developer.skatelescope.org Documentation

SKA SDP Developers

Aug 24, 2023

HOME

This documentation contains introduction to SKA SDP benchmark tests, brief overview of ReFrame and Spack which are building blocks of this repository, installation instructions and description of different tests provided.

CHAPTER

ONE

GETTING STARTED

1.1 Context

This repository contains the SDP benchmark tests for various levels of benchmarking. ReFrame is used as a framework to package these benchmarks to perform automated tests on various HPC systems. It is designed to easily compare results from different systems, system configurations and/or environments. There are three levels of benchmarks defined in the package namely,

- **Level 0**: Kernel benchmarks that characterise various components of the system like FLOPS performance, memory bandwidth, network performance, *etc*.
- **Level 1**: These are representative pieces of real workflows that can be used to characterise the workload of the pipelines. These include processing functions of radio astronomy pipelines like FFTs, gridding, prototype codes, *etc*.
- **Level 2**: These will be the entire pipelines or workflows of the radio-astronomy softwares like WSClean, RASCIL, ASKAP, *etc*.

Note: For level 1 and 2 benchmarks, runtime configurations should be chosen in such a way to model the intended computational workload. Whereas level 0 benchmarks can be used to characterise various existing HPC systems in terms of raw performance.

Some part of this work has been inspired from hpc-tests work from StackHPC who used ReFrame to build performance tests for HPC platforms.

1.2 Currently supported benchmarks

1.2.1 Level 0

- Sample `numpy` operations benchmark is included in the tests.
- Babel Stream
- GPUDirect RDMA (GDR) test.
- HPCG
- HPL
- IMB
- IOR

- NCCL tests
- STREAM
- numpy / cupy FFTs
- Part of HIPPO Team's Funclib

1.2.2 Level 1

- CUDA NIFTY gridder
- Image Domain Gridder (IDG) Test
- Imaging IO Test

1.2.3 Level 2

- RASCIL

All the tests are defined in a portable fashion in `reframe_<application>.py` file in each application directory. Results are compared and plotted using Juypter notebooks, with a `<application>.ipynb` file in each application directory.

1.3 Software stack

The benchmark tests can be run using either platform provided packages or it is possible to deploy our own software stack in the user space using Spack. Moreover, ReFrame gives us a framework to test both platform provided and user deployed software stacks using the notion of partitions. There is no recommendation when it comes to which software stack to use and it depends on the final objective. For example, if we want to compare two different types of architectures, it is advisable to deploy the same software stack on both platforms and then compare the results. More details on how to use Spack and build packages are presented in *Installing packages*.

CHAPTER

TWO

OVERVIEW OF REFRAME

2.1 Introduction

Tests in ReFrame are simple Python classes that specify the basic parameters of the test. These define a generic test, independent of system details like scheduler, MPI libraries, launcher *etc*. These aspects are defined for the system(s) under test in the reframe configuration file `reframe_config.py`. Once the target system is configured, the tests can be run using the Python scripts inside each application folder. This way the logic of the test is abstracted away from the system under test. As we want to test new systems, we need to update the `reframe_config.py` for that system and use the same Python scripts to run benchmark tests. More details on ReFrame can be found in the documentation. In the following sections, a brief overview is provided on how to configure the ReFrame for different system(s) under test. However, this documentation is not meant to be an extensive overview of ReFrame functionalities. The reader is advised to check the ReFrame documentation for more detailed overview.

2.2 ReFrame configuration

Configuration of the different system(s) under test is the vital part of ReFrame workflow. Once the configuration is done properly, running tests will become as simple as running python scripts from CLI. The main parts of the ReFrame configuration are system and partition configuration and defining environments. These configuration aspects are defined in the so-called `reframe_config.py` file that resides in the root of the repository. A typical configuration file looks like this. However, as number of systems increase in the configuration file, it becomes very long and not so easy to read and edit. Hence, in the current repository, the configuration files are split into different systems and stored in `config` folder in the root of the repository. The main configuration file is assembled by importing the individual configurations of each system. Adding new systems or editing existing systems to the configuration will be more easier by using this approach.

2.2.1 System configuration

A system definition contains one or more partitions which are not necessarily actual scheduler partitions, but simply logical separations within the system. Let's dissect one of the system configuration.

```
    'name': 'alaska',
    'descr': 'Default AlaSKA OpenHPC p3-appliances slurm cluster',
    'hostnames': ['alaska-login-0', 'alaska-compute'],
    'modules_system': 'lmod',
    'partitions': [
        {
            'name': 'login',
            'descr': 'Login node of AlaSKA OpenHPC cluster '
```

(continues on next page)

(continued from previous page)

```
                 '(Intel Core Processor (Broadwell, IBRS))',
            'scheduler': 'local',
            'launcher': 'local',
            'environs': [
                'builtin', 'gnu',
            ],
            'processor': {
                **alaska_login_topo,
            },
            'prepare_cmds': [
                'module purge',
            ],
            'extras': {
                'interconnect': '25',  # in Gb/s
            },
        },
        {
            'name': 'compute-gcc9-ompi4-roce-umod',
            'descr': 'AlaSKA OpenHPC cluster with 25Gb/s RoCE with gcc 9.3.0, openmpi 4.
↪1.1 and '
                 'UCX transport layer (Intel Core Processor (Broadwell, IBRS))',
            'scheduler': 'slurm',
            'launcher': 'mpirun',
            'time_limit': '0d8h0m0s',
            'access': [
                '--partition=full',
                '--exclusive',
            ],
            'max_jobs': 8,
            'environs': [
                'babel-stream-omp',
                'builtin',
                'ior',
                'imaging-iotest',
                'imaging-iotest-mkl',
                'imb',
                'numpy',
                'rascil',
            ],
            'modules':  [
                'gcc/9.3.0', 'git/2.31.1',
                'git-lfs/2.11.0', 'openmpi/4.1.1'
            ],
            'variables': [
                # scratch dir
                ['SCRATCH_DIR', '/scratch/mahendra'],
                # use RoCE 25 Gb/s
                ['UCX_NET_DEVICES', 'mlx5_0:1'],
                # UCX likes to spit out tons of warnings. Confine log to errors
                ['UCX_LOG_LEVEL', 'ERROR'],
                # Set locale
                ['LC_ALL', 'en_US.UTF-8'],
```

(continues on next page)

(continued from previous page)

```
            ],
            'processor': {
                **alaska_compute_topo,
            },
            'prepare_cmds': [
                # 'mpirun () { command mpirun --tag-output --timestamp-output '
                # '\"$@\"; }',  # wrap mpirun output tag and timestamp
                'module purge',
                'module use ${SPACK_ROOT}/var/spack/environments/alaska/lmod/linux*/Core
↪',
            ],
            'extras': {
                'interconnect': '25',  # in Gb/s
                'mem': '115234864000',  # total memory in bytes
            },
        },
        {
            'name': 'compute-icc21-impi21-roce-umod',
            'descr': 'AlaSKA OpenHPC cluster with 25Gb/s RoCE with ICC 2021.4.0, '
                     'Intel-MPI 2021.4.0 (Intel Core Processor (Broadwell, IBRS))',
            'scheduler': 'slurm',
            'launcher': 'mpiexec',
            'time_limit': '0d8h0m0s',
            'access': [
                '--partition=full',
                '--exclusive',
            ],
            'max_jobs': 8,
            'environs': [
                'babel-stream-tbb',
                'builtin',
                'intel-hpcg',
                'intel-hpl',
                'imaging-iotest',
                'imaging-iotest-mkl',
                'intel-stream',
            ],
            'modules':  [
                'intel-oneapi-compilers/2021.4.0', ' git/2.31.1',
                'git-lfs/2.11.0', 'intel-oneapi-mpi/2021.4.0',
            ],
            'variables': [
                # scratch dir
                ['SCRATCH_DIR', '/scratch/mahendra'],
                # # use ib (default) https://software.intel.com/content/www/us/en/
↪develop/articles/intel-mpi-library-2019-over-libfabric.html
                # ['FI_VERBS_IFACE', 'ib'],
                # Set locale
                ['LC_ALL', 'en_US.UTF-8'],
            ],
            'processor': {
                **alaska_compute_topo,
```

(continues on next page)

(continued from previous page)

```
            },
            'prepare_cmds': [
                # 'mpiexec () { command mpiexec -prepend-pattern \"[%r]: \" '
                # '\"$@\"; }',  # wrap mpirun with rank tag (intel mpi specific)
                'module purge',
                'module use ${SPACK_ROOT}/var/spack/environments/alaska/lmod/linux*/Core
↪',
            ],
            'extras': {
                'interconnect': '25',  # in Gb/s
```

The first part of the systems configuration is very self-explanatory. The key `hostnames` must have the names of the machines in this system. For instance, in this case, it is hostnames of the login and compute nodes of AlaSKA SLURM cluster.

Note: Note that the hostnames can be provided in the form of regular expressions and within the ReFrame, standard python package `re` is used to match the names with the hostnames

The `module_system` key specifies the type of the environment modules used by the system.

For each system, several partitions can be defined. As stated earlier, they can be physical scheduler partitions or abstract ones. The definition of partitions depends on the user and they can be defined based on type of tests to be performed on the system. Let's look at the first partition of our example here:

```
        {
            'name': 'login',
            'descr': 'Login node of AlaSKA OpenHPC cluster '
                     '(Intel Core Processor (Broadwell, IBRS))',
            'scheduler': 'local',
            'launcher': 'local',
            'environs': [
                'builtin', 'gnu',
            ],
            'processor': {
                **alaska_login_topo,
            },
            'prepare_cmds': [
                'module purge',
            ],
            'extras': {
                'interconnect': '25',  # in Gb/s
            },
        },
```

It is clear that these is a physical partitions of the cluster with is the login node of the AlaSKA cluster. The key `scheduler` defines the underlying workload manager used on the cluster and `launcher` is for the type of MPI wrapper used on the cluster to launch MPI jobs. For the login node, they both are `local` which means the jobs will be run on the shell without any parallel launcher. Typically, this partition can be used to clone the repositories, download datasets and compile the codes. We will come back to `environs` later. `prepare_cmds` are emitted at the top of the job scripts which can be the commands that needed for that partition to run the jobs. Finally, `processor` key specifies the processor topology of the node.

Important: The processor topology can be detected using ReFrame by running following command:

```
reframe/bin/reframe --detect-host-topology=topo.json
```

on the node that we want to get processor topology.

Now let's look into other partitions that we defined for AlaSKA.

```
        {
            'name': 'compute-gcc9-ompi4-roce-umod',
            'descr': 'AlaSKA OpenHPC cluster with 25Gb/s RoCE with gcc 9.3.0, openmpi 4.
↪1.1 and '
                     'UCX transport layer (Intel Core Processor (Broadwell, IBRS))',
            'scheduler': 'slurm',
            'launcher': 'mpirun',
            'time_limit': '0d8h0m0s',
            'access': [
                '--partition=full',
                '--exclusive',
            ],
            'max_jobs': 8,
            'environs': [
                'babel-stream-omp',
                'builtin',
                'ior',
                'imaging-iotest',
                'imaging-iotest-mkl',
                'imb',
                'numpy',
                'rascil',
            ],
            'modules':  [
                'gcc/9.3.0', 'git/2.31.1',
                'git-lfs/2.11.0', 'openmpi/4.1.1'
            ],
            'variables': [
                # scratch dir
                ['SCRATCH_DIR', '/scratch/mahendra'],
                # use RoCE 25 Gb/s
                ['UCX_NET_DEVICES', 'mlx5_0:1'],
                # UCX likes to spit out tons of warnings. Confine log to errors
                ['UCX_LOG_LEVEL', 'ERROR'],
                # Set locale
                ['LC_ALL', 'en_US.UTF-8'],
            ],
            'processor': {
                **alaska_compute_topo,
            },
            'prepare_cmds': [
                # 'mpirun () { command mpirun --tag-output --timestamp-output '
                # '\"$@\"; }',  # wrap mpirun output tag and timestamp
                'module purge',
```

(continues on next page)

(continued from previous page)

```
                'module use ${SPACK_ROOT}/var/spack/environments/alaska/lmod/linux*/Core
↪',
            ],
            'extras': {
                'interconnect': '25',  # in Gb/s
                'mem': '115234864000',  # total memory in bytes
            },
        },
        {
            'name': 'compute-icc21-impi21-roce-umod',
            'descr': 'AlaSKA OpenHPC cluster with 25Gb/s RoCE with ICC 2021.4.0, '
                     'Intel-MPI 2021.4.0 (Intel Core Processor (Broadwell, IBRS))',
            'scheduler': 'slurm',
            'launcher': 'mpiexec',
            'time_limit': '0d8h0m0s',
            'access': [
                '--partition=full',
                '--exclusive',
            ],
            'max_jobs': 8,
            'environs': [
                'babel-stream-tbb',
                'builtin',
                'intel-hpcg',
                'intel-hpl',
                'imaging-iotest',
                'imaging-iotest-mkl',
                'intel-stream',
            ],
            'modules':  [
                'intel-oneapi-compilers/2021.4.0', ' git/2.31.1',
                'git-lfs/2.11.0', 'intel-oneapi-mpi/2021.4.0',
            ],
            'variables': [
                # scratch dir
                ['SCRATCH_DIR', '/scratch/mahendra'],
                # # use ib (default) https://software.intel.com/content/www/us/en/
↪develop/articles/intel-mpi-library-2019-over-libfabric.html
                # ['FI_VERBS_IFACE', 'ib'],
                # Set locale
                ['LC_ALL', 'en_US.UTF-8'],
            ],
            'processor': {
                **alaska_compute_topo,
            },
            'prepare_cmds': [
                # 'mpiexec () { command mpiexec -prepend-pattern \"[%r]: \" '
                # '\"$@\"; }',  # wrap mpirun with rank tag (intel mpi specific)
                'module purge',
                'module use ${SPACK_ROOT}/var/spack/environments/alaska/lmod/linux*/Core
↪',
            ],
```

(continues on next page)

(continued from previous page)

```
            'extras': {
                'interconnect': '25',  # in Gb/s
```

It is clear that these are "abstract" partitions that are based on physical partitions of compute nodes of AlaSKA. For instance, partition `compute-gcc-ompi-roce-umod` supports 25 Gb/s RDMA over Converged Ethernet (RoCE) network interface with GCC 9.3.0 and OpenMPI 4.1.1 using UCX support. List of modules that needed to be loaded every time this partition is used can be specified using `modules` key. To be able to use this partition with the above stated specs, we will have to load OpenMPI 4.1.1 module which is present in `modules` key. The key `access` defines the additional parameters that needed to be passed to the scheduler in order to submit jobs. These typically include the partition that user can access and account name of the user on the system. `max_jobs` is maximum number of concurrent jobs that ReFrame can submit to the scheduler. The `variables` key can be used to define any environment variables that needed to be defined for this partitions before we run the job. Similarly, in the variables we are setting UCX parameter to use RoCE for the transport layer and specifying `mlx5_0:1` port. When we run a test in this partition, ReFrame loads all necessary modules and sets environment variables to use this spec. Likewise, `compute-gcc-impi-roce-umod` partition uses Intel MPI.

This gives a general idea of what system and partition can do in ReFrame framework. It gives a plethora of possibilities to the user to define several partitions and we can run tests on these partitions without changing any generic logic to the test *per se*.

2.2.2 Environment configuration

Partitions then support one or more environments which describe the modules to be loaded, environment variables, options *etc*. Environments are defined separately from partitions so they may be specific to a system and partition, common to multiple systems or partitions, or a default environment may be overridden for specific systems and/or partitions. The third level is the tests themselves, which may also define modules to load etc. as well as which environments, partitions and systems they are valid for. ReFrame then runs tests on combinations of valid partitions and environments. So we can see the hierarchy of configuration using systems, environments and tests.

Consider the environment example shown below:

```
    {
        'name': 'imaging-iotest',
        'target_systems': [
            'juwels-cluster:batch-gcc9-ompi4-ib-smod',
            'juwels-cluster:batch-gcc9-ompi4-ib-smod-mem192',
        ],
        'modules': [
            'HDF5/1.10.6', 'FFTW/3.3.8 ',
            'CMake/3.18.0',
        ],
        'cc': 'mpicc',
        'cxx': 'mpicxx',
        'ftn': 'mpif90',
    },
    {
        'name': 'imaging-iotest',
        'target_systems': [
            'juwels-cluster:batch-icc20-pmpi5-ib-smod',
            'juwels-cluster:batch-icc20-pmpi5-ib-smod-mem192',
        ],
        'modules': [
```

(continues on next page)

(continued from previous page)

```
            'HDF5/1.10.6', 'FFTW/3.3.8 ',
            'CMake/3.18.0'
        ],
        'cc': 'mpiicc',
        'cxx': 'mpiicpc',
        'ftn': 'mpiifort',
    },  # <end JUWELS system software stack>
    {
        'name': 'imaging-iotest-mkl',
        'target_systems': [
            'juwels-cluster:batch-gcc9-ompi4-ib-smod',
            'juwels-cluster:batch-gcc9-ompi4-ib-smod-mem192',
```

As the name of the environment suggests, it is defined for Imaging IO Test. We need to define the key `target_systems` where this environment is valid. Similarly, for each system definition, we need to define the `environs` key to specify the environments that we want to use within that system partition.

Note: The environments defined in `environ` for each system partition must be appear in `target_systems` of that environment and *vice-versa*. Otherwise, ReFrame will complain about missing system partition or environment for a given test

And finally, the `modules` keyword specifies the dependencies of the test we will run within this environment. In the current example of Imaging IO test, we need HDF5 and FFTW libraries for the test and hence, we load them. Additionally, Imaging IO test can use FFTW from Intel MKL libraries as well when Intel OneAPI is available on the system. Hence, we define another environment here that uses FFTW from intel MKL libraries. In this way, environments can be defined for different tests.

It is up to the user how the system, partitions and environments are defined. A very generic systems, partitions and environments can be defined and test related modules and variables can be defined within python test scripts as well.

2.3 ReFrame usage

2.3.1 Basic usage

Once the system and environment configuration is finished, we can run ReFrame tests. Let's consider a simple hello world ReFrame:

```python
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class HelloMultiLangTest(rfm.RegressionTest):

    lang = parameter(['c', 'cpp'])
    arg = parameter(['Mercury', 'Venus', 'Mars'])

    def __init__(self):

        self.valid_systems = ['*']
```

(continues on next page)

(continued from previous page)

```
        self.valid_prog_environs = ['gnu']
        self.tags |= {self.lang, self.arg}
        self.executable_opts = [self.arg, '> hello.out']
        self.sanity_patterns = sn.assert_found(
            r'Hello, World from {}!'.format(self.arg), 'hello.out')

    @run_before('compile')
    def set_sourcepath(self):
        self.sourcepath = f'hello.{self.lang}'
```

The test can be launched using the following command

```
reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_helloworld.py -r
```

This command has to be executed from the root of the repository. This will run **all** the tests defined in `reframe_iotest.py` file. The flag `-C` is used to specify the ReFrame configuation file. Alternatively, an environment variable `RFM_CONFIG_FILE` can be set to avoid passing this variable every time on CLI. The flag `-c` is used to tell ReFrame which test we want to run and finally, `-r` tells ReFrame to actually run the tests. Useful CLI arguments are as follows:

- Option `-l` / `--list` : List all tests defined in the python script
- Option `-L` / `--list-detailed` : List all the dependencies of the tests. More details on test dependencies in ReFrame can be found here.
- Option `--performance-report` : Print the performance metrics at the end of the test
- Option `-p` / `--prgenv` : Choose the environments that we want to run the tests. By default, ReFrame will run tests on all valid environments
- Option `--system` : Choose the system partition to run the tests.
- Option `-t` / `--tag` : Choose the tags we want to confine the tests. More about tags will be discussed later.

2.3.2 Parameterisation of tests

Parameterisation is very powerful feature of ReFrame. In the present example, we defined two sets of parameters namely, `lang` and `arg` . The parameter `lang` specifies the language the source code is written. Both C and C++ source codes of the sample code can be found in `helloworld/src` folder. And the parameter `arg` adds the CLI argument to the source. For example, running `gcc -o helloworld helloworld.c && helloworld Mercury` will print `Hello, World from Mercury!` on the terminal. Now let's check number of tests ReFrame recognises from this simple test by running `reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_helloworld.py -l`. The output is as follows:

```
[ReFrame Setup]
version:           3.8.0-dev.2+8a9ceeda
command:           'reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_
↪helloworld.py -l'
launched by:       mpaipuri@fnancy
working directory: '/home/mpaipuri/ska-sdp-benchmark-tests'
settings file:     'reframe_config.py'
check search path: (R) '/home/mpaipuri/ska-sdp-benchmark-tests/helloworld/reframe_
↪helloworld.py'
stage directory:   '/home/mpaipuri/ska-sdp-benchmark-tests/stage'
```

(continues on next page)

(continued from previous page)

```
output directory:  '/home/mpaipuri/ska-sdp-benchmark-tests/output'

[List of matched checks]
- HelloMultiLangTest_cpp_Venus (found in '/home/mpaipuri/ska-sdp-benchmark-tests/
↪helloworld/reframe_helloworld.py')
- HelloMultiLangTest_c_Mercury (found in '/home/mpaipuri/ska-sdp-benchmark-tests/
↪helloworld/reframe_helloworld.py')
- HelloMultiLangTest_cpp_Mars (found in '/home/mpaipuri/ska-sdp-benchmark-tests/
↪helloworld/reframe_helloworld.py')
- HelloMultiLangTest_c_Venus (found in '/home/mpaipuri/ska-sdp-benchmark-tests/
↪helloworld/reframe_helloworld.py')
- HelloMultiLangTest_c_Mars (found in '/home/mpaipuri/ska-sdp-benchmark-tests/helloworld/
↪reframe_helloworld.py')
- HelloMultiLangTest_cpp_Mercury (found in '/home/mpaipuri/ska-sdp-benchmark-tests/
↪helloworld/reframe_helloworld.py')
Found 6 check(s)

Log file(s) saved in '/home/mpaipuri/ska-sdp-benchmark-tests/reframe.log', '/home/
↪mpaipuri/ska-sdp-benchmark-tests/reframe.out'
```

It is clear that ReFrame found 6 tests, helloworld code with C and 3 arguments and helloworld with C++ and 3 arguments. Let's run these tests and see what output we will get by using `reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_helloworld.py -r` command

```
[ReFrame Setup]
version:           3.8.0-dev.2+8a9ceeda
command:           'reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_
↪helloworld.py -r'
launched by:       mpaipuri@fnancy
working directory: '/home/mpaipuri/ska-sdp-benchmark-tests'
settings file:     'reframe_config.py'
check search path: (R) '/home/mpaipuri/ska-sdp-benchmark-tests/helloworld/reframe_
↪helloworld.py'
stage directory:   '/home/mpaipuri/ska-sdp-benchmark-tests/stage'
output directory:  '/home/mpaipuri/ska-sdp-benchmark-tests/output'

[==========] Running 6 check(s)
[==========] Started on Mon Aug 16 11:20:49 2021

[----------] started processing HelloMultiLangTest_c_Mercury (HelloMultiLangTest_c_
↪Mercury)
[ RUN      ] HelloMultiLangTest_c_Mercury on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_c_Mercury (HelloMultiLangTest_c_
↪Mercury)

[----------] started processing HelloMultiLangTest_c_Venus (HelloMultiLangTest_c_Venus)
[ RUN      ] HelloMultiLangTest_c_Venus on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_c_Venus (HelloMultiLangTest_c_Venus)

[----------] started processing HelloMultiLangTest_c_Mars (HelloMultiLangTest_c_Mars)
[ RUN      ] HelloMultiLangTest_c_Mars on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_c_Mars (HelloMultiLangTest_c_Mars)
```

(continues on next page)

(continued from previous page)

```
[----------] started processing HelloMultiLangTest_cpp_Mercury (HelloMultiLangTest_cpp_
↪Mercury)
[ RUN      ] HelloMultiLangTest_cpp_Mercury on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_cpp_Mercury (HelloMultiLangTest_cpp_
↪Mercury)

[----------] started processing HelloMultiLangTest_cpp_Venus (HelloMultiLangTest_cpp_
↪Venus)
[ RUN      ] HelloMultiLangTest_cpp_Venus on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_cpp_Venus (HelloMultiLangTest_cpp_
↪Venus)

[----------] started processing HelloMultiLangTest_cpp_Mars (HelloMultiLangTest_cpp_Mars)
[ RUN      ] HelloMultiLangTest_cpp_Mars on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_cpp_Mars (HelloMultiLangTest_cpp_
↪Mars)

[----------] waiting for spawned checks to finish
[       OK ] (1/6) HelloMultiLangTest_cpp_Venus on nancy-g5k:frontend using gnu␣
↪[compile: 0.445s run: 0.652s total: 1.136s]
[       OK ] (2/6) HelloMultiLangTest_c_Mars on nancy-g5k:frontend using gnu [compile: 0.
↪142s run: 1.703s total: 1.886s]
[       OK ] (3/6) HelloMultiLangTest_c_Mercury on nancy-g5k:frontend using gnu␣
↪[compile: 0.149s run: 2.160s total: 2.349s]
[       OK ] (4/6) HelloMultiLangTest_cpp_Mars on nancy-g5k:frontend using gnu [compile:␣
↪0.451s run: 0.455s total: 0.946s]
[       OK ] (5/6) HelloMultiLangTest_c_Venus on nancy-g5k:frontend using gnu [compile:␣
↪0.131s run: 2.249s total: 2.427s]
[       OK ] (6/6) HelloMultiLangTest_cpp_Mercury on nancy-g5k:frontend using gnu␣
↪[compile: 0.437s run: 1.738s total: 2.215s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 6/6 test case(s) from 6 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Mon Aug 16 11:20:52 2021
Run report saved in '/home/mpaipuri/.reframe/reports/run-report.json'
Log file(s) saved in '/home/mpaipuri/ska-sdp-benchmark-tests/reframe.log', '/home/
↪mpaipuri/ska-sdp-benchmark-tests/reframe.out'
```

ReFrame ran all the possible tests and they all passed. The way ReFrame judges if a test is passed or failed is through the sanity check. In the `reframe_helloworld.py` script, we define the so-called `sanity_patterns`. As each test should spit out `Hello, World from <arg>!`, ReFrame checks using regular expressions if this line is present in the standard output. If present, ReFrame marks test as pass, else it marks it as fail. Of course more advanced sanity checks can be written for complicated benchmarks. More details on sanity checking can be found here.

2.3.3 Tagging of tests

What if we want to run only subset of tests. This can come handy when we are running relatively big benchmarks when we do not want to run all the tests we defined within the ReFrame script. This can be achieved through the `tag` feature of the ReFrame. That is where the line `self.tags |= {self.lang, self.arg}` comes into play. We are tagging each parameterised test with its tag. We can customise the tags as per our need, for instance, `self.tags |= {"language=%s" % self.lang, "planet=%s" % self.arg}`. To restrict the tests to a given tag, we need to simply provide the `-t` flag at CLI as follows:

```
reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_helloworld.py -t c$ -r
```

Note: Reframe uses regular expressions to match the tags with parameters. In this case, if we use `-t c -t Mercury`, it selects the tests from `cpp` as well as `c` matches with `cpp` in regular expression context. So, we should use the end of line `$` regular expression operator in these sort of situations

Let's check the output of the above command:

```
[ReFrame Setup]
version:           3.8.0-dev.2+8a9ceeda
command:           'reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_
↪helloworld.py -t c$ -r'
launched by:       mpaipuri@fnancy
working directory: '/home/mpaipuri/ska-sdp-benchmark-tests'
settings file:     'reframe_config.py'
check search path: (R) '/home/mpaipuri/ska-sdp-benchmark-tests/helloworld/reframe_
↪helloworld.py'
stage directory:   '/home/mpaipuri/ska-sdp-benchmark-tests/stage'
output directory:  '/home/mpaipuri/ska-sdp-benchmark-tests/output'

[==========] Running 3 check(s)
[==========] Started on Mon Aug 16 11:45:13 2021

[----------] started processing HelloMultiLangTest_c_Mercury (HelloMultiLangTest_c_
↪Mercury)
[ RUN      ] HelloMultiLangTest_c_Mercury on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_c_Mercury (HelloMultiLangTest_c_
↪Mercury)

[----------] started processing HelloMultiLangTest_c_Venus (HelloMultiLangTest_c_Venus)
[ RUN      ] HelloMultiLangTest_c_Venus on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_c_Venus (HelloMultiLangTest_c_Venus)

[----------] started processing HelloMultiLangTest_c_Mars (HelloMultiLangTest_c_Mars)
[ RUN      ] HelloMultiLangTest_c_Mars on nancy-g5k:frontend using gnu
[----------] finished processing HelloMultiLangTest_c_Mars (HelloMultiLangTest_c_Mars)

[----------] waiting for spawned checks to finish
[       OK ] (1/3) HelloMultiLangTest_c_Mercury on nancy-g5k:frontend using gnu␣
↪[compile: 0.143s run: 0.493s total: 0.675s]
[       OK ] (2/3) HelloMultiLangTest_c_Venus on nancy-g5k:frontend using gnu [compile:␣
↪0.136s run: 0.476s total: 0.649s]
[       OK ] (3/3) HelloMultiLangTest_c_Mars on nancy-g5k:frontend using gnu [compile: 0.
```

(continues on next page)

(continued from previous page)

```
↪138s run: 0.451s total: 0.628s]
[----------] all spawned checks have finished

[  PASSED  ] Ran 3/3 test case(s) from 3 check(s) (0 failure(s), 0 skipped)
[==========] Finished on Mon Aug 16 11:45:14 2021
Run report saved in '/home/mpaipuri/.reframe/reports/run-report.json'
Log file(s) saved in '/home/mpaipuri/ska-sdp-benchmark-tests/reframe.log', '/home/
↪mpaipuri/ska-sdp-benchmark-tests/reframe.out'
```

As we can see from the output, only tests with `helloworld.c` has been executed. We can specify multiple tags using as many `-t` options as we want as follows:

```
reframe/bin/reframe -C reframe_config.py -c helloworld/reframe_helloworld.py -t c$ -t␣
↪Mercury -r
```

This will execute only one test using `helloworld.c` and `Mercury` as a command line argument.

Similarly, if there are multiple environments defined for each test, we can confine the test to given environment using `-p` flag.

Note: The CLI arguments for tags, `-t` , name, `-n` , and environment, `-p` , take regular expression as input and match the corresponding names from the tests. Hence, care should be taken while specifying them.

2.3.4 Tests dependencies

One of the typical scenario when benchmarking is to do scalability tests. Using a naive approach of ReFrame to do scalability test would be to clone the repository, compile the sources and run the benchmark for each node/runtime configuration. This is a shear waste of time and resources as all the runs within a given partition and environment share same sources and executable. This can be addressed using test dependencies and fixtures.

An extensive overview of how test dependencies work in ReFrame is out-of-scope of current documentation. The user are advised to check the official documentation of the test dependencies from ReFrame which gives a very good idea of how it works and how to implement them. Similarly, fixtures can be used as well in place of dependencies which is documented with a nice example in the official documentation of ReFrame.

2.3.5 Multiple runs

Since ReFrame v3.12.0 there exists support for the CLI-flag `--repeat=N`. This runs all given tests N times independently of each others and collects their performance variables in independent perflogs. Those perflogs are aggregated by the perflog reading methods in `modules/utils.py`.

CHAPTER

THREE

OVERVIEW OF SPACK

3.1 Introduction

We use Spack to build the software stack locally to run the benchmark tests. Spack supports both Environment Modules and LMod. Here we provide some basic steps to install packages using Spack and integrating them using LMod. More details on integration of Spack with TCL and LMod can be found here.

Note: Some of the documentation provided by Spack on the integration of Spack with LMod is outdated and please be aware that one can run into issues while following Spack tutorials.

3.2 Installation

Installing Spack is trivial. It involves cloning the git repository and sourcing the environment.

```
cd ~
git clone https://github.com/spack/spack.git
```

This clones the repository in the home directory of the user. Next steps involve modifying the `~/.bashrc` to source the environment.

```
export SPACK_ROOT=~/spack
. $SPACK_ROOT/share/spack/setup-env.sh
```

By sourcing this environment, all Spack commands will be available in the shell. This environment also adds module path to `lmod` after we install packages.

3.3 Usage

3.3.1 Basics

Some of the basic Spack commands are provided here. To list all the packages that Spack can install, we can use

```
spack list
```

If we want to search for a particular package, we can add the keyword to `spack list` command. For instance, to check if OpenMPI is available, we can query

```
spack list openmpi
```

To get more details for a given package, we can use `spack info` command.

```
spack info openmpi
```

This command gives all the info about the package, variants, dependencies, *etc*. To check the available versions of a given package, we can use `spack versions <package name>` command.

To install a Spack package, we can simply use `spack install <package name>` and similarly, to uninstall `spack uninstall <package name>`. To install a specific version, use `spack install <package name>@<version>`. The suffix `@` specifies the version number here. More details on installing packages will be discussed in next sections.

3.3.2 Workflow

The general workflow is that we will use compilers provided on the platform to build a "standard" compiler tool chain and in-turn use this tool chain to build all the necessary packages. In this way, we will bring the software stack on different platforms to a common ground and it also helps us to compare the benchmarks on different hardware/architecture provided by different platforms using the same software stack.

First step is to provide a list of compilers that are available to the Spack. This can be done using

```
spack compiler list
```

This should list at least the compiler that is provided by the base OS. New compilers can be added to the list by loading appropriate module. For instance, if there is `gcc-7.3.0` available on the module system, we can add it to Spack compiler list using

```
module load gcc-7.3.0
spack compiler find
```

This command finds the newly available compiler and adds it to the Spack compiler list.

The following step is to build a compiler tool chain using the system provided compiler. Going with the previous example, if system provided compiler is `gcc-7.3.0` and we would like to build, say GCC 9.3.0, we should use following command

```
spack install gcc@9.3.0 %gcc-7.3.0
```

In the above command, we are telling Spack to build GCC version 9.3.0 by using `@`. The suffix `%` is used to specify the compiler toolchain to build the package. Here, we are telling Spack to build GCC 9.3.0 using GCC 7.3.0 that is provided on the system. We can add `-j <number of jobs>` to the install command to build the packages using concurrency.

Once the GCC 9.3.0 is built successfully, we should add it to the list of compilers of Spack. For that we can simply load the compiler first and add to the list.

```
spack load gcc
spack compiler find
```

More details on compiler configuration in Spack can be found in the ` compiler documentation <https://spack.readthedocs.io/en/latest/getting_started.html#spack-compiler-find>`_.

3.4 Installing packages

Now that we have a toolchain built, we would want to build necessary packages to run our codes. In this documentation, we will use OpenMPI as an example to demonstrate the process of building packages using Spack. Let's say we want to build OpenMPI 3.1.3 version on our machine. We can query the specification of this package in the Spack using `spack spec openmpi@3.1.3`. This gives us an output as follows:

```
Input spec
--------------------------------
openmpi@3.1.3

Concretized
--------------------------------
openmpi@3.1.3%gcc@9.3.0~atomics~cuda~cxx~cxx_exceptions+gpfs~internal-hwloc~java~
↪legacylaunchers~lustre~memchecker~pmi~singularity~sqlite3
+static~thread_multiple+vt+wrapper-rpath fabrics=none schedulers=none arch=linux-centos7-
↪broadwell
^hwloc@1.11.13%gcc@9.3.0~cairo~cuda~gl~libudev+libxml2~netloc~nvml+pci+shared␣
↪patches=d1d94a4af93486c88c70b79cd930979f3a2a2b5843708e8c7c1655f18b9fc694 arch=linux-
↪centos7-broadwell
^libpciaccess@0.16%gcc@9.3.0 arch=linux-centos7-broadwell
^libtool@2.4.6%gcc@9.3.0 arch=linux-centos7-broadwell
^m4@1.4.19%gcc@9.3.0+sigsegv arch=linux-centos7-broadwell
^libsigsegv@2.13%gcc@9.3.0 arch=linux-centos7-broadwell
^pkgconf@1.7.4%gcc@9.3.0 arch=linux-centos7-broadwell
^util-macros@1.19.3%gcc@9.3.0 arch=linux-centos7-broadwell
^libxml2@2.9.10%gcc@9.3.0~python arch=linux-centos7-broadwell
^libiconv@1.16%gcc@9.3.0 arch=linux-centos7-broadwell
^xz@5.2.5%gcc@9.3.0~pic libs=shared,static arch=linux-centos7-broadwell
^zlib@1.2.11%gcc@9.3.0+optimize+pic+shared arch=linux-centos7-broadwell
^ncurses@6.2%gcc@9.3.0~symlinks+termlib abi=none arch=linux-centos7-broadwell
^numactl@2.0.14%gcc@9.3.0␣
↪patches=4e1d78cbbb85de625bad28705e748856033eaafab92a66dffd383a3d7e00cc94,
↪62fc8a8bf7665a60e8f4c93ebbd535647cebf74198f7afafec4c085a8825c006 arch=linux-centos7-
↪broadwell
^autoconf@2.69%gcc@9.3.0 arch=linux-centos7-broadwell
^perl@5.34.0%gcc@9.3.0+cpanm+shared+threads arch=linux-centos7-broadwell
^berkeley-db@18.1.40%gcc@9.3.0+cxx~docs+stl␣
↪patches=b231fcc4d5cff05e5c3a4814f6a5af0e9a966428dc2176540d2c05aff41de522 arch=linux-
↪centos7-broadwell
^bzip2@1.0.8%gcc@9.3.0~debug~pic+shared arch=linux-centos7-broadwell
^diffutils@3.7%gcc@9.3.0 arch=linux-centos7-broadwell
^gdbm@1.19%gcc@9.3.0 arch=linux-centos7-broadwell
^readline@8.1%gcc@9.3.0 arch=linux-centos7-broadwell
^automake@1.16.3%gcc@9.3.0 arch=linux-centos7-broadwell
^openssh@8.5p1%gcc@9.3.0 arch=linux-centos7-broadwell
^libedit@3.1-20210216%gcc@9.3.0 arch=linux-centos7-broadwell
^openssl@1.1.1k%gcc@9.3.0~docs+systemcerts arch=linux-centos7-broadwell
```

This says that the OpenMPI 3.1.3 will be built using GCC 9.3.0 (`openmpi@3.1.3%gcc@9.3.0`). The suffices ~ and + are used to specify the configuration options. The line `openmpi@3.1.3%gcc@9.3.0~atomics~cuda~cxx~cxx_exceptions+gpfs ... fabrics=none schedulers=none` tells us that OpenMPI will be build **without** the support of atomics, cuda, C++ bindinfs, HWLoc, Java, LUSTRE, *etc*. Similarly, by default the spec says that it will be build with GPFS support, static libraries, *etc*. The suffix `^` is used to sepcify the dependencies of the package. So, all the packages listed with `^` are dependencies of the OpenMPI and will be installed before installing OpenMPI. To get more details on what each variant mean, we can use `spack info openmpi@3.1.3` command.

To install OpenMPI 3.1.3 using GCC 9.3.0 (that we have already built before) using IB verbs support and integrating with the workload scheduler of the system, we should use following command

```
spack install -j 32 openmpi@3.1.3 %gcc@9.3.0 fabrics=verbs schedulers=auto
```

The "a=b" portions specify the variants. In this case, we are telling Spack to build OpenMPI using IB verbs support for fabrics and by setting schedulers to auto, Spack will detect the workload manager and integrates it with OpenMPI. For instance, we want to build with multiple thread support, we can specify it using

```
spack install -j 32 openmpi@3.1.3+thread_multiple %gcc@9.3.0 fabrics=verbs␣
↪schedulers=auto
```

All the default configuration options can be overridden during the installation using + and ~ suffices. More details can be found in the instllation documentation of the Spack.

3.5 Module files

Although we can load the modules using `spack load <package name>` command, it is preferable that Spack installed packages are integrated into the module tool. Here we will see how we can add Spack installed modules to the environment module tool.

If there is no module tool installed on the system, we should first install the module tool itself. We will use `lmod` in this example. We can install it using following command

```
spack install -j 32 lmod %gcc@9.3.0
```

After successful installation, we should source the `lmod` environment using following command

```
. $(spack location -i lmod)/lmod/lmod/init/bash
```

Here we are using `spack location` command to find the installation location of `lmod`. After this we should re-source the Spack `. share/spack/setup-env.sh` environment so that Spack modules will be put in the module path. Recall we added this line to `~/.bashrc` script so that it will be sourced every time we start a shell.

Note: We should do these steps only if there is no module tool provided by the system. If there is already `lmod` installed on the system, we can skip these steps.

After sourcing, Spack environment, we should be able to see all the modules installed using Spack by querying `module avail`. This can be further verifed by looking at `MODULEPATH` environment variable, where we will see path to Spack installed packages.

Now we have the so-called Non-hierarchical module files, where all the modules that are all generated in the same root directory. Ideally, we would like to use hierarchical module files, where `MODULEPATH` is changed dynamically. In non-hierarchical module file system, it is easy to load incompatible module files at the same time. Using hierarchical module files, we can avoid situations like this by "unlocking" the dependent packages only after the parent packages are loaded. More details on this can be found at Spack tutorials.

The most widely used hierarchy is the so called `Core/Compiler/MPI` where, on top of the compilers, different MPI libraries also unlock software linked to them. In order to do this, we need to add a configuration file `modules.yaml` to the `~/.spack` directory with the following contents

```
modules:
 default:
  enable::
    - lmod
  lmod:
    core_compilers:
     - gcc@7.3.0
    hierarchy:
    - mpi
    hash_length: 0
    projections:
      all: '{name}/{version}'
```

In this configuration, `enable::` means telling Spack to use only `lmod` as the module system. We should add system provided compiler in the `core_compilers` section. By setting `hash_length` to zero, we will eliminate the hashes on the module names. The `projections` section tells spack on how to display module files. In this example, module files will be shown as `openmpi/3.1.3`.

Once we add this file to the home directory, we should regenerate the module files using

```
spack module lmod refresh --delete-tree -y
```

and then update `MODULEPATH` using

```
module unuse $HOME/spack/share/spack/modules/linux-centos7-broadwell
module use $HOME/spack/share/spack/lmod/linux-centos7-x86_64/Core
```

We should unuse the module path that is added everytime we source the Spack environment and then add new module path that points to the core modules. The above two lines can be added to the `~/.bashrc` file. Now using `module avail` command gives an output as follows:

```
-------------------------------------------------- /alaska/mahendra/spack/share/spack/
↪lmod/linux-centos7-x86_64/Core ---------------------------------------------------
gcc/9.3.0    gmp/6.2.1    mpc/1.1.0    mpfr/3.1.6    patch/2.7.6    zlib/1.2.11

---------------------------------------------------------------------- /opt/ohpc/pub/
↪modulefiles ------------------------------------------------------------------------
gnu/5.4.0    gnu7/7.3.0    pmix/2.2.2    prun/1.3

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of␣
↪the "keys".
```

We can see that OpenMPI that we installed does not appear in the list. This is due to the hierarchical module system we are using. Once we load GCC 9.3.0 using `module load gcc/9.3.0`, we will see bunch of other modules available to use.

```
------------------------------------------------ /alaska/mahendra/spack/share/spack/lmod/
↪linux-centos7-x86_64/gcc/9.3.0 -------------------------------------------------
autoconf/2.69          fftw/3.3.9                      intel-oneapi-tbb/2021.3.0    ␣
↪libsigsegv/2.13        openssl/1.1.1k          tar/1.34
```

(continues on next page)

(continued from previous page)

```
automake/1.16.3        findutils/4.8.0                  libbsd/0.11.3              ␣
↪libtool/2.4.6          perl/5.34.0          ucx/1.10.1
berkeley-db/18.1.40    flex/2.6.3                       libedit/3.1-20210216       ␣
↪libxml2/2.9.10         pkgconf/1.7.4        util-linux-uuid/2.36.2
bison/3.7.6            gdbm/1.19                        libevent/2.1.12             m4/
↪1.4.19               py-docutils/0.15.2     util-macros/1.19.3
bzip2/1.0.8            gettext/0.21                     libffi/3.3                 ␣
↪ncurses/6.2            py-setuptools/50.3.2    xz/5.2.5
cmake/3.20.5           hdf5/1.10.7                      libiconv/1.16              ␣
↪numactl/2.0.14         python/3.8.11        zlib/1.2.11           (D)
cpio/2.13              hwloc/1.11.13                    libmd/1.0.3                ␣
↪openmpi/3.1.3          rdma-core/34.0
diffutils/3.7          hwloc/2.5.0                  (D)    libnl/3.3.0                ␣
↪openmpi/4.1.1    (D)    readline/8.1
expat/2.4.1            intel-oneapi-mkl/2021.3.0        libpciaccess/0.16          ␣
↪openssh/8.5p1          sqlite/3.35.5

--------------------------------------------------- /alaska/mahendra/spack/share/spack/
↪lmod/linux-centos7-x86_64/Core ---------------------------------------------------
gcc/9.3.0 (L)    gmp/6.2.1    mpc/1.1.0    mpfr/3.1.6    patch/2.7.6    zlib/1.2.11

----------------------------------------------------------------------- /opt/ohpc/pub/
↪modulefiles -----------------------------------------------------------------------
gnu/5.4.0    gnu7/7.3.0    pmix/2.2.2    prun/1.3

Where:
D:  Default Module
L:  Module is loaded

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of␣
↪the "keys".
```

More details on Spack can be gathered from the extensive documentation of the project. Only basic use case of Spack is covered here and for more advanced use cases, please refer to the documentation.

3.6 Setting up environment for SKA SDP Benchmark tests

This can be done in two different ways. The first and less recommended way is to use the legacy bash script provided in `spack/scripts` folder and choose the appropriate CLI options to install the required packages on the platform. The reproducibility with this approach is not guaranteed, especially when there are changes in the upstream Spack packages.

The second approach which is strongly recommended is to use Spack environment spec files provided for each system in the `spack/spack-tests` folder. Inside the folder we typically find Spack config files of different systems and a ReFrame test to deploy the software stack using those Spack config files. Therefore, these Spack tests are nothing but “meta tests” that need to run before running actual benchmark tests to deploy the necessary software stack.

Both approaches are illustrated in the following sections.

3.6.1 Config based approach

This approach is based on Spack environments which are which are collection of set of packages. A more detailed description of Spack environments is out-of-scope of present context. Spack environments can be defined by Spack configuration files. For each system, typically we will find five different configuration files namely,

- `compilers.yml`: All compiler related information is placed in this file
- `config.yml`: General configuration of Spack can be defined here
- `modules.yml`: We use LMod environment files and related configuration is defined in this file
- `packages.yml`: A list of package preferences are defined here
- `spack.yml`: This is the main file that includes list of packages to install

All these files together define a Spack environment spec. The `packages.yml` will list all the root packages and their dependencies along with the preferred version and variants. A simple example is shown below:

```
packages:
  hdf5:
    variants: ~cxx ~fortran ~hl ~ipo ~java +mpi +shared ~szip ~threadsafe +tools␣
↪api=default build_type=RelWithDebInfo
    version:
      - 1.10.7
```

This config tells Spack that HDF5 preferred version is 1.10.7 with MPI support. So, when we use this file to define Spack environment, Spack will always “try” to build HDF5 with the configuration shown above.

Note: Depending on the complexity of the environment (set of all packages to be installed), Spack may not respect the package spec defined in `packages.yml` file. In order to really constraint a package to certain spec, we need to define that under `spec` in `spack.yml` file.

Under `spec` section in `spack.yml` file, there will be set of packages to be installed in the environment. Thus, as long as we use same config files, we can always deploy the same software stack on same system or even on different systems. This gives us a great deal of reproducibility within and between systems.

A typical workflow in Spack environment is:

- Create an named environment using `spack.yml` file and activate it
- Concretize the environment
- Install the packages
- Generate module files

All the generated module files are placed under `$SPACK_ROOT/var/spack/environments/<env_name>/lmod/<arch>/Core`. Once we add this path to `MODULEPATH`, we can use the Spack packages using `module load` commands.

All these steps are abstracted away from the user by employing ReFrame test to automate this workflow. A separate ReFrame test is defined for each system/partition where the name of the environment is defined using the partition name.

Let’s look into an example. If we want to install all the necessary packages on JUWELS cluster, we need to execute following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
```

(continues on next page)

(continued from previous page)

```
reframe/bin/reframe -C reframe_config.py -c spack/spack-tests/juwels/cluster/reframe_
↪juwelscluster.py --run
```

Important: The user needs to replace the variable `spack_root` in the test file to point to user specific path. This can be done either by directly editing the test file or at the CLI using `-S` option, *e.g.*, `reframe/bin/reframe -C reframe_config.py -c spack/spack-tests/juwels/cluster/reframe_juwelscluster.py -S spack_root=<my_spack_root_path> --run`.

This test will clone Spack repository `v0.17.0`, creates an environment, install the the packages defined in `spack/spack-tests/juwels/cluster/configs/spack.yml` file and creates module files. Depending on the IO performance of the file system where Spack installation is happening, it might take 3 - 4 hrs to install all the packages. So, if the test is taking long time to finish, it is normal. However, this is done only once on a given system, generally on login nodes, before running actual tests.

Tip: Do not update module path using `module use <path>` command when there are multiple clusters with different micro architectures sharing common frontend. This can trigger module conflicts as modules generated for different micro architectures will have same name. For example, in the case of JUWELS supercomputer, cluster partition has Intel Skylake nodes whereas booster partition has AMD nodes. Packages will be built for Intel and AMD architectures separately and so if we have both module paths on `MODULEPATH`, a simple command like `module load gcc/9.3.0` will trigger conflicts or unintended behaviour as module system do not know which module to load. Module path is updated for each partition within benchmark tests to isolate the modules for that partition.

Important: The test will add environment variable `SPACK_ROOT` to the user's `$HOME/.bashrc`, if found. If bash is not the default shell of the user, it is essential to add `SPACK_ROOT` env variable to the profile. We use this variable inside the system configuration to add module path to the module system.

3.6.2 Script based approach

A bootstrap script is provided in `spack/build-spack-modules.bash` to build the entire environment to run SDP benchmark tests. The available options for the bootstrap are as follows:

```
Log is saved to: $REPO_ROOT/spack/spack_install.log
Usage: build-spack-modules.bash [OPTIONS]

OPTIONS:

-d | --dry-run              dry run mode
-s | --with-scheduler       Built OpenMPI with scheduler support
-i | --with-ib              Built UCX with IB transports
-o | --with-opa             Built OpenMPI with psm2 support (Optimized for Intel Omni-
↪path)
-c | --with-cuda            Build OpenMPI and UCX with cuda and other cuda related␣
↪packages
-t | --use-tcl              Use tcl module files [Default is lmod]
-h | --help                 prints this help and exits
```

The script can be run in dry mode to see the commands it will execute on the current system. If the system supports Infiniband, `-s` flag must be passed on CLI to build OpenMPI with IB support. Similarly, if the system has Intel Omni-

Path (OPA), `-o` must be passed. It is **not** possible to use both `-i` and `-o` at the same time as Spack supports only one fabrics type at a time.

Similarly, if there is a batch scheduler support on the system, use `-s` flag to enable scheduler support for OpenMPI. By default, the bootstrap script will build hierarchical modules to be used with `lmod` module system. If the platform supports only `tcl` module system, it must be passed at CLI using `-t` flag.

Finally, all the packages are installed using GCC 9.3.0 as compiler toolchain. This default can be overridden by setting environment variable `TOOLCHAIN` which will take precedence over default value. Similarly, the versions of all packages can be controlled by setting environment variables. More details can be found in the README file in `spack/` folder in the root of the repository.

An example usage for a system that has scheduler and IB supports would be:

```
cd spack
./build-spack-modules.bash -i -s -d  # For dry run
./build-spack-modules.bash -i -s  # For installing packages
```

Once the bootstrapping is finished, assuming all packages have installed without any errors, we need to source `$HOME/.bashrc` file as a final step to use the module files.

3.7 Deploy Spack environments

The following documentation shows how to create Spack environments and install packages for different systems using ReFrame tests.

CHAPTER

FOUR

FRAMEWORK PHILOSOPHY

4.1 Adding new tests

To add a new test to the benchmark suite, follow the following steps: 1. Define whether the test belongs into level 0, level 1 or level 2. Then create a folder in the corresponding location and add the following files:

- `reframe_<test_name>.py`: This is the main test file where we define the test class derived from the Benchmark base classes `BenchmarkBase`, `RunOnlyBenchmarkBase` or `CompileOnlyBenchmarkBase`. Make sure to use the one corresponding to your specific needs. Those classes are developed according to the ReFrame ideas documented in https://reframe-hpc.readthedocs.io/en/stable/tutorial_advanced.html#writing-a-run-only-regression-test.
- `TEST_NAME.ipynb`: Jupyter notebook to plot the performance metrics derived from the test
- `README.md`: A simple readme file that gives high level instructions on where to find the documentation of the test.

2. Define the test procedure:

    - Does the test need some sources or packages from the internet, be it its own sources, python packages or any other dependencies? If yes, create a test dependency that fetches everything. Every test or test dependency that accesses the internet needs to inherit from the `UsesInternet` mixin.

```
class IdgTestDownload(FetchSourcesBase):
    """Fixture to fetch IDG source code"""

    descr = 'Fetch source code of IDG'
```

    - Does the test need to compile fetched dependencies? If yes, create a test dependency that builds the sources. If the sources are fetched in a previous test, be sure to include this as a dependent fixture: `app_src = fixture(DownloadTest, scope='session')`.

```
class IdgTestBuild(CompileOnlyBenchmarkBase):
    """IDG test compile test"""

    descr = 'Compile IDG test from sources'

    # Share resource from fixture
```

(continues on next page)

(continued from previous page)

```
    idg_test_src = fixture(IdgTestDownload, scope='session')

    def __init__(self):
        super().__init__()
        self.valid_prog_environs = [
            'idg-test',
        ]
        self.valid_systems = filter_systems_by_env(self.valid_prog_environs)
        self.maintainers = [
            'Mahendra Paipuri (mahendra.paipuri@inria.fr)'
        ]
        # Cross compilation is not possible on certain g5k clusters. We␣
↪force
        # the job to be non-local so building will be on remote node
        if 'g5k' in self.current_system.name:
            self.build_locally = False

    @run_before('compile')
    def set_sourcedir(self):
        """Set source path based on dependencies"""
        self.sourcesdir = self.idg_test_src.stagedir

    @run_before('compile')
    def set_prebuild_cmds(self):
        """Make local lib dirs"""
        self.lib_dir = os.path.join(self.stagedir, 'local')
        self.prebuild_cmds = [
            f'mkdir -p {self.lib_dir}',
        ]

    @run_before('compile')
    def set_build_system_attrs(self):
        """Set build directory and config options"""
        self.build_system = 'CMake'
        self.build_system.builddir = os.path.join(self.stagedir, 'build')
        self.build_system.config_opts = [
            f'-DCMAKE_INSTALL_PREFIX={self.lib_dir}',
            '-DBUILD_LIB_CUDA=ON',
            '-DPERFORMANCE_REPORT=ON',
        ]
        self.build_system.max_concurrency = 8

    @run_before('compile')
    def set_postbuild_cmds(self):
        """Install libs"""
        self.postbuild_cmds = [
            'make install',
        ]

    @run_before('sanity')
```

3. Write the test itself.

- Define all dependencies as fixture, all parameters as *parameter* and all variables as *variable*. Tests are run for all permutations of parameters, whereas variables can define specific behaviour for a single run (like number of nodes).
- Set the *valid_prog_environs* and the *valid_systems* in the *__init__* method.
- Define the executable and executable options.
- Define the Sanity Patterns. You can define which patterns must and must not appear in the stdout and stderr.

```
            f'cd {self.executable_path}',
        ]

    @run_before('run')
    def post_launch(self):
        """Set post run commands. It includes removing visibility data␣
↪files and running
        read-test"""
        if self.variant == 'write-test':
            self.postrun_cmds.insert(0, 'rm -rf $SCRATCH_DIR_TEST/out%d.h5')
        elif self.variant == 'read-test':
            # run_command returns with default num tasks which is 1
            cmd = self.job.launcher.run_command(self.job).replace(str(1),␣
↪str(self.num_tasks_job))
            # The first job writes the visibility data and this job reads␣
↪them back.
            # This gives us the read bandwidth benchmark
            exec_opts = " ".join([*self.executable_opts[:-1], '--check-
↪existing',
                                  self.executable_opts[-1]])
            self.postrun_cmds.insert(0, f'{cmd} {self.executable} {exec_
↪opts}')
```

- Define the Performance Functions. To extract data from the output stream it is necessary to extract them using regular expressions.

```
        """Set sanity patterns. Example stdout:

        .. code-block:: text

            >>> Total runtime
            gridding:   6.5067e+02 s
            degridding: 1.0607e+03 s
            fft:        3.5437e-01 s
            get_image:  6.5767e+00 s
            imaging:    2.0073e+03 s

            >>> Total throughput
            gridding:   3.12 Mvisibilities/s
            degridding: 1.91 Mvisibilities/s
            imaging:    1.01 Mvisibilities/s

        """
        self.sanity_patterns = sn.all([
```

(continues on next page)

(continued from previous page)

```
            sn.assert_found('Total runtime', self.stderr),
            sn.assert_found('Total throughput', self.stderr),
        ])

    @performance_function('s')
    def extract_time(self, kind='gridding'):
        """Performance extraction function for time. Sample stdout:


        .. code-block:: text

            >>> Total runtime
            gridding:   7.5473e+02 s
            degridding: 1.1090e+03 s
            fft:        3.5368e-01 s
            get_image:  7.2816e+00 s
            imaging:    1.8899e+03 s

        """
        return sn.extractsingle(rf'^{kind}:\s+(?P<value>\S+) s', self.
↪stderr, 'value', float)

    @performance_function('Mvisibilities/s')
    def extract_vis_thpt(self, kind='gridding'):
        """Performance extraction function for visibility throughput.␣
↪Sample stdout:


        .. code-block:: text

            >>> Total throughput
            gridding:   2.69 Mvisibilities/s
            degridding: 1.83 Mvisibilities/s
            imaging:    1.07 Mvisibilities/s

        """
        return sn.extractsingle(rf'^{kind}:\s+(?P<value>\S+) Mvisibilities/s
↪', self.stderr, 'value', float)

    @run_before('performance')
    def set_perf_patterns(self):
        """Set performance variables"""
        self.perf_variables = {
            'gridding s': self.extract_time(),
            'degridding s': self.extract_time(kind='degridding'),
            'fft s': self.extract_time(kind='fft'),
            'get_image s': self.extract_time(kind='get_image'),
            'imaging s': self.extract_time(kind='imaging'),
            'gridding Mvis/s': self.extract_vis_thpt(),
            'degridding Mvis/s': self.extract_vis_thpt(kind='degridding'),
            'imaging Mvis/s': self.extract_vis_thpt(kind='imaging'),
        }
```

(continues on next page)

(continued from previous page)

```
    @run_before('performance')
    def set_reference_values(self):
        """Set reference perf values"""
        self.reference = {
            '*': {
                '*': (None, None, None, 's'),
                '*': (None, None, None, 'Mvis/s'),
            }
        }
```

The sanity- and performance functions are both based on the concept of "Deferrable Functions". Be sure to check out the official documentation on how to use them properly.

Those steps allow you to write a basic ReFrame test. For more in-detail view, take a look at the ReFrame documentation. There is no strict convention on how to name the test. Already provided tests can be used as templates to write new tests. The idea is to provide an environment for a given test and define all the test related variables like modules to load, environment variables to define within this environment. Also, we need to add the `target_systems` to this environments on the systems that we would like to run these tests. The details of adding a new environment and system are presented below.

4. Write a unit test procedure. You can take for example the test class for `CondaEnvManager` located in `unittests/test_conda_env_manager.py`.

   - Create a file test with a class test for your benchmark :

```
import unittest
import modules.conda_env_manager as cem
import os
import logging

# INFO ERROR DEBUG WARNING
# logging.basicConfig(level=logging.DEBUG)
LOGGER = logging.getLogger(__name__)


class CondaEnvManagerTest(unittest.TestCase):
```

   - Add a test method for each functionnality :

```
    def test_create(self):
        logging.info("CondaEnvManagerTest")
        logging.info("---------------------\n")
        logging.info("test_create")
        test = cem.CondaEnvManager("test_create", False)
        test.remove()
        self.assertTrue(test.create())
        self.assertFalse(test.create())
        test.remove()
```

   - Check results via the `self.assertTrue` and `self.assertFalse` methods (*unittest doc <https://docs.python.org/3/library/unittest.html>*) :

```
        self.assertTrue(test.create())
        self.assertFalse(test.create())
```

- Add the name of your unit test file (without the extension) in the list of unit tests to check in the `sdp-unittest.sh` script

```
        self.assertTrue(test.create())
        self.assertFalse(test.create())
```

We recommand to use systematically the `logging` module (*logging doc <https://docs.python.org/3/library/logging.html>*) which provides 6 custom level of screen outputs (e.g. `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`, `NOSET`).

To perform our unit test procedure, we provide an experimental bash script `sdp-unittest.sh` to launch our different unit tests automatically with several available options. We use the package `pytest` (*pytest doc <https://docs.pytest.org/en/7.1.x/contents.html>*) to purpose a parallelized and efficient procedure for the unit test step.

4.2 Adding new system

Every time we want to add a new system, typically we will need to follow these steps:

- Create a new python file `<system_name>.py` in `config/systems` folder.
- Add system configuration and define partitions for the system. More details on how to define a partition and naming conventions are presented later.
- Import this file into `reframe_config.py` and add this new system in the `site_configuration`.
- The final step would be get the processor info using `--detect-host-topology` option on ReFrame of system nodes, place in the `toplogies` folder and include the file in `processor` key for each partition.

The user is advised to consult the ReFrame documentation before doing so. The provided systems can be used as a template to add new systems.

We try to follow a certain convention in defining the system partition. Firstly, we define partitions, either physical or abstract, based on compiler toolchain and MPI implementation such that when we use this system, modules related to compiler and MPI will be loaded. Rest of the modules that are related to test will be added to the `environs` which will be discussed later. Consequently, we should also name these partitions in such a way that we can have a standard scheme. The benefit of having such a scheme is two-fold: able to get high level overview of partition quickly and by choosing an appropriate names, we can filter the systems for the tests easily. An example use case is that we want to run a certain test on all partitions that support GPUs. Using a partition name with `gpu` as suffix, we can simply filter all the partitions looking for a match with string `gpu`.

We use the convention `{prefix}-{compiler-name-major-ver}-{mpi-name-major-ver}-{interconnect-type}-{software-`

- `Prefix` can be name of the partition or cluster.
- **`compiler-name-major-ver` can be as follows:**
  - `gcc9`: GNU compiler toolchain with major version 9
  - `icc20`: Intel compiler toolchain with major version 2020
  - `xl16`: IBM XL toolchain with major version 16
  - `aocc3`: AMD AOCC toolchain with major version 3
- **`mpi-name-major-ver` is the name of the MPI implementation. Some of them are:**
  - `ompi4`: OpenMPI with major version 4

    - `impi19`: Intel MPI with major version 2019
    - `pmpi5`: IBM Spectrum MPI with major version 5
    - `smpi10`: IBM Spectrum MPI with major version 10

- **`interconnect-type` is type of interconnect on the partition.**
    - `ib`: Infiniband
    - `rocm`: RoCE
    - `opa`: Intel Omnipath
    - `eth`: Ethernet TCP
- **`software-type` is type of software stack used.**
    - `smod`: System provided software stack
    - `umod`: User built software stack using Spack
- `suffix` can indicate any special properties of the partitions like gpu, high memory nodes, high priority job queues, *etc*. There can be multiple suffices each separated by a hyphen.

Important: If the package uses calendar versioning, we use only last two digits of the year in the name to be concise. For example, Intel MPI 2019.* would be `impi19`.

For instance, in the configuration shown in *ReFrame configuration* `compute-gcc9-ompi4-roce-umod` tells us that the partition has GCC compiler with OpenMPI. It uses RoCE as interconnect and the softwares are built in user space using Spack.

Important: It is recommended to stick to this convention and there can be more possibilities for each category which should be added as we add new systems.

4.3 Adding new environment

Adding a new system is not enough to run the tests on this system. We need to tell our ReFrame tests that there is a new system available in the config. In order to minimise the redundancy in adding configuration details and avoid modifying the source code of the test, we choose to provide a `environ` for each test. For example, there is HPL test in `apps/level0/hpl` folder and for this test we define a environ in `config/environs/hpl.py`.

Note: System partitions and environments should have one-to-one mapping. It means, whatever environment we define within `environs` section in the system partition, we should put that partition within `target_systems` in each `environ`.

All the modules that are needed to run the test, albeit compiler and MPI, will be added to the `modules` section in each `environ`. For example, lets take a look at `hpl.py` file

```
""""This file contains the environment config for HPL benchmark"""


hpl_environ = [
    {
```

(continues on next page)

(continued from previous page)

```
        'name': 'intel-hpl',
        'cc': 'mpicc',
        'cxx': 'mpicxx',
        'ftn': 'mpif90',
        'modules': [
            'intel-oneapi-mkl/2021.3.0',
        ],
        'variables':[
            ['XHPL_BIN', '$MKLROOT/benchmarks/mp_linpack/xhpl_intel64_dynamic'],
        ],
        'target_systems': [
            'alaska:compute-icc21-impi21-roce-umod',
            # <end - alaska partitions>
            'grenoble-g5k:dahu-icc21-impi21-opa-umod',
            # <end - grenoble-g5k partitions>
            'juwels-cluster:batch-icc21-impi21-ib-umod',
            # <end juwels partitions>
            'nancy-g5k:gros-icc21-impi21-eth-umod',
            # <end - nancy-g5k partitions>
            'cscs-daint:daint-icc21-impi21-ib-umod-gpu',
            # <end - cscs partitions>
        ],
    },
    {
        'name': 'gnu-hpl',
        'cc': '',
        'cxx': '',
        'ftn': '',
        'modules': [
            'amdblis/3.0',
        ],
        # 'variables': [
        #     ['UCX_TLS', 'ud,rc,dc,self']
        # ],
        'target_systems': [
            'juwels-booster:booster-gcc9-ompi4-ib-umod',
            # <end juwels partitions>
        ],
    },
{
        'name': 'intel-hpl',
        'cc': 'mpicc',
        'cxx': 'mpicxx',
        'ftn': 'mpif90',
        'modules': [
            'intel-oneapi-mkl/2023.0.0',
        ],
        'variables':[
            ['XHPL_BIN', '$MKLROOT/benchmarks/mp_linpack/xhpl_intel64_dynamic'],
        ],
        'target_systems': [
            'cscs-daint:daint-icc21-impi21-ib-umod-gpu',
```

(continues on next page)

(continued from previous page)

```
            # <end - cscs partitions>
        ],
    },
]
```

There are two different environments namely `intel-hpl` and `gnu-hpl`. As names suggests, `intel-hpl` uses HPL benchmark shipped out of MKL optimized for Intel chips. Whereas we use `gnu-hpl` for other chips like AMD using GNU toolchain. Notice that `target_systems` for `intel-hpl` has only partitions that have Intel MPI implementation (`impi` in the name) whereas the `gnu-hpl` has `target_systems` have OpenMPI implementation. Within the test, we define only the `valid program` and find valid systems by filtering all systems that have the given environment defined for them.

For instance, we defined a new system partition that has Intel chip with name as `mycluster-gcc-impi-ib-umod`. If we want HPL test to run on this system partition, we add `intel-hpl` to `environs` section in system partition and similarly add the name of this partition to `target_systems` in `intel-hpl` environment. Once we do that, the test will run on this partition without having to modify anything in the source code of the test.

If we want to add a new test, we will need to add new environment and following steps should be followed:

- Create a new file `<env_name>.py` in `config/environs` folder and add environment configuration for the tests. **It is important** that we add this new environment to existing and/or new system partitions that are defined in `target_systems` of the environment configuration.
- Finally, import `<env_name>.py` in the main ReFrame configuration file `reframe_config.py` and add it to the configuration dictionary.

4.4 Adding Spack configuration test

After we define a new system, we need software stack on this system to be able to run tests on it. If the user chooses to use platform provided software stack, this step can be skipped. We need to define Spack config files in order to deploy the software stack. We can user existing config files provided for different systems as a base. Typically, we should only change `compilers.yml`, `modules.yml` and `spack.yml` files for new system. We need to update the system compiler version and their paths in `compilers.yml` and also in `modules.yml` file in `core_compilers` section. Similarly, the desired software stack that will be installed on the system is defined in `spack.yml` file.

Once these configuration files are ready, we need to create a new folder in `spack/spack_tests` folder with name of the system and place all configuration files in `configs/` and define a ReFrame test to deploy this software stack. The user can use the existing test files as template. The ReFrame test file *per se* is very minimal and user needs to put the name of the cluster and path where Spack must be installed in the test body.

4.5 Adding Conda environment via CondaEnvManager

CondaEnvManager is a class to provide a specific conda environment for each test (benchmark) we want to create. The advantage is we remove the different problem of dependancies we could have with the ska-sdp-benchmark environment (e.g. a different version of numpy package). For the moment, we provide this approach only for the `level0/cpu/fft` test and for the associated unit test. We will use the unit test example as a basis below.

Note : The `self.assertTrue()` and `self.assertFalse()` *are just unit test methods.*

4.5.1 Enable the module

```
import modules.conda_env_manager as cem
```

4.5.2 Create a conda environment

When you instance your class, you don't create directly a conda environement with the name you provide :

```
        test = cem.CondaEnvManager("test_remove", False)
        test.create()
```

4.5.3 Remove a conda environment

Remove the created environment and all packages installed.

```
        test.remove()
```

4.5.4 Install a package via pip

The `install_pip` method provide a generic method which is able to download and install directly a package or install from a local folder a package if the package already exists (yet downloaded). You can also provide a `requirement.txt` file directory to install directly multiple packages (e.g. *Download and install manually a list of packages via pip*).

```
        test = cem.CondaEnvManager("test_install_pip_package", False)
        test.create()
        self.assertTrue(test.install_pip("zipp"))
```

4.5.5 Download locally a package via pip

Download locally in a cache folder packages by environment. It is usefull if you have not a connection on a calculation cluster :

```
        test = cem.CondaEnvManager("test_download_install_local_pip_package", False)
        test.create()
        self.assertTrue(test.download_pip("zipp"))
```

4.5.6 Download and install locally a list of packages via pip

We can download via a `requirement.txt` file a list a packages for an environment. Need to provide the directory of your requirement file for the download and installations steps :

```
        test = cem.CondaEnvManager("test_install_download_local_pip_file", False)
        test.create()
        root_dir = os.path.dirname(__file__)
        requirements_true = os.path.join(root_dir, 'resources', 'requirements.txt')
        requirement_false = os.path.join(root_dir, 'resources', 'requirements_false.
```

(continues on next page)

(continued from previous page)

```
↪txt')
        self.assertTrue(test.download_pip(requirements_true))
        self.assertFalse(test.download_pip(requirement_false))
        self.assertTrue(test.install_pip(requirements_true, True))
        self.assertFalse(test.install_pip(requirement_false, True))
```

4.5.7 Purge the pip cache

Remove packages in the cache folder for a created environment.

4.6 Separation of Parts that Access the Internet

Any test or test dependency that accesses the internet needs to implement the mixin `UsesInternet`. This mixin ensures that parts that need to download assets from the internet are run on a partition where internet can be accessed. Any test that accesses the internet and does not implement the `UsesInternet` mixin is **not guaranteed** to have internet available and might fail.

Best practice is to separate all dependencies that need to access the web into dependencies, be it dataset downloads, sources downloads or package installations that fetch from repositories. Those dependencies must implement the `UsesInternet` mixin and should be defined as fixtures for the test.

CHAPTER

FIVE

INSTALLATION AND RUNNING OF SKA SDP BENCHMARK TESTS

5.1 Installation instructions

ReFrame is developed in Python and hence, Python 3.6+ and several python modules are needed to run the tests. We **strongly** recommend to use conda environment to create virtual environments. Installation instructions for Spack is provided separately here in *Installation* section.

1. Clone this repository:

```
git clone https://gitlab.com/ska-telescope/sdp/ska-sdp-benchmark-tests.git
```

2. Install conda (if not already installed):

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

More instructions on installing conda can be found here

3. Create and activate `ska-sdp-benchmark-tests` environment. The conda environment files are placed in folder `share/conda/` in the root of the repository. Currently only x86 and ppc64le architectures are supported. For instance, if the system under test has x86 architecture, we can create conda environment using:

```
cd ska-sdp-benchmark-tests
conda env create -f share/conda/environment-x86.yml
conda activate ska-sdp-benchmark-tests
```

4. Install all dependencies (like ReFrame) using the *share/setup-env.sh* script:

```
source /path/to/ska-sdp-benchmark-tests/share/setup-env.sh
```

This script installs the correct ReFrame version, wraps the reframe binary around a shell function and modifies the `PYTHONPATH` appropriately to get all modules on the path. Once this environment is set, we can invoke `reframe` from anywhere and run tests by providing an absolute path to test files. It also fetches and installs the SDP Perfmon toolkit, compiles it and installs it into the current conda environment. If the perfmon is installed, this script requires a working compiler toolchain and CMake installation. If you don't want to install the perfmon, set `"INSTALL_PERFMON=no"` before sourcing the script.

Note: The installation of SDP Perfmon needs CMake >= 3.17.5.

5.2 Running SKA SDP Benchmark tests

As stated in the *Installation instructions*, once we set up environment for SKA SDP Benchmark tests using the script `setup-env.sh`, we can invoke reframe from anywhere in the system. All the documentation provided in the following subsections assume that this environment is not set up and hence all tests are invoked from repository root. For example, if a test `mytest.py` is located at an absolute path `/path/to/mytest.py`, we can execute that test using either:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
source share/setup-env.sh
reframe -c /path/to/mytest.py -r
```

CHAPTER

SIX

PERFORMANCE METRICS

ReFrame provides the performance logs at the end of successful tests. These performance metrics are the ones that are instrumented within the code or benchmark. In the case of microbenchmarks, these tend to be metric of interest like Gflops, memory bandwidth, latencies, *etc*. Whereas for the application benchmarks, they tend to be higher level metrics like CPU wall time.

In order to optimise the codes, we need more low level metrics than just CPU wall time. One way to obtain these low level metrics is by profiling the code. However, profiling has very high overheads. We can use a solution that falls between these two extremes by monitoring several CPU, memory and network related metrics as the benchmark tests run. A time series data of such metrics can be used to identify the bottlenecks and hotspots relatively quickly and give some insights to developers about performance of codes.

A toolkit implemented to extract several CPU metrics like usage, time, memory consumption, bandwidth, *etc*. It also monitors the low-level metrics from `perf stat` command like FLOPS, L2/L3 bandwidth. In the future, `perf record` profiling output will also be added to the toolkit such that we get a lower level profiling of the code. More details on the toolkit and the metrics it reports can be consulted in the documentation.

Note: We mainly collect these performance metrics for level 1 and level 2 benchmarks (but not exclusively). Typically, level 0 benchmarks uses mini kernels and the performance metrics are already instrumented inside the benchmark code. On the other hand, level 1 and level 2 benchmarks are more complex and application oriented and time series data of several performance measures are desirable to understand the bottlenecks of the code.

Currently, all the metrics recorded from each benchmark are saved in a HDF5 format. It can be found at `perfmetrics/{system}/{partition}/{environment}/{test}/test.h5`. Each run will create two tables in the HDF store with names as `cpu_metrics_<job_id>` and `perf_metrics_<job_id>`, where `job_id` is the ID of the batch scheduler job. These tables can be imported to a Pandas dataframe for plotting and other post-processing.

CHAPTER

SEVEN

LEVEL 0 BENCHMARK TESTS

7.1 CPU tests

7.2 GPU tests

CHAPTER

EIGHT

LEVEL 1 BENCHMARK TESTS

CHAPTER

NINE

LEVEL 2 BENCHMARK TESTS

CHAPTER

TEN

ORGANISATION OF REPOSITORY

10.1 Core

- The source code of benchmark tests can be found in the `apps/` folder. This folder is divided into three sub-folders `level0/`, `level1/` and `level2/`. These folders are further sub-divided where each benchmark test is placed into a folder named after it. Inside each benchmark folder, we typically find three different files namely, `reframe_<testname>.py`, `README.md` and `TESTNAME.ipynb`, where `<testname>` is the name of the benchmark. The logic of the test is included in `reframe_<testname>.py`, details about the documentation can be found in `README.md` and finally, `TESTNAME.ipynb` can be used to plot and tabulate benchmark results.
- All the configuration related files are placed in `config/` folder and they are imported into `reframe_config.py` file. We find two sub-folders in `config/` folder namely, `systems/` and `environs/`. All the system and partitions are defined in `systems/` folder and environments are defined in `environs/` folder. We keep one file for a given platform and define all the partitions of that platform in that file. Similarly, place a environment file for a given test and define all the environments for that test in the file.
- All the utility functions and extra classes defined out of ReFrame are placed in `modules/` folder.
- ReFrame tests use topological information of the platforms like number of sockets, cores, *etc*. Topological information of different platforms are placed in `topologies/` folder. These files are imported and added during configuration of system partitions.

10.2 Documentation

All the documentation is provided in the `docs/` folder. The sources can be found at `docs/src/content/` folder where each section of documentation has a folder. Within each folder, `rst` files can be found that contain core documentation.

10.3 Misc

- To deploy software stack using Spack, the bootstrap script is provided in `spack/` folder. There is a `README` file in the folder with the instructions on how to use the bootstrap script.
- The folder `perflogs/` contain all the performance metrics that are extracted for different benchmarks for different systems and partitions. These logs are used in plotting and tabulating the performance data in the Jupyter notebooks for each test.
- The folder `helloworld/` contains a simple ReFrame test taken from ReFrame documentation demonstrating the use case of its framework.
- The folder `.ci/` contains all the CI related files. It includes dockerfile to build image and all the auxiliary files used inside docker image.

CHAPTER

ELEVEN

DOCUMENTATION OF MODULES

Utility functions for SDP benchmark tests

class modules.utils.**GenerateHplConfig**(*num_nodes*, *num_procs*, *nb*, *mem*, *alpha=0.8*)

Class to generate HPL.dat file for LINPACK benchmark

estimate_n()

Estimate N based on memory and alpha

estimate_pq()

Get best combination of P and Q based on nprocs

get_config()

Write HPL.dat file to outfile location

modules.utils.**sdp_benchmark_tests_root**()

Returns the root directory of SKA SDP Benchmark tests

Returns
Path of the root directory

Return type
str

modules.utils.**parse_path_metadata**(*path*)

Return a dict of reframe info from a results path

Parameters
path – ReFrame stage/output path

Returns
ReFrame system/partition info

Return type
dict

modules.utils.**find_perf_logs**(*root*, *benchmark*)

Get perflog file names for given test

Parameters

- **root** – Root where perflogs exist
- **benchmark** – Name of the benchmark

Returns
List of perflog file names

Return type
list

`modules.utils.`**`read_perflog`**(*path*)

Return a pandas dataframe from a ReFrame performance log. NB: This currently depends on having a non-default handlers_perflog.filelog.format in reframe's configuration. See code. The returned dataframe will have columns for:

- all keys returned by *parse_path_metadata()*
- all fields in a performance log record, noting that: - 'completion_time' is converted to a *datetime.datetime* - 'tags' is split on commas into a list of strs
- 'perf_var' and 'perf_value', derived from 'perf_info' field
- <key> for any tags of the format "<key>=<value>", with values converted to int or float if possible

Parameters
`path` (*`str`*) – Path to log file

Returns
Dataframe of perflogs

Return type
pandas.DataFrame

`modules.utils.`**`load_perf_logs`**(*root='.'*, *test=None*, *extras=None*, *last=False*, *aggregate_multi_runs=<function median>*)

Convenience wrapper around read_perflog().

Parameters

- **`root`** (*`str`*) – Path to root of tree containing perf logs
- **`test`** (*`str`*) – Shell-style glob pattern matched against last directory component to restrict loaded logs, or None to load all in tree
- **`extras`** (*`list`*) – Additional dataframe headers to add
- **`last`** (*`bool`*) – True to only return the most-recent record for each system/partition/enviroment/testname/perf_var combination.
- **`aggregate_multi_runs`** (*`Callable`*) – How to aggregate the perf-values of multiple runs. If None, no aggregation is applied. Defaults to np.median

Returns
Single pandas.dataframe concatenated from all loaded logs, or None if no logs exist

Return type
pandas.DataFrame

`modules.utils.`**`tabulate_last_perf`**(*test*, *root='../../perflogs'*, *extras=None*, ***kwargs*)

Retrieve last perf_log entry for each system/partition/environment.

Parameters

- **`test`** (*`str`*) – Shell-style glob pattern matched against last directory component to restrict loaded logs, or None to load all in tree
- **`root`** (*`str`*) – Path to root of tree containing perf logs of interest - default assumes this is called from an *apps/<application>/* directory
- **`extras`** (*`list`*) – Additional dataframe headers to add

Returns
A dataframe with columns: - case: name of the system, partition and environ - perf_var: Performance variable - add_var: Any additional variable passed as argument

Return type
pandas.DataFrame

`modules.utils.`**`tabulate_partitions`**`(`*root*`)`

Tabulate the list of partitions defined with ReFrame config file and high level overview of each partition. We tabulate only partitions that are found in the perflog directory

Parameters
root (*str*) – Perflog root directory

Returns
A dataframe with all partition details

Return type
pandas.DataFrame

`modules.utils.`**`filter_systems_by_name`**`(`*patterns*`)`

Filter systems based on patterns in the name. If all patterns are found in the name, the system is chosen.

Parameters
patterns (*list*) – List of patterns to be searched

Returns
List of partitions that match the pattern

Return type
list

`modules.utils.`**`filter_systems_by_env`**`(`*envs*`)`

Filter systems based on valid environments defined for them.

Parameters
envs (*list*) – List of environments to be searched

Returns
List of partitions that match the envs

Return type
list

`modules.utils.`**`git_describe`**`(`*dir*`)`

Return a string describing the state of the git repo in which the dir is. See *git describe –dirty –always* for full details.

Parameters
dir (*str*) – Root path of git repo

Returns
Git describe output

Return type
str

`modules.utils.`**`generate_random_number`**`(`*n*`)`

Generate random integer of n digits

Parameters
n (*int*) – Length of the desired random number

Returns
Generated random number

Return type
int

`modules.utils.`**`get_scheduler_env_list`**`(`*scheduler_name*`)`

Return the environment variables that stores different job details of different schedulers

Parameters
`scheduler_name` (*`str`*) – Name of the workload scheduler

Returns
Environment variables dict

Return type
dict

`modules.utils.`**`emit_conda_init_cmds`**`()`

This function emits the command to initialize conda. It temporarily clears the PYTHONPATH, so that no pre-installed dependencies from external or external/perfmon are used. # todo: test if this works even with perfmon

`modules.utils.`**`emit_conda_env_cmds`**`(`*env_name*, *py_ver='3.8'*`)`

This function emits all the commands to create/activate a conda environment. This function assumes conda is installed in the system

Parameters

- **`env_name`** (*`str`*) – Name of the conda env to create/activate
- **`py_ver`** (*`str`*) – Version of python to be used in conda environment (Default: 3.8)

Returns
List of commands to create/activate conda env

Return type
list

`modules.utils.`**`merge_spack_configs`**`(`*input_file*, *output_file*`)`

This function merges all spack config files by replacing *include* keyword with respective yaml file

Parameters

- **`input_file`** – Path to input spack.yaml file
- **`output_file`** – Path to output merged spack.yaml file

Returns
None

CHAPTER TWELVE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX