# Fast fully-reproducible streamlined serial/parallel Monte Carlo/MCMC simulations and visualizations via `ParaMonte::Python` library

**Amir Shahmoradi**[1, 2], **Fatemeh Bagheri**[1], **and Joshua Alexander Osborne**[1]

**1** Department of Physics, The University of Texas, Arlington, TX **2** Data Science Program, The University of Texas, Arlington, TX

**DOI:**

**Software**

- Review
- Repository
- Archive

**Editor:**

**Submitted:** 29 September 2020
**Published:**

## Summary

`ParaMonte::Python` (standing for **Parallel Monte** Carlo in **Python**) is a serial and MPI-parallelized library of (Markov Chain) Monte Carlo (MCMC) routines for sampling mathematical objective functions, in particular, the posterior distributions of parameters in Bayesian modeling and analysis in data science, Machine Learning, and scientific inference. In addition to providing access to fast high-performance serial/parallel Monte Carlo and MCMC sampling routines, the `ParaMonte::Python` library provides extensive post-processing and visualization tools that aim to automate and streamline the process of model calibration and uncertainty quantification in Bayesian data analysis. Furthermore, the automatically-enabled restart functionality of `ParaMonte::Python` samplers ensures seamless fully-deterministic into-the-future restart of Monte Carlo simulations, should any runtime interruptions happen. The `ParaMonte::Python` library is MIT-licensed and is permanently maintained on GitHub.

## Statement of need

Originally developed in 1949 (Metropolis & Ulam, 1949), Monte Carlo simulation techniques, in particular, the Markov Chain Monte Carlo (MCMC) have become one of the most popular methods of uncertainty quantification in quantitative research. A number of Python packages already provide probabilistic programming environments for Markov Chain Monte Carlo simulations within which the user is expected to implement their inference problems in the specific language and syntax designed for the package (Patil, Huard, & Fonnesbeck, 2010), (Team & others, 2017). While such approaches to Monte Carlo simulations can facilitate the implementation of simple inference problems, they can potentially limit the user's ability to implement sophisticated mathematical models whose complexities go beyond what these probabilistic programming language environments can offer.

Other MCMC packages require the objective function to be provided by the user as a black-box with a pre-specified procedural interface (Foreman-Mackey et al., 2019) or require custom definitions for the objective function (Miles, 2019). Regardless of the interface requirements, the majority of the existing Python MCMC environments are limited to Python programming language. Exceptions include the PyStan probabilistic programming environment and `pymcmcstat` which is also available from C++.

The `ParaMonte::Python` library presented in this manuscript aims to fill some of the gaps in the existing tools for (Markov Chain) Monte Carlo simulations. We have built the `ParaMonte::Python` package upon the ParaMonte kernel library which provides high-performance serial/parallel MCMC simulation environment for the C, C++, and Fortran programming languages. Along with `ParaMonte::MATLAB` and the kernel C/C++/Fortran libraries, `ParaMonte::Python` aims to provide a unified Application Programming Interface (API) to a range of (Markov

Chain) Monte Carlo samplers with a similar syntax and usage across several programming languages. This is particularly true about the MATLAB and Python languages, where the syntax of the ParaMonte library as well as the visualization and post-processing tools look and feel almost identical.

As the name of the library indicates, a particular focus of the `ParaMonte::Python` library is to enable scalable parallel Monte Carlo simulations on distributed as well as shared memory architectures. In designing the ParaMonte library, we have followed the *principle of separation of concerns* to separate the computational implementation of objective function from the implementation of the Monte Carlo samplers. In addition, we have developed the library while bearing the following principal design goals in mind:

- **Full automation** of the Monte Carlo simulations while providing extensive descriptions and guidance to the user, in real-time, to ensure the highest level of user-friendliness of the package for running, post-processing, and visualizing Monte Carlo and MCMC simulations.

- **Unified-API** implementation of `ParaMonte::Python` to ensure the library looks and feels almost identical to the API of the ParaMonte library in other programming languages, for example, `ParaMonte::MATLAB`.

- **High-Performance**, meticulously-low-level, implementation of the library's samplers, guaranteeing the fastest-possible Monte Carlo simulations, without compromising the reproducibility or the restart-functionality of the simulations.

- **Parallelizability** of all simulations via distributed-memory MPI communications to ensure the scalability of simulations, from personal laptops to supercomputers, while **requiring zero-parallel-coding efforts from the user**.

- **Zero external-library dependencies** of the kernel of the library, **other than** `numpy`, to ensure hassle-free library installation and Monte Carlo simulation runs. In practice, `ParaMonte::Python` has only one external-library dependency on `numpy` that is essential to perform Monte Carlo simulations, although post-processing and visualization of the results also require `scipy`, `pandas`, `matplotlib`, and `seaborn`. Some of the aforementioned Python libraries (e.g., `seaborn` and `matplotlib`) tend to be unstable or not even available on some computing platforms, in particular, on supercomputers. Therefore, the installation of these libraries is intentionally *not* enforced by the `ParaMonte::Python` installer script since these modules are only needed for the postprocessing and visualization of simulation results. Therefore, the user has the responsibility to install these libraries prior to using `ParaMonte::Python` for post-processing of simulation results. We have made sure to provide ample guidance and warnings to the user about this issue, when the `paramonte` module is imported to the user's Python environment for the first time.

- **Fully-deterministic reproducibility** and **automatically-enabled restart functionality** for all Monte Carlo and MCMC simulations, which guarantee full recovery of simulations, should an interruption happen at any stage.

- **Automatically-enabled comprehensive-reporting and post-processing** of the simulation results and their efficient compact storage in external files to ensure that simulation results will be reproducible and comprehensible in the future.

## Installation

The ParaMonte::Python library is permanently maintained on GitHub and is available at: https://github.com/cdslaborg/paramonte/tree/master/src/interface/Python. Each release of the library is also available on the The Python Package Index (PyPI) repository. The most straightforward method of installation is via `pip` on an Anaconda3 command-prompt on Windows or in a Python-aware Bash terminal,

```
pip install --user --upgrade paramonte
```

A primary goal in the design of the ParaMonte library has been the automation of serial and parallel-scalable Monte Carlo simulations. Therefore, upon the first `import paramonte` action in a Python environment, the library checks for the existence of the MPI runtime libraries on the user's system and if needed, automatically installs the MPI library for parallel ParaMonte simulations. The entire process is performed with the explicit permission from the user.

It is also possible to build the ParaMonte::Python library from the source files. In such case, the entire build process of the library is also fully automated. The one-line commands to automatically and locally build the library on a given system are available on the documentation website of the library. The pre-built ParaMonte::Python library also ships with a `build()` function that can automatically build the library from source, locally, on the user's Unix system. This includes the automatic installation of the GNU Compiler collection, the MPI libraries, and any other build prerequisites (e.g., cmake, ...), if needed. All of these tasks are performed with the explicit permission from the user.

## The ParaDRAM sampler

The current implementation of ParaMonte::Python library contains the ParaDRAM sampler, standing for **Parallel** **D**elayed-**R**ejection **A**daptive **M**etropolis Markov Chain Monte Carlo (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020a), (Shahmoradi & Bagheri, 2020b), (Shahmoradi & Bagheri, 2020c), (Kumbhare & Shahmoradi, 2020).

The ParaDRAM sampler is a special variant of the DRAM algorithm by (Haario, Laine, Mira, & Saksman, 2006) and can be used in serial or parallel mode. Unlike the traditional MCMC samplers where the proposal distribution remains fixed throughout the simulation, the ParaDRAM sampler ensures fast convergence to the objective function by continuously adapting the proposal distribution of the MCMC sampler to the shape of the objective function. The algorithm provides a highly flexible and customizable simulation environment via an extensive number of input specifications for the MCMC simulations. These simulation specifications can be either set by the user, or left to be appropriately determined by the sampler. A complete description of these specifications go beyond the limits of this manuscript, but can be found on the documentation website of the library. In addition, every ParaDRAM or ParaMonte simulation automatically generates an output `*_report.txt` file containing the descriptions of all simulation specifications as well as post-processing of the simulation performance and results.

### Example Usage

A major focus in the development of ParaMonte::Python has been on the user-friendliness of the library and providing dynamic on-the-fly user-guidance and directions toward the next step in the simulation process. As such, setting up a ParaDRAM simulation, whether serially or in parallel (via hundreds of supercomputer cores), and visualizing the results can be achieved with minimal coding efforts by the user. For example, sampling a Multivariate Normal distribution can be achieved via the following code in serial mode,

```
import numpy as np
import paramonte as pm
```

```
def getLogFunc(point): return -0.5 * np.sum( point**2 )
pmpd = pm.ParaDRAM()
pmpd.runSampler ( ndim = 4 # assume 4-dimensional objective function
                , getLogFunc = getLogFunc    # the objective function
                )
```

Each `ParaMonte::Python` simulation generates a number of output files that contain information about one aspect of the simulation. A complete description of these files is provided on the documentation website of the library. The library also provides a number of visualization tools that aim to streamline the post-processing of simulation results. For example, generating a `gridplot` of all sampled states requires only one line of Python command,

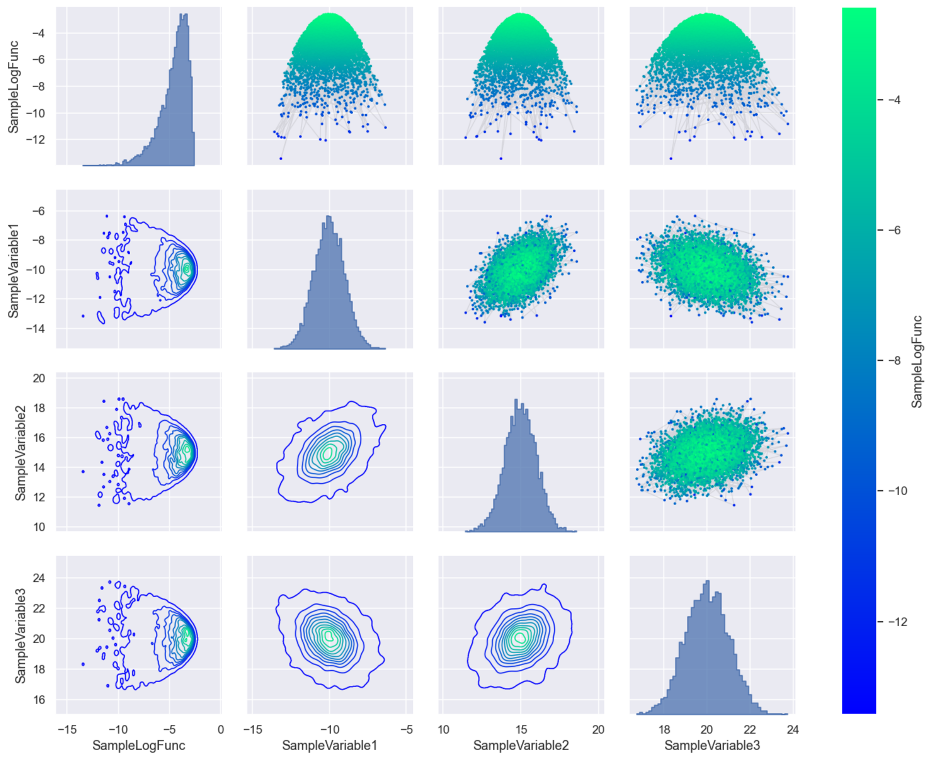


**Figure 1:** An example `grid` plot displaying the results of sampling a 4-dimensional MVN distribution with a random covariance matrix.

```
sample = pmpd.readSample(renabled=True)[0] # input filename is optional
sample.plot.grid() # make a grid plot
```

An example output of the above command is shown in Figure 1 for sampling a 4-dimensional Multivariate Normal (MVN) distribution. Each plotting tool of `ParaMonte::Python`, including the `grid` plot, contains a large number of attributes that are automatically set to the appropriate default values to enable quick streamlined visualizations. If desired, however, these default values can be easily changed by the user to alter the behavior, display, and contents of the resulting figures.

## Efficient compact storage of the output

External storage of the output of Monte Carlo simulations is a highly desired feature for both post-processing of the results and restarting an interrupted simulation. The restart functionality is, in particular, vital for large-scale computationally-expensive parallel Bayesian inference problems. However, as the complexity of the target density and the problem size increase, external storage of the simulation output can quickly become a bottleneck in the computational speed of the simulation.

The ParaDRAM sampler automatically generates multiple output files to ensure seamless post-processing and reproducibility of the results, in particular, the automatic restart of interrupted simulations. To minimize the effects of external input/output (IO) on the runtime speed of the ParaDRAM sampler, we have implemented a novel method of carefully storing the resulting MCMC chains from the sampler in a small, *compact*, yet human-readable (ASCII) format in external output files.

The resulting output **compact-chain** (versus **verbose Markov-chain**) format can lead to a significant speedup of the simulation while demanding 4-100 times less external storage for the simulation output. Additionally, the format of the output chain and restart files can be also set to `binary`, further reducing the memory foot-print of the simulation and increasing the simulation speed. The implementation details of this compact-chain format go beyond the scope of this paper, but are provided in (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020b).

## Monitoring Convergence

The continuous adaptation of the proposal distribution of the adaptive Markov Chain Monte Carlo samplers is an issue that requires special attention and care. Such adaptation, although increases the sampling efficiency, can potentially break the reversibility and ergodic properties of the Markov Chain. Theoretical results, however, indicate that the convergence of the adaptive Markov chain to the target density is guaranteed, as long as the adaptation of the Markov chain monotonically decreases throughout the simulation (Haario et al., 2006).

It is, therefore, crucial to measure and dynamically monitor the amount of adaptation in ParaDRAM simulations to ensure the adaptation of the proposal distribution diminishes progressively throughout the simulation. This is, however, a challenging task that requires the quantification of the difference between two potentially-multidimensional probability distributions. In (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020b), we have introduced a novel technique to circumvent this computationally NP-hard problem by computing an upper bound on the total variation distance between any subsequent pairs of adaptively-updated proposal distributions.

This `AdaptationMeasure` is a real number between 0 and 1 with 0 implying no proposal adaptation and 1 implying extreme adaptation. This quantity is automatically written to the output `*_chain.txt` file for each ParaDRAM simulation, which can be explored and visualized to ensure that the ergodicity and the reversibility properties of the Markov chain hold.

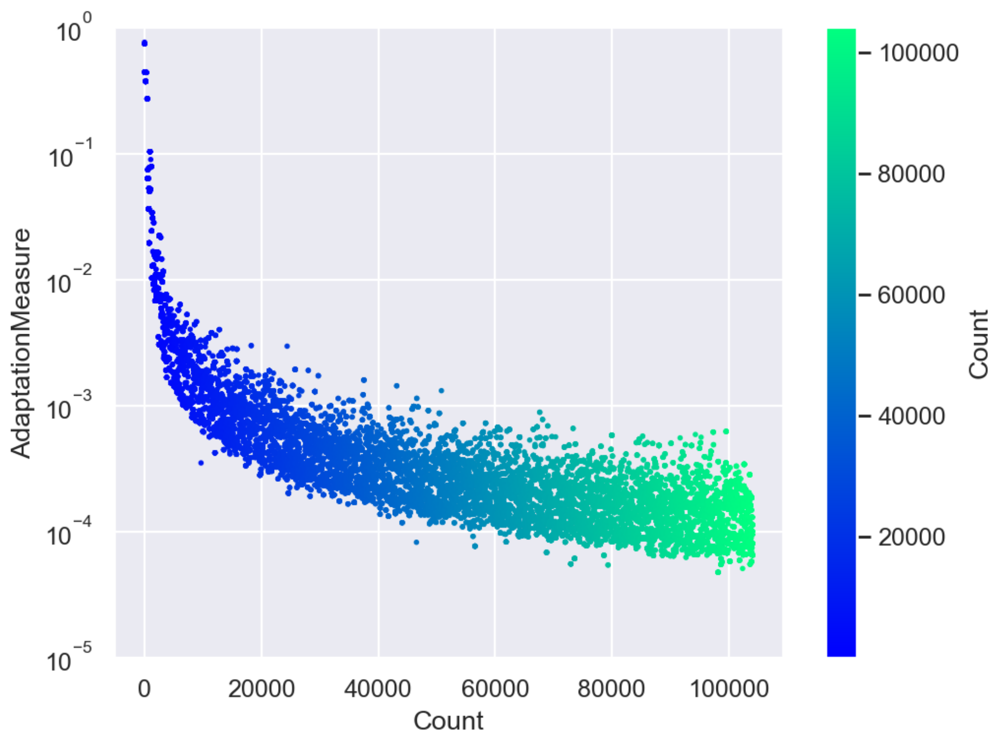


**Figure 2:** An illustration of the diminishing-adaptation criterion of the Delayed-Rejection Adaptive Metropolis Markov Chain Monte Carlo (ParaDRAM) sampler of `ParaMonte::Python` library for the problem of sampling a 4-dimensional Gaussian density function.

Figure 2 illustrates the dynamics of `AdaptationMeasure` for the problem of sampling a 4-dimensional MVN density function. The observed behavior of `AdaptationMeasure` in this figure is precisely the kind of diminishing adaptation one would hope to witness in an adaptive Markov Chain Monte Carlo simulation. Furthermore, a non-diminishing `AdaptationMeasure` can be a strong indicator of lack of convergence of the Markov chain to the target density function.

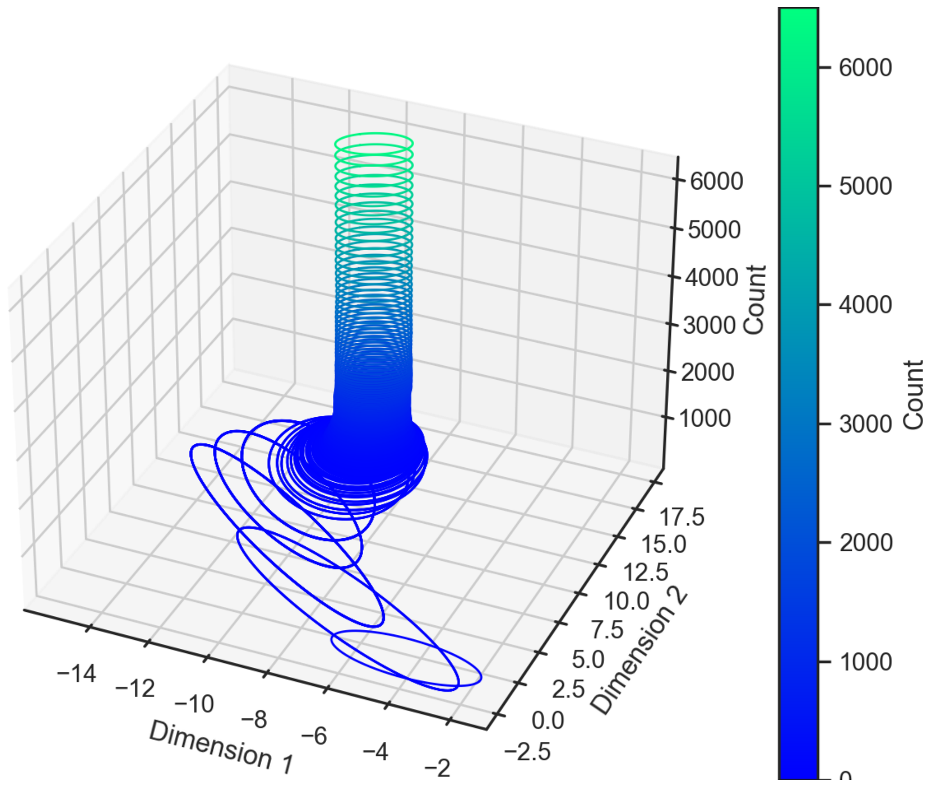


**Figure 3:** A 3D illustration of the dynamic adaptation of the covariance matrix of the 4-dimensional Gaussian proposal distribution of the ParaDRAM sampler for an example problem of sampling a 4-dimensional MVN target density function. As seen, the amount of adaptation of the proposal distribution quickly converges to zero, ensuring the ergodicity of the resulting Markov chain.

Figure 3 illustrates the dynamics of the adaptation of the proposal distribution's covariance matrices throughout a ParaDRAM sampling of the same 4-dimensional MVN target density function. The information displayed in this figure is automatically saved in the output `*_restart.txt` file of every ParaDRAM simulation and can be parsed via a single ParaDRAM command in Python,

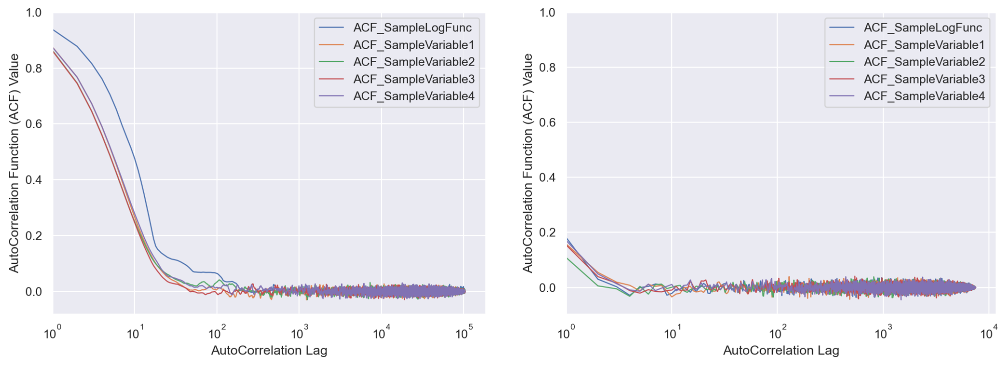
```
restart = pmpd.readRestart(renabled=True)[0]
```

The visualization of this information as depicted in Figure 3 can be achieved via another single-line Python command,

```
restart.plot.covmat3()
```

Just as with other visualization tools of `ParaMonte::Python` library, the aesthetics and contents of the plot can be readily controlled via the attributes of the generated *callable* Python figure object `restart.plot.covmat3`.

## Sample refinement



**(a)** The Markov chain autocorrelation. **(b)** The refined-sample autocorrelation.

**Figure 4:** An illustration of the result of the aggressive and recursive refinements of the output Markov chain that is automatically performed by the ParaDRAM algorithm as part of post-processing of the results.

The ParaDRAM algorithm automatically refines, aggressively and recursively, each output Markov chain from a ParaDRAM simulation, such that the resulting chain exhibits no autocorrelation. Figure 4 compares the autocorrelation of the raw verbose (Markov) chain from an example ParaDRAM simulation run with the autocorrelation of the corresponding refined sample. The recursive refinement procedure that we have implemented in ParaDRAM is extensively detailed in (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020a), (Shahmoradi & Bagheri, 2020b).

## Parallelism

The `ParaMonte::Python` parallel simulations are currently enabled via the Message-Passing-Interface communication paradigm to ensure the high-performance and scalability of the simulations across multiple clusters of processors. The library can automatically detect the presence of MPI runtime libraries required for parallel simulations on the user's system. If any component is missing, it can also automatically install the missing libraries with the explicit permission from the user. The `ParaMonte::Python` samplers currently support two modes of parallelism (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020b),

- The **Perfect Parallelism** (multi-Chain), in which independent instances of the Markov Chain are simulated. Upon finishing the simulation, the ParaDRAM algorithm compares the output refined samples from all processors with each other to ensure that there is no evidence for a lack-of-convergence to the objective function.
- The **Fork-Join Parallelism** (single-Chain), in which a single Markov chain is generated by the main processor, while other processes assist in proposing new states. This mode is the default behavior of the ParaDRAM sampler when used in parallel.

An example of a simple parallel ParaDRAM simulation sampling a 4-dimensional MVN is the following,

```
import numpy as np
import paramonte as pm
def getLogFunc(point): return -0.5 * np.sum( point**2 )
pmpd = pm.ParaDRAM()
pmpd.mpiEnabled = True
pmpd.runSampler ( ndim = 4 # assume 4-dimensional objective function
                , getLogFunc = getLogFunc    # the objective function
                )
```

Compared with serial simulations, the only extra piece of information that is required from the user is the Python statement `pmpd.mpiEnabled = True` in the above script. Other than setting this variable, the `ParaMonte::Python` samplers require *zero parallel coding efforts from the user* to run simulations in parallel.

Assuming the above Python script is saved in a file named `main.py`, running the simulation in parallel on 3 processes is a one-line command in a bash or Anaconda3 terminal,

```
mpiexec -n 3 python main.py
```

Depending on the platform (in particular, on Windows and supercomputers), the calling syntax of the MPI runtime library might be slightly different. Comprehensive details and example simulations in parallel on a variety of platforms are provided in the documentation of the library.

### Setting the optimal number of processors

The ParaMonte samplers automatically compute and output the parallel speedup compared to the serial mode. In addition, the samplers automatically predict the optimal number of processors for the simulation in parallel Fork-Join mode, given the characteristics of the current parallel simulation run. These computations and predictions are automatically stored in the `*_report.txt` files that accompany each simulation.

Although the predicted optimal number of processors is a first-order approximation, it can be a very useful guide to set the number of processes for large-scale computationally demanding simulations. In such cases, the user can run a short parallel simulation with an arbitrary number of processors and check the output report file of the simulation to gain insight into the optimal number of processes for the full-production parallel simulation. A complete description of the algorithm that approximates the optimal number of processes is given in (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020b).

In (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020b), we show that the contribution of multiple processors to the construction of a single Markov Chain in the Fork-Join parallelism paradigm follows a Geometric distribution. Figure 5 illustrates an example contribution distribution of 512 Intel Xeon Phi 7250 processors to the construction of a single Markov chain exploring a 4-dimensional MVN target density by the ParaDRAM sampler in the Fork-Join (`single-chain`) parallelism mode.

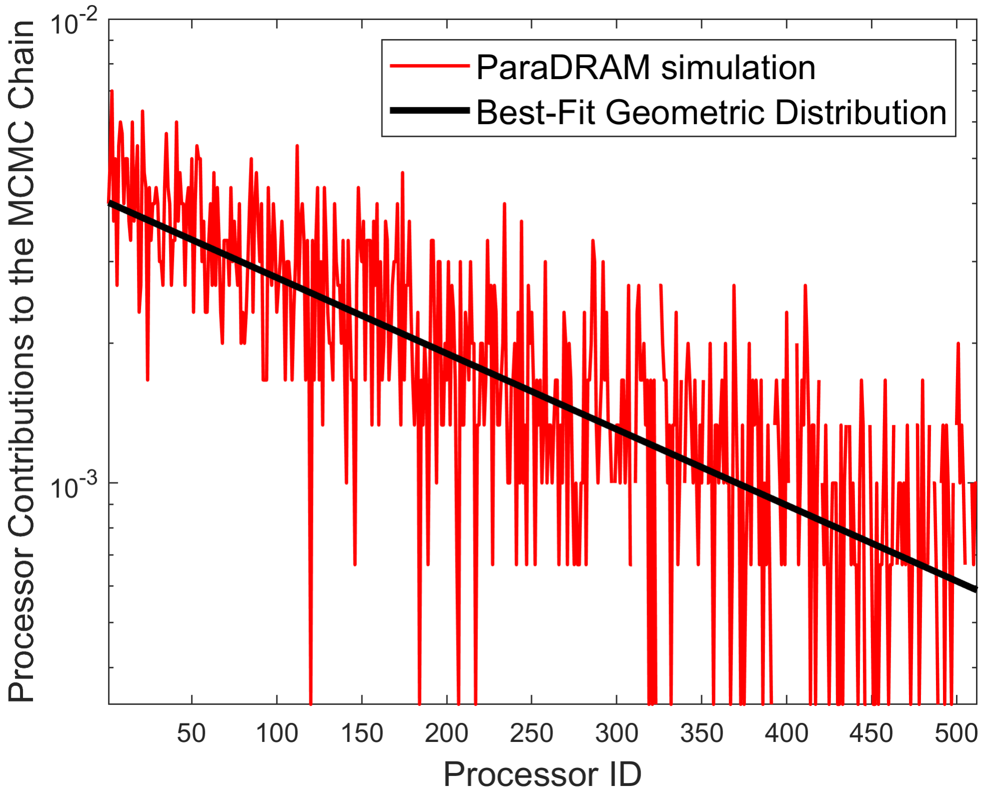


**Figure 5:** The distribution of the contributions of 512 processors to a ParaMonte::Python ParaDRAM simulation via the Fork-Join paradigm. The balck line represents the best-fit Geometric distribution predicted in the post-processing phase of the ParaDRAM simulation (Shahmoradi & Bagheri, 2020), (Shahmoradi & Bagheri, 2020b). The entire data used in this plot is automatically computed and outputted to the external `*_report.txt` file for each parallel ParaMonte simulation.

## Fully-deterministic into-the-future restart functionality

A unique feature of the `ParaMonte::Python` library is the automatically-enabled fully-deterministic into-the-future restart functionality of all serial and parallel Monte Carlo simulations. For example, restarting an interrupted ParaDRAM simulation is as simple as rerunning the simulation with in its original settings. To ensure a seamless fully-deterministic restart, all that is required from the user is to fix the simulation attributes corresponding to the output filename and the seed of the random number generator of the sampler.

The restart functionality of ParaMonte samplers have been carefully designed to enable a fully-deterministic reproduction of the originally interrupted simulation, such that the resulting full chain from the simulation restart would be the same as the original chain, had the simulation not been interrupted in the first place. This identity of the original and restart chains is exact up to 16 digits of decimal precision.

The restart functionality of `ParaMonte::Python` samplers is particularly desired and useful in larges-scale parallel Monte Carlo simulations and Bayesian inverse problems on supercomputers, given the fact that the computational time is frequently limited to less than 24 hours for any given simulation. In such cases, all that is needed to continue a single simulation during multiple separate segments of allocated supercomputer times, is to simply rerun (restart) the incomplete simulation.

## Documentation and Repository

The `ParaMonte::Python` library presented in this manuscript builds upon the ParaMonte library for C, C++, and Fortran (Shahmoradi & Bagheri, 2020a). Although the Python version of the library has been only recently released, the kernel samplers of the ParaMonte library have been already used in number of peer-reviewed publication (Shahmoradi, 2013), (Shahmoradi, 2013), (Shahmoradi & Nemiroff, 2014), (Shahmoradi & Nemiroff, 2015), (Shahmoradi & Nemiroff, 2019), (Osborne, Shahmoradi, & Nemiroff, 2020), (Osborne, Shahmoradi, & Nemiroff, 2020).

An equivalent of `ParaMonte::Python` is also available for MATLAB programming environment, with the same set of simulation and visualization tools and with a syntax and usage that is highly similar to those of `ParaMonte::Python`. Consequently, to ensure the similarity of the ParaMonte package and simulation environment across different programming languages, we have used and enforced the `camelCase` convention in the development of the library in all programming languages, including Python.

Extensive documentation and examples in Python (as well as C, C++, Fortran, MATLAB and other programming languages) are available on the documentation website of the library at: https://www.cdslab.org/paramonte/. The ParaMonte library is MIT-licensed and is permanently located and maintained at https://github.com/cdslaborg/paramonte/tree/master/src/interface/Python.

## Acknowledgements

We acknowledge the use of supercomputing resources at Texas Advanced Computing Center for the development and testing of `ParaMonte::Python` library.

## References

Foreman-Mackey, D., Farr, W. M., Sinha, M., Archibald, A. M., Hogg, D. W., Sanders, J. S., Zuntz, J., et al. (2019). Emcee v3: A python ensemble sampling toolkit for affine-invariant mcmc. *arXiv preprint arXiv:1911.07688*.

Haario, H., Laine, M., Mira, A., & Saksman, E. (2006). DRAM: Efficient adaptive mcmc. *Statistics and computing*, *16*(4), 339–354.

Kumbhare, S., & Shahmoradi, A. (2020). Parallel adapative monte carlo optimization, sampling, and integration in c/c++, fortran, matlab, and python. *Bulletin of the American Physical Society*.

Metropolis, N., & Ulam, S. (1949). The monte carlo method. *Journal of the American statistical association*, *44*(247), 335–341.

Miles, P. R. (2019). Pymcmcstat: A python package for bayesian inference using delayed rejection adaptive metropolis. *Journal of Open Source Software*, *4*(38), 1417.

Osborne, J. A., Shahmoradi, A., & Nemiroff, R. J. (2020). A multilevel empirical bayesian approach to estimating the unknown redshifts of 1366 batse catalog long-duration gamma-ray bursts.

Osborne, J. A., Shahmoradi, A., & Nemiroff, R. J. (2020). A Multilevel Empirical Bayesian Approach to Estimating the Unknown Redshifts of 1366 BATSE Catalog Long-Duration Gamma-Ray Bursts. *arXiv e-prints*, arXiv:2006.01157.

Patil, A., Huard, D., & Fonnesbeck, C. J. (2010). PyMC: Bayesian stochastic modelling in python. *Journal of statistical software*, *35*(4), 1.

Shahmoradi, A. (2013). A multivariate fit luminosity function and world model for long gamma-ray bursts. *The Astrophysical Journal*, *766*(2), 111.

Shahmoradi, A. (2013). Gamma-Ray bursts: Energetics and Prompt Correlations. *arXiv e-prints*, arXiv:1308.1097.

Shahmoradi, A., & Bagheri, F. (2020). ParaDRAM: A cross-language toolbox for parallel high-performance delayed-rejection adaptive metropolis markov chain monte carlo simulations. *arXiv preprint arXiv:2008.09589*.

Shahmoradi, A., & Bagheri, F. (2020a). ParaMonte: A high-performance serial/parallel Monte Carlo simulation library for C, C++, Fortran. *arXiv e-prints*, arXiv:2009.14229.

Shahmoradi, A., & Bagheri, F. (2020b). ParaDRAM: A Cross-Language Toolbox for Parallel High-Performance Delayed-Rejection Adaptive Metropolis Markov Chain Monte Carlo Simulations. *arXiv e-prints*, arXiv:2008.09589.

Shahmoradi, A., & Bagheri, F. (2020c, August). ParaMonte: Parallel Monte Carlo library.

Shahmoradi, A., & Nemiroff, R. (2014). Classification and energetics of cosmological gamma-ray bursts. In *American astronomical society meeting abstracts# 223* (Vol. 223).

Shahmoradi, A., & Nemiroff, R. J. (2015). Short versus long gamma-ray bursts: A comprehensive study of energetics and prompt gamma-ray correlations. *Monthly Notices of the Royal Astronomical Society*, *451*(1), 126–143.

Shahmoradi, A., & Nemiroff, R. J. (2019). A Catalog of Redshift Estimates for 1366 BATSE Long-Duration Gamma-Ray Bursts: Evidence for Strong Selection Effects on the Phenomenological Prompt Gamma-Ray Correlations. *arXiv e-prints*, arXiv:1903.06989.

Team, S. D., & others. (2017). PyStan: The python interface to stan. *Version 2.16. 0.0*.