gLite Advanced Job Management

Porting Application School

Cesar Fernández (cesar.fernandez@usm.cl)

***Valparaíso, 30 November* 2010**

Outline

Special Jobs

- DAG
- Collection
- Parametric
- MPI
- OpenMP
- Practice

Introduction to job's types

The WMS currently supports the following types for Jobs (this does not apply to DAGs and Collections):

Normal a simple batch job

Interactive a job whose standard streams are forwarded to the submitting client

MPICH a parallel application using MPICH-P4 implementation of MPI

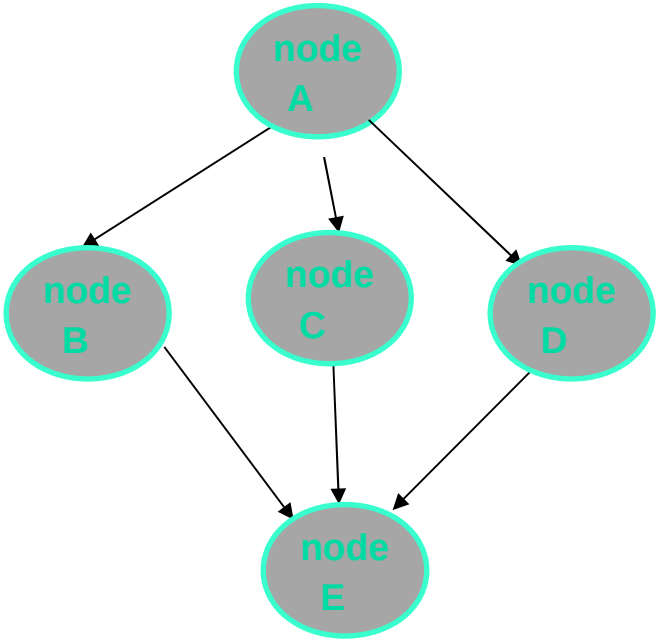
Partitionable a job that can be thought as composed by a set of independent steps/iterations, i.e. a set of independent sub-jobs, each one taking care of a step or of a sub-set of steps, and which can be executed in parallel

Checkpointable a job able to save its state, so that the job execution can be suspended, and resumed later, starting from the same point where it was first stopped

Parametric a job whose JDL contains parametric attributes (e.g. Arguments, StdInput etc.) whose values can be made vary in order to obtain submission of several instances of similar jobs only differing for the value of the parameterized attributes

DAG jobs

- A DAG job is a set of jobs where input, output, or execution of one or more jobs can depend on other jobs
- Dependencies are represented through Directed Acyclic Graphs, where the nodes are jobs, and the edges identify the dependencies



Attribute: InputSandbox

- All nodes that do not contain the InputSandbox attribute in their descriptions inherit the value of these attributes from the one specified for the DAG.

- Nodes representing jobs without InputSandbox have to contain the following specification in their description (empy InputSandbox list):

```
[
        ..................
        InputSandbox = {};
        ..................
]
```

Attribute: Nodes

The *Nodes* attribute is the core of the DAG description;

```
....
Nodes = [  nodefilename1 = [...]
           nodefilename2 = [...]
           .......

           dependencies = ...

]
```

```
Nodefilename1 = [ file = "foo.jdl"; ]
Nodefilename2 =
  [   file = "/home/vardizzo/test.jdl";
      retry = 2;          ]
```

```
Nodefilename1 = [
    description = [ JobType = "Normal";
                    Executable = "abc.exe";
                    Arguments = "1 2 3";
                    OutputSandbox = [...];
                    InputSandbox = [...];
                    ..... ]
    retry = 2;
                ]
```

Attribute: File

```
Nodefilename1 = [ file = "foo.jdl"; ]
Nodefilename2 =
   [   file = "/home/vardizzo/test.jdl";
       retry = 2;          ]
```

- The File attribute is a string representing the path on the local file system ofù a file containing the JDL description of a Job. It is important to note that this kind of representation can only be used when submitting to the WMS through a client (**glite-wms-job-submit**) able to resolve the path locally and to expand the JDL with the full description before passing it to the WMS.

- The File attribute cannot be specified together with the Description attribute within the same node description!

Attribute: Dependencies

- It is a list of lists representing the dependencies between the nodes of the DAG.

```
....
Nodes = [  nodefilename1 = [...]
           nodefilename2 = [...]
           .......

           dependencies = ...

]
```

dependencies =
{nodefilename1, nodefilename2}

MANDATORY : YES!

dependencies = {};

{ nodefilename1, nodefilename2 }

{ { nodefilename1, nodefilename2 }, nodefilename3 }

{ { { nodefilename1, nodefilename2}, nodefilename3}, nodefilename4 }

DAG jdl

```
[
  type = "dag";
  max_nodes_running = 4;
  nodes = [
    nodeA = [
      file ="nodes/nodeA.jdl" ;
    ];
    nodeB = [
      file ="nodes/nodeB.jdl" ;
    ];
    nodeC = [
      file ="nodes/nodeC.jdl" ;
    ];
    nodeD = [
      file ="nodes/nodeD.jdl";
    ];
   dependencies = {
      {nodeA, nodeB},
      {nodeA, nodeC},
      { {nodeB,nodeC}, nodeD }
    }
  ];
]
```

Job Collection

- A job collection is a set of independent jobs that user wants to submit and monitor as a single request
- Jobs of a collection are submitted as DAG nodes without dependencies

```
[
    Type = "collection";
    VirtualOrganisation = “gilda";
    nodes = {
        [  <job descr 1 >],
        [  <job descr 2 >],
          …
        };
]
```

Input Sandboxes

- Input Sandbox can contain:
  - pointer to other files within the DAG/collection
  - URI pointing to files on a remote gridFTP/HTTPS server
  - file paths on the UI machine (i.e. the usual way)

```
InputSandbox = {
    "gsiftp://neo.datamat.it:2811/var/prg/sim.sh",
     root.nodes.nodeA.description.OutputSandbox[0],
    "file:///home/pacio/myconf“ };
```

Only local files (file://) are uploaded to the WMS node

- **File pointed by URIs are directly downloaded on the WN by the JobWrapper just before the job is started**

Output Sandboxes

- **The OutputSandbox attribute lists the files destination of the job output**

```
OutputSandbox = {      "jobOutput","run1/event1",
                       "jobError"                    };
OutputSandboxDestURI = {
    "gsiftp://matrix.datamat.it/var/jobOutput",
    "https://grid003.ct.infn.it:8443/home/cms/event1",
    "gsiftp://matrix.datamat.it/var/jobError"        };
```

A base URI to be applied to all sandbox files can also be specified

```
OutputSandboxBaseDestURI = "gsiftp://neo.datamat.it/home/run1/";
```

- **Files are copied when the job has completed execution by the JobWrapper to the specified destination without transiting on the WMS node**

Parametric Job

- **A parametric job is a job where one or more of its attributes are parameterized**

- **Values of attributes vary according to a parameter**

```
[
  JobType = "Parametric";
  Executable = "/bin/sh";
  Arguments = "md5.sh input_PARAM_.txt";
  InputSandbox = {"md5.sh", "input_PARAM_.txt"};
  StdOutput = "out_PARAM_.txt";
  StdError = "err_PARAM_.txt";
  Parameters = 4;
  ParameterStart = 1;
  ParameterStep = 1;
  OutputSandbox = {"out_PARAM_.txt",
"err_PARAM_.txt"};
]
```

Job monitoring / managing is always done through an unique jobID, as if the job was single (see submission of collection)

Parametric job

- **Parameter can be either a number, or a list of items (typically strings, but not enclosed within double quotes)**

- **Input Sandbox (if present) has to be coherent with parameters**

```
> cat param2.jdl
[
      JobType = "Parametric";
      Executable = “/bin/cat";
            Arguments = “input_PARAM_.txt”;
      InputSandbox = "input_PARAM_.txt";
      StdOutput = "myoutput_PARAM_.txt";
      StdError = "myerror_PARAM_.txt";
      Parameters = {EARTH,MOON,MARS};
      OutputSandbox  = {“myoutput_PARAM_.txt”};
]

> ls
inputEARTH.txt  inputMARS.txt  inputMOON.txt
param2.jdl
```

It is the list of the values the parameter must take.

Parametric job

```
[
JobType = "Parametric";
Executable = "myjob.exe";
StdInput = "input_PARAM_.txt";
StdOutput = "output_PARAM_.txt";
StdError = "error_PARAM_.txt";
Parameters = 100;
ParameterStart = 1;
ParameterStep = 1;
InputSandbox = {"myjob.exe", "input_PARAM_.txt";
OutputSandbox = {"output_PARAM_.txt",
"error_PARAM_.txt"};
]
```

The parameter is a number

the initial number of the running paramenter

the increment of the running parameter between consecutive jobs.

Both attributes, ParameterStart and ParameterStep, can be set only if Parameters is a number.

Parametric job

A parametric structure generates N jobs as follow:
N = (Parameters – ParameterStart)/ParameterStep

Each one containing the same JDL but the parametric attributes for which the “_PARAM_” instruction is replaced with the actual value of the parameter.

Also, the parameters can be represented as follow:

```
Arguments = "_PARAM_";
Parameters = {alpha, beta, gamma};
```

It can be used as a set of numbers:

```
Arguments = “test.sh _PARAM_";
Parameters = 1000;
ParameterStart = 1;
ParameterStep=1;
```

Parametric job

Important Issue:

Be carefull with the command glite-wms-job-output, since to recover all data outputs, it requires that the N jobs had finished. The command only recovers the data outputs of the set of jobs finished.

MPI Overview

- **Execution of parallel jobs is an essential issue for modern informatics and applications.**

- **Most used library for parallel jobs support is MPI (Message Passing Interface)**

- **At the state of the art, parallel jobs can run inside single Computing Elements (CE) only;**

- **Several projects are involved into studies concerning the possibility of executing parallel jobs on Worker Nodes (WNs) belonging to different CEs.**

- **The source code must have been compiled with mpicc libraries**

Requirements & settings

- **From the user's point of view, jobs to be run as MPI are specified setting the JDL JobType attribute to MPICH and specifying the NodeNumber attribute as well.**

E.g.:

…

JobType = "MPICH";

NodeNumber

This attribute defines the required number of CPUs needed for the application

Requirements & settings

- **When the previous two attributes are included in a JDL, the User Interface (UI) *automatically* adds the following expression:**

```
(other.GlueCEInfoTotalCPUs >= NodeNumber) &&
Member (“MPICH”,other.GlueHostApplicationSoftwareRunTimeEnvironment)
```

JDL Requirements expression in order to find out the best resource where the job can be executed.

MPI JDL

```
> $ cat MPItest.jdl
[
Type = "Job";
JobType = "MPICH";
NodeNumber = 4;
Executable = "MPItest.sh";
Arguments = "MPItest";
StdOutput = "MPItest.out";
StdError = "MPItest.err";
InputSandbox = {"MPItest.sh","MPItest.c"};
OutputSandbox =
{"MPItest.err","MPItest.out","mpiexec.out"};
Requirements = (other.GlueCEInfoLRMSType ==
"PBS") || (other.GlueCEInfoLRMSType == "LSF");
]
```

MPI C code

Simple c program for mpi job:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int numprocs;  /* Number of processors */
  int procnum;   /* Processor number */
  /* Initialize MPI */
  MPI_Init(&argc, &argv);
  /* Find this processor number */
  MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
  /* Find the number of processors */
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  printf ("Hello world! from processor %d out of %d\n", procnum, numprocs);
  /* Shut down MPI */
  MPI_Finalize();
  return 0;
}
```

File sh associate with mpi job:

```
#!/bin/sh -x
# the binary to execute
EXE=$1

echo "****************************************************************"
echo "Running on: $HOSTNAME"
echo "As:       " `whoami`
echo "****************************************************************"

echo "****************************************************************"
echo "Compiling binary: $EXE"
echo mpicc -o ${EXE} ${EXE}.c
mpicc -o ${EXE} ${EXE}.c
echo "************************************"

if [ "x$PBS_NODEFILE" != "x" ] ; then
  echo "PBS Nodefile: $PBS_NODEFILE"
  HOST_NODEFILE=$PBS_NODEFILE
fi
```

[…]

```
if [ "x$LSB_HOSTS" != "x" ] ; then
  echo "LSF Hosts: $LSB_HOSTS"
  HOST_NODEFILE=`pwd`/lsf_nodefile.$$
  for host in ${LSB_HOSTS}
  do
    echo $host >> ${HOST_NODEFILE}
  done
fi

if [ "x$HOST_NODEFILE" = "x" ]; then
  echo "No hosts file defined.  Exiting..."
  exit
fi

echo "***************************************************************"
CPU_NEEDED=`cat $HOST_NODEFILE | wc -l`
echo "Node count: $CPU_NEEDED"
echo "Nodes in $HOST_NODEFILE: "
cat $HOST_NODEFILE
echo "***************************************************************"
echo "***************************************************************"
```

[…]

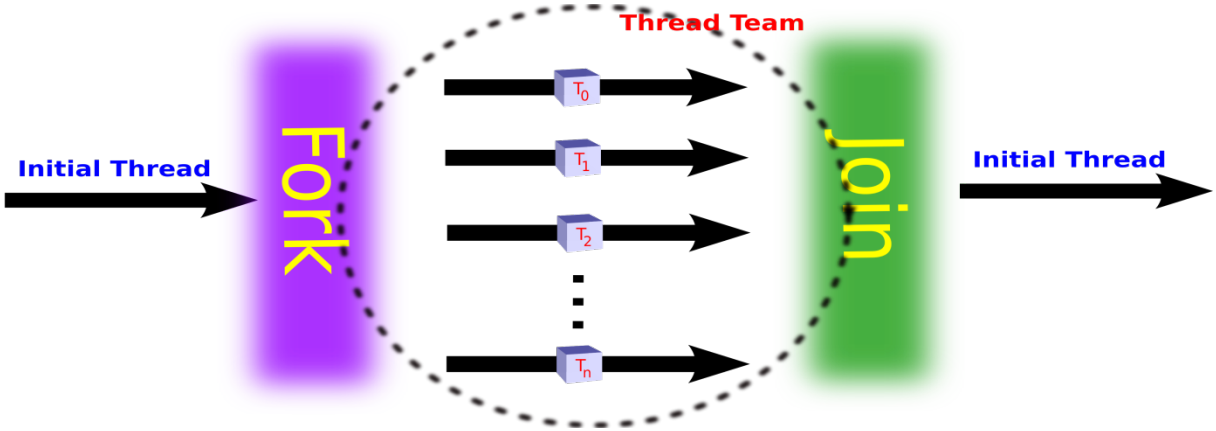
```
CPU_NEEDED=`cat $HOST_NODEFILE | wc -l`
echo "Checking ssh for each node:"
NODES=`cat $HOST_NODEFILE`
for host in ${NODES}
do
  echo "Checking $host..."
  ssh $host hostname
done
echo "****************************************************************"

echo "****************************************************************"
echo "Executing $EXE with mpiexec"
chmod 755 $EXE
mpiexec `pwd`/$EXE > mpiexec.out 2>&1
echo "****************************************************************"

echo "****************************************************************"
echo "Executing $EXE with mpirun"
chmod 755 $EXE
mpirun -np $CPU_NEEDED -machinefile $HOST_NODEFILE `pwd`/$EXE
echo "****************************************************************"
```

OpenMP

- The OpenMP Application Program Interface (API) supports multi-platform shared memory parallel programming in C/C++ and Fortran on all architectures.
- OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.
- Supported by ICC and GCC (version >=4.2)



Multicore job example

```
>cat multicore.jdl
[
Type="Job";
JobType="Normal";
Executable = "multicore.sh";
StdError = "multi.e";
StdOutput = "multi.o";
InputSandbox = {"multicore.sh","multicore.cpp"};
OutputSandbox = {"multi.e", "multi.o"};
Requirements = other.GlueCEInfoLRMSType == "PBS" &&
        other.GlueCEInfoTotalCPUs >=4;
]
```

Multicore job example

```
#include <iostream>
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]) {
        cout<<"\n Ejemplo multicore"<<endl;
        omp_set_num_threads(4);
        int i;
        double inicio1,inicio2;
        inicio1=omp_get_wtime();
        #pragma omp parallel for
        for(i=0;i<4;i++){
                cout<<"\n Hola soy el Thread: "<<omp_get_thread_num()<<endl;
        }
        inicio2=omp_get_wtime();
        cout<<"\n tiempo del loop: "<<inicio2-inicio1<<endl;
        return 0;
}
```

Multicore job example

```
>cat multicore.sh
#!/bin/sh -x
g++ multicore.cpp -o multicore -fopenmp
chmod 777 multicore
./multicore

>glite-wms-job-submit –o salida.txt –a multicore.jdl
>glite-wms-job-status –i salida.txt
>glite-vms-job-output –i salida.txt –dir .
>more job_id/*
```

Practical on WMS

Run an *ls* command on a resource

```
>cat ls_la.jdl
[
 Type = "Job";
  JobType = "Normal";
  Executable = "command.sh";
 Arguments = "-la";
  StdError = "stderr.e";
  StdOutput = "stdout.o";
  InputSandbox = "simple.sh";
  OutputSandbox = {"stderr.e", "stdout.o"};
]
>cat command.sh
#!/bin/sh
/bin/ls $1
```

Submit C/C++ programs

```
>cat hello.cpp
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
        cout<<"\hola mundo”<<endl;
  Return 0;
}
>cat hello.jdl
[
  Executable = "/bin/sh";
  Arguments = "hello.sh";
  StdError = "stderr.err";
  StdOutput = "stdout.out";
  InputSandbox = {"hello.sh","hello.cpp"};
  OutputSandbox = {"stderr.err", "stdout.out"};
]
```

Submit C/C++ programs

```
>cat hello.sh
#!/bin/sh -x
g++ lector.cpp -o lector
chmod 777 lector
./lector
```

Submit

```
glite-wms-job-submit -o jobid.txt -a hello.jdl
```

DAG Sample

```
[
Type = "dag";
max_nodes_running = 5;
nodes = [
        nodeA = [
        description = [
        JobType = "normal";
        Executable = "/bin/date";
        StdOutput = "t5a.out";
        StdError = "t5a.err";
        OutputSandbox = {"t5a.out","t5a.err"}; ];
];
nodeB = [
description = [
        JobType = "normal";
        Executable = "/bin/ls";
        Arguments = “-la”
        StdOutput = "t5b.out";
        StdError = "t5b.err";
        OutputSandbox = {"t5b.out","t5b.err"}; ];
];
dependencies = { { nodeA, nodeB } }; ];
];
```

References

gLite Workload Management System

http://glite.web.cern.ch/glite/packages/R3.0/deployment/glite-WMS/glite-WMS.asp

The Message Passing Interface (MPI) standard

http://www.mcs.anl.gov/research/projects/mpi/

The OpenMP API specification for parallel programming

http://openmp.org