MULE: compile, execution and postprocessing framework

Martin Schreiber

February 24, 2019

Abstract

This is a description of SWEET's build and execution framework. The reason for this development is a unified framework to support execution in a variety of environments without requiring to make changes in the scripts throughout many scripts used in the software development.

With Python being available on all systems, SWEET strongly uses Python in a variety of cases, from compilation, job script generation to analysis of output data.

Contents

1 Terminology

- *Platform*:
  A computer system which can be e.g. a laptop, single workstation, supercomputer
- Physical resources:
  - *Core*:
    Physical processing cores, each core can have its own execution thread without being stalled by other cores.
  - Socket:
    On multi-core systems, several cores are realized on one processor which is attached to a socket. Typically, there's a dedicatd memory controller to each socket. Therefore, sockets must be considered separately to be able to optimize for efficient memory access.
  - *Node*:
    One compute node can consist out of multiple sockets which can access a shared memory. Nodes are able to communicate with each other via distributed memory access interfaces such as MPI.
- Logical resources:
  - *Rank*:
    A rank which is associated to a set of cores on a particular node.
    One rank cannot be associated to multiple nodes
  - *Thread*:
    A set of threads for concurrent execution within one rank.
    One rank can have multiple exclusive threads assigned to it.
    The number of threads is not required to be identical to the number of cores per rank!
- Software:
  This refers to any program which is using MULE.
  [SWEET] The SWEET software development is the only one using MULE so far.

2 Processing pipeline

The execution of software in the area of scientific computing is a non-trivial task, once facing high-performance computing requirements and the pipeline involved of it. Assuming a software product to be given, several steps are involved to execute the software and to postprocess its data. MULE is intended to support these steps under the assumption that the software is willing to support interfaces to and from MULE.

2.1 Requirements & Strategy

We start with a requirement overview imposed by scientific computing.

2.1.1 Parameter studies

It should be possible to run a variety of parameter studies. Such parameter studies can include

1. **Simulation parameters**
   Examples: Density, initial condition.
2. **HPC parameters**
   Examples: Number of cores and nodes, threading activated/deactivated.
3. **Compile parameters**
   Examples involve both different simulations, different HPC parameters or debug features.

Strategy Provide an interface to/from MULE for all these parameters and to make it processable in a convenient way.

2.1.2 Performance portability

The software should be executable on a variety of different platforms (supporting Linux)

- Laptops
- Workstations
- Compute Clusters
- Super Computers

The challenge here is, that all these systems require a different execution environment. E.g. programs on a laptop and on workstations are directly executed whereas programs on compute clusters are started by a scheduler based on an information provided via a batch job description file.

Strategy

- **Hiding platform name**
  The software user should not be required to check on which platform the software is executed on to set e.g. the number of cores per rank to its maximum. In contrast, the user should only see platform specific data via an abstraction interface which provides e.g. the maximum number of per-socket cores or the maximum number of ranks. Based on this, the user should describe how the software should be parallelized.
- **Execution transparency**
  The software user should only provide a description how the application should utilize parallel processing power (see above). The execution itself should be then transparent on laptops and compute clusters. In this way, jobs can be executed on only on compute clusters even if they were written on laptops, but they can be also executed in a variety of computer clusters with different hardware resources.
- **Platform description**
  Platforms which are used to execute the software on are described in an abstract way which is independent to each software.

2.1.3 Support for efficient postprocessing

Postprocessing of a large number of different simulations and coping with huge data sizes for each simulation can be a very challenging task. Therefore, the generated data in form of output files and console output should be post processable in a very efficient way.

Strategy Provide convenient postprocessing scripts which support MULE users in

- extraction of data and the
- consolidation of the data

3 Files

MULE consists out of two components, the generic MULE parts which can be shared among different software developments and the software-specific MULE parts which typically have to be provided by the software developer.

3.1 Setting up MULE's environment variables

MULE relies on properly setup environment variables

```
# Setup directory where 'mule/' and 'mule_local/' directories are stored into
# This is typically the software's root directory
export MULE_SOFTWARE_ROOT="$PWD"

# Setup paths in environment to 'mule/' and 'mule_local/'
export MULE_ROOT="$MULE_SOFTWARE_ROOT/mule"
export MULE_LOCAL_ROOT="$MULE_SOFTWARE_ROOT/mule_local"
```

After this, MULE is able to load all other environment variables via

```
source $MULE_ROOT/bin/load_platform.sh $@
```

After this, we're ready to go.

[SWEET specific], all this is done by loading SWEET's environment variables via

```
source ./local_software/env_vars.sh
```

3.2 MULE's software independent files

We start with the MULE files which are independent to the software (e.g. SWEET). These are available in the directory ./mule/ and are briefly described as follows. It should not be required for standard software developers to make any changes in these directories.

- **mule/bin/**
  - load_platform.sh
    This file should be called to load platform specific environment variables.
    [SWEET] This script is e.g. included by ./local_software/env_vars.sh
  - mule.*
    These are very important **MULE helper scripts** which are discussed separately in Section 3.4.
- **mule/platforms/**
  This directory contains descriptions for a variety of different platforms as well as additional environment variable settings (e.g. platform specific compiler). See mule/platforms/README for additional information and Section 3.5.
- **mule/python/**
  MULE's Python package
  - mule/python/mule:
    - exec_program.py:
      Convenience function to execute a program

        - InfoError.py:
          Pretty output for information and Errors
        - JobParallelizationDimOptions.py
          Specify the parallelization degrees in each dimension (see examples)
        - JobParallelization.py
          Information on the parallelization for the entire simulation (based on DimOptions above).
        - JobPlatformResources.py
          Description of available resources for each individual platform
        - JobPlatforms.py
          Accumulation of all available platforms
        - mule/python/mule/plotting/:
          Plotting support (just for convenience)
        - mule/python/mule/postprocessing/:
          Scripts to support the consolidation of the output data

3.3 MULE's software dependent files

[SWEET] This section is SWEET-specific. Travis CI is a continuous integration server to validate SWEET based on tests provided in '[SWEET]/tests'.

- mule_local/bin/

    - mule.travis.install_git_hook
      [TRAVIS-CI] Install the git hooks to automatically update the Travis description file in '[SWEET]/.travis.yml' every time a git commit is done based on the following file:
    - mule.travis.setup_tests.py
      [TRAVIS-CI] Update '[SWEET]/.travis.yml' based on '[SWEET]/tests/'

- mule_local/git_hook/pre-commit
  [TRAVIS-CI] Git hook for Travis CI

- mule_local/platforms/*/local_software_default.sh
  These scripts provide information on additional 3rd party libraries which has to be installed for each individual platform. Since this has to be specified individually for each platform, it's stored in 'mule_local'

- mule_local/python/mule_local/:
  Software specific Python packages

    - mule_local/python/mule_local/:
    - JobCompileOptions.py
      Software-specific compile options
    - JobGeneration.py
      Software-specific class to support job generation
    - JobMule.py
      This class includes everything.
    - JobRuntimeOptions.py
      Runtime parameters (software program options)

- [SWEETROOT]/SConstruct
  SCons makefile replacement

3.4 MULE helper scripts

There are various helper scripts available to unify the execution and postprocessing of jobs with MULE on different platforms. These scripts are available in 'mule/bin/' which is included in the environment variable PATH. Therefore, these scripts can be directly executed in the command line without specifying the full path.

- load_platform.sh
  Load platform specific environment variables
- mule.benchmark.cleanup_all
  Call 'mule.benchmark.cleanup_job_dirs', 'mule.benchmark.cleanup_plans' and 'mule.benchmark.cleanup_postprocessi
- mule.benchmark.cleanup_job_dirs
  Remove all job_* directories
- mule.benchmark.cleanup_plans
  Remove 'shtns_cfg', 'shtns_cfg_fftw', 'sweet_fftw' and 'genplans_*'
- mule.benchmark.cleanup_postprocessing
  Remove 'output_*txt', 'output_*pdf'
- mule.benchmark.jobs_cancel [job ids...]
  (Compute Cluster specific) Cancel particular jobs currently submitted to job scheduler
- mule.benchmark.jobs_cancel_all
  (Compute Cluster specific) Cancel all jobs currently submitted to job scheduler
- mule.benchmark.jobs_interactive
  (Compute Cluster specific) Request an interactive job
- mule.benchmark.jobs_run_directly
  Run jobs directly within the current shell
- mule.benchmark.jobs_status
  Return the status of all jobs
- mule.benchmark.jobs_submit
  Submit all jobs in the current directory
- mule.benchmark.jobs_submit_with_queue_limit
  Submit jobs in a way which respects the queue limit. Once reaching the queue limit, it waits until a job is finished before submitting another one.
- MULE.benchmarks.jobs_multiplexer
- mule.benchmark.tar_nodata.sh
  Generate a tarball of the benchmark directory without the simulation output data (stored in .csv files)
- mule.jobdata [job directory]
  Print all Python-accessible information which is available for the given job.
  This is extremely handy for postprocessing.

3.5 MULE Platforms

We briefly describe the platform configurations in 'mule/platforms/'. To add a new platform, it's best to duplicate an existing platform (e.g. 99_default). The first numbers specify the priority (00: highest, 99: lowest) of the order in which the platform-specific information is processed.

3.5.1 platforms/??_[platform_id]/env_var.sh

Platform specific environments

3.5.2 platforms/??_[platform_id]/JobPlatformAutodetect.py

autodetect() function which returns if the current platform matches

3.5.3 platforms/??_[platform_id]/JobPlatform.py

Must implement the following interfaces¨ with a reference to JobGeneration handed over as a parameter to all these functions

- *get_platform_id*: Unique string for platform (e.g. pedros_awesome_laptop)
- *get_platform_autodetect*: Return true if this platform was detected, based on MULEPlatformAutodetect.py
- *get_platform_hardware*: Return MULEPlatformHardware with filled in information
- *jobscript_setup*: setup return of job script content
- *jobscript_get_header*: header (e.g. scheduler print_information) for job script
- *jobscript_get_exec_prefix*: prefix before MPI executable
- *jobscript_get_exec_command*: MPI execution (something like "mpirun -n ###
- *jobscript_get_exec_suffix*: suffix aftere MPI executable
- *jobscript_get_footer*: footer (e.g. postprocessing) for job script
- *jobscript_get_compile_command*: suffix aftere MPI executable

4 Using MULE with SWEET

4.1 Setting up the environment (First step!)

Before doing anything within SWEET, we have to load the SWEET environment variables which can be accomplished by

```
[SWEETROOT]$ source ./local_software/env_vars.sh
```

Doing so, there should be a prefix [SWEET...] at the beginning of every line of the prompt.

4.2 Manual building (without MULE and only recommended for development and debugging)

The SCons build system is used to compile SWEET programs. Type

```
$ scons --help
```

in the root SWEET folder for more information. Specifying a program to compile, e.g. 'swe_plane', we can also get additional compile information via

```
$ scons --program=swe_plane --help
```

HINT: Programs can be also compiled with the support of MULE (following section). Then, the SCons parameters are printed to the console using the compile scripts generated by MULE.

4.3 Building via MULE

This section briefly discusses how the job generation works

4.3.1 Initialize JobGeneration

We first get a fresh JobGeneration instance:

```
jg = JobGeneration([platform_id_override])
```

JobGeneration is the centralized point for SWEET's jobscript generation.

User-specified platform Instead of auto-detecting the environment, one can also request a particular platform with the *platform_ id_ override* parameter. An alternate to this is to specify the SWEET_PLATFORM environment variable.

Options A variety of options is made accessible which can be set by the user:

- jg.**runtime**: JobRuntimeOptions:
  with parameters for executing SWEET programs
- jg.**compile**: JobCompileOptions
  with compile-time parameters
- jg.**parallelization**: JobParallelizationOptions
  with options regarding parallelization to execute SWEET on clusters
- jg.platforms: list(JobPlatforms)
  with all available platforms
- jg.platform: JobPlatform
  with the currently used platform (from self.platform). This is automatically loaded based on the information in 'mule/platforms/*/JobPlatform.py'
- jg.platform_hardware: MULEPlatformHardware
  With information provided by platform on the available hardware (number of nodes, number of cores per node, etc.)
- jg.platform_functions: Callback functions which have to be implemented by each platform's description. Such callbacks are related to e.g. returning particular parts of a batch job description file.

The 'runtime', 'compile', and 'parallelization' parts also reflect parts of the described requirements (see Sec. 2.1.1)

4.3.2 SWEET compile & runtime options

Next, some SWEET options can be specified where we differentiate between compile and runtime options.

Program compile options

```
...
jg.compile.program = 'swe_sphere'
jg.compile.mode = 'release'
jg.compile.sphere_spectral_space = 'disable'
...
```

Program runtime options

```
...
jg.runtime.verbosity = 2
jg.runtime.space_res_spectral = 128
...
```

4.3.3 Program (parallel) execution options

SWEET was originally developed to study parallel-in-time approaches, therefore requiring an additional dimension also for parallelization. Without loss of generality, we use only a single dimension for the spatial parallelization. With more than one dimension for parallelization, we have to specify the way how to parallelize in each dimension. For a separation of concert, we do this first of all individually for each dimension:

```
pspace = JobParallelizationDimOptions()

# Use all cores on one socket for each rank
pspace.num_cores_per_rank = jg.platform_hardware.num_cores_per_socket

# Use only one rank in space, since MPI parallelization in space is not available
pspace.num_ranks = 1

# Use as many threads as there are cores per rank
pspace.num_threads_per_rank = pspace.num_cores_per_rank

ptime = JobParallelizationDimOptions()

# Use only one core per rank in the time dimension
pspace.num_cores_per_rank = 1

# Limit the number of ranks by param_max_space_ranks
ptime.num_ranks = min(
param_max_time_ranks,
jg.platform_hardware.num_nodes*jg.platform_hardware.num_sockets_per_node
)

# Use same number of cores for threading
ptime.num_threads_per_rank = ptime.num_cores_per_rank
```

Note, how easy we can specify e.g. to use the max. number of cores on one socket depending on the platform. On a different platform, this will also lead to the utilization of the full number of cores on one socket.

Disclaimer: Even if it is possible to specify a variety of different configurations, this must be supported by the platform specific implementation

The different dimensions are finally combined with

```
jg.parallelization.setup([pspace, ptime], mode)
```

where *mode* does not yet exist, but is planned to specify the way how the space and time parallelization is combined together.

More information on this is available in JobParallelization.py.

Note, that every time if the parallelization parameters are changes, *p.parallelization.setup(...)* must be executed again!

4.3.4 Generation of job script

To generate the job script, we can simply call

```
jg.gen_jobscript_directory('job_bench_'+jg.getUniqueID())
```

where jg.getUniqueID creates a parameter-specific unique ID.

The method

- creates the required directory
- creates the job script file 'run.sh' including the compilation commands before execution of the binary and
- writes out the job script file.

4.3.5 Pickled data for postprocessing (!!!)

One of the most important parts of MULE's way of job generation is that all the parameters which describe how the particular job script was generated is stored in a post processable format in each job's directory in the file 'jobgeneration.pickle'. This allows to extract the information for each job for postprocessing.

4.3.6 Pre-compilation

For HPC systems, it's very often required to compile code on the login nodes. We can accumulate the compile commands by specifying

```
jg.compilecommand_in_jobscript = False
```

This accumulates all compile commands and also ensures, that each compile command exists only once. After all job scripts were generated, we can write out a script with compile commands, e.g. using

```
jg.write_compilecommands("./compile_platform_"+jg.platforms.platform_id+".sh")
```

4.4 Executing jobs

The jobs can be executed directly on the current computer by using

```
$ mule.benchmark.jobs_run_directly
```

or they can be submitted to a compute cluster via

```
$ mule.benchmark.jobs_submit
```

4.5 Checking for finished jobs

On a compute cluster, we can check for all jobs to be finished with

```
$ mule.benchmark.jobs_status
```

which prints the status of all still enqueued or running jobs.

4.6 Postprocessing jobs

Once the jobs are executed, the postprocessing starts and can consist out of several steps.

4.6.1 JobData description

MULE supports the postprocessing by extracting information from the job's console output and from pickled file which is described in the following sections. Such an information can be requested e.g. via

```
$ mule.jobdata job_bench_[...]
```

This will output the content of a dictionary which is generated based on the information available in the specified job directory. Note, that this is based on a 'JobData' class provided by MULE which becomes handy for postprocessing. Next, we will discuss the different ways how to add entries for this dictionary.

4.6.2 Job's console output: [MULE]

Each job's output to the console is stored into a file 'output.out'. This is processed by 'JobData' and searched for lines starting in the following way:

1. Detect lines starting with '[MULE] ', skip other lines
2. The following string is used as a dictionary entry with the name prefixed with 'output.'
3. The string followed after the colon and space (': ') is used as a value.

An example is given by

```
[MULE] simulation_benchmark_timings.main: 19.203482231
```

which will lead to a dictionary entry (using Python notation)

```
'output.simulation_benchmark_timings.main': '19.203482231',
```

in JobData.

4.6.3 Pickled Data

During the job generation, MULE is generating 'jobgeneration.pickle' files containing all information about the particular job. This is stored as a dictionary and is loaded directly by the JobData. To give an example of this information, we see that:

```
compile.sphere_spectral_space => enable
compile.sweet_mpi => enable
compile.threading => omp
compile.unit_test =>
runtime.bench_id => -1
```

4.6.4 Generating pickled data for MULE-processable output

For some cases, the data is not already accessible by the previously described outputs. This can be e.g. the case if the output data should be compared to a reference data, basically reducing the amount of data to a few scalar values.

MULE supports this, as long as this data is written into pickle files.

Generating a file '[filename].pickle' by pickling a dictionary, these dictionaries will be directly added to the JobData, but the directory indices prefixed by '[filename].'.

4.6.5 Reading and processing job data

For the final postprocessing, we need to consolidate all available data. As a first step, we use

```
j = JobsData('./job_bench_*', verbosity=0)
```

which loads the pickled data of all jobs.

It is now possible to directly work on this data or to use further features of MULE:

Grouping together We can finally groups data together in the following way

```
c = JobsDataConsolidate(j)
print("")
print("Groups:")
groups = ['runtime.timestepping_method', 'runtime.max_simulation_time']
job_groups = c.create_groups(groups)
for key, g in job_groups.items():
    print(key)
```

5 Summary

That's it folks! May the Mule be with you!