MDBenchmark Documentation

Release 1.3.3

Max Linke & Michael Gecht

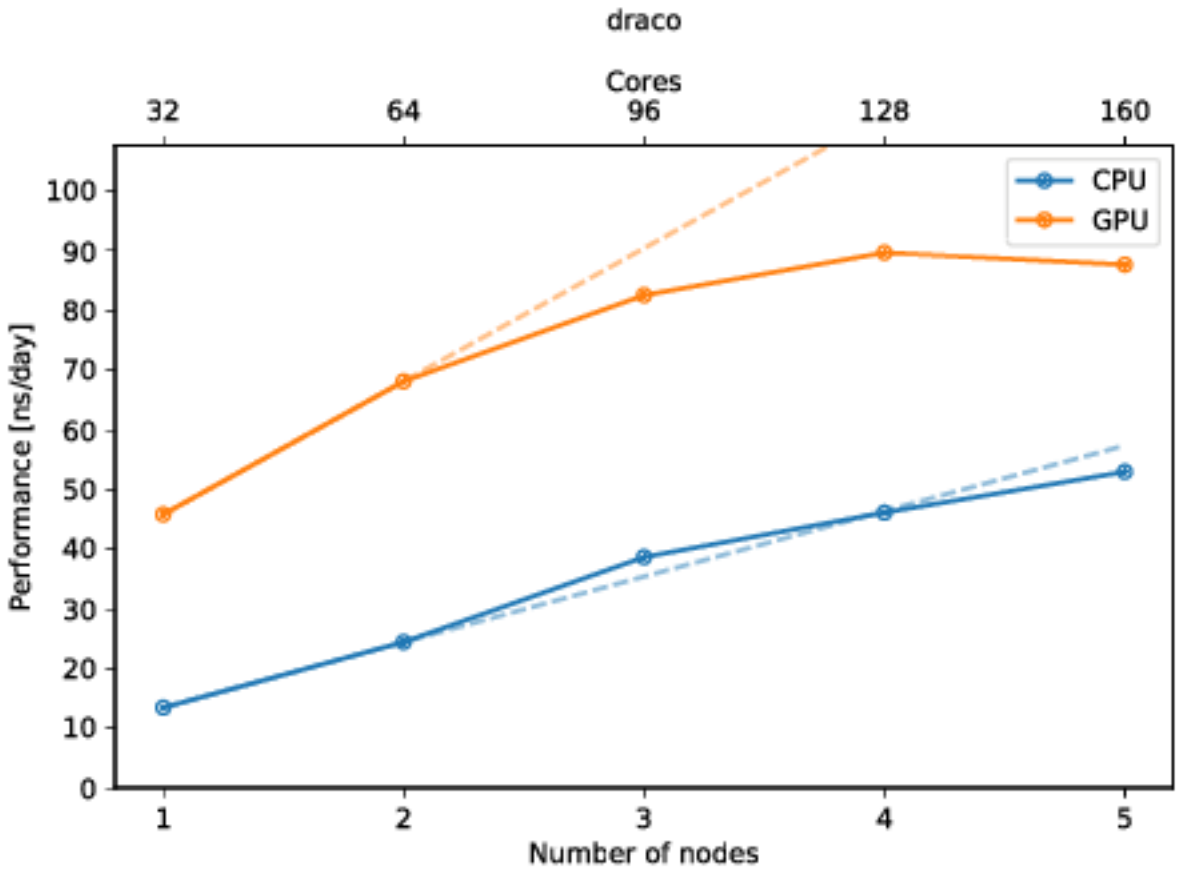
Oct 16, 2018

Contents

MDBenchmark — quickly generate, start and analyze benchmarks for your molecular dynamics simulations.

MDBenchmark is a tool to squeeze the maximum out of your limited computing resources. It tries to make it as easy as possible to set up systems on varying numbers of nodes and compare their performances to each other.

You can also create a plot to get a quick overview of the possible performance (and show of to your friends)! The plot below shows the performance of an molecular dynamics system on up to five nodes with and without GPUs.



CHAPTER 1

Quick start

Follow the next two paragraphs to get a quick start. Extended usage guides can be found below. You can install `mdbenchmark` with your favorite Python package manager. Afterwards you are ready to use `mdbenchmark`.

1.1 Install

If you are familiar with the usual way of installing python packages, just use `pip`:

```
pip install mdbenchmark
```

Anaconda users can install via `conda`:

```
conda install -c conda-forge mdbenchmark
```

Cutting-edge users may prefer `pipenv`:

```
pipenv install mdbenchmark
```

1.2 Usage

Now that the package is installed, you can generate benchmarks for your system. Assuming you want to benchmark a GROMACS 2018.2 simulation on up to 5 nodes, with the TPR file called `md.tpr`, run the following command:

```
mdbenchmark generate -n md --module gromacs/2018.2 --max-nodes 5
```

After generation benchmarks can be submitted:

```
mdbenchmark submit
```

Now, you can also monitor the status of your benchmark with `mdbenchmark`. This will show you the performance of all runs that have finished:

```
mdbenchmark analyze
```

Plotting of the current results can be achieved with `mdbenchmark analyze --plot`.

CHAPTER 2

Content

2.1 Installation

2.1.1 Why isolated environments matter

Installing a new python package into the main python environment of your system can lead to unforeseen consequences. Python packages can have dependencies on different versions of the same package, i.e. `numpy`. If package `packageA` depends on `numpy==1.14.1` and you install `packageB`, which depends on `numpy==1.9.2`, then `packageA` may stop to work. Isolating packages into their own environments makes sure to provide the needed dependencies, while not disrupting the dependencies of other packages (in other environments).

Depending on your setup, there are different ways to create an isolated environment. In the normal Python world, one calls them virtual environment, while users of the Anaconda distribution know them as conda environment.

We recommend to install the package inside a conda environment, while the other ways are also supported.

2.1.2 Install via `conda`

Installation for Anaconda users is handled by `conda`. The following commands create an environment called `benchmark` and install `mdbenchmark` inside.

```
conda create -n benchmark
conda install -n benchmark -c conda-forge mdbenchmark
```

Before every usage of `mdbenchmark`, you need to first activate the conda environment via `source activate benchmark`. After doing this once, you can use `mdbenchmark` for the duration of your shell session.

```
source activate benchmark
```

2.1.3 Install via `pip`

Installation with `pip` should also be done inside a virtual environment.

```
python3 -m venv benchmark-env
```

This created a new directory called `benchmark-env`, if it did not exist before. Now you can activate the environment, as described above.

```
source benchmark-env/bin/activate
```

After activating the environment, you should be able to install the package via `pip`.

Note: The `--user` option leads to the installation of the package in your home directory `$HOME`. If you are not using the option, you may get errors due to missing write permissions.

```
pip install --user mdbenchmark
```

The method requires you to remember where you put the virtual environment and always specify the path when activating. `conda` makes this easier. Several python packages try to make up for this and provide some wrappers, like virtualenvwrapper.

2.1.4 Install via `pipenv`

The easiest way is to install the package is via `pipenv`. First install `pipenv` (refer to its documentation).

```
pip install --user pipenv
```

Now you can let `pipenv` take care of creating the virtual environment. The only downside here is, that you will always need to call `mdbenchmark` from the folder you installed it in.

```
pipenv install mdbenchmark
pipenv run mdbenchmark
```

You can also activate the virtual environment once and then visit different directories afterwards:

```
pipenv shell
cd ..
mdbenchmark
```

2.2 Basic usage of MDBenchmark

Usage of MDBenchmark can be broken down into three points:

1. `generate`
2. `submit`
3. `analyze`

We first generate benchmarks from an input file, e.g., `.tpr` in GROMACS. Afterwards we submit all generated benchmarks to the queuing system of your HPC. Finally, we analyze the performance of each run and generate a plot for easier readability.

MDBenchmark currently supports two MD engines: GROMACS and NAMD. Extensions for AMBER and LAMMPS is planned and help is appreciated. In the following, we will describe the usage of the supported MD engines.

2.2.1 GROMACS

Assuming your TPR file is called `protein.tpr` and you want to run benchmarks with the module `gromacs/2016.4-plumed2.3`, run the following command:

```
mdbenchmark generate --name protein --module gromacs/2016.4-plumed2.3
```

To run benchmarks on GPUs simply add the `--gpu` flag:

```
mdbenchmark generate --name protein --module gromacs/2016.4-plumed2.3 --gpu
```

You can also create benchmarks for different versions of GROMACS:

```
mdbenchmark generate --name protein --module gromacs/2016.4-plumed2.3 --module␣
↪gromacs/2018.2 --gpu
```

2.2.2 NAMD

Note: **NAMD support is experimental.** If you encounter any problems or bugs, we would appreciate to hear from you.

Generating benchmarks for NAMD follows a similar process to GROMACS. Assuming the NAMD configuration file is called `protein.namd`, you will also need the corresponding `protein.pdb` and `protein.psf` inside the same folder.

Warning: Please be aware that all paths given in the `protein.namd` file must be absolute paths. This ensures that MDBenchmark does not destroy paths when copying files around during benchmark generation.

In analogy to the GROMACS setup, you can execute the following command to generate benchmarks for a module named `namd/2.12`:

```
mdbenchmark generate --name protein --module namd/2.12
```

To run benchmarks on GPUs add the `--gpu` flag:

```
mdbenchmark generate --name protein --module namd/2.12-gpu --gpu
```

Be aware that you will need to use different NAMD modules when generating and running GPU and non-GPU benchmarks! To work with GPUs, NAMD needs to be compiled separately and will be probably named differently on the host of your choice. Using the `--gpu` option on non-GPU builds of NAMD may lead to poorer performance and erroneous results.

2.2.3 Usage with multiple modules

You can use this feature to compare multiple versions of one MD engine or different MD engines with each other. Note that the base name for the GROMACS and NAMD files (see above) must to be the same, e.g., *protein.tpr* and *protein.namd*.

mdbenchmark generate –name protein –module namd/2.12 –module gromacs/2016.4

2.3 Generation of benchmarks

We first need to generate benchmarks with MDBenchmark, before we can run and analyze these. All options for benchmark generation are accessible via `mdbenchmark generate`. The options presented in the following text can be chained together in no particular order in one single call to `mdbenchmark generate`.

2.3.1 Specifying the input file

MDBenchmark requires one file to generate GROMACS benchmarks and three files for NAMD. The base name of the input file is provided via the `-n` or `--name` option to `mdbenchmark generate`. The following table lists all files required by the given MD engine.

MD engine	Required files
GROMACS	`.tpr`
NAMD	`.namd`, `.psf`, `.pdb`

If your input file is called `protein.tpr`, then the base name of the file is `protein` and you need to call:

```
mdbenchmark generate --name protein
```

2.3.2 Choosing a MD engine for the benchmark(s)

MDBenchmark assumes that your HPC uses the modules package to manage loading of MD engines. When given the name of a supported MD engine, it will try to find the specified version:

```
mdbenchmark generate --module gromacs/2016.4-plumed2.3
```

It is also possible to specify two or more modules at the same time. MDBenchmark will generate the correct number of benchmark systems for the respective MD engines, sharing all other given options:

```
mdbenchmark generate --module gromacs/2016.4-plumed2.3 --module gromacs/2018.2
```

Also it is possible to mix and match MD engines in a single `mdbenchmark generate` call, if the base name of the files is the same (see above):

```
mdbenchmark generate --module gromacs/2016.4-plumed2.3 --module namd/2.12
```

2.3.3 Skipping module name validation

If MDBenchmark does not manage to determine the naming of your MD engine modules, it will warn you, but continue generating the benchmarks. Contrary, if it manages to determine the naming, but is unable to find the specified version, benchmark generation fails. If you are sure that the name is correct and MDBenchmark is wrong, you can force the generation of benchmark systems with the `--skip-validation` option:

```
mdbenchmark generate --skip-validation
```

2.3.4 Defining the number of nodes to run on

Benchmarks are especially helpful, if you want to figure out on how many nodes you should run your MD job on. You can provide MDBenchmark with a range of nodes to run benchmarks on. The two options defining the range are `--min-nodes` and `--max-nodes` for the lower and upper limit of the range, respectively. If you do not specify either of these two options, MDBenchmark will use the default values of `--min-nodes=1` and `--max-nodes=5`. This would generate a total of 5 benchmarks, running each benchmark on 1, 2, 3, 4 and 5 nodes.

2.3.5 Listing available hosts

MDBenchmark comes with two pre-defined templates for the MPCDF clusters draco and hydra. You can easily create your own job templates, as described *here*. You can list all available job templates via:

```
mdbenchmark generate --list-hosts
```

2.3.6 Defining the job template to run from

MDBenchmark will try to lookup the hostname of your current machine and search for a job template with the same name. If it cannot find the correct file or you want to use one you have written yourself, e.g., named `my_job_template`, simply use the `--host` option:

```
mdbenchmark generate --host my_job_template
```

2.3.7 Running on graphics processing units (GPUs)

The default template for the MPCDF cluster `draco` showcases the ability to run benchmarks on GPUs. Generation of these benchmarks is possible with the `-g` or `--gpu` option:

```
mdbenchmark generate --gpu
```

Note: When generating benchmarks for GPUs, MDBenchmark will also generate the equivalent benchmark for CPUs. If you only want to benchmark on GPUs, you can either delete the CPU folder or not submit these benchmarks. This behavior will be changed in the upcoming version 2.0, where you can choose not to generate CPU benchmarks.

2.3.8 Limiting the run time of benchmarks

You want your benchmarks to run long enough for the MD engine to stop optimizing the performance, but short enough not to waste too much computing time. We currently default to 15 minutes per benchmark, but think that common system sizes (less than 1 million atoms) can be benchmarked in 5-10 minutes on modern HPCs. To change the run time per benchmark, simply use the `--time` option:

```
mdbenchmark generate --time 5
```

This would run all benchmarks for a total of five minutes.

2.4 Submission of benchmarks

After all benchmark systems are generated, you can also use MDBenchmark to submit these to the queuing system on your HPC. We currently support submission to Slurm, SGE and LoadLeveler.

2.4.1 Submitting all generated benchmarks

To submit all generated benchmarks that are recursively found starting in the current directory, use:

```
mdbenchmark submit
```

Note: `mdbenchmark submit` will currently submit all benchmarks in the current folder and its subdirectory without confirmation. Use the `--directory` option to limit this behavior.

2.4.2 Submitting specific benchmarks separately

If you do not want to submit all benchmark systems at once, you can submit them separately with the `--directory` option. Simply define the relative path to the given directory:

```
mdbenchmark submit --directory draco_gromacs/2016.4-plumed2.3
```

2.4.3 Force submitting jobs that were already submitted once

If your jobs were already submitted, but you want to resubmit them once more, you can do so with the `--force` option:

```
mdbenchmark submit --force
```

2.5 Analysis of benchmarks

As soon as the benchmarks have been submitted you can request a summary of the current performance. If a job has not yet finished, not yet started or crashed, MDBenchmark notifies you and marks the affected benchmarks accordingly.

2.5.1 Retrieving the results

The benchmark results can be retrieved immediately after they have been submitted, even if the jobs have not yet started. To do this, simply run:

```
mdbenchmark analyze
```

This will do two things for you:

1. Print the current performance results for all benchmarks found recursively in the current directory.
2. Save the above performance results to a `.csv` file.

The printed results look like this:

```
                    module  nodes  ns/day  run time [min]    gpu        host  ncores
0  gromacs/2016.4-plumed2.3      1  10.878              15  False       draco      32
1  gromacs/2016.4-plumed2.3      2   21.38              15  False       draco      64
2  gromacs/2016.4-plumed2.3      3  34.033              15  False       draco      96
3  gromacs/2016.4-plumed2.3      4       ?              15  False       draco       ?
4  gromacs/2016.4-plumed2.3      5   51.71              15  False       draco     160
```

The results above showcases that MDBenchmark displays jobs that have not finished, started or crashed with a question mark (?).

2.5.2 Defining a name for the CSV file

You can define the name of the output CSV file with the `-o` or `--output-name` option:

```
mdbenchmark analyze --output-name my_benchmark_results.csv
```

2.5.3 Narrow down results to a specific benchmark

Similar to the submission of benchmarks, you can use the `--directory` option to narrow down the performance analysis to a specific path of benchmarks or a single benchmark:

```
mdbenchmark analyze --directory draco_gromacs/2018.2
```

2.5.4 Plotting of benchmark results

MDBenchmark provides a quick and simple way to plot the results of the benchmarks, giving you a `.pdf` file as output. To generate a plot simply use the `--plot` option:

```
mdbenchmark analyze --plot
```

Warning: The plotting function currently only allows to plot CPU and GPU benchmark from the same module, and also assumes that benchmarks were always performed with CPUs. This behavior will be fixed in a future release. If you want to compare different modules with each other, either use the `--directory` option to generate separate plots or create your own plot from the provided CSV file.

2.5.5 Plot the number of cores

You can customize the top of your plot with the `--ncores` option. It accepts an integer value, referring to the number of cores per node. If the option is not given, MDBenchmark will try to read this information from the log file.

2.6 Defining host templates

You can create your own host templates in addition to the ones shipped with the MDBenchmark. We use the `jinja2` Python package for these host templates. Please refer to the *official Jinja2 documentation <http://jinja.pocoo.org/>* for further information on formatting and functionality.

To be detected automatically, a template file must have the same filename as returned by the UNIX command `hostname`. If this is not the case, you can point MDBenchmark to a specific template by providing its name via the `--host` option.

Assuming you created a new host template in your home directory `~/.config/MDBenchmark/my_custom_hostfile`:

```
mdbenchmark generate --host my_custom_hostfile
```

2.6.1 Sun Grid Engine (SGE)

This example shows a HPC running SGE with 30 CPUs per node.

```
#!/bin/bash
### join stdout and stderr
#$ -j y
### change to currend work dir
#$ -cwd
#$ -N {{ name }}
# Number of nodes and MPI tasks per node:
#$ -pe impi_hydra {{ 30 * n_nodes }}
#$ -l h_rt={{ formatted_time }}

module unload gromacs
module load {{ module }}
module load impi

# Run gromacs/{{ version }} for {{ time - 5 }} minutes
mpiexec -n {{ 30 * n_nodes }} -perhost 30 mdrun_mpi -v -maxh {{ time / 60 }} -deffnm {
↪{ name }}
```

2.6.2 Slurm

The following showcases the job template for the MPCDF cluster `draco` using Slurm.

```
#!/bin/bash -l
# Standard output and error:
#SBATCH -o ./{{ name }}.out.%j
#SBATCH -e ./{{ name }}.err.%j
# Initial working directory:
#SBATCH -D ./
# Job Name:
#SBATCH -J {{ name }}
#
# Queue (Partition):
{%- if gpu %}
#SBATCH --partition=gpu
#SBATCH --constraint='gpu'
{%- else %}
{%- if time is lessthan 30 or time is equalto 30 %}
#SBATCH --partition=express
{%- elif time is greaterthan 30 and time is lessthan 240 or time is equalto 240 %}
#SBATCH --partition=short
{%- else %}
#SBATCH --partition=general
```

(continues on next page)

(continued from previous page)

```
{%- endif %}
{%- endif %}
#
# Number of nodes and MPI tasks per node:
#SBATCH --nodes={{ n_nodes }}
#SBATCH --ntasks-per-node=32
# Wall clock limit:
#SBATCH --time={{ formatted_time }}

module purge
module load impi
module load cuda
module load {{ module }}

# Run {{ module }} for {{ time  }} minutes
srun gmx_mpi mdrun -v -maxh {{ time / 60 }} -deffnm {{ name }}
```

2.6.3 LoadLeveler

Here is an example job template for the MPG cluster `hydra` (LoadLeveler).

```
# @ shell=/bin/bash
#
# @ error = {{ name }}.err.$(jobid)
# @ output = {{ name }}.out.$(jobid)
# @ job_type = parallel
# @ node_usage = not_shared
# @ node = {{ n_nodes }}
# @ tasks_per_node = 20
{%- if gpu %}
# @ requirements = (Feature=="gpu")
{%- endif %}
# @ resources = ConsumableCpus(1)
# @ network.MPI = sn_all,not_shared,us
# @ wall_clock_limit = {{ formatted_time }}
# @ queue

module purge
module load {{ module }}

# run {{ module }} for {{ time }} minutes
poe gmx_mpi mdrun -deffnm {{ name }} -maxh {{ time / 60 }}
```

2.6.4 Options passed to job templates

MDBenchmark passes the following variables to each template:

Value	Description
name	Name of the TPR file
gpu	Boolean that is true, if GPUs are requested
module	Name of the module to load
n_nodes	Maximal number of nodes to run on
time	Benchmark run time in minutes
formatted_time	Run time for the queuing system in human readable format (HH:MM:SS)

To ensure correct termination of jobs `formatted_time` is 5 minutes longer than `time`.

MDBenchmark will look for user templates in the xdg config folders defined by the environment variables `XDG_CONFIG_HOME` and `XDG_CONFIG_DIRS` which by default are set to `$HOME/.config/MDBenchmark` and `/etc/xdg/MDBenchmark`, respectively. If the variable `MDBENCHMARK_TEMPLATES` is set, the script will also search in that directory.

MDBenchmark will first search in `XDG_CONFIG_HOME` and `XDG_CONFIG_DIRS` for a suitable template file. This means it is possible to overwrite system-wide installed templates or templates shipped with the package.

CHAPTER 3

Usage reference

3.1 mdbenchmark

Generate, run and analyze benchmarks of GROMACS simulations.

```
mdbenchmark [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

3.1.1 analyze

Analyze finished benchmarks.

```
mdbenchmark analyze [OPTIONS]
```

Options

-d, --directory <directory>
Path in which to look for benchmarks. [default: .]

-p, --plot
Generate a plot of finished benchmarks.

--ncores <ncores>
Number of cores per node. If not given it will be parsed from the benchmarks log file.

-o, --output-name <output_name>
Name of the output .csv file.

3.1.2 generate

Generate benchmarks simulations from the CLI.

```
mdbenchmark generate [OPTIONS]
```

Options

-n, --name <name>
Name of input files. All files must have the same base name.

-g, --gpu
Use GPUs for benchmark. [default: False]

-m, --module <module>
Name of the MD engine module to use.

--host <host>
Name of the job template.

--min-nodes <min_nodes>
Minimal number of nodes to request. [default: 1]

--max-nodes <max_nodes>
Maximal number of nodes to request. [default: 5]

--time <time>
Run time for benchmark in minutes. [default: 15]

--list-hosts
Show available job templates.

--skip-validation
Skip the validation of module names.

3.1.3 submit

Submit benchmarks to queuing system.

benchmarks are searched recursively starting from the directory specified in *–directory*.

Checks whether benchmark folders were already generated, exits otherwise. Only runs benchmarks that were not already started. Can be overwritten with *–force*.

```
mdbenchmark submit [OPTIONS]
```

Options

-d, --directory <directory>
Path in which to look for benchmarks. [default: .]

-f, --force
Resubmit all benchmarks and delete all previous results.

CHAPTER 4

Indices and tables

- search
- genindex

Index

Symbols

M