Noname manuscript No.
(will be inserted by the editor)

ParaDRAM: A Cross-Language Toolbox for Parallel High-Performance Delayed-Rejection Adaptive Metropolis Markov Chain Monte Carlo Simulations

Amir Shahmoradi[1] · **Fatemeh Bagheri**[1]

Received: date / Accepted: date

Abstract We present ParaDRAM, a high-performance **Para**llel **D**elayed-**R**ejection **A**daptive **M**etropolis Markov Chain Monte Carlo software for optimization, sampling, and integration of mathematical objective functions encountered in scientific inference. ParaDRAM is currently accessible from several popular programming languages including C/C++, Fortran, MATLAB, Python and is part of the ParaMonte open-source project with the following principal design goals: 1. **full automation** of Monte Carlo simulations, 2. **interoperability** of the core library with as many programming languages as possible, thus, providing a unified Application Programming Interface and Monte Carlo simulation environment across all programming languages, 3. **high-performance** 4. **parallelizability** and scalability of simulations from personal laptops to supercomputers, 5. virtually **zero-dependence on external libraries**, 6. **fully-deterministic reproducibility** of simulations, 7. **automatic comprehensive reporting** and post-processing of the simulation results. We present and discuss several novel techniques implemented in ParaDRAM to automatically and dynamically ensure the good-mixing and the diminishing-adaptation of the resulting pseudo-Markov chains from ParaDRAM. We also discuss the implementation of an efficient data storage method used in ParaDRAM that reduces the average memory and storage requirements of the algorithm by, a factor of 4 for simple simulation problems to, an order of magnitude and more for sampling complex high-dimensional mathematical objective functions. Finally, we discuss how the design goals of ParaDRAM can help users readily and efficiently solve a variety of machine learning and scientific inference problems on a wide range of computing platforms.

Contents

[1] Department of Physics,
Data Science Program, College of Science
The University of Texas, Arlington, TX 76010
E-mail: a.shahmoradi@uta.edu
E-mail: fatemeh.bagheri@uta.edu

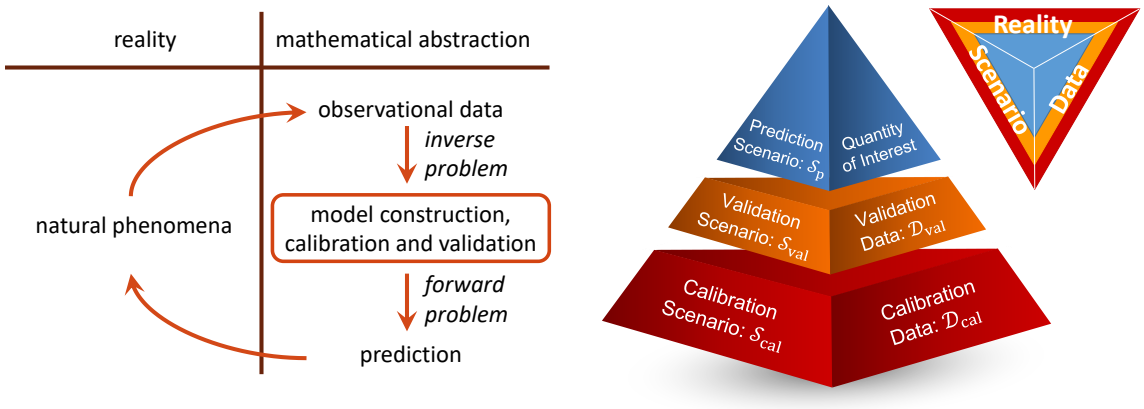
1 Introduction

At the very foundation of predictive science lies the scientific method which involves multiple steps of making observations and developing testable hypotheses and theories of natural phenomena. Once a scientific theory is developed, it can be cast into a mathematical model which then serves as a proxy-seek of the truth. Then, the free parameters of the model, if any, has to be constrained, or *calibrated*, using the *calibration data* in a process known as the *model calibration* or the *inverse-problem*. The validity of the model – and thereby, the scientific theory behind the model – are subsequently tested against a *validation dataset* that has been collected independently of the calibration data. Once the model is calibrated and validated, it can be used to predict the quantity of interest (QoI) of the problem in a process known as the *forward-problem* [49]. This entire procedure is schematically illustrated in Figure 1a. The hierarchy of model calibration, validation, and prediction has become known as the *prediction pyramid*, as depicted in Figure 1b [47, 48, 50, 52].

A major task in the calibration step of every scientific prediction problem is to find the best solution – among the potentially infinite set of all possible solutions – to a mathematically-defined problem, where the mathematical model serves as an abstraction of the physical reality (Figure 1a). Specifically, finding the best solution (i.e., the best-fit model parameters) requires,

1. the construction of one (or more) mathematical objective function(s) that quantifies the goodness of each set of possible parameters for the model, and then,
2. optimizing the objective function(s) (for **parameter tuning**), and/or,
3. sampling the objective function(s) (for **uncertainty quantification**), and/or,
4. integrating the objective function(s) (for **model selection**).

In this process, optimization is primarily performed to obtain the best-fit parameters of the model given the calibration dataset. The history of mathematical optimization dates back to the emergence of modern science during the Renaissance, perhaps starting with a paper by Pierre de Fermat in 1640s on finding the local extrema of differentiable functions [13, 37]. A second revolution in the field of optimization occurred with the (re-)discovery [17] of linear programming by Dantzig in 1947 [12], followed by the first developments in *nonlinear programming* [33], *stochastic programming* [4, 11], and the revival of interest in *network flows*, *combinatorial*



(a) The scientific methodology. **(b)** The prediction pyramid.

Fig. 1: (a) An illustration of the fundamental steps in predictive science, which includes data collection, hypothesis formulation, as well as the construction of a mathematical model and objective function, which is subsequently optimized to constrain the parameters of the model in a process known as *inversion* or *inverse problem*. Once validated, the model can be used to make predictions about the quantity of interest (*forward problem*). **(b) The prediction pyramid**, depicting the three hierarchical levels of predictive inference from bottom to top: Calibration, Validation, and Prediction of the Quantity of Interest (QoI). The rear face of the tetrahedron represents reality (truth), $\mathcal{R}$, about the set of observed phenomena, which is never known to the observer. The front-right face of the tetrahedron represents the observational data, $\mathcal{X}$, which results from the convolution of the truth/reality, $\mathcal{R}$, with various forms of measurement uncertainty. The front-left face represents the scenarios, $\mathcal{S}$, under which data is collected, as well as the set of models that are hypothesized to describe the unknown truth, $\mathcal{R}$ [47, 48, 50, 52]. (Adapted from [49, 63, 64])

optimization [18] and *integer programming* [21] long after the original works of Fermat in the 17^{th} century.

Independently of the rapid developments in the field of mathematical programming, a new branch of science began to sprout in the late 1940s at Los Alamos National Laboratory (LANL), resulting, most notably, from the early works of Enrico Fermi [40, 61], Stanislaw Ulam, John Von Neumann, along with Edward Teller, Marshall Rosenbluth, and Nicholas Metropolis. In a series of articles [e.g., 41, 42, 74] they form the foundations of what becomes known in the following decades as *stochastic simulation* or *Monte Carlo methods*[1]. In their works, the authors propose several seminal methods for sampling strictly non-negative-valued generic mathematical functions in arbitrary dimensions, but mostly in the context of problems encountered in the field of Statistical Physics.

The proposed methodology of Metropolis et al [42] for sampling mathematical density functions was perhaps not fully appreciated by the scientific community until Hastings [26] presented a more generic formulation of the proposed sampling approach of [42], now known as the *Metropolis-Hastings Markov Chain Monte Carlo* (**MH-MCMC**) method. These two monumental articles, along with rapid technological breakthroughs in the world of computers have now enabled researchers to achieve all three aforementioned fundamental goals of the predictive science (*parameter-tuning*, *uncertainty-quantification*, and *model-selection*) in their research.

Optimization and Monte Carlo techniques have played a fundamental role in the emergence of the third pillar of science, *computational modeling* [52, 63], in the 1960s alongside the two original pillars of science: *observation* and *theory*. In particular, the MCMC techniques have become indispensable practical tools across all fields of science, from Astrophysics and Climate

[1] The technique's name 'Monte Carlo' was a suggestion made by Metropolis not so unrelated to Stan Ulam's uncle who used to borrow money from relatives because he "just had to go to Monte Carlo" for gambling [40].

Physics [e.g., 9, 34, 53, 62, 65, 66] to Bioinformatics and Biomedical Sciences [e.g., 30, 35, 36, 67] or Engineering fields [e.g., 51, 70].

Despite their popularity, the MCMC methods, in their original form as laid out by Hastings [26], have a significant drawback: The methods often require hand-tuning of several parameters within the sampling algorithms to ensure fast convergence of the resulting Markov chain to the target density for the particular problem at hand. Significant research has been done over the past decades, in particular during 1990' and 2000' to bring automation to the problem of tuning the free parameters of the MCMC methods. Among the most successful attempts is the algorithm of Haario et al [25], known as the **D**elayed-**R**ejection **A**daptive **M**etropolis MCMC (**DRAM**).

Several packages already provide implementations of variants of the proposed DRAM algorithm in Haario et al [25]. Peer-reviewed open-source examples include `FME` [69] in R, `PyMC` [54] in Python, `mcmcstat` [25] in MATLAB, `mcmcf90` [25] in Fortran, and QUESO [38, 56] in C/C++ programming languages. However, despite implementing the same algorithm (DRAM), these packages dramatically differ in their implementation approach, Application Programming Interface (API), computational efficiency, parallelization, and accessibility from a specific programming environment.

The ParaDRAM algorithm presented in this manuscript attempts to address the aforementioned heterogeneities and shortcomings in the existing implementations of the DRAM algorithm by providing a unified Application Programming Interface and environment for MCMC simulations accessible from multiple programming languages, including C/C++, Fortran, MATLAB, Python, R, with ongoing efforts to support other popular contemporary programming languages. The ParaDRAM algorithm is part of the open-source Monte Carlo simulation library with a codebase currently comprised of approximately 130,000 lines of code in mix of programming languages, including C, Fortran, MATLAB, Python, R, as well as Bash, Batch, and CMake scripting languages and build environments. The ParaDRAM package has been designed while bearing the following design philosophy and goals in mind,

1. **Full automation** of all Monte Carlo simulations to ensure the highest level of user-friendliness of the library and minimal time investment requirements for building, running, and post-processing of MCMC simulations.
2. **Interoperability** of the core library with as many programming languages as currently possible, including C/C++, Fortran, MATLAB, Python, R, with ongoing efforts to support other popular programming languages.
3. **High-Performance** meticulously-low-level implementation of the library to ensure the fastest-possible Monte Carlo simulations.
4. **Parallelizability** of all simulations via two-sided and one-sided MPI/Coarray communications while requiring zero-parallel-coding efforts by the user.
5. **Zero-dependence** on external libraries to ensure hassle-free ParaDRAM simulation builds and runs.
6. **Fully-deterministic reproducibility** and automatically-enabled restart functionality for all simulations up to 16 digits of precision if requested by the user.
7. **Comprehensive-reporting and post-processing** of each simulation and its results, as well as their automatic compact storage in external files to ensure the simulation results will be comprehensible and reproducible at any time in the distant future.

As implied by its name, a particular focus in the design of the ParaDRAM algorithm is to ensure seamlessly-scalable parallelizations of Monte Carlo simulations, form personal laptops to

supercomputers, while requiring absolutely no parallel-coding effort by the user. In the following sections, we will describe the design, implementation, and algorithmic details and capabilities of ParaDRAM. Toward this, we will devote §2, §3, and §4 on the mathematical explanation of the algorithm, including our proposed approach to dynamic monitoring of the diminishing-adaption condition of the DRAM algorithm (§4.1), as well as the parallelization paradigms used in the ParaDRAM algorithm (§4.2). Then, we describe the Application Programming interface of ParaDRAM in §5, including the implementation details of some of the unique features of the algorithm that enhances its computational and memory usage efficiency. Finally, we discuss the practical performance of the ParaDRAM algorithm in §6 and the road ahead for extending this package in §7.

2 The Metropolis-Hastings MCMC algorithm

The original proposed approach to sampling a mathematical objective density function, $f(x)$, by [43] and [26] is based on the brilliant observation that a special type of discrete-time stochastic processes, known as Markov process or Markov chain, can have stationary distributions under some conditions. Suppose a stochastic sequence of random vectors,

$$\{X_{\rm i}: \ i = 1, \ldots, +\infty\} \ , \tag{1}$$

is sampled from the d-dimensional domain of the objective function representing the state-space $\mathcal{X}$, such that it possesses a unique feature known as the Markov property. For notational simplicity, we assume that $\mathcal{X}$ is a finite set. The Markov property of this chain requires that, given the present state, the future random state of the sequence must be independent of the past states,

$$\pi(X_{\rm i+1} = x_{\rm i+1}|X_{\rm i} = x_{\rm i}, \ldots, X_1 = x_1,) = \pi(X_{\rm i+1} = x_{\rm i+1}|X_{\rm i} = x_{\rm i}) \ . \tag{2}$$

Here $\pi(\cdot)$ denotes the probability. The entire Markov process is characterized by an initial state as well as a square *transition matrix*, P, whose elements, $P_{\rm ij}$, describe the probabilities of transitions from each state i to every other possible state j in $\mathcal{X}$,

$$P_{\rm ij} = \pi(X_{\rm n+1} = j|X_{\rm n} = i) \ \ \forall \ n \in \{1, \ldots, +\infty\} \ . \tag{3}$$

If there is a unique transition matrix for the Markov process, then it is a *time-homogenous* Markov chain. Furthermore, if the Markov process is ergodic (i.e., aperiodic and Φ-irreducible, that is, capable of visiting every state in a finite time [58]) and, there is a distribution $f(x)$ such that every possible transition $x \to y$ in the process follows the principle of *detailed balance*,

$$f(x)\pi(y|x) = f(y)\pi(x|y) \ , \tag{4}$$

then, the process can be shown to have a unique *stationary distribution*, $f(x)$, to which it asymptotically approaches. The challenge, however, is to find a transition matrix, $\pi(y|x)$, that obeys the above conditions with respect to the target objective density function of interest, $f(x)$. The revolutionary insight due to [43] and [26] is that, one can define such generic transition matrix with respect to the desired $f(x)$, fulfilling the above conditions, if the transition matrix is split into two separate terms:

1. a proposal step, during which one proposes a new state y distributed according to $q(y|x)$, given the current state x,
2. followed by the acceptance or the rejection of the proposed step according to,

$$\alpha(x,y) = \min\left(1, \frac{f(y)q(x|y)}{f(x)q(y|x)}\right) , \tag{5}$$

such that the transition probability can be written as,

$$\pi(y|x) = q(y|x)\alpha(x,y) + \delta_{\rm x}(y)\sum_{z\in\mathcal{X}}\big(1-\alpha(x,z)\big)\ q(z|x) , \tag{6}$$

where $\delta_{\rm x}(y)$ is an indicator function, a discrete equivalent of the Dirac measure, such that $\delta_{\rm x}(y = x) = 1$. Defining the transition probability according to (6) is sufficient, though not necessary [e.g., 3], to guarantee the asymptotic convergence of the distribution of the resulting Markov chain to the target objective density function, $f(x)$ [7, 72].

In practice, however, convergence to the target density can be extremely slow if the proposal distribution, $q(y|x)$, is too different from $f(x)$ [59], as illustrated in Figure 2. Theoretical results [e.g., 19] indicate that an average acceptance rate of,

$$\langle\alpha\rangle = \frac{\text{the number of accepted MCMC proposed moves}}{\text{the total number of accepted and rejected proposed moves}} = 0.234 , \tag{7}$$

yields the fastest convergence rate in the case of an infinite-dimensional standard MultiVariate Normal (MVN) target density function, although numerical experiments indicate the validity of the results to hold for as low as 5 dimensions. In the absence of a generic universal rule for optimal sampling, a common practice has been to adapt the shape of $q(y|x)$ to the shape of $f(x)$ repeatedly and manually [2] such that the autocorrelation of the resulting MCMC chain is minimized, but this approach is cumbersome and difficult for large-scale simulation problems.

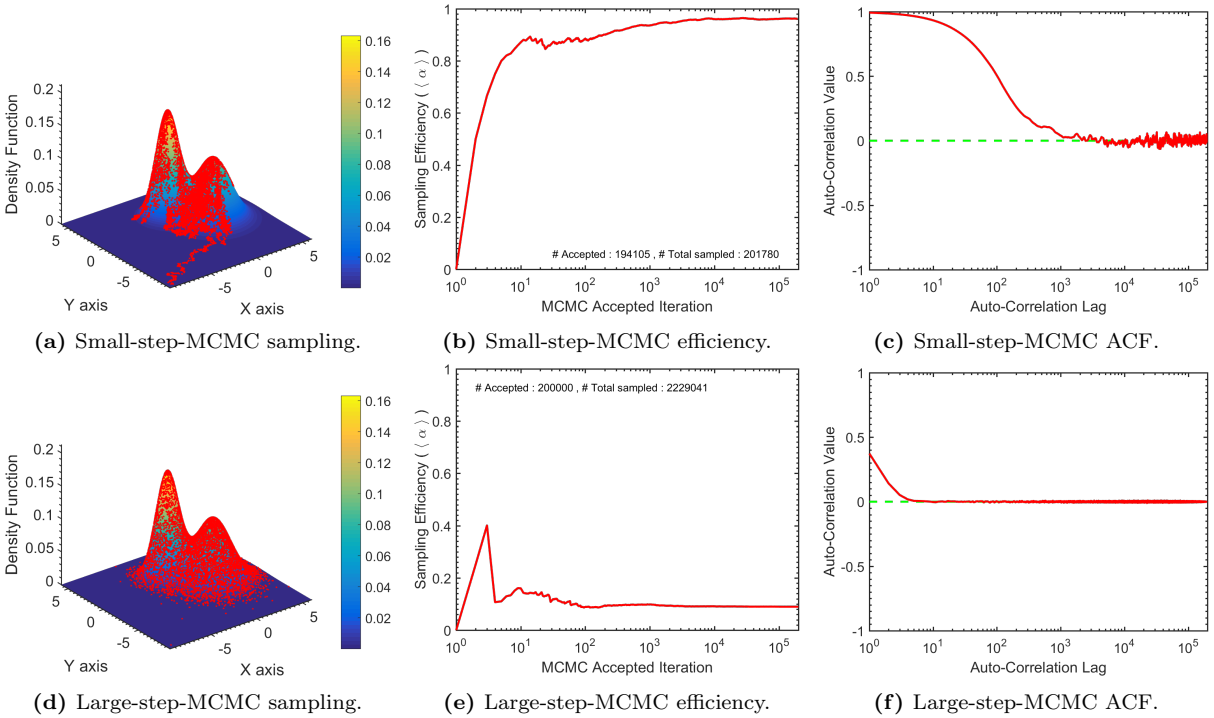
3 The DRAM algorithm

A second major insight, due to Haario et al [24], is that one can progressively and continuously adapt the shape of the proposal distribution based on the currently-sampled points in the entire history of the chain. Although the resulting chain is not Markovian because of the explicit dependence of every new sampled point on all of the past visited points, Haario et al [24] prove the asymptotic convergence of the resulting chain to the target density, $f(x)$. Roberts and Rosenthal [58] also provide more generic conditions under which the ergodicity of the resulting adaptive chain is maintained. The ergodicity property is one of the two pillars upon which the Metropolis-Hastings Markov Chain Monte Carlo is built.

Compared with the traditional MH-MCMC algorithm, the ith-stage acceptance probability of the Adaptive algorithm is modified to the following,

$$\alpha(x_{\rm i}, y) = \min\left(1, \frac{f(y)q_{\rm i}(x_{\rm i}|y, x_{\rm i-1}, \ldots, x_1)}{f(x)q_{\rm i}(y|x_{\rm i}, x_{\rm i-1}, \ldots, x_1)}\right) , \tag{8}$$

A major requirement for the validity of the Adaptive algorithm results is the condition of *diminishing adaptation*, requiring that the difference between the adjacent adapted Markov chain kernels approaches to zero in probability as the sample size grows to infinity. A practical



(a) Small-step-MCMC sampling. **(b)** Small-step-MCMC efficiency. **(c)** Small-step-MCMC ACF.

(d) Large-step-MCMC sampling. **(e)** Large-step-MCMC efficiency. **(f)** Large-step-MCMC ACF.

Fig. 2: An illustration of the importance of an appropriate choice of step size and proposal distribution shape in MCMC simulations. The plots **(a)**, **(b)**, and **(c)** display respectively the MCMC sample, the evolution of the efficiency of the MCMC simulation as defined by (7), and the autocorrelation function (ACF) of the chain of *uniquely-sampled* states via a proposal distribution whose scale is very small compared to the scale of the target density function. The plots **(d)**, **(e)**, and **(f)** represent the same quantities respectively as in the top plots, however, for very large-step-size proposed moves.

implementation of monitoring this condition in the ParaDRAM algorithm is discussed later in §4.1.

Haario et al [25] further combine the adaptive Metropolis algorithm of Haario et al [24] with the Delayed-Rejection (DR) algorithm of Green and Mira [22] to introduce the Delayed-Rejection Adaptive Metropolis (DRAM) algorithm. In brief, the DR algorithm modifies the standard MH-MCMC to improve the efficiency of the resulting estimators, with the basic idea being that, upon rejecting a proposed state according to (5), instead of advancing the chain and retaining the same position as it is done in the MH algorithm, a second-stage move is proposed given the knowledge of the newly-rejected proposed state, perhaps using an entirely different proposal.

The acceptance probability of the proposed state at each level of the DR process can computed such that the reversibility of the Markov chain relative to the target density $f(x)$ is preserved. Denote, respectively, the proposal Probability Density Function (PDF), the proposed state, and the corresponding acceptance probability at the zeroth stage of the delayed-rejection of the ith MCMC step of the DRAM algorithm by $g_0(y_0|x) = q_i(y_0|x)$, y_0, and $\beta_0(x, y_0)$. This zeroth stage is the regular Adaptive Metropolis (AM) algorithm step during which no delayed-rejection occurs. If y_0 is rejected, another proposal is made at the first level of DR via a potentially-new proposal with the PDF $g_1(\cdot)$. The corresponding acceptance probability is,

$$\beta_1(x, y_0, y_1) = \min\left(1, \frac{f(y_1)}{f(x)} \times \frac{g_0(y_0|y_1)g_1(x|y_0, y_1)}{g_0(y_0|x)g_1(y_1|y_0, x)} \times \frac{\big(1 - \beta_0(y_1, y_0)\big)}{\big(1 - \beta_0(x, y_0)\big)}\right), \tag{9}$$

Here, y_0 represents the original AM-proposed state, generated by the AM proposal kernel with the PDF $q(\cdot)$. Once y_0 is rejected, the first DR-proposed state y_1 is generated according to $g_1(\cdot)$, whose acceptance is computed according to (9).
The process of delaying the rejection can be continued for as long as desired at every step of the MH algorithm, with the higher-stage delayed-rejection proposals being allowed to depend on the candidates so far proposed and rejected. Since the entire process is designed to preserve the detailed-balance condition of (4), any acceptance at any stage of the DR process represents a valid MCMC sampling. However, continuing the DR process is feasible only at the cost of increasingly more complex equations for the acceptance probabilities at higher stages of the DR with lower-values. In general, the acceptance rate for the jth stage of the delayed rejection process is,

$$\begin{aligned}
&\beta_{\rm j}(x, y_0, y_1, \ldots, y_{\rm j}) = \\
&\min\left(1, \frac{f(y_{\rm j})}{f(x)} \times \frac{g_0(y_{\rm j-1}|y_{\rm j})g_1(y_{\rm j-2}|y_{\rm j-1}, y_{\rm j})\cdots g_{\rm j}(x|y_1, \ldots, y_{\rm j})}{g_0(y_0|x)g_1(y_1|y_0, x)\cdots g_{\rm i}(y_1|y_{\rm i}, \ldots, y_0, x)} \times \right. \\
&\left. \frac{\big(1-\beta_0(y_{\rm j}, y_{\rm j-1})\big)\big(1-\beta_1(y_{\rm j}, y_{\rm j-1}, y_{\rm j-2})\big)\cdots\big(1-\beta_{\rm j-1}(y_{\rm j}, y_{\rm j-1}, \ldots, y_1, y_0)\big)}{\big(1-\beta_0(x, y_0)\big)\big(1-\beta_1(x, y_0, y_1)\big)\cdots\big(1-\beta_{\rm j-1}(x, y_0, y_1, \ldots, y_{\rm j-1})\big)}\right) ,
\end{aligned} \tag{10}$$

With every new rejection during the DR process, the proposal distribution can be reshaped to explore more probable regions of the parameter space. Therefore, the DRAM algorithm [25] combines the global adaptation capabilities of the AM algorithm [24] with the local adaptation capabilities of the DR algorithm [22]. In the following section, we present a variant of the generic DRAM algorithm that has been implemented in the ParaDRAM routine of the ParaMonte library.

4 The ParaDRAM algorithm

The DRAM algorithm of §3 lays the foundation for the ParaDRAM algorithm. However, a major design goal of ParaDRAM is to provide a *fast* parallel Delayed-Rejection Adaptive Metropolis-Hastings Markov Chain Monte Carlo sampler. This is somewhat contrary to the complex generic form of the acceptance probability of the DRAM algorithm in (10) which can become computationally demanding for very high numbers of DR stages. Furthermore, a problem-specific adaptation strategy is generally required to successfully incorporate the knowledge acquired at each stage of the DR to construct the proposal for the next-stage DR. This is often a challenging task. We note, however, that (10) can be greatly simplified in the case of symmetric Metropolis proposals [44] where the probability of the proposed state only depends on the last rejected state,

$$g_{\rm j}(y_{\rm j}|y_{\rm j-1}, \ldots, y_0, x) = g_{\rm j}(y_{\rm j}|y_{\rm j-1}) = g_{\rm j}(y_{\rm j-1}|y_{\rm j}) \ . \tag{11}$$

In such case, the DR acceptance probability becomes,

$$\beta_{\rm j}(x, y_0, y_1, \ldots, y_{\rm j}) = \min\left(1, \frac{\max\big(0, f(y_{\rm j}) - f(y^*)\big)}{f(x) - f(y^*)}\right) \ , \quad y^* = \underset{0\le{\rm k}<{\rm j}}{\arg\max}\, f(y_{\rm k}), \ j > 0 \ . \tag{12}$$

This symmetric version of the DRAM algorithm provides a fair compromise between the computational efficiency and the variance-reduction benefits of delaying the rejection. Therefore, we

implement the symmetric Metropolis version of the DRAM algorithm in ParaDRAM and defer the implementation of the generic asymmetric form of DRAM to future work.

Despite the symmetry of the algorithm described above, we note that the proposal corresponding to each stage of the DR can still be completely independent of the proposals at other stages of the DR or the zeroth-stage (i.e., the Adaptive algorithm's) proposal distribution. Ideally, the proposal kernel at each stage should be constructed by incorporating the knowledge of the rejected states in the previous stages. In practice, however, such informed proposal construction is challenging.

By contrast, fixing the proposal distributions at all stages of the DR will be frequently detrimental to the performance of the sampler, since delaying the rejection often leads to steps that take the sampler into the vast highly-unlikely valleys and landscapes in the domain of the objective function that yield extremely small acceptance probabilities. In absence of any other relevant information about the structure of $f(x)$, a fair compromise can be made again by allowing the scale factor of the proposal distribution of each level of the DR to either shrink gradually from one stage to the next or be specified by the user before the simulation. The DR process can be then stopped either after a fixed number of stages (again, specified by the user) if all previous DR-stages have been unsuccessful, or by flipping a coin at each DR stage to continue or to stop and return to the AM algorithm. The former strategy is what we have implemented in ParaDRAM. The continuous process of adaptation of the proposal distribution as well as the DR process that is implemented in ParaDRAM is described in Algorithm 1.

4.1 Ensuring the diminishing-adaptation of the DRAM algorithm

In practice, the DRAM algorithm has to stop after a finite number of iterations, for example, as specified by N in Algorithm 1. Since the convergence and ergodicity of the chain generated by the DRAM algorithm is valid only asymptotically, it is crucial to monitor and ensure the asymptotic convergence of the chain to the target probability density function which, with a slight abuse of notations, we represent by $f(x)$. The convergence can be ensured by measuring the *total variation* between the target and the generated distribution from the nth-stage adapted proposal,

$$\lim_{\mathrm{n}\to+\infty} \|\pi_{\mathrm{n}}(\cdot|x) - f(\cdot)\| = 0 \ . \tag{13}$$

This is, however, impossible since the sole source of information about the shape of the target density is the generated chain. To resolve this problem of non-ergodicity of the finite chain due to the continuous adaptation, one conservative community approach has been to perform the adaptation for only a limited time. After a certain period, the adaptation fully stops and the regular MH-MCMC simulation begins with a fixed proposal. Consequently, the entire chain before fixing the proposal is thrown away and the final refined samples are generated only from the fixed-proposal MCMC chain.

This sampling approach, known as the *finite adaptation* [31, 58] is essentially identical to the traditional MH-MCMC approach except in the initial automatic fine-tuning of the proposal distribution, which is done manually in the traditional MH-MCMC algorithm. Although the finite adaptation approach ensures the ergodicity and the Markovian properties of the resulting chain, it suffers from the same class of limitations of the MH-MCMC algorithm. For example, it is not clear when the adaptation process should stop, and what should be the criterion used to automatically determine the stopping time of the adaptation. If the adaptation stops too early in

`Input :` The natural logarithm of the target objective density function, $f(x)$.
`Input :` An initial starting point, x_0, for the DRAM pseudo-Markov chain.
`Input :` An initial proposal distribution with PDF $q_0(\cdot)$.
`Input :` The desired number of states, N, to be sampled from $f(x)$.
`Input :` The maximum possible number of delayed-rejection stages at each MCMC step, M.
`Input :` The vector $S = \{s_1, \ldots, s_{\rm M}\}$ containing the scale factors of the DR proposal distributions.
`Output:` The pseudo-Markov chain $x_1, \ldots, x_{\rm N}$
Initialize
for *i = 1 to N* **do**

1. Propose a candidate state Y from the AM proposal distribution with probability $q_{\rm i-1}(Y = y|x_{\rm i-1}, \ldots, x_1, x_0)$.
2. Set $x_{\rm i} = y$ with probability,

$$\alpha(x_{\rm i-1}, y) = \min\left(1, \frac{f(y)}{f(x_{\rm i-1})}\right) ,$$

otherwise, set $g_0 = q_{\rm i-1}$, $y_0 = y$, $\beta_0(\cdot,\cdot) = \alpha_{\rm i-1}(\cdot,\cdot)$
for *j = 1 to M* **do**

(a) Construct the jth-stage delayed-rejection proposal distribution with PDF $g_{\rm j}(\cdot|\cdot)$, by rescaling the $(j-1)$th-stage DR proposal with PDF $g_{\rm j-1}(\cdot|\cdot)$ with the user-provided scale factor $s_{\rm j}$.
(b) Propose a new candidate $Y_{\rm j} = y_{\rm j}$ with probability $g_{\rm j}(y_{\rm j}|y_{\rm j-1})$.
(c) Set $x_{\rm i} = y_{\rm j}$ with probability,

$$\beta_{\rm j}(x, y_0, y_1, \ldots, y_{\rm j}) = \min\left(1, \frac{\max\left(0, f(y_{\rm j}) - f(y^*)\right)}{f(x) - f(y^*)}\right) ,$$

where,

$$y^* = \underset{0\le {\rm k}<{\rm j}}{\arg\max}\, f(y_{\rm k}),\ j > 0 \ .$$

and **break for**,
otherwise, **continue**

end
if $j > M$ **then**
$x_{\rm i} \leftarrow x_{\rm i-1}$,
continue (no candidate was accepted in the delayed rejection stages)

end

Algorithm 1: The symmetric-proposal DRAM algorithm as implemented in ParaDRAM

the simulation before good-mixing occurs, the resulting fixed-proposal MH-MCMC simulation can potentially suffer from the same slow-convergence issues encountered with the use of MH-MCMC algorithm, necessitating a restart of the adaptive phase of the simulation. This process of adaptation and verification will have to be then continued for as long as needed until the user can confidently fix the proposal distribution to generate the final Markovian chain.

Here we propose a workaround for this problem by noting that the entire adaptation in the DRAM algorithm is contained within the proposal distribution, $q(\cdot)$. Therefore, if we can somehow measure the amount of change between the subsequent adaptations of the proposal distribution, we can indirectly and dynamically assess the importance and the total effects of the adaptation on the chain that is being generated in real-time.

One of the strongest measures of the difference between two probability distributions is given by the metric *total variation distance (TVD)* between the two. For the two distributions, $Q_{\rm i}$ and $Q_{\rm i+1}$, defined over the d-dimensional space $\mathbb{R}^d$ with the corresponding densities, $q_{\rm i}$ and $q_{\rm i+1}$,

the TVD is defined as [73],

$$\text{TVD}(Q_{\text{i}}, Q_{\text{i+1}}) = \frac{1}{2} \int_{\mathbb{R}^{\text{d}}} \left| q_{\text{i}}(x) - q_{\text{i+1}}(x) \right| \, \text{d}x \ . \tag{14}$$

In other words, TVD is half of the L^1 distance between the two distributions. The TVD is by definition a real number between 0 and 1, with 0 indicating the identity of the two distributions and, 1 indicating two completely different distributions. Despite its simple definition, the computation of TVD is almost always intractable, effectively rendering it useless in our practical ParaDRAM algorithm.

To overcome the difficulties with the efficient computation of the TVD, we substitute the TVD with an upper bound on the its value. By definition, this upper bound always holds for any arbitrary pair of distributions and is defined via another metric distance between the two probability distributions, known as the Hellinger distance [28], whose square is defined as,

$$\text{H}^2(Q_{\text{i}}, Q_{\text{i+1}}) = 1 - \int_{\mathbb{R}^{\text{d}}} \sqrt{q_{\text{i}}(x) \ q_{\text{i+1}}(x)} \, \text{d}x \ . \tag{15}$$

By definition, the Hellinger distance is always bounded between 0 and 1, with 0 indicating the identity of the two distributions and, 1 indicating completely different distributions. A simple reorganization of the above equation reveals that the Hellinger distance is the L^2 distance between $\sqrt{q_{\text{i}}}$ and $\sqrt{q_{\text{i+1}}}$. Furthermore, it can be shown that the following inequalities hold between the Hellinger distance and the TVD [73],

$$\frac{1}{2}\text{H}^2(Q_{\text{i}}, Q_{\text{i+1}}) \le \text{TVD}(Q_{\text{i}}, Q_{\text{i+1}}) \le \text{H}(Q_{\text{i}}, Q_{\text{i+1}}) \sqrt{1 - \frac{\text{H}^2(Q_{\text{i}}, Q_{\text{i+1}})}{4}} \le \text{H}(Q_{\text{i}}, Q_{\text{i+1}}) \ . \tag{16}$$

Unlike TVD, the computation of the Hellinger distance is generally more tractable. In particular, the Hellinger distance has closed-form expression in the case of the MultiVariate Normal (MVN) distribution, which is, by far, the most widely-used proposal distribution in all variants of the MCMC method, including the DRAM algorithm. Even in cases where a closed-form expression for a proposal distribution may not exist, the TVD upper-bound computed under the assumption of an MVN proposal distribution might still provide an upper-bound for the TVD of the proposal distribution of interest, under some conditions.

Therefore, we use the inequalities expressed in (16) to estimate an upper bound for total variation distance between any two subsequent updates of the proposal distribution in the DRAM algorithm. This enables us to dynamically monitor and ensure the diminishing adaptation of the DRAM simulations. In practice, we find that the progressive amount of the adaptation of the proposal distribution diminishes fast as a power-law in terms of the MCMC steps. In cases of rapid good mixing, the initial few thousands of steps of the simulation exhibit significant adaptation of the proposal, followed by a fast power-law drop in the amount of adaptation. Some examples of the dynamic adaptation monitoring of the DRAM simulations are discussed in §6.

This practical method of dynamically measuring the adaptation also fulfills one of the major conditions for the ergodicity of the DRAM algorithm, for as long as the simulation can continue [e.g., see theorem (1) in 58]. It also provides an indirect qualitative method of rejecting the convergence to the target density if the TVD upper-bound estimate fails to continuously decrease or, even further increases with the progression of the DRAM simulation.

4.2 The parallelization of the DRAM algorithm

Contemporary scientific problems typically require parallelism to obtain solutions within a reasonable time-frame. As such, a major cornerstone of the ParaDRAM algorithm and the ParaMonte library is to enable seamless parallelization of Monte Carlo simulations without requiring any parallel programming experience from the user. Furthermore, to ensure the scalability of the ParaDRAM algorithm, from personal laptops to hundreds of cores on supercomputers, we have intentionally avoided the use of shared-memory parallelism in the algorithm, most notably, via the OpenMP [10] or OpenACC [14] standards. Nevertheless, this mode of parallelism remains a viable choice for future work.

Instead, the entire parallelization of the ParaDRAM algorithm is currently done via two independent scalable *distributed-memory* parallelism paradigms: 1. the *Message Passing Interface standard (MPI)* [23] and, 2. the *Partitioned global address space (PGAS)* [15, 46]. Unlike shared-memory parallelism, the distributed-memory architecture allows for scalable simulations beyond a single node of physical processors, across a network of hundreds or possibly, thousands of processors. This is an essential feature for parallel Monte Carlo algorithms in the era of ExaScale computing [5], although we will discuss some limitations of the current parallelism implementation of ParaDRAM in §7.

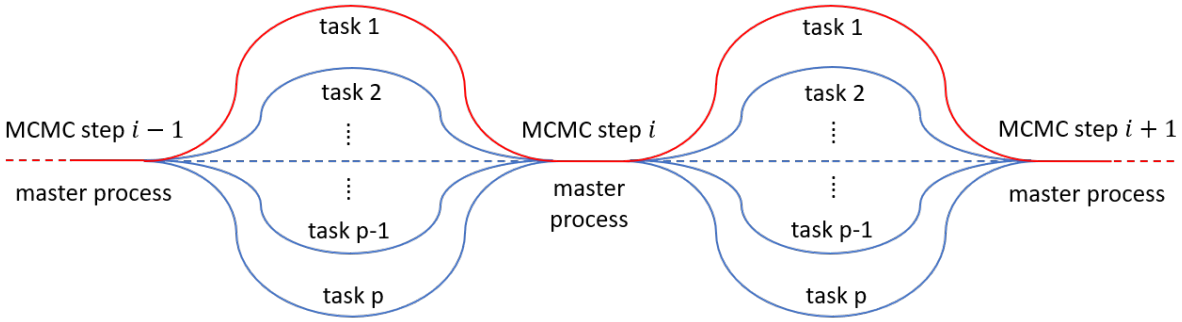
The PGAS parallelism paradigm readily enables Remote Memory Access (RMA), commonly known as one-sided communication, from one processor to all other processors in parallel simulations. This allows multiple data transfers between a set of processes to use a single synchronization operation, thus reducing the total overhead of inter-processor communications. By contrast, the RMA communications via MPI are considerably more challenging to implement. Nevertheless, the current support for the MPI parallelism paradigm is more robust than for the PGAS paradigm. Consequently, the utility of the PGAS parallelization of ParaDRAM currently remains limited to the Fortran language interface to the ParaDRAM algorithm, enabled by the Coarray Fortran [39]. Conversely, the MPI-parallelized version of ParaDRAM is accessible from all available programming language interfaces to ParaDRAM (e.g., C, C++, Fortran, MATLAB, Python, ...), even where the programming language does not officially support the MPI paradigm.

For both the MPI and PGAS communication paradigms in ParaDRAM, two parallelization models are currently implemented: 1. The Fork-Join parallelism [8] and, 2. The Perfect parallelism [45].

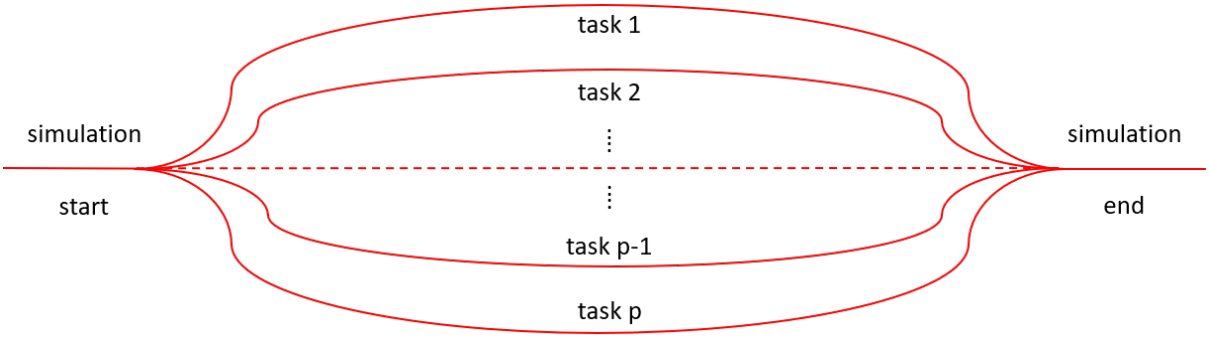
4.2.1 The Fork-Join parallelism

In this (`single-chain`) parallelism mode, the zeroth MPI process (or the first Coarray image) in the simulation is the master process responsible for reading the input data, updating the proposal distribution of the DRAM algorithm, concluding the simulation, and performing any subsequent post-processing of the simulation output data. All other processes in the simulation communicate and share information only with the master process/image 3a.

At each MCMC iteration, information about the current step is broadcasted by the master process to all other processes. Then, each process, including the master, proposes a new state for the chain and calls the user-provided objective function independently of the other processes. The proposed states together with the corresponding objective function values are then communicated to the master process. The master process then checks the occurrence of an accepted new state, proposal by any of the processes including itself. Upon the occurrence of the first



(a) The fork-join parallelism.



(b) The prefect parallelism.

Fig. 3: (a) An illustration of the fork-join parallelism implemented in the current version of the ParaDRAM algorithm. At each iteration of the MCMC simulation, a master process (represented by the red-line) distributes the current state of the sampler with all other processes (represented by the blue-lines). Then each process proposes a move which is either accepted or rejected and the result is returned to the master process for a final decision. **(b)** An illustration of the perfect parallelism implemented in the current version of the ParaDRAM algorithm. Each process runs an MCMC simulation independently of the rest of the processes. The communication cost is, therefore, zero in perfect parallelism.

acceptance, it communicates the new accepted state to all other processes and the next MCMC step begins. If no acceptance occurs, either the same old state is communicated to all processes to continue with the next MCMC step or, the simulation enters the Delayed-Rejection (DR) phase if requested by the user.

When the DR is enabled, the simulation workflow is similar to the above, except that upon rejecting the proposed moves by all processes at each stage of the DR, each process is allowed to only and only take one more delayed-rejection step. The results are then sent back to the master process to decide on whether the next DR stage has to be initiated. However, if an acceptance occurs on any of the processes at any given DR stage, the DR algorithm stops and the simulation returns to the regular adaptive algorithm. However, if no acceptance occurs after the maximum number of DR stages has reached, the last accepted state is communicated again to all processes, and the workflow of adaptive sampling repeats.

This mode of communication between the processes at each stage of the DR is crucial for the computational efficiency of the ParaDRAM algorithm since it is often significantly more costly to call the objective function redundantly (until the maximum number of DR stages is reached) than to communicate a few bytes of information between the processes at each DR stage to check if any acceptance has occurred. In §6, we will present some performance benchmarking results for the two MPI and PGAS fork-join parallelism implementations in ParaDRAM.

4.2.2 The Perfect parallelism

In the Perfect (`multi-chain`) parallelism mode, each MPI process (or Coarray image) runs independently of the other processes (images) to create its DRAM pseudo-Markov chain. This is effectively equivalent to having as many serial versions of the ParaDRAM algorithm to run concurrently and simulate independently of each other. However, unlike multiple independent concurrently-run serial DRAM chains, the ParaDRAM algorithm in the perfect-parallelism mode also performs post-simulation pairwise comparisons of the resulting chains from all processes to check for the convergence of all chains to the same unique target density.

To ensure the multi-chain convergence to the same target density, first, the initial burnin episodes of all chains are automatically removed and each chain is iteratively and aggressively refined (i.e., decorrelated) to obtain a final fully-decorrelated Independent and Identically Distributed (i.i.d.) sample from the target density, corresponding to each chain. Then, the similarities of the individual corresponding columns of all chains are compared with each other via the two-sample Kolmogorov-Smirnov (KS) nonparametric test [32, 68]. Finally, the results of the tests are reported in the output *report* file corresponding to each generated chain.

The perfect `multi-chain` parallelism, as explained in the above, has the benefit of providing an automatic convergence-checking, via the KS test, at the end of the simulation. This is a remarkable benefit that is missing in the fork-join `single-chain` parallelism. On the flip side, the perfect parallelism quickly becomes inferior to the fork-join paradigm for large-scale MCMC simulations, since the pressing issue in such cases is the computational and runtime efficiency of the simulation.

4.3 The final sample refinement

A unique feature of the Markov Chain Monte Carlo simulations is the Markovian property of the resulting chain, which states that the next step in the simulation given the currently visited state is independent of the chain's past. It may, therefore, sound counterintuitive to realize that the resulting sample from a finite-size MCMC simulation could be highly autocorrelated. Notably, however, each visited state in the chain depends on the last state visited before it. This implicit sequential dependence of all points on their past, up to the starting point, is what creates significant autocorrelations within MCMC samples.

For the infinite-length Markov chain of (1) that has converged to its stationary equilibrium distribution, the autocorrelation function is defined as,

$$\text{ACF}(k) = \frac{\mathbb{E}\big[(X_{\text{i}} - \mu)(X_{\text{i+k}} - \mu)\big]}{\sigma^2} \ , \tag{17}$$

where $(\mu, \text{ACF}(0) = \sigma^2)$ represent the mean and the standard deviation of the Markov chain and $\mathbb{E}[\cdot]$ represents the expectation operator. The *Integrated Autocorrelation (IAC)* of the chain is defined with respect to the variance of the estimator of the mean value μ,

$$\text{IAC} = 1 + 2\sum_{\text{k}=1}^{+\infty} \text{ACF}(k) \ , \tag{18}$$

such that,

$$\lim_{\text{n}\to+\infty} \sqrt{\frac{n}{\text{IAC}}} \frac{\mu_{\text{n}} - \mu}{\sigma} \Rightarrow N(0,1) \ , \tag{19}$$

where μ_n represents the sample mean of the chain of length n and '$\Rightarrow$' stands for convergence in distribution. The value of IAC roughly indicates the number of Markov transitions required to obtain an i.i.d. sample from the target distribution of the Markov chain. In practice, one wishes to obtain a finite MCMC sample whose size is at least of the order of the integrated autocorrelation of the chain [57]. This is a challenging goal that is often out of reach since the IAC of the Markov chain is not known a priori. A more accessible approach is to generate a chain for a given predefined length, then de-correlate it to obtain a final *refined* independent and identically distributed (i.i.d.) sample from the target density.

The numerical computation of the IAC, however, poses another challenge to decorrelating MCMC samples since the variance of its estimator in (18) diverges to infinity. A wide variety of techniques have been proposed that aim to estimate the IAC for the purpose of MCMC sample refinement. Among the most popular methods are the Batch Means (BM) [16], the Overlapping Batch Means (OBM) [60], the spectrum fit method [27], the initial positive sequence estimator [20], as well as the auto-regressive processes [e.g., 55].

Thompson [71] performs a series of tests aimed at identifying the fastest and the most accurate method of estimating the IAC. They find that while the auto-regressive process appears to be the most accurate method of estimating the IAC, the Batch Means method provides a fair balance between the computational efficiency and numerical accuracy of the estimate.

Based on the findings of Thompson [71], we have therefore implemented the Batch Means method as the default method of estimating the IAC of the resulting Markov chains from the ParaDRAM sampler. Notably, however, all the aforementioned methods appear to underestimate the value of IAC, in particular, for small chain sizes. Therefore, we have adopted a default aggressive methodology in the ParaDRAM algorithm where the autocorrelation of the chain is removed repeatedly (via any estimator of choice by the ParaDRAM user, such as the BM) until the final repeatedly-refined chain does not exhibit any autocorrelation.

This aggressive refinement of the chain is performed in two separate stages: At the first stage, the full Markov chain is repeatedly refined based on the estimated IAC values from the (non-Markovian) *compact* chain of the uniquely accepted points. This stage essentially removes any autocorrelation in the Markov chain that is due to the choice of too-small step sizes for the proposal distribution (Figure 2a). Once the compact chain of accepted points is devoid of any autocorrelations, the second phase of the Markov chain refinement begins, with the IAC values now being computed from the (*verbose*) Markov chain, starting with the resulting refined Markov chain from the first stage of the refinement (of the compact chain).

We have found by numerous experimentations that the above approach often leads to final refined MCMC samples that are fully decorrelated while not being refined too much due to our aggressive repetitive decorrelation of the full Markov chain. Nevertheless, the above complex methodology for the refinement of the Markov chain can be entirely controlled by the input specifications of the simulation set by the user. For example, the user can request only one round of chain refinement to be performed using only one of format of the chain: compact or verbose (Markov).

5 One API for ParaDRAM across all programming languages

Special care has been made to ensure that the Application Programming Interface (API) of the ParaDRAM algorithm retains highly-similar (if not the same) structure, naming, and calling conventions across all programming languages currently supported by the ParaMonte library.

First and foremost, the interface to the ParaDRAM routine requires only two mandatory pieces of information to be provided by the user:

1. `ndim`: the dimension of the domain of the objective function to be sampled and,
2. `getLogFunc(ndim,point)`: a computational implementation of the objection function in the programming language of choice, which should take as input a 32-bit integer `ndim` and a 64-bit real vector `point` of length `ndim` that represents a state from within the domain of the objective function. On return, the function yields the natural logarithm of the value of the objective function evaluated at `point`. Unlike C/C++/Fortran, in the case of higher-level programming languages such as MATLAB or Python, the calling syntax of the objective function simplifies to `getLogFunc(point)` where the length of `point` is passed implicitly.

The one-API paradigm has been one of the core design philosophies of the ParaMonte library (including the ParaDRAM algorithm) to ensure similar user experience and the availability of the same functionalities from all supported programming language interfaces to the ParaMonte / ParaDRAM library. The full description of all capabilities and details of each of the programming-language interfaces to the ParaDRAM routine goes well beyond the limitations of the current manuscript. Therefore, we will only present some of key identical components of the algorithm shared among all available interfaces to the ParaDRAM sampler.

5.1 The ParaDRAM simulation specifications

The ParaDRAM sampler has been mindfully developed to be as flexible as possible regarding the settings of the simulations. Consequently, there is a long list of input simulation specifications whose complete descriptions go beyond the limits of this paper. We refer the interested reader to permanent repository[2] and the documentation website[3] of the ParaMonte library for the detailed descriptions of the simulation specifications.

Despite the great number and variety of the ParaDRAM simulation specifications, all 39 independent input specification variables currently available in ParaDRAM are optional and automatically set if not provided by the user. In some simulation scenarios, some level of input information may be necessary from the user, for example, when the domain of the objective function does not extend to infinity, in which case, the user can readily specify the boundaries of a cube within which the sampling will be performed.

From within the compiled programming languages, the preferred method of specifying simulation parameters is to store them all within an input file and provide the path to this file at the time of calling the ParaDRAM routine. This approach enables changes to the simulation configuration seamlessly possible without any need to recompile and relink the source codes to build a new executable, which can be a time-consuming process for large-scale simulations. From within all compiled languages (C/C++/Fortran), the simulation specifications can be also passed as a string to the ParaDRAM sampler, instead of passing the path to an external file. In such case, the value of the string could be the contents of the input file (instead of the path to the input file). From within the Fortran language, the users can also pass the simulation specifications as `optional` input arguments to the ParaDRAM sampler.

From within the interpreted programming languages such as MATLAB and Python, the preferred method of specifying the simulation configuration is via the dedicated Object-Oriented Programming (OOP) interface that we have developed in each of these programming language

[2] `https://github.com/cdslaborg/paramonte`

[3] `https://www.cdslab.org/paramonte/`

environments. Nevertheless, the users can also alternatively provide the same input file that is used in compiled languages to configure their ParaDRAM simulations. Given the great flexibility of the interpreted languages, specifying the simulation configuration via an input file seems to be inferior to the OOP interface that we have written for each of these programming language environments.

5.2 The ParaDRAM simulation output files

Each ParaDRAM simulation, performed from within any programming language environment, generates five distinct output files. If the user has specified a simulation name then all output files *prefixed* are prefixed by the user-provided simulation name. Otherwise, the output files are prefixed by a unique automatically-generated random simulation name with a specific pattern, for example: `"./out/ParaDRAM_run_20200101_205458_278_process_1"`, where,

1. `./out/` is the example user-requested directory within which the output simulation files are stored (and if the specified directory does not exist, it is automatically generated),
2. `ParaDRAM` indicates the type of the simulation,
3. `run_yyyymmdd_hhmmss_mmm` indicates the date of the simulation specified by the current year (`yyyy`), month (`mm`), and day (`dd`), followed by the start time of the simulation specified by the hour (`hh`), the minute (`mm`), the second (`mm`), and the millisecond (`mmm`) of the moment of the start of the simulation,
4. `process_1` indicates the ID of the processor that has generated the output files, with `1` indicating the master process (or Coarray image).

The above random prefix-naming convention both documents the exact date and time of the simulation and ensures the uniqueness of the names of the generated output files. In the extremely-rare event of a user-specified filename clash with an existing set of simulation files in the same directory, the simulation will be aborted and the user will be asked to specify a unique name for the new simulation.
Once the uniqueness of the prefix of the simulation output files is ensured, the ParaDRAM sampler generates five separate output files with the same prefix, but with the following suffixes that imply the purpose and the type of the contents of each file,

1. `_chain.txt` or `_chain.bin` indicates a file containing the ParaDRAM output Markov Chain, where the user's choice of the format of the file (`ASCII` vs. `binary`) dictates the file extension (`txt` vs. `bin`).
2. `_sample.txt` indicates a file containing the final refined decorrelated sample from the target density, containing only the refined set of visited states and their corresponding target density values reported in natural logarithm.
3. `_report.txt`: indicates a file containing a comprehensive report of all aspects of the simulation, including the ParaDRAM library version, the computing platform, the user-specified description of the simulation, the user-specified (or automatically-determined) simulation configuration, the description of the individual simulation specifications, any runtime simulation warnings or fatal errors, as well as extensive report on the timing and performance of serial/parallel simulation and extensive postprocessing of the simulation results.
4. `_progress.txt`: indicates a file containing a dynamic report of the simulation progress, such as the dynamic efficiency of the MCMC sampler, time spent since the beginning of the simulation and the predicted time remained to accomplish the simulation.

5. `_restart.txt` or `_restart.bin`: indicates a file containing information required for a deterministic restart of the simulation, should the simulation end prematurely. The user's choice of the format of the file (`ASCII` vs. `binary` dictates the file's extension (`txt` vs. `bin`).

5.3 Efficient compact storage of the Markov Chain

The restart functionality and the ability to handle large-scale simulations that exceed the random-access-memory (RAM) limits of the processor require the ParaDRAM algorithm to continuously store the resulting chain of sampled points throughout the simulation. However, this poses two major challenges to the high efficiency and low memory-footprint of the algorithm:

1. Given the current computational technologies, input/output (IO) from/to external files is on average 2-4 orders of magnitude slower than the RAM IO. This creates a severe bottleneck in the speed of the otherwise high-performing ParaDRAM algorithm, in particular, for large-scale high-dimensional objective functions.
2. Moreover, the resulting output chain files can easily grow to several Gigabytes, even for regular MCMC simulations, making the storage of multiple simulation output files over the long term challenging or impossible.

To minimize the effects of external IO on the performance and the memory-footprint of the algorithm, we propose to store the resulting chain of states from ParaDRAM in a very compact format that dramatically enhances the library's performance and lowers its memory footprint 5-10 times, without compromising the fully-deterministic restart functionality of ParaDRAM or its ability to handle large-scale memory-demanding simulations.
The *compact* (as opposed to *verbose* or Markov) storage of the chain is made possible by noting that the majority of states in a typical Markov chain are identical as a result of the repeated rejections during the sampling. The lower the acceptance probability is, the larger the fraction of repeated states in the verbose Markov chain will be. Therefore, the storage requirements of the chain can be dramatically reduced by keeping track of only the accepted states and assigning weights to them based on the number of times they are sequentially repeated in the Markov chain.
Furthermore, since the contents of the output chain file is peripheral to the contents of the final refined sample file, the ParaDRAM sampler also provides a `binary` output mode, where the chain will be written out in binary format. Although the resulting output chain file is unreadable by human, the binary IO is fast, does not suffer from loss of precision due to the conversion from binary to decimal for external IO, and in general, occupies less memory for same level of accuracy. Nevertheless, we believe the above proposed `compact` chain file format provides a good compromise between, IO speed, memory-footprint, and readability. Therefore, we use the compact ASCII file as the default format of the output chains from ParaDRAM simulations. Users can also specify a third `verbose` format where the resulting Markov chain will be written to the output file *as is*. However, this `verbose` mode of chain IO is not recommended except for debugging or exploration purposes since it significantly degrades the algorithm's performance and increases the memory requirements for the output files.

5.4 The ParaDRAM simulation restart functionality

An integral part of the ParaDRAM algorithm is its automatically-enabled fully-deterministic reproducibility of the simulation results, should a ParaDRAM simulation, whether serial or

parallel end prematurely. In such cases, all the user needs to do in order to restart the simulation from where it was interrupted, is to rerun the simulation (with the same output file prefix).
The ParaDRAM algorithm has been designed to automatically detect the existence of the output simulation files. If all the simulation files already exist, the simulation will be aborted with a message asking the user to provide a unique file-prefix name for the output simulation files. However, if all files exist except the output refined sample file, which is generated in the last stage of the simulation, ParaDRAM enters the restart mode and begins the simulation from where it was interrupted during the last run.
A remarkable feature of the restart functionality of the ParaDRAM algorithm is its fully-deterministic reproduction of the simulation results *into the future*: If a simulation is interrupted and subsequently restarted, the resulting final chain after the restart would be identical, up to 16 digits of precision, to the chain that the sampler would have generated if the simulation had not been interrupted in the first place. To generate a fully deterministic reproducible simulation, all that is needed from the user is to set the seed of the random number generator of the ParaDRAM sampler as part of the input simulation specifications. Additionally, in the case of parallel simulations, it is expected that the same number of processes will be used to run the restart simulation as used for the original interrupted simulation.
The information required for the restart of an interrupted ParaDRAM simulation is automatically written to the output restart file. To minimize the impacts of the restart IO on the performance and the external the memory requirements of the algorithm, the restart file is automatically written in binary format. This default behavior can be overridden by the user by requesting an `ASCII` restart file format in the input simulation specifications to the sampler. In such case, ParaDRAM will also automatically add additional relevant information about the dynamics of the proposal adaptations to the output file. This human-readable information can be then parsed to gain insight into the inner-workings of the ParaDRAM algorithm. An example of such analysis of the dynamics of the proposal adaptation will be later given in §6.

5.5 The optimal number of processors for parallel ParaDRAM simulations

When a parallel MCMC simulation is performed in the fork-join (`single-chain`) parallelism mode, the ParaDRAM algorithm, as part of a default post-processing analysis, attempts to predict the *optimal* number of parallel processors from which the simulation could benefit. In general, the overall parallel efficiency of a software depends on a number of factors including (but not limited to),

1. $T_{\rm s}$: The serial runtime required for all computations that cannot be parallelized and must be performed in serial mode.
2. $T_{\rm p}$: The serial runtime for the fraction of the code that can be equally shared among all processors in parallel.
3. $T_{\rm o}$: The time required for setting up the inter-processor communications, and information exchange, effectively known as the *communication overhead*.

Among the three time measures mentioned in the above, the overhead time is the most complex and hardest to estimate since it is highly software and hardware dependent. Nevertheless, this overhead time can be frequently assumed to linearly grow with the number of processes in the parallel simulation. This is particularly true for the fork-join parallelism paradigm where all inter-process communications happen to and from a master process. Therefore, the overall speedup due to the use of N_p processors in parallel can be computed from a modified form

of the Amdahl's law of strong scaling [1] that takes into account the communication overhead time,

$$S(N_p) \approx \frac{T_\mathrm{s} + T_\mathrm{p}\ (\text{The total serial run time of the simulation})}{T_\mathrm{s} + \frac{T_\mathrm{p}}{N_p} + (N_p - 1) \times T_\mathrm{o}} \ , \tag{20}$$

In the case of the ParaDRAM routine, the runtime of the serial fraction (T_s) is typically on the order of a few tens of nanoseconds to microseconds on the modern architecture, whereas the parallel fraction of the simulation (T_p) – which calls the user-provided objective function – is expected to dominate the simulation runtime. Therefore, compared to T_p and T_o, the serial fraction (T_s) can be safely ignored in large-scale ParaDRAM simulations. Then, to compute the speedup in any given parallel ParaDRAM simulation, T_p and T_o can be respectively estimated from the average runtimes of the parallel and the inter-process communication sections of the code.

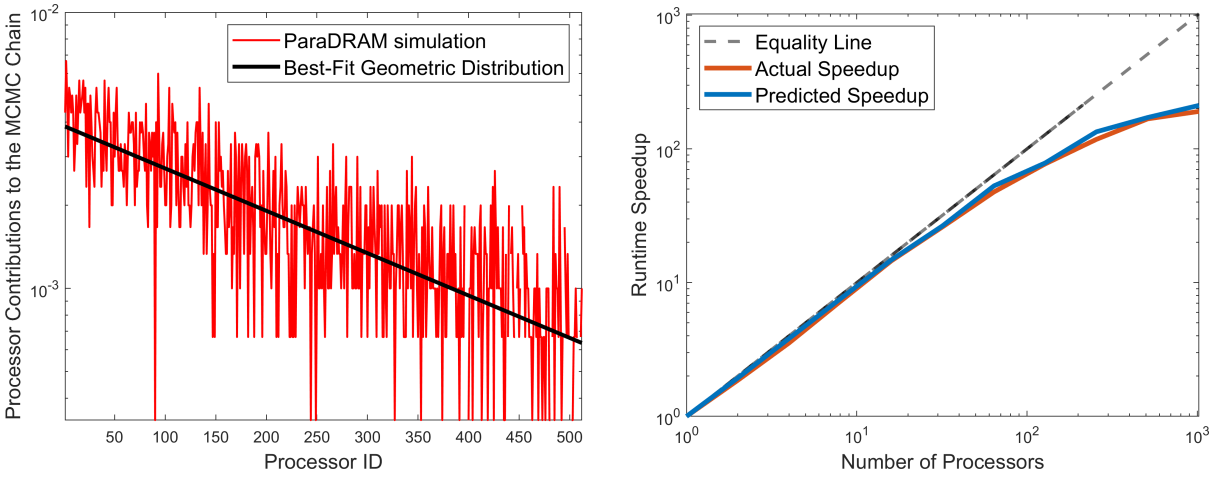
Once T_p and T_o are estimated, we can then predict the simulation speedup over a wide range of number of processors. The maximum predicted speedup then provides an *absolute* upper bound on the number of processors that could benefit the simulation. In practice, however, this *absolute optimal* number of processors is only an upper bound on the actual number of processors from which the given simulation would effectively benefit. In the special case of parallel fork-join MCMC simulations, there is yet another equally-important factor that, along with the communication overhead, limits the overall speedup of parallel simulations. This non-negligible factor is the efficiency of the MCMC sampler.

The role of the average MCMC acceptance rate on the optimal number of processors can be understood by noting that the average number of MCMC steps that need to be taken before an acceptance occurs is roughly proportional to the inverse of the average MCMC acceptance rate. For example, if the average acceptance rate is 0.25, then one would expect an acceptance to occur every four steps. This places a fundamental limit on the number of processors from which the simulation could benefit in parallel.

Quantitatively, the process of accepting a proposed state in a given step of the ParaDRAM algorithm, parallelized over an infinite number of processors ($N_p \to +\infty$), can be modeled as a Bernoulli trial with two possible outcomes: rejection or acceptance of the proposed state. In this process, the probability of an acceptance can be assumed to be represented by the average MCMC acceptance rate ($\overline{\alpha}$). Thus, the probability of an acceptance occurring after k proposals (by the first k processors) follows a Geometric distribution, $\mathcal{G}(\cdot)$, and is given by,

$$\pi_{\mathcal{G}}\big(\text{acceptance} \mid k\big) = \overline{\alpha}\ (1 - \overline{\alpha})^{(k-1)} \ . \tag{21}$$

Practically, however, the workload at each MCMC step is always shared among a finite number of processors which we denote by N_p. In such case, the total fractional contribution (C_i) of the ith processor (out of N_p processors) to the construction of the entire ParaDRAM compact chain is the sum of the probabilities of the occurrences of all acceptances due to the ith processor in the simulation,



(a) Processor contributions to a parallel simulation. **(b)** The parallel strong-scaling: predicted vs. actual.

Fig. 4: (a) An illustration of the contributions of 512 Intel Xeon Phi 7250 processors to a ParaDRAM simulation in parallel (the red curve). The predicted best-fit Geometric distribution from the post-processing phase of the ParaDRAM simulation is shown by the black line. **(b)** A comparison of the parallel-performance of ParaDRAM simulations on a range of processor counts with the performance predictions from the post-processing output of the ParaDRAM sampler. The entire performance data depicted in the plots (a) and (b) of this figure are automatically generated by the ParaDRAM sampler as part of the post-processing of every parallel MCMC simulation.

$$
\begin{aligned}
C_{\mathrm{i}} &\equiv \pi(\text{acceptance} \mid i, N_p) && (22)\\
&= \sum_{\mathrm{j}=0}^{+\infty} \pi_{\mathcal{G}}(\text{acceptance} \mid k = j \times N_p + i) && (23)\\
&= \overline{\alpha} \sum_{\mathrm{j}=0}^{+\infty} (1-\overline{\alpha})^{(j\times N_p+i-1)} && (24)\\
&= \overline{\alpha}\ (1-\overline{\alpha})^{(i-1)} \sum_{\mathrm{j}=1}^{+\infty} \left[(1-\overline{\alpha})^{N_p}\right]^{(j-1)} && (25)\\
&= \frac{\overline{\alpha}\ (1-\overline{\alpha})^{(i-1)}}{1-(1-\overline{\alpha})^{N_p}} \left(1 - \left[(1-\overline{\alpha})^{N_p}\right]^{j\to+\infty}\right) && (26)\\
&= \frac{\overline{\alpha}\ (1-\overline{\alpha})^{(i-1)}}{1-(1-\overline{\alpha})^{N_p}} \quad , \quad i = 1, \ldots, N_p\ , && (27)
\end{aligned}
$$

where (26) and (27) are derived from the cumulative distribution function of the Geometric distribution. Since the occurrence of an acceptance is checked in order from the first (master) process to the last, the first processor has, on average, always the highest contribution to the construction of the MCMC chain, followed by the rest of the processors in order, as implied by 27 and illustrated in Figure 4a. This means that the overall scaling behavior of a parallel ParaDRAM simulation solely depends on the contribution (C_1) of the first processor to the construction of the MCMC chain. The contribution C_1 is in turn determined by the average acceptance rate of the simulation as in (27).

For example, if the MCMC sampling efficiency is 100%, then the entire MCMC output is constructed by the contributions of the first processor. By contrast, the lower the sampling efficiency is, the more evenly the simulation workload will be shared among all processors.

Quantitatively, the maximum speedup for a given N_p number of processors and an average $\overline{\alpha}$ MCMC sampling efficiency can be written as,

$$S(N_p) \approx \frac{T_\mathrm{s} + T_\mathrm{p}}{T_\mathrm{s} + C_1(\overline{\alpha}) \times T_\mathrm{p} + (N_p - 1) \times T_\mathrm{o}} \ , \tag{28}$$

Frequently though, the average acceptance rate (α) of an MCMC simulation is a wildly-varying dynamic quantity during the simulation. Therefore, instead of using the estimated average MCMC sampling efficiency from the simulation, we infer an effective MCMC sampling efficiency by fitting the Geometric distribution of (27) to the contributions of the individual processors to the output chain. In practice, we find that this effective sampling efficiency is frequently slightly larger than the average MCMC sampling efficiency defined as the ratio of the number of accepted states to the full length of the generated (pseudo)-Markov Chain. Figure 4b compares the simulation speedup predicted in the post-processing section of the ParaDRAM algorithm with the actual simulation speedup, for a range of processor counts.

6 Example Results

A wide range of mathematical test objective functions exist with which the performance of the ParaDRAM algorithm can be benchmarked. The presentation of all examples goes beyond the scope of this manuscript. For illustration purposes, here we present the results for a popular multi-modal example test objective function known as the Himmelblau's function [29]. This function is frequently used in testing the performance of optimization algorithms and is defined as,

$$f_\mathrm{H}(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \ , \tag{29}$$

with one local maximum at,

$$f_\mathrm{H}(-0.271, -0.923) \approx 181.617 \ , \tag{30}$$

and four identical local minima located at,

$$f_\mathrm{H}(3.0, 2.0) \approx f_\mathrm{H}(-2.805, 3.131) \approx f_\mathrm{H}(-3.779, -3.283) \approx f_\mathrm{H}(3.584, -1.848) \approx 0.0 \ . \tag{31}$$

However, just as with any type of MCMC sampler, the ParaDRAM algorithm explores the maxima of objective functions as opposed to the minima. Therefore, we modify the original Himmelblau's function of (29) by adding a small value of 0.1 to the function and inverting the entire new function, such that all four minima become maxima and remain well-defined the logarithm of the function is computed and passed to the ParaDRAM algorithm. This simulation can be performed in any of the programming language environments that are currently supported by the ParaMonte library. For brevity, here we suffice to presenting only the simulation codes and results generated in the MATLAB scientific computing language.

A very simple implementation of this simulation in MATLAB is provided below,

```
getLogFunc = @(x) -log( (x(1)^2 + x(2) - 11)^2 + (x(1) + x(2)^2 - 7)^2 + 0.1 );
pm = paramonte();
pmpd = pm.ParaDRAM();
pmpd.runSampler(2, getLogFunc);
```

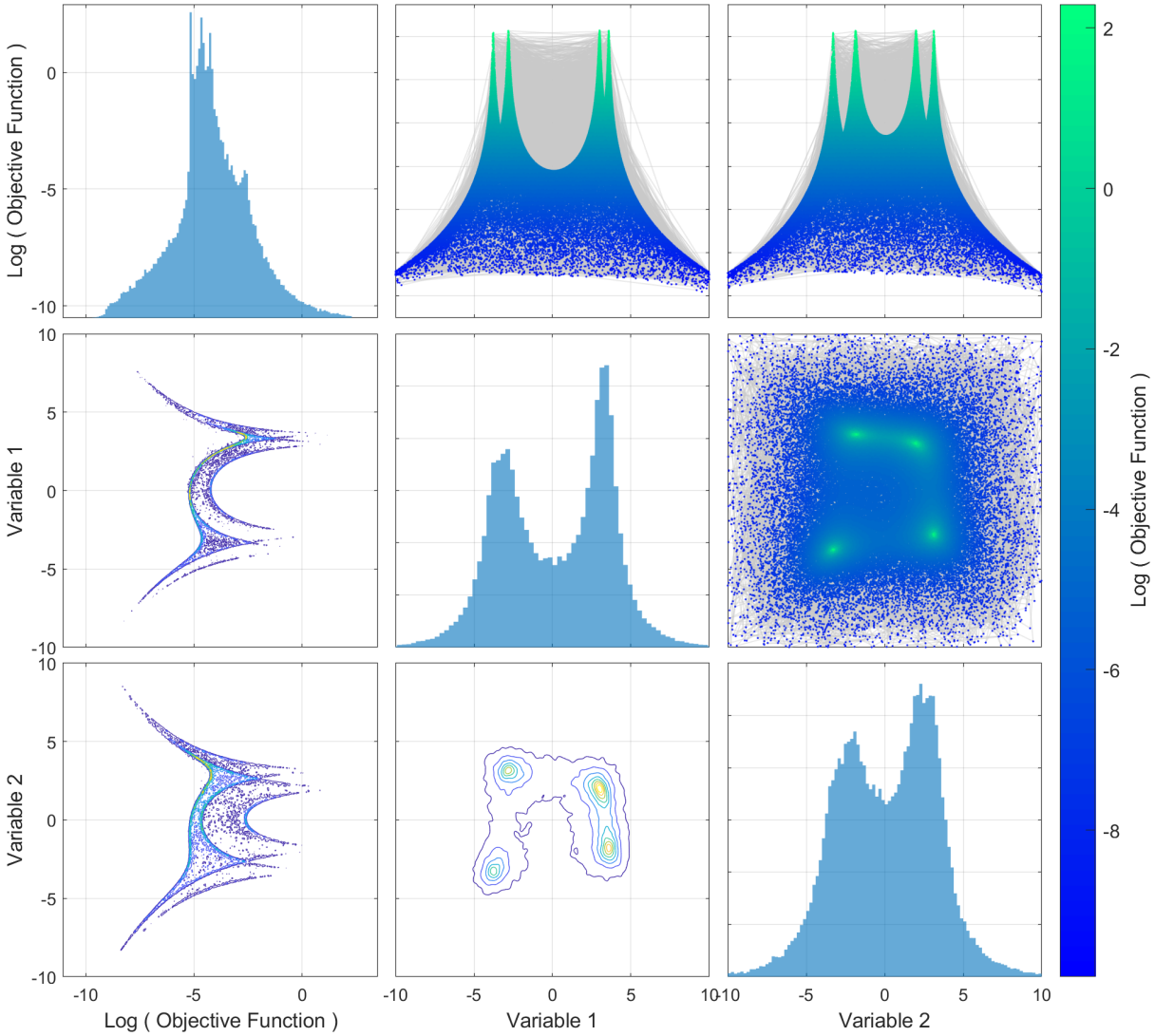
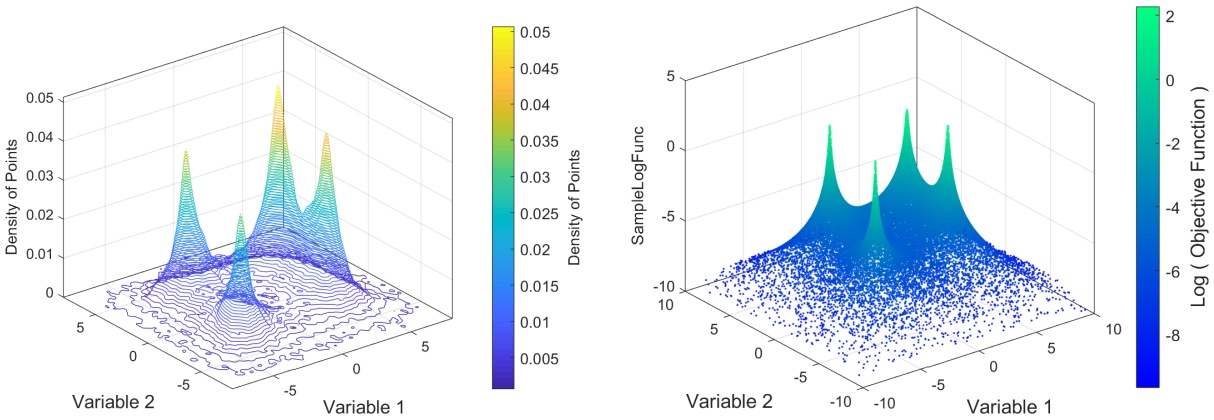


Fig. 5: An illustration of the ParaDRAM simulation output for the problem of sampling Himmelblau's function. The figure data consists of pairs of the Himmelblau's function value and its two input variables, plotted against each other. Only the uniquely-visited states in the domain of Himmelblau's function are shown in the plots. The lower triangle of the plot represents the density contour maps of the sampled points, whereas the upper triangle contains line-scatter plots of pairs of variables, color-coded by the natural logarithm of Himmelblau's function. The mono-color gray lines connect the sequence of points in the chain together. The diagonal plots represent the distributions of the uniquely-visited states within the domain of Himmelblau's function.

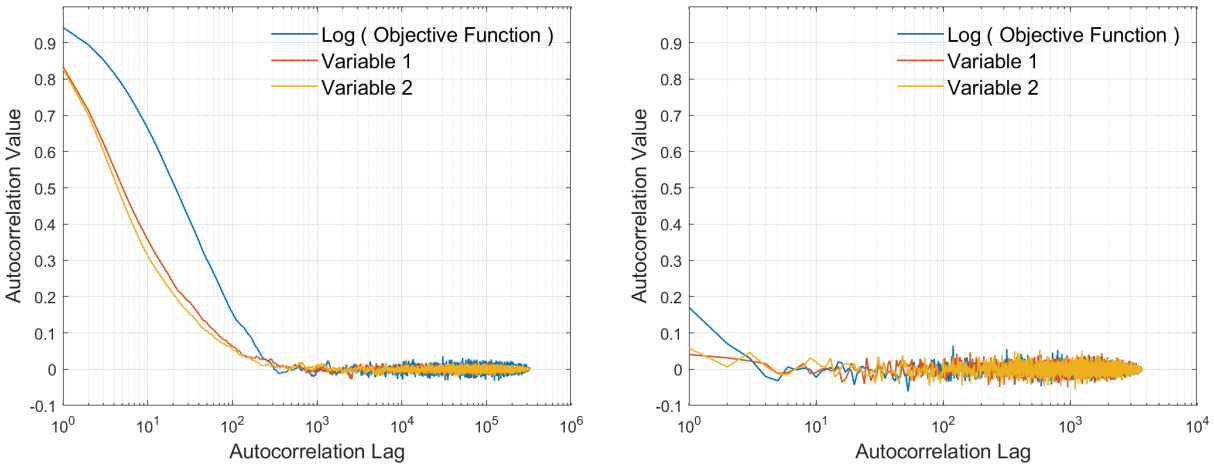
The above minimal code defines the natural logarithm of the 2-dimensional Himmelblau's function as a MATLAB anonymous (Lambda) function named `getLogFunc`, then generates an instance of the `paramonte` class named `pm`, from which an instance of the `ParaDRAM` class is derived and named `pmpd`. Then the `runSampler()` method of the `ParaDRAM` class is called to sample the objective function represented by `getLogFunc`. All simulation specifications for this sampling problem that are not predefined by the user, will be automatically set to appropriate default values by the ParaDRAM algorithm. Once the simulation is finished, the post-processing tools that are shipped with the ParaMonte-MATLAB library can seamlessly parse, analyze, and visualize the output of the simulation.

Figure 5 illustrates a grid-plot of the uniquely-visited points by the ParaDRAM sampler. A better visualization of the density of the uniquely-visited states within the domain of Himmel-



(a) The 3D density contour map. **(b)** The 3D scatter plot.

Fig. 6: (a) An 3D contour map of the ParaDRAM simulation output for the problem of sampling Himmelblau's function. The figure data consists of the density map of the set of all uniquely-visited states by the ParaDRAM sampler within the domain of the objective function. **(b)** A 3D scatter plot of the set of uniquely-visisted states by the ParaDRAM sampler within the domain of Himmelblau's function. All plots are generated via the visualization tools that automatically ship with ParaMonte-MATLAB library.



(a) The MCMC autocorrelation. **(b)** The refined-sample's autocorrelation.

Fig. 7: (a) An illustration of the autocorrelation in the individual variables of the output MCMC chain in the simulation of Himmelblau's function. **(b)** An illustration of the residual autocorrelation in the individual variables of the final output refined sample in the simulation of Himmelblau's function. By default, the ParaDRAM algorithm performs an aggressive and recursive series of MCMC refinements aimed at removing any traces of autocorrelation in the final refined output sample by the ParaDRAM sampler.

blau's function is given in Figure 6a. The ParaDRAM visualization toolbox uses the linear-diffusion-process kernel density estimation method of Botev et al [6] to generate the 2D and 3D contour plots. A 3-dimensional visualization of the structure of Himmelblau's function using the uniquely-visited sates by the ParaDRAM algorithm is given in Figure 6b.

As mentioned in §4.3, the ParaDRAM algorithm performs an aggressive series of sample refinements on the generated Markov chains such that no residual autocorrelation remains in the final output sample. Figure 7 compares the amount of autocorrelation in the original output MCMC chain from the ParaDRAM sampler with the residual autocorrelation in the final refined sample. By default, the ParaDRAM algorithm repeatedly and aggressively refines the output MCMC chain, for as long as needed, such that no traces of autocorrelations remain in the resulting final sample by the algorithm.

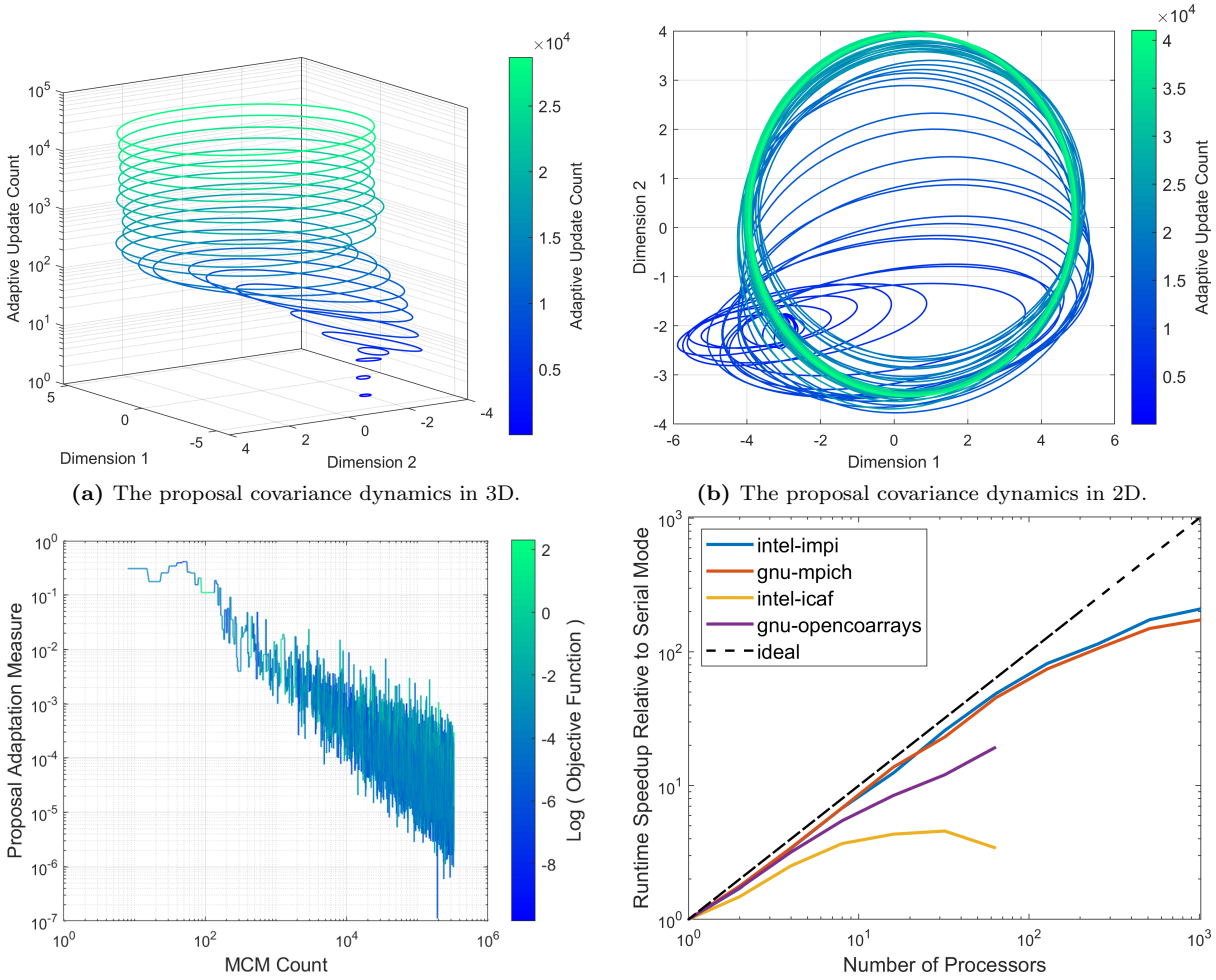
6.1 Monitoring the dynamic adaptation of the proposal distribution of the ParaDRAM sampler

In §4.1 we argued for necessity of ensuring and monitoring the diminishing adaptation condition of the adaptive algorithms, including the ParaDRAM sampler. Therein, we offered a solution for the dynamic monitoring of the changes in the proposal distribution via an adaptation measure whose value is limited to the range $[0, 1]$. Figures 8a and 8b display the dynamic evolution of the proposal distribution of the ParaDRAM routine for the problem of sampling Himmelblau's function. The continuous adaption of the proposal is visualized by the changes in the covariance matrix of the proposal distribution throughout the entire simulation. It is evident from the plots that the proposal adaptation eventually diminishes as desired.
The amount of adaptation in the proposal distribution can be further quantified by the adaptation measure defined in §4.1. Figure 8c illustrates the monotonically-decreasing adaptation of the bivariate Normal proposal distribution used in the sampling of Himmelblau's function via the ParaDRAM sampler. As part of the output chain file, the ParaDRAM algorithm continuously outputs and monitors the diminishing adaptation condition of the DRAM algorithm to ensure the asymptotic ergodicity and the Markovian properties of the resulting output chain. The power-law decay of the adaptation-measure seen in Figure 8c is exactly the kind of diminishing adaptation condition one would hope to see in ParaDRAM simulations.

6.2 Performance benchmarking of the MPI and PGAS parallelism paradigms

Given the multiple different parallelism paradigms currently implemented in the ParaDRAM algorithm, it may be of interest to user of the library to know which parallelism paradigm and/or perhaps what compilers or parallism library implementations yield the best simulation performances. Figure 8d, illustrates the performance benchmarking of the ParaDRAM algorithm for an example 4-dimensional multivariate Normal target density function. Given the simplicity of such sampling problem, the time-cost of calling this objective function was artificially increased so that a more accurate and clear comparison could be made between the strong-scaling results for the MPI and PGAS parallelism paradigms.
We obtained and compared the results for the MPI and PGAS parallelism methods using compilers from two different vendors: the Intel and the GNU compiler suites. In the case of the MPI parallelism, the Intel MPI and the MPICH MPI libraries were used respectively. In the case of the PGAS parallelism, the Intel Coarray and the OpenCoarrays [15, 46] libraries were used respectively. Based on the benchmarking results presented in Figure 8d, the PGAS parallelism as implemented in ParaDRAM performs inferior to the MPI implementation. There are potentially two reasons for such performance difference between the two parallelism paradigms in ParaDRAM,

1. The current version of the ParaDRAM algorithm does not fully exploit the unique readily-available RMA-communication features of the Coarray libraries. This is partly due to the lack of support for the advanced RMA features in the Coarray libraries when the ParaMonte library was originally developed. Recently, however, many new advanced RMA communication features of the Coarray parallelism paradigm have been implemented by multiple open-source and commercial compilers, including the GNU, Intel, and NAG compiler suites. Therefore, we anticipate that a future re-implementation of the PGAS Coarray parallelism in ParaDRAM via the newly-available RMA features will resolve some of the discrepancies



(a) The proposal covariance dynamics in 3D.

(b) The proposal covariance dynamics in 2D.

(c) The diminishing adaptation criterion.

(d) Example strong scaling results: MPI vs. PGAS.

Fig. 8: **(a)** An illustration of the 3-dimensional dynamic adaptation of the covariance matrix of the bivariate Normal proposal distribution of the ParaDRAM sampler for the problem of sampling Himmelblaus's function. **(b)** An illustration of the 2-dimensional dynamic adaptation of the covariance matrix of the bivariate Normal proposal distribution of the ParaDRAM sampler for the problem of sampling Himmelblaus's function. **(c)** An illustration of the diminishing adaptation of the proposal distribution of the ParaDRAM sampler for the problem of sampling Himmelblaus's function. As explained in §4.1, the monotonically-decreasing adaptivity observed in the plot ensures the asymptotic ergodicity and Markovian property of the resulting output ParaDRAM chain. **(d)** An illustration of the strong-scaling results for parallel ParaDRAM simulations using the two different parallelization paradigms implemented in ParaDRAM: 1. the Message Passing Interface (MPI) via the Intel MPI (impi) and MPICH libraries and, 2. the Partitioned Global Address Space (PGAS) via Intel Coarray Fortran (icaf) and OpenCoarrays library. See §7 for an explanation of the performance differences between the strong-scaling results of the PGAS- and MPI- parallelized ParaDRAM simulations.

observed between the performances of the MPI and PGAS parallelization of the ParaDRAM sampler.

2. The currently-available MPI libraries are highly optimized and mature, while the Coarray PGAS libraries have only recently become available.

7 Discussion

Over the past 3 decades, the popularity and the utilities of Monte Carlo simulations has grown exponentially in a wide range of scientific disciplines. In particular, the Markov Chain Monte

Carlo (MCMC) techniques have become indispensable tools for predictive computing and uncertainty quantification. In this work, we presented the ParaDRAM algorithm, a high-performance implementation of the Delayed-Rejection Adaptive Metropolis-Hastings (DRAM) algorithm of Haario et al [25]. The DRAM algorithm is one of the most popular and most successful adaptive MCMC techniques available in the literature that has proven to dramatically outperform the traditional MCMC sampling techniques.

The presented ParaDRAM algorithm is part of the ParaMonte open-source Monte Carlo simulation library, available at `https://github.com/cdslaborg/paramonte`. The library is currently comprised of approximately 130,000 lines of codes primarily in written the C, Fortran, MATLAB, Python, as well as the Bash, Batch, Cmake scripting and build languages. The majors goals in the development of the ParadRAM algorithm have been to bring simplicity, full-automation, comprehensive reporting, and automatic fully-deterministic restart functionality to the inherently stochastic Monte Carlo simulations. In addition, we have careful to design a unified Application Programming Interface (API) to a wide range of popular programming languages in the scientific community, such that the syntax of calling and setting up the ParaDRAM sampler remains almost the same across all programming language environments. Notably, we aimed to achieve the aforementioned goals without compromising the high-performance and the parallel scalability of the algorithm.

To ensure the scalability of parallel ParaDRAM simulations, from personal laptops to the world-class supercomputers, we have adopted and implemented the MPI and PGAS distributed-memory parallelism paradigms in this library. Remarkably, we have been careful to not require any parallel-coding effort or experience from the user in order to build and run parallel ParaDRAM simulations, from any programming language environment.

To maintain the high-performance of the library, we also described in this manuscript an efficient compact storage method for the output MCMC chains from the ParaDRAM simulations. This approach as detailed in §5.3 enables us to maximize the library's IO performance and minimize the external and internal memory-footprints of the library, without any compromise in the automatic fully-deterministic restart functionality feature of the ParaDRAM algorithm in parallel or in serial simulations. We also discussed in §4.1, a novel technique to automatically and dynamically monitor and ensure the diminishing adaptation criterion of the DRAM algorithm.

The ParaDRAM sampler library is currently being actively developed and expanded with new sampling capabilities. Further planned enhancements include but are not limited to: 1. increasing the accessibility of the ParaDRAM library from other popular programming languages. There are currently ongoing efforts to include support for the Java, Julia, Mathematica, and R programming languages. 2. minimizing the effects of improper processor load-balance on the overall performance of the parallel simulations. The recent enhancements and additions to the RMA communication facilities within Coarray-PGAS parallelism paradigms will be a great aid toward achieving this goal.

Acknowledgements

We thank the Texas Advanced Computing Center (TACC) for providing the parallel computing resources for the development and testing of the ParaMonte/ParaDRAM library presented in this manuscript.

References

1. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference, pp 483–485
2. Andrieu C, Moulines É, et al (2006) On the ergodicity properties of some adaptive mcmc algorithms. The Annals of Applied Probability 16(3):1462–1505
3. Barker AA (1965) Monte carlo calculations of the radial distribution functions for a proton? electron plasma. Australian Journal of Physics 18(2):119–134
4. Beale EM (1955) On minimizing a convex function subject to linear inequalities. Journal of the Royal Statistical Society Series B (Methodological) pp 173–184
5. Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J, et al (2008) Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech Rep 15
6. Botev ZI, Grotowski JF, Kroese DP, et al (2010) Kernel density estimation via diffusion. The annals of Statistics 38(5):2916–2957
7. Chib S, Greenberg E (1995) Understanding the metropolis-hastings algorithm. The american statistician 49(4):327–335
8. Conway ME (1963) A multiprocessor system design. In: Proceedings of the November 12-14, 1963, fall joint computer conference, pp 139–146
9. Curcic M (2019) A parallel fortran framework for neural networks and deep learning. In: ACM SIGPLAN Fortran Forum, ACM New York, NY, USA, vol 38, pp 4–21
10. Dagum L, Menon R (1998) Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering 5(1):46–55
11. Dantzig GB (1955) Linear programming under uncertainty. Management science 1(3-4):197–206
12. Dantzig GB (1998) Linear programming and extensions. Princeton university press
13. Du DZ, Pardalos PM, Wu W (2013) Mathematical theory of optimization, vol 56. Springer Science & Business Media
14. Enterprise C (2011) Cray inc. and nvidia and the portland group: The openacc application programming interface, v1. 0 (november 2011)
15. Fanfarillo A, Burnus T, Cardellini V, Filippone S, Nagle D, Rouson D (2014) Opencoarrays: open-source transport layers supporting coarray fortran compilers. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp 1–11
16. Fishman GS (1978) Principles of discrete event simulation.[book review]
17. Floudas CA, Pardalos PM (2001) Encyclopedia of optimization. Springer Science & Business Media
18. Ford Jr LR, Fulkerson DR (1955) A simple algorithm for finding maximal network flows and an application to the hitchcock problem. Tech. rep., DTIC Document
19. Gelman A, Roberts GO, Gilks WR, et al (1996) Efficient metropolis jumping rules. Bayesian statistics 5(599-608):42
20. Geyer CJ (1992) Practical markov chain monte carlo. Statistical science pp 473–483
21. Gomory RE (1963) An algorithm for integer solutions to linear programs. Recent advances in mathematical programming 64:260–302
22. Green PJ, Mira A (2001) Delayed rejection in reversible jump metropolis–hastings. Biometrika 88(4):1035–1053
23. Gropp W, Lusk E, Doss N, Skjellum A (1996) A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing 22(6):789–828
24. Haario H, Saksman E, Tamminen J, et al (2001) An adaptive metropolis algorithm. Bernoulli 7(2):223–242

25. Haario H, Laine M, Mira A, Saksman E (2006) Dram: efficient adaptive mcmc. Statistics and computing 16(4):339–354
26. Hastings WK (1970) Monte carlo sampling methods using markov chains and their applications. Biometrika 57(1):97–109
27. Heidelberger P, Welch PD (1981) A spectral method for confidence interval generation and run length control in simulations. Communications of the ACM 24(4):233–245
28. Hellinger E (1909) Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. Journal für die reine und angewandte Mathematik (Crelles Journal) 1909(136):210–271
29. Himmelblau DM (1972) Applied nonlinear programming [by] David M. Himmelblau. Mcgraw-Hill
30. Jha PK, Cao L, Oden JT (2020) Bayesian-based predictions of covid-19 evolution in texas using multispecies mixture-theoretic continuum models. Computational Mechanics pp 1–14
31. Kass RE, Carlin BP, Gelman A, Neal RM (1998) Markov chain monte carlo in practice: a roundtable discussion. The American Statistician 52(2):93–100
32. Kolmogorov-Smirnov A, Kolmogorov A, Kolmogorov M (1933) Sulla determinazione empírica di uma legge di distribuzione
33. KUHN H, AW T (1951) Nonlinear programming. In: 2nd Berkeley Symposium. Berkeley, University of California Press, pp 481–492
34. Li G, Curcic M, Iskandarani M, Chen SS, Knio OM (2019) Uncertainty propagation in coupled atmosphere–wave–ocean prediction system: a study of hurricane earl (2010). Monthly Weather Review 147(1):221–245
35. Lima E, Oden J, Wohlmuth B, Shahmoradi A, Hormuth D, Yankeelov T, Scarabosio L, Horger T (2017) Selection and validation of predictive models of radiation effects on tumor growth based on noninvasive imaging data. Computer Methods in Applied Mechanics and Engineering
36. Lima E, Oden J, Wohlmuth B, Shahmoradi A, Hormuth II D, Yankeelov T (2017) Ices report 17-14
37. Mahoney MS (1994) The mathematical career of Pierre de Fermat, 1601-1665. Princeton University Press
38. McDougall D, Malaya N, Moser RD (2015) The parallel c++ statistical library for bayesian inference: Queso. arXiv preprint arXiv:150700398
39. Metcalf M, Reid J, Cohen M (2018) Modern Fortran Explained: Incorporating Fortran 2018. Oxford University Press
40. Metropolis N (1987) The beginning of the monte carlo method
41. Metropolis N, Ulam S (1949) The monte carlo method. Journal of the American statistical association 44(247):335–341
42. Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953) Equation of state calculations by fast computing machines. The journal of chemical physics 21(6):1087–1092
43. Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953) Equation of state calculations by fast computing machines. The Journal of Chemical Physics 21(6):1087–1092
44. Mira A, et al (2001) On metropolis-hastings algorithms with delayed rejection. Metron 59(3-4):231–241
45. Moler C (1986) Matrix computation on distributed memory multiprocessors. Hypercube Multiprocessors 86(181-195):31
46. Numrich RW, Reid J (1998) Co-array fortran for parallel programming. In: ACM Sigplan Fortran Forum, ACM New York, NY, USA, vol 17, pp 1–31
47. Oden J, Babuska I, Faghihi D (2004) Predictive computational science: Computer predictions in the presence of uncertainty. Encyclopedia of Computational Mechanics, E Stein, R de Borst, and TJR Hughes, eds, Wiley, Hoboken, NJ
48. Oden JT (2017) Foundations of predictive computational sciences. ICES Reports
49. Oden JT (2018) Adaptive multiscale predictive modelling. Acta Numerica 27:353–450

50. Oden JT, Prudencio EE, Hawkins-Daarud A (2013) Selection and assessment of phenomenological models of tumor growth. Mathematical Models and Methods in Applied Sciences 23(07):1309–1338
51. Oden JT, Babuška I, Faghihi D (2017) Predictive computational science: Computer predictions in the presence of uncertainty. Encyclopedia of Computational Mechanics Second Edition pp 1–26
52. Oden T, Moser R, Ghattas O (2010) Computer predictions with quantified uncertainty, part i. SIAM News 43(9):1–3
53. Osborne JA, Shahmoradi A, Nemiroff RJ (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, 2006.01157
54. Patil A, Huard D, Fonnesbeck CJ (2010) Pymc: Bayesian stochastic modelling in python. Journal of statistical software 35(4):1
55. Plummer M, Best N, Cowles K, Vines K (2006) Coda: convergence diagnosis and output analysis for mcmc. R news 6(1):7–11
56. Prudencio E, Schulz K (2012) The parallel C++ statistical library queso: Quantification of uncertainty for estimation, simulation and optimization. In: Alexander M, D'Ambra P, Belloum A, Bosilca G, Cannataro M, Danelutto M, Martino B, Gerndt M, Jeannot E, Namyst R, Roman J, Scott S, Traff J, Valle G, Weidendorfer J (eds) Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science, vol 7155, Springer Berlin Heidelberg, pp 398–407
57. Robert CP, Casella G, Casella G (2010) Introducing monte carlo methods with r, vol 18. Springer
58. Roberts GO, Rosenthal JS (2007) Coupling and ergodicity of adaptive markov chain monte carlo algorithms. Journal of applied probability 44(2):458–475
59. Rosenthal JS, et al (2011) Optimal proposal distributions and adaptive mcmc. Handbook of Markov Chain Monte Carlo 4(10.1201)
60. Schmeiser B (1982) Batch size effects in the analysis of simulation output. Operations Research 30(3):556–568
61. Segre E (1955) Fermi and neutron physics. Reviews of Modern Physics 27(3):257
62. Shahmoradi A (2013) A multivariate fit luminosity function and world model for long gamma-ray bursts. The Astrophysical Journal 766(2):111
63. Shahmoradi A (2017) Multilevel bayesian parameter estimation in the presence of model inadequacy and data uncertainty. arXiv preprint arXiv:171110599
64. Shahmoradi A (2017) Multilevel Bayesian Parameter Estimation in the Presence of Model Inadequacy and Data Uncertainty. arXiv e-prints arXiv:1711.10599, 1711.10599
65. Shahmoradi A, Nemiroff RJ (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
66. Shahmoradi A, Nemiroff RJ (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, 1903.06989
67. Shahmoradi A, Sydykova DK, Spielman SJ, Jackson EL, Dawson ET, Meyer AG, Wilke CO (2014) Predicting evolutionary site variability from structure in viral proteins: buriedness, packing, flexibility, and design. Journal of molecular evolution 79(3-4):130–142
68. Smirnov N (1948) Table for estimating the goodness of fit of empirical distributions. The annals of mathematical statistics 19(2):279–281
69. Soetaert K, Petzoldt T, et al (2010) Inverse modelling, sensitivity and monte carlo analysis in r using package fme. Journal of Statistical Software 33(3):1–28
70. Taghizadeh L, Karimi A, Stadlbauer B, Weninger WJ, Kaniusas E, Heitzinger C (2020) Bayesian inversion for electrical-impedance tomography in medical imaging using the nonlinear poisson–boltzmann equation. Computer Methods in Applied Mechanics and Engineering 365:112,959

71. Thompson MB (2010) A comparison of methods for computing autocorrelation time. arXiv preprint arXiv:10110175
72. Tierney L (1998) A note on metropolis-hastings kernels for general state spaces. Annals of applied probability pp 1–9
73. Tsybakov AB (2008) Introduction to nonparametric estimation. Springer Science & Business Media
74. Von Neumann J (1951) 13. various techniques used in connection with random digits