ReFrame: A regression framework for checking the health of large HPC systems

CUG 2017 conference, Redmond, WA, USA
V. Karakasis, V. H. Rusu, A. Jocksch, J.-G. Piccinali, G. Peretti-Pezzi
May 11, 2017
Outline

- Regression testing in HPC
- What is ReFrame?
- Writing a test in ReFrame
- CSCS use case
Regression testing of HPC systems
Why is it so important?

- Ensures quality of service
- Reduces downtime
- Early detection of problems
Regression testing of HPC systems
But it’s a painful story

- In-house custom solutions per center
- Non portable monolithic regression tests
  - Tightly coupled to the system configuration and programming env.
- Large maintenance overhead
  - Replicated code of the system interaction details
  - Test’s logic is lost in unrelated lower level details
Regression testing of HPC systems
But it’s a painful story

- In-house custom solutions per center
- Non portable monolithic regression tests
  - Tightly coupled to the system configuration and programming env.
- Large maintenance overhead
  - Replicated code of the system interaction details
  - Test’s logic is lost in unrelated lower level details

No one wants to implement a new regression test!
What is ReFrame?

A new regression framework that

- allows writing portable HPC regression tests in Python,
- abstracts away the system interaction details,
- lets users focus solely on the logic of their test.

https://github.com/eth-cscs/reframe
Design goals

- Productivity
- Portability
- Ease of use
- Robustness
ReFrame’s architecture

Regression test API

System abstractions

Job schedulers

Environment abstractions

Job launchers

Shell script generators

Environment loaders

Operating System

Pluggable backends
The regression test pipeline

A series of well defined phases that each regression test goes through

- Pick next test
- Supports system? NO
- Supports environment? YES
- Setup test
- Compile test
- Run test
- Check sanity
- Check performance
- Cleanup test resources
- YES
import os
from reframe.core.pipeline import RegressionTest

class HelloWorldTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('hello_world', os.path.dirname(__file__), **kwargs)
        self.descr = 'Hello World C Test'
        self.sourcepath = 'hello.c'
        self.valid_systems = ['daint:gpu', 'daint:mc', 'daint:login']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu']
        self.sanity_patterns = {'-': {'Hello, World!': []}}

    def _get_checks(**kwargs):
        return [HelloWorldTest(**kwargs)]
Writing regression tests in ReFrame
A “Hello, World!” example

A regression test needs not to care about

- how access to system partitions is gained,
- how programming environments are switched,
- how its environment is set up,
- how a sanity/performance pattern is looked up in its output,
- how a job script is generated and submitted and if it’s needed at all.
Writing regression tests in ReFrame
Specifying the test’s environment

```python
def __init__(self, **kwargs):
    ...  
    self.modules = [ 'cuda-toolkit', 'cray-libsci_acc' ]
    self.variables = {
        'CRAY_CUDA_MPS' : '1',
        'OMP_NUM_THREADS' : '16'
    }
```

- Modules will be loaded and environment variables will be set during the test’s setup phase
- Corresponding instructions will be emitted in the generated run script
  - Possible module conflicts are handled automatically
Writing regression tests in ReFrame
Differentiating per system

- Different configurations to be tested, workarounds etc.
Writing regression tests in ReFrame
Differentiating per system

- Different configurations to be tested, workarounds etc.

```python
def __init__(self, **kwargs):
    ...
    if self.current_system.name == 'dom':
        self.num_tasks = 72
        # workaround for Dom due to bug #XXX
        self.modules += ['foo']
    else:
        self.num_tasks = 192
```

Differentiation per system partition must be done inside the `setup()` method
Writing regression tests in ReFrame

Customizing compilation – Example: OpenMP compilation flags

```python
def __init__(self, **kwargs):
    ...
    # user-defined member variable
    self.prgenv_flags = {
        'PrgEnv-cray' : '-homp',
        'PrgEnv-gnu'  : '-fopenmp',
        'PrgEnv-intel': '-openmp',
        'PrgEnv-pgi'  : '-mp'
    }

def compile(self):
    flag = self.prgenv_flags[self.current_environ.name]
    self.current_environ.cflags = flag
    super().compile()
```

– Also support for running pre- and post-compilation commands
Writing regression tests in ReFrame
Sanity and performance checking

- Regex pattern matching
- Associate callback actions to patterns matched
- Automatic reference value resolution for performance tests
- Stateful parsing support
  - Support for common aggregate operations (min, max, sum, average)
- Search for patterns in multiple files
Writing regression tests in ReFrame
Sanity and performance checking

- Simple grep-like matching:

```python
def __init__(self, **kwargs):
    ...
    self.sanity_patterns = {'-': {'Hello, World!': []}}
```
Writing regression tests in ReFrame
Sanity and performance checking

- Simple grep-like matching:

```python
def __init__(self, **kwargs):
    ...
    self.sanity_patterns = { '‐': {'Hello, World!': []} }
```

- Pattern matching with associated action callbacks:

```python
def __init__(self, **kwargs):
    ...
    self.sanity_patterns = {
        '‐': {
            'final result:s+(?P<res>\d+\.\d*)': [
                ('res', float, lambda value, **kwargs: \
                    standard_threshold(value, (1., -1e-5, 1e-5)))
            ],
        }
    }
```
Writing regression tests in ReFrame
Sanity and performance checking

def __init__(self, **kwargs):
    ...
    self.perf_patterns = {
        '-': {
            'long_pattern (?P<days_ns>\S+) days/ns' : [
                ('days_ns', float, standard_threshold)
            ]
        }
    }
    self.reference = {
        'daint:gpu' : { 'days_ns' : (0.71, None, 0.10) },
        'daint:mc' : { 'days_ns' : (0.90, None, 0.10) } }

For each matched tag standard_threshold(val, ref) will be called.

- val is the value of the matched tag
- ref is looked up in self.reference
Writing regression tests in ReFrame
Sanity and performance checking

```python
def __init__(self, **kwargs):
    ...
    self.perf_patterns = {
        '-' : {
            'long_pattern (?P<days_ns>\S+) days/ns' : [
                ('days_ns', float, standard_threshold)
            ]
        }
    }
    self.reference = {
        'daint:gpu' : { 'days_ns' : (0.71, None, 0.10) },
        'daint:mc' : { 'days_ns' : (0.90, None, 0.10) } }
```

For each matched tag `standard_threshold(float(val), ref)` will be called.

- `val` is the value of the matched tag
- `ref` is looked up in `self.reference`
Writing regression tests in ReFrame
Sanity and performance checking – Stateful parsing

“The average performance of the first 100 steps must be within 10% of the reference value for this system.”
Writing regression tests in ReFrame
Sanity and performance checking – Stateful parsing

“The average performance of the first 100 steps must be within 10% of the reference value for this system.”

- ReFrame’s action callbacks come in very handy in such situations
  - Create an object holding the desired state and update it with every match
- ReFrame supports also eof callback actions, allowing to take a decision after processing the whole output
Writing regression tests in ReFrame
Sanity and performance checking – Stateful parsing

“The average performance of the first 100 steps must be within 10% of the reference value for this system.”

- ReFrame’s action callbacks come in very handy in such situations
  - Create an object holding the desired state and update it with every match
- ReFrame supports also eof callback actions, allowing to take a decision after processing the whole output
- Use ReFrame’s provided parsers that cover the most common cases
Writing regression tests in ReFrame

Organizing the regression tests

```
mychecks/
  compile/
    helloworld/
      helloworld.py
  src/  # <- source files are resolved relative to this directory
    hello.c
runonly/
  app/
  src/  # <- test resources files can be put simply here
    input.txt
    apptest.py
```

- Default check path in `<reframe-install-prefix>/checks/`
- Resources directory can also be customized per test
ReFrame’s front-end

Configuring for a new site

- Systems
  - Hostname identification patterns
  - ReFrame’s stage and output directories

- System logical partitions
  - Job scheduler
  - Environment to always load on that partition
  - Scheduler options enabling access to that partition
  - List of programming environments to test

- Programming environments
  - Modules
  - Environment variables
  - Compilers and default flags
ReFrame’s front-end
Configuring for a new site – Piz Daint example

'systems' : {
'daint' : {
'hostnames' : [ 'daint', 'daint\d+' ],
'partitions' : {
'login' : {
'scheduler' : 'local',
'environ' : [ 'PrgEnv-cray', 'PrgEnv-gnu',
               'PrgEnv-intel', 'PrgEnv-mpi' ],
'descr' : 'Login nodes'
},
'gpu' : {
'scheduler' : 'nativeslurm',
'module' : [ 'daint-gpu' ],
'access' : [ '--constraint=gpu' ],
'environ' : [ 'PrgEnv-cray', 'PrgEnv-gnu',
             'PrgEnv-intel', 'PrgEnv-mpi' ],
'descr' : 'Hybrid nodes (Haswell/P100)'},
'}}}
ReFrame’s front-end
Configuring for a new site – Piz Daint example (cont’d)

'environments' : {
    '*': {
        'PrgEnv-gnu': {
            'type': 'ProgEnvironment',
            'modules': [ 'PrgEnv-gnu' ],
        },
        ...}
    }

'kesch': {  # PrgEnv-gnu redefinition for Piz Kesch
    'PrgEnv-gnu': { 
        'type': 'ProgEnvironment',
        'modules': [ 'PrgEnv-gnu' ],
        'cc': 'mpicc',
        'cxx': 'mpicxx',
        'ftn': 'mpif90',
    }
}
}
ReFrame’s front-end
Command-line interface

ReFrame goes through three phases when invoked:
- Discovery and loading of regression tests
- Selection/filtering of the loaded tests
  - By name, programming environment, tags
- Action on the final set of tests
  - Listing or execution

In case of a test failure, test’s files are left intact in its stage directory:
- User can inspect and try to manually reproduce the error
ReFrame’s front-end
Command-line interface

ReFrame goes through three phases when invoked:

- Discovery and loading of regression tests
- Selection/filtering of the loaded tests
  - By name, programming environment, tags
- Action on the final set of tests
  - Listing or execution

In case of a test failure, test’s files are left intact in its stage directory:

- User can inspect and try to manually reproduce the error
The CSCS use case

Sanity and performance checking of Piz Daint

- **Production test suite**
  - Wide variety of tests running daily overnight
  - Testing hybrid and multicore system partitions as well as login nodes
  - 157 tests run, 437 test cases in total

- **Maintenance test suite**
  - Run before and after each maintenance session
  - Slurm functionality, I/O of core filesystems, performance of critical apps
  - \( \leq 30 \text{ min} \)
The CSCS use case
Comparison with our old shell script based solution

<table>
<thead>
<tr>
<th>Component</th>
<th>Old framework</th>
<th>ReFrame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>N/A</td>
<td>3660 loc</td>
</tr>
<tr>
<td>Front-end</td>
<td>1038 loc</td>
<td>958 loc</td>
</tr>
<tr>
<td>Regression tests</td>
<td>14635 loc</td>
<td>2985 loc</td>
</tr>
</tbody>
</table>
## The CSCS use case

Comparison with our old shell script based solution

<table>
<thead>
<tr>
<th>Component</th>
<th>Old framework</th>
<th>ReFrame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>N/A</td>
<td>3660 loc</td>
</tr>
<tr>
<td>Front-end</td>
<td>1038 loc</td>
<td>958 loc</td>
</tr>
<tr>
<td>Regression tests</td>
<td>14635 loc</td>
<td><strong>2985 loc</strong></td>
</tr>
<tr>
<td>Avg. regression file size</td>
<td>179 loc</td>
<td>93 loc</td>
</tr>
<tr>
<td>Avg. regression test size</td>
<td>179 loc</td>
<td><strong>25 loc</strong></td>
</tr>
</tbody>
</table>

Almost $5 \times$ reduction of the total amount of regression test code!
Conclusions and future directions

ReFrame makes writing regression tests for HPC systems an easy task!

- Actively developed
- More teams inside CSCS have started to adopt it in their projects
- Publicly available at https://github.com/eth-cscs/reframe
Conclusions and future directions

*ReFrame makes writing regression tests for HPC systems an easy task!*

- Actively developed
- More teams inside CSCS have started to adopt it in their projects
- Publicly available at https://github.com/eth-cscs/reframe

**High-priority items from our backlog**

- Proper logging
- Backend for the PBS scheduler
- Asynchronous execution of regression tests

*Try it out, give us some feedback!*
Thank you for your attention