developer.skatelescope.org Documentation

Marco Bartolini

Dec 28, 2021

HOME

This documentation contains the overview of SDP workflow performance monitoring tool, instructions to install and usage examples.

CHAPTER

ONE

OVERVIEW

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, export data in various formats and generate a job report with plots from different metrics.

1.1 Design

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.

Besides several CPU related metrics, the toolkit reports several performance metrics for NVIDIA GPUs. Python bindings of NVIDIA Management Library (NVML) is used to monitor the metrics.

1.2 Available metrics

Currently, the toolkit reports following metrics:

- Hardware 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), *etc*.
- `perf` events like hardware and software events, hardware cache events and different types of FLOP counts.
- NVIDIA GPU performance metrics.

All these metrics are gathered and exported in different formats (including JSON, CSV, H5 tables).

CHAPTER TWO

PREREQUISITES

The following prerequisites must be installed to use monitoring toolkit:

- python >= 3.7
- git

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/sdp/ska-sdp-perfmon.git
cd ska-sdp-perfmon
```

To install all the required dependencies

```
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/sdp/ska-sdp-perfmon.
↪git@main#egg=ska-sdp-perfmon"
```

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

CHAPTER

THREE

USAGE

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

```
usage: perfmon [-h] [-d [SAVE_DIR]] [-p [PREFIX]] [-l [LAUNCHER]] [-i SAMPLING_FREQ]␣
↪[-c [CHECK_POINT]] [-r] [-e {csv,hdf5,parquet,pickle,feather,orc} [{csv,hdf5,
↪parquet,pickle,feather,orc} ...]] [--system [SYSTEM]] [--partition [PARTITION]]
             [--env [ENV]] [--name [NAME]] [-v]

optional arguments:
-h, --help            show this help message and exit
-d [SAVE_DIR], --save_dir [SAVE_DIR]
                      Base directory where metrics will be saved. This directory␣
↪should
                      be available from all compute nodes. Default is $PWD
-p [PREFIX], --prefix [PREFIX]
                      Name of the directory to be created to save metric data. If␣
↪provided,
                      metrics will be located at $SAVE_DIR/$PREFIX.
-l [LAUNCHER], --launcher [LAUNCHER]
                      Launcher used to launch mpi tasks
-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
-r, --gen_report      Generate plots and job report
-e {csv,hdf5,parquet,pickle,feather,orc} [{csv,hdf5,parquet,pickle,feather,orc} ...],␣
↪--export {csv,hdf5,parquet,pickle,feather,orc} [{csv,hdf5,parquet,pickle,feather,
↪orc} ...]
                      Export results to different file formats
--system [SYSTEM]     Name of the system (only when used with SDP Benchmark tests)
--partition [PARTITION]
                      Name of the partition (only when used with SDP Benchmark tests)
--env [ENV]           Name of the environment (only when used with SDP Benchmark␣
↪tests)
--name [NAME]         Name of the test (only when used with SDP Benchmark tests)
-v, --verbose         Enable verbose mode. Display debug messages
```

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 reservation. Typically, NFS mounted home directories can be used for this directory. The option `prefix` can be used to create a sub directory within `$SAVE_DIR` where actual metrics will be saved. For instance, job ID can be used as prefix so that multiple job metrics will be located under same `$SAVE_DIR`. If `--prefix` is used, metrics can be found at `$SAVE_DIR/$PREFIX`. If no `prefix` option is passed, toolkit will place all metrics under `$SAVE_DIR` directory.
- The option `launcher` can be used to specify which `mpi` wrapper is used to launch parallel jobs. if not specified, toolkit will try to identify the launcher.
- 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 hardware metadata, CPU related and `perf stat` metrics.
- 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` or `--gen_report` option must be passed on the CLI.
- We can also export metric data in different formats using `--export` flag. Currently, toolkit is capable of exporting data in CSV, pickle, HDF5, Parquet, Feather and ORC formats.

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`.

3.2 GPU metrics

The toolkit automatically detects for the presence of NVIDIA GPUs and if found, it extracts the metrics irrespective of batch job is using GPUs or not. In order to detect the GPUs, `nvidia-smi` command is executed and return status is checked.

Note: At the moment, only NVIDIA GPUs are supported. The toolkit will not be able to monitor metrics for other types of GPUs.

3.3 Monitored metrics

3.3.1 Hardware metadata

Currently, 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. In the case of NVIDIA GPUs, several GPU related infos available from NVML library is used.

3.3.2 CPU related metrics

The metrics reported in this part are both process specific and system wide. The process pid that is captured from the batch scheduler's 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 (uJ).
- 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.

3.3.3 Perf stat metrics

Perf stat metrics are monitored by executing

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

Currently only Broadwell, Haswell, SkyLake, SandyBridge, Zen and Zen3 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.

3.3.4 NVIDIA GPU metrics

As stated before, NVML library is used to query for several device metrics for NVIDIA GPU cards. The reported metrics are as follows:

- Clock frequency info for Graphics, SM and memory
- Error Correcting Code (ECC) counts for Single and double precision
- Usage of GPU and BAR1 memory
- Device temperature, fan speed (if exists), number of processes
- PCI expresses send and receive throughputs
- Power usage and GPU throttling time due to power and thermal constraints
- GPU and memory utilization rates

All these metrics are reported for each GPU separately. A prefix of form `gpu-{num}` is added to host name of each node, where `{num}` is the GPU device number to differentiate different GPUs.

3.4 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

# 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 perfmon &
# mpirun --map-by node -np $SLURM_JOB_NUM_NODES perfmon &
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 `perfmon` script using `srun` as a background process. `srun` launches the `perfmon` 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 file based Inter Process Communicator (IPC). 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
```

(continues on next page)

(continued from previous page)

```
#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 $SLURM_JOB_NUM_NODES --ntasks-per-node 1 perfmon &
export MON_PID=$!
# mpirun --map-by node -np $SLURM_JOB_NUM_NODES perfmon &
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"

#! 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)
```

(continues on next page)

(continued from previous page)

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

sleep 5

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

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-perfmon/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
#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
```

(continues on next page)

(continued from previous page)

```
# 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 perfmon -i 5 -v -r -e &
sleep 2
mpirun --map-by node -np 2 ./matmul 1500

wait
```

3.5 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 if `--gen_report` flag is enabled. If a `--prefix` is used to name the directory all the files will be placed under this directory. If not the toolkit places all the files in `$SAVE_DIR`. Typically, under this folder we will find following sub-directories and files

- `configs/`: Contains configuration files of the toolkit used for each node. It will be dumped only when `-v` flag is enabled.
- `metrics/`: This folder contains all the metrics in JSON files.
- `plots/`: All the generated plots in png format are placed in this folder
- `job-report-{job-id}.pdf`: Job report with all the plots included
- `*.csv, *.h5, *.orc, *parquet, *.feather, *.pkl`: Exported files in different formats with metric as name.

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": {
```

(continues on next page)

(continued from previous page)

```
        "type": "array",
        "items": {
          "type": "number"
        }
      },
      "cpu_time": {
        "type": "array",
        "items": {
          "type": "number"
        }
      },
      "ib_io_counters": {
        "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"
            }
          },
```

(continues on next page)

(continued from previous page)

```
          "write_bytes": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "write_count": {
            "type": "array",
            "items": {
              "type": "number"
            }
          }
        }
      },
      "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": {
```

(continues on next page)

(continued from previous page)

```
        "type": "array",
        "items": {
          "type": "number"
        }
      },
      "net_io_counters": {
        "type": "object",
        "required": [],
        "properties": {
          "bytes_recv": {
            "type": "array",
            "items": {
              "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": {
```

(continues on next page)

(continued from previous page)

```
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "uncore-0": {
            "type": "array",
            "items": {
              "type": "number"
            }
          },
          "dram-0": {
            "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` and GPU metrics data are 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.

3.6 Plotting data

Recommended way to plot metric data is to export data into csv or any other available format and load them into Pandas dataframe. Examples can be found in `perfmon/common/plots` folder on how to plot data using Pandas dataframe.

CHAPTER

FOUR

API DOCUMENTATION

The following sections provide the API documentation of the source code of SDP workflow performance monitoring toolkit.

4.1 Perfmon Reference

4.1.1 Configuration

This file contains config related functions and classes

class `perfmon.cfg.__init__.`**`GlobalConfiguration`**(*args*)

Global configuration with defaults

`check_gpus`()

This method checks for presence of NVIDIA GPUs

`create_config`()

Entry point of teh class

`make_dirs`()

This method creates the directory that put all artefacts of toolkit

`populate_config`()

This method adds necessary common info to config dict

4.1.2 Common

“This package contains modules related to creating dataframe

class `perfmon.common.df.__init__.`**`CreateDataFrame`**(*metric*, *config*)

This class contains all methods to create a dataframe from JSON data

`check_non_default_metrics`(*content*)

Check for non default metrics

`create_dataframe`(*content*)

This method creates and returns dataframe from the metric data

`go`()

Entry point to the class

`initialise_header_names`()

This method initialises the names of headers for each metric

“This package contains classes to export metric data

class perfmon.common.export.__init__.**ExportData**(*config*, *df_dict*)
This class contains all methods to export dataframe into different data store types

get_lock_file()
Get the lock file to update database exports

go()
Entry point to the class

release_lock()
Releases lock file

This file contains initialisation functions for logging

class perfmon.common.logging.__init__.**HostnameFilter**(*name=''*)

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

perfmon.common.logging.__init__.**logger_config**(*global_config*)
shortcut method for initializing logging

Parameters **global_config** (*dict*) – Dict containing all the configuration info

Returns Logger initiated based on config passed

Return type logger object

This module contains functions that are related to perf stat metrics

perfmon.common.perf.__init__.**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

perfmon.common.perf.__init__.**derived_perf_event_list**(*perf_events*)
This function returns list of perf events implemented for a given processor and micro architecture

Parameters **perf_events** (*dict*) – Dictionary of perf events

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

Return type dict

perfmon.common.perf.__init__.**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

perfmon.common.perf.__init__.**get_working_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

`perfmon.common.perf.__init__.`**`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.

`perfmon.common.perf.__init__.`**`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

Return type dict

Raises ***`PerfEventListNotFoundError`*** – If perf events yml file is not found

This module contains class for detecting process PIDs for various schedulers

`class` `perfmon.common.pid.__init__.`**`GetJobPid`**(*config*)

Class to get the main job PID for different workload managers. Currently SLURM, PBS and OAR schedulers are supported

`go`()

This is driver method to find job PID

“This package contains functions to plot gathered metrics

`class` `perfmon.common.plots.__init__.`**`GenPlots`**(*config*, *df_dict*)

This class contains all plotting methods (Only for CPU metrics)

`apply_plot_settings`(*plot_type*, *metric_att*, *mean_max*, *ax*)

This method applies the common settings to the plots

`check_non_default_metrics`(*df*)

Check if IB, mem. bandwidth and RAPL metrics are available in collected metrics

`combined_plotting_engine`(*metric*, *metric_att*, *comb_ts_df*, *comb_metric_df*)

Plotting engine for combined metrics

`static convert_ts_datetime`(*df*)

Convert timestamps in df to datetime format

`static get_global_mean_max`(*mean_max_all*)

Get global mean max of metric from host data

`go`()

Entry point for plotting

`make_plots`(*df*)

This method plots both per host and combined metrics

plot_metric_data(*df*)
Make plots for the cpu metric data

plotting_engine(*host_name*, *metric*, *metric_att*, *ax*, *data*)
Main engine to create plots

static replace_neg_values(*df*)
Replace negative values in df to preceding positive values

This module contains functions related to processor specific info

`perfmon.common.processor.__init__.`**get_cpu_spec**()
This function extracts the vendor and cpu architectures using `archspec` module

Returns Name of the vendor str: Micro architecture

Return type str

This module contains class to generate job report

class `perfmon.common.report.__init__.`**GenReport**(*config*)
This class does all the post monitoring steps like making plots and generating reports

create_job_report(*content*)
Create a job report using FPDF module

go()
Entry point for creating report

initialise_plot_per_page()
Initialises plot related parameters

Utility functions related to devices on the platform

`perfmon.common.utils.devices.`**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

`perfmon.common.utils.devices.`**ibstat_ports**()
This function returns Infiniband ports if present

Returns A dict with IB port names and numbers

Return type dict

Utility functions for command execution

`perfmon.common.utils.execute_cmd.`**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 `perfmon.common.utils.execute_cmd.`**`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

Utility functions for manipulating json files

`perfmon.common.utils.json_wrappers.`**`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

`perfmon.common.utils.json_wrappers.`**`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

`perfmon.common.utils.json_wrappers.`**`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

Class to lock files

`class` `perfmon.common.utils.locks.`**`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.

Utility functions for parsing

class perfmon.common.utils.parsing.**RawFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Class SmartFormatter prints help messages without any formatting or unwanted line breaks, acivated when help starts with R|

perfmon.common.utils.parsing.**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

Class to create pdf file

class perfmon.common.utils.pdf.**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

Utility functions for psutil process finder

perfmon.common.utils.process.**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

perfmon.common.utils.process.**get_proc_info**(*pid*)
Convenient wrapper around psutil.Process to catch exceptions

perfmon.common.utils.process.**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

Utility functions

perfmon.common.utils.utilities.**dump_yaml**(*config*)
Dump config files (for debugging)

perfmon.common.utils.utilities.**get_project_root**()

Get root directory of the project

Returns Full path of the root directory

Return type str

perfmon.common.utils.utilities.**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

perfmon.common.utils.utilities.**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

perfmon.common.utils.utilities.**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

4.1.3 Core

This file contains class to launch monitoring process

class perfmon.core.metrics.__init__.**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

This file common functions that are needed to monitor metrics

perfmon.core.metrics.common.**check_metric_data**(*data_struct*)

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

perfmon.core.metrics.common.**dump_metrics_async**(*data*, *outfile*)

Dump metrics asynchronously

Parameters

- **data** (*dict*) – Data to be dumped to disk
- **outfile** (*str*) – Path of the outfile

`perfmon.core.metrics.common.`**`get_child_procs`**(*user*, *procs*)

Get list of children processes in user namespace

Parameters

- **user** (*str*) – User name
- **procs** (*object*) – psutil proc iterator

Returns List of children processes in user space

Return type list

`perfmon.core.metrics.common.`**`get_cumulative_metric_value`**(*metric_type*, *procs*, *data*)

This method gets cumulative metric account for all childs for a given metric type

This file contains base class to monitor CPU metrics

class `perfmon.core.metrics.cpu.`**`MonitorCpuUsage`**(*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

`dump_metrics`()

Dump metrics to JSON file and re-initiate cpu_metrics dict

`get_cpu_usage`()

This method gets all CPU usage statistics

`get_energy_metrics`()

This method gets energy metrics from RAPL powercap interface

`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)

`initialise_cpu_metrics_params`()

This method initialises the CPU metric related parameters

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

This file contains base class to monitor GPU metrics

class perfmon.core.metrics.gpu.**MonitorNvidiaGpuMetrics**(*config*)
    Engine to monitor gpu related metrics

add_timestamp()
    This method adds timestamp to the data

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

get_clock_info()
    This method gets different clock info metrics

get_ecc_metrics()
    This method gets ECC error counts

get_memory_usage()
    This method gets memory usage

get_metrics_data()
    Extract metrics data

get_misc_metrics()
    This method gets different misc metrics

get_new_host_name(*gpu_dev_num*)
    Append GPU number to host name

get_power_metrics()
    This method gets power metrics

get_utilization_rates()
    This method gets all utilization statistics

initialise_gpu_metrics_params()
    This method initialises the GPU metric related parameters

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

This file contains base class to monitor perf stat metrics

class perfmon.core.metrics.perfcounters.**MonitorPerfCounters**(*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

Functions to monitor RAPL energy metrics

perfmon.core.metrics.cpumetrics.energy.**rapl_energy_readings**(*rapl_devices*, *data*)
This method gets energy metrics from RAPL powercap interface

Functions to monitor memory related metrics

perfmon.core.metrics.cpumetrics.memory.**get_memory_bandwidth**(*mem_bw_event*, *procs*)
This method returns memory bandwidth based on perf LLC load misses event

perfmon.core.metrics.cpumetrics.memory.**memory_usage**(*mem_bw_event*, *procs*, *data*)
This method gets memory usage

Functions to monitor other metrics

perfmon.core.metrics.cpumetrics.misc.**misc_metrics**(*procs*, *data*)
This method gets IO, file descriptors and thread count

Functions to monitor network related metrics

perfmon.core.metrics.cpumetrics.network.**ib_io_counters**(*ib_ports*, *data*)
This method gets the IB port counters

perfmon.core.metrics.cpumetrics.network.**network_io_counters**(*data*)
This method gets the system wide network IO counters

Functions to monitor CPU usage metrics

perfmon.core.metrics.cpumetrics.usage.**get_cpu_percent**(*cpu_aggregation_interval*, *procs*)
This method gives CPU percent of parent and its childs

perfmon.core.metrics.cpumetrics.usage.**get_cpu_time**(*procs*)
This method gets cumulative CPU time from parent and its childs

This module contains all NVIDIA GPU related metrics functions

perfmon.core.metrics.gpumetrics.nvidia.__init__.**device_query**(*func*, **args*)
Convenience wrapper to query different metrics for NVIDIA GPUs

Parameters **func** (str) – Name of the API function

Returns Metric value

Return type list

Functions to monitor clock frequency info related metrics for NVIDIA GPUs

`perfmon.core.metrics.gpumetrics.nvidia.clock.`**`clock_info`**`(`*data*`)`
This method gets NVIDIA GPU clock info for memory, graphics and SM

Functions to monitor ECC error counts for NVIDIA GPUs

`perfmon.core.metrics.gpumetrics.nvidia.errors.`**`ecc_error_counts`**`(`*data*`)`
This method gets NVIDIA GPU ECC error counts for SP and DP

Functions to monitor memory related metrics for NVIDIA GPUs

`perfmon.core.metrics.gpumetrics.nvidia.memory.`**`memory_usage`**`(`*data*`)`
This method gets NVIDIA GPU memory and BAR1 memory usage

Functions to monitor misc metrics like temperature, fan speed for NVIDIA GPUs

`perfmon.core.metrics.gpumetrics.nvidia.misc.`**`misc_metrics`**`(`*data*`)`
This method gets misc NVIDIA GPU metrics

Functions to monitor power related metrics for NVIDIA GPUs

`perfmon.core.metrics.gpumetrics.nvidia.power.`**`power_usage`**`(`*data*`)`
This method gets NVIDIA GPUs power usage metrics

`perfmon.core.metrics.gpumetrics.nvidia.power.`**`power_violation_report`**`(`*data*`)`
This method gets NVIDIA GPUs throttling period due to constraints

Functions to get GPU utilization rates

`perfmon.core.metrics.gpumetrics.nvidia.utilization.`**`get_encoder_decoder_util_rates`**`(`*data*`)`
This method gets encoder and decoder utilization rates

`perfmon.core.metrics.gpumetrics.nvidia.utilization.`**`get_gpu_mem_util_rates`**`(`*data*`)`
This method gets GPU and memory utilization rates

4.1.4 Exceptions

This file contains the custom exceptions defined for monitoring tools.

exception `perfmon.exceptions.__init__.`**`ArchitectureNotFoundError`**
Processor architecture not found

exception `perfmon.exceptions.__init__.`**`BatchSchedulerNotFound`**
Batch scheduler not implemented or not recognised

exception `perfmon.exceptions.__init__.`**`CommandExecutionFailed`**
Command execution exception

exception `perfmon.exceptions.__init__.`**`JobPIDNotFoundError`**
Step job PID not found

exception `perfmon.exceptions.__init__.`**`KeyNotFoundError`**
Key not found in the dict

exception `perfmon.exceptions.__init__.`**`MetricGroupNotImplementedError`**
Requested metric group not implemented

exception `perfmon.exceptions.__init__.`**`PerfEventListNotFoundError`**
Perf event list not implemented

exception `perfmon.exceptions.__init__.`**`ProcessorVendorNotFoundError`**

Processor vendor not implemented

4.1.5 Perfevents

“This package contains perf events lists for different architectures

4.1.6 Schemas

“This package contains schemas for perfmon toolkit

This is schema for dataframe

This is schema for metrics data

This is schema for plots

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX