Greasy User Guide

BSC Support Team (support@bsc.es)

2021-10-01

Contents

1 Introduction

Greasy is a tool designed to make easier the deployment of embarrassingly parallel simulations in any environment. It is able to run in parallel a list of different tasks, schedule them and run them using the available resources. It is the perfect tool to use, for example, when your application is a serial program, and you need to run a large number of instances with different parameters. Greasy packs all these separate runs and uses the resources granted to run as many tasks as possible in parallel. As this tasks finish, Greasy will continue starting the tasks that were waiting for resources.

Since one of the main principles of Greasy is to keep it simple for the user, the list of tasks is just that: a list of tasks in a text file. Then, each line in the file becomes a task to be run by Greasy. It is able to manage dependencies between tasks, or to rerun a task in case of failure if desired.

Greasy can be easily configured by default with a configuration file, and can be also customized for each particular execution using environment variables. It also provides a log

system where all greasy actions will be recorded to keep track of what is the progress of your run.

2 Installation

2.1 How to install Greasy

Greasy can be installed in any Linux machine or cluster. However, additional requirements may be necessary depending on the features desired:

- SLURM: If available, Greasy can use it to spawn tasks to remote nodes inside a job.
- SSH: If available, Greasy can use it to spawn tasks to remote nodes.
- MPI: If available, Greasy comes with an engine working over MPI that may be used to run the tasks in a cluster.

If none of these packages is available and usable, Greasy will only be able to run tasks using the local node. The standard installation procedure is:

```
tar xvzf greasy-X.Y.tar.gz
cd greasy-X.Y
./configure --prefix=<path-to-greasy-installation-dir>
make
make install
```

Additional configuration options may be specified to configure in order to enable the additional engines to the greasy runtime. See ./configure –help for details.

2.2 Installation structure

We will call GREASY_HOME to the root directory of the installation. Once installed, a Greasy installation will contain the following files under this directory structure:

- *bin/*
  - **greasy**: The program wrapper that runs greasy.
  - **greasybin**: The actual binary. It must not be called directly, but from the wrapper greasy.
  - **greasycolorlog**: Utility script to give some color to the greasy logfiles.
- *etc/*
  - **greasy.conf**: Global configuration file.
- *doc/*
  - **greasy_userguide.pdf**: This file.
- *example/*

- **bsc_greasy.*.job**: An example of job scripts to submit Greasy to the batch system. It is also valid to run Greasy interactively in any machine.
- **short-example.txt**: a very short example of task file.
- **example.txt**: an example of task file containing both valid and invalid tasks.

3 Greasy Usage

3.1 Writing the task file

The task file is the most important element, as here the user defines the tasks to be executed. As mentioned before, each line of the file becomes a task from Greasy's point of view. This way, each line contains exactly what you want to execute, for example, the path to the binary followed by the arguments with the necessary redirections. The line number corresponds to what we call the task ID. This task ID is useful to understand the log file and when dependencies are needed.

Lines starting with # are treated as comments, and are not processed, but still reserve an 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 way, it is easy to use an editor to show the file line numbers (ie: vim), and address any task through its task ID just jumping into the corresponding file line number.

Working directory is specified at the beginning of the line, preceding the task command and between [@ and @]. This syntax allows to execute the task in the directory specified. As an example:

```
task 1 # executed on the directory which greasy has been launched from
[@ /home/user/greasy @] task 2 # executes ‘‘cd /home/user/greasy && task 2’’
```

Dependencies are specified at the beginning of the line, right before the task and between **[**# and #**]**. All tasks that must end before the current task must be included here, and more than one task may be included as a dependency. The best way to understand dependency syntax is through an example. This is an example task file with some dependencies among tasks:

```
task 1
[# 1 #] task 2
[# -2, 2 #] task 3
task 4
[# 2-4 #] task 5
```

In this case, tasks 1 and 4 do not have any dependencies. Task 2 depends on task 1, and task 3 depends on tasks 1 and 2. Note that with the "-2" we specify a relative dependency: the parent task is the one 2 lines above, which is task 1. And finally, task 5 depends on tasks 2, 3 and 4. In this case, we specified the dependencies as a range. Remember that you can add more dependencies separating the elements with commas, as in task 3.

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. In other words, you can only add dependencies to tasks defined in previous lines in the task file.

In summary, when writing a Greasy task file, just keep in mind these simple rules:

- Each line is a task
- Lines starting with # are comments.
- There is a one-to-one correspondence between the file line number and the task ID.
- It is possible to specify the working directory for each task with [@ <path> @].
- Use tasks ID to express dependencies at the beginning of the line, and between brackets and sharps **[**# <list of dependencies goes here> #**]** .
- And regarding dependencies, remember:
  - Only backward dependencies are allowed.
  - Use "," to separate dependencies.
  - Use "**-**" to express a rank of dependencies. i.e.: [# 3–6 #] means that the current task depends on task 3 though 6 (including both).
  - Use "**-**" also to express relative dependencies. i.e.: [# -1 #] means that the current task depends on the previous task.
  - You can combine "**,**" and "-" as you want to separate tokens.
- A reference example (example/example.txt) of the syntax could be the following:

```
# this line is a comment
/bin/sleep 2
# the following task is 4. Tasks IDs 1 and 3 do not exist.
/bin/sleep 4
/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 9"
[# -2 #] /bin/sleep 11
# the following task is invalid because tasks 1 and 3 do not exist
[#1-3#] /bin/sleep 13
# the following task will be run after completion of tasks 2, 5, 6 and 7
[#2, 5 - 7 #] /bin/sleep 15
# the following task will be executed on the directory /tmp/scratch
[@ /tmp/scratch @] pwd
# it is possible to combine dependencies and working directory for a task
[@ /tmp/scratch @][# -2 #] echo ‘‘It works!’’
```

3.2 Configuring Greasy

Greasy is configured using two different methods when it is executed: the configuration file and environment variables. It is mandatory to have a valid configuration file with the basic parameters set, and is located at GREASY_HOME/etc/greasy.conf.

In addition to the file, these parameters can be defined or overridden by the environment variables. That means that a parameter defined both in the configuration file and the environment will take the value of the latter.

Here is a brief summary of the different parameters:

- **GreasyEngine**: Specify the engine to use to schedule and run the jobs. Default is basic.
  - *basic*: It is the standard implementation of the engine. It only allows local runs unless ssh or Slurm are available in the system (See BasicRemoteMethod).
  - *mpi*: MPI implementation of the scheduler. Only available if MPI is present and enabled at build time. When using this mode, one cpu will be reserved only for scheduling tasks at runtime.
  - *thread*: thread implementation for shared memory computers. It only works in a single node.
- **BasicRemoteMethod**: Spawn method for the Basic engine. Values are *ssh*, *srun* or *none*. If none is set, Basic engine will only support local runs.
- **StrictCheck**: This variable controls whether greasy will continue or stop if any error is found in the tasks file. For example, there could be syntax errors or tasks with bad dependencies, etc. In such cases greasy can keep on running and execute the rest of valid tasks. Of course, these tasks will be marked as invalid and they will be reported as such at the end. Possible values: *yes* or *no*.
- **MaxRetries**: Maximum number of retries of a task in case of failure. If value is 0 or it is not set, then no retries will be attempted if any task fails. Possible values: a number $>= 0$.
- **LogFile**: Path to the file where the log will be written. If not set or empty, the log entries will be printed out to standard error.
- **LogLevel**: Verbosity of the log. The higher the number is, the more verbose the output will be. Each level includes the previous ones. Level 3 is recommended for most cases. Possible values are:
  - *0* : Silent mode. No log information will be recorded..
  - *1* : Error mode. Only fatal errors will be recorded.
  - *2* : Warning mode. All errors and warnings will be recorded.
  - *3* : Information mode. Standard information, warnings and errors will be recorded.
  - *4* : Debug mode. Information mode + some debug information.
  - *5* : Developer mode. Debug mode + developer information about the internals of Greasy. Not recommended for normal use.
- **Nworkers**: The number of concurrent tasks that will run in parallel. It is better if defined using environment, as it is much more flexible across different Greasy executions. At least one worker has to be defined in order to run Greasy.

- **NodeList**: The list of nodes where Greasy will run tasks for the basic engine. It has to be at least Nworkers nodes separated by commas. If not set, Greasy will run only in the local node. Note that if multiple workers have to run in the same node, that node has to be in the list as many times as that number of workers per node. For example, with quad-core nodes, each node in the list should appear 4 times.

There are some considerations regarding the configuration of Greasy:

- Greasy has native support for Slurm clusters. If Greasy detects that it is running inside a Slurm job, it will automatically set Nworkers and NodeList parameters according to the job resources, and will redirect the log file to greasy-<jobid>.log.
- In order to set any parameter via environment, the name of the variable will be *GREASY_* followed by the name of the parameter in uppercase. For example, to change the *LogFile* parameter, the corresponding environment variable would be *GREASY_LOGFILE*.

3.3 Running Greasy

There is an example of script to run greasy under *example* directory in *GREASY_HOME*. The best way to get started with Greasy is copying this example directory to the place where you want to work, and then use the provided templates to set up your Greasy execution.

Greasy can be run interactively or using a batch system. There is an example script called *bsc_greasy.job*, which can be adapted to work in both ways. If run interactively, the following changes are needed before launching the script:

- Set FILE to the path of your task file.
- Set the GREASY_NWORKERS and GREASY_NODELIST to the desired values.
- Optionally, modify the GREASY_LOGFILE if you want the log to be written in another file.

As mentioned above, the script is also prepared to be submitted as a BSC /RES job using mnsubmit. In this case, some additional modifications to the script are necessary:

- Set the total_tasks directive to the number of workers in the job (plus one if using MPI engine).
- Set the wall_clock_limit directive to the estimate amount of time to run the entire task file.
- There is no need to set GREASY_NWORKERS or GREASY_NODELIST in this case, since the values will be automatically inferred from job configuration.

3.4 Understanding the log file

While Greasy is still running or after its execution, it is a good idea to check the log. The log provides information of what is being done at any time, when and where the tasks are executed, and possible errors or problems occurred during the execution. LogLevel 3 (Information mode) is recommended as it generates only the minimum useful information.

Here is an example of log records extracted from an execution of the short-example.txt with 3 workers:

```
[2012-02-14 16:50:15] Start greasing short-example.txt
[2012-02-14 16:50:15] INFO: BASIC engine is ready to run with 3 workers
[2012-02-14 16:50:15] INFO: Allocating task 1
[2012-02-14 16:50:15] INFO: Allocating task 2
[2012-02-14 16:50:15] INFO: Allocating task 3
[2012-02-14 16:50:20] INFO: Task 3 completed successfully on node lnx.site. Elapsed: 00:00:05
[2012-02-14 16:50:25] INFO: Task 2 completed successfully on node lnx.site. Elapsed: 00:00:10
[2012-02-14 16:50:35] INFO: Task 1 completed successfully on node lnx.site. Elapsed: 00:00:20
[2012-02-14 16:50:35] INFO: BASIC engine finished
[2012-02-14 16:50:35] INFO: Summary of 3 tasks: 3 OK, 0 FAILED, 0 CANCELLED, 0 INVALID.
[2012-02-14 16:50:35] INFO: Total time: 00:00:20
[2012-02-14 16:50:35] INFO: Resource Utilization: 58.33\%
[2012-02-14 16:50:35] Finished greasing short-example.txt
```

As you can see, every record comes with a timestamp as a prefix, making easier to understand the behaviour of Greasy and the task scheduling throughout the whole execution. Having a quick look at the log, we see that the program started at 16:50:15 and finished at 16:50:35. We can also observe that the engine used is the default *basic* and that it is configured with 3 workers.

Since there are only 3 tasks to allocate and 3 workers, Greasy allocates all the tasks to all workers at the beginning. Later, Greasy records tasks completions reporting success or error, the node where the task was launched and the time it took to run.

Finally, when all tasks have been executed, Greasy shows some statistics about the execution, such as the tasks completed successfully, the failures or the canceled tasks. It is also very useful to check the global elapsed time and the resource utilization. The higher this latter value is, the more efficiently Greasy used the resources available to run the tasks. If you feel that this number is too low, consider changing the number of workers in order to have them busy the maximum time possible.

3.5 Something went wrong: the restart file

Sometimes things do not work as expected, and it is possible that some tasks or even the entire Greasy execution finishes abnormally. When that happens, Greasy generates a restart file containing those tasks that have failed, have been canceled or have just not been executed. The restart file is a valid task file for Greasy, so it can be used to run the tasks that could not run properly the first time. Here is an example of the restart file, running example.txt:

```
#
# Greasy restart file generated at 2012-02-14 16:53:50
# Original task file: example.txt
# Log file: greasy.log
#
# Warning: Task 2 failed
/usr/bin/hostname
# Warning: Task 13 was cancelled due to a dependency failure
```

```
[# 8 #] /bin/sleep 13
# Warning: Task 15 failed
/usr/bin/hostname
# Warning: Task 22 failed
[ 1 ] /bin/sleep 22
# Warning: Task 24 failed
[ 1 #] /bin/sleep 24
# Invalid tasks were found. Check these lines on example.txt:
# 23, 26, 27, 29, 31, 32
# End of restart file
```

There is a little header which provides information about the restart itself: when it was created, the task file that was being executed and the log file of the execution. This information is very useful to link the tasks in the restart with the original execution and figure out what happened.

When a task was failed or cancelled, Greasy adds a comment identifying the task in the original file and telling the reason why it is in the restart. If the task was not able to run because Greasy was told to quit before, then there will be no comment added. Finally, At the end of the restart file, if there were invalid tasks in the original task file, because there were syntax or semantic errors, they are listed with their corresponding IDs.

4 Support & Contact

For any question, doubt or bug report about Greasy, please send an e-mail to:

- support@bsc.es

Greasy is in continuous process of development, so if you feel that you have an idea to improve greasy with new features, or just want to provide some feedback, do not hesitate to contact to the address above. We will be delighted to receive your comments and suggestions to make Greasy even more useful and powerful.