GREASY2

Introduction

This intends to be a user's guide to the evolution of GREASY we call GREASY2. In the future we expect to remove the "2" from the name when the original GREASY is superseded. In this document we will use the name GREASY to refer GREASY2. As you probably already know, this is a utility software designed to easy the deployment of embarrassingly parallel simulations, in principle in any environment. Although currently the underlaying scheduler is based on MPI, we have great plans to extend it to make it more usable in different environments and even in a grid!. So, stay tuned !!!

The typical situation is a user who needs to submit a very large number of tasks (binaries). These tasks might have dependencies among them, in such case GREASY is able to handle these dependencies if they are properly indicated.

GREASY is basically a wrapper that launches a scheduler for the bunch of tasks that need to be sent. The user then submits the wrapper requesting some amount of resources and the internal scheduler will manage the individual task executions. At the end, the system writes a log file for the user to keep track of the executions that failed or succeeded. Eventually, a restart file is created with those tasks that didn't complete successfully for any reason.

Usage

The first thing the user have to do is to create a file with the list of tasks to execute. We will talk more on this file later. But keep in mind that this is a file driven process and all the tasks that the user wants to run have to be specified in a text file.
In this new version (GREASY2) the tasks files are **ONLY** compatible from the previous version if they **do not contain any dependency specification**. This is very important to have in mind because one of the things changed in the new version is the dependency specification syntax to make GREASY more robust.

Now it is time to take a look at the installation directory structure to understand the important files:

- <Instalation dir>/
  - bin/
    - **greasy** : this is the binary to run
    - **greasycolorlog** : Utility script to view the log file in a colorful and friendly way
  - etc/
    - **greasy.conf** : Global configuration file (See section below in this document)
  - doc/
    - **greasy_userguide.pdf** : This file
  - examples/
    - **bsc_greasy.job** : Example of a BSC submission script.
    - **example.txt** : Example of a tasks file.
    - **short-example.txt** : Short example of a tasks file.

Once the tasks are defined and since we supposed you want to submit the tasks to a batch system; then it is time to create a batch script. The following is an example batch script used in MareNostrum (http:/ / www. bsc. es/ marenostrum-support-services) Let's named it as "bsc_greasy.job".

```
#!/bin/bash
# @ job_name = greasy
```

```
# @ initialdir = ./
# @ output = greasy_%j.out
# @ error  = greasy_%j.err

#############################################################
# Fill here with a number of cpus according to your needs
# Keep in mind that one task is reserved for the master
#############################################################
# @ total_tasks = 4

########################################
# and set an appropiate wall_clock_limit
########################################
# @ wall_clock_limit = 01:10:00

MACHINE=`echo $BSC_MACHINE | tr '[:lower:]' '[:upper:]'`

if [ "$MACHINE" =  "UV100" ]; then
  MPIRUN=mnrun
elif [ "$MACHINE" = "NVIDIA"  ]; then
  MPIRUN=mnrun
else
  echo "Machine $BSC_MACHINE not supported"
  exit
fi

EXE=/gpfs/apps/$MACHINE/GREASY/2.0/bin/greasy

########################################
# Here goes the path to the file where you
# have the list of tasks
########################################
FILE=short-example.txt


########################################
# Here goes the path to the place where the
# logs will be written
# By default, if not set, it will use:
# ./greasy.log
########################################
export GREASY_LOGFILE=greasy-$SLURM_JOBID.log

$MPIRUN $EXE $FILE
```

As you can see there are only a few things to set up:

- **number of cpus to use** (remember that always one cpu is used to run the scheduler, so it is always recommended to set this parameter to the number of cpus needed plus one)
- **Wall clock time**. Depending on the batch system this parameter is often mandatory
- **Path to the tasks file**. Passed as the only parameter to the GREASY binary.
- **Set the environment variable GREASY_LOGFILE**. This controls the output of the logs generated by GREASY. If the variable is set, then this file is created and used to record the logs. If no variable is set, then the system will look into the global configuration file (*greasy.conf*) for the param *logFile*. If neither *logFile* nor the environment variable is found, the logs will be printed in standard error.

The final step is to submit the job. The syntax depends on the batch system you are working on. In the case of MareNostrum:

```
mnsubmit bsc_greasy.job
```

and that's it ... gr*easy* right?

Tasks file

This is the more important element as here the user defines the list of tasks to be executed. One important feature of the tasks file is the capability to express dependencies among tasks. Dependencies are specified at the beginning of the line, right before the task and between **[#** and **#]**. It is important to point out that only backward dependencies are allowed. That means that you can only add dependencies to tasks with ID less than the current task ID. What it is the same to say that you can only add dependencies to tasks defined in previous lines in the tasks file.

There are some simple rules to follow:

- *Each line is a task*: and we mean it. Each line contains exactly what you want to execute. For example the path to the binary followed by the arguments with the necessary redirections, etc, etc ...
- *Lines starting with # are comments.* So these lines are not processed.
- *There is a one-to-one correspondence between the file line number and the task ID* . That means, for example, if you have the first line commented and the second line contains a valid task, then this task will have ID=2. This was done in this way because it is easy to use an editor to show the file line numbers (ie: vim). So, anyone can easily address any task through its task ID just jumping into the corresponding file line number.
- *Use tasks ID to express dependencies:* at the beginning of the line, and between brackets and sharps (**[# <list of dependencies goes here> #]**), you can express the dependencies for this task.
- *And remember:*

1. **only backward dependencies are allowed**
2. use "**,**" to separate dependencies
3. use "**-**" to express a rank of dependencies. ie: [# 3-6 #] ... says that the current task depends on task 3 though 6 (including both)
4. use "**-**" also to express relative dependencies. ie: [# -1 #] ... says that the current task depends on the previous task. In the same way you can also write, for example [# -3 #] to express a relative dependency with the 3rd task before the current one.
5. you can combine "**,**" and "-" as you want

Example of a tasks file:

```
# this line is a comment
/bin/sleep 2
# the following task is will have task ID = 4 ;) . Tasks IDs 1 and 3 will not exist.
/usr/bin/hostname
/bin/sleep 5
```

```
/bin/sleep 6
/bin/sleep 7
# the following task will be run after completion of the "sleep 5"
[# 5 #]       /bin/sleep 9
# the following task will be run after completion of the "sleep 11"
[# -2 #]      /bin/sleep 11
# the following task have syntax errors because tasks 1 and 3 does not exist
[#1-3#]           /bin/sleep 13
# the following task will be run after completion of the "sleep 2" and  "sleep 5" and  "sleep 6" and  "sleep 7"
[#2, 5 - 7 #]      /bin/sleep 15
```

Global configuration file

There is a file to control the global behavior of GREASY. This file is **<Instalation dir>/etc/greasy.conf** and it is read at the beginning when greasy is run. The file is quite self-explanatory there are a few configuration options we will summarize:

1. **GreasyEngine**: Specify which scheduler will be used. So far only MPI is allowed but in the future it will be possible to use SLURM.
2. **StrictCheck**: This variable controls whether greasy will continue or stop if any error is found in the tasks file. For example, there could be some errors in the tasks file like syntax errors or tasks with bad dependencies, etc. In such cases greasy can go on running and (of course these tasks will fail and they will be reported at the end) execute the rest of valid tasks.
3. **JobLimit**: Limit of concurrent submitted jobs. Unlimited if not set.
4. **LogFile**: Tells greasy where to write the log file if there is no environment variable GREASY_LOGFILE set. If neither of both are set, the log is printed out to standard error.
5. **LogLevel**: Sets the "Loglevel" to use. The higher the number is, the more verbose the output will be. Each level includes the previous ones. Levels 2 or 3 are recommended.

```
These are the logLevel options:
0 : Silent. No log information will be generated.
1 : Error. Only fatal errors will be recorded.
2 : Warning. All errors and warnings will be recorded.
3 : Info. Standard information will be printed.
4 : Debug. Show debug information. Warning: Logs will grow very fast!
5 : Devel. Show development debug info.Warning: Logs will grow VERY VERY fast!
```

Outcomes

At the end of the execution the user will end up having (along with the .err and .out files generated for the batch system) two files:

- **The greasy log file** : (see above *logFile* in the global configuration file) Here the user can follow the work-flow for the tasks. See which ones completed sucessfully: when started, which processor was assigned to, how long the run took. Also you can see which ones failed and why. One important feature in the log file to check is the statistic information. This is very useful, for example, to see if you are wasting resources
- **<name of the tasks file>.rst** : This intends to be used as a restart file. When greasy ends, all failed tasks due to any reason, are recorded. Tasks that failed due to syntax errors (and all its dependents) are commented and some text is added to warn the user. If the run is stopped at some point, the restart file is also created and it will include also all the non-finished tasks (queued tasks, running tasks, etc...). This file is very useful to follow the progress

of the simulation and avoid rewriting new tasks files.

Article Sources and Contributors

GREASY2 *Source*: https://scire.bsc.es/index.php?oldid=5310 *Contributors*: Jnaranjo