Recent Advances in ReFrame

8th EasyBuild User Meeting

April 24, 2023

Vasileios Karakasis (NVIDIA) and Theofilos Manitaras (CSCS)

Summary

1. First part (Vasileios)
    a. Community update
    b. Overview of ReFrame 4.0 changes
    c. Review of some of the less well known pre 4.0 features, that are quite useful

2. Second part (Theofilos)
    a. Overview of the programmable configuration in ReFrame
    b. Using programmable configuration for container-based testing
    c. Using programmable configuration for user-environment-based testing

ReFrame

ReFrame is a powerful framework that enables system testing and performance testing as code with unique HPC features.

- Composable tests written in Python allowing the creation of reusable test libraries
- Multi-dimensional test parameterisation
- Support for test fixtures
- Parallel execution of tests
- Programmable configuration
- Support for multiple HPC schedulers, modules systems, build systems and container runtimes
- Integration with Elastic and Graylog for feeding directly performance data from tests
- CI integration through Gitlab child pipelines

ReFrame community

- Documentation: https://reframe-hpc.readthedocs.io
  - 300–400 unique readers monthly from all over the world
- Slack workspace (more than 230 members):
  - Join us through this link.
- Github
  - ReFrame HPC community group: https://github.com/reframe-hpc
    - Collection of public forks of site test repositories
  - 45 contributors since the beginning
  - Backlog: https://github.com/orgs/reframe-hpc/projects/1
  - Code: https://github.com/reframe-hpc/reframe
    - Give it a ⭐ !
- PyPI: https://pypi.org/project/ReFrame-HPC/
  - More than 8K downloads/month according to pepy.tech

New development workflow since 4.0

- We introduced a `develop` branch
  - New features go to this branch
  - The `/latest` docs point to this branch
  - This is the **default** Github branch
- The `master` branch remains as the **release** branch
  - All releases are made from `master`
  - Bug fixes, documentation updates, minor enhancements target this branch directly
  - Patch-level releases every one or two weeks and are merged into **`develop`**
  - **`develop`** will be merged into **`master`** just before the next minor or major release
- Pros
  - Quick release of bug fixes on top of stable releases
  - We can follow more accurately the semantic versioning scheme
- Cons
  - Periodical synch'ing of `develop` and `master`
  - Might sometimes be confusing which branch a PR should target

Check also the updated contribution guide: https://github.com/reframe-hpc/reframe/wiki/contributing-to-reframe

Changes in ReFrame 4 – Dropped features

All features deprecated in 3.x versions are dropped:

- **`@parameterized_test`** is replaced by the **`parameter`** builtin
- The test's **`name`** is now read-only
- The various test method decorators are only accessible through their builtin names (e.g., **`@run_after`** instead of **`@rfm.run_after`**)
- **`--force-local`**, **`--strict`** and **`--ignore-check-conflicts`** options are dropped
- The **`schedulers`** configuration section is replaced by a **`sched_options`** section inside each partition definition.
  - NOTE: This is broken in 4.0 and fixed in 4.1
- Test's **`variables`** attribute is deprecated over the new **`env_vars`**.

More here: https://reframe-hpc.readthedocs.io/en/stable/whats_new_40.html#dropped-features-and-deprecations

Changes in ReFrame 4 – New features

Configuration can be split in multiple files

- Scoping was already a feature through the use of **`target_systems`**
- Now scopes or parts of the configuration can be split in multiple files
- No need to maintain huge configuration files and repeat the builtin config; the configuration file contains only the information it needs to!
  - In this example, no need to redefine the **`generic`** system and **`builtin`** environment; they are still valid
  - No need to redefine logging config or any other section.
- **`-C`** option can now be chained

```
site_configuration = {
    'systems': [
        {
            'name': 'tresa',
            'descr': 'My Mac',
            'hostnames': ['tresa'],
            'modules_system': 'nomod',
            'partitions': [
                {
                    'name': 'default',
                    'scheduler': 'local',
                    'launcher': 'local',
                    'environs': ['gnu'],
                }
            ]
        }
    ],
    'environments': [
        {
            'name': 'clang',
            'cc': 'clang',
            'cxx': 'clang++',
            'ftn': '',
            'target_systems': ['tresa']
        },
    ]
}
```

Changes in ReFrame 4 – New features (cont'd)

- The filelog log handler for file-based performance logging has been fundamentally revised
  - Default output is CSV so that it can be easily post-processed
  - A header line is printed in every file
  - If the logged fields change, a new log file is created with an updated header and the old is backed up
- Custom parallel launchers can be directly defined in the configuration file
  - No need extending the framework!
  - … but DO extend it and submit a PR if others can benefit from it!
- New backends
  - Apptainer container platform
  - Scheduler backend for the Flux Framework

Changes in ReFrame 4 – New features (cont'd)

New test naming scheme

- Informational and human readable
- Tests can be selected by name, by hash or by variant using the **`-n`** option
  - **`-n '^osu_.*'`**
  - **`-n /03d6f48f`**
  - **`-n osu_allreduce_test@3`**

Test or fixture name *Test parameters* *Test hash* *Test case info*

```
- osu_allreduce_test %mpi_tasks=16 /03d6f48f @generic:default+builtin
    ^build_osu_benchmarks ~generic:default+builtin 'osu_binaries /5cf701b0 @generic:default+builtin
      ^fetch_osu_benchmarks ~generic 'osu_benchmarks /9fc7952e @generic:default+builtin
```

Fixture scope *Fixture variable name*

Changes in ReFrame 4 – New features (cont'd)

- New **`--dry-run`** option (since 4.1)
  - Generates all scripts that will be executed and validates as much of the test as possible
  - Tests can also check if in dry-run mode and adapt by calling **`is_dry_run()`**.
- Support for custom formatting of JSON records sent to Elastic (since 4.1)
  - Set the callable **`json_formatter`** in the **`httpjson`** perflog handler
  - Useful to meet requirements of remote schemas and other constraints
- New **`--reruns`** and **`--duration`** options for stress testing (since 4.2)
  - Repeatedly run the same test session until a number of runs is reached or a timeout expires
  - Results and failure statistics will be reported from all runs
  - Use with care! 😁
    - **`reframe -n gpu_burn_check --distribute=all --duration=24h`**

Old but gold – System/Environment features

Extended syntax for **`valid_systems`** and **`valid_prog_environs`** (since 3.11)

- Tests are no more bound to specific system names and/or environment names
- The test can list the features or the properties of a system and/or environment that is valid or not valid for.

Test syntax

```
# AND features
valid_systems = ['+gpu +ib']
# OR features
valid_systems = ['+gpu', '+ib']
# NOT features
valid_prog_environs = ['-cuda']
# Select extras
valid_prog_environs = ['%mpi_kind=mpich']
```

Example config

```
'partitions': [
    {
        'name': 'mypart',
        'environs': ['myenv', ...],
        'features': ['gpu', 'ib'],
    },
    ...
],
'environments': [
    {
        'name': 'myenv',
        'features': ['cuda', 'mpi'],
        'extras': {'mpi_kind': 'mpich'}
    },
    ...
]
```

More in Theo's part

Old but gold – Command-line options

- The **`-S`** or **`--setvar`** option (since 3.8 with subsequent refinements)
  - Sets test variables from the command-line
  - Allows also to set variables in nested fixtures
  - Very useful for running test interactively and for experimentation
- Clone and distribute tests all over the cluster (since 3.12)
  - **`--repeat=N`**: repeat selected tests **`N`** times
  - **`--distribute[=STATE]`**: run the selected tests on every node in **`STATE`**
    - Can be combined with the **`-J`** option as well:
    - **`reframe -J reservation=foo --distribute=all --repeat=10 -n my_stress_test -r`**
- Generate Gitlab CI child pipelines running the selected tests:
  - **`--ci-generate`** (since 3.4.1)
  - Control the CI pipeline from within the test using **`ci_extras`** (since 4.2)

E.g.: `ci_extras = {'gitlab': {'only': {'refs': ['merge_requests']}}}`

Old but gold – Execution modes

- Named collections of command line options defined in the configuration file and selectable with the **`--mode`** command line option (since 2.5):
  - Treat reframe execution as a black box
  - "Record" a test experiment for future (especially useful in combination with variables and the **`-S`** option)
  - Command-line options are combined with those implicitly passed by the mode

```
reframe --mode=ping_perf -r
reframe --mode=ping_perf -S clients=10 -r
reframe --mode=ping_perf -S foo=bar -r
reframe --mode=ping_perf --exec-policy=async -r
```

Example config

```
'modes': [
    {
        'name': 'ping_perf',
        'options': [
            '-c tests/ping.py',
            '-S clients=1',
            '-S interval=100',
            '-n ping_test',
            '--exec-order=uid',
            '--performance-report',
            '--exec-policy=serial',
            '--keep-stage-files'
        ]
    }
]
```

Old but gold – Programmable configuration

ReFrame's configuration file is essentially a Python module

- You can dynamically generate system/environment entries
  - Useful in cloud environments where the default hostname-based system entry auto-detection is not helpful
  - Useful for generating environments specs based on runtime metadata (more from Theo)
- Site-specific fine-tuning
  - Custom parallel launchers (since 4.0)
  - Log record formatting for sending to Elastic (since 4.1)

Old but gold – Programmable configuration (cont'd)

Custom launcher

```
from reframe.core.backends import register_launcher
from reframe.core.launchers import JobLaucher


@register_launcher('slrun')
class MySmartLauncher(JobLauncher):
    def command(self, job):
        return ['slrun', '-n', job.num_tasks, ...]

site_configuration = {
    'systems': [
        {
            'name': 'my_system',
            'partitions': [
                {
                    'name': 'my_partition',
                    'launcher': 'slrun',
                    ...
                }
            ]
        }
    ]
}
```

Custom record formatter

```
def prepend_prefix(record, extras, ignore_keys):
    json_record = {}
    for k, v in record.__dict__.items():
        if not k.startswith('_') and
           k not in ignore_keys:
            json_record[f'my_{k}'] = v

    return json.dumps([json_record])


site_configuration = {
    'logging': [{
        'handlers_perflog': [{
            'type': 'httpjson',
            'url': 'http://elastic_server/',
            'level': 'info',
            'json_formatter': prepend_prefix
        }]
    }]
}
```

Old but gold – Dynamic test generation

The **`make_test()`** API call allows to create tests programmatically (since 3.10)

```python
import reframe.core.builtins as builtins
from reframe.core.meta import make_test


def set_message(obj):
    obj.executable_opts = [obj.message]

def validate(obj):
    return sn.assert_found(obj.message, obj.stdout)

hello_cls = make_test(
    'HelloTest', (rfm.RunOnlyRegressionTest,),
    {
        'valid_systems': ['*'],
        'valid_prog_environs': ['*'],
        'executable': 'echo',
        'message': builtins.variable(str)
    },
    methods=[
        builtins.run_before('run')(set_message),
        builtins.sanity_function(validate)
    ]
)
```

```python
class HelloTest(rfm.RunOnlyRegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    executable = 'echo'
    message = variable(str)

    @run_before('run')
    def set_message(self):
        self.executable_opts = [self.message]

    @sanity_function
    def validate(self):
        return sn.assert_found(self.message, self.stdout)

hello_cls = HelloTest
```

This is what the **`--distribute`** and **`--repeat`** options leverage internally

Domain-specific test generation using **`make_test`**

Example: Generate a series of STREAM benchmark workflows using a domain-specific spec file

Domain spec file

```
stream_workflows:
  - elem_type: 'float'
    array_size: 16777216
    num_iters: 10
    num_cpus_per_task: 4
  - elem_type: 'double'
    array_size: 1048576
    num_iters: 100
    num_cpus_per_task: 1
  - elem_type: 'double'
    array_size: 16777216
    num_iters: 10
    thread_scaling: [1, 2, 4, 8]
```

Generated tests

```
- stream_test_2 %num_threads=8 %stream_binaries.elem_type=double
%stream_binaries.array_size=16777216 %stream_binaries.num_iters=10
/7b20a90a
    ^stream_build %elem_type=double %array_size=16777216
%num_iters=10 ~tresa:default+gnu 'stream_binaries /1dd920e5
- stream_test_2 %num_threads=4 %stream_binaries.elem_type=double
%stream_binaries.array_size=16777216 %stream_binaries.num_iters=10
/7cbd26d7
    ^stream_build %elem_type=double %array_size=16777216
%num_iters=10 ~tresa:default+gnu 'stream_binaries /1dd920e5
- stream_test_2 %num_threads=2 %stream_binaries.elem_type=double
%stream_binaries.array_size=16777216 %stream_binaries.num_iters=10
/797fb1ed
    ^stream_build %elem_type=double %array_size=16777216
%num_iters=10 ~tresa:default+gnu 'stream_binaries /1dd920e5
<...>
Found 6 check(s)
```

Domain-specific test generation using **`make_test`**

The idea

- Everything happens in a normal test file
- The spec file is passed in an environment variable
- The test file reads the spec, generates the tests using **`make_test`** and registers them with the **`simple_test`** decorator.

Full code at https://github.com/reframe-hpc/reframe/pull/2866.

A standard test file

```python
def load_specs():
    spec_file = os.getenv('STREAM_SPEC_FILE')
    with open(spec_file) as fp:
        return yaml.safe_load(fp)

def generate_tests(specs):
    tests = []
    for i, spec in enumerate(specs['stream_workflows']):
        test_body = {}
        thread_scaling = spec.pop('thread_scaling', None)
        test_body = {
            'stream_binaries': builtins.fixture(
                stream.stream_build, scope='environment', variables=spec)
        }
        methods = []
        if thread_scaling:
            def _set_num_threads(test):
                test.num_cpus_per_task = test.num_threads

            test_body['num_threads'] = builtins.parameter(thread_scaling)
            methods.append(builtins.run_after('init')(_set_num_threads))

        tests.append(make_test(f'stream_test_{i}',
                               (stream.stream_test,), test_body, methods))
    return tests

# Register the tests with the framework
for t in generate_tests(load_specs()):
    rfm.simple_test(t)
```

Future outlook

- Improve reporting and post processing of reports
  - Search and compare easily with past reports
- Generalise the system entry auto-detection method, so that it becomes easier to integrate with cloud environments
- Allow test parameterisation from the command line
  - Re-parameterise a test based on an existing parameterise
  - Parameterise a test based on an existing variable
- Generalise test filtering
  - E.g., based on variable values

We are limited in bandwidth but we are more than happy to accept your contributions!