SWEET's unified building & execution framework

Martin Schreiber

September 28, 2018

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 scrip generation to analysis of output data.

1 Terminology

- *Platform*:
  A computer system which can be e.g. a laptop, single workstation, supercomputer
- Physical resources:
  - *Core*:
    Physical processing cores
  - *Node*:
    One compute node which can have multiple sockets
  - *Socket*:
    One socket consists out of
- 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!

2 Files:

- [SWEETROOT]/platforms/*/
  - env_vars.sh:
    Environment variables to include for this environment, e.g. overrides for compilers
  - SWEETPlatform.py:
    Script which includes callbacks to gain platform-specific information
- [SWEETROOT]/python_mods:
  - InfoError.py
    Some convenience functions to output Information and Errors
  - SWEETCompileOptions.py
    Compile specific options

  - SWEETJobGeneration.py
    Main class used throughout job generation
  - SWEETParallelizationDimOptions.py
    Per-dimension specified parameters for parallelization
  - SWEETParallelization.py
    Accumulated parameters for parallelization (based on SWEETParallelizationDimOptions)
  - SWEETParameters.py
    Parameters for some commonly used simulation scenarios
  - SWEETPlatformHardware.py
    Information on hardware (number of cores, number of nodes, etc.) to be provided by Platform
  - SWEETPlatforms.py
    Platform detection mechanism based on information in [SWEETROOT]/platforms/*/SWEETPlatform.py
  - SWEETRuntimeOptions.py
    Runtime parameters

- [SWEETROOT]/SConstruct
  scons makefile replacement

3 Loading SWEET environment

Loading the SWEET environment is accomplished by including the file

`[SWEETROOT]/local_software/env_vars.sh`

in the current environment.

3.1 Example

```
martin@martinium:~/workspace/sweet$ source ./local_software/env_vars.sh
Default environment (nothing)
SUCCESS! SWEET environment variables loaded
[SWEET] martin@martinium:~/workspace/sweet$
```

Note the prefix [SWEET] which is now before every line of the prompt.

3.2 Platform specific parts

The script env_vars.sh looks up all files in

`$SWEET_ROOT/platforms/??_*`

and includes all env_vars.sh files from this directory. It is then up to the script to detect the local platform and provide some additional overrides, if required

3.3 Supported overrides

The following example shows how to set environment variables in the platform-specific env_vars.sh. This results in SCons using these compiler binaries

```
export SWEET_F90=gfortran
export SWEET_CC=gcc
export SWEET_CPP=g++
export SWEET_MPICC=mpigcc
export SWEET_MPICXX=mpigxx
export SWEET_MPIF90=mpifc
```

4 Building

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

```
$ scons --help
```

in the root SWEET folder for more information.

4.1 Platform-specific compilers

See above (Supported overrides)

4.2 Dependencies

The building process is triggered via the command "scons" which is a makefile replacement written in Python.

- SConstruct
  - sconscript
    Source code detection
  - python_mods/SWEETCompileOptions.py
    Accumulation of all compile options

4.3 Shared infrastructure with job generation

The compile options themselves are generated in SConstruct to avoid a dependency of SConstruct libraries in other Python codes.

SWEETCompileOptions.py only provides a container for all compile options to be shared with the generation of the job scripts. This allows to compile manually by using "scons" directly or indirectly by specifying compiler options and using SWEETCompileOptions to generate the scons parameters.

This adds plenty of flexibility for automatic job script generation:

- Unique program binary name identifier
- Unique job identifier including the program binary identifier
- All compile options accumulated in a single place

5 Job generation

This section briefly discusses how the job generation works

5.1 Initialize SWEETJobGeneration

The user of the building framework first generates a fresh JobGeneration instance:

```
p = SWEETJobGeneration([platform_id_override])
```

SWEETJobGeneration is the centralized point for SWEET jobscript generation.

5.1.1 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.

5.1.2 Options

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

- self.runtime: SWEETRuntimeOptions:
  with parameters for executing SWEET programs
- self.compile: SWEETCompileOptions
  with compile-time parameters
- self.parallelization: SWEETParallelizationOptions
  with options regarding parallelization to execute SWEET on clusters
- self.platforms: list(SWEETPlatforms)
  with all available platforms
- self.platform: SWEETPlatform
  with the currently used platform (from self.platform)
- self.platform_hardware: SWEETPlatformHardware
  With information provided by platform on the available hardware (number of nodes, number of cores per node, etc.)
- self.platform_functions: Callback functions in SWEETPlatform which have to be implemented by each platform.

Here, the platform-specific data such as in self.platforms is loaded from the matching platform in "platforms/*/SWEETPlatform.py

5.2 SWEET compile & runtime options

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

5.2.1 Program compile options

```
...
p.compile.program = 'swe_sphere'
p.compile.plane_or_sphere = 'sphere'
p.compile.plane_spectral_space = 'disable'
...
```

5.2.2 Program runtime options

```
...
p.runtime.verbosity = 2
p.runtime.mode_res = 128
p.runtime.phys_res = -1
...
```

5.3 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 = SWEETParallelizationDimOptions()
# Use all cores on one socket for each rank
pspace.num_cores_per_rank = p.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 = SWEETParallelizationDimOptions()
# 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,
p.platform_hardware.num_nodes*p.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 combined together in a particular way with

```
p.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 SWEETParallelization.py.

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

5.4 Generation of job script

To generate the job script, we can simply call

```
p.write_jobscript('script_'+p.getUniqueID()+'/run.sh')
```

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

The method

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

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

```
p.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

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

asdf

7 Platform configuration

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.

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

First, an autodetection must be performed

All scripts must include a test for the platform as well.

7.2 platforms/??_[platform_id]SWEETPlatform.py

Must implement the following interfaces¨ with a reference to SWEETJobGeneration 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
- *get_platform_hardware*: Return SWEETPlatformHardware 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