GitLab runner for HPC systems

In rootless mode, by relying on ENROOT and SLURM.

Overview

Purpose and Features

This set of scripts aims at enabling user-level (no root access required) Continuous Integration on HPC clusters by relying on Gitlab runner's custom executors, ENROOT as a rootless container solution replacement for docker and the SLURM job scheduler when using computing nodes. It also optionally supports Ccache to speed up compilation times on CI jobs. This tool was inspired by the NHR@KIT Cx Project which provides ENROOT and GitLab-runner on their clusters. It is used in production in some of the Ginkgo Software's pipelines.

SLURM usage is optional in this set of scripts as it is considered that many of the simple CI steps, such as compilation, will happen on a login node to optimize computing time and resource sharing. Currently, the script uses non-interactive job submission and waiting loops to ensure the correct completion of the job on the cluster.

A typical use case for this series of scripts is the following:

- build: configure, build, install all the software on the login node.
- test: reuse the previous container to launch tests on a compute node with device access through SLURM.
- benchmark: also reuse the previous container to launch a benchmarking job on a compute node with device access through SLURM, then delete the container.

See the usage example for concrete details.

Dependencies

There are several standard Linux commands used on top of ENROOT and SLURM commands. For some commands, the script can rely on non-standard/GNU-only options. This is for now not optimized.

Always required:

- Gitlab runner (user mode)
- Enroot
- Flock
- Bash
- grep

With SLURM:

- sacct, squeue, scancel, sbatch, srun
- GNU ls, wc, head, tr, cut, awk, ...
- option extglob

Code Structure

The code structure is simple, there are the standard GitLab-runner custom executor scripts:

- `config.sh`: describes the executor;
- `prepare.sh`: prepares the enroot container from a docker image, uses an image cache, optionally reuses existing container instead;
- `run.sh`: either directly runs the GitLab commands on a local enroot container, or submits a job that executes everything in bulk;
- `cleanup.sh`: delete the container if not requested otherwise, cleanup the SLURM job if needed.

The main configuration variables and functions are defined in the following files:

- `include.sh`: contains the main (non slurm) configuration options;
- `slurm_utils.sh`: contains most functions and configurations taking for SLURM functionality.

Configuration Variables

The following variables control some aspects of the script functionality. They can be set as job variables in the script or on the web pages. In the script, they need to be accessed as `${CUSTOM_ENV_<VARIABLE>}`.

Global Options These variables are not SLURM specific and can be used in the default `ENROOT` only mode.

- `CI_JOB_IMAGE` (YAML script `image:` option): a standard docker image for enroot to instantiate a container from; If it is hosted on gitlab (`CI_REGISTRY` is set), it will be accessed via the default token `CI_REGISTRY_PASSWORD`.

- `CI_WS`: a directory with shared data access across all nodes to be used as a workspace.

Optional:

- `USE_NAME`: instead of an automatically generated name, use a specific name for the container (and SLURM job if applicable). This name needs to be unique! When not specified, the name will be `GitLabRunnerEnrootExecutorID${CUSTOM_ENV_CI_JOB_ID}`.
- `NVIDIA_VISIBLE_DEVICES`: a value passed to the enroot container to control NVIDIA device visibility. When no GPU is available or used, `void` should be passed.
- `CCACHE_MAXSIZE`: sets a custom maximum limit to the Ccache directory size.
- `KEEP_CONTAINER`: a non-zero value allows to not delete the container after usage, except if an error occurred.
- `ENROOT_REMAP_ROOT`: a non-zero value allows adds the enroot option –root

Volumes:

- `VOL_NUM`: sets the number of volumes configured to be mounted in the container.
- `VOL_1_SRC`: sets the source directory (on the cluster) for the first volume.
- `VOL_1_DST`: sets the destination directory (in the container) for the first volume.

SLURM Behavior When any of these variables are set, instead of directly running the container on the node where `gitlab-runner` is running, this will submit a job instead. These variables allow to control the SLURM job submission and related behavior.

- `SLURM_PARTITION`: the value of the SLURM `--partition` parameter, e.g., `gpu`.
- `SLURM_EXCLUSIVE`: when non-zero, adds the SLURM `--exclusive` parameter.
- `SLURM_TIME`: the value of the SLURM `--time` parameter, e.g. `0:30:00`.
- `SLURM_GRES`: the value of the SLURM `--gres` parameter.
- `SLURM_ACCOUNT`: the value of the SLURM `--account` parameter.
- `USE_SLURM`: if no other variables are set, setting this enables SLURM mode of execution for this job.

These variables control the SLURM job waiting loop behavior:

- `SLURM_UPDATE_INTERVAL`: the sleeping time between two job status checks.
- `SLURM_PENDING_LIMIT`: the job pending time waiting limit, the default is 12 hours.
- `SLURM_TIME`: when specified, this changes the running time waiting limit to that value, the default is 24 hours.

Installation

The instructions are for a standard Linux system that already supports user mode GitLab and has enroot installed (see dependencies). Also, refer to the NHR@KIT CI user documentation which detail this setup on their systems.

Installing a gitlab-runner

The standard `gitlab-runner install` command can be used. Make sure to select the custom executor, see gitlab runner registration documentation. Here is an example of what a runner configuration can look like, usually found in `~/.gitlab/config.toml`:

```
[[runners]]
  name = "enroot executor"
  url = "https://gitlab.com"
  token = "<token>"
  executor = "custom"
  builds_dir = "/workspace/scratch/my-ci-project/gitlab-runner/builds/"
  cache_dir = "/workspack/scratch/my-ci-project/gitlab-runner/cache/"
  environment = ["CI_WS=/workspace/scratch/my-ci-project",
                 "VOL_1_SRC=/workspace/scratch/my-ci-project/ccache", "VOL_1_DST=/ccache",
                 "VOL_2_SRC=/workspace/scratch/my-ci-project/test_data", "VOL_2_DST=/test_da
                 "NUM_VOL=2", "CCACHE_MAXSIZE=40G"]
  [runners.custom_build_dir]
    enabled = false
  [runners.custom]
    config_exec = "/<path_to>/gitlab-hpc-ci-cb/config.sh"
    prepare_exec = "/<path_to>/gitlab-hpc-ci-cb/prepare.sh"
    run_exec = "/<path_to>/gitlab-hpc-ci-cb/run.sh"
    cleanup_exec = "/<path_to>/gitlab-hpc-ci-cb/cleanup.sh"
```

Enroot and Cluster Setup

On machines using `systemd` and `logind`, enable lingering for your user so that the gitlab-runner daemon can persist when logged off: `loginctl enable-linger ${USER}`. To check if the property is active, use the command: `loginctl show-user $USER --property=Linger`, which should output `Linger=yes`.

As detailed in global options, it is required to set the environment variable `CI_WS` either in the runner configuration or in the script to be used as a workspace for storing enroot containers, caching, and more.

After the new GitLab runner has been configured, lingering is enabled and the other cluster setup steps are finished, start your runner in user mode with the following commands on a `systemd`-based system:

```
# Enable your own gitlab-runner and start it up
systemctl --user enable --now gitlab-runner
```

```
# Check that the gitlab runner is running
systemctl --user status gitlab-runner
```

Volume mounting and Ccache setup

A generic volume mounting interface is provided. This is useful for Ccache support but can be used for other aspects as well. It is configured through multiple environment variables: 1. `VOL_NUM` specifies the number of volumes configured. 2. `VOL_1_SRC` is the volume source (on the cluster), e.g. `${CI_WS}/ccache` 3. `VOL_1_DST` is the volume destination (in the container), e.g. `/ccache`

A full example is available in Installing a gitlab-runner.

Usage Example

Assuming that the code of `default_build` contains the code for compiling your software in the required setting, and `default_test` contains the equivalent of `make test`, the following gitlab-ci YAML configuration will:

- `my_build_job`: build the software on the node running gitlab-runner (no SLURM), keep the container's state for the next job
- `my_test_job`: test the software on a compute node on the `gpu` SLURM partition with one GPU and a time limit of 30 minutes. Then delete the container (no `KEEP_CONTAINER` is set).

Note that this works because both use the same custom name `simple_hpc_ci_job`, which needs to be unique, but shared among the jobs of the same pipeline.

```
stages:
  - build
  - test

my_build_job:
  image: ubuntu:xenial
  stage: build
  <<: *default_build
  variables:
    USE_NAME: "simple_hpc_ci_job"
    KEEP_CONTAINER: "ON"
    NVIDIA_VISIBLE_DEVICES: "void"
  tags:
    - my_enroot_runner

slurm_test_job:
  image: ubuntu:xenial
  stage: test
  <<: *default_test
  variables:
```

```
    USE_NAME: "simple_hpc_ci_job"
    SLURM_PARTITION: "gpu"
    SLURM_EXCLUSIVE: "ON"
    SLURM_GRES: "gpu:1"
    SLURM_TIME: "00:30:00"
  dependencies: [ "my_build_job" ]
  tags:
    - my_enroot_runner
```

`after_script` The `after_script` step is never executed inside a SLURM job, but always directly executed instead. It is assumed that this script is only used for cleanup or similar purpose.

License

Links

- NHR@KIT CI User documentation
- Gitlab runner's custom executors
- Gitlab runner custom executor examples