maximum number of VMs allowed by the provider
Property • Name • Value	Multiple entries of provider-specific properties • Name of the property • Value of the property

ImageList
 	
 • Image
 	
    – InstallDir
 	
    – WorkingDir
 	
    – User
    – Package
 	
      ∗ Source
      ∗ Target
 	
      ∗ InstalledSoftware
	Multiple entries of VM images available at the provider
	
	 • VM image
	
	    – Path of the COMPSs worker scripts in the image
	
	    – COMPSs working directory in the deployed instances
	
	    – Account username
	    – Multiple entries of local packages that have to be deployed in new instances
	
	      ∗ Local path of the package
	      ∗ Path where to deploy the package in the new instance
	
	      ∗ List of software included in the package

InstanceTypes • Resource	List of resource types that are available in the provider • Resource description

Table 3: Configuration of project.xml file, tag $< Provider >$

The next sections provide a description of each of the currently available connectors.

2.5.4.1 Cloud connectors: Amazon EC2

The COMPSs runtime features a connector to interact with the Amazon Elastic Compute Cloud (EC2).

Amazon EC2 offers a well-defined pricing system for VM rental. A total of 8 pricing zones are established, corresponding to 8 different locations of Amazon datacenters around the globe. Besides, inside each zone, several per-hour prices exist for VM instances with different capabilities. The EC2 connector stores the prices of standard on-demand VM instance types (t1.micro, m1.small, m1.medium, m1.large and m1.xlarge) for each zone. Spot instances are not currently supported by the connector.

When the COMPSs runtime chooses to create a VM in the Amazon public Cloud, the EC2 connector receives the information about the requested characteristics of the new VM, namely the number of cores, memory, disk and architecture (32/64 bits). According to that information, the connector tries to find the VM instance type in Amazon that better matches those characteristics and then requests the creation of a new VM instance of that type.

Once an EC2 VM is created, a whole hour slot is paid in advance; for that reason, the connector keeps the VM alive at least during such period, saving it for later use if necessary. When the task load decreases and a VM is no longer used, the connector puts it aside if the hour slot has not expired yet, instead of terminating it. After that, if the task load increases again and the EC2 connector requests a VM, first the set of saved VMs is examined in order to find a VM that is compatible with the requested characteristics. If one is found, the VM is reused and becomes eligible again for the execution of tasks; hence, the cost and time to create a new VM are not paid. A VM is only destroyed when the end of its hour slot is approaching and it is still in saved state.

Table 4 summarizes the provider-specific properties that must be defined in the project.xml file for the Amazon EC2 connector.

Placement	Location of the amazon datacentre to use
Access Key Id	Identifier of the access key of the Amazon EC2 account
Secret Key Id	Identifier of the secret key of the Amazon EC2 account
Key host location	Path to the SSH key in the local host, used to connect to the VMs
KeyPair name	Name of the key pair to use
SecurityGroup name	Name of the security group to use

Table 4: Properties of the Amazon EC2 connector.

2.5.4.2   Cloud connectors: rOCCI

The connector uses the rOCCI binary client[1] (version newer or equal than 4.2.5) which has to be installed in the node where the COMPSs main application is executed.

This connector needs additional files providing details about the resource templates available on each provider. This file is located under $< COMPSs\_INSTALL\_DIR >$ $/configuration/xml/templates$ path. Additionally, the user must define the virtual images flavors and instance types offered by each provider; thus, when the runtime decides the creation of a VM, the connector selects the appropriate image and resource template according to the requirements (in terms of CPU, memory, disk, etc) by invoking the rOCCI client through Mixins (heritable classes that override and extend the base templates).

Table 5 contains the rOCCI specific properties that must be defined under the *Provider* tag in the *project.xml* file and Table 5 contains the specific properties that must be defined under the *Instance* tag.

Provider	
ca-path	Path to CA certificates directory
user-cred	Path of the VOMS proxy
auth	Authentication method, x509 only supported
owner	Optional. Used by the VENUS-C Job Manager (PMES)
jobname	

Table 5: rOCCI extensions in the project.xml file.

Instance	Multiple entries of resource templates.
Type	Name of the resource template. It has to be the same name than in the previous files
CPU	Number of cores
Memory	Size in GB of the available RAM
Disk	Size in GB of the storage
Price	Cost per hour of the instance

Table 6: Configuration of the $< provider > .xml$ templates file.

---

[1]https://appdb.egi.eu/store/software/rocci.cli

3 Results and logs

3.1 Results

When executing a COMPSs application we consider different type of results:

- **Application Output:** Output generated by the application.

- **Application Files:** Files used or generated by the application.

- **Tasks Output:** Output generated by the tasks invoked from the application.

Regarding the application output, COMPSs will preserve the application output but will add some pre and post output to indicate the COMPSs Runtime state. Figure 4 shows the standard output generated by the execution of the Simple Java application. The green box highlights the application *stdout* while the rest of the output is produced by COMPSs.

Figure 4: Output generated by the execution of the *Simple* Java application with COMPSs

Regarding the application files, COMPSs **does not modify** any of them and thus, the results obtained by executing the application with COMPSs are the same than the ones generated by the sequential execution of the application.

Regarding the tasks output, COMPSs introduces some modifications due to the fact that tasks can be executed in remote machines. After the execution, COMPSs stores the *stdout* and the *stderr* of each job (a task execution) inside the */home/$USER/.COMPSs/$APPNAME/$EXEC_NUMBER/jobs/* directory of the main application node.

Figures 5 and 6 show an example of the results obtained from the execution of the *Hello* Java application. While Figure 5 provides the output of the sequential execution of the application (without COMPSs), Figure 6 provides the output of the equivalent COMPSs execution. Please note that the sequential execution produces the " Hello World! (from a task) " message in the *stdout* while the COMPSs execution stores the message inside the *job1_NEW.out* file.

Figure 5: Sequential execution of the *Hello* java application

```
compss@bsc:~/workspace_java/hello/jar$ runcompss -d hello.Hello
Using default location for project file: /opt/COMPSs/Runtime/scripts/user/../../configuration/xml/projects/project.xml
Using default location for resources file: /opt/COMPSs/Runtime/scripts/user/../../configuration/xml/resources/resources.xml

---------------- Executing hello.Hello ------------------------

WARNING: IT Properties file is null. Setting default values
[   API]  -  Deploying COMPSs Runtime v1.3 (build 20150820-1617.rnull)
[   API]  -  Starting COMPSs Runtime v1.3 (build 20150820-1617.rnull)
[   API]  -  Initializing components
[   API]  -  Ready to process tasks
Hello World! (from main application)
[   API]  -  Creating task from method sayHello in hello.HelloImpl
[   API]  -  There is 0 parameter
[   API]  -  No more tasks for app 1
[   API]  -  Getting Result Files 1
[   API]  -  Stop IT reached
[   API]  -  Stopping AP...
[   API]  -  Stopping TD...
[   API]  -  Stopping Comm...
[   API]  -  Execution Finished

----------------------------------------------------------
compss@bsc:~$ more ~/.COMPSs/hello.Hello_01/jobs/job1_NEW.out
2015-08-20 16:47:31,615 [DEBUG|integratedtoolkit.Worker|Executor] Creating sandbox for job 1
WORKER - Parameters of execution:
  * Method class: hello.HelloImpl
  * Method name: sayHello
  * Parameter types:
  * Parameter values:
Hello World! (from a task)
2015-08-20 16:47:31,616 [DEBUG|integratedtoolkit.Worker|Executor] Removing sandbox for job 1
```

Figure 6: COMPSs execution of the *Hello* java application

3.2 Logs

COMPSs includes three log levels for running applications but users can modify them or add more levels by editing the logger files under the */opt/COMPSs/Runtime/configuration/log/* folder. Any of these log levels can be selected by adding the $--log\_level =<debug|info|off>$ flag to the *runcompss* command. The default value is *off*.

The logs generated by the NUM_EXEC execution of the application APP by the user USER are stored under */home/$USER/.COMPSs/$APP/$EXEC_NUMBER/* folder (from this point on: **base log folder**). The $EXEC\_NUMBER$ execution number is automatically used by COMPSs to prevent mixing the logs of data of different executions.

When running COMPSs with **log level off** only the errors are reported. This means that the *base log folder* will contain two empty files (**runtime.log** and **resources.log**) and one empty folder (*jobs*). If somehow the application has failed, the *runtime.log* and/or the *resources.log* will not be empty and a new file per failed job will appear inside the *jobs* folder to store the *stdout* and the *stderr*. Figure 7 shows the logs generated by the execution of the Simple java application (without errors) in **off** mode.


```
.COMPSs/
└── [4.0K]  simple.Simple_01
    ├── [4.0K]  jobs
    ├── [   0]  resources.log
    ├── [   0]  runtime.log
    └── [4.0K]  tmpFiles
```

Figure 7: Structure of the logs folder for the Simple java application in **off** mode

When running COMPSs with **log level info** the *base log folder* will contain two files (**runtime.log** and **resources.log**) and one folder (*jobs*). The **runtime.log** file contains the execution information retrieved from the master resource, including the file

16

transfers and the job submission details. The **resources.log** file contains information about the available resources such as the number of processors of each resource (slots), the information about running or pending tasks in the resource queue and the created and destroyed resources. The jobs folder will be empty unless there has been a failed job. In this case it will store, for each failed job, one file for the *stdout* and another for the *stderr*. As an example, Figure 8 shows the logs generated by the same execution than the previous case but with **info** mode.

Figure 8: Structure of the logs folder for the Simple java application in **info** mode

The runtime.log and resources.log are quite large files, thus they should be only checked by advanced users. For an easier interpretation of these files the COMPSs Framework includes a monitor tool. For further information about the COMPSs Monitor please check Section 4.2.

Figures 9 and 10 provide the content of these two files generated by the execution of the *Simple* java application.
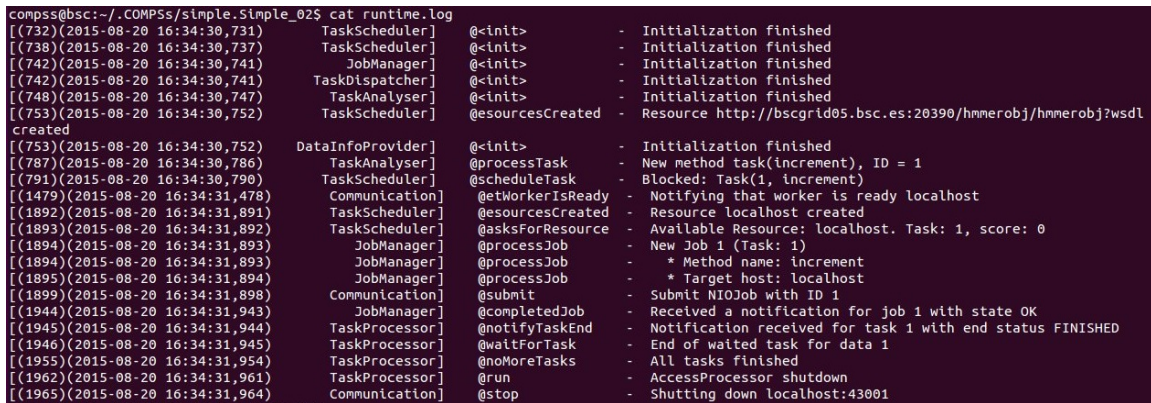

Figure 9: runtime.log generated by the execution of the *Simple* java application

Running COMPSs with **log level debug** generates the same files as the info log level but with more detailed information. Additionally, the *jobs* folder contains two files per **submitted** job; one for the *stdout* and another for the *stderr*. In the other hand, the COMPSs Runtime state is printed out on the *stdout*. Figure 11 shows the logs generated by the same execution than the previous cases but with **debug** mode.

The runtime.log and the resources.log files generated in this mode can be **extremely large**. Consequently, the users should take care of their quota and manually erase these files if needed.

When running Python applications a *pycompss.log* file is written inside the *base log folder* containing debug information about the specific calls to PyCOMPSs.

```
compss@bsc:~/.COMPSs/simple.Simple_02$ cat resources.log
TIMESTAMP = 1440081270727
INFO_MSG = [New resource available in the pool. Name = http://bscgrid05.bsc.es:20390/hmmerobj/hmmerobj?wsdl]
TIMESTAMP = 1440081270752
LOAD_INFO = [
        CORE_INFO = [
                COREID = 0
                NO_RESOURCE = 0
                TO_RESCHEDULE = 0
                ORDINARY = 0
                MIN = 100
                MEAN = 100
                MAX = 100
        ]
]

TIMESTAMP = 1440081271891
INFO_MSG = [New resource available in the pool. Name = localhost]
TIMESTAMP = 1440081271962
INFO_MSG = [Stopping all workers]
TIMESTAMP = 1440081271962
LOAD_INFO = [
        CORE_INFO = [
                COREID = 0
                NO_RESOURCE = 0
                TO_RESCHEDULE = 0
                ORDINARY = 0
                MIN = 56
                MEAN = 56
                MAX = 56
        ]
]
```

Figure 10: resources.log generated by the execution of the *Simple* java application

```
.COMPSs/
└── [4.0K]  simple.Simple_03
    ├── [4.0K]  jobs
    │   ├── [   0]  job1_NEW.err
    │   └── [ 380]  job1_NEW.out
    ├── [ 612]  resources.log
    ├── [ 70K]  runtime.log
    └── [4.0K]  tmpFiles
```

Figure 11: Structure of the logs folder for the Simple java application in **debug** mode

Furthermore, when running *runcompss* with additional flags (such as monitoring or tracing) additional folders will appear inside the *base log folder*. The meaning of the files inside these folders is explained in Section 4.

4   COMPSs Tools

4.1   Application graph

At the end of the application execution a dependency graph can be generated representing the order of execution of each type of task and their dependencies. To allow the final graph generation the *-g* flag has to be passed to the *runcompss* command; the graph file is written in the *base_log_folder/monitor/complete_graph*.dot at the end of the execution.

Figure 12 shows a dependency graph example of a *SparseLU* java application. The graph can be visualized by running the following command:

```
compss@bsc:~$ gengraph ~/.COMPSs/sparseLU.arrays.SparseLU_01/monitor/complete_graph.dot
```
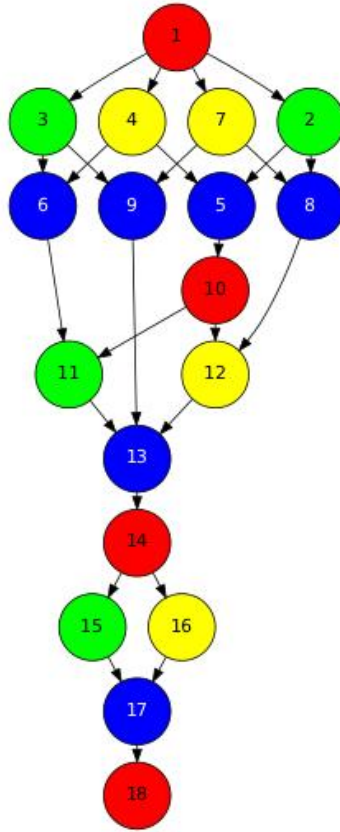

Figure 12: The dependency graph of the SparseLU application

4.2   COMPSs Monitor

The COMPSs Framework includes a Web graphical interface that can be used to monitor the execution of COMPSs applications. COMPSs Monitor is installed as a service and can be easily managed by running any of the following commands:

```
compss@bsc:~$ sudo service compss-monitor usage
Usage: /usr/sbin/service compss-monitor
            {start | stop | reload | restart | try-restart | force-reload | status}
```

4.2.1 Service configuration

The COMPSs Monitor service can be configured by editing the */opt/COMPSs/Tools/ monitor/apache-tomcat/conf/compss-monitor.conf* file which contains one line per property:

- *IT_MONITOR* Default directory to retrieve monitored applications (defaults to the *.COMPSs* folder inside the *root* user).

- *COMPSs_MONITOR_PORT* Port where to run the compss-monitor web service (defaults to 8080).

- *COMPSs_MONITOR_TIMEOUT* Web page timeout between browser and server (defaults to 20s).

4.2.2 Usage

In order to use the COMPSs Monitor users need to start the service as shown in Figure 13.

```
compss@bsc:~$ /etc/init.d/compss-monitor start
* Starting COMPs Monitor
* Checking JAVA Installation...
Warning: JRE_HOME not defined
Info: JAVA_HOME found.
  Loading JRE_HOME from JAVA_HOME
    Success
* Checking IT_HOME...
WARNING: IT_HOME not defined. Trying default location /opt/COMPSs/
    Success
* Checking IT_MONITOR...
      IT_MONITOR=/home/compss/.COMPSs/
    Success
* Checking COMPSs Monitor Port...
Warning: COMPSs_MONITOR_PORT not defined.
  Loading from configuration file.
      COMPSs_MONITOR_PORT=8080
    Success
* Checking COMPSs Monitor Timeout...
Warning: COMPSs_MONITOR_TIMEOUT not defined.
  Loading from configuration file.
      COMPSs_MONITOR_TIMEOUT=20000
    Success
* Configuring COMPSs Monitor service...
    Success
Using CATALINA_BASE:   /opt/COMPSs/Tools/monitor/apache-tomcat
Using CATALINA_HOME:   /opt/COMPSs/Tools/monitor/apache-tomcat
Using CATALINA_TMPDIR: /opt/COMPSs/Tools/monitor/apache-tomcat/temp
Using JRE_HOME:        /usr/lib/jvm/java-7-openjdk-amd64/jre
Using CLASSPATH:       /opt/COMPSs/Tools/monitor/apache-tomcat/bin/bootstrap.jar:/opt/COMPSs/Tools/monitor/apache-tomcat/bin/tomcat-juli.jar
Tomcat started.
```

Figure 13: COMPSs Monitor start command

And use a web browser to open the specific URL:

```
compss@bsc:~$ firefox http://localhost:8080/compss-monitor &
```
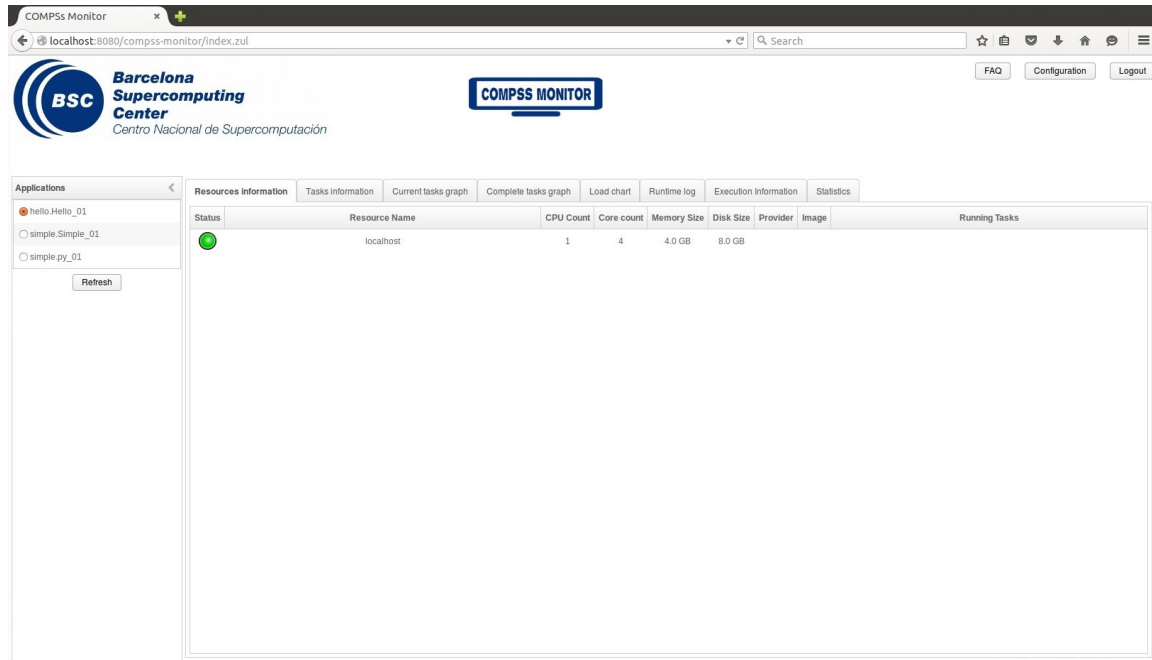
Figure 14: COMPSs monitoring interface

The COMPSs Monitor allows to monitor applications from different users and thus, users need to first login to access their applications. As shown in Figure 14, the users can select any of their executed or running COMPSs applications and display it.

To enable **all** the COMPSs Monitor features, applications must run the runcompss command with the *-m* flag. This flag allows the COMPSs Runtime to store special information inside inside the *log base folder* under the *monitor* folder (see Figures 15 and 16). Only advanced users should modify or delete any of these files. If the application that a user is trying to monitor has not been executed with this flag, some of the COMPSs Monitor features will be disabled.

Figure 15: Execution of the Simple Java application with the monitoring flag enabled

Figure 16: Logs generated by the Simple java application with the monitoring flag enabled

4.2.3 Graphical Interface features

In this section we provide a summary of the COMPSs Monitor supported features available through the graphical interface:

- **Resources information**
  Provides information about the resources used by the application

- **Tasks information**
  Provides information about the tasks definition used by the application

- **Current tasks graph**
  Shows the tasks dependency graph currently stored into the COMPSs Runtime

- **Complete tasks graph**
  Shows the complete tasks dependecy graph of the application

- **Load chart**
  Shows different dynamic charts representing the evolution over time of the resources load and the tasks load

- **Runtime log**
  Shows the runtime log

- **Execution Information**
  Shows specific job information allowing users to easily select failed or uncompleted jobs

- **Statistics**
  Shows application statistics such as the accumulated cloud cost.

The webpage also allows users to configure some performance parameters of the monitoring service by accessing the *Configuration* button at the top-right corner of the web page.

For specific COMPSs Monitor feature configuration please check our *FAQ* section at the top-right corner of the web page.

> **Attention**: To enable all the COMPSs Monitor features applications must run with the $-m$ flag.

4.3 Application tracing

COMPSs Runtime can generate a post-execution trace of the execution of the application. This trace is useful for performance analysis and diagnosis.

A trace file may contain different events to determine the COMPSs master state, the task execution state or the file-transfers. The current release does not support file-transfers informations.

During the execution of the application, an XML file is created in the worker nodes to keep track of these events. At the end of the execution, all the XML files are merged to get a final trace file.

In this manual we only provide information about how to obtain a trace and about the available Paraver (the tool used to analyze the traces) configurations. For further information about the application instrumentation or the trace visualization and configurations please check the *COMPSs Tracing Manual* available at `http://compss.bsc.es`.

4.3.1 Trace Command

In order to obtain a post-execution trace file one of the following options **-t, –tracing, –tracing=true, –tracing=basic** must be added to the runcompss command. All this options activate the basic tracing mode; the advanced mode is activated with the option **–tracing=advanced**. For further information about advanced mode check the *COMPSs Tracing Manual*. Next, we provide an example of the command execution with the basic tracing option enabled for a java k-means application.

```
*** RUNNING JAVA APPLICATION KMEANS
Resolved: /path/to/jar/kmeans.jar:
---------------- Executing kmeans.Kmeans ------------------------

Extrae: WARNING!
Extrae: WARNING! XML parser version and property 'xml-parser-id' do not match. Check the XML file. Trying
    to proceed...
Extrae: WARNING!
Extrae: xml-parser-id found 'Id: xml-parse.c 3682 2015-11-26 14:32:27Z harald $' when expecting 'Id: xml-
    parse.c 3918 2016-03-11 14:59:01Z harald $'.
Welcome to Extrae 3.3.0 (revision 3966 based on extrae/trunk)
Extrae: Parsing the configuration file (/opt/COMPSs/Runtime/scripts/user/../../configuration/xml/tracing/
    extrae_basic.xml) begins
Extrae: Tracing package is located on /opt/COMPSs/Dependencies/extrae/
Extrae: Generating intermediate files for Paraver traces.
Extrae: Warning! change-at-time time units not specified. Using seconds
Extrae: PAPI domain set to ALL for HWC set 1
Extrae: HWC set 1 contains following counters < PAPI_TOT_INS (0x80000032) PAPI_TOT_CYC (0x8000003b)
    PAPI_L2_DCM (0x80000002) PAPI_L3_TCM (0x80000008) > - never changes
Extrae: Warning! change-at-time time units not specified. Using seconds
WARNING: IT Properties file is null. Setting default values
[   API] -  Deploying COMPSs Runtime v1.4 (build 20160412-1147.r2040)
[   API] -  Tracing is activated
[   API] -  Starting COMPSs Runtime v1.4 (build 20160412-1147.r2040)
```

```
...
...
...
merger: Output trace format is: Paraver
merger: Extrae 3.3.0 (revision 3966 based on extrae/trunk)
mpi2prv: Assigned nodes < Marginis >
mpi2prv: Assigned size per processor < <1 Mbyte >
mpi2prv: File set-0/TRACE@Marginis.0000001904000000000000.mpit is object 1.1.1 on node Marginis assigned to
        processor 0
mpi2prv: File set-0/TRACE@Marginis.0000001904000000000001.mpit is object 1.1.2 on node Marginis assigned to
        processor 0
mpi2prv: File set-0/TRACE@Marginis.0000001904000000000002.mpit is object 1.1.3 on node Marginis assigned to
        processor 0
mpi2prv: File set-0/TRACE@Marginis.0000001980000001000000.mpit is object 1.2.1 on node Marginis assigned to
        processor 0
mpi2prv: File set-0/TRACE@Marginis.0000001980000001000001.mpit is object 1.2.2 on node Marginis assigned to
        processor 0
mpi2prv: File set-0/TRACE@Marginis.0000001980000001000002.mpit is object 1.2.3 on node Marginis assigned to
        processor 0
mpi2prv: File set-0/TRACE@Marginis.0000001980000001000003.mpit is object 1.2.4 on node Marginis assigned to
        processor 0
mpi2prv: File set-0/TRACE@Marginis.0000001980000001000004.mpit is object 1.2.5 on node Marginis assigned to
        processor 0
mpi2prv: Time synchronization has been turned off
mpi2prv: A total of 9 symbols were imported from TRACE.sym file
mpi2prv: 0 function symbols imported
mpi2prv: 9 HWC counter descriptions imported
mpi2prv: Checking for target directory existance... exists, ok!
mpi2prv: Selected output trace format is Paraver
mpi2prv: Stored trace format is Paraver
mpi2prv: Searching synchronization points... done
mpi2prv: Time Synchronization disabled.
mpi2prv: Circular buffer enabled at tracing time? NO
mpi2prv: Parsing intermediate files
mpi2prv: Progress 1 of 2 ... 5% 10% 15% 20% 25% 30% 35% 40% 45% 50% 55% 60% 65% 70% 75% 80% 85% 90% 95%
    done
mpi2prv: Processor 0 succeeded to translate its assigned files
mpi2prv: Elapsed time translating files: 0 hours 0 minutes 0 seconds
mpi2prv: Elapsed time sorting addresses: 0 hours 0 minutes 0 seconds
mpi2prv: Generating tracefile (intermediate buffers of 838848 events)
        This process can take a while. Please, be patient.
mpi2prv: Progress 2 of 2 ... 5% 10% 15% 20% 25% 30% 35% 40% 45% 50% 55% 60% 65% 70% 75% 80% 85% 90% 95%
    done
mpi2prv: Warning! Clock accuracy seems to be in microseconds instead of nanoseconds.
mpi2prv: Elapsed time merge step: 0 hours 0 minutes 0 seconds
mpi2prv: Resulting tracefile occupies 991743 bytes
mpi2prv: Removing temporal files... done
mpi2prv: Elapsed time removing temporal files: 0 hours 0 minutes 0 seconds
mpi2prv: Congratulations! ./trace/kmeans.Kmeans_compss_trace_1460456106.prv has been generated.
[   API]  -  Execution Finished
```

At the end of the execution the trace will be stored inside the *trace* folder under the application log directory.

```
compss@bsc:~$ cd .COMPSs/kmeans.Kmeans/trace/
compss@bsc:~$ ls -1
hmmerobj.kmeans.Kmeans_compss_trace_1460456106.pcf
hmmerobj.kmeans.Kmeans_compss_trace_1460456106.prv
hmmerobj.kmeans.Kmeans_compss_trace_1460456106.row
```

4.3.2   Trace visualization

The traces generated by an application execution are ready to be visualized with *Paraver*.
Paraver is a powerful tool developed by *BSC* that allows users to show many views of
the trace data by means of different configuration files. Users can manually load, edit or
create configuration files to obtain different trace data views.

If *Paraver* is installed, issue the following command to visualize a given tracefile:

```
compss@bsc:~$ wxparaver path/to/trace/trace_name.prv
```

For further information about *Paraver* please visit the following site:

> http://www.bsc.es/computer-sciences/performance-tools/paraver

4.4 COMPSs IDE

COMPSs IDE is an Integrated Development Environment to develop, compile, deploy and execute COMPSs applications. It is available through the *Eclipse Market* as a plugin and provides an even easier way to work with COMPSs.

For further information please check the *COMPSs IDE User Guide* available at: `http://compss.bsc.es` .

5 Special Execution Platforms

This section provides information about how to run COMPSs Applications in specific platforms such as *Docker*, *Chameleon* or *MareNostrum*.

5.1 Docker

5.1.1 Introduction

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux. In addition to the Docker container engine, there are other Docker tools that allow users to create complex applications (Docker-Compose) or to manage a cluster of Docker containers (Docker Swarm).

COMPSs supports running a distributed application in a Docker swarm cluster.

5.1.2 Requirements

In order to use COMPSs with Docker, some requirements must be fulfilled:

- Have **Docker** and **Docker-Compose** installed in your local machine.

- Have an available **Docker swarm cluster** and its swarm manager ip and port to access it remotely.

- A **Dockerhub account**. Dockerhub is an online repository for Docker images. We don't currently support another sharing method besides uploading to Dockerhub, so you will need to provide a username. This has the advantage that it takes very little to upload the image, since Dockerhub will just need the delta layers from the base image of COMPSs.

For more information about Docker and how to install different components, visit Docker site: `https://www.docker.com/`

5.1.3 Execution

To execute COMPSs in a Docker swarm cluster, you must use the **runcompss-docker** command, instead of runcompss.

The command **runcompss-docker** has some **additional arguments** that will be needed by COMPSs to run your application in a distributed Docker swarm cluster environment. The rest of typical arguments (classpath, project, etc.) will be delegated to runcompss command.

These **mandatory** additional arguments must go **before** the typical runcompss arguments. The **runcompss-docker additional arguments** are:

- **--w, --worker-containers:**
  Specifies the number of **worker containers** the app will execute on. One more container will be created to host the **master**. If you have enough nodes in the swarm cluster, each container will be executed by one node.
  Example: **--worker-containers=3**

- **--c, --context-dir:**
  Specifies the **context directory** of the app. The context directory is a local directory that **must contain the needed binaries and input files of the app**. In its simplest case, it will contain the executable file (a .jar for example). Take into account that you should not put unnecessary files in the context-directory, since **it will be copied to all the nodes** that need it.

  Example: **--context-dir='/home/compss-user/my-app-dir'**, where my-app-dir contains 'app.jar', 'data1.dat' and 'data2.csv', for example.

- **--s, --swarm-manager:**
  Specifies the swarm manager ip and port (format: ¡ip¿:¡port¿). You can test if the swarm manager really works and is reachable from your machine running from your machine the Docker hello-world container.
  Example: **--swarm-manager='129.114.108.8:4000'**

- **--u, --username:**
  Specifies a **Dockerhub username**, to upload the app image, so the workers can pull it in runtime. As stated in the requirements sections, this is needed to share your container application image with the nodes that need it.

As an **optional** argument:

- **--n, --no-refresh-app-image:**
  If this flag is on, the **app image won't be uploaded** to Dockerhub. Workers won't pull the image either. Use this flag if the application has not changed since the last running. This way the execution will be **faster**, and you won't need to specify the Dockerhub username nor write its password. But remember! If you make any change to the application, run an execution without this flag at least once, to update the online application image.

Here is the **format** you must use with **runcompss-docker** command:

```
runcompss-docker --worker-containers=N
                 --context-dir='CTX_DIR'
                 --swarm-manager='<ip>:<port>'
                 --username='dockerhub_username'
                 [rest of classic runcompss args]
```

Or alternatively, in its shortest form:

```
runcompss-docker --w=N  --c='CTX_DIR' --s='<ip>:<port>' --u='dockerhub_username'
                 [rest of classic runcompss args]
```

The runcompss-docker command creates a Docker image from a common COMPSs docker image by adding your context directory to it. Then it uploads this image to Dockerhub and Docker Compose is in charge of spawning the different application containers to the docker swarm manager. Then the Docker Swarm starts the containers and the application.

The COMPSs Docker base image is available in the Dockerhub. In case you need it, you can pull it using the following command:

```
docker pull compss/compss
```

5.1.4   Execution with TLS

If your cluster uses **TLS** or has been created using **Docker-Machine**, you will have to **export two environment variables** before using runcompss-docker:

On one hand, **DOCKER_TLS_VERIFY** environment variable will tell Docker that you are using TLS:

```
export DOCKER_TLS_VERIFY="1"
```

On the other hand, **DOCKER_CERT_PATH** variable will tell Docker where to find your TLS certificates. As an example:

```
export DOCKER_CERT_PATH="/home/compss-user/.docker/machine/machines/my-manager-node"
```

In case you have created your cluster using docker-machine, in order to know what your *DOCKER_CERT_PATH* is, you can use this command:

```
docker-machine env my-swarm-manager-node-name | grep DOCKER_CERT_PATH
```

In which *swarm-manager-node-name* must be changed by the name docker-machine has assigned to your swarm manager node.

With these environment variables set, you are ready to use **runcompss-docker** in a cluster using TLS.

5.1.5 Execution results

The execution results will be retrieved from the master container of your application.

If your context-directory name is **'matmul'**, then your results will be saved in the **'matmul-results'** directory, which **is located in the SAME directory as your context-directory is in**.

Inside the **'matmul-results'** directory you will have:

- A folder named **'matmul'** with all the result files that were in the same directory as the executable when the application execution ended. More precisely, this will contain the context-directory state right after finishing your application execution.

  Additionally, and for more advanced debug purposes, you will have some inter-midiate files created by runcompss-docker(Dockerfile, project.xml, resources.xml), in case you want to check for more complex errors or details.

- A folder named **'debug'**, which (in case you used the runcompss debug option (**-d**)), will contain the **'.COMPSs'** directory, which contains another directory in which there are the typical debug files runtime.log, jobs, etc.
  Remember **.COMPSs** is a **hidden** directory, take this into account if you do **ls** inside the debug directory (add the **-a** option).

To make it simpler, we provide a **tree visualization** of an example of what your directories should look like after the execution. In this case we executed the **Matmul example application** that we provide you:

```
.
├── matmul
│   └── matmul.jar
└── matmul-results
    ├── debug
    │   └── .COMPSs
    │       └── matmul.files.Matmul_01
    │           ├── jobs
    │           │   ├── job1_NEW.err
    │           │   ├── job1_NEW.out
    │           │   ├── job2_NEW.err
    │           │   ├── job2_NEW.out
    │           │   ├── job3_NEW.err
    │           │   └── job3_NEW.out
    │           ├── resources.log
    │           ├── runtime.log
    │           └── tmpFiles
    └── matmul
        ├── A.0.0
        ├── A.0.1
        ├── A.0.2
        ├── B.0.0
        ├── B.0.1
        ├── B.0.2
        ├── Dockerfile
        ├── matmul.jar
        ├── project.xml
        └── resources.xml
```

5.1.6 Execution examples

And here is one example to run the Matmul example application. In this case, we are specifying:

- Use **5 worker docker containers**. They will be distributed amongst the swarm cluster nodes as balanced as possible.

- The **context directory** will be '/home/compss-user/my-app-dir'.

- The **swarm-manager ip** will be 129.114.108.8, with the swarm manager located in the **port** 4000.

- The **Dockerhub username** will be john123 (the **password** will be asked when executing runcompss-docker).

- The **classpath** will be '/home/compss-user-john/matmul/matmul.jar', and we will use **debug (-d)**.

- Finally, as we would do with the typical runcompss, we specify the **main class** name and its **parameters** (16 and 4 in this case).

And this is how you would run **runcompss-docker**:

```
runcompss-docker --worker-containers=5 \
                 --context-dir='/home/compss-user/my-app-dir' \
                 --swarm-manager='129.114.108.8:4000' \
                 --username='john123' \
                 --classpath=/home/compss-user/my-app-dir/my-app.jar \
                 -d \
                 matmul.objects.Matmul 16 4
```

Here we show another example using the short arguments form, with the KMeans example application, that we provide to you:

```
runcompss-docker --w=30 --c='./kmeans-app' --s='110.3.14.159:26535' --u='test1947' \
                 --classpath=./kmeans/kmeans.jar \
                 kmeans.KMeans
```

5.2 Chameleon

5.2.1 Introduction

The Chameleon project is a configurable experimental environment for large-scale cloud research based on a *OpenStack* KVM Cloud. With funding from the *National Science Foundation (NSF)*, it provides a large-scale platform to the open research community allowing them explore transformative concepts in deeply programmable cloud services, design, and core technologies. The Chameleon testbed, is deployed at the *University of Chicago* and the *Texas Advanced Computing Center* and consists of 650 multi-core cloud nodes, 5PB of total disk space, and leverage 100 Gbps connection between the sites.

The project is led by the *Computation Institute* at the *University of Chicago* and partners from the *Texas Advanced Computing Center* at the *University of Texas* at Austin, the *International Center for Advanced Internet Research* at *Northwestern University*, the *Ohio State University*, and *University of Texas* at *San Antoni*, comprising a highly qualified and experienced team. The team includes members from the *NSF* supported *FutureGrid* project and from the *GENI* community, both forerunners of the *NSFCloud* solicitation under which this project is funded. Chameleon will also sets of partnerships with commercial and academic clouds, such as *Rackspace*, *CERN* and *Open Science Data Cloud (OSDC)*.

For more information please check `https://www.chameleoncloud.org/` .

5.2.2 Execution

Currently, COMPSs can only handle the Chameleon infrastructure as a cluster (deployed inside a lease). Next, we provide the steps needed to execute COMPSs applications at Chameleon:

- Make a lease reservation with 1 minimum node (for the COMPSs master instance) and a maximum number of nodes equal to the number of COMPSs workers needed plus one

- Instantiate the master image (based on the published image *COMPSs_1.3_CC-CentOS7*)

- Attach a public IP and login to the master instance (the instance is correctly contextualized for COMPSs executions if you see a COMPSs login banner)

- Set the instance as COMPSs master by running */etc/init.d/chameleon_init start*

- Copy your CH file (API credentials) to the Master and source it

- Run the *chameleon_cluster_setup* script and fill the information when prompted (you will be asked for the name of the master instance, the reservation id and number of workers). This scripts may take several minutes since it sets up the all cluster.

- Execute your COMPSs applications normally using the *runcompss* script

As an example you can check this video `https://www.youtube.com/watch?v=BrQ6anPHjAU` performing a full setup and execution of a COMPSs application at Chameleon.

5.3 SuperComputers

To maintain the portability between different environments, COMPSs has a pre-build structure (see Figure 17) to execute applications in SuperComputers. For this purpose, users must use the *enqueue_compss* script provided in the COMPSs installation. This script has several parameters (see *enqueue_compss -h*) that allow users to customize their executions for any SuperComputer.
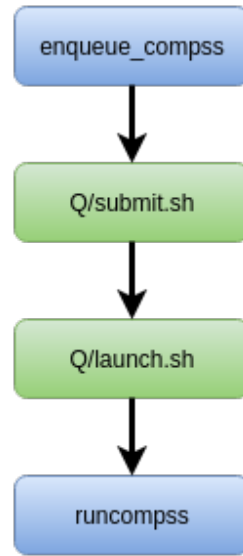
Figure 17: Structure of COMPSs queue scripts. In Blue general scripts, in Green system dependant scripts

To make this structure work, the administrators must define a submit and a launch script that are system dependant. To develop this scripts the current COMPSs installation provides scripts for the *MareNostrum III* SuperComputer that can be used as an example (based on LSF) or you can contact us at `support-compss@bsc.es` .

5.3.1 MareNostrum III

For information about how to submit COMPSs applications at MareNostrum III (BSC) please refer to the *COMPSs at BSC* manual available at `http://compss.bsc.es/releases/compss/latest/docs/COMPSs_MareNostrum_Manual.pdf` .

6   Common Issues

This section provides answers for the most common issues of the execution of COMPSs applications. For specific issues not covered in this section, please do not hesitate to contact us at:

<div align="center">

support-compss@bsc.es

</div>

6.1   How to debug

When the application does not behave as expected the first thing users must do is to run it in **debug** mode executing the *runcompss* command withthe *-d* flag to enable the debug log level.

In this case the application execution will produce the following files:

- runtime.log

- resources.log

- jobs folder

First, users should check the last lines of the runtime.log. If the file-transfers or the tasks are failing an error message will appear in this file. If the file-transfers are successfully and the jobs are submitted, users should check the *jobs* folder and look at the error messages produced inside each job. Users should notice that if there are $_RESUBMITTED$ files something inside the job is failing.

6.2   Tasks are not executed

If the tasks remain in **Blocked** state probably there are no existing resources matching the specific task constraints. This error can be potentially caused by two facts: the resources are not correctly loaded into the runtime, or the task constraints do not match with any resource.

In the first case, users should take a look at the *resouces.log* and check that all the resources defined in the *project.xml* file are available to the runtime. In the second case users should re-define the task constraints taking into account the resources capabilities defined into the *resources.xml* and *project.xml* files.

6.3   Jobs fail

If all the application's tasks fail because all the submitted jobs fail, it is probably due to the fact that there is a resource miss-configuration. In most of the cases, the resource that the application is trying to access has no passwordless access through the configured user. This can be checked by:

- Open the project.xml. (The default file is stored under */opt/COMPSs/ Runtime/-configuration/xml/projects/project.xml*

- For each resource annotate its name and the value inside the *User* tag. Remember that if there is no *User* tag COMPSs will try to connect this resource with the same username than the one that launches the main application.

- For each annotated resourceName - user please try *ssh user@resourceName*. If the connection asks for a password then there is an error in the configuration of the ssh access in the resource.

The problem can be solved running the following commands:

```
compss@bsc:~$ scp ~/.ssh/id_dsa.pub user@resourceName:./mydsa.pub
compss@bsc:~$ ssh user@resourceName "cat mydsa.pub >> ~/.ssh/authorized_keys; rm ./mydsa.pub"
```

These commands are a quick solution, for further details please check the *Additional Configuration* section inside the *COMPSs Installation Manual* available at our website http://compss.bsc.es.

6.4 Compilation error: @Method not found

When trying to compile Java applications users can get some of the following compilation errors:

```
error: package integratedtoolkit.types.annotations does not exist
import integratedtoolkit.types.annotations.Constraints;
                                          ^
error: package integratedtoolkit.types.annotations does not exist
import integratedtoolkit.types.annotations.Method;
                                          ^
error: package integratedtoolkit.types.annotations does not exist
import integratedtoolkit.types.annotations.Parameter;
                                          ^
error: package integratedtoolkit.types.annotations.Parameter does not exist
import integratedtoolkit.types.annotations.Parameter.Direction;
                                                    ^
error: package integratedtoolkit.types.annotations.Parameter does not exist
import integratedtoolkit.types.annotations.Parameter.Type;
                                                    ^
error: cannot find symbol
@Parameter(type = Type.FILE, direction = Direction.INOUT)
^
  symbol:   class Parameter
  location: interface APPLICATION_Itf

error: cannot find symbol
@Constraints(processorCoreCount = 2)
^
  symbol:   class Constraints
  location: interface APPLICATION_Itf

error: cannot find symbol
@Method(declaringClass = "application.ApplicationImpl")
^
  symbol:   class Method
  location: interface APPLICATION_Itf
```

All these errors are raised because the *compss-engine.jar* is not listed in the CLASS-PATH. The default COMPSs installation automatically inserts this package into the

CLASSPATH but it may have been overwritten or deleted. 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 is to manually export 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 recommend 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
```

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

6.5 Jobs failed on method reflection

When executing an application the main code gets stuck executing a task. Taking a look at the *runtime.log* users can check that the job associated to the task has failed (and all its resubmissions too). Then, opening the *jobX_NEW.out* or the *jobX_NEW.err* files users find the following error:

```
[ERROR|integratedtoolkit.Worker|Executor] Can not get method by reflection
integratedtoolkit.nio.worker.executors.Executor$JobExecutionException: Can not get method by reflection
        at integratedtoolkit.nio.worker.executors.JavaExecutor.executeTask(JavaExecutor.java:142)
        at integratedtoolkit.nio.worker.executors.Executor.execute(Executor.java:42)
        at integratedtoolkit.nio.worker.JobLauncher.executeTask(JobLauncher.java:46)
        at integratedtoolkit.nio.worker.JobLauncher.processRequests(JobLauncher.java:34)
        at integratedtoolkit.util.RequestDispatcher.run(RequestDispatcher.java:46)
        at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.NoSuchMethodException: simple.Simple.increment(java.lang.String)
        at java.lang.Class.getMethod(Class.java:1678)
        at integratedtoolkit.nio.worker.executors.JavaExecutor.executeTask(JavaExecutor.java:140)
        ... 5 more
```

This error is due to the fact that COMPSs cannot find one of the tasks declared in the Java Interface. Commonly this is triggered by one of the following errors:

- The *declaringClass* of the tasks in the Java Interface has not been correctly defined.

- The parameters of the tasks in the Java Interface do not match the task call.

- The tasks have not been defined as *public*.

6.6   Jobs failed on reflect target invocation null pointer

When executing an application the main code gets stuck executing a task. Taking a look at the *runtime.log* users can check that the job associated to the task has failed (and all its resubmissions too). Then, opening the *jobX_NEW.out* or the *jobX_NEW.err* files users find the following error:

```
[ERROR|integratedtoolkit.Worker|Executor]
java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at integratedtoolkit.nio.worker.executors.JavaExecutor.executeTask(JavaExecutor.java:154)
        at integratedtoolkit.nio.worker.executors.Executor.execute(Executor.java:42)
        at integratedtoolkit.nio.worker.JobLauncher.executeTask(JobLauncher.java:46)
        at integratedtoolkit.nio.worker.JobLauncher.processRequests(JobLauncher.java:34)
        at integratedtoolkit.util.RequestDispatcher.run(RequestDispatcher.java:46)
        at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.NullPointerException
        at simple.Ll.printY(Ll.java:25)
        at simple.Simple.task(Simple.java:72)
        ... 10 more
```

This cause of this error is that the Java object accessed by the task has not been correctly transferred and one or more of its fields is null. The transfer failure is normally caused because the transferred object is not serializable.

Users should check that all the object parameters in the task are either implementing the serializable interface or following the *java beans* model (by implementing an empty constructor and getters and setters for each attribute).

6.7   Tracing merge failed: too many open files

When too many nodes and threads are instrumented, the tracing merge can fail due to an OS limitation, namely: the maximum open files. This problem usually happens when using advanced mode due to the larger number of threads instrumented. To overcome this issue users have two choices. **First option**, use *Extrae* parallel MPI merger. This merger is automatically used if COMPSs was installed with MPI support. In Ubuntu you can install the following packets to get MPI support:

```
sudo apt-get install libcr-dev mpich2 mpich2-doc
```

To check if COMPSs was deployed with MPI support, you can check the installation log and look for the following *Extrae* configuration output:

```
Package configuration for Extrae 3.3.0 based on extrae/trunk rev. 3966:
----------------------
Installation prefix: /gpfs/apps/MN3/COMPSs/Trunk/Dependencies/extrae
Cross compilation: no
CC: gcc
CXX: g++
Binary type: 64 bits


MPI instrumentation: yes
  MPI home: /apps/OPENMPI/1.8.1-mellanox
  MPI launcher: /apps/OPENMPI/1.8.1-mellanox/bin/mpirun
```

On the other hand, if you already installed COMPSs, you can check *Extrae* configuration executing the script */opt/COMPSs/Dependencies/extrae/etc/configured.sh.* Users should check that flags *–with-mpi=/usr* and *–enable-parallel-merge* are present and that MPI path is correct and exists. Sample output:

```
EXTRAE_HOME is not set. Guessing from the script invoked that Extrae was installed in /opt/COMPSs/
     Dependencies/extrae
The directory exists .. OK
Loaded specs for Extrae from /opt/COMPSs/Dependencies/extrae/etc/extrae-vars.sh


Extrae SVN branch extrae/trunk at revision 3966


Extrae was configured with:
$ ./configure --enable-gettimeofday-clock --without-mpi --without-unwind --without-dyninst --without-
     binutils --with-mpi=/usr --enable-parallel-merge --with-papi=/usr --with-java-jdk=/usr/lib/jvm/java-7-
     openjdk-amd64/ --disable-openmp --disable-nanos --disable-smpss --prefix=/opt/COMPSs/Dependencies/
     extrae --with-mpi=/usr --enable-parallel-merge --libdir=/opt/COMPSs/Dependencies/extrae/lib


CC was gcc
CFLAGS was -g -O2 -fno-optimize-sibling-calls -Wall -W
CXX was g++
CXXFLAGS was -g -O2 -fno-optimize-sibling-calls -Wall -W


MPI_HOME points to /usr and the directory exists .. OK
LIBXML2_HOME points to /usr and the directory exists .. OK
PAPI_HOME points to /usr and the directory exists .. OK
DYNINST support seems to be disabled
UNWINDing support seems to be disabled (or not needed)
Translating addresses into source code references seems to be disabled (or not needed)


Please, report bugs to tools@bsc.es
```

Disclaimer: the parallel merge with MPI will not bypass the system's maximum number of open files, just distribute the files among the resources. If all resources belong to the same machine, the merge will fail anyways.

The **second option** is to increase the OS maximum number of open files. For instance, in Ubuntu:

```
nano /etc/init.d/neo4j-service
```

Add « ulimit –n 40000 » just before the start-stop-daemon line in the do_start section.

Please find more details on the COMPSs framework at
http://compss.bsc.es