



COMP SUPERSCALAR

User Manual

Application execution guide

VERSION: 1.3

October 25, 2015



**Barcelona
Supercomputing
Center**

Centro Nacional de Supercomputación

This manual only provides information about how to execute COMPSs applications. Specifically, it details the options, results and logs of an execution and it provides an overview of the COMPSs tools usage. It is highly recommended to follow 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 full COMPSs application examples (codes, execution commands, results, logs, etc.) please refer to the *COMPSs Sample Applications* available at <http://compss.bsc.es/> .

Contents

| | | |
|----------|---|-----------|
| 1 | COMP Superscalar (COMPSs) | 1 |
| 2 | Executing COMPSs applications | 2 |
| 2.1 | Prerequisites | 2 |
| 2.2 | Runcompss command | 2 |
| 2.3 | Running a COMPSs application | 3 |
| 2.3.1 | Running Java applications | 3 |
| 2.3.2 | Running Python applications | 4 |
| 2.3.3 | Running C/C++ applications | 5 |
| 2.4 | Additional configurations | 5 |
| 2.4.1 | Resources file | 6 |
| 2.4.2 | Project file | 7 |
| 2.5 | Configuration examples | 7 |
| 2.5.1 | Services configuration example | 7 |
| 2.5.2 | Cluster configuration (static resources) example | 8 |
| 2.5.3 | Shared Disks configuration example | 9 |
| 2.5.4 | Cloud configuration (dynamic resources) example | 10 |
| 2.5.4.1 | Cloud connectors: Amazon EC2 | 14 |
| 2.5.4.2 | Cloud connectors: rOCCI | 15 |
| 3 | Results and logs | 16 |
| 3.1 | Results | 16 |
| 3.2 | Logs | 17 |
| 4 | COMPSs Tools | 20 |
| 4.1 | Application graph | 20 |
| 4.2 | COMPSs Monitor | 20 |
| 4.2.1 | Service configuration | 21 |
| 4.2.2 | Usage | 21 |
| 4.2.3 | Graphical Interface features | 23 |
| 4.3 | Application tracing | 24 |
| 4.3.1 | Trace Command | 24 |
| 4.3.2 | Trace Configurations | 25 |
| 4.3.3 | Trace examples | 26 |
| 4.4 | COMPSs IDE | 28 |
| 5 | Common Issues | 29 |
| 5.1 | How to debug | 29 |
| 5.2 | Tasks are not executed | 29 |
| 5.3 | Jobs sistematically fail | 29 |
| 5.4 | Compilation error: @Method not found | 30 |
| 5.5 | Jobs failed on method reflection | 31 |
| 5.6 | Jobs failed on reflect target invocation null pointer | 32 |

List of Figures

| | | |
|----|---|----|
| 1 | Execution of a Java COMPSs application. | 4 |
| 2 | Execution of a Python COMPSs application. | 4 |
| 3 | Execution of a C++ COMPSs application. | 5 |
| 4 | Ouput generated by the execution of the <i>Simple</i> java application with COMPSs | 16 |
| 5 | Sequential execution of the <i>Hello</i> java application | 16 |
| 6 | COMPSs execution of the <i>Hello</i> java application | 17 |
| 7 | Structure of the logs folder for the Simple java application in off mode . . | 17 |
| 8 | Structure of the logs folder for the Simple java application in info mode . . | 18 |
| 9 | runtime.log generated by the execution of the <i>Simple</i> java application . . . | 18 |
| 10 | resources.log generated by the execution of the <i>Simple</i> java application . . | 19 |
| 11 | Structure of the logs folder for the Simple java application in debug mode | 19 |
| 12 | The dependency graph of the SparseLU application | 20 |
| 13 | COMPSs Monitor start command | 21 |
| 14 | COMPSs monitoring interface | 22 |
| 15 | Execution of the Simple java application with the monitoring flag enabled . | 23 |
| 16 | Logs generated by the Simple java application with the monitoring flag enabled | 23 |
| 17 | Examples of complex traces | 27 |

List of Tables

| | | |
|---|---|----|
| 1 | Configuration of resources.xml file, tag <i>< CloudProvider ></i> | 11 |
| 2 | Configuration of project.xml file, tag <i>< Cloud ></i> | 12 |
| 3 | Configuration of project.xml file, tag <i>< Provider ></i> | 13 |
| 4 | Properties of the Amazon EC2 connector. | 14 |
| 5 | rOCCI extensions in the project.xml file. | 15 |
| 6 | Configuration of the <i>< provider > .xml</i> templates file. | 15 |
| 7 | Available paraver configurations for COMPSs Applications | 26 |

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 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 applications for COMPSs please refer to the *COMPSs User Manual: Application development guide* available at the <http://compss.bsc.es/> webpage.

2.2 Runcompss command

All COMPSs applications are executed using the **runcompss** command which is invoked as follows:

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

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

The application arguments are the arguments that the users' main application receives. If needed, they can be empty.

The runcompss command allows the users to customize each COMPSs execution by specifying different options. For clarity purposes, parameters are grouped in *Runtime configuration*, *Tools enablers* and *Advanced options*. Users can add any of them to the runcompss call by following the next usage description.

```
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=<bool>, - -tracing, -t Generation of traces (true/false)
                                   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: .

- -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 it is important to notice that the application files **must** be in the **CLASSPATH**. Thus, when launching a COMPSs application, users can manually pre-set the **CLASSPATH** environment variable or can add the `- -classpath` option to the `runcompss` command.

The next three sections provide specific information for launching COMPSs applications 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 at our 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 are taking advantage of the default value of the `- -classpath` option to automatically add the jar file to the classpath (by moving to the directory


```

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.

which contains the jar file before launching the COMPSS application). However, we can explicitly do this by exporting the **CLASSPATH** variable or by providing the **-classpath** value. Next, we provide another two 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 Python application with COMPSS users only need to provide the **-lang=python** option to the **runcompss** command. Next, we provide an example of a simple python application.

```

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 C/C++ application with COMPSs users need to compile the C/C++ application by using the *buildapp* command. For further information please refer to the *COMPSs User Manual: Application development guide* document available at our webpage <http://compss.bsc.es>. Once done, to run the C/C++ application with COMPSs, the users only need to provide the *-lang=c* option to the *runcompss* command. Next, we provide an example of a simple C++ application:

```
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

COMPSs has **two** configuration files: *resources.xml* and *project.xml*. These files are meant to provide 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 >* inside the *runcompss* command. The default files are located under the */opt/COMPSs/Runtime/configuration/xml/* path. Users can edit these two files manually or can take advantage of the *Eclipse IDE* tool developed for COMPSs. For further information about the *Eclipse IDE* please refer to Section 4.4.

Next sections provide more in-depth information about the *resources.xml* and the *project.xml* files, listing and explaining the available tags.

2.4.1 Resources file

The resources file provides information about **all** the available resources. 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`.

The typical structure of this file is one entry per available resource defining its name, its capabilities and its requirements. Administrators can define several resource capabilities (see example in next listing) but we would like to underline the importance of **Processor CoreCount**. This capability represents the number of available cores in the resource and it is used to schedule the correct number of tasks into a resource. 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 specific execution information. Particularly, it selects and configures in which resources is the application going to be executed. Consequently, the resources that appear in this file are a **subset** of the resources described in the *resources.xml* file. This file should normally be managed by the application users and will usually change from execution to execution. Its full definition schema can be found at */opt/COMPSs/Runtime/configuration/xml/projects/project_schema.xsd*.

The typical structure of this file is one entry per worker, indicating the resource name that it uses and some configuration properties (see example in next listing). 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 launches 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 but not least, 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. Moreover there are *project.xml* and *resources.xml* examples for each case that can be used as templates.

2.5.1 Services configuration example

To allow COMPSs applications use WebServices 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 as shown in the following example.

```
<?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 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 configuration (static resources) example

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. Deploy the application manually on each resource.

Once these steps are done, users need to configure the *resources.xml* and the *project.xml* files accordingly. On the following lines, we provide examples about configuration files for Grid and Cluster environments, which can serve as a reference

```
<?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>
    <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 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 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. The following example states that the *sharedDisk0* is mounted on 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>
```

Although, the example only contains the definition of a single shared disk, the *Disks* tag can have multiple disk child nodes.

2.5.4 Cloud configuration (dynamic resources) example

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 the Cloud by means of Cloud connectors. Each connector implements the interaction of the runtime with a given Cloud provider, more precisely by 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.

In order to add cloud resources the *resources.xml* file can contain one or more *<CloudProvider>* tags that encompass the information about a particular Cloud 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 <ul style="list-style-type: none"> ● Image <ul style="list-style-type: none"> – Architecture – OSType – ApplicationSoftware <ul style="list-style-type: none"> * Software – SharedDisks <ul style="list-style-type: none"> * Disk – Adaptors <ul style="list-style-type: none"> * Adaptor | Multiple entries of VM templates <ul style="list-style-type: none"> ● VM image <ul style="list-style-type: none"> – Architecture of the VM image – Operative System installed in the VM image – Multiple entries of software installed in the VM image <ul style="list-style-type: none"> * Software installed in the VM image – Multiple entries of shared disks mounted in the VM image <ul style="list-style-type: none"> * Disk description – Multiple entries of adaptors available at the VM image <ul style="list-style-type: none"> * Adaptor name and specific properties |
| InstanceTypes <ul style="list-style-type: none"> ● Resource <ul style="list-style-type: none"> – Capabilities <ul style="list-style-type: none"> * Processor * StorageElement * Memory | Multiple entries of resource templates <ul style="list-style-type: none"> ● Instance type offered by the provider <ul style="list-style-type: none"> – Hardware details of instance type <ul style="list-style-type: none"> * Architecture and number of available cores * Size in GB of the storage * PhysicalSize, in GB of the available RAM |

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

The *project.xml* complements the information about Cloud providers specified 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 $\langle Cloud \rangle$ tag and Table 2 summarizes the information that users need to specify inside the $\langle Provider \rangle$ 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 $\langle Cloud \rangle$

| | |
|---|---|
| LimitOfVMs | Maximum number of VMs allowed by the provider |
| Property <ul style="list-style-type: none"> • Name • Value | Multiple entries of provider-specific properties <ul style="list-style-type: none"> • Name of the property • Value of the property |
| ImageList <ul style="list-style-type: none"> • Image <ul style="list-style-type: none"> – InstallDir – WorkingDir – User – Package <ul style="list-style-type: none"> * Source * Target * InstalledSoftware | Multiple entries of VM images available at the provider <ul style="list-style-type: none"> • VM image <ul style="list-style-type: none"> – 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 <ul style="list-style-type: none"> * Local path of the package * Path where to deploy the package in the new instance * List of software included in the package |
| InstanceTypes <ul style="list-style-type: none"> • Resource | List of resource types that are available in the provider <ul style="list-style-type: none"> • Resource description |

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

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

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

In order to execute a COMPSs application in the cloud, the rOCCI connector has to be configured properly. The connector uses the rOCCI binary client¹ (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 flavour and instance types offered by each provider; thus, when the runtime asks for 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 user 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.

```
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 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 task output, COMPSs does introduce some modifications due to the fact that tasks can be executed in remote machines. Considering that we call a job the execution of a task in a given resource, after the execution, COMPSs stores the *stdout* and the *stderr* of each job inside the */home/\$USER/.COMPSs/\$APPNAME/\$EXEC_NUMBER/jobs/* directory.

Figures 5 and 6 show an example of the results obtained from the execution of the *Hello* java application. While Figure 5 provides the sequential execution of the application, Figure 6 provides its equivalent COMPSs execution. Notice 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.

```
compss@bsc:~/workspace_java/hello/jar$ java -cp hello.jar hello.Hello
Hello World! (from main application)
Hello World! (from a task)
```

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 execution number `NUM_EXEC` of the application `APP` by the user `USER` are stored under `/home/$USER/.COMPSS/$APP/$NUM_EXEC/` folder (from this point on: **base log folder**). The execution number is automatically tracked to prevent the loss of data from previous executions, but users do not need to take care of this value.

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 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, for each failed job, it will store 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.

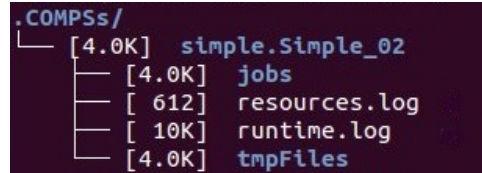


Figure 8: Structure of the logs folder for the Simple java application in **info** mode

The runtime.log and resources.log are quite large files and, thus, 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.

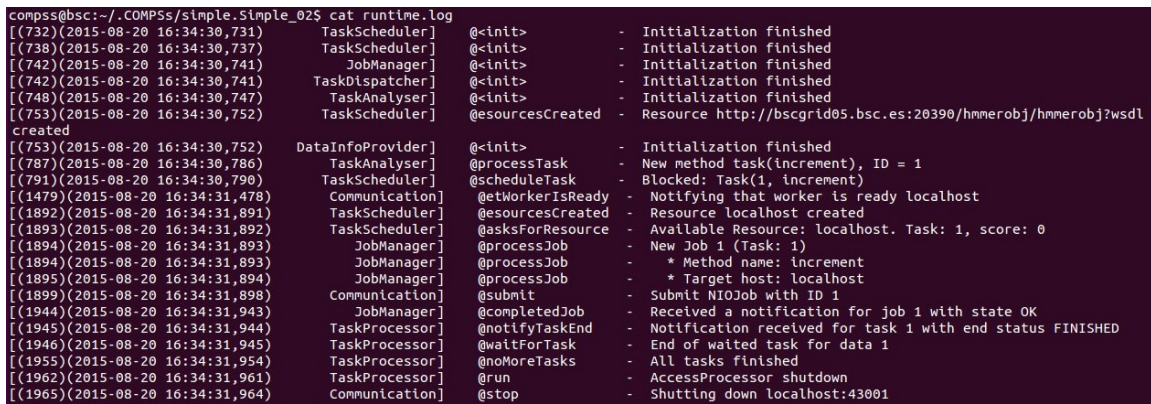


Figure 9: runtime.log generated by the execution of the *Simple* java application

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 and erase these files if needed.

When running Python applications a *pycompss.log* file will appear 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 other runcompss 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 `runcompss` command must be run with the `-g` flag and the final graph will appear at 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
```

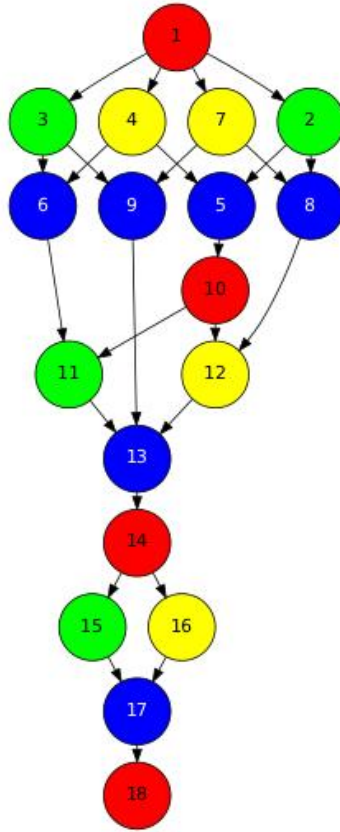


Figure 12: The dependency graph of the SparseLU application

4.2 COMPSs Monitor

The COMPSs Framework exposes a Web Service with a graphical interface that can be used to monitor either finished and running 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.
- `COMPSs_MONITOR_PORT` Port where to run the compss-monitor web service.
- `COMPSs_MONITOR_TIMEOUT` Web page timeout between browser and server.

By default, the `IT_MONITOR` points to the `.COMPSs` folder inside the `root` user, opens the web service in the port 8080 and has a 20s timeout.

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 COMPSs 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 &
```

The COMPSs Monitor allows to monitor applications from different users and thus, users need to login first to grant access to their applications. Once done, as shown in Figure 14, the users can select any of their executed or on-going COMPSs applications and display it.

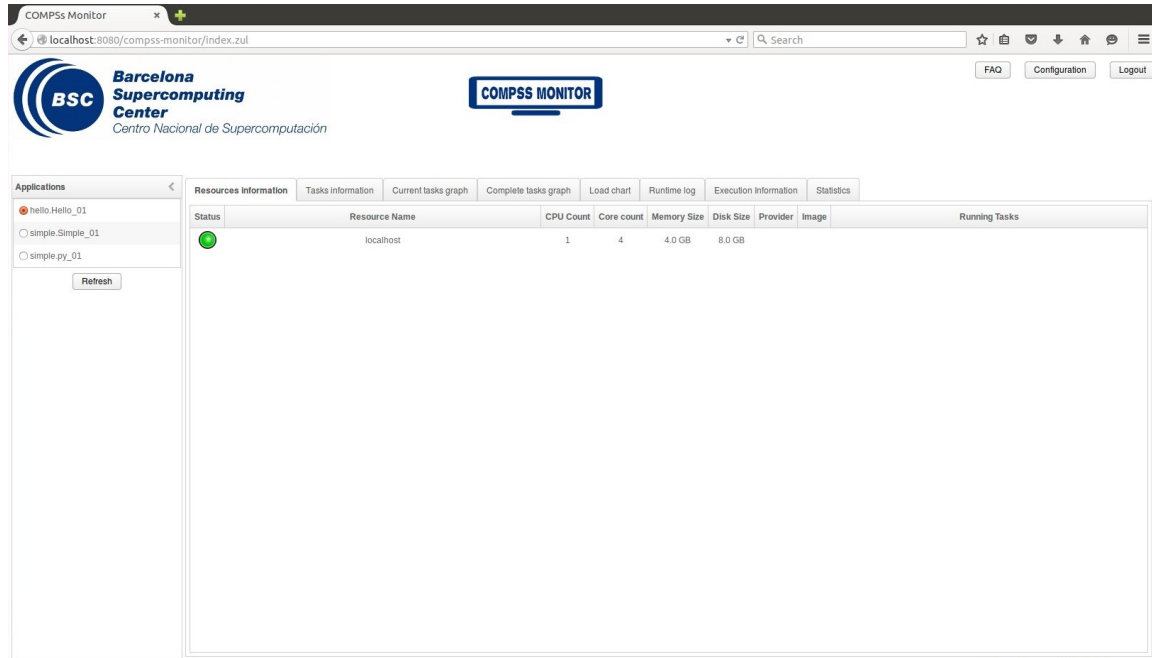


Figure 14: COMPSs monitoring interface

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 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.

```

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

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

WARNING: IT Properties file is null. Setting default values
[ API] - Deploying COMPSSs Runtime v1.3 (build 20150820-1617.rnull)
[ API] - Starting COMPSSs Runtime v1.3 (build 20150820-1617.rnull)
[ API] - Initializing components
[ API] - Ready to process tasks
[ API] - Opening file /home/compss/workspace_java/simple/jar/counter in mode WRITE
[ API] - File URI: any://bsc//home/compss/workspace_java/simple/jar/counter
[ API] - File target Location: /home/compss/workspace_java/simple/jar/counter
Initial counter value is 1
[ API] - Creating task from method increment in simple.SimpleImpl
[ API] - There is 1 parameter
[ API] - Parameter 1 has type FILE_T
Final counter value is 2
[ 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
-----

```

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

```

compss@bsc:~$ cd .COMPSSs/
compss@bsc:~/.COMPSSs$ tree
.
├── simple.Simple_01
│   ├── jobs
│   │   ├── job1_NEW.err
│   │   └── job1_NEW.out
│   ├── monitor
│   │   ├── complete_graph.dot
│   │   ├── COMPSSs_state.xml
│   │   └── current_graph.dot
│   ├── resources.log
│   ├── runtime.log
│   └── tmpFiles

```

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

4.2.3 Graphical Interface features

Next we provide a summary of the COMPSSs 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 COMPSSs Runtime
- **Complete tasks graph**
Shows the complete tasks dependency 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.

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

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.

4.3 Application tracing

COMPSs Runtime can generate a post-execution trace of the distributed 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. Despite the fact that in the current release we do not support file-transfers, we intend to support them in a near future release.

During the execution of the application, an XML file is created at 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 parameter configurations. For further information about the application instrumentation or the trace visualization please check the *COMPSs Tracing Manual* available at our website <http://compss.bsc.es>.

4.3.1 Trace Command

In order to obtain a post-execution trace file the option `-t` must be added to the `runcompss` command. Next we provide an example of the command execution with the tracing option enabled for the Hmmer java application.

```
compss@bsc:~$ runcompss -t --classpath=/home/compss/workspace_java/hmmerobj/jar/hmmerobj.jar
                                     hmmerobj.HMMPfam
                                     /sharedDisk/Hmmer/smart.HMMs.bin /sharedDisk/Hmmer/256seq
```

```

/home/compss/out.txt 2 8 -A 222

----- Executing hmmerobj.HMMPfam -----

WARNING: IT Properties file is null. Setting default values
Welcome to Extrae 3.1.1rc (revision 3360 based on extrae/trunk)
Extrae: Warning! EXTRAE_HOME has not been defined!.
Extrae: Generating intermediate files for Paraver traces.
Extrae: Intermediate files will be stored in /home/compss/workspace_java/hmmerobj/jar
Extrae: Tracing buffer can hold 500000 events
Extrae: Tracing mode is set to: Detail.
Extrae: Successfully initiated with 1 tasks

Extrae: Warning! API tries to initialize more than once
Extrae: Previous initialization was done by API

[ API] - Starting COMPSs Runtime v1.3 (build 20151016-1931.rnull)

...
...
...

[ API] - No more tasks for app 1
[ API] - Getting Result Files 1
[ API] - Execution Finished

Extrae: Intermediate raw trace file created : /home/compss/workspace_java/hmmerobj/jar/
set-0/TRACE@bsc.000003163700000000000000.mpit
Extrae: Intermediate raw sym file created : /home/compss/workspace_java/hmmerobj/jar/set
-0/TRACE@bsc.000003163700000000000000.sym
Extrae: Deallocating memory.
Extrae: Application has ended. Tracing has been terminated.

merger: Output trace format is: Paraver
merger: Extrae 3.1.1rc (revision 3360 based on extrae/trunk)

mpi2prv: Checking for target directory existence... exists, ok!
mpi2prv: Selected output trace format is Paraver
mpi2prv: Stored trace format is Paraver
mpi2prv: Parsing intermediate files
mpi2prv: Removing temporal files... done
mpi2prv: Congratulations! ./trace/hmmerobj.HMMPfam_compss_trace_1440151114.prv has been
generated.

-----

```

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

```

compss@bsc:~$ cd .COMPSs/hmmerobj.HMMPfam/trace/
compss@bsc:~$ ls -l
hmmerobj.HMMPfam_compss_trace_1444922077.pcf
hmmerobj.HMMPfam_compss_trace_1444922077.prv
hmmerobj.HMMPfam_compss_trace_1444922077.row

```

4.3.2 Trace Configurations

The traces generated by an application execution are ready to be visualized with *Paraver*. *Paraver* is a powerful tool developed at the *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.

In Table 7 we provide information about the different pre-build configurations that we distribute with COMPSs and that can be found under the */opt/COMPSs/Dependencies/paraver/cfgs/* folder.

| | |
|-------------------------------------|---|
| <i>2dp_runtime_state.cfg</i> | 2D plot of runtime state |
| <i>2dp_tasks.cfg</i> | 2D plot of tasks duration |
| <i>3dh_duration_runtime.cfg</i> | 3D Histogram of runtime execution |
| <i>3dh_duration_tasks.cfg</i> | 3D Histogram of tasks duration |
| <i>compss_runtime.cfg</i> | Shows COMPSs Runtime events (at master and workers) |
| <i>compss_tasks_and_runtime.cfg</i> | Shows COMPSs Runtime events (at master and workers) and tasks execution |
| <i>compss_tasks.cfg</i> | Shows tasks execution |
| <i>compss_tasks_numbers.cfg</i> | Shows tasks execution by task id |
| <i>compss_transfers.cfg</i> | Shows transfer time spent by each worker |

Table 7: Available paraver configurations for COMPSs Applications

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

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

4.3.3 Trace examples

To end this section, let's present some examples of COMPSs traces. COMPSs traces can be much complex as the number of workers or tasks grows. Just to illustrate this, the following pictures show traces with a greater number of workers and tasks.

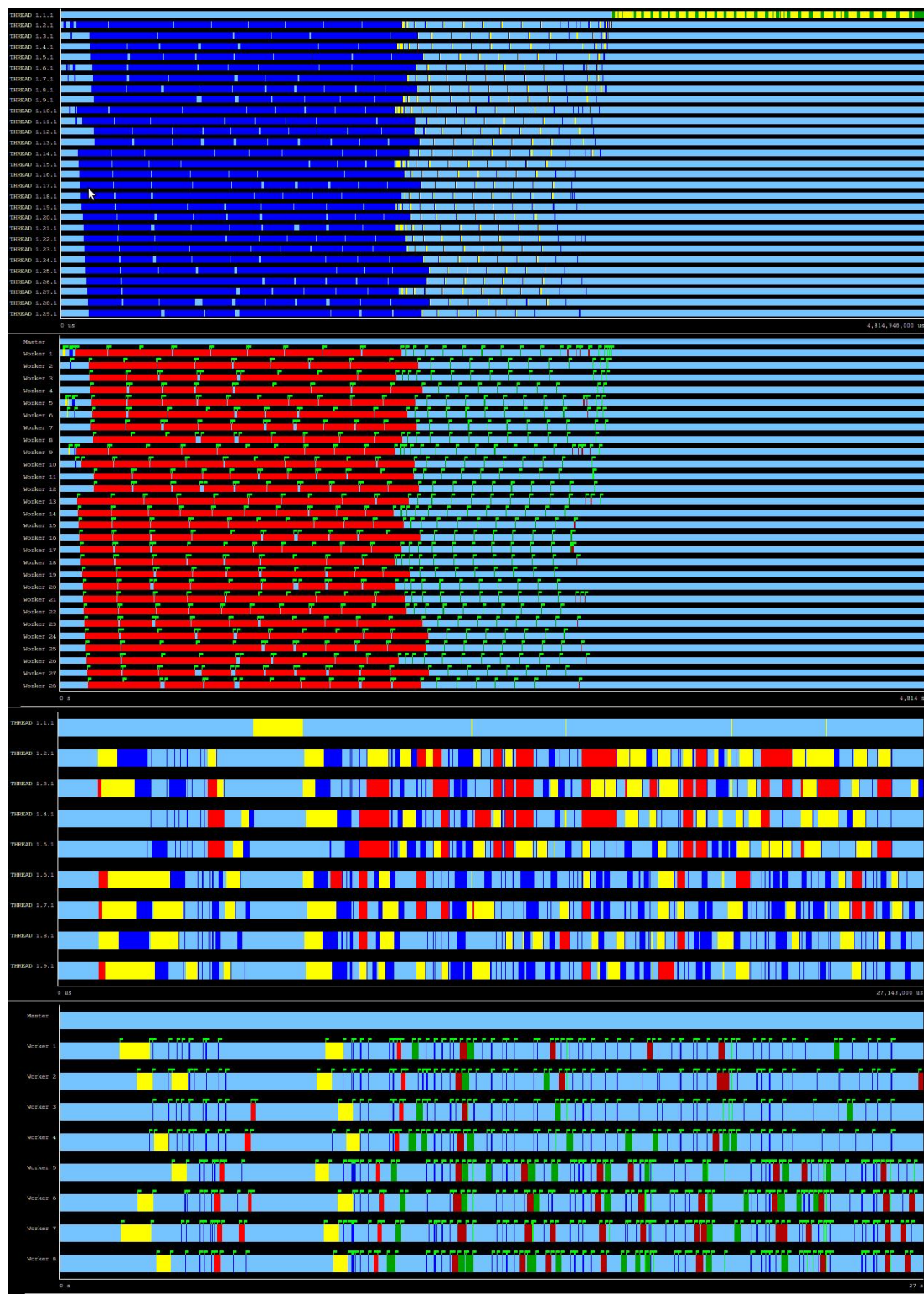


Figure 17: Examples of complex traces

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 our webpage: <http://compss.bsc.es> .

5 Common Issues

This section provides answers for the most common issues that users encounter while beginning to execute COMPSs applications. Nevertheless, if your specific issues are not covered through this section, please do not hesitate to contact us at:

`support-compss@bsc.es`

5.1 How to debug

When the application does not behave as expected the first thing users must do is to run it in **debug** mode. We remember that the `runcompss` command allows to introduce a `-d` flag to enable the debug log level.

Once done, application execution will produce the following files:

- `runtime.log`
- `resources.log`
- `jobs` folder

Firstly, 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. Secondly, 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.

5.2 Tasks are not executed

If the tasks remain in **Blocked** state means that there are no existing resources matching the specific task constraints. This error can be potentially caused by two facts: in one hand, because the resources are not correctly loaded into the runtime and, in the other hand, because the task constraints do not really match with any resource.

In the first case, users should take a look at the `resources.log` and check that all the resources defined in the `project.xml` file are available for 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.

5.3 Jobs systematically fail

If all the application tasks fail because all the submitted jobs fail, it is probably due to the fact that there is a resource misconfiguration. In most of the cases, the resource that the application is trying to access has no passwordless access through the configured user. Users can try if this is their case by executing the following steps:

- Open the `project.xml` that the application is using. Remember that 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 has been a passwordless missconfiguration.

If there has been a passwordless missconfiguration you can solve it by 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>.

5.4 Compilation error: @Method not found

When trying to compile java applications users 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 available in the CLASSPATH. The default COMPSs installation automatically inserts this package into the CLASSPATH but some users may have overwritten or deleted it. Please check that your environment variable CLASSPATH contains the *compss-engine.jar* location by running the following command:

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

If the result of the previous command is empty it means that you are missing the *compss-engine.jar* package in your classpath.

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

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

However, you will need to remember to export this variable every time you log out and back in again. Consequently, we 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.

5.5 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 caused because 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
- The tasks have not been defined as *public*

5.6 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 error is caused because 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).

Please find more details on the COMPSs framework at

`http://compss.bsc.es`