Flux

Release 0.13.0

Apr 29, 2020

Contents

CHAPTER 1

Quick Start

A quick introduction to Flux and flux-core.

1.1 Building the Code

1.1.1 Spack: Recommended for curious users

Flux maintains an up-to-date package in the spack develop branch. If you're already using spack, just run the following to install flux and all necessary dependencies:

```
$ spack install flux-sched
```

The above command will build and install the latest tagged version of flux-sched and flux-core. To install the latest master branches, use the `@master` version specifier: `spack install flux-sched@master`. If you want Flux to manage and schedule Nvidia GPUs, include the `+cuda` variant: `spack install flux-sched+cuda`. This builds a CUDA-aware version of hwloc.

For instructions on installing spack, see Spack's installation documentation.

1.1.2 Docker: Recommended for quick, single-node deployments

Flux has a continuously updated Docker image available for download on Docker Hub. If you already have docker installed, just run the following to download the latest Flux docker image and start a container from it:

```
$ docker run -ti fluxrm/flux-sched:latest bash
```

Note: Multi-node docker deployments of Flux is still an ongoing area of research. This installation method is recommended for developers and users curious to try single-node instances of Flux.

1.1.3 Manual: Recommended for developers and contributors

Ensure the latest list of requirements are installed. The currrent list of build requirements are detailed here.

Clone current flux-core master:

```
$ git clone https://github.com/flux-framework/flux-core.git
Initialized empty Git repository in /home/fluxuser/flux-core/.git/
$ cd flux-core
```

Build flux-core. In order to build python bindings, ensure you have python-3.6 and python-cffi available in your current environment:

```
$ ./autogen.sh && PYTHON_VERSION=3 ./configure --prefix=$HOME/local
Running aclocal ...
Running libtoolize ...
Running autoheader ...
...
$ make -j 8
...
```

Note: Flux still supports python-2.7, but we recommend that you use python-3.6 or higher as the Python community has stopped maintaining this version as of 2020. The `PYTHON_VERSION` environment variable adds a suffix to the python interpreter executable. Configure would look for python3 in the example above.

Ensure all is right with the world by running the built-in `make check` target:

```
$ make check
Making check in src
...
```

Clone current flux-sched master:

```
$ git clone https://github.com/flux-framework/flux-sched.git
Initialized empty Git repository in /home/fluxuser/flux-sched/.git/
$ cd flux-sched
```

Build flux-sched. By default, flux-sched will attempt to configure against flux-core found in the specified `--prefix` using the same `PYTHON_VERSION`:

```
$ ./autogen.sh && ./configure --prefix=$HOME/local
Running aclocal ...
Running libtoolize ...
Running autoheader ...
...
$ make
...
```

Ensure all is right with the world by running the built-in `make check` target:

```
$ make check
Making check in src
...
```

1.2 Starting a Flux Instance

In order to use Flux, you first must initiate a Flux *instance* or *session*.

A Flux session is composed of a hierarchy of `flux-broker` processes which are launched via any parallel launch utility that supports PMI. For example, `srun`, `mpiexec.hydra`, etc., or locally for testing via the `flux start` command.

Before a Flux instance can be started, keys must be generated to encrypt and authenticate Flux messages. This step is only required for first-time users of flux.

```
$ flux keygen
Saving /home/fluxuser/.flux/curve/client
Saving /home/fluxuser/.flux/curve/client_private
Saving /home/fluxuser/.flux/curve/server
Saving /home/fluxuser/.flux/curve/server_private
$
```

To start a Flux session with 4 brokers on the local node, use `flux start`:

```
$ flux start --size=4
$
```

A flux session can be also be started under Slurm using PMI. To start by using `srun(1)`, simply run the `flux start` command without the `--size` option under a Slurm job. You will likely want to start a single broker process per node:

```
$ srun -N4 -n4 --pty flux start
srun: Job is in held state, pending scheduler release
srun: job 1136410 queued and waiting for resources
srun: job 1136410 has been allocated resources
$
```

After broker wireup is completed, the Flux session starts an "initial program" on rank 0 broker. By default, the initial program is an interactive shell, but an alternate program can be supplied on the `flux start` command line. Once the initial program terminates, the Flux session is considered complete and brokers exit.

To get help on any `flux` subcommand or API program, the `flux help` command may be used. For example, to view the man page for the `flux-hwloc(1)` command, use

```
$ flux help hwloc
```

`flux help` can also be run by itself to see a list of commonly used Flux commands.

1.3 Interacting with a Flux Session

There are several low-level commands of interest to interact with a Flux session. For example, to view the total resources available to the current instance, `flux hwloc info` may be used:

```
$ flux hwloc info
4 Machines, 144 Cores, 144 PUs
```

The size, current rank, comms URIs, logging levels, as well as other instance parameters are termed "attributes" and can be viewed and manipulated with the `lsattr`, `getattr`, and `setattr` commands, for example.

```
$ flux getattr rank
0
$ flux getattr size
4
```

The current log level is also an attribute and can be modified at runtime:

```
$ flux getattr log-level
6
$ flux setattr log-level 4  # Make flux quieter
$ flux getattr log-level
4
```

To see a list of all attributes and their values, use `flux lsattr -v`.

Log messages from each broker are kept in a local ring buffer. When log level has been quieted, recent log messages for the local rank may be dumped via the `flux dmesg` command:

```
$ flux dmesg | tail -4
2016-08-12T17:53:24.073219Z broker.info[0]: insmod cron
2016-08-12T17:53:24.073847Z cron.info[0]: synchronizing cron tasks to event hb
2016-08-12T17:53:24.075824Z broker.info[0]: Run level 1 Exited (rc=0)
2016-08-12T17:53:24.075831Z broker.info[0]: Run level 2 starting
```

Services within a Flux session may be implemented by modules loaded in the `flux-broker` process on one or more ranks of the session. To query and manage broker modules, Flux provides a `flux module` command:

```
$ flux module list
Module                   Size Digest  Idle  S Service
job-exec              1274936 D83AE37    4  S
job-manager           1331496 1F432DD    4  S
kvs-watch             1299400 AA90CE6    4  S
kvs                   1558712 7D8432C    0  S
sched-simple          1241744 AA85006    4  S sched
job-info              1348608 CA590E9    4  S
barrier               1124360 DDA1A3A    4  S
cron                  1202792 1B2DFD1    0  S
connector-local       1110736 5AE480D    0  R
job-ingest            1214040 19306CA    4  S
userdb                1122432 0AA8778    4  S
content-sqlite        1126920 EB0D5E9    4  S content-backing
aggregator            1141184 5E1E0B6    4  S
```

The most basic functionality of these service modules can be tested with the `flux ping` utility, which targets a builtin `*.ping` handler registered by default with each module.

```
flux ping --count=2 kvs
kvs.ping pad=0 seq=0 time=0.648 ms (1F18F!09552!0!EEE45)
kvs.ping pad=0 seq=1 time=0.666 ms (1F18F!09552!0!EEE45)
```

By default the local (or closest) instance of the service is targeted, but a specific rank can be selected with the `--rank` option.

```
$ flux ping --rank=3 --count=2 kvs
3!kvs.ping pad=0 seq=0 time=1.888 ms (CBF78!09552!0!1!3!BBC94)
3!kvs.ping pad=0 seq=1 time=1.792 ms (CBF78!09552!0!1!3!BBC94)
```

The `flux-ping` utility is a good way to test the round-trip latency to any rank within a Flux session.

1.4 Flux KVS

The key-value store (kvs) is a core component of a Flux instance. The `flux kvs` command provides a utility to list and manipulate values of the KVS. For example, hwloc information for the current instance is loaded into the kvs by the `resource-hwloc` module at instance startup. The resource information is available under the kvs key `resource.hwloc`. For example, the count of total Cores available on rank 0 can be obtained from the kvs via:

```
$ flux kvs get resource.hwloc.by_rank
{"[0-3]": {"NUMANode": 2, "Package": 2, "Core": 36, "PU": 36, "cpuset": "0-35"}}
```

See `flux help kvs` for more information.

1.5 Launching Work in a Flux Session

Flux has two methods to launch "remote" tasks and parallel work within a session. The `flux exec` utility is a low-level remote execution framework which depends on as few other services as possible and is used primarily for testing. By default, `flux exec` runs a single copy of the provided `COMMAND` on each rank in a session:

```
$ flux exec flux getattr rank
0
3
2
1
```

Though individual ranks may be targeted:

```
$ flux exec -r 3 flux getattr rank
3
```

The second method for launching and submitting jobs is a Minimal Job Submission Tool named "mini". The "mini" tool consists of a `flux mini` frontend command; `flux job` is another low-level tool that can be used for querying job information.

For a full description of the `flux mini` command, see `flux help mini`.

- Run 4 copies of hostname.

```
$ flux mini run -n4 --label-io hostname
3: quartz15
2: quartz15
1: quartz15
0: quartz15
```

- Run an MPI job (for MPI that supports PMI).

```
$ flux mini run -n128 ./hello
completed MPI_Init in 0.944s.  There are 128 tasks
completed first barrier
completed MPI_Finalize
```

- Run a job and immediately detach. (Since jobs are KVS based, jobs can run completely detached from any "front end" command.)

```
$ flux mini submit -n128 ./hello
4095117099008
```

Here, the allocated ID for the job is immediately echoed to stdout.

- View output of a job.

```
$ flux job attach 4095117099008
completed MPI_Init in 0.932s.  There are 128 tasks
completed first barrier
completed MPI_Finalize
```

- List jobs.

```
$ flux jobs
        JOBID USER        NAME       STATE    NTASKS NNODES  RUNTIME RANKS
1378382512128 fluxuser    sleep      RUN           1      1   5.015s 0
1355649384448 fluxuser    sleep      RUN           1      1   6.368s 0
```

CHAPTER 2

Contributing

The Flux Framework team welcomes all contributors for bug fixes, code improvements, new features, simplifications, documentation, and more. Please do not hesitate to contact us with any questions or concerns.

This guide details how to contribute to Flux Framework projects in a standardized and efficient manner. Our projects follow the Collective Code Construction Contract (C4.1) which describes an optimal collaboration model for open source projects, based on the GitHub fork+pull model.

2.1 Pull Requests

Pull requests are the primary way code changes are incorporated into Flux Framework projects. All projects use the GitHub fork and pull model, in which contributors develop on a branch of their personal fork of the project and create pull requests in order to request those changes be merged into the main repository.

Use a pull request (PR) toward a repository's master branch to propose your contribution, including the specific issue number your PR resolves. If you are planning significant code changes, or have any questions, please open an issue and reference it in your PR. To contribute to a Flux Framework project:

- Fork the project.
- Clone your fork. `git clone git@github.com:[username]/flux-framework/[project].git`
- Create a topic branch to contain your change. `git checkout -b new_feature`
- Create feature or add fix, and add tests if possible.
- Make sure everything still passes `make check`.
- Rebase your commits into meaningfully labeled, easily digestible chunks, ideally without errors.
- Push the branch to your GitHub repo. `git push origin new_feature`
- Create a PR against flux-framework/[project] and describe what your change does and why you think it should be merged. List any outstanding "to do" items.
- Each PR will be subjected to automated tests under travis-ci.org.

Please note that PRs should be rebased onto the master of the target repository, merge commits will be rejected by the `pr-validator` script.

PRs that are a work in progress are welcomed, but should have *WIP:* prefix or *work-in-progress* label to avoid automerging by mergify.io.

2.2 Developer Guidelines

- Review other issues and PRs to ensure you are not duplicating effort.
- When possible, submissions should include relevant unit and system tests. Coverage reports will be autogenerated for all submitted PRs.
- Update any appropriate documentation or open an issue to do so.
- Adhere to the coding style guide. The components of Flux written in C follow the Kernighan & Ritchie coding style, with exceptions enumerated in RFC 7. The components of Flux written in Python follow the black code style.
- Commit etiquette:
  - Avoid merge commits.
  - Separate work so that each commit fixes one problem. For example, keep code cleanup and refactoring in separate commits from new features, fixes, or functionality changes.
  - Use `subsystem:` prefix in commit titles. The `subsystem` name should be the minimal text needed to reference what has changed, without wasting too much of the commit subject character limit. For example, `broker:`, `kvs:` or `doc:` are good names, `lib:` or `job:` are not.
  - Reference related issues in the commit body.
  - Each commit should have a message with title and body as described in C4.1 (e.g., imperative phrasing, wrap lines at 72 characters).

For more details about C4.1, including commit requirements, see RFC 1.

CHAPTER 3

CORAL Systems

The LLNL and ORNL CORAL systems Lassen, Sierra, and Summit are pre-exascale supercomputers built by IBM. They run a specialized software stack that requires additional components to integrate properly with Flux. These components are provided as Lmod modules on all three systems.

To setup your environment to use these modules on the LLNL systems Lassen and Sierra, run:

```
module use /usr/global/tools/flux/blueos_3_ppc64le_ib/modulefiles
```

If you are using the ORNL system Summit, run:

```
module use /sw/summit/modulefiles/ums/gen007flux/Core
```

3.1 Launching Flux

Launching Flux on CORAL systems requires a shim layer to provide PMI on top of the PMIx interface provided by the CORAL system launcher jsrun. PMI is a common interface for bootstrapping parallel applications like MPI, SHMEM, and Flux. To load this module, run:

```
module load pmi-shim
```

We also suggest that you launch Flux using jsrun with the following arguments:

```
PMIX_MCA_gds="^ds12,ds21" jsrun -a 1 -c ALL_CPUS -g ALL_GPUS --bind=none -n ${NUM_
↪NODES} flux start
```

The `PMIX_MCA_gds` environment variable works around a bug in OpenPMIx that causes a hang when using the PMI compatibility shim. The `${NUM_NODES}` variable is the number of nodes that you want to launch the Flux instance across. The remaining arguments ensure that all on-node resources are available to Flux for scheduling.

3.2 Launching Spectrum MPI within Flux

If you want to run MPI applications compiled with Spectrum MPI under Flux, then two steps are required. First, load our copy of Spectrum MPI that includes a backported fix from OpenMPI:

```
module load spectrum-mpi/2019.06.24-flux
```

Note: Future releases of Spectrum MPI will include this patch, making loading this module unnecessary.

Second, when you run a Spectrum MPI binary under flux, enable Flux's Spectrum MPI plugin. From the CLI, this looks like:

```
flux mini run -o mpi=spectrum my_mpi_binary
```

From the Python API, this looks like:

```
#!/usr/bin/env python3

import os
import flux
from flux import job

fh = flux.Flux()

jobspec = job.JobspecV1.from_command(['my_mpi_binary'])
jobspec.environment = dict(os.environ)
jobspec.setattr_shell_option('mpi', 'spectrum')

jobid = job.submit(fh, jobspec)
print(jobid)
```

3.3 Scheduling GPUs

On all systems, Flux relies on hwloc to auto-detect the on-node resources available for scheduling. The hwloc that Flux is linked against must be configured with `--enable-cuda` for Flux to be able to detect Nvidia GPUs.

You can test to see if your system default hwloc is CUDA-enabled with:

```
lstopo | grep CoProc
```

If no output is produced, then your hwloc is not CUDA-enabled.

If running on an LLNL CORAL system, you can load a CUDA-enabled hwloc with:

```
module load hwloc/1.11.10-cuda
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search