Artifact for Multi-Language Probabilistic Programming

1 Overview

Many different probabilistic programming languages exist that specialize to specific kinds of probabilistic programs, broadly falling into the categories of approximate and exact inference.

This artifact for Multi-Language Probabilistic Programming provides the MultiPPL compiler. MultiPPL is a host compiler of two syntactically and semantically different probabilistic programming languages: an approximate language leveraging importance sampling, and an exact language using binary decision diagrams (BDDs) for knowledge compilation. Our work demonstrates sound interoperation of the inference strategies underlying these two languages.

In our work, we make the following contributions:

- We introduce MultiPPL, a multi-language in the style of [3].
  - This artifact implements MultiPPL as a compiler hosting Disc, a language using exact inference, and Cont, and language using approximate inference.
- Our paper constructs two models of MultiPPL by combining appropriate semantic domains for Disc and Cont programs and prove that these two semantics agree, establishing correctness of the implementation.
  - We provide a proof in the appendix, with the implementation provided through the provided source code.
- We validate the practical effectiveness of MultiPPL by showing that its practicality on heterogeneous examples with complex conditional dependency structures and compare it to PSI[2] and Pyro[1].
  - This claim is validated by Figures 11 and 14. This artifact reproduces and supports these claims with the `multippl-benchmark` script that generates the L1 columns exactly, and reproduces the wall-clock columns relative to the running machine's hardware.

2 Hardware Dependencies

The `multippl` binary requires a Linux operating system on x86$_{64}$ or arm64 architecture[1]. MultiPPL has minimal hardware requirements – it runs a single-threaded, CPU-bound process – but certain MultiPPL programs will require more RAM depending on the complexity of Disc-encoded discrete conditional independency structures.

When verifying the benchmarks and reproducing Figures 11 and 14 we intentionally evaluate memory-intensive MultiPPL programs. These benchmarks can still be safely run on commercial hardware but do take a significant time to run, single-threaded. Depending on how much memory is available, it is possible to parallelize this evaluation and we provide guidelines in the Step-by-Step Instructions.

[1]: For OSX users, we provide docker images built natively for both architectures.

3 Getting Started

Software requirements for MultiPPL are:

- for the `multippl` binary: `tree-sitter`, `cargo` and `rustc` 1.73.0
- for benchmark evaluation: `python` 3.11.6 (with `pyro` 1.8.6, `numpy`, and `scipy`), `psi`, `dice`, `bash`, `ghc`, `bc`

The docker image provides all of the above in `$PATH`, as well as the `multippl-benchmark` script (an alias to `bench/multippl-benchmark.sh`) which runs the benchmarks and produces Fig 11 and 14. For convenience: `zsh`, `emacs`, and `vim` are also available.

Use the docker image to enter a `zsh` shell, replacing `x86_64` with `arm64` depending your operating system architecture.

```
$ wget https://zenodo.org/records/<TBD>/files/multippl-docker-x86_64.tar.gz
$ docker load --input multippl-docker-x86_64.tar.gz
$ docker run -it multippl:latest
```

The provided image is large (>1Gb downloaded, ∼8.5Gb decompressed) so `wget` and `docker load` will take a few minutes to run. Using docker you can also bind a local directory to the container by passing the `-v <HOSTDIR>:<DOCKERDIR>` flag:

```
$ docker run -v $PWD/local:/data/local -it multippl:latest
```

Code examples below assume that they are run in the zsh shell providde by docker. Once in the shell, note the provided examples and source code.

```
# echo "PWD: $PWD" && ls *
PWD: /data
examples:
beta-bernoulli-loop.yo ladder.yo             two-coins-prod.yo
```

```
beta-bernoulli.yo      multimodal-forward.yo two-coins.yo
discrete-choice.yo     multimodal-iter.yo
discrete.yo            multimodal.yo

multippl-source:
Cargo.lock  ZENODO.org  examples    multippl           ttg
Cargo.toml  ZENODO.pdf  flake.lock  nix
README.org  bench       flake.nix   tree-sitter-multippl
```

Run `multippl` on a provided example,

```
# multippl --file examples/beta-bernoulli.yo --steps 1000
0.39581321746787285
80ms
```

Give the source of a MultiPPL program, indicated by the `--file` flag, MultiPPL generates a sampler and produces 1000 samples (indicated by the `--steps` flag). This output is stochastic and will change for each evaluation. To fix the generated output, we can give the random number generator a seed with the `--rng` flag:

```
# multippl --file examples/beta-bernoulli.yo --steps 1000 --rng 1
0.3903635695727347
79ms
```

If validating the benchmarks in the Step-by-Step Instructions, run an individual benchmark to ensure that all binaries are present:

```
# cd multippl-source/bench/grids/3x3/
# time python bench.py --num-runs 1 --threads 1 --initial-seed 1
logdir = logs//grids/3x3/2025-01-01/19:12/
cont.yo(n:1000): 100%|                          | 1/1 [00:00<00:00,  8.63it/s]
diag.yo(n:1000): 100%|                          | 1/1 [00:00<00:00,  5.91it/s]
WARNING! saw unexpected file main.psi
exact.yo(n:1): 100%|                            | 1/1 [00:00<00:00, 87.91it/s]
main.py(n:1000): 100%|                          | 1/1 [00:05<00:00,  5.23s/it]
python bench.py --num-runs 1 --threads 1 --initial-seed 1
4.94s user 0.70s system 99% cpu 5.689 total
```

The `bench.py` script does not run PSI by default. We can verify its presence using `which`:

```
# which psi
/bin/psi
```

4 TODO Step-by-Step Instructions

The following step-by-step instructions describe how to reproduce Figures 11 and 14 from Section 4.2 in our paper submission. L1 data generated will match Figure 11 and 14 exactly. Wall-clock timing will vary by hardware, however relative running time will be consistent with our provided results.

For a comprehensive, example-driven tutorial of MultiPPL, please refer to the `README.org` file in our source code (`multippl-source.tar.gz`). The main `README.org` additionally describes what MulitPPL's importance sampling output looks like for Disc programs, and provides an example of sample consistency.

The `multippl-benchmark` script helps fine-tune the running time of the full evaluation and generates Figure 11 and 14. Because this benchmark is large (100 runs of 1000 program executions for 4/5 languages) we document runtime expectations, different configurations for running `multippl-benchmark`, and how to run benchmarks individually.

4.0.1 TODO Expected running time of `multippl-benchmark`

Figures 11 and 14 from our paper are produced by running all evaluations, single threaded, on an AMD EPYC 7543 Processor with 2.8GHz and 500 GB of RAM. All examples produced in this document and in our README are run on a Thinkpad T14s Gen 3 with an AMD Ryzen 7 PRO 6850U (4.768GHz) CPU and 16GB of RAM.

While there are no explicit hardware requirements for produce Fig 11 and Fig 14, running all benchmarks sequentially will take 8h 21m on the T14s laptop to finish the non-PSI evaluations. Under this sequential configuration, PSI will take an impractical amount of time evaluating 400 programs, only to timeout at the 30-minute mark (used in our figures). Additionally, the benchmark cannot run PSI in parallel on commercial hardware as PSI is recorded to take up to 21G per thread of residential memory for certain programs. This may cause your laptop/desktop to crash.

4.1 Running `multippl-benchmark`

4.1.1 TODO Quick Start

To run our benchmarks sequentially with the default configuration (100 runs, including PSI) simply run

```
$ multippl-benchmark all --logdir /data/logs
```

This outputs all program outputs to the `/data/logs` directory. It may be helpful to run `docker` with the `-v $PWD/logs:/data/logs` flag, so that logs and the cached tables can saved locally.

To run a "quick" evaluation, you can parallelize the benchmark for the non-PSI evaluations, change the timeout (in minutes), limit to 10 runs, and disable evaluating PSI:

```
$ multippl-benchmark all --logdir /data/logs --num-threads 4 \
                         --timeout-min 2 --num-runs 10 --no-psi
```

The above command takes 1h 4m on the T14s laptop referenced above and the provided `--timeout-min` flag will not skip any evaluations. Example logs of the "full" and "quick" evaluations run on the T14s can be found in the `multippl-source/zenodo/example-runs/` folder. Evaluations use fixed seed and users should expect to see the same L1 values for comparable configurations. Examining these logs also gives a better sense of the duration for individual benchmarks. Logs in this directory also reference the rk3588 CPU chipset – this is a 4+4-core CPU (four 2.4GHz cores, four 1.8GHz cores) with 32GB RAM. Evaluations on the corresponding laptop limit docker resource usage to 16GB of memory. Example runs include the following:

- a default run without PSI on T14s (8h 21m): `sequential-100runs-no-psi_T14s-x86_64.log`
- a default run without PSI on rk3588 (22h 34m): `sequential-100runs-no-psi_rk3588-aarch64.log`
- a quick run on T14S (1h 4m): `4threads-10runs-no-psi_T14s-x86_64.log`
  - in this evaluation `grids/9x9` times out for 8 runs of the MultiPPL program (`diag.yo`) and 8 runs of the Disc-only program (`exact.yo`). This is visible in the logs as "counts are not consistent for . `$LOGDIR`! Got: 10 2 2 10." In the final `/data/logs/timeouts` file `grids/9x9` shows 98 timeouts for the associated files (this file is relative to the full 100 evaluations).
- a quick run on rk3588 (2h 13m): `4threads-10runs-no-psi_rk3588-aarch64.log`
  - in this evaluation `grids/9x9` times out for 9 runs for the Disc-only program `exact.yo`, and fails to execute the MultiPPL program `diag.yo`.

4.1.2 Additional configuration options for `multippl-benchmark`

The `multippl-benchmark` script provides more parameters which may be of interest:

```
$ multippl-benchmark
multippl-benchmark (all|tabulate) [OPTIONS]

subcommand: all -- run all benchmarks (psi benchmarks last), then tabulate

    --num-threads NUM_THREADS Number of threads to use for non-psi benchmarks.
                              Default: 1.
    --num-runs NUM_RUNS       Number of runs to use for non-psi benchmarks.
                              Default: 100.
    --num-steps NUM_STEPS     Number of steps per run to use for non-psi,
                              approximate benchmarks. Default: 1000.
    --psi-threads PSI_THREADS Number of threads to use for psi benchmarks.
                              Default: 1.
    --psi-runs PSI_RUNS       Number of runs to use for psi benchmarks.
                              Default: 100.
    --no-psi                  Skip PSI benchmarks.

    --timeout-min TIMEOUT_MIN Number of minutes before a timeout.
                              Default 30.
    --logdir LOGDIR           Directory to store execution logs.
                              Defaults to $PWD/logs.

subcommand: tabulate -- skip benchmarks and tabulate
    --logdir LOGDIR           Directory to store execution logs.
                              Defaults to $PWD/logs.
```

Running the evaluations with the `all` sub-command will produce program executions logged to `$LOGDIR/<category>/<experiment>/<date>/<HH:MM>/`. Additionally, statistics will be aggregated at the end of each experiment and partial results will be output to the terminal. We store this data in the `<date>` folder as `<HH:MM>.data.json`.

The final table is produce by `bench/tabulate.py` and is automatically invoked by `multippl-benchmark all`. Alternatively, it is possible to run this individually using the `multippl-benchmark tabulate` subcommand. The `tabulate.py` script will print all output to the terminal and cache results in the `LOGDIR` folder. If the final table is not produced, the log directory should be cleared and the benchmark should be re-evaluated to prevent rendering a table of partial information. Cached results include:

- `$LOGDIR/hybrid.rich` the table corresponding to Figure 11
- `$LOGDIR/discrete.rich` the table corresponding to Figure 14
- `$LOGDIR/timeouts` a tally of all programs which have timed out.

4.2 Running individual benchmarks

To run an individual benchmark, you must change directory to the experiment in question and run `python ./bench.py`, as well as `python ./avg.py` to render the results.

```
# cd multippl-source/bench/grids/3x3/
# python bench.py --num-runs 1 --initial-seed 1
# python avg.py
```

Python scripts `./bench.py` and `./avg.py` are symlinks to the respective files in `/data/multippl-source/be`
but must be run from a directory with PSI/MultiPPL/Pyro main files.

Additionally, `/data/multippl-source/bench` includes `multippl-benchmark.sh`, the source for the `multippl-benchmark` command, and `tabulate.py`, used to produce the final tables in the `multippl-benchmark tabulate` subcommand.

The `/data/multippl-source/bench/` folder structure is as follows:

- `arrival/` contains subdirectories `tree-15`, `tree-31`, and `tree-63`.
- `bayesnets/` contains subdirectories `alarm`, and `insurance`.
- `grids/` contains subdirectories `3x3`, `6x6`, and `9x9` corresponding to the 9, 36, and 81 evaluations in Fig 11.
- `gossip/` contains subdirectories `g4`, `g10`, and `g20`

Each directory has a mainfile corresponding to the benchmarked tool:

- `main.psi` refers to the PSI program evaluated
- `main.py` refers to the Pyro program evaluated. When imported as a library it provides the derived groundtruth using auxiliary files `truth.py` or `truth.sh`, depending on the benchmark.
- `main.yo` refers to a MultiPPL program with interoperation that is evaluated. We call this file `diag.yo` for the `grids` evaluations, as this specifies the collapsing strategy for interoperation.
- `cont.yo` refers to a MultiPPL program which only defines a Cont program.
- `exact.yo` refers to a MultiPPL program which only defines a Disc program.

Invoking `python bench.py` in an experiment's subdirectory:

- generates logs in the current directory under `logs/`
- default to using half of the threads visible to docker, and
- does **not** run PSI by default
- needs a seed to evaluate deterministically

The flags for `bench.py` differs from `multippl-benchmark`:

```
$ cd /data/multippl-source/bench/arrival/tree-15
$ python bench.py --help
usage: bench.py [-h] [--psi] [--num-runs NUM_RUNS] [--num-steps NUM_STEPS]
                [--initial-seed INITIAL_SEED] [--noti] [--threads THREADS]
                [--logdir LOGDIR]

options:
```

```
-h, --help            show this help message and exit
--psi
--timeout-min TIMEOUT_MIN
--num-runs NUM_RUNS
--num-steps NUM_STEPS
--initial-seed INITIAL_SEED
--noti
--threads THREADS
--logdir LOGDIR
```

Running `avg.py` will also produce two tables and a json object which summarizes the data files in the `$LOGDIR`. These will be cached in the `$LOGDIR`. The tables produced do not guarantee a fixed order: one corresponds to a row of a table, the other will provide a compact view which is easier to compare L1 or wall-clock time. Both tables are output to the terminal when the full benchmarks are produced with `multippl-benchmark`

5 Reusability Guide

A detailed reusability guide in the form of an example-driven tutorial can be found in the `README.org` file in root of our source code (`multippl-source.tar.gz`). This tutorial provides overviews of the Disc and Cont grammars and builds up intuition to run `multippl` on new inputs and examples.

Additional files exist for each submodule of our software, including:

- `multippl/README.org`: a high-level documentation of our compiler passes and entry points for the curious hacker.
  - From a user's perspective, we consider our compiler to be reusable in that it is able to run programs according to our provide grammar.
  - From a researcher's perspective, we consider our compiler to be *hackable* as it provides a library for reuse. The core artifact demonstrates a small ecosystem for sound interoperation of inference. In practice, multi-language probabilistic programming aims to clarify ways to define safe interoperation for practical probabilistic programming languages.
- `ttg/README.org`: a description of our Trees that Grow abstraction.
  - this is not intended to be a reusable component of our software.
- `tree-sitter-multippl/README.org`: high-level overview of our tree-sitter grammar, as well as how to build and test the grammar, and how to get reuse the `*.so` file for syntax highlighting in an emacs distribution.
  - this is portable and can be reused (ie, for syntax highlighting), but this is not the intention of this module.

The docker image used throughout this evaluation is only used for evaluation purposes.

For development, a nix derivation provides a reusable development environment. In this environment, users may additionally validate the test suite (`cargo nextest run`) and generate documentation (`cargo doc`), but this functionality is limited to `x86_64-linux` and `aarch64-linux` users. A summary of the provided nix features can be found in the project's `README.org` file.

References

[1] Eli Bingham et al. "Pyro: Deep Universal Probabilistic Programming". In: *The Journal of Machine Learning Research* (Jan. 1, 2019). PUB6573. arXiv: `1810.09538 [cs, stat]`. URL: `https://dl.acm.org/doi/abs/10.5555/3322706.3322734` (visited on 10/29/2020).

[2] Timon Gehr, Sasa Misailovic, and Martin Vechev. "PSI: Exact Symbolic Inference for Probabilistic Programs". In: *Proc. of ESOP/ETAPS* 9779 (2016), pp. 62–83. ISSN: 16113349. DOI: `10.1007/978-3-319-41528-4_4`.

[3] Jacob Matthews and Robert Bruce Findler. "Operational Semantics for Multi-Language Programs". In: *ACM SIGPLAN Notices* 42.1 (2007), pp. 3–10. DOI: `10.1145/1190215.1190220`.