RADIUSS CI Documentation

Release 0.1

Adrien M. Bernede

Aug 18, 2021

Shared Documentation

RADIUSS CI is a sub project form the RADIUSS initiative focusing on sharing resource and documentation on Continuous Integration among RADIUSS projects.

LLNL's RADIUSS project (Rapid Application Development via an Institutional Universal Software Stack) aims to broaden usage across LLNL and the open source community of a set of libraries and tools used for HPC scientific application development.

CHAPTER 1

Uberenv and CI Shared Documentation

In RADIUSS, we designed a streamlined process to build your project with its dependencies using Spack and Uberenv, and add a basic CI to test those builds in Gitlab.

Before getting started, it is a good idea to read the LC specific documentation for Gitlab. In particular, the "Getting Started" and "Setup Mirroring" sub-pages *will help*.

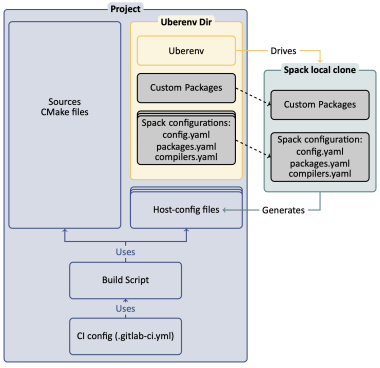
The main steps are:

1. Get Uberenv. See *Uberenv Guide*.
2. Setup CI. See *GitLab CI Guide*.
3. Create build_and_test script.

1.1 Uberenv Guide

This documents the setup and usage of Uberenv..

Uberenv can be used to generate custom host-config files, driven by a Spack spec. This host-config file will point to the dependencies installed with Spack, making the build process of the project straightforward.



1.1.1 Getting Started

Here are some preliminary steps to follow to setup Uberenv, depending on how you get Uberenv.

Getting Uberenv by clone/fetch/copy

1. Get uberenv.py script.

   Clone/Fetch/Copy it from LLNL/uberenv into a `uberenv` directory, not as a submodule.

2. Edit uberenv/project.json.

   Set your project package name, and other parameters like Spack reference commit/tag (we suggest the latest release tag).

3. Add radiuss-spack-configs submodule.

   - Use `git submodule add` to get radiuss-spack-config.
   - Create a symlink `uberenv/spack_configs` that points to `radiuss-spack-configs`.

4. Add custom packages.

   If you need to make local modifications to your project package or a dependency package, you may put it in a corresponding directory:
   `uberenv/packages/<package_name>/package.py`.

5. Make sure that <project>/package.py generates a host-config cmake file.

   This is usually done adding a specific stage to the package (see for example the hostconfig stage in Umpire, CHAI, etc.).

Getting Uberenv as a submodule

1. Get uberenv.py script.

   Use `git submodule add` to get uberenv into a `uberenv` directory.

2. Edit .uberenv.json.

   Create `.uberenv.json` in a directory that is a parent of `uberenv`. Set your project package name, and other parameters like Spack reference commit/tag (we suggest the latest release tag).

3. Add radiuss-spack-configs submodule.

   - Use `git submodule add` to get radiuss-spack-config in a second submodule or custom location.
   - In `.uberenv.json` set `spack_configs_path` to point to `<some_path>/radiuss-spack-configs`.

4. Add custom packages.

   - If you need to make local modifications to your project package or a dependency package, you may put it in a corresponding directory:
     `<some_path>/packages/<package_name>/package.py`.
   - In `.uberenv.json` set `spack_packages_path` to point to `<some_path>/packages`

5. Make sure that <project>/package.py generates a host-config cmake file.

   This is usually done adding a specific stage to the package (see for example the hostconfig stage in Umpire, CHAI, etc.).

1.1.2 Generating <Project> host-config files

This mechanism will generate a cmake configuration file that reproduces the configuration Spack would have generated in the same context. It contains all the information necessary to build <Project> with the described toolchain.

In particular, the host-config file will setup:

- flags corresponding with the target required (Release, Debug).
- compilers path, and other toolkits (cuda if required), etc.
- paths to installed dependencies.

This provides an easy way to build <Project> based on Spack and encapsulated in Uberenv.

Uberenv role

Uberenv helps by doing the following:

- Pulls a blessed version of Spack locally.
- If you are on a known operating system (like TOSS3), we have defined compilers and system packages so you don't have to rebuild the world, _e.g._ CMake, or MPI.
- Overrides <Project> Spack packages with the local ones if any. (see `scripts/uberenv/packages`).
- Covers both dependencies and project build in one command.

Uberenv will create a directory `uberenv_libs` containing a Spack instance with the required <Project> dependencies installed. It then generates a host-config file (`<config_dependent_name>.cmake`) at the root of <Project> repository.

Note: One common source of error when using Uberenv is that the `uberenv_libs` folder is out of date. To resolve, make sure this folder is deleted before running new scripts for the first time because this folder needs to be regenerated.

Before to start

Machine specific configuration

Depending on the machine/system, <Project> may or may not provide a spack configuration allowing to use uberenv right away.

Check in the machines/systems supported in `scripts/uberenv/spack_configs`. Per machine, <Project> will provide `compilers.yaml`, `packages.yaml`, and `config.yaml`. The latter being possibly shared with other machines/systems.

Vetted specs

Then, one can easily check what specs are tested in CI. For example, when looking for the gcc versions tested on quartz:

```
git grep "SPEC" .gitlab/quartz-jobs.yml | grep "gcc"
```

MacOS case

It is not trivial to provide a universal configuration for MacOS. Instead, the developper will likely have to complete the `packages.yaml` file in order to adapt the location and version of externally installed dependencies.

Using Uberenv to generate the host-config file

```
$ python scripts/uberenv/uberenv.py
```

Note: On LC machines, it is good practice to do the build step in parallel on a compute node. Here is an example command: `srun -ppdebug -N1 --exclusive python scripts/uberenv/uberenv.py`

Unless otherwise specified Spack will default to a compiler. It is recommended to specify which compiler to use: add the compiler spec to the `--spec=` Uberenv command line option.

On blessed systems, compiler specs can be found in the Spack compiler files in our repository: `scripts/uberenv/spack_configs/<system type>/compilers.yaml`.

Some options

We already explained `--spec=` above:

- `--spec=%clang@9.0.0`
- `--spec=%clang@8.0.1+cuda`

The directory that will hold the Spack instance and the installations can also be customized with `--prefix=`:

- `--prefix=<Path to uberenv build directory (defaults to ./uberenv_libs)>`

Building dependencies can take a long time. If you already have a Spack instance you would like to reuse (in supplement of the local one managed by Uberenv), you can do so with the `--upstream=` option:

- `--upstream=<path_to_my_spack>/opt/spack ...`

Using host-config files to build <Project>

When a host-config file exists for the desired machine and toolchain, it can easily be used in the CMake build process:

```
$ mkdir build && cd build
$ cmake -C  <path_to>/<host-config>.cmake ..
$ cmake --build -j .
$ ctest --output-on-failure -T test
```

CI usage

In RAJA, Umpire and CHAI, Uberenv is used in CI context to automate both the installation of dependencies and the generation on the host-config files.

All this is managed through a single script, that is usable outside of CI.

```
$ SPEC="%clang@9.0.0 +cuda" scripts/gitlab/build_and_test.sh --deps-only
```

```
$ HOST_CONFIG=<path_to>/<host-config>.cmake scripts/gitlab/build_and_test.sh
```

Note: Making the CI scripts usable outside CI context is recommended since, by definition, it has been vetted. It also ensures that this script is usable in interactive mode, making it easier to test.

1.1.3 Debugging

In the workflow described above, there are 4 levels of scripts to control the build of a package. From the lower to the higher level:

- The *build system* is controlled by the host-config file (generated by Spack or not).
- The *Spack package* is controlled by the spec provided and the configuration.
- *Uberenv* takes a spec and a json configuration file.
- A *build_and_test script* also sometimes called test driver. The one in Umpire and RAJA requires a spec and some other control variables.

Now, when it comes to debugging, each level has some requirements to reproduce a failing build:

- The *build_and_test script* typically runs in CI context. This means that it may not be designed to run outside CI. It is better if it does, and we try to do that in RADIUSS, but it is not guaranteed.
- *Uberenv* is meant to provide a turnkey way to install the project and its dependencies. It is usually a good way to reproduce a build on the same machine. The CI creates working directories in which the uberenv install directory _may_ persist, but it is better to reproduce in a local clone.
- Reproducing a build with *Spack* requires a deep knowledge of it. But Uberenv helps a lot with that. We advice that you use Uberenv to generate the Spack instance. Then, loading the spack instance generated and working with it is safe.
- Going down to the *build system* is also doable, especially when using host-config files. Once spack has installed the dependencies and generated the host-config files, it can be used to control the build of the code and this should not require using Spack.

1.2 GitLab CI Guide

This section documents the use of GitLab CI for various projects under the RADIUSS scope. All RADIUSS projects are hosted on GitHub. Their repositories are mirrored at LLNL using a self-managed GitLab instance to test the software on Livermore Computing (LC) machines.

The figure illustrates the dual CI/CD operations: Pull requests (PRs) and software releases trigger CI both in the Cloud and at LLNL.

1.2.1 Configuration

GitLab CI configuration is provided in the `.gitlab-ci.yml` file at the root of the project. Variables and stages are defined at a high level in the file while details are generally maintained separately to avoid having a large, monolithic configuration file. Different hardware architectures and software stacks are supported by LC so machine-specific GitLab runner `tags` are included to support limiting some jobs to specific machines.

The logically separated configuration blocks are maintained in files under the `.gitlab` subdirectory. Key stages are defined in separate `YAML` files included in the configuration with the `include` keyword. Shell script files defining the work performed by GitLab runners are specified under the `script` entry of the corresponding job.

Again, the configuration is split in logical blocks instead of having only one unique linear file. This is permitted by local `includes` feature in GitLab CI, and all the included files are gathered in `.gitlab` directory.

```
$ ls -c1 .gitlab
butte-jobs.yml
butte-templates.yml
lassen-jobs.yml
lassen-templates.yml
quartz-jobs.yml
quartz-templates.yml
```

Machine names is a logical divider in the configuration of our CI: different machines allow to test on various hardware architectures and software stacks, they are identified with runner `tags`. It is also a practical divider because script and in particular allocation method varies with the machine.

All jobs for a given machine: <machine>-jobs.yml

For each machine, build and test jobs are gathered in `<machine>-jobs.yml`. There, a job definition looks like:

```yaml
clang_9_0_0 (build and test on quartz):
  variables:
    SPEC: "%clang@9.0.0"
  extends: .build_and_test_on_quartz
```

With 3 elements:

- The jobs name, which has to be unique. Here `clang_9_0_0 (build and test on quartz)`.
- The `extends:` section with the inherited properties.
- The rest: everything specific to this particular job.

The goal is to use "inheritance" to share as much as possible, keeping the final job definition as short as possible.

So, what is `.build_and_test_on_quartz`? It is defined in the corresponding `<machine>-templates.yml`.

Machine specific templates: <machine>-templates.yml

```yaml
.on_quartz:
  tags:
    - shell
    - quartz
  rules:
    - if: '$CI_COMMIT_BRANCH =~ /_qnone/ || $ON_QUARTZ == "OFF"' #run except if ...
      when: never
    - if: '$CI_JOB_NAME =~ /release_resources/'
      when: always
    - when: on_success

.build_and_test_on_quartz:
  stage: q_build_and_test
  extends: [.build_toss_3_x86_64_ib_script, .on_quartz]
```

Inheritance can be nested and multiple. In practice it is pretty much working as an ordered inclusion where the lowest level keys are overridden by the latest if conflicting, higher level keys being merged.

At this point, our job looks like this:

```
clang_9_0_0 (build and test on quartz):
  variables:
    SPEC: "%clang@9.0.0"
  stage: q_build_and_test
  extends: [.build_toss_3_x86_64_ib_script]
  tags:
    - shell
    - quartz
  rules:
    - if: '$CI_COMMIT_BRANCH =~ /_qnone/ || $ON_QUARTZ == "OFF"' #run except if ...
      when: never
    - if: '$CI_JOB_NAME =~ /release_resources/'
      when: always
    - when: on_success
```

Machine agnostic templates are left in .gitlab-ci.yml

The remaining `.build_toss_3_x86_64_ib_script` is to be found in the root `.gitlab-ci.yml` because it describes properties for the job shared on all machines:

```
.build_toss_3_x86_64_ib_script:
  script:
    - echo ${ALLOC_NAME}
    - export JOBID=$(squeue -h --name=${ALLOC_NAME} --format=%A)
    - echo ${JOBID}
    - srun $( [[ -n "${JOBID}" ]] && echo "--jobid=${JOBID}" ) -t 15 -N 1 scripts/
↪gitlab/build_and_test.sh
  artifacts:
    reports:
      junit: junit.xml
```

So that, in the end, our job full definition is:

```
clang_9_0_0 (build and test on quartz):
  variables:
    SPEC: "%clang@9.0.0"
  stage: q_build_and_test
  script:
    - echo ${ALLOC_NAME}
    - export JOBID=$(squeue -h --name=${ALLOC_NAME} --format=%A)
    - echo ${JOBID}
    - srun $( [[ -n "${JOBID}" ]] && echo "--jobid=${JOBID}" ) -t 15 -N 1 scripts/
↪gitlab/build_and_test.sh
  artifacts:
    reports:
      junit: junit.xml
  tags:
    - shell
    - quartz
  rules:
    - if: '$CI_COMMIT_BRANCH =~ /_qnone/ || $ON_QUARTZ == "OFF"' #run except if ...
      when: never
```

(continues on next page)

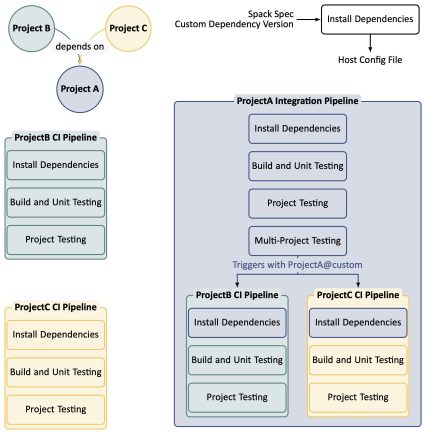
(continued from previous page)

```
    - if: '$CI_JOB_NAME =~ /release_resources/'
      when: always
    - when: on_success
```

1.2.2 Multi-Project Testing Workflow

Multi-Project Testing consists, for example, in testing a new version of a dependency (e.g. Umpire library) in a project that depends on it (CHAI) by triggering the latter CI with the latest version of the former.

This capability is permitted by "pipeline triggers" feature in GitLab, and illustrated here with Umpire/CHAI/RAJA.



GitLab configuration for Multi-Project Testing

```
trigger-chai:
  stage: multi_project
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop" || $MULTI_PROJECT == "ON"' #run only if ...
  variables:
    UPDATE_UMPIRE: develop
  trigger:
    project: radiuss/chai
    branch: develop
    strategy: depend
```

Here, in Umpire, CHAI CI pipeline is triggered when on `develop` branch. CHAI pipeline will know what version of Umpire to use through the environment variable `UPDATE_UMPIRE`.

Note: In CHAI, Umpire is installed with Spack (see Uberenv Guide, CI usage), but Spack can't take a specific commit at the moment as a version specifier, so we are limited to branches declared in the package.

The update mechanism relies on a small change in CHAI CI build script:

```
umpire_version=${UPDATE_UMPIRE:-""}

if [[ -n ${umpire_version} ]]
then
```

(continues on next page)

(continued from previous page)

```
    extra_deps="${extra_deps} ^umpire@${umpire_version}"
fi
```

1.2.3 GitLab CI template example

gitlab-ci.yml

```
##############################################################################
# General GitLab pipelines configurations for supercomputers and Linux clusters
# at Lawrence Livermore National Laboratory (LLNL).
#
# This entire pipeline is LLNL-specific
##############################################################################

# We define the following GitLab pipeline variables:
#
# GIT_SUBMODULE_STRATEGY:
# Tells Gitlab to recursively update the submodules when cloning umpire
#
# ALLOC_NAME:
# On LLNL's quartz, this pipeline creates only one allocation shared among jobs
# in order to save time and resources. This allocation has to be uniquely named
# so that we are able to retrieve it.
variables:
  GIT_SUBMODULE_STRATEGY: recursive
  ALLOC_NAME: ${CI_PROJECT_NAME}_ci_${CI_PIPELINE_ID}

# Normally, stages are blocking in Gitlab. However, using the keyword "needs" we
# can express dependencies between job that break the ordering of stages, in
# favor of a DAG.
# In practice q_*, l_* and b_* stages are independently run and start immediately.
stages:
  - environment
# quartz stages
  - q_allocate_resources
  - q_build_and_test
  - q_release_resources
# lassen stages
  - l_build_and_test

# These are also templates (.name) that define project specific build commands.
# If an allocation exist with the name defined in this pipeline, the job will
# use it (slurm specific).
.build_toss_3_x86_64_ib_script:
  script:
    - echo ${ALLOC_NAME}
    - export JOBID=$(squeue -h --name=${ALLOC_NAME} --format=%A)
    - echo ${JOBID}
    - srun $( [[ -n "${JOBID}" ]] && echo "--jobid=${JOBID}" ) -t 15 -N 1 scripts/
↪gitlab/build_and_test.sh

# Lassen uses a different job scheduler (spectrum lsf) that does not
# allow pre-allocation the same way slurm does.
.build_blueos_3_ppc64le_ib_p9_script:
```

(continues on next page)

(continued from previous page)

```
  script:
    - lalloc 1 -W 15 scripts/gitlab/build_and_test.sh

# "include" allows us to split the configuration in logical blocks
include:
  - local: .gitlab/quartz-templates.yml
  - local: .gitlab/quartz-jobs.yml
  - local: .gitlab/lassen-templates.yml
  - local: .gitlab/lassen-jobs.yml
```

quartz-templates.yml

```
####
# Shared configuration of jobs for quartz
.on_quartz:
  tags:
    - shell
    - quartz
  rules:
    - if: '$CI_COMMIT_BRANCH =~ /_qnone/ || $ON_QUARTZ == "OFF"'
      when: never
    - if: '$CI_JOB_NAME =~ /release_resources/'
      when: always
    - when: on_success

####
# In pre-build phase, allocate a node for builds
allocate_resources (on quartz):
  variables:
    GIT_STRATEGY: none
  extends: .on_quartz
  stage: q_allocate_resources
  script:
    - salloc -N 1 -c 36 -p pdebug -t 20 --no-shell --job-name=${ALLOC_NAME}

####
# In post-build phase, deallocate resources
# Note : make sure this is run even on failure of the build phase
# (see .on_quartz rules)
release_resources (on quartz):
  variables:
    GIT_STRATEGY: none
  extends: .on_quartz
  stage: q_release_resources
  script:
    - export JOBID=$(squeue -h --name=${ALLOC_NAME} --format=%A)
    - ([[ -n "${JOBID}" ]] && scancel ${JOBID})

####
# Generic quartz build job, extending build script
.build_and_test_on_quartz:
  extends: [.build_toss_3_x86_64_ib_script, .on_quartz]
  stage: q_build_and_test
```

quartz-jobs.yml

```
clang_10:
  variables:
    SPEC: "+fortran %clang@10.0.1"
  extends: .build_and_test_on_quartz
```

lassen-templates.yml

```
####
# Shared configuration of jobs for lassen
.on_lassen:
  variables:
  tags:
    - shell
    - lassen
  rules:
    - if: '$CI_COMMIT_BRANCH =~ /_lnone/ || $ON_LASSEN == "OFF"'
      when: never
    - when: on_success

.build_and_test_on_lassen:
  stage: l_build_and_test
  extends: [.build_blueos_3_ppc64le_ib_p9_script, .on_lassen]
  needs: []
```

lassen-jobs.yml

```
ibm_clang_9:
  variables:
    SPEC: "+fortran %clang@9.0.0ibm"
  extends: .build_and_test_on_lassen
```

1.3 Using Spack Pipeline

Spack provides a feature to generate a GitLab pipeline that will build all the specs of a given environment in a GitLab pipeline. This feature is documented by Spack and was originally introduced to run on cloud resource.

We intend to illustrate the use of Spack pipelines on an HPC cluster, and not for deployment, but rather for testing.

1.3.1 Using a unique instance of Spack

One of the first special feature that appears in our situation is that we don't have Spack already on the system, and we don't want to clone it in each job.

Requirements

1. We want to use a single instance of Spack accross all the CI jobs, for performance reasons.

2. We don't want two independent pipelines to use the same instance of Spack, to avoid locks and conflicts on the Spack version used.
3. We don't want to download Spack history again if we already have it.

Implementation

In terms of implementation, we can start with a script to get Spack.

This code is called in CI by a dedicated job:

Note: We define the script outside of the CI. This is a good practice for testing and sharing the CI script outside CI.

But the most critical part is how to share the location of Spack with the child pipeline.

We first create a global variable for the path, made "unique" by using the commit ID:

Then we propagate it to the child pipelines in the trigger job:

The important thing to note here is that we need to change the variable name to pass it to the child pipeline. This has been reported to GitLab.

Managing the GPG key

One of the first blocking point while attempting to share a single Spack instance will be GPG key management.

In the general case each build jobs will register the mirror key, which will result in deadlocks in our case. We can instead register the key ahead of time.

We will produce a GPG key using an instance of Spack, and then create a back-up location, so that we can use it for any spack instance we create in the CI context. This is already detailed in spack mirror/build cache tutorial.

```
$ spack gpg create "My Name" "<my.email@my.domain.com>"
gpg: key 070F30B4CDF253A9 marked as ultimately trusted
gpg: directory '/home/spack/spack/opt/spack/gpg/openpgp-revocs.d' created
gpg: revocation certificate stored as '/home/spack/spack/opt/spack/gpg/openpgp-revocs.
↪d/ABC74A4103675F76561A607E070F30B4CDF253A9.rev'
$ mkdir ~/private_gpg_backup
$ cp ~/spack/opt/spack/gpg/*.gpg ~/private_gpg_backup
$ cp ~/spack/opt/spack/gpg/pubring.* ~/mirror
```

Then we simply need to point the variable `SPACK_GNUPGHOME` to the location where the key is stored. Spack won't try to register the key if this variable is already set.

Only generate one pipeline per stage

Because we are using a single instance of Spack, we want to avoid race conditions that could cause locks.

In practice it forced us to isolate each job that generates a pipeline in a separate stage. That is because of potential locks when concretizing two specs at the same time.

1.3.2 Configuring the environment

There are several things to think of when configuring a Spack environment for CI on a cluster.

Isolate spack configuration

We need to make sure the environment (in particular the configuration files) is isolated. In practice it implies to workaround the user scope of configuration, as we want our pipeline to behave in a deterministic manner whoever triggers it.

The most robust way to do so is to make sure that the environment is the only place where the configuration is defined, and override everything else.

Note: We use `include` to keep things clear and split the config in separate files, but everything could be in a single `spack.yaml` file.

For example, we use the double colon override syntax in the `packages` section:

The same needs to be applied to the `compilers` section.

We also make sure to move any cache or stage directory to the Spack root dir, making them specific to that instance by design:

Note: We do not use the `::` syntax on the `config` section. That is because we assume that it will not be affected as much by the user scope. However, note that we use it on the `build_stage` subsection, since it is a list that would otherwise consist in the merge of all the scopes.

Performance optimization

Several ways of improvement we are exploring:

- At the moment, each pipeline starts with a clone of Spack. Even if we do a shallow clone, this takes between 2 and 8 minutes in our observations.
- The working directory is in the `workspace` filesystem, which is slow. We do not need persistence of our temporary files, so we could renounce to it and work in the node shared memory `/dev/shm`. Our first experiments suggests that this would greatly improve the performances.

Shared configuration files

We are planning to share the configuration files (`compilers.yaml` and `packages.yaml` for the most part) in another open-source repo.

This will help ensure consistency in out testing accross LLNL open-source projects. This is already in use in RAJA, Umpire and CHAI. Projects could still add their own configurations.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search