COMP Superscalar

User Guide

Barcelona Supercomputing Center

Centro Nacional de Supercomputación

Chapter 1 Introduction

1.1 What is COMP Superscalar?

COMP Superscalar is a new version of GRID Superscalar which aims to easing the development of Grid applications. COMP Superscalar exploits the inherent parallelism of applications when running them on the Grid. However, with respect to its predecessor, COMP Superscalar has three main distinctive features:

- The runtime of COMP Superscalar is formed by a set of *components*, each one in charge of a given functionality. This componentised runtime follows the Grid Component Model (GCM), a component model especially designed for the Grid whose reference implementation is provided by ProActive.

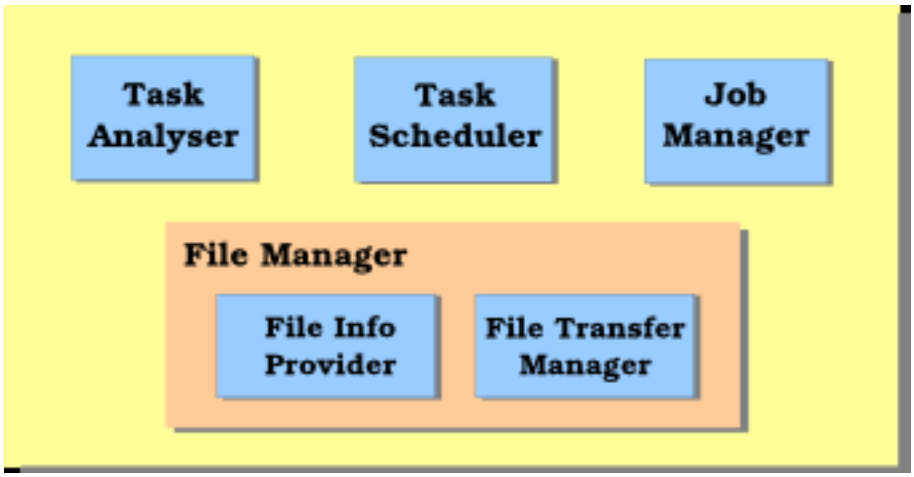
- COMP Superscalar offers a straightforward programming model that targets Java and C/C++ applications. The simplicity of this programming model keeps the Grid transparent to the user, who is able to program his applications in a Grid-unaware fashion. The user is only required to select the tasks to be run on the Grid, while the application can remain completely free of Grid-related calls.

- COMP Superscalar can use a wide range of Grid middlewares thanks to the JavaGAT API. JavaGAT provides COMP Superscalar with a uniform interface for job submission and file transfer operations, being able to choose between different middlewares like Globus, UNICORE or SSH.

1.2 COMPSs component design

The runtime of COMP Superscalar presents a componentised structure that is based on the Grid Component Model (GCM). Each of the runtime subcomponents encapsulates a given functionality that contributes to the overall execution of the application.

An important feature of the GCM-based runtime is that the subcomponents can easily be deployed and distributed among different hosts, so that the processing of the application tasks is parallelized and the throughput of the runtime increases.



1.2.1.Task Analyzer

It receives incoming tasks from the application and detects their precedence, building a task dependency graph. Like in GRIDSs, a COMPSs task is a method invocation, made from the application code, that will be executed on the Grid.

When requested to process a task, this component looks for data dependencies between the new task and all previous ones. Once a task has all its dependencies solved, the Task Analyzer sends it to the Task Scheduler.

1.2.2. Task Scheduler

It decides where to execute the dependency-free tasks received from the Task Analyzer. This decision is made according to a certain scheduling algorithm and taking into account three information sources: first, the available Grid resources and their capabilities; second, a set of user-defined task constraints; and third, the location of the data required by the task. The scheduling strategy could be changed on demand, thanks to the dynamic and reconfigurable features of GCM.

1.2.3. Job Manager

It is in charge of job submission and monitoring. It receives the scheduled tasks from the Task Scheduler and delegates the necessary file transfers to the File Manager. When the transfers for a task are completed, it transforms the task into a Grid job in order to submit it for execution on the Grid, and then controls the proper completion of the job. It could implement some fault-tolerance mechanisms in response to a job failure.

1.2.4. File Manager

It takes care of all the operations where files are involved. It is a composite component which encompasses the File Information Provider and the File Transfer Manager components. The former gathers all information related with files: what kind of file accesses have been done, which versions of each file exist and where they are located. The latter is the component that actually transfers the files from one host to another; it also informs the File Information Provider about the new location of files.

Chapter 2 Programming with COMPSs

2.1 Configuration Files

The following configuration files have to be edited to prior to execution

- resource_list
- project.xml
- resources.xml

2.1.1. Resources List

Resources List file contains the information to access the remote nodes where the workers have to be deployed (username, hostname and directory where to copy the files).

```
user@node1:~
user@node2:~
user@node3:~
```

2.1.2.Project File

This file indicates at COMPSs some information about each worker node that will be used by our application.

It contains information about in which directory we could find the worker binary, the worker files, the user who executes the application and the limit of simultaneous tasks that will be executed on each node.

The following example illustrates how have to be coded this information in case to deal with only two workers (node2 and node3) and one master (node1).

For each worker there must be defined:

- InstallDir: Where the worker will be deployed.
- WorkingDir: Where all temporary files will be created.
- User: Machine user (it needs to have his/her pair of keys created and exported.)
- Limit of task: Maximum number of task that can run simultaneously, it's preferable to choose this number equal at number of cores that has the node.

```
<Project>
 <Worker Name="node2">
  <InstallDir>/home/user1/IT_worker/</InstallDir>
  <WorkingDir>/home/user1/IT_worker/files/</WorkingDir>
  <User>user1</User>
  <LimitOfTasks>2</LimitOfTasks>
</Worker>

<Worker Name="node3">
  <InstallDir>/home/user1/IT_worker/</InstallDir>
  <WorkingDir>/home/user1/IT_worker/files/</WorkingDir>
  <User>user1</User>
  <LimitOfTasks>2</LimitOfTasks>
</Worker> </Project>
```

2.1.3. Resources File

This file contains information about the capabilities of each machine; such information is used by the COMPSs runtime for scheduling purposes.

The file format follows the Information Modelling approach, which is currently being standardized by Open Grid Services Architecture WG of the Open Grid Forum.

The attributes allowed are:

- **Host:** Maximum number of task that will accept simultaneously and queues length.
- **Processor:** Number of CPUs, their frequency and architecture.
- **Operative System:** Type (Windows or Linux) and the maximum number of processes per user.
- **Storage Elements:** Global storage size and Access Time.
- **Memory:** Physical and Virtual size or even Time Access.
- **Installed Software:** Software that have to be installed on the node that will execute this job.
- **Service:** Services that must be running in order to allocate the job on the node.
- **Cluster:** Cluster identification name where the Grid node is.
- **FileSystem:** FileSystem kind restriction that have to be running on the worker node.
- **Network Adaptor:** The network adaptor manufacturer name that this node fits.
- **Job Policy:** The policy that has to accomplish the Job in order to be executed on the resource.
- **Access Control Policy**: The Access Policy Scheme like Virtual Organization Membership Service (VOMS).

We can also indicate the requirements for each of the node.

The following picture illustrates an example of resources.xml with one worker.

```
<ResourceList>
 <Resource Name="node2">
  <Capabilities>
    <Host>
     <TaskCount>0</TaskCount>
     <Queue>short</Queue>
    </Host>
    <Processor>
     <Architecture>Intel</Architecture>
     <Speed>3.6</Speed>
     <CPUCount>2</CPUCount>
    </Processor>
    <OS>
     <OSType>Linux</OSType>
     <MaxProcessesPerUser>32</MaxProcessesPerUser>
    </OS>
    <StorageElement>
     <Size>60</Size>
    </StorageElement>
    <Memory>
     <PhysicalSize>0.5</PhysicalSize>
     <VirtualSize>8</VirtualSize>
    </Memory>
    <ApplicationSoftware>
     <Software>Xerces</Software>
     <Software>Xalan</Software>
    </ApplicationSoftware>
    <Service/>
    <VO/>
    <Cluster/>
    <FileSystem/>
    <NetworkAdaptor/>
    <JobPolicy/>
    <AccessControlPolicy/>
  </Capabilities>
  <Requirements/>
 </Resource>
</ResourceList>
```

Note: this files are not required in case of batch execution of the COMPSs application, please refer to Appendix A for more details.

2.2 Environment variables

Some variable has to be set in the user's environment; please ask your system administrator for the actual values or refer to the installation guide. Assuming a COMPSs distribution under the /apps/COMPSs directory, the following has to be executed:

```
user@node:~ export IT_HOME=/apps/COMPSs
user@node:~ export PROACTIVE_HOME=
user@node:~ export GAT_LOCATION=
user@node:~ export CLASSPATH=$CLASSPATH:$IT_HOME/lib/IT.jar
user@node:~ export GS_HOME=$IT_HOME/bindinglib
user@node:~ export PATH=$PATH:$GS_HOME/bin
user@node:~ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$GS_HOME/lib
```

2.3 Develop Java Applications

In this section the steps required in the development of a Java COMPSs application will be illustrated; a counter sequential application will be used to explain the porting to COMPSs.

2.3.1. Reorganizing the code

A COMPSs application is composed by two parts:

- Master Application: this is the main code which will be executed on the Master Node and contains the calls to the user-selected methods
- Worker Tasks: it corresponds to the implementation of the tasks. These tasks will be run on a remote worker host.

Java interface is used to declare the methods to be executed on the Grid, along with Java annotations that specify necessary metadata about the tasks. These metadata can be of three different kinds:

1. For each parameter of each method, its type (currently, we support the file type, the primitive types and the string type) and its direction (IN, OUT or INOUT).
2. The Java class that contains the code of the method.
3. The constraints that a given resource must fulfil to execute a certain method, such as the required operating system or storage capacity.

In our example, we divide the code in 3 different files:

- sequential.Simple.Simple.java (Main): it corresponds to the main sequential code of the application. The user can select any method called from this code to be run as a remote task.

- worker.Simple.SimpleImpl.java (Worker): it contains the implementation of the user-selected tasks.

- SimpleItf.java (Java Annotated Interface): it declares the methods selected to be run as tasks and some metadata about them.

2.3.2. Task Selection and Interface creation

First of all, the user has to select the methods to be run on the Grid as remote tasks. This is done providing a Java interface which declares such methods, along with some necessary metadata in the form of Java annotations.

Next we give more details about the task metadata.

Method-level Metadata: for each selected method, we have the following metadata:

- **@ClassName:** mandatory. It specifies the class that implements the method.
- **@MethodConstraints**: optional. The user can specify the capabilities that a resource must have in order to run a method. In particular, he can specify the following requirements:
- **Host:**
    - Queue (hostQueue) : Name of the queue where task will be submitted.
- **Processor:**
    - Number of processors (processorCPUCount): minimum number of processors .
    - Speed (processorSpeed): minimum clock frequency in Ghz.
    - Architecture(processorArchitecture): User-defined (Intel, AMD, ppc, i586, i386, …).
- **Memory:**
    - Physical Size (memoryPhysicalSize): Amount of GB of Physical memory.
    - Virtual Size (memoryVirtualSize): Amount of GB of Virtual memory.
    - Access Time (memoryAccessTime): ns to access to data.
    - STR (memorySTR): memory transmission rate in GB/s.
- **Storage:**
    - Size (storageElemSize): Amount of GB of Storage element.
    - AccessTime (storageElemAccessTime): ns to access to data stored in the element.
    - STR (storageElemSTR): storage element transmission rate in GB/s.
- **Operating System:**
    - Type (operatingSystemType) could be Linux or Windows.
- **Software (appSoftware)**: Free string to set any kind of Software applications separated by comma ",".

Parameter-level Metadata (@ParamMetadata): for each parameter of each method, the user must define:

- **Direction:** Direction.IN, Direction.INOUT or Direction.OUT in Java are always passed by value.
- **String**: Type.STRING. It can only have IN direction, since Java Strings are immutable.
- **File:** Type.FILE. It can have any direction (IN, OUT or INOUT). The real Java type associated with a FILE parameter is a String which contains the path to the file. However, if the user specifies a parameter as a FILE, COMPSs will treat it as such.
- **Other types:** although COMPSs does not support objects as task parameters yet, the user has the possibility to marshal an object into a file and pass it as a task parameter of FILE type. The object has to be unmarshalled by the task code.
-
- **Type:** COMPSs supports the following types for task parameters:
  Basic types: Type.BOOLEAN, Type.CHAR, Type.BYTE, Type.SHORT, Type.INT, Type.LONG, Type.FLOAT, Type.DOUBLE. They can only have IN direction, since primitive types in Java are always passed by value.

- **String**: Type.STRING. It can only have IN direction, since Java Strings are immutable.
- **File:** Type.FILE. It can have any direction (IN, OUT or INOUT). The real Java type associated with a FILE parameter is a String which contains the path to the file. However, if the user specifies a parameter as a FILE, COMPSs will treat it as such.
- **Other types:** although COMPSs does not support objects as task parameters yet, the user has the possibility to marshal an object into a file and pass it as a task parameter of FILE type. The object has to be unmarshalled by the task code.

Other restrictions about the task methods:

- **Return type:** always VOID.
- **Method modifiers:** the method has to be STATIC.

In the Simple application example, the method that will be executed on the Grid is Increment():

- **Increment():** Increments the value of a counter stored on a file.

The Increment implementation can be found in worker.Simple.SimpleImpl.class and needs a single input parameter: a string containing a path to the file counterFile. Besides, in this example there is a constraint on the operative system of the node.

```
package sequential.simple;
import integratedtoolkit.types.annotations.*;
import integratedtoolkit.types.annotations.ParamMetadata.*;

public interface SimpleItf {
   @MethodConstraints(operatingSystemType = "Linux")
   @ClassName("worker.simple.SimpleImpl")

   void increment(
     @ParamMetadata(type = Type.FILE, direction = Direction.INOUT)
     String file
   );
}
```

At the end of the execution this file contains the maximum value of the counter.

Application Code

COMPSs offers to the programmer the possibility of leaving the application completely unchanged, i.e. no API calls need to be included in the main application code in order to run the selected tasks on the Grid. COMPSs will be in charge of, on the fly, replace the invocations to the user-selected methods by the creation of remote tasks. Moreover, the COMPSs will be automatically started and stopped at the beginning and at the end of the application, respectively. Regarding the access to files from the main application code, it will be taken care of by COMPSs as well.

The code below shows the main code of the sequential Simple application. COMPSs is able to take this sequential code and replace, at execution time, the call to the Increment() method by the creation of a task.

```
package sequential.simple;

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

import worker.simple.SimpleImpl;

public class Simple {

  public static void main(String[] args) {
    String counterName = "counter.txt";
    int initialValue = 1;

    /*-------------------------------------------------------------
    Creation of the file which will contain the counter variable
    -------------------------------------------------------------*/
    try {
      FileOutputStream fos = new FileOutputStream(counterName);
      fos.write(initialValue);
      System.out.println("Initial counter value is " + initialValue);
      fos.close();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
    /*-----------------------------------------------
              Execution of the program
    -----------------------------------------------*/
           SimpleImpl.increment(counterName);
    /*-----------------------------------------------
          Reading from an object stored in a File
    -----------------------------------------------*/
    try {
      FileInputStream fis = new FileInputStream(counterName);
      System.out.println("Final counter value is " + fis.read());
      fis.close();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

On the worker side we have the following code:

```
package worker.simple;

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

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

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

Optional: Including API calls

As explained above, COMPSs does not require the user to modify the main code of the application. However, COMPSs also offers the possibility to use up to 3 API methods in the application code. In particular, the API offers methods to start and stop runtime and to open files to work with them locally:

- **startIT():** Starts the runtime.
- **stopIT(boolean):** Stops the runtime. The boolean indicates if the runtime must be terminated (true) or if it will be restarted later (false).
- **openFile (String OpenMode):** Opens a file to work locally with it. This operation needs the path of the file and the open mode (OpenMode.READ, OpenMode.WRITE, OpenMode.APPEND).

Using the API gives the programmer more control over the application, more precisely in two ways: on the one hand, the runtime can be stopped at any point of the application and optionally restarted again later, which allows the programmer to execute some parts of the application locally and sequentially.

On the other hand, when working with a file, the access mode can be specified so that COMPSs could know if another version of that file is going to be generated.

The below code is an example of the inclusion of API calls:

```
package sequential.simple;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import worker.simple.SimpleImpl;
public class Simple {

public static void main(String[] args) {
    String counterName = "counter.txt";
    int initialValue = 1;

    /*---------------------------------------------------------
    Creation of the file which will contain the counter variable
    ---------------------------------------------------------*/
    try {
      FileOutputStream fos = new FileOutputStream(counterName);
      fos.write(initialValue);
      System.out.println("Initial counter value is " + initialValue);
      fos.close();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
    /*-------------------------------------------------
              Execution of the program
    -------------------------------------------------*/
    // Start IT
    IntegratedToolkit it = new IntegratedToolkitImpl();
    it.startIT();
    // Execution
    SimpleImpl.increment(counterName);
    // Stop  IT
    it.stopIT(true);
    /*-------------------------------------------------
          Reading from an object stored in a File
    -------------------------------------------------*/
    try {
      FileInputStream fis = new FileInputStream(counterName);
      System.out.println("Final counter value is " + fis.read());
      fis.close();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

2.3.3. Compiling and Deploying the Application

Once the code is modified as shown in the previous example, we are able to compile and execute the application.

In order to get an execution-ready application first of all we have to copy the application source code in the Master side node. The source code files have to be structured in ***$IT_HOME/gridunawareapps/src*** folder according to the example of the following table.

In this example case, we could find various applications: Cholesky, Matmul and Simple.

We have to distribute our application files following the next classification. On ***/src/worker*** we must be able to find all the classes containing the worker code. Depending on how main program uses COMPSs we can distribute the main program files in two different directories. If the main program has no calls to API, ***/src/sequential*** will contain the main program code and the Java Annotated Interface.

On the other hand, if main program uses the partial loader we should place the files in ***/src/api***.

In our example, we copied Simple.java and SimpleItf.java files on ***/src/sequential*** and ***/src/api*** directory and SimpleImpl.java to ***/src/worker*** directory.

/src/api/	/src/sequential	/src/worker
Simple/ **Simple.java** **SimpleItf.java** Cholesky/ Cholesky.java CholeskyItf.java Matmul/ Matmul.java MatmulItf.java	**Simple/** **Simple.java** **SimpleItf.java** Cholesky/ Cholesky.java CholeskyItf.java Matmul/ Matmul.java MatmulItf.java	**Simple/** **SimpleImpl.java** Cholesky/ CholeskyImpl.java CholeskyAppException.java Block.java Matmul/ MatmulImpl.java MatmulAppException.java Block.java

We can now compile the applications that are inside the directories typing in ***$IT_HOME/*** path: ***"ant guapp"***. This command will generate all the application .class files compiling it with javac and then will save all this files in packages inside the build directory (that could be found in: ***$IT_HOME/gridunawareapps/build/***).

Then automatically generate the .jar file leaving it in lib directory (***$IT_HOME/gridunawareapps/lib/***).

When the compile part is done, we are ready to deploy the worker application part to Grid nodes. By typing ***"ant worker"*** from the ***$IT_HOME/*** folder, the worker application part is automatically deployed to the Grid nodes that have been previously specified on the resources_list file.

Therefore, we finally have the application ready to be executed on the Grid.

2.3.4.Executing the Application on the GRID

Run an application on COMPSs it is a quite simple job. There's a script on gridunawareapps directory called *guapp.sh* which let us execute any application in a simple way.

The script has 6 parameters:

1. **mode:** We could choose between two modes:

- Sequential: It executes the code on the master in a sequential way without using the framework toolkit.
- IT: Uses the toolkit, so code will run on master and workers depending on the configuration previously done.

2. **projFile**: Absolute file path to the xml file containing the Project File described above.
3. **resFile:** Absolute path to the xml file containing the Resources File described on 4.1.2.
4. **loader**: User can choose two kinds of loader:
   - Total: There are no direct calls to the API.
   - Partial: There are calls to the API
5. **fullAppPath**: Packages and class where COMPSs will find the main method of the application that we want to run.
6. **AppParameters:** Parameters for the application.

In our example, we have to launch the execution through the next command:

```
./guapp.sh IT /home/user/COMPSs/xml/projects/project.xml
/home/user/COMPSs/xml/resources/resources.xml total
sequential.simple.Simple
```

2.3.5.Sample Applications

This section has illustrated the COMPSs programming model by means of a simple example. More sample applications can be found in the **$IT_HOME/gridunawareapps/src/** folder.

2.4 Develop C/C++ Applications

2.4.1. Task Selection and Interface Creation

The user has to select the methods to be run on the Grid as remote tasks. For that purpose, he has to provide an interface file which declares such methods, along with some necessary parameter metadata.

When choosing this tasks, it is important to note that COMPSs has some restrictions in task DataType parameters.

IN parameters:

COMPSs only supports basic types. To use complex types a *parameter marshalling* is needed building a String that contains the path to a File where COMPSs could find the object.

OUT parameters:

Currently COMPSs only supports File type as OUT parameter of a method.

INOUT parameters:

It works similar to OUT parameters, but in this case the file must exist in order to read from it.

RETURN type:

RETURN type is always void. If you want to return some object, an OUT parameter is needed.

COMPSs supports the following basic type parameters:

- **Basic types:** *char, boolean, short, long, longlong, int, float, double, file.*
- **String:** If it is used as an input parameter containing some value we must use the *string* type. If it is used as a path to a File it must be set to *file* type.

In Simple application (that will be used as section example), the function that will be executed on the Grid is Increment().

Increment needs a single input parameter which is going to be a string containing a filePath to the counterFile.

There is an INOUT parameter which contains the value of the counter. At the end of the execution this value contains the maximum value of the counter. We also will need a file parameter that will be the input/output parameter of increment method.

The Simple application interface has to be defined as the following way in a text file named *myapplication.idl*:

```
interface INCREMENT  {
        void increment(inout File filename);
};
```

2.4.2.Reorganizing the code

This section covers the development of C/C++ applications with COMPSs.

Dividing the Code:

The sequential application has to be divided in two parts, the Master Code that contains the calls to the remote methods, and the Worker Code that implements the remote functions.

Master Code

This part of the application, make calls to the functions that will be executed on the worker nodes.

Typically, this is composed by the ***myapplication.cc*** which is the main code written by the application developer.

The master code file extension MUST be .cc (C++ code)

Interface Definition File

This file should be present in the master and worker directory during the building procedure as we have seen in the previous item.

Worker Code

The part of the application that is executed on the worker nodes.

This is composed by the ***myapplication-functions.c***, where the functions defined in myapplication.idl file are implemented.

Organizing the Code:

In our example, The Simple application, we have to modify some aspects of the old sequential code in order to fit in COMPSs framework.

In order to achieve it, we need to modify the original sequential application:

```
#include <time.h>
#include <stdio.h>
#include <errno.h>
#include "GS_compss.h" //Added code
#include "Simple.h"    //Added code

int main(int argc, char **argv)
{
  long int t = time(NULL);
  FILE *fp;
  char filename[15]="counter.txt";
  int initialValue = 1;
  int finalValue=0;
 GS_On(PRJ_FILE, RES_FILE, MASTER_DIR, APPNAME); //Added code

  increment(filename);

 fp = fopen(filename, "r");
  fscanf (fp,"%d",&finalValue);
  printf("Final Counter Value is: %d \n", finalValue);
  fclose(fp);
  GS_Off(0); //Added code

  fp = fopen(filename, "w");
  fprintf(fp, "%d", initialValue);
  printf("Initial Counter Value is: %d \n", initialValue);
  fclose(fp);

  printf("Total time:\n");
  t = time(NULL) - t;

  printf("%li Hours, %li Minutes, %li Seconds\n", t / 3600, (t %
3600)  / 60, (t % 3600) % 60);

  return 0;
}
```

As shown in the previous Figure, we need to include the ***GS_compss.h*** and ***Simple.h*** files.

The *Simple.h* is automatically generated by COMPSs so *Simple.h* has to be renamed as your own application name like *appname.h*. ***GS_On(PRJ_FILE, RES_FILE, MASTER_DIR, APPNAME)*** and ***GS_Off(0)*** directives indicates to start and stop the COMPSs runtime. In the worker, where the functions are actually implemented, the include has to be added.

```
#include <stdio.h>
#include <errno.h>
#include "Simple.h" //Added code


void increment(char *filename)
{
int counterValue=0;

FILE *fp;

fp = fopen(filename, "r");
fscanf (fp,"%d",&counterValue);
fclose(fp);

counterValue++;

fp = fopen(filename, "w");
fprintf(fp, "%d", counterValue);
fclose(fp);

}
```

2.4.3. Compiling and Deploying the Application

This section contains the steps to compile and deploy an example application called Simple.

Before starting the compilation the user has to create a master and a worker directory; in this figure example also a *files* directory is created as working dir.

Actually, the working directory could take any name and could be allocated according to the user preference. Then, the working directory path specified in project.xml file have to be changed.

```
user@loginnode:~> mkdir simple_app
user@loginnode:~> cd simple_app
user@loginnode:~/simple_app> mkdir master
user@loginnode:~/simple_app> mkdir worker
user@loginnode:~/simple_app>
user@loginnode:~/simple_app> cd worker/
user@loginnode:~/simple_app/worker> mkdir files
user@loginnode:~/simple_app/worker>
```

Copy the Simple.cc and Simple.idl files in the master directory, and Simple-functions.c and Simple.idl in the worker directory.

```
user@loginnode:~> cd simple_app/
user@loginnode:~/simple_app> ls
master project.xml resources.xml worker
user@loginnode:~/simple_app> cd master/
user@loginnode:~/simple_app/master> ls
Simple.cc Simple.idl
user@loginnode:~/simple_app/master> cd ..
user@loginnode:~/simple_app> cd worker/
user@loginnode:~/simple_app/worker> ls
files Simple-functions.c Simple.idl
user@loginnode:~/simple_app/worker>
```

Compile the master and worker code using the ***gsbuild*** command.

The usage of this command is shown in help:

```
user@loginnode:~/simple_app/worker> gsbuild
Usage: gsbuild build <component> <appname> <project_path>
<resources_path>
Usage: gsbuild copy <component> <appname>
Usage: gsbuild clean <component> <appname>

Available actions:
copy Setup a compilation environment for the component for
customization.
build Build the selected component.
clean Remove generated binaries.

Available components:
master Build or copy the master part.
worker Build or copy the worker part.
all Build or copy the master and workers parts.

<appname> Corresponds to the name of the application used for source
files
and IDL files.
<projectpath> Corresponds to the path of the project description file.
<resourcespath> Corresponds to the path of the resources description
file.
```

There are 3 available actions when using the gsbuild command: copy, build and clean.

Using the copy action of gsbuild:

Compiling the Master:

The gsbuild copy action generates all the files used to compile the application.

This is usually used during the development of the application when editing on Makefile may be needed.

```
user@loginnode:~/simple_app> gsbuild copy master simple
```

This will copy all the files needed to compile the master code of the application in the master directory; then the *autogen.sh* script has to be executed with the project and resources xml files as parameters.

```
user@loginnode:~/simple_app>./autogen.sh
/home/user/simple_app/project.xml/home/user/simple_app/resources.xml
```

Finally, the master code can be compiled typing:

```
user@loginnode:~/simple_app/master> make
```

The last two steps have to be executed every time any change is applied to Makefile.am or configure.in,

Compiling the worker:

```
user@loginnode:~/simple_app> gsbuild copy worker Simple
```

To compile the worker code the same steps of the master have to be followed with the only difference that the *autogen.sh* doesn't require parameters.

```
user@loginnode:~/simple_app/worker>./autogen.sh
```

If the Makefile.am or configure.ac are edited after the compilation, the autogen.sh script has to be executed again and the code compiled as well.

There is the possibility to generate all the environment files (master and worker) using the ***all*** option of the copy action :

```
user@loginnode:~/simple_app> gsbuild copy all Simple
```

```
user@loginnode:~/simple_app> ./autogen.sh
```

Using the build action of gsbuild:

The build action generates and compiles master and worker in a completely automatic way.

This command creates the master/worker directory, copies the required files for the compilation, compiles the code and cleans the directory removing the intermediate files.

This can be useful if there is no need to modify build files (Makefile, configure, etc) and/or to separate master and worker source files (and the *all* option of *build* action is used).

```
user@loginnode:~/simple_app> ls
project.xml resources.xml Simple.cc Simple-functions.c Simple.idl
```

Building the Master:

In order to generate the master component only, we need to execute the following command:

```
user@loginnode:~/simple_app>gsbuild build master simple
\/home/user/simple_app/project.xml /home/user/simple_app/resources.xml
Building master
Running Autogen with:
Project File: /home/user/simple_app/project.xml
Resources File: /home/user/simple_app/resources.xml
```

Note: The project and resources files paths must be absolute.

```
user@loginnode:~/simple_app> ls
master project.xml resources.xml Simple.cc Simple-functions.c
Simple.idl
user@loginnode:~/simple_app> cd master
user@loginnode:~/simple_app/master> ls
config_master.sh Simple Simple.cc Simple.idl
```

Building the Worker:

In order to build the worker, we have to proceed similarly as the previous master example:

```
user@loginnode:~/simple_app> ls
master project.xml resources.xml Simple.cc Simple-functions.c
Simple.idl

user@loginnode:~/simple_app> gsbuild build worker Simple
/home/user/simple_app/project.xml /home/user/simple_app/resources.xml
Building worker
Running Autogen...
```

When the compilation process finishes the worker directory should look like the following picture:

```
user@loginnode:~/simple_app/worker> ls
config_worker.sh Simple-functions.c Simple.idl Simple-worker workerGS
workerGS_script.sh workerGS.sh
```

There is the possibility to build master and worker code at once using the ***all*** option of the build action:

```
user@loginnode:~/simple_app> ls
project.xml resources.xml Simple.cc Simple-functions.c Simple.idl
user@loginnode:~/simple_app> gsbuild build all Simple
/home/user/simple_app/project.xml /home/user/simple_app/resources.xml
Building all:
Building Master...
Running Autogen with:
Project File: /home/user/simple_app/project.xml
Resources File: /home/user/simple_app/resources.xml
```

At the end of building process the application directory should look like:

```
...
Command successful.

user@loginnode:~/simple_app> ls
master project.xml resources.xml Simple.cc Simple-functions.c
Simple.idl worker
```

Using the clean action of gsbuild:

The clean action deletes all the binary files generated by the build process; it can be used with the ***master***, ***worker*** *and* ***all*** options.

2.4.4. Executing The Application

Executing the application in interactive mode:

To execute a C/C++ COMPSs application in interactive mode simply invoke the myapplication executable:

```
user@node:~/simple_app> cd master
user@node:~/simple_app/master> ./simple
```

The log of the execution can be found in /user/home/it.log.

Appendix A Executing the application in a queue management system:

The current version of COMPSs only supports the *Slurm+MOAB* batch processing system. To execute a COMPSs application the *submit_compss.sh* script has to be used; this script is copied in the master directory by the gsbuild command. The script will automatically create the resources.xml and project.xml files using the pool of resources assigned by the queue system.

For Java applications the usage of the script is the following:

```
user@node:~/simple_app> ./submit_compss.sh NPROCS WALL_CLOCK_LIMIT
LOADER APPNAME ARGS
```

Where:

- NPROCS: number of processors to use.
- WALL_CLOCK_LIMIT: the limit of wall clock time. It must be set it to a value greater than the real execution time for your application and smaller than the time limits granted to the user.
- LOADER: 'partial' or 'total' (see 2.3.4).
- APPNAME: fully qualified name of the class implementing the Main() of the application.
- ARGS: the arguments of the application.

For C/C++ applications:

```
user@loginnode:~/simple_app> ./submit_compss.sh NPROCS
WALL_CLOCK_LIMIT EXECUTABLE ARGS
```

Where:

- NPROCS: number of processors to use.
- WALL_CLOCK_LIMIT: the limit of wall clock time. It must be set it to a value greater than the real execution time for your application and smaller than the time limits granted to the user.
- EXECUTABLE: absolute path of the master binary
- ARGS: the arguments of the application.

This script will generate a *slurm_script.sh* script that has to be used as input to the SLURM wrapper command for enqueueing the job.