



GRAM Tutorial - Part II

NGS Induction Event, Guy Warner, NeSC Training Team



This tutorial extends the previous tutorial into the more advanced topics of the globus GRAM API (C and Java), multiple jobs running concurrently, and using the Message Passing Interface (MPI). The assumption is made that an understanding of job submission and compiling code was acquired from the previous tutorial. Please consult the previous tutorial, which may be found [here](#), if in any doubt.

This tutorial optionally involves some editing of files. To use any graphical tool you must first launch Exceed. After the splash screen no further screen will appear - this is what is meant to happen. You should also ensure that your Putty session has X11 forwarding enabled. To edit a file use the command

```
kwwrite <filename> &
```

where <filename> is the name of the file you wish to edit. You can safely ignore any messages that appear in your terminal window.

The C API

1. Open an ssh session onto training-ui.nesc.ed.ac.uk. Before starting the tutorial it is necessary to ensure you have a running grid-proxy (follow this [link](#) for how to renew your proxy).
2. The first topic for this tutorial is a simple example of the globus API. Whilst this tutorial covers in more detail the Java API, how to compile code using the C API is first shown. The aim is to only show the stages needed to compile code that uses the C API since this involves more steps than for the Java API. A guide to the functions of the GRAM C API may be found at http://www-unix.globus.org/api/c/globus_gram_client/html/index.html.

Before starting this section change to the directory in your account containing files needed for this section of the tutorial:

```
cd ~/gram2/c_api
```

Any program that needs to be compiled against an API needs to find the relevant header files (files containing the definitions of the functions), the libraries of the functions and any other relevant definitions. The Globus toolkit provides a method of generating a file containing all the necessary definitions. This file may be used from within a Makefile (a file used by the "make" program to help automate the process of building an application - very similar to the "ant" program for java code) to compile the code. Since globus is a tool available across the spectrum of unix derivatives it needs to be able to handle different variations of compiler and binary format (called flavor's in globus terminology). To compile any code you must select an appropriate for which a run time library (rtl) and development version (dev) are available. Availability of flavors and different configurations can also depend on which parts of the globus API the code being developed will use. For this tutorial one part only is being used - globus_gram_client. To find the available flavors use the command

```
$GPT_LOCATION/sbin/gpt-query globus_gram_client
```

You should get the below output

```
2 packages were found in /opt/globus that matched your query:

packages found that matched your query
  globus_gram_client-gcc32dbg-dev pkg version: 4.1.1
  globus_gram_client-gcc32dbg-rtl pkg version: 4.1.1
```

In this listing the only available flavor is "gcc32dbg"

3. Having now chosen a suitable flavor a file can be created with all the necessary definitions by using the command:

```
globus-makefile-header --flavor gcc32dbg globus_gram_client >
globus_header
```

If you examine the file "globus_header" with the command

```
cat globus_header
```

the advantage of having an automated system for generating this configuration is apparent.

4. Having generated the necessary configuration it can be included in a Makefile and the code compiled. Examine the "Makefile" and you will that "globus_header" is included and the commands that are used to compile your code. Compile the command by typing

```
make
```

and then run it with the command

```
./apidemo
```

. This code simply tests that the queue "grid-data.rl.ac.uk/jobmanager-pbs" can be contacted (a ping type message is sent).

5. If you examine the code you will see that it is necessary to initialize the part of the globus api code being used:

```
globus_module_activate(GLOBUS_GRAM_CLIENT_MODULE);
```

and similarly to deactivate it. If you miss this out of your code it will still compile, but you will encounter errors at run time. ***Modify the code and introduce an error into the name of the queue being tested.*** If make complains about being already up to date use the command

```
make clean
```

before repeating the make command.

[Forward to the Java API](#) ►

Java API (The CoGKit)

The second part of the API section focuses on using the Java API for Globus - known as the CoG (Community Grid) Kit. The tutorial uses version 1.2 of the CoG Kit (the next release - version 4 has just recently been released).

Note for anyone unfamiliar with java:

This section does not require a knowledge of Java. The focus is instead on how to use an API to interact with the underlying Globus architecture. A brief introduction to java can be found at <http://forge.nesc.ac.uk/docman/view.php/44/116/leena2.doc>.

Please also note: the phrase "after the line(s)" means after the line described and before the next (non-comment) line.

1. Several files have already been created for you, so firstly change directory:

```
cd ~/gram2/java_api
```

This folder contains a folder called "cogExample". This folder contains all the java files needed for this tutorial including "Utility.java" (the source for the java class "cogExample.Utility") that provides a series of helper functions that are not connected to the CoGKit usage and hence need not be looked at in this tutorial. Also in this folder is "build.xml", which is used by ant (a commonly used java tool) to automate the process of building each stage of the tutorial.

2. It is necessary to set the CLASSPATH to point to all of the relevant CoGKit jar files (java libraries). If you are not familiar with java, the CLASSPATH contains a list of the locations of all the jar files and classes your program depends on. Load the appropriate modules with the command

```
module load ant cog
```

Unfortunately the CoGKit does not read all of the Globus environmental variables correctly, and instead relies on its own configuration file to specify the same information. Normally this configuration file has to be created by the user, however a script has been installed on the UI that will generate an appropriate configuration file for you. This may be run with the command

```
mk.cog.properties
```

3. The starting point for this section of the tutorial is to repeat in Java the same ping of a globus queue that was demonstrated in the previous section. This starting code has already been written for you and exists in the file "cogExample/GatekeeperPing.java". This code (and all subsequent java code) is designed so that the queue to be used is passed as a command line parameter. Compile stage1 with the command:

```
ant stage1
```

If all goes well you should get the below output:

```
[user00@pub-234 java_api]$ ant stagel
Buildfile: build.xml

stagel:
    [javac] Compiling 2 source files

BUILD SUCCESSFUL
Total time: 4 seconds
```

You can now run the code by typing:

```
java cogExample.GatekeeperPing grid-data.rl.ac.uk/jobmanager-pbs -q
RXXXXXX
```

Because the queue used during the tutorials is only available during the tutorial, the below example output and all subsequent example outputs are using a different queue. If this program successfully ran then similar output to the below should be obtained:

```
[user00@pub-234 java_api]$ java cogExample.GatekeeperPing grid-data.rl.ac.uk/
jobmanager-pbs -q cpul
Testing contact with grid-data.rl.ac.uk/jobmanager-pbs -q cpul
Contact successfully tested
```

4. Having got a simple program to run the next step is to expand the program to submit and manage a simple job. This step involves three new files:
- cogExample/JobSubmission.java: The class that does all the job submission and management.
 - cogExample/SubmitSingleJob.java: This just defines the executable (and its parameters) the job will run and then calls the job submission function from the JobSubmission class.
 - cogExample/GJListen.java: A class that 'listens' to the status of a job.

Throughout this tutorial new files are introduced. All of these files are commented and if you are familiar with Java then you are encouraged to examine each file.

Before going into the details of the code, the RSL (Resource Specification Language) job definition language is introduced. An RSL string contains all the information needed for a job to run. In the previous tutorial when the option '-x' was used this was just forcing extra information to be included in the RSL string. Both globus-job-run and globus-job-submit are actually wrappers that produce the relevant RSL and pass this to the program globusrun. Run the command

```
globus-job-run -dumprsl grid-data.rl.ac.uk/jobmanager-pbs -q RXXXXXX /
bin/hostname -f
```

By adding the '-dumprsl' option globus-job-run (and similarly globus-job-submit) merely displays the RSL and not does not run any job. You should get the following string:

```
&(executable="/bin/hostname")
(queue="cpul")
(arguments= "-f")
```

Note that the format of an RSL string has changed with Globus Toolkit version 3 and above to use XML.

5. Examine the file cogExample/JobSubmission.java by using the text editor:

```
kwrite cogExample/JobSubmission.java &
```

The comments in this file explain how the example works. Before continuing read through these comments. This example will:

1. submit this RSL string as an interactive job
2. attach a process to listen for job status changes
3. run the job and wait for it to finish

Interactive in this context means that the code maintains a connection to the running job (comparable to globus-job-run). The opposite to interactive is batch (comparable to globus-job-submit). Compile the code

by typing:

```
ant stage2
```

Once the code has successfully been compiled run the example:

```
java cogExample.SubmitSingleJob grid-data.rl.ac.uk/jobmanager-pbs -q  
RXXXXX
```

You should get output similar to

```
[user00@pub-234 java_api]$ java cogExample.SubmitSingleJob grid-data.rl.ac.uk/  
jobmanager-pbs -q cpul  
Testing contact with grid-data.rl.ac.uk/jobmanager-pbs -q cpul  
Contact successfully tested  
Job Details:  
contact = grid-data.rl.ac.uk/jobmanager-pbs -q cpul  
command = /bin/hostname -f  
RSL = &(executable=/bin/hostname)  
      (arguments=-f)  
Status: https://grid-data.rl.ac.uk:64016/24212/1127081664/ = PENDING  
Status: https://grid-data.rl.ac.uk:64016/24212/1127081664/ = ACTIVE
```

You should quickly notice that something is not right. The job appears to be running and no errors have been received but the job is failing to reach the "DONE" state. There are two options available now: *either wait for the job to timeout (this can take a while) or type <CTRL> - c to abort the program.*

6. The reason the job failed was that no process for handling the standard output and error of the job was implemented. In the same way as in the previous tutorial files were staged onto the head node it is necessary for the node running a job to be able to stage (via the head node) files back to the local machine. This is done by running locally a GASS (Global Access to Secondary Storage) server which uses the HTTP protocol to transfer the files. Whilst it is possible to run one GASS server for multiple jobs it then becomes difficult to associate each job with its outputs. A better solution is to run one instance of the GASS server for each job.

Modify the code to start a GASS server before submitting the job. You should already have cogExample/JobSubmission.java open in the text editor. All the necessary changes occur in the submit function. There are four changes that need to be made.

1. It is necessary to include the relevant classes into the program. Immediately after the line

```
import org.globus.gram.GramJob;
```

add the line

```
import org.globus.io.gass.server.GassServer;
```

7. Start the GASS server and obtain the URL it is running at. After the lines

```
public void submit(String contact, String executable, String params) {  
    try  
    {
```

add the lines

```
GassServer gass = new GassServer();  
String gassUrl = gass.getURL();
```

8. The next stage is to include in the RSL information about the GASS Server so that the site running the job can know where to stage the standard output/error to. After the lines

```

if (params != null)
{
    rsl += "(arguments="+params+")";
}

```

insert the line

```

rsl += "(stdout=" + gassUrl + "/dev/stdout)" + "(stderr=" + gassUrl + "/dev/stderr)";

```

9. Finally, the utility function 'display' can also display the GASS URI. Note that the `_id` parameter is important for the next section of this tutorial. Modify the line

```

Utility.display(_id,contact,null,executable + " " + params,rsl);

```

to read

```

Utility.display(_id,contact,gassUrl,executable + " " + params,rsl);

```

- Compile the code with the command

```

ant stage3

```

and then **run the program** in the same way as previously. You should get output similar to:

```

Testing contact with grid-data.rl.ac.uk/jobmanager-pbs -q cpul
Contact successfully tested
Job Details:
    contact = grid-data.rl.ac.uk/jobmanager-pbs -q cpul
    GASS Server = https://129.215.30.234:20000
    command = /bin/hostname -f
    RSL = &(executable=/bin/hostname)
        (arguments=-f)
        (stdout=https://129.215.30.234:20000/dev/stdout)
        (stderr=https://129.215.30.234:20000/dev/stderr)
Status: https://grid-data.rl.ac.uk:64028/5965/1127663915/ = PENDING
Status: https://grid-data.rl.ac.uk:64028/5965/1127663915/ = ACTIVE
grid-data04.rl.ac.uk
Status: https://grid-data.rl.ac.uk:64028/5965/1127663915/ = DONE

```

- The code in the previous stage made no attempt to handle the standard output and error of the job, they were merely redirected to the standard output and error of the java program. The CoG Kit however includes functionality that makes it possible to handle any data staged back to the GASS Server. This is done by introducing a class to handle this data, which in this example this class is created in 'cogExample/JOListen.java' that has been already created for you. The changes that need to be made this time to 'cogExample/JobSubmission.java' are as follows:

1. As in the previous stage the relevant class must be imported. After the line

```

import org.globus.io.gass.server.GassServer;

```

add the line

```

import org.globus.io.gass.server.JobOutputStream;

```

- The second stage is to attach a copy of the JOListen class to the GASS server. Note that it is important that the first parameter to the `registerJobOutputStream` corresponds to the last part of the devices named in the RSL string. For simplicity the jobs standard output and error have been merged and are handled by one instance of the class. After the line

```

Utility.display(_id,contact,gassUrl,executable + " " + params,rsl);

```

add the following lines

```
JOListen joListen = new JOListen();
JobOutputStream outStream = new JobOutputStream (joListen);
gass.registerJobOutputStream("out", outStream);
gass.registerJobOutputStream("err", outStream);
```

- The final step is to display the standard output/error received. A simple utility function has been created for you to display this output (the need for this function will become more apparent in the next section of the tutorial. After the line

```
while( gjListen.running ) { Thread.sleep(1000); }
```

add the lines

```
System.out.println(_id + "The following output/error was received");
Utility.printOutput(_id,joListen.output);
```

- Compile the code with the command

```
ant stage4
```

and then *run the program*, in the same way as previously. You should get the output

```
Testing contact with grid-data.rl.ac.uk/jobmanager-pbs -q cpu1
Contact successfully tested
Job Details:
  contact = grid-data.rl.ac.uk/jobmanager-pbs -q cpu1
  GASS Server = https://129.215.30.234:20000
  command = /bin/hostname -f
  RSL = &(executable=/bin/hostname)
        (arguments=-f)
        (stdout=https://129.215.30.234:20000/dev/stdout)
        (stderr=https://129.215.30.234:20000/dev/stderr)
Status: https://grid-data.rl.ac.uk:64026/28699/1127738894/ = ACTIVE
Status: https://grid-data.rl.ac.uk:64026/28699/1127738894/ = PENDING
Status: https://grid-data.rl.ac.uk:64026/28699/1127738894/ = DONE
The following output/error was received
grid-data06.rl.ac.uk
```

[◀ Back to the C API](#)

[Forward to Parallel Jobs ▶](#)





GRAM Tutorial - Part II

NGS Induction Event,
Guy Warner, NeSC Training
Team



Parallel Non-Communicating Jobs

The simplest form of project to take advantage of the ability to run parallel non-communicating jobs is where the problem to be solved is 'trivially parallel'. Trivially Parallel is when multiple copies of the same job can be run with different input parameters or data. The important characteristic is that once the jobs have started running they don't need to know what the other jobs are doing. When they have finished running, a final job may be run to collate the results. This sort of problem does not require any complicated message passing to a master program (which will be looked at later). The simplest way to store the results is to use a file (or database) which may be analysed later. For this tutorial we will look at a (somewhat contrived) simple mathematical problem. The problem to be solved is to integrate, using the trapezium rule (for details on the trapezium rule see <http://www.answers.com/topic/trapezium-rule>) the function

$$f(x) = \begin{cases} 1 - x^2 & -1 < x \leq 1/3 \\ -x^2 & 1/3 < x < 1 \end{cases}$$

If you are not familiar with mathematics it is probably sufficient to know that the exact answer is 2/3. The trapezium rule solves this problem numerically and hence a numerical error is expected - this example demonstrates how to reduce this error by using multiple jobs simultaneously.

The jobs that are run in this example use two perl scripts as their executables. These scripts are in a subdirectory of the java_api folder called "parallel". The two scripts are:

- integrate.pl - the script that carries out the numerical integration. This script accepts the parameters (in this order): starting point, end point and the name of the file to store the output.
- sum.pl - the script that collates the results. It accepts the parameters (in this order): base_name - the base name of all the output files (in this example this is always "multijobData") and segment_total - the number of segments the problem was split into.

The java code from the previous section is extended to support threading. This is done by using "cogExample/SubmitMultipleJobs.java" as a replacement for "cogExample/SubmitSingleJob.java". Due to the numerical nature of the problem it is possible to split the problem into several segments ([-1.0, -0.6], [-0.6, -0.2], [-0.2, 0.2], [0.2, 0.6] and [0.6, 1.0]). This new java class firstly controls the creation of the threads and passes the executable name (in this case integrate.pl) and appropriate segment information in the parameter string to the thread. The parameter "_id" is now used to distinguish the output from each thread. Once all of the threads have finished running a final job is submitted to run the "sum.pl" executable.

Note for the non-programmer: Threading is the capability of handling multiple flows of control within a single application or process. Care must be taken when accessing variables shared between multiple threads.

1. The above described perl files can be found in the subdirectory "parallel". Change to this directory with the command

```
cd parallel
```


and **upload the perl files to the head node using gsiscp**. Then change back to the java_api directory with the command

```
cd ..
```

2. Next compile the code with the command

```
ant stage5
```

3. If the compile was successful run your program by typing

```
java cogExample.SubmitMultipleJobs grid-data.rl.ac.uk/  
jobmanager-pbs -q RXXXXX
```

You should get similar output to the below (which has been truncated to avoid needless repetition):

```
Testing contact with grid-data.rl.ac.uk/jobmanager-pbs -q cpul  
Contact successfully tested  
Job 2: Job Details:  
Job 2: contact = grid-data.rl.ac.uk/jobmanager-pbs -q cpul  
Job 2: GASS Server = https://129.215.30.234:20000  
Job 2: command = integrate.pl -0.19999999 0.20000005 multijob.2  
Job 2: RSL = &(executable=integrate.pl)  
Job 2: (arguments=-0.19999999 0.20000005 multijob.2)  
Job 2: (stdout=https://129.215.30.234:20000/dev/stdout)  
Job 2: (stderr=https://129.215.30.234:20000/dev/stderr)  
  
...  
  
Job 0: Status: https://grid-data.rl.ac.  
uk:64036/10773/1127744975/                = PENDING  
Job 0: Status: https://grid-data.rl.ac.  
uk:64036/10773/1127744975/                = ACTIVE  
Job 1: Status: https://grid-data.rl.ac.  
uk:64044/10774/1127744975/                = PENDING  
Job 1: Status: https://grid-data.rl.ac.  
uk:64044/10774/1127744975/                = ACTIVE  
  
...  
  
Job 1: The following output/error was received  
Job 1: grid-data04.rl.ac.uk: Running with args -0.6 -  
0.19999999                                multijob.1  
  
...  
  
Job 5: Job Details:  
Job 5: contact = grid-data.rl.ac.uk/jobmanager-pbs -q cpul  
Job 5: GASS Server = https://129.215.30.234:20006  
Job 5: command = sum.pl multijob 5  
Job 5: RSL = &(executable=sum.pl)  
Job 5: (arguments=multijob 5)  
Job 5: (stdout=https://129.215.30.234:20006/dev/stdout)  
Job 5: (stderr=https://129.215.30.234:20006/dev/stderr)  
Job 5: Status: https://grid-data.rl.ac.  
uk:64023/11166/1127745009/                = PENDING  
Job 5: Status: https://grid-data.rl.ac.  
uk:64023/11166/1127745009/                = ACTIVE  
Job 5: Status: https://grid-data.rl.ac.  
uk:64023/11166/1127745009/                = DONE
```

```
Job 5: The following output/error was received
Job 5: 0.6728000325
```

Parallel Communicating Jobs

For the final part of this tutorial an example of running a parallel communicating job (often referred to as an MPI job) is introduced. Projects that are not trivially parallel and where run time communication between the parallel components is required typically make use of MPI. The Message Passing Interface (MPI) library provides the tools for passing these messages between parallel components running on different nodes. The implementation of MPI being used in this tutorial is "mpich-gm". Full details on this implementation may be found at <http://www.myri.com/scs/index.html> (note that an understanding of the implementation details is not required for this tutorial).

1. Before starting this example change to the directory in your accounts that contains the MPI example code with the command

```
cd ~/gram2/mpi
```

and then *upload (using gsiscp) the file "mpi_ex.c" to grid-data.rl.ac.uk*

2. As with the fortran example in the previous tutorial, to compile code for MPI requires access to libraries that are not available to you locally. Open a new gsissh connection onto grid-data.rl.ac.uk. You may find it easiest to work with two Putty connections to training-ui.nesc.ed.ac.uk , one for running commands on training-ui.nesc.ed.ac.uk and one for running commands on grid-data.rl.ac.uk. **Load the module "clusteruser"**. This is actually a module that acts a wrapper to the collection of modules needed to run MPI jobs. You can see the list of modules that have been loaded by using one or both of the following commands:

```
module list
module display clusteruser
```

3. Now compile the example MPI job with the command

```
mpicc -w mpi_ex.c -o mpi_ex
```

If you try to run the program you have just built you will encounter error messages. For the sake of simplicity in this tutorial we will not look at how to run the program from this environment, but concentrate instead on how to run it as a job using the globus commands.

4. From training-ui.nesc.ed.ac.ukrun the command

```
globus-job-submit grid-data.rl.ac.uk/jobmanager-pbs -q RXXXXX -
np 4 -x '&(jobtype=mpi)(environment=(NGSMODULES clusteruser))'
mpi_ex
```

There are two important changes to observe with this command. Firstly, this time 4 processes, which may or may not run on the same node, have been requested for the job to run on (the -np 4 part of the command). Secondly the jobtype has been set to MPI Once this job has successfully run you should be able to retrieve something similar to the following output

```
Process 0 on host grid-data12.rl.ac.uk broadcasting to all processes
Receiving from all other processes

Received a message from process 2 on grid-data14.rl.ac.uk
Received a message from process 1 on grid-data13.rl.ac.uk

Received a message from process 3 on grid-data15.rl.ac.uk

Ready
```

5. Optional (requiring a knowledge of C): modify the code so that each of the processes other than process 0 send back an additional piece of information (for example a random number). Several other example MPI examples have been provided in a subfolder of your current folder called "extra_examples". You might find comparison with these codes helpful in solving this task. Also try running these programs as jobs (the fortran code will need to use the mpif77 compiler).

◀ [Back to the Java API](#)

