developer.skatelescope.org Documentation

SKA SDP Developers

Feb 02, 2023

HOME

1 Getting Started **3**
- 1.1 Context . . . 3
- 1.2 Currently supported benchmarks . . . 3
- 1.3 Software stack . . . 4

2 Overview of ReFrame **5**
- 2.1 Introduction . . . 5
- 2.2 ReFrame configuration . . . 5
- 2.3 ReFrame usage . . . 12

3 Overview of Spack **19**
- 3.1 Introduction . . . 19
- 3.2 Installation . . . 19
- 3.3 Usage . . . 19
- 3.4 Installing packages . . . 21
- 3.5 Module files . . . 22
- 3.6 Setting up environment for SKA SDP Benchmark tests . . . 24
- 3.7 Deploy Spack environments . . . 27

4 Framework philosophy **31**
- 4.1 Adding new test . . . 31
- 4.2 Adding new system . . . 35
- 4.3 Adding new environment . . . 36
- 4.4 Adding Spack configuration test . . . 38

5 Installation and Running of SKA SDP Benchmark tests **39**
- 5.1 Installation instructions . . . 39
- 5.2 Running SKA SDP Benchmark tests . . . 40

6 Performance metrics **41**

7 Level 0 Benchmark Tests **43**
- 7.1 CPU tests . . . 43
- 7.2 GPU tests . . . 59

8 Level 1 Benchmark Tests **71**
- 8.1 CUDA NIFTY gridder performance benchmark . . . 71
- 8.2 IDG Test . . . 73
- 8.3 Imaging IO Test . . . 76

9 Level 2 Benchmark Tests **85**

9.1 RASCIL 85

10 Organisation of repository **91**
10.1 Core 91
10.2 Documentation 91
10.3 Misc 91

11 Documentation of modules **93**

12 Indices and tables **103**

Python Module Index **105**

Index **107**

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': '1152348640000',  # 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:

```
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 de-

fault 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.

3.7.1 AlaSKA

The packages in `spack/spack_tests/alaska/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/alaska/reframe_alaska.py --
↪run
```

class spack.spack_tests.alaska.reframe_alaska.**AlaskaSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on AlaSKA Cluster

3.7.2 Dahu - Grenoble - Grid5000

The packages in `spack/spack_tests/grid5000/grenoble/dahu/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/grid5000/grenoble/dahu/
↪reframe_dahug5k.py --run
```

class spack.spack_tests.grid5000.grenoble.dahu.reframe_dahug5k.**G5kDahuSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on dahu cluster on Grid5000 at Grenoble site

3.7.3 Troll - Grenoble - Grid5000

The packages in `spack/spack_tests/grid5000/grenoble/troll/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/grid5000/grenoble/troll/
↪reframe_trollg5k.py --run
```

class spack.spack_tests.grid5000.grenoble.troll.reframe_trollg5k.**G5kTrollSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on troll cluster on Grid5000 at Grenoble site

3.7.4 Gemini - Lyon - Grid5000

The packages in `spack/spack_tests/grid5000/lyon/gemini/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/grid5000/lyon/gemini/
↪reframe_geminig5k.py --run
```

class spack.spack_tests.grid5000.lyon.gemini.reframe_geminig5k.**G5kGeminiSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on gemini cluster on Grid5000 at Lyon site

3.7.5 Gros - Nancy - Grid5000

The packages in `spack/spack_tests/grid5000/gros/nancy/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/grid5000/gros/nancy/
↪reframe_geminig5k.py --run
```

class spack.spack_tests.grid5000.nancy.gros.reframe_grosg5k.**G5kGrosSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on gros cluster on Grid5000 at Nancy site

3.7.6 Grouille - Nancy - Grid5000

The packages in `spack/spack_tests/grid5000/nancy/grouille/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/grid5000/nancy/grouille/
↪reframe_geminig5k.py --run
```

class spack.spack_tests.grid5000.nancy.grouille.reframe_grouilleg5k.**G5kGouilleSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on grouille cluster on Grid5000 at Nancy site

3.7.7 JUWELS Cluster

The packages in `spack/spack_tests/juwels/cluster/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/juwels/cluster/reframe_
↪juwelscluster.py --run
```

class spack.spack_tests.juwels.cluster.reframe_juwelscluster.**JClusterSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on JUWELS Cluster

3.7.8 JUWELS Booster

The packages in `spack/spack_tests/juwels/booster/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/juwels/booster/reframe_
↪juwelsbooster.py --run
```

class spack.spack_tests.juwels.booster.reframe_juwelsbooster.**JBoosterSpackEnv**(**args*, ***kwargs*)

Test to create Spack env on JUWELS Booster

3.7.9 Marconi100

The packages in `spack/spack_tests/marconi100/configs/spack.yml` can be installed using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c spack/spack_tests/marconi100/reframe_
↪juwelsbooster.py --run
```

class spack.spack_tests.marconi100.reframe_marconi100.**Marconi100SpackEnv**(**args*, ***kwargs*)

Test to create Spack env on Marconi100 cluster

CHAPTER

FOUR

FRAMEWORK PHILOSOPHY

4.1 Adding new test

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 ReFrame Regression Test class
- `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:

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

    descr = 'Fetch source code of IDG'
    sourcesdir = 'https://git.astron.nl/RD/idg.git'
    cnd_env_name = CONDA_ENV_NAME
```

    - 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(rfm.CompileOnlyRegressionTest):
    """IDG test compile test"""

    descr = 'Compile IDG test from sources'

    # Share resource from fixture
    idg_test_src = fixture(IdgTestDownload, scope='session')

    def __init__(self):
        self.valid_prog_environs = [
            'idg-test',
        ]
        self.valid_systems = filter_systems_by_env(self.valid_prog_environs)
```

(continues on next page)

(continued from previous page)

```
        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')
    def set_sanity_patterns(self):
        """Set sanity patterns"""
        self.sanity_patterns = sn.assert_not_found('error', self.stderr)
```

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.

```
    @run_before('sanity')
    def set_sanity_patterns(self):
        """Set sanity patterns. Example stdout:

        .. code-block:: text

            # Fri Jul 23 15:15:41 2021[1,0]<stdout>:Operations:

        We check number of time the above line is printed and compare it␣
↪with number of
        sub grid workers
        """
        num_messages = sn.len(sn.findall(r'(.*):(\s*)Operations', self.
↪stdout))
        self.sanity_patterns = sn.assert_eq(
            num_messages,
            self.num_nodes * self.num_sm[self.variant] *
            self.benchmark[self.size]['subgrid-workers']
        )
```

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

```
    @run_before('sanity')
    def set_sanity_patterns(self):
        """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([
            sn.assert_found('Total runtime', self.stderr),
            sn.assert_found('Total throughput', self.stderr),
        ])

    @performance_function('s')
    def extract_time(self, kind='gridding'):
```

(continues on next page)

(continued from previous page)

```
        """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'),
        }

    @run_before('performance')
    def set_reference_values(self):
        """Set reference perf values"""
        self.reference = {
            '*': {
```

(continues on next page)

(continued from previous page)

```
                '*': (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.

4.2 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>
        ],
    },
]
```

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.

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.

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

7.1.1 HPCG benchmark

Context

This is HPCG microbenchmark test with a default problem size of 104. The benchmark does several matrix vector operations on sparse matrices. More details about the benchmark can be found at HPCG benchmark website.

Note: Currently, the implemented test uses the optimized version of benchmark shipped by Intel MKL library. We use GNU compiler toolchain for running the test on AMD processors. For the case of IBM POWER processors, IBM shipped XL compiler toolchain is used to run the benchmark

Test size

Currently, two different variables can be controlled for running tests. They are

- number of nodes to run the benchmark
- problem size

By default, benchmark will run on single node. If the user wants to run on multiple nodes, we can set the `num_nodes` variable at the run time. Similarly, the default problem size is 104 and it can be set at runtime using `problem_size` variable.

Note: Even if more than one node is used in the test, the resulting performance metric, `Gflop/s`, is always reported per node.

Note: The problem size must be a multiple of 8. This is the requirement from the HPCG benchmark *per se*.

Test types

Currently there are three different types of tests implemented:

- `HpcgXlTest`: HPCG with IBM XL toolchain
- `HpcgGnuTest`` : HPCG with GNU GCC toolchain
- `HpcgMklTest`: HPCG shipped with Intel MKL package

If a system has two valid tests, we can restrict the test using `-n` flag on the CLI. It is shown in *Usage*.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/hpcg/reframe_hpcg.py --run --
↪performance-report
```

We can set number of nodes on the CLI using:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/hpcg/reframe_hpcg.py --run --
↪performance-report -S num_nodes=2
```

Similarly, problem size of the benchmark can be altered at runtime using:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/hpcg/reframe_hpcg.py --run --
↪performance-report -S problem_size=120
```

For instance, if a system has both `HpcgGnuTest` and `HpcgMklTest` as valid tests and if we want to run only `HpcgMklTest`, we can use `-n` flag as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/hpcg/reframe_hpcg.py --run --
↪performance-report -n HpcgMklTest
```

Test class documentation

class apps.level0.cpu.hpcg.reframe_hpcg.**HpcgMixin**(**args*, ***kwargs*)

Common regression test attributes for HpcgGnuTest and HpcgMklTest

test_settings()

Common test settings

set_git_commit_tag()

Fetch git commit hash

set_executable_opts()
Set executable options

export_env_vars()
Export env variables using OMPI_MCA param for OpenMPI

set_tags()
Add tags to the test

set_keep_files()
List of files to keep in output

set_sanity_patterns()
Set sanity patterns. Example stdout:

extract_gflops()
Performance extraction function for Gflops

set_perf_patterns()
Set performance variables

class apps.level0.cpu.hpcg.reframe_hpcg.**HpcgXlTest**(**args*, ***kwargs*)
Main class of HPCG test based on IBM Xl

build_executable()
Set build system and config options

set_num_tasks_threads()
Set number of MPI and OpenMP tasks

set_executable()
Set executable

set_outfile()
Set name of output file

job_launcher_opts()
Set job launcher options

class apps.level0.cpu.hpcg.reframe_hpcg.**HpcgGnuTest**(**args*, ***kwargs*)
Main class of HPCG test based on GNU

set_num_tasks()
Set number of tasks for job

set_prebuild_cmds()
Copy Make file into setup folder

build_executable()
Set build system and options

set_executable()
Set executable

set_outfile()
Set name of output file

job_launcher_opts()

Set job launcher options

class apps.level0.cpu.hpcg.reframe_hpcg.**HpcgMklTest**(**args*, ***kwargs*)

Main class of HPCG test based on MKL

set_num_tasks()

Set number of tasks for job

set_env_vars()

Set job specific env variables

set_executable()

Set executable

set_executable_opts()

Override executable options from Mixin class

set_output_file()

Set output file name

set_mkl_env()

Source the env vars to get all necessary libraries on PATH

7.1.2 HPL benchmark

Context

This is HPL microbenchmark test with using a single node as default test parameter. It is used as reference benchmark to provide data for the Top500 list and thus rank to supercomputers worldwide. HPL rely on an efficient implementation of the Basic Linear Algebra Subprograms (BLAS).

Note: Currently, the implemented test uses the optimized version of benchmark shipped by Intel MKL library.

Test types

Currently, two different tests are defined namely,

- `HplGnuTest`: Based on GNU toolchain for non Intel processors
- `HplMklTest`: Using benchmark shipped out of Intel MKL library for Intel processors

On Intel chips, we can use the precompiled binary that comes out-of-the-box from Intel MKL library. But for non Intel systems, we need to compile using GNU tool chain with customized make file. In the directory `makes/`, we provide the make file for AMD chips using BLIS as BLAS library. We can choose which test to run during runtime using CLI which is discussed in *Usage*.

Test configuration file

The prerequisite to run HPL benchmark is `HPL.dat` file that contains several benchmark parameters. A sample configuration file looks like

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
4            # of problems sizes (N)
29 30 34 35  Ns
4            # of NBs
1 2 3 4      NBs
0            PMAP process mapping (0=Row-,1=Column-major)
3            # of process grids (P x Q)
2 1 4        Ps
2 4 1        Qs
16.0         threshold
3            # of panel fact
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)
2            # of recursive stopping criterium
2 4          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
3            # of recursive panel fact.
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

More details on each parameter can be found in the tuning section of benchmark documentation. This link can be used to generate a `HPL.dat` file for a given runtime configuration. Another useful link in this context is here.

Currently, the test supports automatic generation of `HPL.dat` file based on the system configuration. The class `GenerateHplConfig` in `modules.utils` is used for this purpose. The problem size of HPL is dependent on the available system memory and it is generally recommended to set a size that occupies at least 80% of the system memory. It means for systems that have big memory, a very huge problem size can be generated which can take a very long for the benchmark to run. Thus, we capped the system memory to 200 GB to avoid these very long run times.

For the intel processors, we use **one MPI process** per node but for AMD chips, we use number of L3 caches as number of MPI processes and use number of cores attached to each L3 cache as number of OpenMP threads.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/hpl/reframe_hpl.py --run --
↪performance-report
```

We can set number of nodes on the CLI using:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/hpl/reframe_hpl.py --run --
↪performance-report -S num_nodes=2
```

To choose a particular test during runtime using `-n` option as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/hpl/reframe_hpl.py --run --
↪performance-report -n HplGnuTest
```

Test class documentation

class apps.level0.cpu.hpl.reframe_hpl.**HplMixin**(**args*, ***kwargs*)

Common methods and attributes for HPL main tests

test_settings()

Common test settings

set_git_commit_tag()

Fetch git commit hash

export_env_vars()

Export env variables using OMPI_MCA param for OpenMPI

generate_config()

Generate HPL config file and place it in stagedir

set_tags()

Add tags to the test

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
# Finished      1 tests with the following results:
#               1 tests completed and passed residual checks,
#               0 tests completed and failed residual checks,
#               0 tests skipped because of illegal input values
# ----------------------------------------------------------------------------
↪--
# End of Tests.
```

extract_gflops()
Performance extraction function for Gflops. Sample stdout:

set_perf_patterns()
Set performance variables

set_reference_values()
Set reference perf variables

class apps.level0.cpu.hpl.reframe_hpl.**HplGnuTest**(**args*, ***kwargs*)
Main class of HPL test based on GNU toolchain

set_num_tasks()
Set number of processes and threads based on L3 cache

set_job_env_vars()
Set job specific env variables

download_hpl()
Download HPL 2.3 source code

set_topdir_makefile()
Set TOPDIR var in Makefile and copy to stagedir

emit_prebuild_cmds()
Make clean if already exists

build_executable()
Set build system and config options

set_executable()
Set executable name

job_launcher_opts()
Set job launcher options

class apps.level0.cpu.hpl.reframe_hpl.**HplMklTest**(**args*, ***kwargs*)
Main class of HPL test based on MKL

set_num_tasks()
Set number of tasks for job

set_omp_threads()
Set number of OpenMP threads

set_executable()
Set executable name

7.1.3 Intel MPI Benchmarks

Context

Intel MPI Benchmarks (IMB) are used to measure application-level latency and bandwidth, particularly over a high-speed interconnect, associated with a wide variety of MPI communication patterns with respect to message size.

Note: Currently, only benchmarks from `IMB-MPI1` components are included in the test.

Included benchmarks

Currently, the test includes following benchmarks:

- Pingpong
- Uniband
- Biband
- Sendrecv
- Allreduce
- Alltoall
- Allgather

By default all the above listed benchmarks will be run by the test. However, the user can choose subset of these benchmarks at the runtime using CLI. This will be discussed in *Usage*.

Number of MPI processes

By default benchmarks like Uniband, Biband, *etc*, are run with MPI processes varying from 2, 4, 8 and so on until the number of physical cores on the nodes. In order to reduce total number of benchmarks, only two runs for each benchmark is chosen

- Run with 1 MPI process per node
- Run with `N` MPI processes per node where `N` is number of physical cores.

Effectively using this configuration, we are running test that establish upper and lower bounds of benchmark metrics and thus minimising the time required for benchmarks to run.

Test configuration

The only file that is needed for the test to run is placed in `src/` folder which provides the list of message sizes to be tested in the benchmark. The file must be as follows:

```
0
4096
16384
131072
1048576
4194304
```

If we want to test for more message sizes, simply add new lines in the file and place it in `src/` folder.

Test variables

Different variables are available for the user to change the runtime configuration of the tests. They are listed as follows:

- `variants`: Benchmark variants that can be chosen as listed in *Included benchmarks* (default: All benchmarks listed in *Included benchmarks*).
- `mem`: Memory allocated per MPI process (default: 1).
- `timeout`: Timeout for running the benchmark for each message size (default: 2).

The variables `mem` and `timeout` are specific to IMB and more details about these variables can be found in the documentation.

Tip: For benchmarks like Alltoall, Allgather, Allreduce involving many nodes, runs with bigger message sizes might timeout. In this case increase the `timeout` variable. Similarly, nodes with many cores and smaller memory size can pose problems when running benchmarks. As stated in *Number of MPI processes*, as many MPI processes as number of physical cores are used for benchmark runs. So, if the node has `N` physical cores and less than `N GB` of DRAM, benchmarks will fail due to lack of sufficient memory. In this case reduce the `mem` variable which reduces the memory allocated for each MPI process.

All these variables can be configured at the runtime from the CLI using `-S` flag of ReFrame. It will be discussed in *Usage*.

Test parameterisation

The tests are parameterised based on the variable `tot_nodes`. The current default value is 2. This variable can **only** be configured from CLI using environment variable `IMBTEST_NODES`. Based on the `tot_nodes` value, the closest power of 2 is estimated and parameterised tests on different number of nodes generated in the powers of 2 are used. For instance, if `tot_nodes` is 64, tests are run on 2, 4, 8, 16, 32 and 64 nodes. For each run, all the requested benchmarks will be executed with different number of MPI processes as described in *Number of MPI processes*.

If the user wants to restrict number of nodes to only few runs, we can do it using `-t` flag on the CLI.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/imb/reframe_imb.py --exec-
↪policy=serial --run --performance-report
```

Important: It is absolutely necessary to use `--exec-policy=serial` option while running these benchmarks. By default ReFrame will execute tests in asynchronous mode, where all tests are executed at the same time. As we are interested in network latency and bandwidth metrics, it is advised to run these benchmarks serially so that they do not interfere with each other.

We can choose benchmark variants from CLI. For example, if we want to run only Uniband and Biband benchmarks:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/imb/reframe_imb.py --exec-
↪policy=serial --run --performance-report -S variants="Uniband","Biband"
```

Similarly other variables can also be configured from CLI. To use `mem` as 0.5 and `timeout` as 3.0:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/imb/reframe_imb.py --exec-
↪policy=serial --run --performance-report -S mem=0.5 -S timeout=3.0
```

To set the total number of nodes from CLI using `IMBTEST_NODES` environment variable, use following:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
IMBTEST_NODES=16 reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/imb/reframe_
↪imb.py --exec-policy=serial --run --performance-report
```

Finally, to select only few parameterised tests, we can use `-t` flag. For example, if `tot_nodes` is set to 16 and if we want to run only tests where number of nodes are 8 and 16, we can do following:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
IMBTEST_NODES=16 reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/imb/reframe_
↪imb.py --exec-policy=serial --run --performance-report -t 8$ -t 16$
```

All the above mentioned CLI flags can be used together without any side effects.

Test class documentation

class apps.level0.cpu.imb.reframe_imb.**ImbMixin**(**args*, ***kwargs*)

Common test attributes for IMB test

set_git_commit_tag()

Fetch git commit hash

get_l3_cache()

Get L3 cache size in MB and line size

set_num_tasks()

Set number of tasks for job

get_msg_lens()

Read input message lengths

set_executable()

Set executable name

export_env_vars()

Export env variables using OMPI_MCA param for OpenMPI

set_executable_opts()

Set executable options

add_launcher_options()

Add job launcher commands

set_tags()

Add tags to the test

set_sanity_patterns()

Set sanity patterns. Example stdout:

parse_stdout(*msg_len*, *var*, *ind*)

Read stdout file to parse perf variables

extract_bw(*msg_len=0*, *var='PingPong'*, *ind=3*)

Performance extraction function for bandwidth

extract_time(*msg_len=0*, *var='PingPong'*, *ind=2*)

Performance extraction function for latency

set_perf_patterns()

Set performance variables. Sample stdout

```
# e.g.
#----------------------------------------------------
# Benchmarking Uniband
# #processes = 4
#----------------------------------------------------
       #bytes #repetitions   Mbytes/sec      Msg/sec
            0         1000         0.00      9233819
```

set_reference_values()

Set reference perf variables

class apps.level0.cpu.imb.reframe_imb.**ImbPingpongTest**(**args*, ***kwargs*)

Main class of IMB Pingpong test

set_executable_opts()

Set executable options

set_sanity_patterns()

Set sanity patterns. We override the method in Mixin Class. Example stdout:

set_perf_patterns()

Set performance variables. We override the method in Mixin Class. Sample stdout

```
# e.g.
# #----------------------------------------------------
# # Benchmarking PingPong
# # #processes = 2
# #----------------------------------------------------
#        #bytes #repetitions      t[usec]   Mbytes/sec
#             0         1000         3.51         0.00
# #----------------------------------------------------
```

class apps.level0.cpu.imb.reframe_imb.**ImbOneCoreTests**(**args*, ***kwargs*)

Main class of all IMB variants tests using one core per node

class apps.level0.cpu.imb.reframe_imb.**ImbAllCoreTests**(**args*, ***kwargs*)

Main class of IMB variants tests using all cores per node

set_num_tasks()

Set number of tasks for job

7.1.4 IOR benchmark

Context

IOR is designed to measure parallel file system I/O performance through a variety of potential APIs. This parallel program performs writes and reads to/from files and reports the resulting throughput rates. The tests are configured in such a way to minimise the page caching effect on I/O bandwidth. See here for more details.

Note: In order to run this test, an environment variable `SCRATCH_DIR` must be defined in the system partition with the path to the scratch directory of the platform. Otherwise the test will fail.

Test variables

Several variable are defined in the tests which can be configured from the command line interface (CLI). They are summarised as follows:

- `num_nodes`: Number of nodes to run the test (default is 4)
- `num_mpi_tasks_per_node`: Number of MPI processes per node (default is 8)
- `block_size`: Block size of IOR test (default is 1g)
- `transfer_size`: Transfer size of IOR test (default is 1m)
- `num_segments`: Number of segments of IOR test (default is 1)

The variables `block_size`, `transfer_size` and `num_segments` are IOR related. More details on these variables can be found at IOR documentation.

Any of these variables can be overridden from the CLI using `-S` option of ReFrame. The examples are presented in *Usage*.

Test parameterisation

The test is parameterised with respect to two parameters namely I/O interface and file type. There are 3 different I/O interfaces available

- `posix`: POSIX I/O
- `mpiio`: MPI I/O
- `hdf5`: HDF5

We can write data to a single file or use file-per-process approach and tests are parameterised as follows:

- `single`: Single file for all processes
- `fpp`: File per process

The parameterised tests can be controlled by tags which will be shown in the *Usage* section.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/ior/reframe_ior.py --exec-
↪policy=serial --run --performance-report
```

Note: It is extremely important to use `--exec-policy=serial` for this particular test. By default, ReFrame executes the tests in asynchronous mode which means multiple jobs are executed at the same time if partition allows to do so. However, for this type of IO test, we do not want all the jobs using the underlying file system at the same time. So, we switch to serial execution where only one job at a time is executed on the partition.

To configure the test variables presented in *Test variables* section we can use `-S` option as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/ior/reframe_ior.py --exec-
↪policy=serial --run --performance-report -S num_nodes=2
```

Multiple variables can be configured simple by repeating `-S` flag for each variable as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/ior/reframe_ior.py --exec-
↪policy=serial --run --performance-report -S num_nodes=2 -S block_size=10g
```

By default all parameterised tests will be executed for a given partition. The list of tests can be obtained using:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/ior/reframe_ior.py -l
```

which will give following output :

```
[ReFrame Setup]
version:           3.9.0-dev.3+adca255d
command:           'reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/ior/
↪reframe_ior.py -l'
launched by:       mahendra@alaska-login-0.novalocal
working directory: '/home/mahendra/work/ska-sdp-benchmark-tests'
settings file:     'reframe_config.py'
check search path: (R) '/home/mahendra/work/ska-sdp-benchmark-tests/apps/level0/cpu/ior/
↪reframe_ior.py'
stage directory:   '/home/mahendra/work/ska-sdp-benchmark-tests/stage'
output directory:  '/home/mahendra/work/ska-sdp-benchmark-tests/output'

[List of matched checks]
- IorTest_hdf5_single (found in '/home/mahendra/work/ska-sdp-benchmark-tests/apps/level0/
↪cpu/ior/reframe_ior.py')
- IorTest_posix_single (found in '/home/mahendra/work/ska-sdp-benchmark-tests/apps/
↪level0/cpu/ior/reframe_ior.py')
```

(continues on next page)

(continued from previous page)

```
- IorTest_mpiio_single (found in '/home/mahendra/work/ska-sdp-benchmark-tests/apps/
↪level0/cpu/ior/reframe_ior.py')
- IorTest_posix_fpp (found in '/home/mahendra/work/ska-sdp-benchmark-tests/apps/level0/
↪cpu/ior/reframe_ior.py')
- IorTest_mpiio_fpp (found in '/home/mahendra/work/ska-sdp-benchmark-tests/apps/level0/
↪cpu/ior/reframe_ior.py')
- IorTest_hdf5_fpp (found in '/home/mahendra/work/ska-sdp-benchmark-tests/apps/level0/
↪cpu/ior/reframe_ior.py')
Found 6 check(s)

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

As we can see from the output, ReFrame will execute all IO types and file type tests. In order to choose only few parameterised tests, we can use `-t` flag to restrict the tests to given parameters. For example, to run only POSIX IO interface and Single file variant

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/ior/reframe_ior.py --exec-
↪policy=serial --run --performance-report -t posix$ -t single$
```

can be used. As in the case of `-S` option, `-t` can also be repeated as many times as user want.

Test class documentation

class apps.level0.cpu.ior.reframe_ior.**IorTest**(**args*, ***kwargs*)

Main class of IOR read and write tests

set_param_tags()

Add parameter tags to the test

set_git_commit_tag()

Fetch git commit hash

set_num_tasks_reservation()

Set number of tasks for job reservation

set_num_mpi_tasks()

Set number of MPI tasks

set_tags()

Add tags to the test

patch_job_launcher()

Monkey mock the job launcher command

set_executable()

Set executable name

set_executable_opts()

Set executable options

export_env_vars()

Export env variables using OMPI_MCA param for OpenMPI

chdir_to_scratch()

Add prerun command to change PWD to scratch dir before commencing test

job_launcher_opts()

Set job launcher options

set_sanity_patterns()

Set sanity patterns. Example stdout

```
# Max Write: 940.74 MiB/sec (986.44 MB/sec)
# Max Read:  1303.68 MiB/sec (1367.01 MB/sec)
# Finished            : Mon Oct 18 10:52:25 2021
```

extract_write_bw()

Performance extraction function for extract write bandwidth. Sample stdout

```
# Max Write: 940.74 MiB/sec (986.44 MB/sec)
```

extract_read_bw()

Performance extraction function for extract read bandwidth. Sample stdout

```
# Max Read:  1303.68 MiB/sec (1367.01 MB/sec)
```

set_perf_patterns()

Set performance variables

set_reference_values()

Set reference perf variables

7.1.5 STREAM benchmark

Context

STREAM is used to measure the sustainable memory bandwidth of high performance computers. The source code is available here.

Note: Currently, the implemented test uses only Intel compiler that is optimized for Intel processors. A generic GNU compiled stream test will be added in the future.

Test configuration

STREAM benchmark uses 3 arrays of size `N` to perform different kernels. The most relevant and interesting kernel is "Triad" kernel. In the test we use the size of the arrays in such a way that they occupy 60 % of the system memory. In this way, we are sure that caching effects are avoided while running the benchmark.

The `Makefile` in the `src/` folder contains all the optimized compiler flags used for Intel compiler to extract maximum peformance.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/cpu/stream/reframe_stream.py --
↪run --performance-report
```

Test class documentation

class apps.level0.cpu.stream.reframe_stream.**StreamTest**(**args*, ***kwargs*)

Main class of Stream test based on Intel compiler

set_git_commit_tag()

Fetch git commit hash

set_num_tasks_reservation()

Set number of tasks for job reservation

set_env_vars()

Set OpenMP environment variables

get_array_size()

Set array size to be 60% of main memory

set_tags()

Add tags to the test

set_launcher()

Set launcher to local to avoid appending mpirun or srun

build_executable()

Set build system and config options

export_env_vars()

Export env variables using OMPI_MCA param for OpenMPI

set_executable()

Set executable

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
# ----------------------------------------------------------------
# Solution Validates: avg error less than 1.000000e-13 on all three arrays
# ----------------------------------------------------------------
```

extract_bw(*kind='Copy'*)

Performance function to extract bandwidth. Sample stdout:

```
# Function    Best Rate MB/s  Avg time     Min time     Max time
# Copy:           42037.6     0.003859     0.003806     0.004004
# Scale:          41047.7     0.003917     0.003898     0.003942
# Add:            45138.5     0.005347     0.005317     0.005372
# Triad:          46412.1     0.005202     0.005171     0.005238
```

`set_perf_patterns()`
: Set performance variables

`set_reference_values()`
: Set reference perf values

7.2 GPU tests

7.2.1 Babel Stream benchmark

Context

Babel Stream is inspired from STREAM benchmark to measure the memory bandwidth on GPUs. It supports several other programming models for CPUs as well. More details can be found in the documentation.

Note: Although the benchmark supports various programming models, currently the test uses only OMP, TBB and CUDA models.

Test variants

Currently, three different variants of the benchmark are included in the test. They are

- `omp`: Using OpenMP threading model
- `tbb`: Using Intel's TBB model
- `cuda`: Using CUDA model for GPUs

The test is parameterised for these models and a specific test can be chosen at the runtime using `-t` flag on CLI. An example is shown in the *Usage*.

Test configuration

Like STREAM benchmark, Babel stream uses 3 arrays of size `N` for different kernels. The size of the arrays that will be used in the benchmark kernels can be configured at the run time using `mem_size` variable. Currently, the default value for `mem_size` is 0.4, which means the array size is chosen in such a way that all three arrays will occupy 40 % of total memory available.

Note: Depending on the GPU, sometimes we might get an error saying not enough space availble to store buffers. Decrease the `mem_size` in that case to allocate smaller arrays.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/babel_stream/reframe_
↪babelstream.py --run --performance-report
```

To run only `omp` variant and skip rest of the models, use `-t` flag as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/babel_stream/reframe_
↪babelstream.py -t omp$ --run --performance-report
```

To change the default value of `mem_size` during runtime, use `-S` flag. For example to use 30% of total memory:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/babel_stream/reframe_
↪babelstream.py -S mem_size=0.3 --run --performance-report
```

Test class documentation

class apps.level0.gpu.babel_stream.reframe_babelstream.**BabelStreamTest**(**args*, ***kwargs*)

Babel stream test main class

set_num_tasks()

Set number of tasks for job

set_array_size()

Set array size to be a certain percentage of main memory

set_launcher()

Set launcher to local to avoid appending mpirun or srun

build_executable()

Set build system and config options

export_env_vars()

Export env variables using OMPI_MCA param for OpenMPI

set_my_tags()

Add tags to the test

set_executable()

Set name of executable and runtime options

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
# BabelStream
# Version: 3.4
# Implementation: OpenMP
# Running kernels 100 times
# Precision: double
# Array size: 268.4 MB (=0.3 GB)
# Total size: 805.3 MB (=0.8 GB)
# Function    MBytes/sec  Min (sec)   Max         Average
# Copy        69666.441   0.00771     0.01172     0.00783
# Mul         67689.368   0.00793     0.01323     0.00811
# Add         75708.142   0.01064     0.01792     0.01090
# Triad       76265.085   0.01056     0.01411     0.01071
# Dot         103668.530  0.00518     0.01109     0.00547
```

extract_bw(*kind='Copy'*)

Extract bandwidth metric

set_perf_patterns()

Set performance metrics

set_reference_values()

Set reference perf metrics

7.2.2 GPUDirect RDMA Benchmark Tests

Context

The GPUDirect RDMA (GDR) technology exposes GPU memory to I/O devices by enabling the direct communication path between GPUs in two remote systems. This feature eliminates the need to use the system CPUs to stage GPU data in and out intermediate system memory buffers. As a result the end-to-end latency is reduced and the sustained bandwidth is increased (depending on the PCIe topology).

The GDRCopy (GPUDirect RDMA Copy) library leverages the GPUDirect RDMA APIs to create CPU memory mappings of the GPU memory. The advantage of a CPU driven copy is the very small overhead involved. That is helpful when low latencies are required.

Note: OSU micro benchmark suite is used to test the GDR capabilities in the current test setting. A more lower level verbs tests can also be used if the user wishes to remove the overhead imposed by MPI.

Included benchmarks

Currently, the test includes following categories of benchmarks:

Type of benchmark:

- `bw`: Uni directional bandwidth test
- `bibw`: Bi directional bandwidth test
- `latency`: Latency test

Communication type:

- `D_D`: Device to device

- `D_H`: Device to host
- `H_D`: Host to device

By default all the combination of tests will be performed. Both types of tests are parameterised and the user can select one or more of these tests at the run time using tags. This will be discussed in *Usage*.

Each of these tests will be executed in four different modes:

- GPUDirect RDMA and GDR Copy Enabled
- GPUDirect RDMA Enabled and GDR Copy Disabled
- GPUDirect RDMA Disabled and GDR Copy Enabled
- GPUDirect RDMA and GDR Copy Disabled

This will enable us investigate the effect of each component on the bandwidth and latency.

Benchmark configuration

There are two important variables for this test that need to be taken care of. They are

- `net_adptr`: Network adapter to use (default: `mlx5_0:1`)
- `ucx_tls`: UCX transport modes (default: `['rc', 'cuda_copy']`)

The value for `net_adptr` can be passed in two different ways:

- In the system/partition configuration as the key value pair in `extras` field.
- Other option would be to use `-S` flag at CLI to set the variable.

The value defined using `-S` flag has precedence over the system configuration value. If none of them are set, default value is used in the test. An example on how to define in `extras` is as follows:

```
'extras': {
            'interconnect': '100',  # in Gb/s
            'gpu_mem': '42505076736',  # in bytes
            'gdr_test_net_adptr': 'mlx5_0:1',  # NIC that has end-to-end connectivity␣
↪for GDR test
        }
```

An optimal settings of these variables is necessary in order to leverage the available bandwidth of Infiniband (IB) stack. We should choose the network adapter that has end-to-end connectivity with GPUs.

Tip: We can get this information from `nvidia-smi topo -m` command output. A typical output from this command can be as follows:

```
        GPU0    mlx5_0  mlx5_1  mlx5_2  mlx5_3  mlx5_4  mlx5_5  CPU Affinity    NUMA␣
↪Affinity
GPU0     X      NODE    NODE    PIX     PIX     PIX     PIX     0-19    0
mlx5_0  NODE     X      PIX     NODE    NODE    NODE    NODE
mlx5_1  NODE    PIX      X      NODE    NODE    NODE    NODE
mlx5_2  PIX     NODE    NODE     X      PIX     PIX     PIX
mlx5_3  PIX     NODE    NODE    PIX      X      PIX     PIX
mlx5_4  PIX     NODE    NODE    PIX     PIX      X      PIX
mlx5_5  PIX     NODE    NODE    PIX     PIX     PIX      X
```

(continues on next page)

(continued from previous page)

```
Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.
↪g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges␣
↪within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host␣
↪Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

We have to make sure to use adapter with the PIX attribute (single PCIe bridge). In this case mlx5_2, mlx5_3, mlx5_4 and mlx5_5 are directly connected to PCI express and we can choose any of them

---

Similarly, for the case of UCX transport methods, we can choose the ones that are available on the system. This information can be gathered using `ucx_info -d` which lists all the available transport methods. These default variables can be overridden from CLI which will be shown in *Usage*.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/gdr_test/reframe_gdr.py --
↪run --performance-report
```

If we want to set `ucx_tls` to `['dc', 'cuda-copy']` and `net_adptr` to `mlx5_3:1` we can use `-S` flag as follows

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/gdr_test/reframe_gdr.py -S␣
↪net_adptr=mlx5_3:1 -S ucx_tls=dc,cuda_copy --run --performance-report
```

Similarly, if we want to restrict the tests to only `D_D` (device to device) and `bw` (uni bandwidth), we can use tags as follows

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/gdr_test/reframe_gdr.py -t D_
↪D -t bw$ --run --peformance-report
```

Test class documentation

class apps.level0.gpu.gdr_test.reframe_gdr.**GpuDirectRdmaTest**(**args*, ***kwargs*)

GPU Direct RDMA test to benchmark bandwidth and latency between inter node GPUs

gen_msg_sizes()

Generate the message sizes used in benchmark in bytes

override_net_adptr_from_sys_config()

Override network adapter variable if found in sys config

set_git_commit_tag()

Fetch git commit hash

set_tags()

Add tags to the test

set_env_variables()

Set environment variables

patch_job_launcher()

Monkey mock the job launcher command

set_test_cases()

Define all the test cases and corresponding env variables

add_launcher_options()

Add job launcher options

set_executable()

Set executable and options

get_full_job_cmd()

Get full job command to use it in different tests

set_prerun_cmds()

Set prerun commands. Set env variables for case of RDMA and GDR copy enabled

set_postrun_cmds()

Set post run commands. Run rest of the cases

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
# Test with RDMA_GDR_Copy_Enabled started
# OSU MPI-CUDA Bandwidth Test v5.7.1
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size      Bandwidth (MB/s)
# 1                      1.89
# 2                      3.80
# 4                      7.65
# 8                     15.25
# Test with RDMA and GDR Copy Enabled finished
```

parse_stdout(*msg_size*, *case*)

Read the stdout file to extract perf metrics

Parameters

- **msg_size** (*int*) – Size of the message in bytes
- **case** (*str*) – Test case

Returns
Metric value

Return type
float

extract_bw(*msg_size=1*, *case='RDMA_GDR_Copy_Enabled'*)
Performance function to extract uni bandwidth

extract_bibw(*msg_size=1*, *case='RDMA_GDR_Copy_Enabled'*)
Performance function to extract bi bandwidth

extract_latency(*msg_size=1*, *case='RDMA_GDR_Copy_Enabled'*)
Performance function to extract latency

set_perf_patterns()
Set performance variables

set_reference_values()
Set reference perf values

7.2.3 NCCL performance benchmarks

Context

NCCL is a stand-alone library of standard communication routines for GPUs, implementing all-reduce, all-gather, reduce, broadcast, reduce-scatter, as well as any send/receive based communication pattern. It has been optimized to achieve high bandwidth on platforms using PCIe, NVLink, NVswitch, as well as networking using InfiniBand Verbs or TCP/IP sockets.

In this test, we are only interested in the intra-node communication latencies and bandwidths and so, we run this test on single node with multiple GPUs. The benchmarks report the so-called bus bandwidth that can be used to compare with underlying hardware peak bandwidth for collective communications. More details on how the bus bandwidth is estimated can be found at nccl tests repository.

Note: Each benchmark runs in two different modes namely, in-place and out-of-place. An in-place operation uses the same buffer for its output as was used to provide its input. An out-of-place operation has distinct input and output buffers.

Test variants

The test is parameterised to run following communication benchmarks:

- `sendrecv`
- `gather`
- `scatter`
- `reduce`
- `all_gather`

- `all_reduce`

A specific test can be chosen at the runtime using `-t` flag on CLI. An example is shown in the *Usage*.

Test configuration

The tests can be configured to change the minimum and maximum sizes of the messages that will be used in benchmarks. They can be configured at the runtime using `min_size` and `max_size` variables. The default values are 8 bytes and 128 MiB, respectively.

Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/nccl_test/reframe_nccltest.
↪py --run --performance-report
```

To run only `scatter` and `gather` variants and skip rest of the benchmarks, use `-t` flag as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/nccl_test/reframe_nccltest.
↪py -t scatter$ -t gather$ --run --performance-report
```

To change the default value of `min_size` during runtime, use `-S` flag. For example to use 1 MiB of `min_size`:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/nccl_test/reframe_nccltest.
↪py -S min_size=1M --run --performance-report
```

Test class documentation

class apps.level0.gpu.nccl_test.reframe_nccltest.**NcclTestDownload**(**args*, ***kwargs*)

Fixture to fetch NCCL test source code

set_sanity_patterns()

Set sanity patterns

class apps.level0.gpu.nccl_test.reframe_nccltest.**NcclTestBuild**(**args*, ***kwargs*)

NCCL tests compile test

set_sourcedir()

Set source directory from dependencies

set_build_system_opts()

Set build system options

class apps.level0.gpu.nccl_test.reframe_nccltest.**NcclPerfTest**(**args*, ***kwargs*)

NCCL performance tests main class

gen_msg_sizes()

Generate list of message sizes

set_sourcesdir()

Set source directory

set_git_commit_tag()

Fetch git commit hash

set_num_tasks_reservation()

Set number of tasks for job reservation

set_tags()

Add tags to the test

set_launcher()

Set launcher to local to avoid appending mpirun or srun

set_executable()

Set executable name

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
# # Out of bounds values : 0 OK
# # Avg bus bandwidth    : 0.791943
```

parse_stdout(*msg_size*, *place*, *ind*)

Read stdout file to extract perf variables

extract_algbw(*msg_size=None*, *place='in'*)

Performance function to extract algorithmic bandwidth

extract_busbw(*msg_size=None*, *place='in'*)

Performance function to extract bus bandwidth

extract_time(*msg_size=None*, *place='in'*)

Performance function to extract latency

set_perf_patterns()

Set performance variables. Sample stdout:

```
# # nThread 1 nGpus 2 minBytes 8 maxBytes 134217728 step: 2(factor) warmup␣
↪iters: 5 iters: 20 validation: 1
# #
# # Using devices
# #   Rank  0 Pid  16042 on grouille-1 device  0 [0x21] A100-PCIE-40GB
# #   Rank  1 Pid  16042 on grouille-1 device  1 [0x81] A100-PCIE-40GB
# #
# #                                                     out-of-place                ␣
↪     in-place
# #       size         count      type     time   algbw   busbw  error     time␣
↪  algbw   busbw  error
# #        (B)    (elements)                (us)  (GB/s)  (GB/s)            (us)␣
↪ (GB/s)  (GB/s)
#            8             2     float    23.17    0.00    0.00  0e+00    22.75␣
```

(continues on next page)

(continued from previous page)

```
 ↪   0.00    0.00  0e+00
#          16             4     float    22.65    0.00    0.00  0e+00    22.68␣
 ↪   0.00    0.00  0e+00
#          32             8     float    22.44    0.00    0.00  0e+00    22.54␣
 ↪   0.00    0.00  0e+00
#          64            16     float    22.83    0.00    0.00  0e+00    22.37␣
 ↪   0.00    0.00  0e+00
#         128            32     float    22.72    0.01    0.01  0e+00    22.64␣
 ↪   0.01    0.01  0e+00
#         256            64     float    22.67    0.01    0.01  0e+00    22.47␣
 ↪   0.01    0.01  0e+00
#         512           128     float    22.42    0.02    0.02  0e+00    22.26␣
 ↪   0.02    0.02  0e+00
#        1024           256     float    22.63    0.05    0.05  0e+00    22.50␣
 ↪   0.05    0.05  0e+00
#        2048           512     float    22.47    0.09    0.09  0e+00    22.52␣
 ↪   0.09    0.09  0e+00
#        4096          1024     float    23.33    0.18    0.18  0e+00    23.28␣
 ↪   0.18    0.18  0e+00
#        8192          2048     float    25.22    0.32    0.32  0e+00    24.90␣
 ↪   0.33    0.33  0e+00
#       16384          4096     float    33.57    0.49    0.49  0e+00    33.95␣
 ↪   0.48    0.48  0e+00
#       32768          8192     float    48.07    0.68    0.68  0e+00    49.19␣
 ↪   0.67    0.67  0e+00
#       65536         16384     float    68.66    0.95    0.95  0e+00    72.52␣
 ↪   0.90    0.90  0e+00
#      131072         32768     float    115.3    1.14    1.14  0e+00    114.1␣
 ↪   1.15    1.15  0e+00
#      262144         65536     float    176.9    1.48    1.48  0e+00    174.8␣
 ↪   1.50    1.50  0e+00
#      524288        131072     float    334.0    1.57    1.57  0e+00    342.7␣
 ↪   1.53    1.53  0e+00
#     1048576        262144     float    643.3    1.63    1.63  0e+00    599.7␣
 ↪   1.75    1.75  0e+00
#     2097152        524288     float   1125.6    1.86    1.86  0e+00   1077.9␣
 ↪   1.95    1.95  0e+00
#     4194304       1048576     float   2813.4    1.49    1.49  0e+00   2670.2␣
 ↪   1.57    1.57  0e+00
#     8388608       2097152     float   5561.3    1.51    1.51  0e+00   5497.1␣
 ↪   1.53    1.53  0e+00
#    16777216       4194304     float    11070    1.52    1.52  0e+00    10950␣
 ↪   1.53    1.53  1e+00
#    33554432       8388608     float    22215    1.51    1.51  0e+00    22687␣
 ↪   1.48    1.48  1e+00
#    67108864      16777216     float    45987    1.46    1.46  0e+00    46600␣
 ↪   1.44    1.44  1e+00
#   134217728      33554432     float    95433    1.41    1.41  0e+00    96707␣
 ↪   1.39    1.39  1e+00
# # Out of bounds values : 0 OK
# # Avg bus bandwidth    : 0.778536
# #
```

set_reference_values()
Set reference perf values

7.2.4 Funclib Test

Context

This test runs functions from *ska-sdp-func <https://gitlab.com/ska-telescope/sdp/ska-sdp-func>*. Currently implemented are tests for DFT and Phase Rotation.

Test variables

The DFT test supports two different polarisations as parameters.

The Phaserotation test supports two parameters, which are configured as tuples in one ReFrame parameter. Those two parameters are "baselines" which are the number of baselines to be tested and "times".

Environment variables

By default, the test will create a `conda` environment and run inside it for the sake of isolation. This can be controlled using env variable `CREATE_CONDA_ENV`. By setting it to `NO`, the test WILL NOT create a `conda` environment.

Similarly, the performance metrics are monitored using the perfmon toolkit. If the user does not want to monitor metrics, it can be achieved by setting `MONITOR_METRICS=NO`.

Usage

The tests can be run using the following commands:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/hippo_func_lib/reframe_
↪funclib_test.py --run --performance-report
```

If we want to change the variables to non default values, we should use `-S` flag. For example, if we want to run only 5 major cycles and 64 frequency channels, use

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level0/gpu/hippo_func_lib/reframe_
↪funclib_test.py -S start=1000000 --run --performance-report
```

Test class documentation

class apps.level0.gpu.hippo_func_lib.reframe_funclib_test.**FunclibTestDownload**(**args*, ***kwargs*)

Fixture to fetch ska-sdp-func source code

class apps.level0.gpu.hippo_func_lib.reframe_funclib_test.**FunclibTestBuild**(**args*, ***kwargs*)

Funclib test compile test

set_sourcedir()

Set source path based on dependencies

set_prebuild_cmds()

Make local lib dirs

set_build_system_attrs()

Set build directory and config options

set_postbuild_cmds()

Install libs

set_sanity_patterns()

Set sanity patterns

class apps.level0.gpu.hippo_func_lib.reframe_funclib_test.**FunclibDftTest**(**args*, ***kwargs*)

set_launcher()

Set launcher to local as it is no multi-node application

set_sanity_patterns()

Expected Output For every device and every version, the following block is printed to stdout:

set_perf_patterns()

Set performance metrics

set_reference_values()

Set reference performance values.

class apps.level0.gpu.hippo_func_lib.reframe_funclib_test.**PhaserotTest**(**args*, ***kwargs*)

set_launcher()

Set launcher to local as it is no multi-node application

set_sanity_patterns()

Expected Output For every device the following block is printed to stdout:

set_perf_patterns()

Set performance metrics

set_reference_values()

Set reference performance values.

CHAPTER

EIGHT

LEVEL 1 BENCHMARK TESTS

8.1 CUDA NIFTY gridder performance benchmark

8.1.1 Context

CUDA NIFTY Gridder (CNG) is a CUDA implementation of NIFTY gridder to (de)grid interferometric data using improved w-stacking algorithm.

In this test, we are interested in performance of CNG on different GPU devices. In order to stress the gridder, we use SKA1 MID synthetic dataset with configurable image size. More details on the design of the benchmark can be found in `src/` folder.

Note: The benchmark test uses visibility data that is randomly generated for a given uvw coverage. It is very expensive to do a DFT on this data to estimate the accuracy of the CNG. Hence, no accuracy tests are performed within this benchmark.

8.1.2 Test configuration

The tests can be configured to change the minimum and maximum number of frequency channels that will be used in the benchmark. Similarly, the image size can also be configured at the runtime. Available variables that can be configured at runtime are

- `min_chans`: Minimum number of frequency channels as power of 2 (default: 0)
- `max_chans`: Maximum number of frequency channels as power of 2 (default: 11)
- `img_size`: Image size as multiple of 1024 (default: 8)

With default variables, the benchmark will test on a image size of 8192 x 8192 pixels using 1 to 1024 frequency channels. They can be configured from CLI using `-S` flag which will be shown in *Usage*.

8.1.3 Usage

The test can be run using following commands.

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level1/cng_test/reframe_cngtest.py --
↪run --performance-report
```

To run using 16k image and till 4096 frequency channels, use -S option as

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level1/cng_test/reframe_cngtest.py -S␣
↪img_size=16 -t max_chans=13 --run --performance-report
```

8.1.4 Test class documentation

class apps.level1.cng_test.reframe_cngtest.**CngTest**(**args*, ***kwargs*)

CUDA NIFTY Gridder (CNG) performance tests main class

set_conda_prerun_cmds()

Emit conda env prerun commands

set_source_path()

Get source path attribute

set_tags()

Add tags to the test

install_cng_test()

Installs cng_test in conda env

set_launcher_opts()

Set job launcher options

set_executable()

Set executable name

set_git_commit_tag()

Override git tag method from base class

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
All tests have successfully finished
```

parse_stdout(*num_chan*, *perf*)

Read stdout file to extract perf variables

extract_time(*num_chan=None*, *perf='invert'*)

Performance function to extract (de)gridding times

extract_vis(*num_chan=None*, *perf='vis'*)

Performance function to extract number of visibilities

set_perf_patterns()

Set performance variables. Sample stdout:

```
--------------------------------------------------------------------------------
↪-------
# Image size:                           4096 x 4096
# Pixel size (in degrees):              4.099e-05
# Field of view (in degrees):           1.679e-01
# Minimum frequency:                    1.300e+09
# Maximum frequency:                    1.360e+09
# Number of baselines:                  312048
# Integration interval (in sec):        1800
# Precision:                            sp
# Accuracy:                             1e-05
# Number of iterations:                 10
================================================================================
            CUDA NIFTY Benchmark results using synthetic SKA1 MID dataset
================================================================================
# # Channels        # Visibilities    Invert time [s]   Predict time [s]
# 1                 312048           0.30428           0.09089
# 2                 624096           0.33928           0.09925
# 4                 1248192          0.34887           0.10189
# 8                 2496384          0.38257           0.11387
# 16                4992768          0.43269           0.14284
# 32                9985536          0.53047           0.17708
# 64                19971072         0.73875           0.26098
# 128               39942144         1.26618           0.44106
# 256               79884288         2.17768           0.74469
# 512               159768576        4.34335           1.34687
# 1024              319537152        8.66043           2.64405
--------------------------------------------------------------------------------
↪-------
# End of table
# All tests have successfully finished
```

set_reference_values()

Set reference perf values

8.2 IDG Test

8.2.1 Context

The image-domain gridder (IDG) is a new, fast gridder that makes w-term correction and a-term correction computationally very cheap. It performs extremely well on gpus. The source code is hosted on ASTRON GitLab repository and documentation can be found here.

8.2.2 Test variables

The test supports several runtime configurable variables:

- `layout`: Antenna layout. Available options are `SKA1_low` and `SKA1_mid`. (default is *SKA1_low*)
- `num_cycles`: Number of major cycles (default: 10)
- `num_stations`: Number of antenna stations (default: 100)
- `gridsize`: Gridsize used for IDG (default: 8192)
- `num_chans`: Number of frequency channels (default: 128)

This benchmark uses either `SKA1_low` or `SKA1_mid` antenna layout and generate random visibility data to do gridding and degridding. We use only one node and one GPU to run the benchmark and report various performance metrics. All these variables can be configured at the runtime which will be discussed in *:ref:idg usage*.

8.2.3 Environment variables

By default, the test will create a `conda` environment and run inside it for the sake of isolation. This can be controlled using env variable `CREATE_CONDA_ENV`. By setting it to `NO`, the test WILL NOT create `conda` environment.

Similarly, the performance metrics are monitored using the perfmon toolkit. If the user does not want to monitor metrics, it can be achieved by setting `MONITOR_METRICS=NO`.

8.2.4 Usage

The tests can be run using the following commands:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level1/idg_test/reframe_idgtest.py --
↪run --performance-report
```

If we want to change the variables to non default values, we should use `-S` flag. For example, if we want to run only 5 major cycles and 64 frequency channels, use

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level1/idg_test/reframe_idgtest.py -S␣
↪num_cycles=5 -S num_chans=64 --run --performance-report
```

8.2.5 Test class documentation

class apps.level1.idg_test.reframe_idgtest.**IdgTestDownload**(**args*, ***kwargs*)

Fixture to fetch IDG source code

class apps.level1.idg_test.reframe_idgtest.**IdgTestBuild**(**args*, ***kwargs*)

IDG test compile test

set_sourcedir()

Set source path based on dependencies

set_prebuild_cmds()

Make local lib dirs

set_build_system_attrs()

Set build directory and config options

set_postbuild_cmds()

Install libs

set_sanity_patterns()

Set sanity patterns

class apps.level1.idg_test.reframe_idgtest.**IdgTest**(**args*, ***kwargs*)

Main class of IDG benchmark tests

check_vars()

Check test variables

set_executable()

Set executable path and executable

set_tags()

Add tags to the test

get_num_nodes()

Get number of nodes from total cores requested and number of cores per node

set_num_tasks_job()

This method sets tasks for the job. We use this to override the num_tasks set for reservation. Using this approach we can set num_tasks to job in a more generic way

set_env_vars()

Set environment variables

add_launcher_options()

Set job launcher options

pre_launch()

Set prerun commands. It includes setting scratch directory and pre run commands from base class

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
>>> 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
```

extract_time(*kind='gridding'*)

Performance extraction function for time. Sample stdout:

```
>>> 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
```

extract_vis_thpt(*kind='gridding'*)

Performance extraction function for visibility throughput. Sample stdout:

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

set_perf_patterns()

Set performance variables

set_reference_values()

Set reference perf values

8.3 Imaging IO Test

8.3.1 Context

This is a prototype exploring the capability of hardware and software to deal with the types of I/O loads that the SDP will have to support for full-scale operation on SKA1 (and beyond). The benchmark is written in plain C and uses MPI for communication. The source code is hosted on SKA GitLab repository and documentation can be found here.

8.3.2 Test parameterisation

Currently, the benchmark supports three different parameterisations namely, - Variant of benchmark - Number of cores - Size of benchmark

Within variant, three different tests are defined namely, dry, write and read tests. As name suggests dry test runs all the computations without writing the data to the disk. This benchmark can be used to assess the computational performance of the prototype. Write test does the computations and writes the data to the disk and hence, benchmarks the I/O performance of the underlying file system. And finally, read test reads the data that has been written to the disk and gives the read performance.

Size of the benchmark defines how big of the benchmark we would want to run. The size `low-small` indicates small image for SKA1 LOW configuration, whereas `low-large` is large image (96k) for SKA1 LOW configuration. The same holds for `mid-small` and `mid-large`, albeit, for SKA1 MID, large image size is 192k.

They are defined in the ReFrame test as follows:

```
variant = parameter(['dry-test', 'write-test', 'read-test'])
num_cores = parameter(1 << i for i in range(min, max))
size = parameter(['tiny', 'low-small', 'low-large', 'mid-small', 'mid-large'])
```

8.3.3 Environment variables

All these parameterisations are provided as tags to the ReFrame tests and hence, we can simply choose which benchmark we want to run by specifying appropriate tags on command line. By default, `min` and `max` used to parameterise `num_cores` are 9 and 14, respectively. However, these variables can be overridden using custom environment variables `IMAGINGIOTEST_MIN` and `IMAGINGIOTEST_MAX`, respectively.

By default, the test will create a `conda` environment and run inside it for the sake of isolation. This can be controlled using env variable `CREATE_CONDA_ENV`. By setting it to `NO`, the test WILL NOT create `conda` environment.

Similarly, the performance metrics are monitored using the perfmon toolkit. If the user does not want to monitor metrics, it can be achieved by setting `MONITOR_METRICS=NO`.

8.3.4 Test filtering

The tests can be run using the following commands:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level1/imaging_iotest/reframe_iotest.py␣
↪--run --performance-report
```

But first let's see the tests generated by ReFrame using `--list` command as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level1/imaging_iotest/reframe_iotest.py␣
↪--list
```

The output is shown below:

```
[ReFrame Setup]
version:           3.10.0-dev.3+1407ae75
command:           '/home/mpaipuri/benchmark-tests/main/reframe/bin/reframe -c apps/
↪level1/imaging_iotest/reframe_iotest.py -l'
launched by:       mpaipuri@fnancy.nancy.grid5000.fr
working directory: '/home/mpaipuri/benchmark-tests/main'
settings file:     '/home/mpaipuri/benchmark-tests/main/reframe_config.py'
check search path: (R) '/home/mpaipuri/benchmark-tests/main/apps/level1/imaging_iotest/
↪reframe_iotest.py'
stage directory:   '/home/mpaipuri/benchmark-tests/main/stage'
output directory:  '/home/mpaipuri/benchmark-tests/main/output'

 [List of matched checks]
 - ImagingIOTest_read_test_low_small_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_large_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_small_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_large_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_large_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
```

(continues on next page)

(continued from previous page)

```
 - ImagingIOTest_dry_test_tiny_128 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_small_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_large_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_large_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_large_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_small_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_large_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_small_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_small_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_small_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_small_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_tiny_8 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_large_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_tiny_128 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_large_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_small_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_tiny_32 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_small_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_small_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_tiny_64 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_tiny_8 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_tiny_16 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_large_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_small_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_large_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_tiny_32 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
```

(continues on next page)

(continued from previous page)

```
 - ImagingIOTest_read_test_low_small_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_tiny_16 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_large_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_large_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_tiny_128 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_large_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_large_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_large_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_large_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_small_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_large_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_small_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_large_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_large_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_large_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_small_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_large_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_tiny_64 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_small_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_large_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_tiny_64 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_small_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_small_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_large_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_large_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_small_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
```

(continues on next page)

(continued from previous page)

```
 - ImagingIOTest_write_test_low_small_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_large_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_low_small_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_small_128 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_tiny_32 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_large_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_low_large_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_small_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_small_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_small_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_large_8 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_mid_small_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_mid_small_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_read_test_tiny_8 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_small_64 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_write_test_mid_large_32 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_low_small_16 (found in '/home/mpaipuri/benchmark-tests/main/
↪apps/level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTest_dry_test_tiny_16 (found in '/home/mpaipuri/benchmark-tests/main/apps/
↪level1/imaging_iotest/reframe_iotest.py')
 - ImagingIOTestBuild (found in '/home/mpaipuri/benchmark-tests/main/apps/level1/imaging_
↪iotest/reframe_iotest.py')
 Found 76 check(s)

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

As we can see, there are a lot of 76 tests generated by ReFrame and they are for a given system partition. If we defined multiple partitions and environments, number of tests will be multiplied by number of partitions and environments. This is not very ideal (unless we have infinite resources to run these tests on) and usually we use test filtering to run specific tests.

For example, if we want to run `dry-test` with 8 nodes and `low-small` test case, following commands must be used

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
IMAGINGIOTEST_MIN=3 IMAGINGIOTEST_MAX=4 reframe/bin/reframe -C reframe_config.py -c apps/
```

(continues on next page)

(continued from previous page)

```
↪level1/imaging-iotest/reframe_iotest.py --tag dry-test$ --tag low-small$ --run --
↪performance-report
```

The environment variables `IMAGINGIOTEST_MIN=3` and `IMAGINGIOTEST_MAX=4` will generate one test with 8 nodes for the reservation. Similarly, tags `dry-test$` and `low-small$` will select only tests with those tags. We can also restrict the tests for a given partition using `--system` flag and programming environment with `-p` flag.

8.3.5 Test class documentation

class apps.level1.imaging_iotest.reframe_iotest.**ImagingIOTestDownload**(**args*, ***kwargs*)

Fixture to fetch Imaging IO test source code

set_postrun_cmds()

Pull LFS objects

class apps.level1.imaging_iotest.reframe_iotest.**ImagingIOTestBuild**(**args*, ***kwargs*)

Imaging IO test compile test

set_sourcedir()

Set source path based on dependencies

set_build_system_attrs()

Set build directory and config options

set_sanity_patterns()

Set sanity patterns

class apps.level1.imaging_iotest.reframe_iotest.**ImagingIOTest**(**args*, ***kwargs*)

Main class of Imaging IO runtime tests

set_executable()

Set executable path and executable

set_tags()

Add tags to the test

get_num_nodes()

Get number of nodes from total cores requested and number of cores per node

set_subgrid_workers()

Set number of subgrid workers based on NUMA nodes

set_num_tasks_job()

This method sets tasks for the job. We use this to override the num_tasks set for reservation. Using this approach we can set num_tasks to job in a more generic way

set_num_threads()

Set number of OpenMP threads and OpenMP environment variables

add_launcher_options()

Set job launcher options

set_executable_opts()

Set iotest executable options

pre_launch()

Set prerun commands. It includes setting scratch directory and pre run commands from base class

post_launch()

Set post run commands. It includes removing visibility data files and running read-test

set_sanity_patterns()

Set sanity patterns. Example stdout:

```
# Fri Jul 23 15:15:41 2021[1,0]<stdout>:Operations:
```

We check number of time the above line is printed and compare it with number of sub grid workers

extract_stream_time()

Performance extraction function for stream time. Sample stdout:

```
Fri Jul 23 15:15:41 2021[1,2]<stdout>:Streamed for 3.53s
```

extract_degrid_flop()

Performance extraction function for degrid flop. Sample output:

```
# Fri Jul 23 15:15:41 2021[1,0]<stdout>: degrid 108.943 Gflops (10.9 GFlop/s,␣
↪92012544/92012544 visibilities, 1.47 GB, rate 0.15 GB/s, 25432 chunks)
```

extract_degrid_flops()

Performance extraction function for degrid flops. Sample output:

```
# Fri Jul 23 15:15:41 2021[1,0]<stdout>: degrid 108.943 Gflops (10.9 GFlop/s,␣
↪92012544/92012544 visibilities, 1.47 GB, rate 0.15 GB/s, 25432 chunks)
```

extract_degrid_rate()

Performance extraction function for degrid rate. Sample output:

```
# Fri Jul 23 15:15:41 2021[1,0]<stdout>: degrid 108.943 Gflops (10.9 GFlop/s,␣
↪92012544/92012544 visibilities, 1.47 GB, rate 0.15 GB/s, 25432 chunks)
```

extract_fft_flop()

Performance extraction function for fft flop. Sample output:

```
# Fri Jul 23 15:15:41 2021[1,0]<stdout>: FFTs 3.296 Gflop (0.3 Gflop/s)
```

extract_fft_flops()

Performance extraction function for fft flops. Sample output:

```
# Fri Jul 23 15:15:41 2021[1,0]<stdout>: FFTs 3.296 Gflop (0.3 Gflop/s)
```

extract_write_bw()

Performance extraction function for write bandwidth. Sample output:

```
# Fri Jul 23 15:15:41 2021[1,2]<stdout>:Writer 2: Wait: 2.02671s, Read: 0.
↪158911s, Write: 1.28306s, Idle: 0.0658979s
```

extract_read_bw()

Performance extraction function for read bandwidth. Sample output:

```
# Fri Jul 23 15:15:41 2021[1,2]<stdout>:Writer 2: Wait: 2.02671s, Read: 0.
↪158911s, Write: 1.28306s, Idle: 0.0658979s
```

for dry tests, read time will be zero. To avoid ZeroDivisionError, we add a small threshold value

set_perf_patterns()

Set performance variables

set_reference_values()

Set reference perf values

CHAPTER

NINE

LEVEL 2 BENCHMARK TESTS

9.1 RASCIL

9.1.1 Context

The Radio Astronomy Simulation, Calibration and Imaging Library expresses radio interferometry calibration and imaging algorithms in python and numpy. The interfaces all operate with familiar data structures such as image, visibility table, gain table, *etc*. The source code is hosted on SKA GitLab repository and documentation can be found here.

9.1.2 Test parameterisation

Currently, the benchmark supports three different parameterisations namely, - Number of nodes - Size of benchmark

Size of the benchmark defines how big of the benchmark we would want to run. In this case, size refers to number of frequency channels we are going to use to make the continuum image. Finally, scalability of the test can be specified using number of nodes. They are defined in the ReFrame test as follows:

```
size = parameter(['small', 'large', 'very-large', 'huge'])
num_nodes = parameter(1 << i for i in range(int(min), int(max)))
```

9.1.3 Environment variables

By default, the variables `min` and `max` are defined as 3 and 7, respectively. So, this parameterisation creates tests with number of nodes ranging from 8 to 64 in multiples of 2. The user can override the `min` and `max` variables using custom test environment variables `RASCILTEST_MIN` and `RASCILTEST_MAX`, respectively. If any of these environment variables are set, they will take precedence over default values. All these parameterisations are provided as tags to the ReFrame tests and hence, we can simply choose which benchmark we want to run by specifying appropriate tags on command line.

By default, the test will create a `conda` environment and run inside it for the sake of isolation. This can be controlled using env variable `CREATE_CONDA_ENV`. By setting it to `NO`, the test WILL NOT create `conda` environment.

Similarly, the performance metrics are monitored using the perfmon toolkit. If the user does not want to monitor metrics, it can be achieved by setting `MONITOR_METRICS=NO`.

9.1.4 Usage

The tests can be run using the following commands:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level2/rascil/reframe_rascil.py --run --
↪performance-report
```

But first let's see the tests generated by ReFrame using `--list` command as follows:

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c apps/level2/rascil/reframe_rascil.py --list
```

The output is shown below:

```
[ReFrame Setup]
version:           3.8.0-dev.2+8a9ceeda
command:           'reframe/bin/reframe -C reframe_config.py -c apps/level2/rascil/
↪reframe_rascil.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/apps/level2/rascil/
↪reframe_rascil.py'
stage directory:   '/home/mpaipuri/ska-sdp-benchmark-tests/stage'
output directory:  '/home/mpaipuri/ska-sdp-benchmark-tests/output'

[List of matched checks]
- RascilTest_small_100 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_large_16 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_large_50 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_large_100 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilAndDatasetDownloadTest (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/
↪level2/rascil/reframe_rascil.py')
- RascilTest_small_25 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_large_25 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_large_8 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_small_16 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_small_50 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilTest_small_8 (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/
↪rascil/reframe_rascil.py')
- RascilBuildTest (found in '/home/mpaipuri/ska-sdp-benchmark-tests/apps/level2/rascil/
↪reframe_rascil.py')
```

(continues on next page)

(continued from previous page)

```
Found 12 check(s)

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, there are a lot of 12 tests generated by ReFrame and they are for a given system partition. If we defined multiple partitions and environments, number of tests will be multiplied by number of partitions and environments.

For example, if we want to run `small` test with 8 nodes, following commands must be used

```
cd ska-sdp-benchmark-tests
conda activate ska-sdp-benchmark-tests
reframe/bin/reframe -C reframe_config.py -c aapps/level2/rascil/reframe_rascil.py --tag␣
↪dry-test --tag 8$ --tag small --run --performance-report
```

9.1.5 Test class documentation

class apps.level2.rascil.reframe_rascil.**RascilAndDatasetDownloadTest**(**args*, ***kwargs*)

Fetch RASCIL sources and datasets

download_dataset()

Download dataset using subprocess command

set_sanity_patterns()

Set sanity patterns

class apps.level2.rascil.reframe_rascil.**RascilBuildTest**(**args*, ***kwargs*)

RASCIL build test

set_sourcedir()

Set source directory from dependencies

set_env_variables()

Set env variables

pre_launch()

Install dependencies of RASCIL before installing RASCIL

set_launcher()

Set launcher to local to avoid appending mpirun or srun to python command

set_executable()

Set executable as python to install RASCIL using setup.py

set_sanity_patterns()

Set sanity patterns

class apps.level2.rascil.reframe_rascil.**RascilTest**(**args*, ***kwargs*)

Main class of RASCIL runtime tests

convert_datestring_timestamp(*time_str*, *fmt*)

Convert date string to time stamp

Parameters

- **time_str** (*str*) – Date time string

- **fmt** (*str*) – Format of date time string

Returns
Time stamp of the date time string

Return type
str

set_tags()
Add tags to the test

set_dependencies()
Set dependencies of the test

set_executable(*RascilBuildTest*)
Set executable path from dependencies

skip_tests()
Skip test based on rules defined

set_num_tasks_job()
Set number of MPI tasks of the job

set_env_variables()
Set OpenMP threads and related environment variables

set_nfreq_channels()
Set number of frequency channels based on number of nodes

set_launcher()
Set launcher to local to avoid appending mpirun or srun to python command

setup_dask_launcher_cmd()
Setup dask launcher command

symlink_dataset()
Symlink dataset from prefix to stagedir

set_executable_opts()
Set RASCIL executable options

set_keep_files()
Set list of files that we want to keep in output folder

rm_dask_worker_space()
Set RESULTS DIR env variable

pre_launch()
Set pre run commands. It includes setting up dask cluster

set_sanity_patterns()
Set sanity patterns. When RASCIL finishes the job successfully, it creates image files in fits format. We check if the files are created as sanity check

extract_times(*var='create_blockvisibility_from_ms'*)
Generic performance extraction function to extract time

extract_wall_time()

Performance function to extract wall time. Sample output:

```
# 26/07/2021 05:41:22 PM.110 rascil-logger INFO Started  : 2021-07-26 13:52:07.
↪487374
# 26/07/2021 05:41:22 PM.110 rascil-logger INFO Finished : 2021-07-26 17:41:22.
↪110077
```

extract_imaging_time()

Performance extraction function for imaging time

extract_processing_time()

Performance extraction function for processing time. Sample output:

```
# 27/07/2021 08:57:15 PM.171 rascil-logger INFO Total processor time 1102.612␣
↪(s), total wallclock time 161.490 (s),
```

set_perf_variables()

Set performance variables

set_reference_values()

Set reference perf values

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

Extra functionality for reframe

modules.reframe_extras.**patch_launcher_command**(*job*)

Patch the job launcher command for mpirun and mpiexec. We remove -np/-n arguments as we pass them manually inside ReFrame test

modules.reframe_extras.**patch_job_names**(*job*, *stagedir*)

Patch the job name, stdout and stderr to remove special characters injected by fixtures

class modules.reframe_extras.**CachedRunTest**(**args*, ***kwargs*)

Mixin.

Classes using this can be derive from `rfm.RunOnlyRegressionTest` Assumes saved output files are in a directory `cache/` in same parent directory (and with same directory tree) as the `output/` and `stage/` directories.

set `self.use_cache` to True or a path relative to cwd.

NB Any class using this MUST NOT change self.executable in a method decorated with `@rfm.run_before('run')` as that will override functionality here.

no_run()

Turn the run phase into a no-op

copy_saved_output()

Copy saved output files to stage dir.

class modules.reframe_extras.**CachedCompileOnlyTest**(**args*, ***kwargs*)

A compile-only test with caching of binaries between `reframe` runs.

Test classes derived from this class will save `self.executable` to a `./builds/{system}/{partition}/{environment}/{self.name}` directory after compilation. However if this path (including the filename) exists before compilation (i.e. on the next run):

- No compilation occurs
- **`self.sourcesdir` is set to this directory, so `reframe` will copy the** binary to the staging dir (as if compilation had occured)
- A new attribute `self.build_path` is set to this path (otherwise None)

Note that `self.sourcesdir` is only manipulated if no compilation occurs, so compilation test-cases which modify this to specify the source directory should be fine.

conditional_compile()

Point sourcedir to build_dir if it exists

copy_executable()

Copy executables to build_dir if it compiles

class modules.reframe_extras.**NoBuild**(**args*, ***kwargs*)

A no-op build system

class modules.reframe_extras.**CachedGitCloneTest**(**args*, ***kwargs*)

A git clone-only test with caching between the `reframe` runs.

This test clones the git repository and copies the repository to `$PWD/src` folder on the first run. Once the `$PWD/src` is there with repository inside it, it simply sets this directory as `self.sourcesdir` so new cloning is avoided for the next run.

If a new clone is needed the user simply need to remove $PWD/src before running reframe.

We need to set few variables for this test to work

- `repo_name`: Name of the repository
- `repo_url`: URL of git repository

check_req_variables()

Check essential variables if they are defined

set_sources()

Set source dir

set_executable()

Set a noop exe

copy_sources_to_cache()

Copy sources to cache dir

class modules.reframe_extras.**FetchSourcesBase**(**args*, ***kwargs*)

Fixture to fetch source code

set_conda_prerun_cmds()

Emit conda env prerun commands

set_sanity_patterns()

Set sanity patterns

class modules.reframe_extras.**PerfmonMixin**(**args*, ***kwargs*)

create_perfmetrics_dir()

Create a dir for storing perfmetrics

create_fake_job_id_local_scheduler()

If scheduler is local, make a fake job ID

patch_job_launcher()

Monkey mock the job launcher command

get_scheduler_env_vars()

Get scheduler specific env variables

export_env_variables()

Make a string of env variables that will be exported to all nodes

get_mpi_launcher_cmd()

Get MPI launcher command with additional options

set_postrun_cmds()

Set postrun commands

class modules.reframe_extras.**MultiRunMixin**(**args*, ***kwargs*)

This mixin creates a uniquely identifiable ID for tests that are run multiple times. Over all tests that are run with one reframe call, this yields the same unique key.

class modules.reframe_extras.**AppBase**(**args*, ***kwargs*)

A base test class that includes common methods for application reframe tests.

This base class defines several methods that are common to different reframe tests

property num_tasks_assigned

Get number of job tasks

property run_command_emitted

Get job run command

get_max_nodes_partition()

Get maximum number of available nodes in partition

check_test_feasibility()

Check if test is feasible based on number of nodes requested

set_num_tasks_reservation()

Set number of tasks for job reservation

set_git_commit_tag()

Get git commit hash

class modules.reframe_extras.**SpackBase**(**args*, ***kwargs*)

A base class that includes common methods for deploying software stack using Spack

check_cluster_var()

Checks if cluster name is valid and skips the test if not

check_build_cache()

Checks if mirror name is defined if build cache is provided

set_keep_files()

Keep spack config file in output

export_spack_root()

Modify .bashrc to export SPACK_ROOT env variable

apply_patch()

Apply patches to Spack

emit_spack_installation_cmds()

Commands to check if Spack is installed on the host

set_num_tasks()

Reserve all cores on the node if job is remote

emit_spack_add_mirror_cmds()

Commands to add mirrors to spack if build cache is found

emit_spack_env_rm_cmds()
Commands to removes Spack env if already exists

merge_spack_configs()
Merge all seperate Spack config files into spack.yaml and place it stage dir

emit_spack_env_create_cmds()
Commands to create spack env

get_sys_compiler()
Read system compiler from config files

emit_compiler_install_cmds()
Commands to install GCC 9.3.0 and Intel 2021.4.0 and adds to compiler.yml

set_local_launcher()
Set launcher to be local

set_executable()
Set executable and options

set_postrun_cmds()
Set post run commands that generate module files

set_sanity_patterns()
Set sanity patterns

modules.reframe_extras.**slurm_node_info**(*partition=None*)
Get information about slurm nodes. Returns a sequence of dicts, one per node with keys/values all strs as follows:

- "NODELIST": name of node
- "NODES": number of nodes
- "PARTITION": name of partition, * appended if default
- "STATE": e.g. "idle"
- "CPUS": str number of cpus
- "S:C:T": Extended processor information: number of sockets, cores, threads in format "S:C:T"
- "MEMORY": Size of memory in megabytes
- "TMP_DISK": Size of temporary disk space in megabytes
- "WEIGHT": Scheduling weight of the node
- "AVAIL_FE": ?
- "REASON": The reason a node is unavailable (down, drained, or draining states) or "none"

See `man sinfo` for `sinfo --Node --long` for full details.

Parameters
partition (*str*) – Name of slurm partition to query, else information for all partitions is returned.

Returns
nodes

Return type
list

modules.reframe_extras.**hostlist_to_hostnames**(*s*)

Convert a Slurm 'hostlist expression' to a list of individual node names. Uses *scontrol* command.

Parameters
s (*str*) – Host list

Returns
Hostnames

Return type
list

class modules.reframe_extras.**SchedulerInfo**(*rfm_partition*, *exclude_states=None*, *only_states=None*)

Information from the scheduler.

The returned object has attributes:

- num_nodes: number of nodes
- sockets_per_node: number of sockets per node
- pcores_per_node: number of physical cores per node
- lcores_per_node: number of logical cores per node

If rfm_partition is None the above attributes describe the **default** scheduler partition. Otherwise the following sbatch directives in the *access* property of the ReFrame partition will affect the information returned:

- --partition
- --exclude

Parameters

- **rfm_partition** – reframe.core.systems.SystemPartition
- **exclude_states** – sequence of str, exclude nodes in these Slurm node states
- **only_states** – sequence of str, only include nodes in these Slurm node states

slurm_info()

Get SLURM scheduler info

oar_info()

Get OAR scheduler info

local_info()

Get info when scheduler is local

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.

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

a

apps.level0.cpu.hpcg.reframe_hpcg, 43
apps.level0.cpu.hpl.reframe_hpl, 46
apps.level0.cpu.imb.reframe_imb, 49
apps.level0.cpu.ior.reframe_ior, 54
apps.level0.cpu.stream.reframe_stream, 57
apps.level0.gpu.babel_stream.reframe_babelstream, 59
apps.level0.gpu.gdr_test.reframe_gdr, 61
apps.level0.gpu.hippo_func_lib.reframe_funclib_test, 69
apps.level0.gpu.nccl_test.reframe_nccltest, 65
apps.level1.cng_test.reframe_cngtest, 71
apps.level1.idg_test.reframe_idgtest, 73
apps.level1.imaging_iotest.reframe_iotest, 76
apps.level2.rascil.reframe_rascil, 85

m

modules.reframe_extras, 93
modules.utils, 97

s

spack.spack_tests.alaska.reframe_alaska, 27
spack.spack_tests.grid5000.grenoble.dahu.reframe_dahug5k, 27
spack.spack_tests.grid5000.grenoble.troll.reframe_trollg5k, 27
spack.spack_tests.grid5000.lyon.gemini.reframe_geminig5k, 28
spack.spack_tests.grid5000.nancy.gros.reframe_grosg5k, 28
spack.spack_tests.grid5000.nancy.grouille.reframe_grouilleg5k, 28
spack.spack_tests.juwels.booster.reframe_juwelsbooster, 29
spack.spack_tests.juwels.cluster.reframe_juwelscluster, 29
spack.spack_tests.marconi100.reframe_marconi100, 29

INDEX

A

add_launcher_options() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
add_launcher_options() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
add_launcher_options() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
add_launcher_options() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 81
AlaskaSpackEnv (*class in spack.spack_tests.alaska.reframe_alaska*), 27
AppBase (*class in modules.reframe_extras*), 95
apply_patch() (*modules.reframe_extras.SpackBase method*), 95
apps.level0.cpu.hpcg.reframe_hpcg module, 43
apps.level0.cpu.hpl.reframe_hpl module, 46
apps.level0.cpu.imb.reframe_imb module, 49
apps.level0.cpu.ior.reframe_ior module, 54
apps.level0.cpu.stream.reframe_stream module, 57
apps.level0.gpu.babel_stream.reframe_babelstream module, 59
apps.level0.gpu.gdr_test.reframe_gdr module, 61
apps.level0.gpu.hippo_func_lib.reframe_funclibtest module, 69
apps.level0.gpu.nccl_test.reframe_nccltest module, 65
apps.level1.cng_test.reframe_cngtest module, 71
apps.level1.idg_test.reframe_idgtest module, 73
apps.level1.imaging_iotest.reframe_iotest module, 76
apps.level2.rascil.reframe_rascil module, 85

B

BabelStreamTest (*class in apps.level0.gpu.babel_stream.reframe_babelstream*), 60
build_executable() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgGnuTest method*), 45
build_executable() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgXlTest method*), 45
build_executable() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49
build_executable() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
build_executable() (*apps.level0.gpu.babel_stream.reframe_babelstream method*), 60

C

CachedCompileOnlyTest (*class in modules.reframe_extras*), 93
CachedGitCloneTest (*class in modules.reframe_extras*), 94
CachedRunTest (*class in modules.reframe_extras*), 93
chdir_to_scratch() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 57
check_build_cache() (*modules.reframe_extras.SpackBase method*), 95
check_cluster_var() (*modules.reframe_extras.SpackBase method*), 95
check_req_variables() (*modules.reframe_extras.CachedGitCloneTest method*), 94
check_test_feasibility() (*modules.reframe_extras.AppBase method*), 95
check_vars() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
CngTest (*class in apps.level1.cng_test.reframe_cngtest*), 72

conditional_compile() (*modules.reframe_extras.CachedCompileOnlyTest method*), 93
convert_datestring_timestamp() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 87
copy_executable() (*modules.reframe_extras.CachedCompileOnlyTest method*), 93
copy_saved_output() (*modules.reframe_extras.CachedRunTest method*), 93
copy_sources_to_cache() (*modules.reframe_extras.CachedGitCloneTest method*), 94
create_fake_job_id_local_scheduler() (*modules.reframe_extras.PerfmonMixin method*), 94
create_perfmetrics_dir() (*modules.reframe_extras.PerfmonMixin method*), 94

D

download_dataset() (*apps.level2.rascil.reframe_rascil.RascilAndDataset method*), 87
download_hpl() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49

E

emit_compiler_install_cmds() (*modules.reframe_extras.SpackBase method*), 96
emit_conda_env_cmds() (*in module modules.utils*), 100
emit_conda_init_cmds() (*in module modules.utils*), 100
emit_prebuild_cmds() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49
emit_spack_add_mirror_cmds() (*modules.reframe_extras.SpackBase method*), 95
emit_spack_env_create_cmds() (*modules.reframe_extras.SpackBase method*), 96
emit_spack_env_rm_cmds() (*modules.reframe_extras.SpackBase method*), 95
emit_spack_installation_cmds() (*modules.reframe_extras.SpackBase method*), 95
estimate_n() (*modules.utils.GenerateHplConfig method*), 97
estimate_pq() (*modules.utils.GenerateHplConfig method*), 97
export_env_variables() (*modules.reframe_extras.PerfmonMixin method*), 94
export_env_vars() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 45
export_env_vars() (*apps.level0.cpu.hpl.reframe_hpl.HplMixin method*), 48
export_env_vars() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
export_env_vars() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
export_env_vars() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
export_env_vars() (*apps.level0.gpu.babel_stream.reframe_babelstream. method*), 60
export_spack_root() (*modules.reframe_extras.SpackBase method*), 95
extract_algbw() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTes method*), 67
extract_bibw() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaT method*), 65
extract_busbw() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTes method*), 67
extract_bw() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 53
extract_bw() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
extract_bw() (*apps.level0.gpu.babel_stream.reframe_babelstream.Babel method*), 61
extract_bw() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 65
extract_degrid_flop() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
extract_degrid_flops() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
extract_degrid_rate() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
extract_fft_flop() (*apps.level1.imaging_iotest.reframe_iotest.Imaging method*), 82
extract_fft_flops() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
extract_gflops() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 45
extract_gflops() (*apps.level0.cpu.hpl.reframe_hpl.HplMixin method*), 48
extract_imaging_time() (*apps.level2.rascil.reframe_rascil.RascilTest*

method), 89
extract_latency() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 65
extract_processing_time() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 89
extract_read_bw() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 57
extract_read_bw() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
extract_stream_time() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
extract_time() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 53
extract_time() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method*), 67
extract_time() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
extract_time() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
extract_times() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
extract_vis() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
extract_vis_thpt() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 76
extract_wall_time() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
extract_write_bw() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 57
extract_write_bw() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82

F

FetchSourcesBase (*class in modules.reframe_extras*), 94
filter_systems_by_env() (*in module modules.utils*), 100
filter_systems_by_name() (*in module modules.utils*), 99
find_perf_logs() (*in module modules.utils*), 98
FunclibDftTest (*class in apps.level0.gpu.hippo_func_lib.reframe_funclib_test*), 70
FunclibTestBuild (*class in apps.level0.gpu.hippo_func_lib.reframe_funclib_test*), 70
FunclibTestDownload (*class in apps.level0.gpu.hippo_func_lib.reframe_funclib_test*), 70

G

G5kDahuSpackEnv (*class in spack.spack_tests.grid5000.grenoble.dahu.reframe_dahug5k*), 27
G5kGeminiSpackEnv (*class in spack.spack_tests.grid5000.lyon.gemini.reframe_geminig5k*), 28
G5kGouilleSpackEnv (*class in spack.spack_tests.grid5000.nancy.grouille.reframe_grouilleg5k*), 29
G5kGrosSpackEnv (*class in spack.spack_tests.grid5000.nancy.gros.reframe_grosg5k*), 28
G5kTrollSpackEnv (*class in spack.spack_tests.grid5000.grenoble.troll.reframe_trollg5k*), 28
gen_msg_sizes() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdma method*), 64
gen_msg_sizes() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTes method*), 66
generate_config() (*apps.level0.cpu.hpl.reframe_hpl.HplMixin method*), 48
generate_random_number() (*in module modules.utils*), 100
GenerateHplConfig (*class in modules.utils*), 97
get_array_size() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
get_config() (*modules.utils.GenerateHplConfig method*), 97
get_full_job_cmd() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRd method*), 64
get_l3_cache() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
get_max_nodes_partition() (*modules.reframe_extras.AppBase method*), 95
get_mpi_launcher_cmd() (*modules.reframe_extras.PerfmonMixin method*), 94
get_msg_lens() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
get_num_nodes() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
get_num_nodes() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOT method*), 81
get_scheduler_env_list() (*in module modules.utils*), 100
get_scheduler_env_vars() (*modules.reframe_extras.PerfmonMixin method*), 94
get_sys_compiler() (*modules.reframe_extras.SpackBase method*), 96
git_describe() (*in module modules.utils*), 100
GpuDirectRdmaTest (*class in*

apps.level0.gpu.gdr_test.reframe_gdr), 64

H

hostlist_to_hostnames() (*in module modules.reframe_extras*), 96
HpcgGnuTest (*class in apps.level0.cpu.hpcg.reframe_hpcg*), 45
HpcgMixin (*class in apps.level0.cpu.hpcg.reframe_hpcg*), 44
HpcgMklTest (*class in apps.level0.cpu.hpcg.reframe_hpcg*), 46
HpcgXlTest (*class in apps.level0.cpu.hpcg.reframe_hpcg*), 45
HplGnuTest (*class in apps.level0.cpu.hpl.reframe_hpl*), 49
HplMixin (*class in apps.level0.cpu.hpl.reframe_hpl*), 48
HplMklTest (*class in apps.level0.cpu.hpl.reframe_hpl*), 49

I

IdgTest (*class in apps.level1.idg_test.reframe_idgtest*), 75
IdgTestBuild (*class in apps.level1.idg_test.reframe_idgtest*), 74
IdgTestDownload (*class in apps.level1.idg_test.reframe_idgtest*), 74
ImagingIOTest (*class in apps.level1.imaging_iotest.reframe_iotest*), 81
ImagingIOTestBuild (*class in apps.level1.imaging_iotest.reframe_iotest*), 81
ImagingIOTestDownload (*class in apps.level1.imaging_iotest.reframe_iotest*), 81
ImbAllCoreTests (*class in apps.level0.cpu.imb.reframe_imb*), 53
ImbMixin (*class in apps.level0.cpu.imb.reframe_imb*), 52
ImbOneCoreTests (*class in apps.level0.cpu.imb.reframe_imb*), 53
ImbPingpongTest (*class in apps.level0.cpu.imb.reframe_imb*), 53
install_cng_test() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
IorTest (*class in apps.level0.cpu.ior.reframe_ior*), 56

J

JBoosterSpackEnv (*class in spack.spack_tests.juwels.booster.reframe_juwelsbooster*), 29
JClusterSpackEnv (*class in spack.spack_tests.juwels.cluster.reframe_juwelscluster*), 29
job_launcher_opts() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgGnuTest method*), 45
job_launcher_opts() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgXlTest method*), 45
job_launcher_opts() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49
job_launcher_opts() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 57

L

load_perf_logs() (*in module modules.utils*), 98
local_info() (*modules.reframe_extras.SchedulerInfo method*), 97

M

Marconi100SpackEnv (*class in spack.spack_tests.marconi100.reframe_marconi100*), 30
merge_spack_configs() (*in module modules.utils*), 101
merge_spack_configs() (*modules.reframe_extras.SpackBase method*), 96
module
    apps.level0.cpu.hpcg.reframe_hpcg, 43
    apps.level0.cpu.hpl.reframe_hpl, 46
    apps.level0.cpu.imb.reframe_imb, 49
    apps.level0.cpu.ior.reframe_ior, 54
    apps.level0.cpu.stream.reframe_stream, 57
    apps.level0.gpu.babel_stream.reframe_babelstream, 59
    apps.level0.gpu.gdr_test.reframe_gdr, 61
    apps.level0.gpu.hippo_func_lib.reframe_funclib_test, 69
    apps.level0.gpu.nccl_test.reframe_nccltest, 65
    apps.level1.cng_test.reframe_cngtest, 71
    apps.level1.idg_test.reframe_idgtest, 73
    apps.level1.imaging_iotest.reframe_iotest, 76
    apps.level2.rascil.reframe_rascil, 85
    modules.reframe_extras, 93
    modules.utils, 97
    spack.spack_tests.alaska.reframe_alaska, 27
    spack.spack_tests.grid5000.grenoble.dahu.reframe_dahug 27
    spack.spack_tests.grid5000.grenoble.troll.reframe_trol 27

spack.spack_tests.grid5000.lyon.gemini.reframe_gemini_lyon, 28
spack.spack_tests.grid5000.nancy.gros.reframe_gros_nancy, 28
spack.spack_tests.grid5000.nancy.grouille.reframe_grouille_nancy, 28
spack.spack_tests.juwels.booster.reframe_juwels_booster, 29
spack.spack_tests.juwels.cluster.reframe_juwels_cluster, 29
spack.spack_tests.marconi100.reframe_marconi100, 29
modules.reframe_extras
module, 93
modules.utils
module, 97
MultiRunMixin (*class in modules.reframe_extras*), 95

N

NcclPerfTest (*class in apps.level0.gpu.nccl_test.reframe_nccltest*), 66
NcclTestBuild (*class in apps.level0.gpu.nccl_test.reframe_nccltest*), 66
NcclTestDownload (*class in apps.level0.gpu.nccl_test.reframe_nccltest*), 66
no_run() (*modules.reframe_extras.CachedRunTest method*), 93
NoBuild (*class in modules.reframe_extras*), 94
num_tasks_assigned (*modules.reframe_extras.AppBase property*), 95

O

oar_info() (*modules.reframe_extras.SchedulerInfo method*), 97
override_net_adptr_from_sys_config() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64

P

parse_path_metadata() (*in module modules.utils*), 98
parse_stdout() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 53
parse_stdout() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
parse_stdout() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method*), 67
parse_stdout() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
patch_job_launcher() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
patch_job_launcher() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
patch_job_launcher() (*modules.reframe_extras.PerfmonMixin method*), 94
patch_job_names() (*in module modules.reframe_extras*), 93
patch_launcher_command() (*in module modules.reframe_extras*), 93
PerfmonMixin (*class in modules.reframe_extras*), 94
PhaserotTest (*class in apps.level0.gpu.hippo_func_lib.reframe_funclib_test*), 70
post_launch() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
pre_launch() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
pre_launch() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 81
pre_launch() (*apps.level2.rascil.reframe_rascil.RascilBuildTest method*), 87
pre_launch() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88

R

RascilAndDatasetDownloadTest (*class in apps.level2.rascil.reframe_rascil*), 87
RascilBuildTest (*class in apps.level2.rascil.reframe_rascil*), 87
RascilTest (*class in apps.level2.rascil.reframe_rascil*), 87
read_perflog() (*in module modules.utils*), 98
rm_dask_worker_space() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
rpm_command_emitted (*modules.reframe_extras.AppBase property*), 95

S

SchedulerInfo (*class in modules.reframe_extras*), 97
sdp_benchmark_tests_root() (*in module modules.utils*), 97
set_array_size() (*apps.level0.gpu.babel_stream.reframe_babelstream.B method*), 60
set_build_system_attrs() (*apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibTest method*), 70
set_build_system_attrs() (*apps.level1.idg_test.reframe_idgtest.IdgTestBuild method*), 75
set_build_system_attrs() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTestBuild*

method), 81
set_build_system_opts() (*apps.level0.gpu.nccl_test.reframe_ncclteset_executable() (apps.level2.rascil.reframe_rascil.RascilBuildTest method*), 87
set_executable() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_executable() (*modules.reframe_extras.CachedGitCloneTest method*), 94
set_executable() (*modules.reframe_extras.SpackBase method*), 96
set_executable_opts() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 44
set_executable_opts() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMklTest method*), 46
set_executable_opts() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
set_executable_opts() (*apps.level0.cpu.imb.reframe_imb.ImbPingpongTest method*), 53
set_executable_opts() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
set_executable_opts() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 81
set_executable_opts() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_git_commit_tag() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 44
set_git_commit_tag() (*apps.level0.cpu.hpl.reframe_hpl.HplMixin method*), 48
set_git_commit_tag() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
set_git_commit_tag() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
set_git_commit_tag() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
set_git_commit_tag() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
set_git_commit_tag() (*apps.level0.gpu.nccl_test.reframe_ncclteset_build_system_opts() (apps.level0.gpu.nccl_test.reframe_ncclteset_conda_prerun_cmds() (apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
set_conda_prerun_cmds() (*modules.reframe_extras.FetchSourcesBase method*), 94
set_dependencies() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_env_variables() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
set_env_variables() (*apps.level2.rascil.reframe_rascil.RascilBuildTest method*), 87
set_env_variables() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_env_vars() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMklTest method*), 46
set_env_vars() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
set_env_vars() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
set_executable() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgGnuTest method*), 45
set_executable() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMklTest method*), 46
set_executable() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgXlTest method*), 45
set_executable() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49
set_executable() (*apps.level0.cpu.hpl.reframe_hpl.HplMklTest method*), 49
set_executable() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
set_executable() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
set_executable() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
set_executable() (*apps.level0.gpu.babel_stream.reframe_babelstream.BabelStreamTest method*), 60
set_executable() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
set_executable() (*apps.level0.gpu.nccl_test.reframe_ncclteset_git_commit_tag() (apps.level0.gpu.nccl_test.reframe_ncclteset_git_commit_tag() (apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 67
set_executable() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
set_executable() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
set_executable() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest*
set_git_commit_tag()

(*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
set_git_commit_tag() (*modules.reframe_extras.AppBase method*), 95
set_job_env_vars() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49
set_keep_files() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 45
set_keep_files() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_keep_files() (*modules.reframe_extras.SpackBase method*), 95
set_launcher() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
set_launcher() (*apps.level0.gpu.babel_stream.reframe_babelstream.BabelStreamTest method*), 60
set_launcher() (*apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FuncLibTest method*), 70
set_launcher() (*apps.level0.gpu.hippo_func_lib.reframe_funclib_test.PhoenixTest method*), 70
set_launcher() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method*), 67
set_launcher() (*apps.level2.rascil.reframe_rascil.RascilBenchmarkTest method*), 87
set_launcher() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_launcher_opts() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
set_local_launcher() (*modules.reframe_extras.SpackBase method*), 96
set_mkl_env() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMklTest method*), 46
set_my_tags() (*apps.level0.gpu.babel_stream.reframe_babelstream.BabelStreamTest method*), 60
set_nfreq_channels() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_num_mpi_tasks() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
set_num_tasks() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgGnuTest method*), 45
set_num_tasks() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMklTest method*), 46
set_num_tasks() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49
set_num_tasks() (*apps.level0.cpu.hpl.reframe_hpl.HplMklTest method*), 49
set_num_tasks() (*apps.level0.cpu.imb.reframe_imb.ImbAllCoreTests method*), 54
set_num_tasks() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 52
set_num_tasks() (*apps.level0.gpu.babel_stream.reframe_babelstream.BabelStreamTest method*), 60
set_num_tasks() (*modules.reframe_extras.SpackBase method*), 95
set_num_tasks_job() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
set_num_tasks_job() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 81
set_num_tasks_job() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_num_tasks_reservation() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
set_num_tasks_reservation() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
set_num_tasks_reservation() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method*), 67
set_num_tasks_reservation() (*modules.reframe_extras.AppBase method*), 95
set_num_tasks_threads() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgXlTest method*), 45
set_num_threads() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 81
set_omp_threads() (*apps.level0.cpu.hpl.reframe_hpl.HplMklTest method*), 49
set_outfile() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgGnuTest method*), 45
set_outfile() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgXlTest method*), 45
set_output_file() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMklTest method*), 46
set_param_tags() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
set_perf_patterns() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 45
set_perf_patterns() (*apps.level0.cpu.hpl.reframe_hpl.HplMixin method*), 49
set_perf_patterns() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 53
set_perf_patterns() (*apps.level0.cpu.imb.reframe_imb.ImbPingpongTest method*), 53
set_perf_patterns() (*apps.level0.cpu.ior.reframe_ior.IorTest*

method), 57
set_perf_patterns() (apps.level0.cpu.stream.reframe_stream.StreamTest method), 58
set_perf_patterns() (apps.level0.gpu.babel_stream.reframe_babelstream.BabelStreamTest method), 61
set_perf_patterns() (apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method), 65
set_perf_patterns() (apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibDfsTest method), 70
set_perf_patterns() (apps.level0.gpu.hippo_func_lib.reframe_funclib_test.PhaserotTest method), 70
set_perf_patterns() (apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method), 67
set_perf_patterns() (apps.level1.cng_test.reframe_cngtest.CngTest method), 72
set_perf_patterns() (apps.level1.idg_test.reframe_idgtest.IdgTest method), 76
set_perf_patterns() (apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method), 83
set_perf_variables() (apps.level2.rascil.reframe_rascil.RascilTest method), 89
set_postbuild_cmds() (apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibDftBuild method), 70
set_postbuild_cmds() (apps.level1.idg_test.reframe_idgtest.IdgTestBuild method), 75
set_postrun_cmds() (apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method), 64
set_postrun_cmds() (apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method), 81
set_postrun_cmds() (modules.reframe_extras.PerfmonMixin method), 95
set_postrun_cmds() (modules.reframe_extras.SpackBase method), 96
set_prebuild_cmds() (apps.level0.cpu.hpcg.reframe_hpcg.HpcgGnuTest method), 45
set_prebuild_cmds() (apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibDftBuild method), 70
set_prebuild_cmds() (apps.level1.idg_test.reframe_idgtest.IdgTestBuild method), 74
set_prerun_cmds() (apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method), 64
set_reference_values() (apps.level0.cpu.hpl.reframe_hpl.HplMixin method), 49
set_reference_values() (apps.level0.cpu.imb.reframe_imb.ImbMixin method), 53
set_reference_values() (apps.level0.cpu.ior.reframe_ior.IorTest method), 57
set_reference_values() (apps.level0.cpu.stream.reframe_stream.StreamTest method), 59
set_reference_values() (apps.level0.gpu.babel_stream.reframe_babelstream.BabelStream method), 61
set_reference_values() (apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method), 65
set_reference_values() (apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibDft method), 70
set_reference_values() (apps.level0.gpu.hippo_func_lib.reframe_funclib_test.PhaserotTe method), 70
set_reference_values() (apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method), 68
set_reference_values() (apps.level1.cng_test.reframe_cngtest.CngTest method), 73
set_reference_values() (apps.level1.idg_test.reframe_idgtest.IdgTest method), 76
set_reference_values() (apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method), 83
set_reference_values() (apps.level2.rascil.reframe_rascil.RascilTest method), 89
set_sanity_patterns() (apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method), 45
set_sanity_patterns() (apps.level0.cpu.hpl.reframe_hpl.HplMixin method), 48
set_sanity_patterns() (apps.level0.cpu.imb.reframe_imb.ImbMixin method), 53
set_sanity_patterns() (apps.level0.cpu.imb.reframe_imb.ImbPingpongTest

method), 53
set_sanity_patterns() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 57
set_sanity_patterns() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
set_sanity_patterns() (*apps.level0.gpu.babel_stream.reframe_babelstream.BabelStreamTest method*), 60
set_sanity_patterns() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
set_sanity_patterns() (*apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibTestBuild method*), 70
set_sanity_patterns() (*apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibTestRun method*), 70
set_sanity_patterns() (*apps.level0.gpu.hippo_func_lib.reframe_funclib_test.PhasemethodTest method*), 70
set_sanity_patterns() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method*), 67
set_sanity_patterns() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclTestBuild method*), 66
set_sanity_patterns() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
set_sanity_patterns() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
set_sanity_patterns() (*apps.level1.idg_test.reframe_idgtest.IdgTestBuild method*), 75
set_sanity_patterns() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 82
set_sanity_patterns() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTestBuild method*), 81
set_sanity_patterns() (*apps.level2.rascil.reframe_rascil.RascilAndDatasetDownload method*), 87
set_sanity_patterns() (*apps.level2.rascil.reframe_rascil.RascilBuildTest method*), 87
set_sanity_patterns() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_sanity_patterns() (*modules.reframe_extras.FetchSourcesBase method*), 94
set_sanity_patterns() (*modules.reframe_extras.SpackBase method*), 96
set_source_path() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
set_sourcedir() (*apps.level0.gpu.hippo_func_lib.reframe_funclib_test.FunclibTestBuild method*), 70
set_sourcedir() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclTestBuild method*), 66
set_sourcedir() (*apps.level1.idg_test.reframe_idgtest.IdgTestBuild method*), 74
set_sourcedir() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTestBuild method*), 81
set_sourcedir() (*apps.level2.rascil.reframe_rascil.RascilBuildTest method*), 87
set_sources() (*modules.reframe_extras.CachedGitCloneTest method*), 94
set_sourcesdir() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method*), 67
set_subgrid_workers() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 81
set_tags() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 45
set_tags() (*apps.level0.cpu.hpl.reframe_hpl.HplMixin method*), 48
set_tags() (*apps.level0.cpu.imb.reframe_imb.ImbMixin method*), 53
set_tags() (*apps.level0.cpu.ior.reframe_ior.IorTest method*), 56
set_tags() (*apps.level0.cpu.stream.reframe_stream.StreamTest method*), 58
set_tags() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
set_tags() (*apps.level0.gpu.nccl_test.reframe_nccltest.NcclPerfTest method*), 67
set_tags() (*apps.level1.cng_test.reframe_cngtest.CngTest method*), 72
set_tags() (*apps.level1.idg_test.reframe_idgtest.IdgTest method*), 75
set_tags() (*apps.level1.imaging_iotest.reframe_iotest.ImagingIOTest method*), 81
set_tags() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
set_test_cases() (*apps.level0.gpu.gdr_test.reframe_gdr.GpuDirectRdmaTest method*), 64
set_topdir_makefile() (*apps.level0.cpu.hpl.reframe_hpl.HplGnuTest method*), 49
setup_dask_launcher_cmd() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88

skip_tests() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88
slurm_info() (*modules.reframe_extras.SchedulerInfo method*), 97
slurm_node_info() (*in module modules.reframe_extras*), 96
spack.spack_tests.alaska.reframe_alaska module, 27
spack.spack_tests.grid5000.grenoble.dahu.reframe_dahug5k module, 27
spack.spack_tests.grid5000.grenoble.troll.reframe_trollg5k module, 27
spack.spack_tests.grid5000.lyon.gemini.reframe_geminig5k module, 28
spack.spack_tests.grid5000.nancy.gros.reframe_grosg5k module, 28
spack.spack_tests.grid5000.nancy.grouille.reframe_grouilleg5k module, 28
spack.spack_tests.juwels.booster.reframe_juwelsbooster module, 29
spack.spack_tests.juwels.cluster.reframe_juwelscluster module, 29
spack.spack_tests.marconi100.reframe_marconi100 module, 29
SpackBase (*class in modules.reframe_extras*), 95
StreamTest (*class in apps.level0.cpu.stream.reframe_stream*), 58
symlink_dataset() (*apps.level2.rascil.reframe_rascil.RascilTest method*), 88

T

tabulate_last_perf() (*in module modules.utils*), 99
tabulate_partitions() (*in module modules.utils*), 99
test_settings() (*apps.level0.cpu.hpcg.reframe_hpcg.HpcgMixin method*), 44
test_settings() (*apps.level0.cpu.hpl.reframe_hpl.HplMixin method*), 48