developer.skatelescope.org Documentation

Marco Bartolini

Sep 16, 2021

HOME

CHAPTER

ONE

SDP BENCHMARK SUITE

The following sections provide the context, prerequisites and usage of the benchmark suite.

1.1 About

The aim of this package is to create a SDP benchmark suite with the available prototype pipelines that can be tested on different production HPC machines and hardwares. The development of a benchmark suite of this kind was proposed in SKA Computing Hardware Risk Mitigation Plan (SKA-TEL-SKO-0001083). This package automates the deployment of benchmarks, i.e., from compiling the code to parsing the output to get relevant metrics. The package is developed using modular approach and hence, as more prototype codes become available in the future, they can be readily integrated into this benchmark suite.

1.1.1 Available Benchmarks

Currently, the benchmark suite contains only the imaging IO test code developed by Peter Wortmann. More benchmark pipelines will be added to the suite in the future.

1.1.2 Imaging IO test

The aim of this section is to give a high level overview of what imaging IO code does. The input to the code is a sky image and output is the visibility data. The code currently implements only the "predict" part of the imaging algorithm. More details can be found here. There are several input parameters to the benchmark, which can be found in the documentation of the code. More details about the code and algorithms can be found in this memo.

1.2 Prerequisites

The following prerequisites must be installed to use benchmark suite:

- python >= 3.6
- singularity
- git
- OpenMPI (with threading support)

For the Imaging IO test, we should have following dependencies installed.

- cmake

- git lfs
- HDF5 (doesn't need to have parallel support)
- FFTW3

The benchmark suite **will not** install any missing dependencies and therefore it is necessary to install or load all the dependencies before running benchmark suite.

The benchmark suite is tested on both Linux and MacOS. It has not been tested on Windows and there is not guarantee that it will work on Windows systems.

On the MacOS, `homebrew` can be used to install all the listed dependencies.

1.2.1 Installation

Currently, the suite is not packaged into a python library. So, the only way to use is to clone the git repository and running the main script.

To set up the repository and get configuration files:

```
git clone https://gitlab.com/ska-telescope/platform-scripts
cd ska-sdp-benchmark-suite
```

To install all the required python modules

```
pip3 install --user -r requirements.txt
```

If you would like to run unit tests as well, we should install the python modules in `requirements-tests.txt` too.

1.2.2 Configuration Files

The benchmark suite uses configuration files to define various options. The configuration file is divided into two different pieces to abstract away the system based configuration from benchmark related configuration. Samples of these two configuration files can be found in the repository.

The default benchmark configuration file defines global settings and benchmark related settings. The global settings can be overridden by the CLI. More details on the global settings are presented in *Usage* section of the documentation.

The system dependent configuration file defines all the relevant settings based on the system the benchmark suite is running on. It includes names of the modules to load, MPI specific settings like mca parameters, interconnect, *etc*. and batch scheduler settings. Often these settings are defined once for the system and they do not have to be changed after.

It is recommended to copy this file to home directory of the user and modify the config file according to the system settings. By default the benchmark suite reads this config file from project directory (`platform-scripts/ska-sdp-benchmark-suite/sdpbenchmarks/config/.ska_sdp_bms_system_config.yml`) and user home directory (`$HOME/.ska_sdp_bms_system_config.yml`). The config file in the home directory has the precedence over the project directory. Different options in the configuration file are explained in the comments in each of the default files present in the repository.

1.3 Usage

The main script to run the benchmark suite is `sdpbenchrun.py` that can be found in `ska-sdp-benchmark-suite/scripts`. The suite can be launched with a default configuration using following command:

```
python3 sdpbenchrun.py
```

This will run the default config file.

To get all the available options, we should run `python3 sdpbenchrun.py --help`. This gives you following output:

```
usage: sdpbenchrun.py [-h] [-r [RUN_NAME]] [-b BENCHMARKS [BENCHMARKS ...]]
                      [-c [CONFIG]] [-m RUN_MODE [RUN_MODE ...]]
                      [-n NUM_NODES [NUM_NODES ...]] [-d [WORK_DIR]]
                      [-t [SCRATCH_DIR]] [-s] [-p [REPETITIONS]] [-o] [-l]
                      [-e] [-a] [-v] [--version]

SKA SDP Pipeline Benchmark Tests

optional arguments:
-h, --help            show this help message and exit
-r [RUN_NAME], --run_name [RUN_NAME]
Name of the run
-b BENCHMARKS [BENCHMARKS ...], --benchmarks BENCHMARKS [BENCHMARKS ...]
List of benchmarks to run.
-c [CONFIG], --config [CONFIG]
Configuration file to use. If not provided, load
default config file.
-m RUN_MODE [RUN_MODE ...], --run_mode RUN_MODE [RUN_MODE ...]
Run benchmarks in container and/or bare metal.
-n NUM_NODES [NUM_NODES ...], --num_nodes NUM_NODES [NUM_NODES ...]
Number of nodes to run the benchmarks.
-d [WORK_DIR], --work_dir [WORK_DIR]
Directory where benchmarks will be run.
-t [SCRATCH_DIR], --scratch_dir [SCRATCH_DIR]
Directory where benchmark output files will be stored.
-s, --submit_job      Run the benchmark suite in job submission mode.
-p [REPETITIONS], --repetitions [REPETITIONS]
Number of repetitions of benchmark runs.
-o, --randomise       Randomise the order of runs.
-l, --ignore_rem      In case of unfinished jobs, skip remaining jobs on
relaunch.
-e, --export          Export all stdout, json and log files from run_dir and
compresses them.
-a, --show            Show running config and exit.
-v, --verbose         Enable verbose mode. Display debug messages.
--version             show program's version number and exit
```

By default all the CLI arguments can be provided in the configuration file as well. However, the options passed by CLI will override the options given in the configuration file.

1.3.1 Arguments

If not `--run_name` argument is provided, by default, benchmark suite creates a name using hostname and current timestamp. It is advised to give a run name to better organise different benchmark runs. User-defined configuration files can be provided using `--config` option. **Note** that system specific user configuration should be placed in the home directory of the user as suggested in *Configuration Files*. The benchmark suite supports running in the batch submission mode and it can be invoked by `--submit_job` option. All the benchmark related files are placed in the directory specified to `--work_dir` option. This includes source codes, compilation files, std out files, *etc*. Currently only imaging IO test is included in the benchmark suite and depending on the configuration used, it will generate huge amount of data. `--scratch_dir` can be used to write this kind of data. **Important** that users must have permissions to read and write to both `--work_dir` and `--scratch_dir`. It is also important to note that fastest file system available to the machine should be used as `--scratch_dir` as we are interested in high I/O bandwidths. `--repetitions` specify number of times each experiment is repeated. Similarly, using `--randomise` option randomises the order of experiments. This helps to minimise the cache effects.

1.3.2 Example use cases

To run with user defined config `path/to/my_config.yml` and giving a run name `trial_run`

```
python3 sdpbenchrun.py -r trial_run -c path/to/my_config.yml
```

To override run mode in the default config file and run with both bare-metal and container modes

```
python3 sdpbenchrun.py -m bare-metal singularity
```

Multiple inputs to an argument must be delimited by the space.

To run the benchmark with 2, 4 and 8 nodes

```
python3 sdpbenchrun.py -n 2 4 8
```

Note that in such case it is better to use batch submission mode with `-s` option. Else, we will have to make a reservation for 8 nodes and for certain runs, we will use only 2 or 4 nodes and rest of the nodes will be ideal. When `-s` option is added, the benchmark suite will essentially submit jobs to scheduler based on the configuration provided in the system config file. Currently, only SLURM and OAR batch schedulers are supported.

To export the compressed files of the results, use

```
python3 sdpbenchrun.py -e
```

In the similar way, multiple parameters can be defined for the benchmark configuration. More details on how to provide them is documented in the default configuration file.

Based on the provided configuration files and CLI arguments, a parameter space is created and benchmark runs with different combinations in the parameter space are realised.

1.3.3 Job submission mode

By default the benchmark suite runs in interactive mode, which means the benchmarks are run on the current node. It can be run using SLURM too as follows:

```
#!/bin/bash

#SBATCH --time=01:00:00
#SBATCH -J sdp-benchmarks
#SBATCH --no-requeue
#SBATCH --exclusive

module load gnu7 openmpi3 hdf5

workdir=$PWD

# Make sure we have the right working directory
cd $workdir

echo -e "JobID: $SLURM_JOB_ID\n======"
echo "Time: `date`"
echo "Running on master node: `hostname`"
echo "Current directory: `pwd`"
echo "Output directory: $outdir"
echo -e "\nnumtasks=$SLURM_NTASKS, numnodes=$SLURM_JOB_NUM_NODES, OMP_NUM_THREADS=
↪$OMP_NUM_THREADS"

CMD="python3 sdpbenchrun.py -n $SLURM_JOB_NUM_NODES"

echo -e "\nExecuting command:\n==================\n$CMD\n"
eval $CMD

echo -e "\n=================="
echo "Finish time: `date`"
```

This script will run the benchmark suite with default configuration and number of nodes in the SLURM reservation. Sometimes, it is not convenient to run the benchmarks in this way. For example, if we want to do scalability tests, we need to run benchmark with different number of nodes. In this case, one option is to submit several SLURM jobs for different number of nodes. However, this can be done even more in a streamlined fashion. Lets say we want to run on 2, 4, 8 ,16 and 32 nodes. By invoking job submission mode, we can simply use

```
python3 sdpbenchrun.py -r scalability_test -n 2 4 8 16 32 -s
```

on the login node. This will submit SLURM job scripts and we are naming our test run as `scalability_run`. Once all the jobs are finished, if we we rerun the same command again, the benchmark suite will parse the output from the benchmarks and save all the data. In case if not all the jobs have finished, the benchmark suite will collect the results from the jobs that are finished and show the state of unfinished jobs.

In case for some reason one of the jobs failed to finish successfully, the benchmark suite will mark this test as fail. If we run the benchmark suite with the same configuration, the benchmark has already information on which tests have successfully finished and which tests have failed. Eventually, it will run only the tests that have failed. This works both in interactive and job submission mode.

1.3.4 Collection of results

The test results are saved to the `--work-dir` in JSON format. Typically, for `iotest`, the results can be found at `$WORK_DIR/iotest/out`. There will be two sub-directories under this directory named `std-out` and `json-out`. The `std-out` contains the std output from the benchmark runs, whereas `json-out` will have all the information about the benchmark run and relevant metrics parsed from the std out files.

Typical schema of the JSON output file is

```
{
  "benchmark_name": <name of the benchmark test>,
  "benchmark_args": {<arguments of the benchmark test>},
  "batch_script_info": {<info about batch scheduler>},
  "benchmark_metrics_info": {<metrics parsed from benchmark test>}
}
```

All the meta data about the benchmark experiment can be found in the JSON file. It is possible to reproduce the experiment with the data available in the JSON file.

1.4 API Documentation

The following sections provide the API documentation of different files of the benchmark suite.

1.4.1 SDP Benchmark Engine

This module runs the SDP benchmark codes

class sdpbenchmarks.sdpbenchmarkengine.**SdpBenchmarkEngine**(*config=None*)

SDP BENCHMARKS ENGINE

cleanup()

Run the cleanup phase - collect the results from each benchmark

pre_flight()

Perform pre-flight checks.

run()

Run the benchmark at the head of _bench_queue and recurse

start()

Entrypoint for suite.

1.4.2 Imaging I/O test

This module contains the functions to run Imaging IO benchmark.

sdpbenchmarks.imagingiobench.**check_iotest_arguments**(*conf*)

Checks the arguments passed in the config file

Parameters **conf** (*dict*) – A dict containing configuration.

Returns Error code 0 OK, 1 Not OK

Return type int

sdpbenchmarks.imagingiobench.**compile_imaging_iotest**(*conf*)

Compiles Imaging IO test by cloning the code from Git

Parameters **conf** (*dict*) – A dict containing configuration.

Returns 0 OK, 1 Not OK

Return type int

Raises ***ImagingIOTestError*** – An error occurred during compiling of the code

sdpbenchmarks.imagingiobench.**create_bench_conf**(*tag*, *run_mode*, *num_nodes*, *rep*, *rec_set*, *vis_set*, *chunk_sizes*)

Creates a dict containing the parameters for a given run

Parameters

- **tag** (*str*) – Tag of the benchmark run (given at the CLI)
- **run_mode** (*str*) – Mode of the run
- **num_nodes** (*int*) – Number of nodes
- **rep** (*int*) – Number of repetitions
- **rec_set** (*str*) – Image size
- **vis_set** (*str*) – Antennas configuration
- **chunk_sizes** (*str*) – Chunk sizes for time and frequency domains

Returns A dict containing all the parameters

Return type dict

sdpbenchmarks.imagingiobench.**extract_metrics**(*filename*, *mpi_startup*)

Extract data transfer metrics from benchmark output

Parameters

- **filename** (*str*) – Name of the stdout file to parse
- **mpi_startup** (*str*) – Startup command of MPI (mpirun or srun)

Returns A dict containing streamer, producer and writer metrics

Return type dict

sdpbenchmarks.imagingiobench.**get_command_to_execute_bench**(*conf*, *param*)

This function forms the command string to be executed

Parameters

- **conf** (*dict*) – A dictionary containing configuration parameters
- **param** (*dict*) – A Parameter dict with all relevant arguments for the run

Returns Complete command string

Return type str

sdpbenchmarks.imagingiobench.**get_mpi_args**(*conf*, *num_nodes*, *num_omp_threads*, *num_processes*)

Extract all MPI specific arguments and form a string

Parameters

- **conf** (*dict*) – A dict containing the config parameters of benchmark
- **num_nodes** (*int*) – Number of nodes
- **num_omp_threads** (*int*) – Number of OpenMP threads

- **num_processes** (*int*) – Number of MPI processes

Returns A string containing all MPI related arguments

Return type str

sdpbenchmarks.imagingiobench.**get_num_processes**(*conf*, *rec_set*, *num_nodes*)
Estimates producers, streamers OpenMP threads.

Parameters

- **conf** (*dict*) – A dict containing configuration settings
- **rec_set** (*str*) – image parameters
- **num_nodes** (*int*) – number of nodes

Returns total cpu cores (only physical) on all nodes combined, threads per each core, number of OpenMP threads, number of producers, number of MPI processes

Return type list

sdpbenchmarks.imagingiobench.**get_telescope_config_settings**(*param*)
This function returns the telescope related configurations

Parameters **param** (*dict*) – Parameter class with all relevant arguments for the run

Returns String with time, frequency and w-stacking arguments

Return type str

sdpbenchmarks.imagingiobench.**prepare_iotest**(*conf*)
Prepare IO Imaging Benchmark installation.

Parameters **conf** (*dict*) – A dict containing configuration.

Returns 0 OK , 1 Not OK

Return type int

sdpbenchmarks.imagingiobench.**print_key_stats**(*run_prefix*, *metrics*)
This prints the key metrics to stdout

Parameters

- **run_prefix** (*str*) – Prefix of the run
- **metrics** (*dict*) – A dict containing stats of the Imaging IO bench

sdpbenchmarks.imagingiobench.**run_iotest**(*conf*)
Run Imaging IO test.

Parameters **conf** (*dict*) – A dict containing configuration.

Returns < 0 all runs failed, = 0 all runs passed, > 0 few runs passed

Return type int

Raises **Exception** – An error occurred in building parameter sweep, running or submitting job

1.4.3 Utility Functions

This module contains the utility functions.

class sdpbenchmarks.utils.**ParamSweeper**(*persistence_dir*, *params=None*, *name=None*, *randomise=True*)

This class is inspired from execo library (http://execo.gforge.inria.fr/doc/latest-stable/) except this is very simplified version of the original. The original one is developed for large scale experiments and thread safety. Here what we are interested is the state of each run that can be tracked and remembered when launching the experiments.

done(*combination*)
Marks the iterable as done

get_done()
Returns the iterable of finished runs

get_ignored()
Returns the iterable of ignored runs

get_inprogress()
Returns the iterable of runs in progress

get_next()
Returns the iterable next run

get_remaining()
Returns the iterable of remaining to iterate on

get_skipped()
Returns the iterable of skipped runs

get_submitted()
Returns the iterable of submitted jobs (when batch scheduler is used)

get_sweeps()
Returns the iterable of what to iterate on

ignore(*combination*)
Marks the iterable as ignored

set_sweeps(*params=None*)
This method sets the sweeps to be performed

skip(*combination*)
Marks the iterable as skipped

submit(*combination*)
Marks the iterable as submitted

sdpbenchmarks.utils.**create_scheduler_conf**(*conf*, *param*, *bench_name*)
Prepares a dict with parameters that will create a job submit

Parameters

- **conf** (*dict*) – A dict containing configuration.
- **param** (*dict*) – A dict containing all parameters for the run
- **bench_name** (*str*) – Name of the benchmark

Returns A dict with parameters that need to submit a job file

Return type dict

Raises ***KeyNotFoundError*** – An error occurred while looking for a key in conf or param

sdpbenchmarks.utils.**exec_cmd**(*cmd_str*)
This method executes the given command

Parameters **cmd_str** (*str*) – Command to execute

Returns A subprocess.run output with stdout, stderr and return code in the object

Raises ***ExecuteCommandError*** – An error occurred during execution of command

sdpbenchmarks.utils.**execute_command_on_host**(*cmd_str*, *out_file*)
This method executes the job on host

Parameters

- **cmd_str** (*str*) – Command to execute
- **out_file** (*str*) – Name of the stdout/stderr file

Raises ***ExecuteCommandError*** – An error occurred during execution of command

sdpbenchmarks.utils.**execute_job_submission**(*cmd_str*, *run_prefix*)
This method submits to SLURM job scheduler and returns job ID

Parameters

- **cmd_str** (*str*) – Command string to be submitted
- **run_prefix** (*str*) – Prefix of the bench run

Returns ID of the submitted job or raises exception in case of failure

Return type int

Raises ***JobSubmissionError*** – An error occurred during the job submission

sdpbenchmarks.utils.**get_job_status**(*conf*, *job_id*)
Returns the status of the batch job

Parameters

- **conf** (*dict*) – A dict containing configuration of batch scheduler
- **job_id** (*int*) – ID of the job

Returns current status of the job

Return type str

sdpbenchmarks.utils.**get_project_root**()
Get root directory of the project

Returns Full path of the root directory

Return type str

sdpbenchmarks.utils.**get_sockets_cores**(*conf*)
Returns the number of sockets and cores on the compute nodes. For interactive runs, lscpu can be used to grep the info. When using the script to submit jobs from login nodes, lscpu cannot be used and sinfo for a given partition is used.

Parameters **conf** (*dict*) – A dict containing configuration settings

Returns Number of sockets on each node, number of physical cores on each node (num_socekts * num_cores per socket), number of threads inside each core

Return type list

Raises ***KeyNotFoundError*** – An error occurred if key is not found in g5k dict that contains lscpu info for different clusters

sdpbenchmarks.utils.**load_modules**(*module_list*)
This function purges the existing modules and loads given modules

Parameters **module_list** (*str*) – List of modules to load

sdpbenchmarks.utils.**log_failed_cmd_stderr_file**(*output*)
This method dumps the output to a file when command execution fails

Parameters **output** (*str*) – stdout and stderr from execution of command

Returns Path of the file

Return type str

sdpbenchmarks.utils.**pull_image**(*uri*, *container_mode*, *path*)
This pulls the image from the registry. It returns error if image is not pullable

Parameters

- **uri** (*str*) – URI of the image
- **container_mode** (*str*) – Docker or Singularity container
- **path** (*str*) – Path where image needs to be saved (only for singularity). It will overwrite if image already exists.

Returns 0 - OK 1 - Not OK

Return type int

sdpbenchmarks.utils.**reformat_long_string**(*ln_str*, *width=70*)
This method reformats command string by breaking it into multiple lines.

Parameters

- **ln_str** (*str*) – Long string to break down
- **width** (*int*) – Width of each line (Default is 70)

Returns Same string in multiple lines to ease readability

Return type str

sdpbenchmarks.utils.**standardise_output_data**(*bench_name*, *conf*, *param*, *metrics*)
This method saves all the data of the benchmark run in json format. The aim is to put all the info tp be able to reproduce the run.

Parameters

- **bench_name** (*str*) – Name of the benchmark
- **conf** (*dict*) – A dict file containing all configuration info
- **param** (*dict*) – A parameter dict file
- **metrics** (*dict*) – Dict file containing all the metric data

Raises ***KeyNotFoundError*** – An error occurred while looking for a key in conf or param

sdpbenchmarks.utils.**submit_job**(*conf*, *job_id*)
This method submits job to the scheduler

Parameters

- **conf** (*dict*) – A dict containing all parameters needed for job submission

- **job_id** (*int*) – Job ID of the previous job. In case of dependent jobs, this is necessary

Returns ID of the submitted job

Return type int

sdpbenchmarks.utils.**sweep**(*parameters*)
This method accepts a dict with possible values for each parameter and creates a parameter space to sweep

Parameters **parameters** (*dict*) – A dict containing parameters and its values

Returns All possible combinations of the parameter space

Return type list

sdpbenchmarks.utils.**which**(*cmd*, *modules*)
This function loads the given modules and returns of path of the requested binary if found or None

Parameters

- **cmd** (*str*) – Name of the binary
- **modules** (*str*) – modules to load

Returns Path of the binary or None if not found

Return type str

sdpbenchmarks.utils.**write_oar_job_file**(*conf_oar*)
This method writes a OAR job file to submit with sbatch

Parameters **conf_oar** (*dict*) – A dict containing all OAR job parameters

Returns Name of the file

Return type str

Raises ***JobScriptCreationError*** – An error occurred in creation of job script

sdpbenchmarks.utils.**write_slurm_job_file**(*conf_slurm*)
This method writes a SLURM job file to submit with sbatch

Parameters **conf_slurm** (*dict*) – A dict containing all SLURM job parameters

Returns Name of the file

Return type str

Raises ***JobScriptCreationError*** – An error occurred in creation of job script

sdpbenchmarks.utils.**write_tgcc_job_file**(*conf_slurm*)
This method writes a SLURM job file for TGCC Irene machine to submit with ccc_msub

Parameters **conf_slurm** (*dict*) – A dict containing all SLURM job parameters

Returns Name of the file

Return type str

Raises ***JobScriptCreationError*** – An error occurred in creation of job script

1.4.4 Exceptions

This module contains the custom exceptions defined for benchmark suite.

exception sdpbenchmarks.exceptions.**BenchmarkError**
Error in benchmarking the codes

exception sdpbenchmarks.exceptions.**ExecuteCommandError**
Command execution exception

exception sdpbenchmarks.exceptions.**ExportError**
Error in exporting results

exception sdpbenchmarks.exceptions.**ImagingIOTestError**
Error in Imaging IO test benchmakr run

exception sdpbenchmarks.exceptions.**JobScriptCreationError**
Error in generating the job script to submit

exception sdpbenchmarks.exceptions.**JobSubmissionError**
Job submission to batch scheduler failed

exception sdpbenchmarks.exceptions.**KeyNotFoundError**
Key missing in the dict

CHAPTER

TWO

SDP PERFORMANCE METRIC MONITORING TOOL

The following sections provide the context, prerequisites and usage of the cpu metric monitoring toolkit.

2.1 About

The aim of this toolkit is to monitor CPU related performance metrics for SDP pipelines/workflows in a standardised way. Often different HPC clusters have different ways to monitor and report performance related metrics. We will have to adopt our scripts to each machine to be able to extract this data. This toolkit address this gap by providing an automatic and standardised way to collect and report performance metrics. As of now, the toolkit can collect both system wide and job related metrics during the job execution on all the nodes in a multi-node job, save them to the disk (in JSON and excel formats) and generate a job report with plots from different metrics.

2.1.1 Idea

As submitting and controlling jobs on HPC machines are often realised by batch schedulers, this toolkit is based on workload managers. Along with SLURM, one of the commonly used batch scheduler in the HPC community, the toolkit can handle PBS and OAR schedulers. SLURM's `scontrol listpids` command gives the Process IDs (pids) of different job steps. Similarly, OAR and PBS provides tools to capture PIDs of jobs. By getting the pid of the main step job, we can monitor different performance metrics by using combination of python's `psutil` package, proc files and `perf stat` commands. The toolkit is developed in Python.

2.1.2 Available metrics

Currently, the toolkit reports following data or metrics:

- Hardware and software metadata of all the compute nodes in the reservation.
- CPU related metrics like CPU usage, memory consumption, system-wide network I/O traffic, Infiniband traffic (if supported), meta data of the processes, *etc*.
- `perf` events like hardware and software events, hardware cache events and different types of FLOP counts.

All these metrics are gathered and saved in JSON and/or excel formats for easy readability.

2.2 Prerequisites

The following prerequisites must be installed to use monitoring toolkit:

- python >= 3.7
- git

2.2.1 Installation

Currently, the way to install this toolkit is to git clone the repository and then install it.

To set up the repository and get configuration files:

```
git clone https://gitlab.com/ska-telescope/platform-scripts
cd ska-sdp-monitor-cpu-metrics
```

To install all the required python modules

```
pip3 install --user -r requirements.txt
```

And finally, install the package using

```
python3 setup.py install
```

Another way is to use `--editable` option of `pip` installation as follows:

```
pip install "--editable=git+https://gitlab.com/ska-telescope/platform-scripts.
↪git@master#egg=ska-sdp-monitor-metrics&subdirectory=ska-sdp-monitor-cpu-metrics"
```

This command clones the git repository and runs `python3 setup.py install`. This line can be directly added to the `conda` environment files.

2.3 Usage

As stated in the introduction, currently the toolkit is made to work with SLURM and OAR job reservations. The main script to run the benchmark suite is `sdpmonitormetrics`. The launch script has following options:

```
Monitor CPU and Performance metrics for SDP Workflows

optional arguments:
-h, --help                                          show this help message and exit
-d [], --save_dir []                                Directory where metrics will be␣
↪saved. This directory should be available
                                                    from all compute nodes. Default␣
↪is $PWD/job-metrics-$jobID.
-f [PREFIX], --prefix [PREFIX]                      Prefix to add to the exported␣
↪files
-i [SAMPLING_FREQ], --sampling_freq [SAMPLING_FREQ]  Sampling interval to collect␣
↪metrics. Default value is 30 seconds.
-c [CHECK_POINT], --check_point [CHECK_POINT]       Checking point time interval.␣
↪Default value is 900 seconds.
-p, --perf_metrics                                  Collect perf metrics. Default␣
↪metrics that will be collected are SW/HW meta
```

(continues on next page)

(continued from previous page)

```
                                                    data and CPU metrics.
-r, --gen_report                                    Generate plots and job report
-e, --export_xlsx                                   Export results to excel file␣
↪with job ID as sheet name
-b, --export_db                                     Export results to a SQL database
-v, --verbose                                       Enable verbose mode. Display␣
↪debug messages.
--version                                           show program's version number␣
↪and exit

-------------------------------------------------------------
```

2.3.1 Arguments

The option `--save_dir` specifies the folder where results are saved. **It is important** that this folder should be accessible from all nodes in the SLURM reservation. Typically, NFS mounted home directories can be used for this directory. The `--sampling_freq` option tells the toolkit how frequently it should poll for collecting metrics. The default value is 30 sec. The more often we collect the metrics, the more overhead the toolkit will have on the system usage. By default, the toolkit only collects software/hardware metadata and CPU related metrics. If we want `perf stat` metrics, we should give `-p` option on the CLI. The toolkit is capable of check pointing the data and the time period between check points can be configured using `--check_point` flag. If the user wants to generate a job report with plots from different metrics, `-r` option must be passed on the CLI.

We can also ask for metric data in excel sheet by passing `--export_xlsx` flag. Typically, this file will be updated between different runs with job ID as sheet name. Thus, this excel file will have all the results in one place to ease the process of plotting. Similarly, `--export_db` flag tells the toolkit to export the metric data into a SQL database. The tables in SQL data base are named using the convention `cpu_metrics_<job_id>` for CPU metrics and `perf_metrics_<job_id>` for perf metrics, where `<job_id>` is the ID of the batch job.

The toolkit runs in silent mode, where all the stdout is logged to a log file. This is done to not to interfere with the main job step stdout. Typically, the log file can be found in `$SAVE_DIR/ska_sdp_monitoring_metrics.log`.

2.3.2 Monitored metrics

Software and hardware metadata

Currently, the toolkit reports the software versions of docker, singularity, python, OpenMPI and OS. For the hardware metadata, we parse the output of linux command `lscpu` to report several informations. In addition, information about system memory is also reported.

CPU related metrics

The metrics reported in this part are both process specific and system wide. The process pid that is captured from the SLURM step job is used to monitor process specific metrics. Some metrics like network I/O counters are only reported system wide. The metrics reported are as follows:

- CPU time from parent and child processes (Process specific).
- CPU percent from parent and child processes. Typically, for a multi-threaded job, this value will be more than 100% as it gives sum of the cpu usages from all cores, where the process is running (Process specific).
- CPU percent (System wide)
- Network I/O counters, which includes send/receive bytes and packets (System wide).

- Infiniband I/O traffic (System wide, if supported) which includes send/receive bytes and packets.
- Memory consumption which includes RSS, VMS, USS, shared and swap usage (USS gives process specific memory consumption, process specific) in bytes.
- Memory consumption in percentage (Process specific).
- Memory bandwidth (read only, process specific) in bytes/second.
- I/O statistics (Process specific), which includes read/write bytes.
- RAPL power metrics (System wide) in micro Joules.
- Number of threads of parents and children (Process specific).
- Timestamp list

Each metric is reported as list of values that correspond to the timestamps. Memory bandwidth is estimated using OFFCORE_RESPONSE perf metrics. We can only get the read bandwidth using this perf counter. Memory bandwidth reported with this toolkit **should only be** regarded as a proxy to the "actual" bandwidth.

Note: All metrics are reported as raw data without any post-processing. For instance, to estimate the power consumption from the RAPL metrics, we need to do forward differencing of the metric data and divide it by sampling time to get power consumption in micro Watts. Similarly, for network and Infiniband I/O statistics, we will have to do similar computation to get bandwidths.

Perf stat metrics

Perf stat metrics are monitored by executing

```
perf stat -e <event_list> -p <process_pids> sleep <collection_time>
```

Currently only broadwell and skylake chips are supported. More intel micro architectures and also AMD ones will be added to the toolkit. Note that the supported perf events differ for different micro architectures and so, not all the listed events might be available for all the cases.

Hardware events:

- cycles
- instructions
- cache-misses
- cache-references
- branches
- branch-misses

Software events:

- context-switches
- cpu-migrations

Caches:

- L2 cache bandwidth
- L3 cache bandwidth

FLOPS:

- Single precision FLOPS
- Double precision FLOPS

Hardware and software events are named perf events in `perf stat` and available in both Intel and AMD chips. The cache bandwidths and FLOPS have processor specific event codes. These events are taken from likwid project. Most of these events are claimed to be tested on different processors from the project maintainers.

Note: Along with the raw counter numbers, derived counters are also provided in the metric data. FLOPS are provided in MFLOPS/second, whereas bandwidths are provided in MB/s.

2.3.3 Example use cases

Typical use case is shown as follows:

```
#!/bin/bash

#SBATCH --time=00:30:00
#SBATCH -J sdp-metrics-test
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --no-requeue
#SBATCH --exclusive
#SBATCH --output="slurm-%J.out"

WORK_DIR=/path/to/matmul/executable
MON_DIR=/path/to/ska-sdp-montor-cpu-metrics

# Make sure we have the right working directory
cd $WORK_DIR

echo -e "JobID: $SLURM_JOB_ID\n======"
echo "Time: `date`"
echo "Running on master node: `hostname`"
echo "Current directory: `pwd`"

srun -n $SLURM_JOB_NUM_NODES --ntasks-per-node 1 sdpmonitormetrics &
# mpirun --map-by node -np $SLURM_JOB_NUM_NODES sdpmonitormetrics &
mpirun -np ${SLURM_JOB_NUM_NODES} ./matmul 2000

wait
```

This simple SLURM script reserves two nodes and runs matrix multiplication using `mpirun`. Now looking at the line immediately preceding `mpirun` we notice that we are running `sdpmonitormetrics` script using `srun` as a background process. `srun` launches the `sdpmonitormetrics` script on all nodes in the reservation, where it runs in the background. The first step the script does is to get the process pid of the main step job (in this case `mpirun -np ${SLURM_JOB_NUM_NODES} ./matmul 2000`) and collects the metrics for this process and its child. Once the process is terminated, the script does some post processing to merge all the results, make plots and generate report. **It is important** to have a `wait` command after the main job, else the toolkit script wont be able to do post-processing and save the results. The main step job can be launched with either `mpirun` or `srun`. Similarly, the toolkit can be launched with either of them.

Sometimes, processes will not tear down cleanly even after the main job has finished. For example, this case can arise when dask is used as a parallelisation framework and scheduler is not stopped after the main job. The toolkit monitors the process id of the main job and keeps monitoring till it is killed. So, in this situation will keep monitoring till the

end of reservation time. To avoid this issue, we can use a Inter Process Communicator (IPC) using a simple file. After the main job, we can add a line `echo "FINISHED" > .ipc-$SLURM_JOB_ID` and the toolkit keeps reading this `.ipc-$SLURM_JOB_ID` file and once it reads `FINISHED`, it will stop monitoring. This is very simple and portable solution for this kind of problem. Also, we are adding a `wait` command, the SLURM job will wait till the end of the reservation period in this case. To avoid such condition, we can wait for exclusively only monitor job by capturing its PID.

```
#!/bin/bash

#SBATCH --time=00:30:00
#SBATCH -J sdp-metrics-test
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --no-requeue
#SBATCH --exclusive
#SBATCH --output="slurm-%J.out"

WORK_DIR=/path/to/matmul/executable
MON_DIR=/path/to/ska-sdp-montor-cpu-metrics

# Make sure we have the right working directory
cd $WORK_DIR

echo -e "JobID: $SLURM_JOB_ID\n======"
echo "Time: `date`"
echo "Running on master node: `hostname`"
echo "Current directory: `pwd`"

srun -n $SLURM_JOB_NUM_NODES --ntasks-per-node 1 sdpmonitormetrics &
export MON_PID=$!
# mpirun --map-by node -np $SLURM_JOB_NUM_NODES sdpmonitormetrics &
mpirun -np ${SLURM_JOB_NUM_NODES} ./matmul 2000
echo "FINISHED" > .ipc-$SLURM_JOB_ID

wait $MON_PID
```

This sample script shows how to use the toolkit for dask jobs.

```
#!/bin/bash

#SBATCH --time=00:30:00
#SBATCH -J sdp-metrics-test
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --mail-type=FAIL
#SBATCH --no-requeue
#SBATCH --exclusive
#SBATCH --output="slurm-%J.out"

MON_DIR=/path/to/ska-sdp-montor-cpu-metrics

SCHEFILE=$PWD/${SLURM_JOB_ID}.dasksche.json
WORKSPACE=$PWD/dask-worker-space

rm -rf $SCHEFILE
rm -rf $WORKSPACE

export DASK_SCHEDULER_FILE="$SCHEFILE"
```

(continues on next page)

(continued from previous page)

```
#! Set up python
echo -e "Running python: `which python`"
echo -e "Running dask-scheduler: `which dask-scheduler`"

cd $SLURM_SUBMIT_DIR
echo -e "Changed directory to `pwd`.\n"

JOBID=${SLURM_JOB_ID}
echo ${SLURM_JOB_NODELIST}

scheduler=$(scontrol show hostnames $SLURM_JOB_NODELIST | uniq | head -n1)

echo "run dask-scheduler"
ssh ${scheduler} python3 `which dask-scheduler` --port=8786 --scheduler-file=
↪$SCHEFILE  &

sleep 5

echo "Monitoring script"
srun -n $SLURM_JOB_NUM_NODES --ntasks-per-node 1 python3 sdpmonitormetrics &
export MON_PID=$!

echo "run dask-worker"
srun -n ${SLURM_JOB_NUM_NODES} python3 `which dask-worker` --nanny --nprocs 4 --
↪interface ib0 --nthreads 1\
--memory-limit 200GB --scheduler-file=$SCHEFILE ${scheduler}:8786 &

echo "Scheduler and workers now running"

#! We need to tell dask Client (inside python) where the scheduler is running
echo "Scheduler is running at ${scheduler}"

CMD="python3 src/cluster_dask_test.py ${scheduler}:8786 | tee cluster_dask_test.log"

eval $CMD
echo "FINISHED" > .ipc-$SLURM_JOB_ID

wait $MON_PID
```

The above script monitors the dask workers. **Note that** dask workers and scheduler should be teared down cleanly for this approach to work. If not, use the approach provided in the above example to wait for monitor job by capturing its PID.

These scripts are the source file for matrix multiplication is available in the repository for testing purposes in `ska-sdp-monitor-cpu-metrics/tests` folder.

In the case of PBS jobs, we should do a little hack for the toolkit to work. We have not tested the toolkit on production ready PBS cluster. From the local tests, it is found that the environment variable `PBS_NODEFILE` is only available on the first node in the reservation. We need this file to be accessible from all nodes for the toolkit to work properly. So, the hack is to copy this nodefile to the local directory (which is often NFS mounted home directory where all nodes can access) and set a new environment variable called `PBS_NODEFILE_LOCAL` and export to all nodes. Now the toolkit looks for this variable and reads node list from this variable. This can be done in following way:

```
#!/bin/bash

#PBS -N metrics-test
```

(continues on next page)

(continued from previous page)

```
#PBS -V
#PBS -j oe
#PBS -k eod
#PBS -q workq
#PBS -l walltime=01:00:00
#PBS -l select=2:ncpus=6:mpiprocs=12

cd /home/pbsuser

# We need to copy the nodefile to CWD as it is not available from all compute nodes␣
↪in the reservation
cp $PBS_NODEFILE nodefile

# Later we export a 'new' env variable PBS_NODEFILE_LOCAL using mpirun to the␣
↪location of copied local nodefile
mpirun --map-by node -np 2 -x PBS_NODEFILE_LOCAL=$PWD/nodefile sdpmonitormetrics -i 5␣
↪-v -r -e &
sleep 2
mpirun --map-by node -np 2 ./matmul 1500

wait
```

2.3.4 Output files

Upon successful completion of the job and monitoring task, we will find following files inside the metrics directory that is created by the toolkit.

```
job-metrics-{job-id}
├── data
│   ├── cpu_metrics.json
│   ├── meta_data.json
│   └── perf_metrics.json
├── job-report-{job-id}.pdf
└── plots
    ├── bytes_recv_per_node.png
    ├── bytes_recv_total.png
    ├── bytes_sent_per_node.png
    ├── bytes_sent_total.png
    ├── core_power_per_node.png
    ├── core_power_total.png
    ├── cpu_percent_sys_average.png
    ├── cpu_percent_sys_per_node.png
    ├── dram_power_per_node.png
    ├── dram_power_total.png
    ├── memory_bw_average.png
    ├── memory_bw_per_node.png
    ├── package_power_per_node.png
    ├── package_power_total.png
    ├── packets_recv_per_node.png
    ├── packets_recv_total.png
    ├── packets_sent_per_node.png
    ├── packets_sent_total.png
    ├── port_rcv_data_per_node.png
    ├── port_rcv_data_total.png
    ├── port_rcv_packets_per_node.png
```

(continues on next page)

(continued from previous page)

```
    ├── port_rcv_packets_total.png
    ├── port_xmit_data_per_node.png
    ├── port_xmit_data_total.png
    ├── port_xmit_packets_per_node.png
    ├── port_xmit_packets_total.png
    ├── uncore_power_per_node.png
    ├── uncore_power_total.png
    ├── uss_average.png
    └── uss_per_node.png
```

Typically, `job-id` is job ID of the SLURM job, `node-0-hostname` is the hostname of first node in the reservation and so on. The JSON files `meta_data.json` and `cpu_metrics.json` have metric data from all the hosts. Folder `raw_node` contains same metrics but for each node separately. All the generated plots of the metrics are placed in `plots` folder. Finally, a job report `report-{job-id}.pdf` is generated will all the plots included. If the export to excel option is asked, excel files are also generated and placed in the save directory.

The schema for the `cpu_metrics.json` file is shown as follows:

```
 {
"type": "object",
"required": [],
"properties": {
  "host_names": {
    "type": "array",
    "items": {
      "type": "string"
    }
  },
  "node-0-hostname": {
    "type": "object",
    "required": [],
    "properties": {
      "child_proc_md": {
        "type": "array",
        "items": {
          "type": "string"
        }
      },
      "cpu_percent": {
        "type": "array",
        "items": {
          "type": "number"
        }
      },
      "cpu_percent_sys": {
        "type": "array",
        "items": {
          "type": "number"
        }
      },
      "cpu_time": {
        "type": "array",
        "items": {
          "type": "number"
        }
      },
      "ib_io_counters": {
```

(continues on next page)

(continued from previous page)

```
        "type": "object",
        "required": [],
        "properties": {
          "port_rcv_data": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "port_rcv_packets": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "port_xmit_data": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "port_xmit_packets": {
            "type": "array",
            "items": {
              "type": "number"
            }
          }
        }
      },
      "io_counters": {
        "type": "object",
        "required": [],
        "properties": {
          "read_bytes": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "read_count": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "write_bytes": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "write_count": {
            "type": "array",
            "items": {
              "type": "number"
            }
          }
```

(continues on next page)

(continued from previous page)

```
        }
      },
      "memory_full_info": {
        "type": "object",
        "required": [],
        "properties": {
          "swap": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "uss": {
            "type": "array",
            "items": {
              "type": "number"
            }
          }
        }
      },
      "memory_info": {
        "type": "object",
        "required": [],
        "properties": {
          "rss": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "shared": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "vms": {
            "type": "array",
            "items": {
              "type": "number"
            }
          }
        }
      },
      "memory_percent": {
        "type": "array",
        "items": {
          "type": "number"
        }
      },
      "net_io_counters": {
        "type": "object",
        "required": [],
        "properties": {
          "bytes_recv": {
            "type": "array",
            "items": {
```

(continues on next page)

(continued from previous page)

```
            "type": "number"
          }
        },
        "bytes_sent": {
          "type": "array",
          "items": {
            "type": "number"
          }
        },
        "packets_recv": {
          "type": "array",
          "items": {
            "type": "number"
          }
        },
        "packets_sent": {
          "type": "array",
          "items": {
            "type": "number"
          }
        }
      }
    },
    "num_fds": {
      "type": "array",
      "items": {
        "type": "number"
      }
    },
    "num_threads": {
      "type": "array",
      "items": {
        "type": "number"
      }
    },
    "parent_proc_md": {
      "type": "object",
      "required": [],
      "properties": {}
    },
    "rapl_powercap": {
      "type": "object",
      "required": [],
      "properties": {
        "core-0": {
          "type": "array",
          "items": {
            "type": "number"
          }
        },
        "uncore-0": {
          "type": "array",
          "items": {
            "type": "number"
          }
        },
        "dram-0": {
```

(continues on next page)

(continued from previous page)

```
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "package-0": {
            "type": "array",
            "items": {
              "type": "number"
            }
          }
        }
      },
      "time_stamps": {
        "type": "array",
        "items": {
          "type": "number"
        }
      }
    }
  },
  "sampling_frequency": {
    "type": "number"
  }
}
```

where the field `host_names` contains all the names of the nodes in the SLURM reservation. The CPU metric data is organised for each host separately, where data for field `node-0-hostname` corresponds to data for `node-0` in the reservation and so on. The `perf` metrics data is also organised in a similar way.

For example, if we want to inspect the memory consumption in percentage on, say `example-host-0` node, we can query it simply as `cpu_metrics['example-host-0']['memory_percent']` in python. This gives us list of values for each timestamp given in `cpu_metrics['example-host-0']['timestamps']`. Note that timestamps for different hosts are saved separately as there can be synchronisation issues between different nodes in the cluster. It is also worth noting that integer timestamps are used and so, monitoring with a frequency of less than a second is not possible.

2.4 API Documentation

The following sections provide the API documentation of different files of the benchmark suite.

2.4.1 Set up monitoring

This file does pre-processing steps of the metric data collection

class monitormetrics.prepostmonitoring.setupmonitoring.**SetUpMonitoring**(*config*)

Set up all the required steps to start monitoring

acquire_lock()

Acquires lock by locking the created file

create_ipc_file()

Create Inter process communicator (IPC) file. It will be used to signal the toolkit that main job has ended

create_lock_file()
Creates a lock file

create_node_list()
Create a list of node names

find_launcher_type()
Find which launcher is used to launch MPI jobs

get_job_id()
Find the workload manager and job ID. Exit if none found

prepare_dirs()
Make directories to save the results

prepare_file_paths()
Prepare file paths for different result files

release_lock()
Release lock file

shutdown()
Finish monitoring

start()
Preprocessing steps of data collection

2.4.2 SW and HW metadata

This file contains the class to extract software and hardware metadata

class monitormetrics.getmetadata.**GetSWHWMetadata**(*config*)
Engine to extract metadata

collect()
Collect all metadata.

static collect_cpu_info()
Collect all relevant CPU information.

collect_hmd()
Collect Hardware MetaData (HW_MD).

static collect_memory_info()
Collect system memory.

collect_smd()
Collect Software MetaData (SW_MD)

2.4.3 Monitor metrics

This file initiates the script to extract real time perf metrics

class monitormetrics.monitormetrics.**MonitorPerformanceMetrics**(*config*)
Engine to extract performance metrics

get_job_pid()
This method calls function to get job PID

start_collection()
Start collecting CPU metrics. We use multiprocessing library to spawn different processes to monitor cpu and perf metrics

2.4.4 CPU metrics

This file initiates the script to extract real time cpu metrics

class monitormetrics.cpumetrics.cpumetrics.**MonitorCPUMetrics**(*config*)
Engine to monitor cpu related metrics

add_ib_counters_to_dict()
Add IB counters to base dict

add_mem_bw_to_dict()
Add memory bandwidth to base dict

add_metrics_cpu_parameters()
This method adds metrics key/value pair in cpu parameter dict

add_rapl_domains_to_dict()
Add RAPL domain names to base dict

add_timestamp()
This method adds timestamp to the data

check_availability_ib_rapl_membw()
This method checks if infiniband and RAPL metrics are available

check_metric_data()
This method checks if all the metric data is consistent with number of timestamps

dump_metrics()
Dump metrics to JSON file and re-initiate cpu_metrics dict

get_cpu_percent()
This method gives CPU percent of parent and its childs

get_cpu_time_from_parent_and_childs()
This method gets cumulative CPU time from parent and its childs

get_cpu_usage()
This method gets all CPU usage statistics

get_cumulative_metric_value(*metric_type*)
This method gets cumulative metric account for all childs for a given metric type

get_energy_metrics()
This method gets energy metrics from RAPL powercap interface

get_ib_io_counters()
This method gets the Infiniband port counters

get_memory_bandwidth()
This method returns memory bandwidth based on perf LLC load misses event

get_memory_usage()
This method gets memory usage

get_metrics_data()
Extract metrics data

get_misc_metrics()
This method gets IO, file descriptors and thread count

get_network_traffic()
Get network traffic from TCP and Infiniband (if supported)

get_sys_wide_net_io_counters()
This method gets the system wide network IO counters

initialise_cpu_metrics_params()
This method initialises the CPU metric related parameters

run()
This method extracts the cpu related metrics for a given pid

2.4.5 Perf metrics

This file initiates the script to extract real time perf metrics

class monitormetrics.cpumetrics.perfmetrics.**MonitorPerfEventsMetrics**(*config*)
Engine to extract performance metrics

add_timestamp()
This method adds timestamp to the data

compute_derived_metrics()
This method computes all the derived metrics from parsed perf counters

dump_avail_perf_events()
Dump the available perf event list for later use

dump_metrics()
Dump metrics to JSON file and re-initiate perf_metrics dict

get_list_of_pids()
This method gets the list of pids to monitor by adding children pids to parents

initialise_perf_metrics_data_dict()
This method initialises the perf metric related parameters

make_perf_command()
This method make the perf command to run

static match_perf_line(*pattern*, *cmd_out*)
This method builds perf output pattern and get matching groups

parse_perf_cmd_out(*cmd_out*)
This method parses perf command output and populate perf data dict with counter values

post_parsing_steps()
Steps to be made after parsing all metrics

run()
This method extracts perf metrics for a given pid

set_up_perf_events()
This method checks for available perf events, tests them and initialise the data dict

setup_perf_monitor()
Setup steps for monitoring perf metrics

2.4.6 Post monitoring

2.4.7 Utility Functions

This module contains utility functions for gathering CPU metrics

class monitormetrics.utils.utils.**FileLock**(*protected_file_path*, *timeout=None*, *delay=1*, *lock_file_contents=None*)

A file locking mechanism that has context-manager support so you can use it in a `with` statement. This should be relatively cross compatible as it doesn't rely on `msvcrt` or `fcntl` for the locking.

exception FileLockException

Exception to the file lock object

acquire(*blocking=True*)

Acquire the lock, if possible. If the lock is in use, and *blocking* is False, return False. Otherwise, check again every *self.delay* seconds until it either gets the lock or exceeds *timeout* number of seconds, in which case it raises an exception.

available()

Returns True iff the file is currently available to be locked.

lock_exists()

Returns True iff the external lockfile exists.

locked()

Returns True iff the file is owned by THIS FileLock instance. (Even if this returns false, the file could be owned by another FileLock instance, possibly in a different thread or process).

purge()

For debug purposes only. Removes the lock file from the hard disk.

release()

Get rid of the lock by deleting the lockfile. When working in a *with* statement, this gets automatically called at the end.

class monitormetrics.utils.utils.**PDF**(*config*)

custom PDF class that inherits from the FPDF

footer()

This method defines footer of the pdf

header()

This method defines header of the pdf

page_body(*images*)

This method defines body of the pdf

print_page(*images*)

This method add an empty pages and populates with images/text

monitormetrics.utils.utils.**check_perf_events**(*perf_events*)

This function check if all perf groups are actually working. We will only probe the working counters during monitoring

Parameters **perf_events** (*dict*) – A dict of found perf events

Returns A dict of working perf events

Return type dict

monitormetrics.utils.utils.**dump_json**(*content*, *filename*)

This function appends data to an existing json content. It creates a new file if no existing file found.

Parameters

- **content** (*dict*) – Dict to write into JSON format
- **filename** (*str*) – Name of the file to load

monitormetrics.utils.utils.**execute_cmd**(*cmd_str*, *handle_exception=True*)
Accept command string and returns output.

Parameters

- **cmd_str** (*str*) – Command string to be executed
- **handle_exception** (*bool*) – Handle exception manually. If set to false, raises an exception to the caller function

Returns Output of the command. If command execution fails, returns 'not_available'

Return type str

Raises **subprocess.CalledProcessError** – An error occurred in execution of command iff handle_exception is set to False

monitormetrics.utils.utils.**execute_cmd_pipe**(*cmd_str*)
Accept command string and execute it using piping and returns process object.

Parameters **cmd_str** (*str*) – Command string to be executed

Returns Process object

Return type object

monitormetrics.utils.utils.**find_procs_by_name**(*name*)
Return a list of processes matching 'name'

Parameters **name** (*str*) – name of the process to find

Returns List of psutil objects

Return type list

monitormetrics.utils.utils.**get_cpu_model_family**(*cpu*)
" This function gives CPU model and family ids from CPUID instruction

Parameters **cpu** (*object*) – CPUID object

Returns Family and model IDs

Return type list

monitormetrics.utils.utils.**get_cpu_model_names_for_non_x86**()
This function tries to extract the vendor, model and cpu architectures for non x86 machines like IBM POWER, ARM

Returns Name of the vendor model name/number of the processor micro architecture of the processor

Return type str

monitormetrics.utils.utils.**get_cpu_vendor**(*cpu*)
This function gets the vendor name from CPUID instruction

Parameters **cpu** (*object*) – CPUID object

Returns Name of the vendor

Return type str

monitormetrics.utils.utils.**get_cpu_vendor_model_family**()
This function gets the name of CPU vendor, family and model parsed from CPUID instruction

Returns Name of the vendor, CPU family and model ID

Return type list

monitormetrics.utils.utils.**get_mem_bw_event**()
This function returns the perf event to get memory bandwidth

Returns A string to get memory bandwidth for perf stat command

Return type str

monitormetrics.utils.utils.**get_parser**(*cmd_output*, *reg='lscpu'*)
Regex parser.

Parameters

- **cmd_output** (*str*) – Output of the executed command
- **reg** (*str*) – Regex pattern to be used

Returns Function handle to parse the output

monitormetrics.utils.utils.**get_perf_events**()
This function checks the micro architecture type and returns available perf events. Raises an exception if micro architecture is not implemented

Returns Perf events with event name dict: Derived perf metrics from event counters

Return type dict

Raises **PerfEventsNotFoundError** – An error occurred while looking for perf events

monitormetrics.utils.utils.**get_rapl_devices**()
This function gets all the packages, core, uncore and dram device available within RAPL powercap interface

Returns A dict with package names and paths

Return type dict

monitormetrics.utils.utils.**get_value**(*input_dict*, *target*)
Find the value for a given target in dict

Parameters

- **input_dict** (*dict*) – Dict to search for key
- **target** (*Any*) – Key to search

Returns List of values found in d

Return type list

monitormetrics.utils.utils.**ibstat_ports**()
This function returns Infiniband ports if present

Returns A dict with IB port names and numbers

Return type dict

monitormetrics.utils.utils.**load_json**(*filename*)
This function loads json file and return dict

Parameters **filename** (*str*) – Name of the file to load

Returns File contents as dict

Return type dict

monitormetrics.utils.utils.**merge_dicts**(*exst_dict*, *new_dict*)
Merge two dicts. old_content is updated with data from new_content

Parameters

- **exst_dict** (*dict*) – Existing data in the dict
- **new_dict** (*dict*) – New data to be added to the dict

Returns updated exst_dict with contents from new_dict

Return type dict

monitormetrics.utils.utils.**proc_if_running**(*procs*)
Check if all processes are running and returns a False if all of them are terminated

Parameters **procs** (*list*) – List of psutil process objects

Returns Running status of the processes

Return type bool

monitormetrics.utils.utils.**replace_negative**(*input_list*)
This function replaces the negative values in numpy array with mean of neighbours. If the values happen to be at the extremum, it replaces with preceding or succeding elements

Parameters **input_list** (*list*) – A list with positive and/or negative elements

Returns A list with just positive elements

Return type list

monitormetrics.utils.utils.**write_json**(*content*, *filename*)
This function writes json content to a file

Parameters

- **content** (*dict*) – Dict to write into JSON format
- **filename** (*str*) – Name of the file to load

2.4.8 Processor specific data

This file contains processor specific information like model names, families and perf events

monitormetrics.utils.processorspecific.**cpu_micro_architecture_name**(*vendor_name*, *model_id*, *family_id*)
This function gives the name of the micro architecture based on CPU model and family IDs

Parameters

- **vendor_name** (*str*) – Name of the vendor
- **model_id** – Model ID of the CPU
- **family_id** – Family ID of the cpu

Returns Name of the micro architecture

Return type str

Raises

- ***ProcessorVendorNotFoundError*** – An error occurred while looking for processor vendor.
- ***KeyNotFoundError*** – An error occurred while looking for micro architecture

monitormetrics.utils.processorspecific.**llc_cache_miss_perf_event**(*processor_vendor*, *micro_architecture*)

This function gives the event code and umask for LLC cache miss event for different architectures

Parameters

- **processor_vendor** (*str*) – Vendor of the processor
- **micro_architecture** (*str*) – Name of the micro architecture of the processor

Returns String containing event code and umask

Return type str

Raises ***ProcessorVendorNotFoundError*** – An error occurred while looking for processor vendor.

monitormetrics.utils.processorspecific.**perf_event_list**(*micro_architecture*)

This function returns list of perf events implemented for a given processor and micro architecture

Parameters **micro_architecture** (*str*) – Name of the micro architecture

Returns A dict with name and event code of perf events dict: A dict with derived perf metrics and its formula

Return type dict

2.4.9 Exceptions

This file contains the custom exceptions defined for monitoring tools.

exception monitormetrics.utils.exceptions.**BatchSchedulerNotFound**
Batch scheduler not implemented or not recognised

exception monitormetrics.utils.exceptions.**CommandExecutionFailed**
Command execution exception

exception monitormetrics.utils.exceptions.**JobPIDNotFoundError**
Step job PID not found

exception monitormetrics.utils.exceptions.**KeyNotFoundError**
Key not found in the dict

exception monitormetrics.utils.exceptions.**PerfEventListNotFoundError**
Perf event list not implemented

exception monitormetrics.utils.exceptions.**ProcessorVendorNotFoundError**
Processor vendor not implemented

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

s

INDEX

E

F

G

H

I

J

K

L

M

P

R

S

W