Chapter 1: Developing Codes for the Grid

1 Best Practices

This document lists out issues for the algorithm developers to keep in mind while developing the respective codes. Keeping these in mind will alleviate a lot of problems while trying to run the codes on the Grid through workflows.

1.1 Supported Platforms

Most of the hosts making a Grid run variants of Linux or in some case Solaris. The Grid middleware mostly supports UNIX and it's variants.

1.1.1 Running on Windows

The majority of the machines making up the various Grid sites run Linux. In fact, there is no widespread deployment of a Windows-based Grid. Currently, the server side software of Globus does not run on Windows. Only the client tools can run on Windows. The algorithm developers should not code exclusively for the Windows platforms. They must make sure that their codes run on Linux or Solaris platforms. If the code is written in a portable language like Java, then porting should not be an issue.

If for some reason the code can only be executed on windows platform, please contact the pegasus team at pegasus aT isi dot edu . In certain cases it is possible to stand up a linux headnode in front of a windows cluster running Condor as it's scheduler.

1.2 Packaging of Software

As far as possible, binary packages (preferably statically linked) of the codes should be provided. If for some reason the codes, need to be built from the source then they should have an associated makefile (for C/C++ based tools) or an ant file (for Java tools). The building process should refer to the standard libraries that are part of a normal Linux installation. If the codes require non-standard libraries, clear documentation needs to be provided, as to how to install those libraries, and make the build process refer to those libraries.

Further, installing software as root is not a possibility. Hence, all the external libraries that need to be installed can only be installed as non-root in non-standard locations.

1.3 MPI Codes

If any of the algorithm codes are MPI based, they should contact the Grid group. MPI can be run on the Grid but the codes need to be compiled against the installed MPI libraries on the various Grid sites. The pegasus group has some experience running MPI code through PBS.

1.4 Maximum Running Time of Codes

Each of the Grid sites has a policy on the maximum time for which they will allow a job to run. The algorithms catalog should have the maximum time (in minutes) that the job can run for. This information is passed to the Grid sites while submitting a job, so that Grid site does not kill a job before that published time expires. (It's OK if the job runs only a fraction of the max time).

1.5 Codes cannot specify the directory in which they should be run

Codes are installed in some standard location on the Grid Sites or staged on demand. However, they are not invoked from directories where they are installed. The codes should be able to be invoked from any directory, as long as one can access the directory where the codes are installed.

This is especially relevant, while writing scripts around the algorithm codes. At that point specifying the

relative paths do not work. This is because the relative path is constructed from the directory where the script is being invoked. A suggested workaround is to pick up the base directory where the software is installed from the environment or by using the dirname cmd or api. The workflow system can set appropriate environment variables while launching jobs on the Grid.

1.6 No hard-coded paths

The algorithms should not hard-code any directory paths in the code. All directories paths should be picked up explicitly either from the environment (specifying environment variables) or from command line options passed to the algorithm code.

1.7 Wrapping legacy codes with a shell wrapper

When wrapping a legacy code in a script (or another program), it is necessary that the wrapper knows where the executable lives. This is accomplished using an environmental variable. Be sure to include this detail in the component description when submitting a component for use on the Grid -- include a brief descriptive name like GDA_BIN.

1.8 Propogating back the right exitcode

A job in the workflow is only released for execution if its parents have executed successfully. Hence, it is very important that the algorithm codes exit with the correct error code in case of success and failure. The algorithms should exit with a status of 0 in case of success, and a non zero status in case of error. Failure to do so will result in erroneous workflow execution where jobs might be released for execution even though their parents had exited with an error.

The algorithm codes should catch all errors and exit with a non zero exitcode. The successful execution of the algorithm code can only be determined by an exitcode of 0. The algorithm code should not rely upon something being written to the stdout to designate success for e.g. if the algorithm code writes out to the stdout SUCCESS and exits with a non zero status the job would be marked as failed.

In *nix, a quick way to see if a code is exiting with the correct code is to execute the code and then execute echo $?.

```
$ component-x input-file.lisp
... some output ...
$ echo $?
0
```

If the code is not exiting correctly, it is necessary to wrap the code in a script that tests some final condition (such as the presence or format of a result file) and uses exit to return correctly.

1.9 Static vs. Dynamically Linked Libraries

Since there is no way to know the profile of the machine that will be executing the code, it is important that dynamically linked libraries are avoided or that reliance on them is kept to a minimum. For example, a component that requires libc 2.5 may or may not run on a machine that uses libc 2.3. On *nix, you can use the ldd command to see what libraries a binary depends on.

If for some reason you install an algorithm specific library in a non standard location make sure to set the LD_LIBRARY_PATH for the algorithm in the transformation catalog for each site.

1.10 Temporary Files

If the algorithm codes create temporary files during execution, they should be cleared by the codes in case of errors and success terminations. The algorithm codes will run on scratch file systems that will also be used by others. The scratch directories get filled up very easily, and jobs will fail in case of directories running out of free space. The temporary files are the files that are not being tracked explicitly through the workflow generation process.

1.11 STDOUT/STDERR Handling

The stdout and stderr should be used for logging purposes only. Any result of the algorithm codes should be saved to data files that can be tracked through the workflow system.

1.12 Configuration Files

If your code requires a configuration file to run and the configuration changes from one run to another, then this file needs to be tracked explicitly via the Pegasus WMS. The configuration file should not contain any absolute paths to any data or libraries used by the code. If any libraries, scripts etc need to be referenced they should refer to relative paths starting with a ./xyz where xyz is a tracked file (defined in the workflow) or as $ENV-VAR/xyz where $ENV-VAR is set during execution time and evaluated by your application code internally.

1.13 Code Invocation and input data staging by Pegasus

Pegasus will create one temporary directory per workflow on each site where the workflow is planned. Pegasus will stage all the files required for the execution of the workflow in these temporary directories. This directory is shared by all the workflow components that executed on the site. You will have no control over where this directory is placed and as such you should have no expectations about where the code will be run. The directories are created per workflow and not per job/alogrithm/task. Suppose there is a component component-x that takes one argument: input-file.lisp (a file containing the data to be operated on). The staging step will bring input-file.lisp to the temporary directory. In *nix the call would look like this:

```
$ /nfs/software/component-x input-file.lisp↵
```

Note that Pegasus will call the component using the full path to the component. If inside your code/script you invoke some other code you cannot assume a path for this code to be relative or absolute. You have to resovle it either using a dirname $0 trick in shell assuming the child code is in the same directory as the parent or construct the path by expecting an enviornment variable to be set by the workflow system. These env variables need to be explicitly published so that they can be stored in the transformation catalog.

Now suppose that internally, component-x writes its results to /tmp/component-x-results.lisp. This is not good. Components should not expect that a /tmp directory exists or that it will have permission to write there. Instead, component-x should do one of two things: 1. write component-x-results.lisp to the directory where it is run from or 2. component-x should take a second argument output-file.lisp that specifies the name and path of where the results should be written.

1.14 Logical File naming in DAX

The logical file names used by your code can be of two types.

- Without a directory path e.g. f.a, f.b etc
- With a directory path e.g. a/1/f.a, b/2/f.b

Both types of files are supported. We will create any directory structure mentioned in your logical files on the remote execution site when we stage in data as well as when we store the output data to a permanent location. An example invocation of a code that consumes and produces files will be

```
$/bin/test --input f.a --output f.b
```

OR

```
$/bin/test --input a/1/f.a --output b/1/f.b
```

Note: A logical file name should never be an absolute file path. E.g. /a/1/f.a (there should not be a starting /)