Science Build Rules

Apr 24, 2023

Contents

CHAPTER 1

Overview of science-build-rules

`science-build-rules` is a suite of Python classes and utilities that make it easier to create automated builds with Spack, Singularity and Anaconda. It is designed to run with `science-build-environment`, but it can also be run independently. The basic structure is described below:

Typical build will do the following steps:

1. Read and validate configuration. This is done by the `ConfReader`-class.
2. Build the software based on build rules. This is done by subclasses of the `Builder`-class.
3. Test the installed software. This step is not yet implemented.
4. Deploy software from the build system into a target system with a desired deployment strategy. This is done by subclasses of the `Deployer`-class.

Idea in "build-rules" is that a series of operations is performed in a specific way, defined by the `Builder`-class and configuration files. Each subclass of `Builder` is tool specific, for example `SpackBuilder` creates builds with Spack. The user chooses the subclass of `Builder` designed for the tool they wish to use and modifies configuration files to match their needs.

Before doing a build, configuration files are loaded in and validated. The build and deployment commands, which are based on the configuration files, are then predefined and wrapped in subclasses of the `Rule`-class. Each subclass of `Builder` and `Deployer` can have their own configuration yaml-files and corresponding schemas.

CHAPTER 2

Installation of science-build-rules

CHAPTER 3

Builders

3.1 Common features among builders

All of the builders share some common features.

3.1.1 Configuration file structure

When working with `science-build-environment`, all configuration files should be stored in the following file structure:

```
<build rules repo>/configs/<build target>/<builder>/*.yaml
```

e.g.:

```
~/science-build-rules/configs/centos/spack/packages.yaml
```

All configuration files should be in yaml-format.

3.1.2 build_config.yaml and deployment_config.yaml

All builders have these configuration files. `build_config.yaml` contains the configuration for the builders and its format depends on the builder. `deployment_config.yaml` contains configuration for the deployers. Its format is described in the *Deployers-page*.

3.2 Spack-builder

Spack-builder uses Spack for installing software. Before running the builder `spack` should be available in the shell that launches the build.

After validating the configuration structure, the build runs the following build rules:

1. Reindex installed packages
2. Remove old compilers configuration file
3. Add existing compilers
4. Install compilers
5. Install packages
6. Recreate modules

CHAPTER 4

Deployers

4.1 Common features among deployers

Deployers are shared among builders. Each Deployer has its own configuration format and deployer strategy.

4.2 Rsync deployer

Rsync deployer uses `rsync` to deploy software. The configuration needs to include at least:

```
- method: 'rsync'
  target_host: 'user@server'
  source: '/path/to/installation'
  dest: '/path/to/installation'
```

Other optional parameters are:

- `working_directory: '/path/to/working_directory'` *default:* **None**. A parameter that can be used to give the working directory for the rsync command in a case where relative paths need to be used instead of absolute paths.
- `delete: True/False` *default:* **False**. If set **True**, rsync deletes extraneous files from the dest dir.
- `rsync_flags: '[flags]'` *default: '-surlptDxv'*.
- `ssh_command: '[command]'` *default: ssh*. The ssh command for rsync.
- `set_sbit: True/False` *default:* **False**. If set **True**, sets sbit for the rsynced files and directories.

CHAPTER 5

Configuring a build

5.1 Spack builds

5.1.1 build_config.yaml

Overview

The main configuration for a spack build is in build_config.yaml.

The file should contain three keys:

- `target_architecture`: This dictionary defines the default target arcitecture.
- `compilers`: This array defines the desired compilers.
- `packages`: This array defines desired end products.

target_architecture

The `target_architecture`-dictionary should contain the following keys:

- `platform`: Platform that spack should target (e.g. `linux`).
- `os`: Operating system that spack should target (e.g. `centos7`).
- `arch`: Architecture that spack should target (e.g. `westmere`).

A sample configuration might be something like:

```
target_architecture:
  platform: linux
  os: centos7
  arch: westmere
```

compilers

The `compilers`-array consists of individual compilers as dictionaries. These compilers are evaluated in sequential order from top to bottom. System compilers that are used to install other compilers should be positioned at the start of the array.

Each compiler can contain the following keys:

- `name`: Name of the compiler in spack (e.g. `gcc`).
- `version`: Version of the compiler in spack (e.g. `9.2.0`).
- `system_compiler`: Boolean value that tells if the compiler is a system compiler (Default: false).
- `licenses`: Array of license files that need to be copied into the installation directory. More information on this at the licenses-page (TODO) (e.g. `[license.lic]`).
- `variants`: Additional variants that the installation should use (e.g. `+binutils` for `gcc`).
- `dependencies`: Additional dependencies for the installation. Compilers that are built by system compilers should depend on them. Further compilers should also depend on main compiler. Otherwise the compilers might try to build themselves again. (e.g. `%gcc@4.8.5` for `gcc@9.2.0` and `%gcc@9.2.0` for `intel-parallel-studio`).
- `extra_flags`: Array of extra flags that should be given to `spack install`-command (e.g. `--jobs 4` to limit the build to four cpus).
- `flags`: Dictionary of flag-parameters that should be written to `~/.spack/linux/compilers.yaml`. These flags are then added to every build done with these compilers. Possible keys are `cflags`, `cxxflags`, `cppflags`, `fflags`, `ldflags`, `ldlibs` (e.g. `{ 'cflags': '-g' , 'cxxflags': '-g' }` would compile all C and C++ codes with debug flags). Architecture flags are added automatically by `target_architecture`.
- `target_architecture`: Target architecture for building this compiler. This is important if the system compiler cannot compile software to the desired default architecture. Do note that this does not change the target for software built with this compiler. It only changes the target for compiling this compiler. Structure is the same as for `target_architecture`.

Only `name` and `version` are required, but in practice one usually needs to use most of the parameters. An example configuration might look something like this:

```
compilers:
  - name: gcc
    version: 4.8.5
    system_compiler: true
    flags:
      cflags: -O2 -g
      cxxflags: -O2 -g
      fflags: -O2 -g
  - name: 'gcc'
    version: 9.2.0
    variants:
      - +piclibs
    dependencies:
      - %gcc@4.8.5
    flags:
      cflags: -O2 -g -ftree-vectorize
      cxxflags: -O2 -g -ftree-vectorize
      fflags: -O2 -g -ftree-vectorize
    target_architecture:
```

(continues on next page)

(continued from previous page)

```
      platform: linux
      os: centos7
      arch: x86_64
  - name: intel-parallel-studio
    version: cluster.2019.3
    licenses:
      - license.lic
    dependencies:
      - %gcc@9.2.0
    flags:
      cflags: -O2 -g
      cxxflags: -O2 -g
      fflags: -O2 -g
    target_architecture:
      platform: linux
      os: centos7
      arch: x86_64
```

packages

The `packages`-array consists of individual packages as dictionaries. These packages are evaluated in sequential order from top to bottom.

Each package can contain the following keys:

- `name`: Name of the package in spack (e.g. `gcc`).
- `version`: Version of the package in spack (e.g. `9.2.0`).
- `licenses`: Array of license files that need to be copied into the installation directory. More information on this at the licenses-page (TODO) (e.g. `[license.lic]`).
- `variants`: Additional variants that the installation should use (e.g. `fabrics=verbs` for `openmpi`).
- `dependencies`: Additional dependencies for the installation. (e.g. `%gcc@9.2.0` or `^python@3:`).
- `extra_flags`: Array of extra flags that should be given to `spack install`-command (e.g. `--jobs 4` to limit the build to four cpus).
- `target_architecture`: Target architecture for building this package. Structure is the same as for `target_architecture`.

Only `name` and `version` are required. Default variants and versions should be set in `packages.yaml`. An example configuration might look something like this:

```
packages:
  - name: 'openmpi'
    version: 3.1.4
  - name: 'python'
    version: 3.7.4
  - name: 'r'
    version: 3.6.1
  - name: 'py-gpaw'
    version: 1.3.0
    variants:
      - '+fftw'
      - '+mpi'
      - '+scalapack'
```