sample Documentation

Release v0.1.0

Tristan Carel

Feb 17, 2020

Contents

HPCBench is a Python package that allows you to specify and execute benchmarks. It provides:

- an API to describe how to execute benchmarks utilities and gather metrics.
- A way to describe benchmark campaigns in YAML format.
- command line utilities to execute your campaigns, and post-process generated metrics in various ways:
  - Merging metrics from different campaigns
  - Exporting metrics in CSV format
  - Exporting data to Elasticsearch
  - PDF report generation

HPCBench assumes that the various benchmark tools/binaries are managed elsewhere and does not provide features for building and maintaining benchmark software.

NB: This Python package is still in pre-alpha stage, and not suitable for production.

CHAPTER 1

Installation

HPCBench is in the Python Package Index.

1.1 Installation with pip

We recommend using pip to install hpcbench on all platforms:

```
$ python -m pip install hpcbench
```

To upgrade HPCBench with pip:

```
$ python -m pip install --upgrade hpcbench
```

1.2 Dependencies

HPCBench supports both python 2.7 and 3.4+.

CHAPTER 2

Overview

2.1 CLI

The main interface through which **HPCbench** is used is a set of command line utilities:

- ben-sh: Execute a tests campaign on your workstation
- ben-csv: Extract metrics of an existing campaign in csv format
- ben-umb: Extract metrics of an existing campaign
- ben-elastic: Push campaign data to Elasticsearch
- ben-nett: Execute a tests campaign on a cluster
- ben-merge: Merge campaign output directories
- ben-tpl: Generate HPCBench plugin scaffolds, see *usage* for more information on plugin generation.

ben-sh and **ben-nett** expect a YAML file describing the campaign to execute. The structure of this YAML file is detailed in the *campaign file reference*.

2.2 Campaign YAML description

2.2.1 HPCBench Campaign file reference

HPCBench uses a YAML file (see YAML cookbook) to describe a benchmark campaign. Topics of this reference page are organized by top-level key to reflect the structure of the Campaign file itself.

output_dir

This top-level attribute specifies the output directory where HPCBench stores the benchmark results. The default value is “hpcbench-%Y%m%d-%H%M%S” You can also specify some variables enclosed in braces, specifically:

- node: value of ben-sh "-n" option.

This also includes environment variables (prefixed with $). For instance for a daily report with the node name inside the directory, you can use: "hpcbench-{node}-$USER-%Y%m%d"

Network configuration reference

A Campaign is made of a set of nodes that will be benchmarked. Those nodes can be tagged to create groups of nodes. The tags are used to constrain benchmarks to be run on subsets of nodes.

nodes

Specify which nodes are involved in the tests campaign. Here is an sample describing a cluster of 2 nodes.

```
network:
  nodes:
    - srv01
    - srv02
    - gpu-srv01
    - gpu-srv02
```

Nodes can also be specified using the ClusterShell *NodeSet* syntax. For instance

```
network:
  nodes:
    - srv[0-1,42,060-062]
```

is equivalent to:

```
network:
  nodes:
  - srv0
  - srv2
  - srv42
  - srv060
  - srv061
  - srv062
```

tags

Specify groups of nodes.

A tag can be defined with an explicit node list, a regular expression of node names, a recursive to other tags, or a SLURM constraint.

For instance, given the set of nodes defined above, we can define the *cpu* and *gpu* tags as follow:

```
network:
  nodes:
    - srv01
    - srv02
    - gpu-srv01
    - gpu-srv02
  tags:
```

(continues on next page)

(continued from previous page)

```yaml
    cpu:
      nodes:
        - srv1
        - srv2
    gpu:
      match: gpu-.*
    all-cpus:
      constraint: skylake
    all:
      tags: [cpu, gpu]
```

All methods are being used:

- **nodes** expects an exhaustive list of nodes. The ClusterShell *NodeSet* syntax is also supported.
- **match** expects a valid regular expression
- **tags** expects a list of tag names
- **constraint** expects a string. This tag does not references node names explicitly but instead delegates it to SLURM. The value of the constraint tag is given to the sbatch options through the *–constraint* option.

cluster

If value is "slurm", then the network `nodes` is filled based on the output of the `info` command. A tag will be also added for every (partition, feature) tuple formatted like this: `{partition}_{feature}`.

slurm_blacklist_states

List of SLURM node states used to filter-out nodes when `cluster` option is set to `slurm`. Default states are down, drained, draining, error, fail, failing, future, maint, and reserved.

ssh_config_file

Optional path to a custom SSH configuration file (see man ssh_config(5)). This can be used to provide HPCBench access to cluster nodes without passphrase by using a dedicated SSH key.

For instance:

```
Host *.my-cluster.com
User hpc
IdentityFile ~/.ssh/hpcbench_rsa
```

remote_work_dir

Working path on remote nodes. Default value is `.hpcbench` Relative paths are relative from home directory.

installer_template

Jinja template to use to generate the shell-script installer deployed on cluster's nodes. Default value is `ssh-installer.sh.jinja`

installer_prelude_file

Optional path to a text file that will be included at the beginning of the generated shell-script installer. This can be useful to prepare the working environment, for instance to make Python 2.7, or Python 3.3+ available in `PATH` environment variable if this is not the case by default.

max_concurrent_runs

Number of concurrent benchmarks executed in parallel in the cluster. Default is 4.

pip_installer_url

HPCBench version to install on nodes. By default it is the current `ben-nett` version managing the cluster. This is an argument given to `pip` installer, here are a some examples:

- `hpcbench==2.0` to force a version available PyPi
- `git+http://github.com/BlueBrain/hpcbench@master#egg=hpcbench` to install the bleeding edge version.
- `git+http://github.com/me/hpcbench@feat/awesome-feature#egg=hpcbench` to deploy a fork's branch.

Benchmarks configuration reference

The **benchmarks** section specifies benchmarks to execute on every tag.

- key: the tag name or "*". "*" matches all nodes described in the *network.nodes* section.
- value: a dictionary of name -> benchmark description.

```
benchmarks:
  cpu:
    test_cpu:
      type: sysbench
  '*':
    check_ram
      type: random_ram_rw
```

Tag specific sbatch parameters

When running in *SLURM mode* a special *sbatch* dictionary can be used. This dictionary will be used when generating the sbatch file specific to this tag, allowing parameters to be overwritten.

```
process:
  type: slurm
  sbatch:
    time: 01:00:00
    tasks-per-node: 1

benchmarks:
  cpu:
    sbatch:
```

(continues on next page)

(continued from previous page)

```yaml
      hint: compute_bound
      tasks-per-node: 16
    test_cpu:
      type: sysbench
```

Benchmark configuration reference

Specify a benchmark to execute.

type

Benchmark name.

```yaml
benchmarks:
  cpu:
    test_cpu:
      type: sysbench
```

attributes (optional)

*kwargs** arguments given to the benchmark Python class constructor to override default behavior, which is defined in the benchmark class.

```yaml
benchmarks:
  gpu:
    test_gpu:
      type: sysbench
      attributes:
        features:
        - gpu
```

exec_prefix (optional)

Command prepended to every commands spawned by the tagged benchmark. Can be either a string or a list of string, for instance:

```yaml
benchmarks:
  cpu:
    mcdram:
      exec_prefix: numactl -m 1
      type: stream
```

srun (optional)

When hpcbench is run in *srun* or *slurm* benchmark execution mode, this key roots a list of options, which are passed to the *srun* command. Note that only the long form option names should be used (i.e. *–nodes* instead of *-N*). These options overwrite the global options provided in the *process* section. To disable a global srun option simply declare the option without providing a value. if an option without value (e.g. *–exclusive*) is to be used in *srun*, the key should be assigned to *true*.

```yaml
benchmarks:
  cpu:
    osu:
      srun:
        nodes: 8
        ntasks-per-node: 36
        hint:
        exclusive: true
      type: osu
```

spack (optional)

Dictionary to specify spack related configuration. Supported attributes are:

- **specs**: list of spack specs to install before executing benchmarks. *bin* directory of install directories are be prepended to *PATH*.

For instance:

```yaml
benchmarks:
    '*':
        test01:
            type: stream
            spack:
                specs:
                - stream@intel+openmp
```

attempts (optional)

Dictionary to specify the number of times a command must be executed before retrieving its results. Those settings allow benchmark execution on warm caches. Number of times can be either specified statically or dynamically.

The static way to specify the number of times a command is executed is through the `fixed` option.

```yaml
benchmarks:
    '*':
        test01:
            type: stream
            attempts:
                fixed: 2
```

All executions are present in the report but only metrics of the last run are reported. The `sorted` key allows to change this behavior to reorder the runs according to criteria.

```yaml
benchmarks:
    '*':
        test01:
            type: imb
            attempts:
                fixed: 5
                sorted:
                  sql: metrics__latency
                  reverse: true
```

sql can be a string or a list of string in kwargsql format. They are used to sort hpcbench.yaml reports. reverse is optional and allows to reverse the sort order. In this example, the report with the smallest latency is picked.

The dynamic way allows you to execute the same command over and over again until a certain metric converges. The convergence condition is either fixed with the epsilon parameter or relative with percent.

```
benchmarks:
    '*':
        test01:
            type: stream
            attempts:
                metric: bandwidth
                epsilon: 50
                maximum: 5
```

Every commands of the stream benchmark will be executed:

- as long as the difference of bandwidth metric between two consecutive runs is above 50.
- at most 5 times

```
benchmarks:
    '*':
        test01:
            type: stream
            attempts:
                metric: bandwidth
                percent: 10
                maximum: 5
```

Every commands of the stream benchmark will be executed:

- as long: `abs(bandwidth(n) - bandwidth(n - 1)) < bandwidth(n) * percent / 100`
- at most 5 times

environment (optional)

A dictionary to add environment variables. Any boolean values; true, false, yes not, need to be enclosed in quotes to ensure they are not converted to python True or False values by the YAML parse. If specified, this section supersedes environment variables emitted by benchmark.

```
benchmarks:
  '*':
    test_cpu:
      type: sysbench
      environment:
        TEST_ALL: 'true'
        LD_LIBRARY_PATH: /usr/local/lib64
```

modules (optional)

List of modules to load before executing the command. If specified, this section supersedes modules emitted by benchmark.

cwd (optional)

Specifies a custom working directory.

Precondition configuration reference

This section specifies conditions to filter benchmarks execution.

```
benchmarks:
  '*':
    cpu_numactl_0:
      exec_prefix: [numctl, -m, 0]
      type: stream
    cpu_numactl_1:
      exec_prefix: [numctl, -m, 1]
      type: stream
    disk:
      type: mdtest
precondition:
  cpu_numactl_0: HPCBENCH_MCDRAM
  cpu_numactl_1:
    - HPCBENCH_MCDRAM
    - HPCBENCH_CACHE
```

- **cpu_numactl_0** benchmark needs the `HPCBENCH_MCDRAM` environment variable to be defined for being executed.
- **cpu_numactl_1** benchmark needs either `HPCBENCH_MCDRAM` or `HPCBENCH_CACHE` environment variables to defined for being executed.
- **disk** benchmark will be executed in all cases.

Process configuration reference

This section specifies how `ben-sh` execute the benchmark commands.

type (optional)

A string indicating the execution layer. Possible values are:

- `local` (default) directs HPCbench to spawn child processes where `ben-sh` is running.
- `slurm` will use SLURM mode. This will cause HPCBench to generate for each tag in the network, which is used by at least one benchmark, one **sbatch** file. The batch file is then submitted to the scheduler. By default this batch file will invoke hpcbench on the allocated nodes and execute the benchmarks for this tag.
- `srun` will use srun to launch the benchmark processes. When HPCBench is being executed inside the self-generated batch script, it will use by default the `srun` mode to run the benchmarks.

commands (optional)

This dictionary allows setting alternative *srun* or *sbatch* commands or absolute paths to the binaries.

```
process:
  type: slurm
  commands:
    sbatch: /opt/slurm/bin/sbatch
    srun: /opt/slurm/bin/sbatch
```

srun and sbatch (optional)

The `srun` and `sbatch` dictionaries provide configurations foe the respective SLURM commands.

```
process:
  type: slurm
  sbatch:
    account: users
    partition: über-cluster
    mail-type: ALL
  srun:
    mpi: pmi2
```

executor_template (optional)

Override default Jinja template used to generate shell-scripts in charge of executing benchmarks. Default value is:

```
#!/bin/sh
{%- for var, value in environment.items() %}
export {{ var }}={{ value }}
{%- endfor %}
cd "{{ cwd }}"
exec {{ " ".join(command) }}
```

If value does not start with shebang, then it is considered like a file location.

Global metas dictionary (optional)

If present at top-level of YAML file, content of `metas` dictionary will be merged with those from every execution (see `hpcbench.api.Benchmark.execution_context`) Those defined in `execution_context` take precedence.

Environment variable expansion

Your configuration options can contain environment variables. HPCBench uses the variable values from the shell environment in which *ben-sh* is run. For example, suppose the shell contains EMAIL=root@cscs.ch and you supply this configuration:

```
process:
  type: slurm
  sbatch:
    email=$EMAIL
    partition=über-cluster
```

When you run ben-sh with this configuration, HPCBench will look for the EMAIL environment variable in the shell and substitutes its value in.

If an environment variable is not set, substitution fails and an exception is raised.

Both $VARIABLE and ${VARIABLE} syntax are supported. Additionally, it is possible to provide inline default values using typical shell syntax:

${VARIABLE:-default} will evaluate to default if VARIABLE is unset or empty in the environment. ${VARIABLE-default} will evaluate to default only if VARIABLE is unset in the environment. ${#VARIABLE} will evaluate to the length of the environment variable. Other extended shell-style features, such as ${VARIABLE/foo/bar}, are not supported.

You can use a $$ (double-dollar sign) when your configuration needs a literal dollar sign. This also prevents HPCBench from interpolating a value, so a $$ allows you to refer to environment variables that you don't want processed by HPCBench.

2.2.2 HPCBench Standard Benchmark

The "standard" benchmark is a general purpose benchmark. Its configuration is passed thru the `attributes` field in the YAML campaign file.

It is made of 3 top attributes, each of one being a dictionary:

- **metrics**: specifies how to extract information from command output
- **executables**: describe the commands to execute
- **shells** (optional): provide more flexibility to build the commands to execute

A trivial example

Before getting into all the details, here's a basic standard benchmark configuration to get started with:

```
benchmarks:
  '*':
    simple:
      type: standard
      attributes:
        executables:
            - command: ["echo", "42"]
        metrics:
            the_answer_to_everything:
                match: "(.*)"
                type: Cardinal
```

Metrics configuration reference

This section specifies the metrics the benchmark has to extract from command outputs, as dictionary "name" -> "configuration"

A metric configuration is a dictionary made of the following keys:

match

The regular expression used to extract the value from the program execution output. The expression must specify one and only one group used to extract the proper value.

The string to match has trailing whitespace removed.

type

The metric type, as specified in the *hpcbench.API.Metric*

multiply_by (optional)

If any, the extracted value will be multiplied by the specified value. It is useful to convert a unit, for instance from flop to Gflop.

category (optional)

The benchmark category the metric applies to. Default is `standard`.

from (optional)

Specifies the output file to look for. Default is `stdout`.

when (optional)

Provides a way to override fields above according to the metas of the executed command. Conditions may be declared as a list, the first condition evaluated to `true` providing a given property is used.

A condition is a dictionary composed of the following attributes:

- conditions: a dictionary of "meta_name" -> "value" where value is either a value of a list of values.
- match, multiply_by, from (optional): provide value that supersedes default one if conditions above are met.

For instance:

```
benchmarks:
  '*':
    my-test:
      type: standard
      attributes:
        metrics:
          simulation_time:
            match: "\\s+total compute time:\\s(\\d*\\.?\\d+) \\[ms\\]"
            type: Second
            multiply_by: 0.001
            when:
            -
              conditions:
                compiler: [gcc, icc]
                branch: feat/bump-performance
```

(continues on next page)

(continued from previous page)

```
                match: "\\s+total pool clone time:\\s(\\d*\\.?\\d+) \\[ms\\]"
                multiply_by: 1.0
```

This example describes how the `simulation_time` metric has to be extracted and computed. In the general case:

- the regular expression used to extract the metric is the "… total compute time" expression
- The type of the metrics is `Metric.Second`
- The extracted value will by multiplied by 0.001

But when the *compiler* metas is either "gcc" or "icc" **and** when the *branch* meta is "feat/bump_performance":

- the regular expression is different
- the multiplication factor is 1

Executables configuration reference

This section specifies the commands the benchmark has to execute. It is made of a list of dictionaries. Each dictionary describes a set of commands to run. They are composed of the following keys:

command

Describes the command to launch. It must be a list of elements. Elements support the Python Format Specification Mini-Language where the possible attributes are the metas describe below.

metas

A list of dictionary or the dictionary itself if this is the only one. Each dictionary describes a set of metas values.

```
executables:
-
  command: [echo, {foo}, {bar}]
  metas:
  -
    foo: 1
    bar: [2, 3]
  - foo: [4, 5]
    bar: [6, 7]
```

Using a list of values allows you to describe a combination of commands. In the example above, it means launching 6 commands:

- `echo 1 2`
- `echo 1 3`
- `echo 4 6`
- `echo 4 7`
- `echo 5 6`
- `echo 5 7`

It is possible to specify several metas at once:

```yaml
executables:
-
  command: [echo, {foo}, {bar}]
  metas:
  -
    foo: 1
    bar: [2, 3]
  - "foo, bar": [[4, 6], [5, 7]]
```

This sample is equivalent to the previous.

Some functions can also be called to specify the list of values a meta can take, among:

- `range`, same as Python range builtin
- `linspace`, `geomspace`, `linspace`, `arange`, same as NumPy corresponding functions.
- `correlate`, to specify multi metas at once.

In this case, the meta description is a dictionary providing the following attributes:

- `function`: name of the function to call
- `args`: optional list of arguments given to the function
- `kwargs`: optional dictionary of keywords arguments given to the function

For instance:

```yaml
executables:
-
  command: [echo, {foo}, {bar}]
  metas:
  -
    foo: 1
    bar:
      function: range
      args: [2, 4]
```

Will launch the 2 commands:

- `echo 1 2`
- `echo 1 3`

The `correlate` signature is as follow: * a mandatory list of series given in the `args` section * 2 optional arguments: `explore` and `with_overflow`

A serie is made of a list of arguments givento a NumPY function returning the values the meta has to take, for instance:

`[geomspace, 32, 1, num=6]`

allowed functions are: `geomspace`, `logspace`, `linspace`, `arange` an additional *_cast=<type>* allows you to cast the result of the NumPy function, for instance: `[geomspace, 32, 1, num=6, _cast=int]`

For example:

```yaml
executables:
-
  command: [mycommand, -p, {processes}, -t, {threads}]
  metas:
  -
    "[processes, threads]":
```

(continues on next page)

(continued from previous page)

```
      function: correlate
      args:
      - [geomspace, 8, 1, num=4, _cast=int]
      - [geomspace, 1, 8, num=4, _cast=int]
```

Will launch the following 4 commands:

- `mycommand -p 8 -t 1`
- `mycommand -p 4 -t 2`
- `mycommand -p 2 -t 4`
- `mycommand -p 1 -t 8`

The `explore` optional argument allows you to test additional combinations by modifying every combinations by given matrices

For example:

```
executables:
-
  command: [mycommand, -p, {processes}, -t, {threads}]
  metas:
  -
    "[processes, threads]":
      function: correlate
      args:
      - [geomspace, 4, 1, num=3, _cast=int]
      - [geomspace, 1, 4, num=3, _cast=int]
      kwargs:
        explore:
        - [0, 1]
```

Will launch the following 8 commands:

- `mycommand -p 4 -t 1`
- `mycommand -p 2 -t 2`
- `mycommand -p 1 -t 4`
- `mycommand -p 4 -t 2`
- `mycommand -p 2 -t 3`

If the optional boolean `with_overflow` keyword argument was set to True, then an additional `(1, 1)` command would have been triggered, corresponding to the initial (1, 4) combination plus (1, 1) matrix. Instead of having (1, 5), the first value of the `threads` serie would had been used, resulting in the `(1, 1)` value. Because such combination is usually pointless, the `with_overflow` default value is False.

category (optional)

A category to ease classification. Default value is "standard".

Shells configuration reference

This sections describes a list of commands that may prefix the commands specified in the `executables` section. It is composed of a list of dictionary. Each dictionary is made of the following keys:

commands

A list of shell commands, for instance

```
shells:
- commands:
  - . /usr/share/lmod/lmod/init/bash
  - . $SPACK_ROOT/share/spack/setup-env.sh
  - spack install myapp@{branch} %{compiler}
  - spack load myapp@{branch} %{compiler}
```

Specified commands also support the Python Format Specification Mini-Language to use the metas of execution context. Those metas can be either those define in the `shells` or `executables` section.

metas

Provides either a list of a dictionary providing additional metas values or the dictionary itself if this is the sole dictionary. Like in the `executables` section, it describes a combination of metas.

```
shells:
- commands:
  - . /usr/share/lmod/lmod/init/bash
  - . $SPACK_ROOT/share/spack/setup-env.sh
  - spack install myapp@{branch} %{compiler}
  - spack load myapp@{branch} %{compiler}
  metas:
    compiler: [gcc, icc]
```

2.3 API

The purpose of the HPCBench API is to provide a consistent and unified layer:

- to execute, and parse results of existing benchmarks utilities (Linpack, IOR, . . .)
- to use extracted metrics to build figures

Both system benchmarks (e.g. STREAM, linpack), as well as software benchmarks should be implemented using this API. Most users parametrize benchmarks in the above-mentioned campaign YAML file. More advanced users will want to implement their own benchmarks based on the API.

For more information check the module reference

CHAPTER 3

Getting Started

As of now, only a few benchmarks are supported. This section assumes that you have at least installed the `sysbench` utility on your workstation.

3.1 Launch a campaign on your workstation

3.1.1 Create your first campaign YAML file

Create a `local-campaign.yaml` file with the following content:

```
benchmarks:
  '*':
    test:
      type: sysbench
```

3.1.2 Launch the benchmark on your workstation

Execute the following command:

```
$ ben-sh local-campaign.yaml
```

This will create a `hpcbench-<date>` in the directory with the benchmark's results. Although the user is not meant to be manually checking results inside the output directory at this point take a look at `hpcbench-<date>/<hostname>/*/test/metrics.json`. You will find that this file contains the collected metrics data from sysbench. The raw logs and stdout's can be found further down the directory tree.

Note: Do not manually edit files inside the output directory. HPCBench offers a number of utilities to export and post-process the collected results.

3.2 Launch a campaign on a set of nodes

The YAML config file is getting a little more complex. For instance create the following `remote-campaign.yaml`:

```
network:
  nodes:
    - localhost
benchmarks:
  '*':
    test:
      type: sysbench
```

You can add computer nodes to the `nodes` section.

3.2.1 Launch the benchmark

Use the `ben-nett` utility to execute the campaign on every nodes. It uses SSH to submit jobs so you have to make sure you can access those nodes without passphrase. For this you could use the `ssh_config_file` key in YAML to specify a custom configuration (see) *campaign file reference*):

```
$ ben-nett remote-campaign.yaml
```

3.2.2 How to create a new benchmark Python module?

It is possible to create your own benchmark based on a tool that has not been so far supported by HPCBench. This is done by generating an external plugin scaffold using `ben-tpl` and implementing the benchmark execution and metrics parsing inside the generated classes.

Here is the basic workflow:

1. **First create a default JSON file describing the Python module:** `ben-tpl benchmark -g config.json`
2. Update fields in `config.json`
3. **Generate the Python module template in the current directory:** `ben-tpl benchmark config.json`
4. Edit the `benchmark.py` file
5. When ready you can install the module, with `pip` for instance.

CHAPTER 4

Development Guide

4.1 Prerequisites

4.1.1 Docker

Elasticsearch is required to execute some of the unit-tests. The easiest way to accomplish this, is to use Docker containers.

Quick'n'dirty Docker installation:

```
$ curl -fsSL get.docker.com -o get-docker.sh
$ sh get-docker.sh
$ curl -L https://github.com/docker/compose/releases/download/1.15.0/docker-compose-
↪`uname -s`-`uname -m` > /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
```

Post-installation instructions to use Docker without root privileges (logout/login) required:

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

Test your docker installation with:

```
$ docker run --rm hello-world
```

4.2 Build instructions

Grab the source code:

```
$ git clone https://github.com/BlueBrain/hpcbench.git
$ cd hpcbench
```

We suggest you use a dedicated virtual environment. For that you can use either virtualenv package or pyenv, which is even better.

With `pyenv`:

```
$ pyenv virtualenv hpcbench
$ pyenv local hpcbench
```

Alternatively, with `virtualenv`:

```
$ virtualenv .env
$ . .env/bin/activate
```

Then:

```
$ pip install tox
$ tox -e py27
```

`tox` is configured to test HPCBench against different Python versions. To test against a specific python version you can supply it the `-e` parameter:

```
$ tox -e py27 --
$ tox -e py36 --
```

The `--tests` parameter can be used to run only one specific unit test module or class:

```
$ tox  -e py36 -- --tests tests/test_driver.py
```

4.2.1 Elasticsearch

To start an Elasticsearch container, you can use the `misc/dc` script wrapper on top of `docker-compose`:

```
$ misc/dc up -d
```

It will start an Elasticsearch container listening on port 9200 and a Kibana instance listening on port 5612.

Let's now try to ping Elasticsearch:

```
$ curl localhost:9200
{
  "name" : "jQ-BcoF",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "yGP7_Q2gSU2HmHpnQB-jzg",
  "version" : {
    "number" : "5.5.1",
    "build_hash" : "19c13d0",
    "build_date" : "2017-07-18T20:44:24.823Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.0"
  },
  "tagline" : "You Know, for Search"
}
```

You can also access Kibana at http://localhost:5601 The `Dev Tools` is one of the most handy Elasticsearch client for humans.

4.2.2 Testing it all out

Unit-tests assume that Elasticsearch is running on localhost. You can define the `UT_ELASTICSEARCH_HOST` environment variable to specify another location:

```
$ ELASTICSEARCH_HOST=server01:9200 tox
```

4.2.3 How to integrate a new benchmark utility in the HPCBench repository?

1. First make sure you can properly build the project and tests pass successfully. It may be tempting to skip this part, but please don't.
2. Create a dedicated Git branch.
3. Create a new Python module in `hpcbench/benchmark` directory named after the utility to integrate.
4. In this new module, implement `hpcbench.api.Benchmark` and `hpcbench.MetricsExtractor` classes.
5. Register the new module in `setup.py [hpcbench.benchmarks]` entrypoint so that it can be found by HPCBench.
6. Create a dedicated unit test class in *tests/benchmark/* directory. The purpose of this test is to make sure that: * your Benchmark class is properly defined, and usable by HPCBench. * your metric extractor is properly working, without having to launch the

    utility itself.
7. To properly test your metrics extractors, some outputs of the benchmark utility will be added to the repository. For every category of your benchmark, create a file title `tests/benchmark/<test_module_name>.<category>.stdout` with the benchmark utility's output. These files will be automatically used. Do not hesitate to take inspiration from the `tests/benchmark/test_sysbench.py` test module.
8. Run the test-suites until it passes:

    ```
    $ tox
    ```
9. Submit a pull-request

CHAPTER 5

LICENSE

This software is released under MIT License.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index