BenchPRO

Release 1.3

Matthew Cawood

Jun 21, 2023

CONTENTS

BenchPRO is a utility that automates, simplifies and standarizes the process of building applications, executing benchmarks and collecting results on HPC systems.

CHAPTER

ONE

OVERVIEW

BenchPRO provides a benchmarking framework that enforces a standardized approach to compiling and running performance benchmarks. The utility automatically collects and stores significant provenance data associated with the benchmark. The framework also allows performance engineers and domain experts to share their well optimized benchmark 'recipes' in a reproducible manner. This way, someone with limited background of a workflow or science domain can run benchmarks through the framework, compare performance to previous results and examine provenance data to help identify the root cause of any discrepancies. This framework significantly enhances the reproducibility of benchmarking efforts and reduces the labor required to maintain a benchmark suite. The utility was designed to meet the following set of goals:

- Automate the process of building applications, running benchmarks and storing result data.
- Structure the framework to promote the standardization of techniques and workflows as a step towards improving benchmark reproducibility.
- Accommodate a number of benchmarking activities like comparative performance assessments, regression testing, and scalability studies.
- Store as much provenance data as possible for future reference.
- Provide an intuitive way of exploring and comparing benchmark results.

CHAPTER

TWO

BENCHPRO BASICS

2.1 Quick Start

BenchPRO provides a number of example application and benchmark profiles. This Quick Start walks through the process of building an application, running a benchmark and capturing the result on Frontera.

1. Load the BenchPRO module:

```
ml use /scratch1/hpc_tools/benchpro/modulefiles
ml benchpro
```

2. Provide your SLURM allocation using the BPS settings interface:

```
bps allocation=[your allocation]
```

3. Build LAMMPS and run the LJ melt simualation with:

```
bp -b lammps
bp -B ljmelt
```

4. Two jobs will have started, a LAMMPS complication job, and a LJ melt benchmark job. Check the status of LAMMPS with:

```
bp -la
bp -qa lammps
```

5. Query the state of your benchmark with:

```
bp -lr
bp -qr [jobid]
```

6. Once the jobs are complete, capture your result and provenance data to the database:

```
bp -C
```

2.2 New User Guide

This section describes how to use BenchPRO and its features to automate your benchmarking process. To start fast, refer to the *Quick Start* guide.

Note: This guide uses long format input arguments for context, corresponding short format arguments are described *here*.

2.2.1 Terminology

Application: a program or set of programs compiled and used to execute benchmark workloads.
Benchmark: a specific workload/simulation/dataset used to produce a figure of merit. Typically has an application dependency.
Task: an execution instance (via the scheduler or locally on the node) of a compilation or benchmark run.
Template file: a shell script with some variables declared.
Config file: contains a set of variable key-value pairs used to populate the template.
Profile: an application or benchmark available within BenchPRO (i.e. a config & corresponding template file pair)
Overload: replacing a default setting or variable with another one.

2.2.2 Setup BenchPRO

Add BenchPRO to your MODULEPATH and load it:

```
ml use /scratch1/hpc_tools/benchpro/modulefiles
ml benchpro
ml save (optional)
```

Run the initial setup, set your SLURM allocation, then print some helpful info:

```
bp --validate
bps allocation=A-ccsc
bp --version
bp --help
bp --defaults
bp --notices
```

2.2.3 Compile an Application

This section will walk you through installing the LAMMPS application onto Frontera. This guide should also work on other example applications and other TACC systems.

First, print all the pre-configured example applications and benchmark profiles currently provided by BenchPRO with

```
benchpro --avail
```

Install the LAMMPS application with

```
benchpro --build lammps
```

List applications currently installed

```
benchpro --listApps
```

You should see that the status of LAMMPS is `DRY RUN`, this is because dry run mode is enabled by default (`dry_run=True`). Therefore BenchPRO generated a LAMMPS compilation script but did not submit the job to the scheduler. This is useful for testing and troubleshooting a workflow without impacting the system scheduler. You can obtain more information about your LAMMPS build with:

```
benchpro --queryApp lammps
```

Pertenant information is shown here, you can also examine the build script (by default named `job.qsub`) located in the `path` directory. You can submit this LAMMPS compilation script to the scheduler manually, or

Remove the existing dry_run version of LAMMPS with

```
benchpro --delApp lammps
```

Overload the default 'dry_run' value and rebuild LAMMPS with

Check the details and status of your LAMMPS compilation again with

```
benchpro --queryApp lammps
```

In this example, parameters in `$BPS_INC/build/config/frontera/lammps.cfg` were used to populate the template script `$BPS_INC/build/template/lammps.template` and produce a job script within a hierarchical directory structure under `$BP_APPS` ($SCRATCH/benchpro by default). Parameters for the scheduler, system architecture and compile-time optimizations, as well as a module file, were all automatically generated. You can load your LAMMPS module manually with `ml frontera/.../lammps`. Each application built with BenchPRO has a build report generated in order to preserve compilation metadata. BenchPRO uses the module file and build report whenever this application is used to execute a benchmark. You can manually examine LAMMPS's build report located in the build directory or by using the `--queryApp` argument.

2.2.4 Execute a Benchmark

We can now run a benchmark with our LAMMPS installation.

Note: There is no need to wait for the LAMMPS compilation job to complete, BenchPRO is able to create scheduler job dependencies between tasks as required (i.e. the benchmark job will depend on the successful completion of the compilation job). In fact, if the setting `build_if_missing=True`, BenchPRO would detect that LAMMPS was not available for the current system when attempting to run a benchmark and build it automatically without us doing the steps above. The process to run a benchmark is similar to application compilation; a configuration file is used to populate a template script. A benchmark run is specified with `--bench / -B`. Once again you can check for available benchmarks with the `--avail` argument.

Permanently disable the dry run mode with `bps dry_run=False` so that we don't have to overload manually overload the setting on the command line. Refer to the Changing settings section for more information.

Execute the Lennard-Jones benchmark for LAMMPS with

```
benchpro --bench ljmelt
```

Check the benchmark report with

```
benchpro --queryResult ljmelt
```

As this benchmark was the most recent BenchPRO job executed, you can use a useful shortcut to check this report

```
benchpro --last
```

Note: In this example, parameters in `$BPS_INC/bench/config/lammps_ljmelt.cfg` were used to populate the template `$BPS_INC/bench/template/lammps.template`. Much like the application build process, a benchmark report was generated to store metadata associated with this run. It is stored in the benchmark working directory and will be used in the next step to capture the result to the database.

2.2.5 Capture Benchmark Result

Note: A BenchPRO result is considered to be in one of four states, 'pending', 'complete', 'failed' or 'captured'. The benchmark result will remain on the local system until it has been captured to the database, at which time its state is updated to `captured` or `failed`.

Once the benchmark job has been completed, capture results to the database with:

```
benchpro --capture
```

Note: Your unique instance of LAMMPS was recently compiled and is not present in the database, therefore it is also captured to the database automatically.

Display the status of all benchmark runs with

```
benchpro --listResults
```

Query the results database with

```
benchpro --dbList
```

You can print an abridged report of your benchmark with

You can also query your LAMMPS application entry in the database using the [APPID] from above

```
benchpro --dbApp [APPID]
```

Once you are satisfied the benchmark result is valid and its associated files have been uploaded to the database, you can remove the local files with

```
benchpro --delResult captured
```

2.2.6 Web frontend

The captured applications and benchmark results for the TACC site are available through a web portal at http://benchpro.tacc.utexas.edu/

2.3 Integrating your Application

BenchPRO requires two input files (collectively refered to as a 'profile') to build an application: a configuration file containing variables, as well as a template script which will be populated with these variables and executed.

2.3.1 Configuration file

The configuration files used for compiling application contains the following fields:

Label	Required?	Description
[general]		
code	Y	Application identifier.
version	Y	Application version label, accepts x.x, x-x, or strings like 'stable'.
system	N	TACC system identifier, if left blank will use $TACC_SYSTEM.
build_prefix	N	Custom build (outside of default tree).
template	N	Overwrite default build template file.
module_use	N	Path to be added to MODULEPATH, for using nonstandard modules.
sched_cfg	N	Name of nonstandard scheduler config file to use.
[modules]	NOTE: user may add as many custom fields to this section as needed.	
compiler	Y	Module name of compiler, eg: 'intel/18.0.2' or just 'intel' for default.
mpi	Y	Module name of MPI, eg: 'impi/18.0.2' or just 'impi' for default.
[config]	NOTE: user may add as many fields to this section as needed.	
arch	N	Generates architecture specific optimization flags. If left blank will use system default, set to 'system' to combine with 'opt_flags' below
opt_flags	N	**Used to add additional optimization flags, eg: '-g -ipo' etc. If arch is not** set, this will be only optimization flags used.
build_label	N	Custom build label, replaces arch default eg: skylake-xeon. Required if 'opt_flags' is set and 'arch' is not
bin_dir	N	Path to executable within application directory, eg: bin, run etc.
exe	Y	Name of application executable, used to check compilation was successful.
collect_hw_stats	N	Runs the hardware stats collection tool after build.
script_additions	N	Filename in $BP_HOME/templates, to be added to build script.

general

Fields for general application info are provide, such as name and version. You can also specify a custom system label, which limits this application to that specific system (useful for avoiding inadvertantly building the wrong application on a given system). By default BenchPRO will attempt to match this config file with its corresponsing template file using the application name. You can overwrite this default template file by adding the `template` field to this section.

modules

The section `compiler` and `mpi` are required, while more modules can be specified if needed. Every module must be available on the local machine, if you are cross compiling to another platform (e.g. to frontera-rtx) and require system modules not present on the current node, you can set *check_modules=False* in user.ini to bypass this check.

`module_use` can be provided to add a nonstandard path to MODULEPATH

config

where variables used in the build template script can be added.

additionally, you can define as many additional parameters as needed for your application in this section. Eg: configure flags, build options, etc. All parameters `[param]` defined here will be used to fill <<<`[param]`>>> variables of the same name in the template file, thus consistent naming is important. This file must be located in `$BP_HOME/build/config`, preferably with the naming scheme `[label].cfg`.

files

This section allows the user to have BenchPRO automatically collect input files for the build process. BenchPRO currently supports `local` and `download` operations. If BenchPRO detects that the local or downloaded file is an archive, it will automatically extract the archive to the correct working directory. BenchPRO will search for local files in the $BP_REPO directory. The format of the file staging section is:

```
local = [list],[of],[files]
download = [list],[of],[URLs]
```

This file staging process occurs in 1 of 2 ways, depending on the state of `sync_staging` in `$BP_HOME/user.ini`. BenchPRO will either synchronously copy/download/extract during the script creation process, alternatively the file staging will occur during job execution itself. In either case BenchPRO will confirm that the file or URL specified exists before continuing.

overload

This section allows you to modify default parameters for this specific application. Refer to *Changing settings* for additional information.

2.3.2 Build template file

The build template file is used to gerenate a contextualized build script which will executed to compile the application. Variables are defined with <<<[param]>>> syntax and populated with the variables defined in the config file above. If a <<<[param]>>> in the build template is not successfully populated and `exit_on_missing=True` in `$BP_HOME/user.ini`, BenchPRO will abort the build process. You are able to make use of the `local_repo` variable defined in `$BP_HOME/user.ini` to store and use files locally, if you'd rather manage your input files manually. This file must be located in `$BP_HOME/build/templates`, preferablly with the naming scheme `[code_label].template`.

The contextualized build script generated by BenchPRO will have the format:

```
[scheduler options]
[job level details]
[export environment variables]
[load modules]
[file staging]
[user section]
[executable check]
```

2.3.3 Module template file (optional)

It is possible to define your own .lua module template and store it in `$BP_HOME/build/templates` with the naming scheme `[code_label].module`, alternatively BenchPRO will generate a generic module file for you.

The application added above would be built with the following command:

```
benchpro --build [code_label]
```

2.3.4 Example Script

Below is an application compilation job script generated by BenchPRO. Everything outside the 'USER SECTION' is produced by BenchPRO, which depends on parameters provided in the configuration file, as well as the current system and architecture.

```
#!/bin/bash
#SBATCH -J lammps_build
#SBATCH -o /scratch1/06280/mcawood/benchpro/apps/frontera/cascadelake/intel22/
↪intel22impi/lammps/23Jun2022/default/stdout.log
#SBATCH -e /scratch1/06280/mcawood/benchpro/apps/frontera/cascadelake/intel22/
↪intel22impi/lammps/23Jun2022/default/stderr.log
#SBATCH -p small
#SBATCH -t 01:00:00
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -A A-ccsc
echo "START `date +"%Y"-%m-%dT%T` `date +"%s"`"

echo "JobID:    ${SLURM_JOB_ID}"
echo "User:     ${USER}"
echo "Hostlist: ${SLURM_NODELIST}"

export   working_path=/scratch1/06280/mcawood/benchpro/apps/frontera/cascadelake/intel22/
```

(continues on next page)

(continued from previous page)

```
↪intel22impi/lammps/23Jun2022/default/build
export   install_path=/scratch1/06280/mcawood/benchpro/apps/frontera/cascadelake/intel22/
↪intel22impi/lammps/23Jun2022/default/install
export     local_repo=/scratch1/06280/mcawood/benchpro/repo
export        version=23Jun2022
export      opt_flags='-O2 -xCORE-AVX512 -sox'
export            exe=lmp_intel_cpu_intelmpi
export    build_label=default
export        threads=8

# Create application directories
mkdir -p ${install_path}
mkdir -p ${working_path} && cd ${working_path}

# [config]
export                arch=cascadelake
export           opt_flags='-O2 -xCORE-AVX512 -sox'
export         build_label=default
export             bin_dir=
export                 exe=lmp_intel_cpu_intelmpi
export        collect_stats=True
export     script_additions=
export           local_repo=/scratch1/06280/mcawood/benchpro/repo
export               cores=56
export               nodes=1
export              stdout=stdout.log
export              stderr=stderr.log

# [modules]
export            compiler=intel/22.1.2
export                 mpi=intel22/impi/22.1.2

# Load modules
ml use /scratch1/hpc_tools/benchpro-dev/modulefiles
ml use /scratch1/projects/compilers/modulefiles
ml benchpro
ml intel/22.1.2
ml intel22/impi/22.1.2
ml

# Stage Files
stage https://web.corral.tacc.utexas.edu/ASC23006/apps/lammps-23Jun2022.tgz

# Compiler variables
export      CC=icc
export     CXX=icpc
export      FC=ifort
export     F77=ifort
export     F90=ifort
export   MPICC=mpicc
export  MPICXX=mpicxx
export MPIFORT=mpifort
```

(continues on next page)

(continued from previous page)

```
#------USER SECTION------
...
#------------------------

ldd ${install_path}/lmp_intel_cpu_intelmpi
echo "END `date +"%Y"-%m-%dT%T` `date +"%s"`"
```

2.4 Integrating your Benchmark

The process of setting up an application benchmark is much the same as the build process; a config file is used to populate a benchmark template.

2.4.1 Benchmark config file

The configuration files used for running benchmarks contains the following fields:

Label	Required?	Description
[requirements]	NOTE: user may add as many fields to this section as needed.	
code	N	This benchmark requires an installed application matching code=""
version	N	This benchmark requires an installed application matching version=""
build_label	N	This benchmark requires an installed application matching build_label=""
system	N	This benchmark requires an installed application matching system=""
[runtime]		
nodes	Y	Number of nodes on which to run, accepts comma-delimited list.
ranks_per_node	N	MPI ranks per node, accepts comma-delimited list.
threads	N	Threads per MPI rank, accepts comma-delimited list.
gpus	N	Number of GPUs to run on, accepts comma-delimited list.
max_running_jobs	N	Sets maximum number of concurrent running scheduler jobs.
hostlist	Depends	Either hostlist or hostfile required if on local system (bench_mode=local).
hostfile	Depends	
[config]	NOTE: user may add as many fields to this section as needed.	
dataset	Y	Benchmark dataset label, used in ID string.
exe	N	Application executable.
bench_label	N	Optional naming string.
collect_hw_stats	N	Run hardware info collection after benchmark.
script_additions	N	File in $BP_HOME/templates to add to benchmark job script.
arch	N	Required architecture for this benchmark, e.g. cuda.
[result]		
description	N	Result explanation/description.
output_file	N	File to redirect stdout, if empty will use stdout for sched jobs, or 'output_file' from user.ini for local job.
method	Y	Results extraction method. Currently 'expr' or 'script' modes supported.
expr	Depends	Required if 'method=expr'. Expression for result extraction from file (Eg: "grep 'Performance' <file> | cut -d ' ' -f 2")".
script	Depends	Required if 'method=script'. Filename of script for result extraction.
unit	Y	Result units.

requirements

where fields are defined to create requirements to an application. More fields produce a finer, more specific application selection criteria.

runtime

where job setup parameters are defined.

config

where bench script parameters are defined.

result

where result collection parameters are defined.

Any additional parameters may be defined in order to setup the benchmark, i.e dataset label, problem size variables etc. This file must be located in `$BP_HOME/bench/config`, preferably with the naming scheme `[label].cfg`.

2.4.2 Benchmark template file

As with the build template. The benchmark template file is populated with the parameters defined in the config file above. This file should include setup of the dataset, any required pre-processing or domain decomposition steps if required, and the appropriate mpi_exec command. You are able to make use of the `local_repo` variable defined in `$BP_HOME/user.ini` to copy local files.

This file must be located in `$BP_HOME/bench/templates`, with the naming scheme `[label].template`.

The benchmark added above would be run with the following command:

```
benchpro --bench [dataset]
```

Note: BenchPRO will attempt to match your benchmark input to a unique config filename. The specificity of the input will depend on the number of similar config files. It may be helpful to build with `dry_run=True` initially to confirm the build script was generated as expected, before removing and rebuilding with `dry_run=False` to launch the build job.

2.4.3 Example Script

Below is a benchmark job script generated by BenchPRO. Everything outside the 'USER SECTION' is produced by BenchPRO, which depends on parameters provided in the configuration file, as well as the current system and architecture.

```
#!/bin/bash
#SBATCH -J ljmelt
#SBATCH -o /scratch1/06280/mcawood/benchpro/results/pending/mcawood_ljmelt_2023-06-20T14-
↪01_001N_56R_01T_00G/stdout.log
#SBATCH -e /scratch1/06280/mcawood/benchpro/results/pending/mcawood_ljmelt_2023-06-20T14-
↪01_001N_56R_01T_00G/stderr.log
#SBATCH -p small
```

(continues on next page)

(continued from previous page)

```
#SBATCH -t 00:10:00
#SBATCH -N 1
#SBATCH -n 56
#SBATCH -A A-ccsc
echo "START `date +"%Y"-%m-%dT%T` `date +"%s"`"

echo "JobID:    ${SLURM_JOB_ID}"
echo "User:     ${USER}"
echo "Hostlist: ${SLURM_NODELIST}"

export         working_path=/scratch1/06280/mcawood/benchpro/results/pending/mcawood_
↪ljmelt_2023-06-20T14-01_001N_56R_01T_00G
export          output_file=stdout.log
export             mpi_exec=ibrun
export          base_module=/scratch1/06280/mcawood/benchpro/apps/modulefiles
export           app_module=frontera/cascadelake/intel22/intel22impi/lammps/23Jun2022/
↪default
export              threads=1
export                ranks=56
export                nodes=1
export                 gpus=0

# Create working directory
mkdir -p ${working_path} && cd ${working_path}

export             template=lammps
export          bench_label=ljmelt
export              dataset=ljmelt_4M_per_node_250_steps
export        collect_stats=True
export     script_additions=
export                  exe=lmp_intel_cpu_intelmpi
export                 arch=
export           local_repo=/scratch1/06280/mcawood/benchpro/repo
export               stdout=stdout.log
export               stderr=stderr.log
export      OMP_NUM_THREADS=${threads}

# Load Modules
ml use /scratch1/hpc_tools/benchpro-dev/modulefiles
ml use ${base_module}
ml benchpro
ml ${app_module}
ml

# [files]
stage https://web.corral.tacc.utexas.edu/ASC23006/datasets/in.ljmelt_4M_per_node_250_
↪steps


#-------USER SECTION------
...
#-------------------------
```

(continues on next page)

(continued from previous page)

```
# Provenance data collection script
$BPS_INC/resources/scripts/stats/collect_stats $BP_RESULTS/pending/mcawood_ljmelt_2023-
↪06-20T14-01_001N_56R_01T_00G/hw_report

echo "END `date +"%Y"-%m-%dT%T` `date +"%s"`"
```

2.5 Available test cases

BenchPRO is packaged with a number of included applications and benchmark datasets as examples. Some of these examples are generalized to work on any RHEL based system while others have been configured specifically for TACC systems.

Application	Datasets
AMBER20	JAC_NPT_16
	JAC_NVE_16
	Cellulose_NPT
	Cellulose_NVE
	FactorIX_NPT
	FactorIX_NVE
	STMV_NPT
	STMV_NVE
	JAC_NPT_20
	JAC_NVE_20
GROMACS	gromacs_PEP
	gromacs_RIB
LAMMPS	ljmelt
NAMD	namd_apoa1
	namd_stmv
OpenFOAM	cavity_s
	cavity_xl
	SimpleBenchMarkLarge
Quantum Espresso	AUSURF112
SPECFEM3D	
SWIFTsim	Eagle100
	Eagle25
	Pmill_large
WRF	new_conus12km
	new_conus2.5km
	maria1km
MILC	18x18x18x36

In addition several synthetic benchmarks are included:

High Performance Linpack (Intel binary)
High Performance Linpack (CUDA binary)
High Performance Conjugate Gradients
STREAM (x86 & CUDA)

CHAPTER

THREE

INSTALLING BENCHPRO

3.1 Database Installation

3.2 Site Installation

The BenchPRO site package is available from the benchpro-site repo. You should setup the backend result collection database for your site before proceeding with this installation.

Clone the repo

```
git clone https://github.com/TACC/benchpro-site
cd benchpro-site
```

Setup your site specific settings in `site.sh`.

Run the installation script with

```
./install [ssh-key]
```

You will be prompted to provide the database user's private key generated in the datase installation process. Alternatively you can provide it on the command line as the first argument to the install script.

CHAPTER

FOUR

ADVANCED TOPICS

4.1 Useful Features

BenchPRO supports a number of additional features which may be of use.

4.1.1 Changing settings

BenchPRO supports several mechanisms for modifying default settings. These settings can control BenchPRO functionality or any parameters associated with your application or benchmarking process.

1. One-Time, via commandline argument

To temporarily modify these parameters for a single run, the `-o / --overload` argument is available.

Example: enable dry_run mode to test modifications to your benchmark script:

```
benchpro -b lammps --overload dry_run=True
```

Example: run LAMMPS benchmark with modified runtime parameters:

```
benchpro -B ljmelt --overload nodes=16 ranks_per_node=8 threads=6
```

Example: run a collection of benchmarks across a range of hardware, allowing only 1 active task at a time:

```
vim layout.txt
> nodes = 16,32,64
> ranks_per_node=2
> threads=28

benchpro -B ljmelt gromacs_stmv new_conus12km --overload layout.txt max_running_jobs=1
```

2. Profile specific, via configuration file

The compilation/benchmark config files support parameter overloading, which is applied only to that profile.

```
[overload]
sync_staging = True
build_mode = local
```

The above example enforces synchronous file staging as well as execution on the local node via a subshell, rather than submitted to the scheduler. These overloads are only applied to this specific profile.

3. Permentantly, via user.ini

You are able to permenantly modify defaults. Do this by adding key-value pairs to `$BP_HOME/user.ini`. These parameters will be applied to every task BenchPRO executes. To simplify interacting with this overloads, use the BenchSET (`bps`) utility.

Example

```
bps dry_run=False
```

4.1.2 Input list support

Comma delimited lists of nodes, ranks and threads are supported which can be useful for automating scaling and optimization investigations. These lists can be specified in the config file, or via the overload feature detailed above. A list of nodes will be iterated over, and for each, the list of threads and ranks will both be iterated over. If the single thread and multiple ranks are specified, the same thread value will be used for all ranks, and vice versa. If ranks and threads and both larger than a single value but not equal length, an exception will be raised.

Example 1: Run LAMMPS on 4, 8 and 16 nodes, first using 4 ranks per node with 8 threads each, and then 8 ranks per node using 4 threads each:

```
benchpro --bench ljmelt --overload nodes=4,8,16 ranks_per_node=4,8 threads=8,4
```

From this example, the resulting set of runs would look like:

```
Nodes=  4, ranks= 4, threads= 8
Nodes=  4, ranks= 8, threads= 4
Nodes=  8, ranks= 4, threads= 8
Nodes=  8, ranks= 8, threads= 4
Nodes= 16, ranks= 4, threads= 8
Nodes= 16, ranks= 8, threads= 4
```

4.1.3 Local build and bench modes

Allows you to run the generated scripts in a shell on the local machine rather than via the scheduler. Useful for evaluating hardware which is not integrated into the scheduler.

In user.ini *build_mode* and *bench_mode* are responsible for selecting this feature. Values *sched* or *local* are accepted, or an exception will be raised. You can opt to build locally and run via the scheduler, or vice a versa.

4.1.4 Benchmarks with no application dependency

Some benchmarks such as synthetics are microbenchmarks do require an application be compiled and module created. You are able to create a benchmark without any dependency to an application. This is done by not specifying any values in the [requirements] section of the benchmark config file.

4.2 File Format Reference

This page provides a reference to the various input and file structures provided by BenchPRO.

4.2.1 Environment Variables

BenchPRO uses environment variables to configure paths and behaviour. Variables with prefix '**BP_**' are user facing and editable via the setting overload mechanism detailed here. The variables with '**BPS_**' prefix are used internally and should not be editted by the user.

Variable	Description
$BPS_BIN	BenchPRO binaries, added to $PATH
$BPS_COLLECT	BenchPRO's result collection blackhole; used to cache results to be pushed to database.
$BPS_HOME	BenchPRO package installation directory.
$BPS_INC	Site directory containing examples and configuration files.
$BPS_LOG	Log file from package installation.
$BPS_MODULES	Path to BenchPRO's modulefiles directory.
$BPS_SYSTEM	System label.
$BPS_VERSION	Version information.
$BPS_VERSION_STR	Version information, including build ID hash.
$BP_HOME	User file path [Default= $HOME/benchpro].
$BP_APPS	Application install root directory.
$BP_RESULTS	Benchmark root directory.
$BP_REPO	Local file repository directory.

4.2.2 User files layout

The file structure of the BenchPRO user directory is configured as follows:

```
./benchpro
├── bench
│   ├── config
│   └── template
├── build
│   ├── config
│   └── template
├── log
├── resources
│   └── scripts
│       └── results
└── user.ini
```

4.2.3 Input Arguments

4.2.4 Global settings

Global settings are defined in the file `$BP_HOME/user.ini`

4.2.5 Application config files

These config files contain parameters used to populate the application build template file, config files are broken in sections corresponding to general settings, system modules and configuration parameters.

Label	Required?	Description
[general]		
code	Y	Application identifier.
version	Y	Application version label, accepts x.x, x-x, or strings like 'stable'.
system	N	TACC system identifier, if left blank will use $TACC_SYSTEM.
build_prefix	N	Custom build (outside of default tree).
template	N	Overwrite default build template file.
module_use	N	Path to be added to MODULEPATH, for using nonstandard modules.
sched_cfg	N	Name of nonstandard scheduler config file to use.
[modules]	NOTE: user may add as many custom fields to this section as needed.	
compiler	Y	Module name of compiler, eg: 'intel/18.0.2' or just 'intel' for default.
mpi	Y	Module name of MPI, eg: 'impi/18.0.2' or just 'impi' for default.
[config]	NOTE: user may add as many fields to this section as needed.	
arch	N	Generates architecture specific optimization flags. If left blank will use system default, set to 'system' to combine with 'opt_flags' below
opt_flags	N	**Used to add additional optimization flags, eg: '-g -ipo' etc. If arch is not** set, this will be only optimization flags used.
build_label	N	Custom build label, replaces arch default eg: skylake-xeon. Required if 'opt_flags' is set and 'arch' is not
bin_dir	N	Path to executable within application directory, eg: bin, run etc.
exe	Y	Name of application executable, used to check compilation was successful.
collect_hw_stats	N	Runs the hardware stats collection tool after build.
script_additions	N	Filename in $BP_HOME/templates, to be added to build script.

4.2.6 Benchmark config file

These config files contain parameters used to populate the benchmark template script. The file structure is:

4.2.7 Directory structure

Directory	Purpose
$BP_HOME/build	config and template files for compiling applications.
$BP_HOME/bench	config and template files for running benchmarks.
$BP_HOME/log	Build, bench and catpure log files.
$BP_HOME/resources	Contains useful content including modulefiles, hardware collection and result validation scripts.

4.3 Database Structures

4.3.1 Application database

Column	Type	Modifiers	Storage
code	character varying(50)	not null	extended
version	character varying(50)	not null	extended
system	character varying(50)	not null	extended
compiler	character varying(50)	not null	extended
mpi	character varying(50)	not null	extended
modules	character varying(200)	not null	extended
opt_flags	character varying(200)		extended
exe_file	character varying(50)	not null	extended
build_prefix	character varying(200)	not null	extended
task_id	character varying(50)	not null	extended
app_id	character varying(50)	not null	extended
build_label	character varying(50)		extended
module_use	character varying(100)		extended
username	character varying(50)	not null	extended
exec_mode	character varying(100)	not null	extended
bin_dir	character varying(50)		extended
script	character varying(50)		extended
stderr	character varying(50)	not null	extended
stdout	character varying(50)	not null	extended
elapsed_time	integer		plain
end_time	timestamp with time zone		plain
submit_time	timestamp with time zone	not null	plain

4.3.2 Results database

Column	Type	Modifiers	Storage
username	character varying(50)	not null	extended
system	character varying(50)	not null	extended
submit_time	timestamp with time zone	not null	plain
task_id	character varying(50)	not null	extended
nodes	integer	not null	plain
ranks	integer	not null	plain
threads	integer	not null	plain
dataset	character varying(50)	not null	extended
result	numeric(20,3)	not null	main
result_unit	character varying(50)	not null	extended
resource_path	character varying(100)	not null	extended
nodelist	character varying(1000)	not null	extended
description	character varying(100)	not null	extended
elapsed_time	integer		plain
end_time	timestamp with time zone		plain
capture_time	timestamp with time zone	not null	plain
job_status	character varying(100)	not null	extended
app_id	character varying(50)	not null	extended
gpus	integer	not null	plain
exec_mode	character varying(100)	not null	extended

CHAPTER

FIVE

BENCHPRO URLS

- Documentation https://benchpro.readthedocs.io/en/latest/
- Main Repo https://github.com/TACC/benchpro
- Database Repo https://github.com/TACC/benchpro-db