ELSEVIER

Korali: Efficient and scalable software framework for Bayesian uncertainty quantification and stochastic optimization

Sergio M. Martin[a], Daniel Wälchli[a], Georgios Arampatzis[a], Athena E. Economides[a], Petr Karnakov[a], Petros Koumoutsakos[a,b,*]

[a] *Computational Science and Engineering Laboratory, ETH Zürich, CH-8092, Switzerland*
[b] *Institute of Applied Computational Sciences, Harvard University, Cambridge, MA, USA*

Abstract

We present Korali, an open-source framework for large-scale Bayesian uncertainty quantification and stochastic optimization. The framework relies on non-intrusive sampling of complex multiphysics models and enables their exploitation for optimization and decision-making. In addition, its distributed sampling engine makes efficient use of massively-parallel architectures while introducing novel fault tolerance and load balancing mechanisms. We demonstrate these features by interfacing Korali with existing high-performance software such as APHROS, LAMMPS (CPU-based), and MIRHEO (GPU-based) and show efficient scaling for up to 512 nodes of the CSCS Piz Daint supercomputer. Finally, we present benchmarks demonstrating that Korali outperforms related state-of-the-art software frameworks.

1. Introduction

Over the last thirty years, High-Performance Computing (HPC) architectures have enabled high-resolution simulations of physical systems ranging from atoms to galaxies. HPC has also reduced the cost and turnaround time of such simulations, making them invaluable predictive and design tools across all fields of science and engineering. Multiple simulations at resolutions that would have been impossible a decade ago are routinely employed in optimization and design. The transformation of these simulations into actionable decisions requires the quantification of their uncertainties. In recent years, HPC has become central in the way that we conduct science with massive amounts of data. Such data are used to develop and calibrate physical models as well as to quantify the uncertainties of their predictions. The integration of data and physical models has a history of over 300 years, dating back to Laplace and Copernicus and to the framework known as Bayesian inference. However, due to its computational cost, the application of Bayesian inference has been, until recently, limited to simple models or through inexpensive approximations.

* Corresponding author at: Institute of Applied Computational Sciences, Harvard University, Cambridge, MA, USA.
  E-mail address: petros@seas.harvard.edu (P. Koumoutsakos).

Bayesian inference requires sampling of distributions with dimensionality greater than or equal to the number of model parameters. The sampling necessitates numerous evaluations, making the process computationally demanding, particularly when the underlying model requires hundreds of compute hours per evaluation. Moreover, special care is necessary to develop sampling algorithms that harness the capabilities of modern supercomputers [1]. The sampling involved in Bayesian inference serves as a bridge to stochastic optimization algorithms [2] that aim to identify the probability distribution of the parameters that maximize a particular cost function. Stochastic optimization algorithms such as CMA-ES, MBOA and Natural Gradient Optimization [3,4] are non-intrusive, thus they operate without knowledge or modification of the physical models, through input/output relations.

The need for efficient deployment of optimization and uncertainty quantification algorithms has motivated to the development of several statistical frameworks [5–11]. However, to the best of our knowledge, only a few such frameworks are well-suited for deployment in massively parallel computer architectures [12,13]. In this paper, we present Korali, a new framework for Bayesian uncertainty quantification (UQ) and optimization. The framework enables efficient large-scale sampling while providing mechanisms for fault-tolerance, load balancing, and reproducibility, which are essential requirements for emerging supercomputers [14]. We demonstrate these capabilities guided by three motivating studies. First, a high-dimensional optimization study of the shape of fluid transporting pipes. Second, a hierarchical Bayesian analysis of the dissipation parameter of human RBC membranes. Lastly, a Bayesian study on the parameters of a coarse-grained (CG) water model. In addition, we provide a synthetic benchmark that compares the performance of Korali with that of other state-of-the-art frameworks for optimization on up to 512 nodes of the CSCS Piz Daint supercomputer.

The rest of this paper is organized as follows: in Section 2, we present the principles behind the unifying framework; in Section 3, we present the framework's design; in Section 4, we present the results of three experimental cases; in Section 5, we discuss state of the art UQ frameworks and compare their efficiency, and; in Section 6, we present conclusions and future work.

2. Unified approach to optimization and sampling

We designed Korali to exploit the common patterns in Bayesian optimization and sampling while exposing a unifying interface. Consider a computational model, represented by a function f, that depends on parameters $\boldsymbol{\vartheta}$ with unknown values and possibly other parameters $\boldsymbol{x}$ with known values. We wish to infer the values for the parameters $\boldsymbol{\vartheta}$ such that the model evaluated at known parameters $\boldsymbol{x}_i$ will approximate given values y_i for $i = 1, \ldots, N$. The variables y_i are called the *data* and usually correspond to experimental measurements. Since the measurements are typically affected by random errors, a probabilistic model links the measurements y_i with the model evaluations $f(\boldsymbol{\vartheta}, \boldsymbol{x}_i)$. This model is represented by a known probability density function $p(\boldsymbol{y}|\boldsymbol{\vartheta}; \vec{x})$ where $\boldsymbol{y} = \{y_1, \ldots, y_N\}$ and $\vec{x} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and summarizes our assumptions about the origin of the data. For fixed $\vec{x}$ and $\boldsymbol{y}$ the density is a function of the parameters $\boldsymbol{\vartheta}$ and is called the *likelihood* function. Furthermore, any information on the parameters $\boldsymbol{\vartheta}$ that is known prior to observing any data is encoded in a density function $p(\boldsymbol{\vartheta})$. The distribution of the parameters $\boldsymbol{\vartheta}$ conditioned on the known values y_i is given by Bayes' theorem: $p(\boldsymbol{\vartheta}|\boldsymbol{y}; \vec{x}) \propto p(\boldsymbol{y}|\boldsymbol{\vartheta}; \vec{x})p(\boldsymbol{\vartheta})$. The posterior density function can either be optimized or sampled.

By optimizing the posterior density we obtain a single value for the vector $\boldsymbol{\vartheta}$ that represents the value of the parameters with the highest probability. If the derivatives of the posterior with respect to $\boldsymbol{\vartheta}$ are available, a local optimization method can be used, e.g., the Adam algorithm [15]. Otherwise, a derivative free optimization algorithm can be used, e.g., evolution strategy (ES) algorithms. At every iteration, this type of algorithms draw samples from a parametrized distribution family and rank them from highest to lowest value. For example, the CMA-ES [2] uses this ranking and the history of the evolution to update the mean and the covariance matrix of a normal distribution. In the limit, the normal distribution converges to a Dirac distribution located at an optimal value. The covariance at each iteration can be interpreted as scaled approximation of the Hessian of the objective function at the mean of the normal distribution.

If we sample the posterior distribution instead of optimizing it, we obtain a set of samples that represent the volume of the distribution in the parameter space. This extended information, compared to the single parameter obtained with optimization, allows us to estimate the uncertainty in the inference of the parameters. Moreover, the uncertainty can be propagated to quantities of interest of the computational model, assessing this way the uncertainty in the predictions of the model. If derivatives are available, algorithms like Hamiltonian Monte Carlo (HMC) [16] can be utilized that accelerate the sampling and are efficient in high-dimensional spaces. In the opposite case,
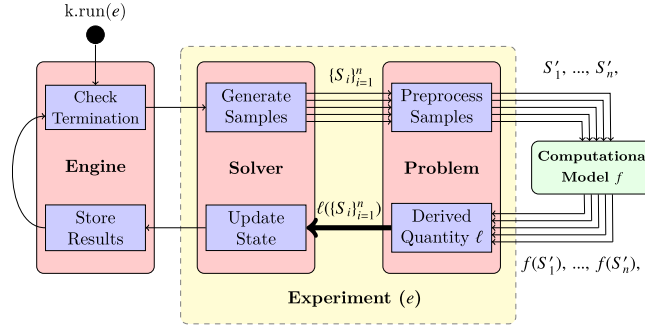
Fig. 1. Generation-based workflow of the sampling engine. The *solver* module generates *samples* S_i. The *problem* module pre-processes the samples into S_i' and passes them to a computational model. The results of the computational model $f(S_i')$ are collected and post-processed by *problem* into $\ell(S_i)$. Thereafter the derived quantities $\ell(S_i)$ are relayed to the *solver* and the state of the solver is updated according to the rules of the algorithm.

derivative-free algorithms, similar to Metropolis–Hastings, nested sampling (NS) [17] or Transitional Markov chain Monte Carlo (TMCMC) [18] can be utilized. In particular, the NS and the TMCMC algorithms benefit from a parallel evaluation of samples, allowing the acceleration of sampling of problems that involve computationally demanding models.

The common pattern between optimization and sampling algorithms is the iterating cycle of evaluation of the posterior density function. Additionally, for algorithms like ES, NS and TMCMC many function evaluations can be executed in parallel within the same iteration.

3. Framework design

The framework is specially tailored for the execution of massively parallel population-based algorithms that rely on generating and evaluating sets of parameters (samples) iteratively. At each iteration, a set of n samples S_i are evaluated by a statistical model ℓ. The statistical model to use is given by the choice of *problem*, *e.g.*, *Optimization*, to search the optimum of an objective function; *Sampling*, to sample an unnormalized density function, and; *Bayesian Inference*, to sample the posterior distribution and uncertainty of an inverse Bayesian problem. The statistical model evaluation may require the execution of a computational model (f) representing, *e.g.,* the simulation of a physical system.

To solve a given problem, the framework runs an *experiment* (see Fig. 1). Experiments are user-defined (for a detailed description of the user interface, see Appendix A) and contains the required configuration for the problem, the solver algorithm, and the *variable space*. The variable space represents the range of values within which the solution is to be identified. Variables are uniquely identified by their name and can be restricted either through an upper and a lower bound, or described by a prior distribution.

Experiments run under the control of the framework's sampling engine. The engine will coordinate the exchange of samples between the experiment and the computational model until the solver terminates. Fig. 1 shows the workflow of the engine when executing a given experiment. A *generation* represents a cycle in which the experiment produces and evaluates a population of samples. We define a *sample* S_i as a particular selection of values within the feasible variable space.

The first generation starts when the user runs `k.run(e)`, where `e` is the user-defined experiment object and `k` represents an instance of the engine. The first step in every generation is to check whether any of the defined termination criteria has been met. If not, the engine yields execution to the solver algorithm, which generates an initial population of samples $\{S_i\}_{i=1}^n$ and relays them to the problem module for pre-processing. During this stage, the samples are transformed to the correct input format for the computational model. The samples (S_i') are then passed on to the computational model for evaluation ($\{f(S_i')\}_{i=1}^n$).

Upon receiving the results from the computational model, the problem module calculates a derived quantity, e.g., the log-likelihood $\ell(S)$ for a problem of type Bayesian inference, and passes it back to the solver module. The solver uses these quantities to update its internal state and produce partial results, which serve as the basis for creating the next sample population during the next generation.

3.1. Distributed sampling engine

The sampling engine supports the parallel evaluation of multiple computational models using message passing interface (MPI). To enable parallel sampling, the user runs multiple instances of the Korali application, typically via the *mpiexec* command.[1] The engine determines the number k of processes that have been instantiated and assigns roles to them. The first process assumes the role of the *engine*, managing sample production and evaluation, as shown in Fig. 1. The rest of $k - 1$ processes assume the role of *workers*, whose task is to wait for incoming samples, run the computational model f, and to return their results.

The distribution of samples and collection of results is managed by a *distribution conduit* module between the experiment and the computational model. This conduit keeps track of the state of each worker (i.e., `idle`, `working`, `pending`), and assigns incoming samples workers that are in the `idle` state. As soon as a sample is received, a worker transitions to `busy` state, during which it executes the model f. When the worker finishes executing f, it returns the results and sets its state to `pending`, indicating that the engine can collect the result. The worker state transitions back to `idle` only after the conduit has retrieved the result. The engine employs an opportunistic strategy for work distribution in which it maintains a common queue of pending samples from which all workers are served. The conduit distributes samples on a one-by-one basis, in real-time, as soon as a worker becomes `idle`.

The engine also supports the evaluation of parallel (MPI-based) models. For this case, the execution conduit creates a set of *worker teams*, each assigned a subset of MPI ranks. All ranks from the same team work together in the evaluation of incoming samples. Users define the number of MPI ranks per team (m) through the ''Ranks Per Worker Team'' configuration parameter. For a run with N MPI ranks (as specified in the MPI launch command), the conduit assigns one rank to the sampling engine, and creates k worker teams, each with $\lfloor (N - 1)/m \rfloor$ ranks. Every worker team owns their private MPI communicator, which allows message passing between the ranks contained therein. Any MPI-based computational model passed to Korali should use this team-specific communicator for message exchanges. To identify the ranks in a given team, the conduit module appends an `MPI Communicator` field to the sample indicating which group corresponds to the receiving worker. With this value, the model can determine the m number of ranks in the team and the rank identifiers therein. The model can then operate like a regular MPI application and produce a result collaboratively.

A novelty in the sampling engine is the ability to execute multiple independent experiments concurrently. The goal is to maximize the pool of pending samples at any moment during execution, maximizing worker utilization in the presence of load imbalance, i.e. an uneven distribution of work among workers (see Section 4.2). Fig. 2 shows the engine's dataflow when executing two experiments simultaneously (e_0, and e_1) on a supercomputer cluster. The engine switches its execution context between both experiments, continuously polling whether either is ready to advance to the next generation or return partial results for storage. During execution, each experiment produces and evaluates its own set of samples (S' for e_0 and T' for e_1).

The distribution conduit manages each of the experiment's samples independently, distributing them among the common set of workers. Depending on which experiment has issued the sample, the conduit indicates to the worker which computational model to run. In this case, f, if the sample belongs to e_0, or; g, if the sample belongs to e_1. The results are asynchronously returned to the collection module, which distributes them back to the corresponding experiment. The engine evaluates each experiment's termination criteria after the given experiment reaches the end of its current generation. Experiments advance independently from each other, storing their results in separate folders, and the engine returns when all of them complete.

3.2. Modularity and fault-tolerance

The framework can be extended by adding new problem types and solver modules. To integrate a new module, developers create a new folder in the source code, containing three files: the module's base C++ class declaration header (`.hpp`) file, its method definitions in the source (`.cpp`) file, and a configuration (`.config`) file. Although the module class may contain any arbitrary number of method definitions, it must satisfy a common interface of abstract virtual methods. These methods are necessary to run a generation-based paradigm and depend on whether the new module is of *problem* or *solver* type.

---

[1] For systems that do not support MPI, the `Concurrent` execution mode can be used to run with multiple concurrent processes using a fork/join strategy instead.
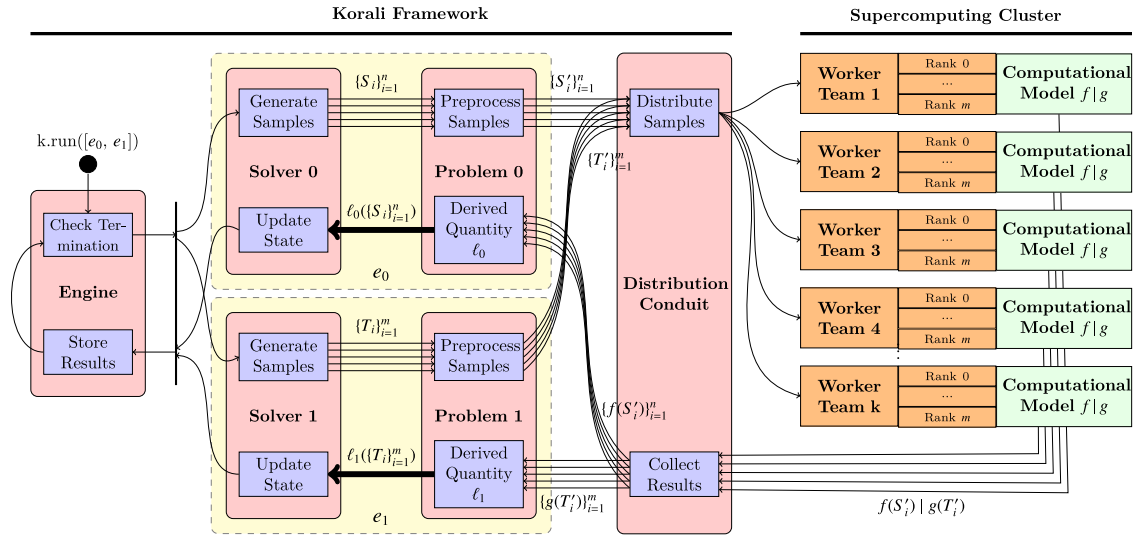
Fig. 2. Dataflow of the framework's sampling engine running two experiments, e_0 and e_1, concurrently. Samples from both experiments are passed through the distribution conduit, which assign them to any available *idle* worker. Here, workers represent teams of m MPI ranks, which collaborate to compute an assigned sample. Workers evaluate the model corresponding to the sample's experiment (f, for e_0, or; g, for e_1). Upon receiving results, the distribution conduit assigns them to the corresponding experiment.
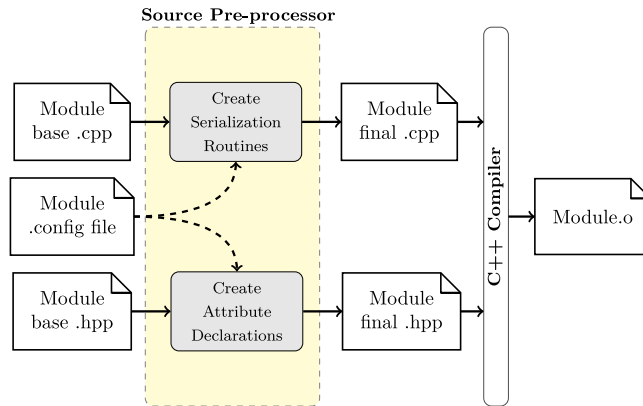

Fig. 3. Korali's source pre-processor takes the module's `.cpp` and `.hpp` files as inputs and appends to them automatically-generated class attributes and serialization routines, based on its `.config` file. The output source is compiled into an object file and linked to the framework.

The purpose of the configuration file is to automatize and enforce the generation of serialization routines, which write the entire state of a module into a `.json` file. The engine calls these routines at the end of every generation to save the internal state of a module. The state file serves as a checkpoint from which the execution can be resumed in case of failure or to split large jobs into shorter stints. The engine also stores the internal state of all random number generators to guarantee that the same intermediate and final results are obtained in case of resuming from a file. This approach guarantees reproducible results in long-running experiments that require more than one job to complete.

The engine's source pre-processor creates the serialization routines automatically before compilation, based on the fields declared in the configuration file, as shown in Fig. 3. The pre-processor enforces that no class members are declared in the header (`.hpp`) file. Instead, class members should be inserted as entries in the configuration file specifying a member name, a C++ datatype, a description text, and a default value. In this way, the framework ensures that the entire internal state of the module is accounted for during serialization. The pre-processor adds these configuration fields as class members to the module's base `.hpp` header file automatically, so that they can
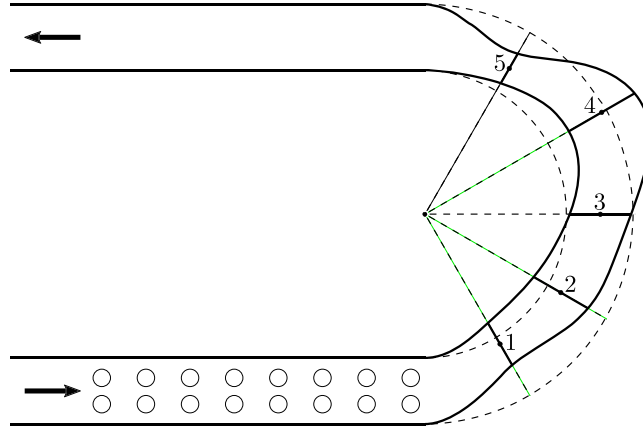
Fig. 4. Pipe shape parametrized by width and radial offset, marked with numbers 1 to 5, along five equiangular directions indicated by straight dashed lines. Offsets are relative to the baseline shape (dashed lines U-turn). The overall shape of the pipe results from interpolation. Circles show the initial positions of bubbles and the arrows indicate the flow direction.

be referenced from the `.cpp` file. The declaration and definition of the serialization methods are automatically generated and added to the module's source. A secondary product of code pre-processing is the automatic production of documentation pages for the module's entry in the user manual [19].

4. Experimental evaluation

We tested Korali on three research studies. First, an optimization study of the shape of fluid transporting pipes. This study shows the use of Korali on a large-scale, high-dimensional optimization job. Second, a hierarchical Bayesian analysis of the dissipation parameter of human RBC membranes. This study exemplifies the efficiency gains due to scheduling simultaneous experiments in the presence of load imbalance. Lastly, a Bayesian study on the parameters of a coarse-grained water model. This study demonstrates the fault-tolerance mechanisms and reproducibility mechanisms in Korali.

As computational platform, we used both XC40 and XC50 partitions of *Piz Daint* [20], a Cray supercomputer located at the *Swiss National Supercomputing Centre (CSCS)*. The XC40 partition comprises 1'813 compute nodes, each equipped with two Intel Xeon E5-2695-v4 18-core processors running at 2.10 GHz and 128 GB RAM. The XC50 partition comprises 5'704 compute nodes, each equipped with a single Intel Xeon E5-2690-v3 12-core processor, running at 2.60 GHz and 64 GB RAM, and an NVIDIA "Tesla" P100 graphics processing unit (GPU) with 16 GB of device memory. In Appendix B, we provide all resources required to reproduce the results presented in this paper.

4.1. Study 1: Fluid transporting pipes

Pipe networks are commonly used to convey flowing substances at various scales ranging from microfluidics to industrial facilities. The flow pattern in a pipe is determined by its shape and the flow conditions. Here we apply Korali to optimize the shape of a two-dimensional pipe which transports liquid with bubbles. The pipe consists of two straight segments connected with a U-turn, as illustrated in Fig. 4, where circulating bubbles can coalesce into larger bubbles. We consider three cases of optimization with different objectives: (i) minimize the transport time of bubbles, *i.e.,* the time for all bubbles to exit the pipe; (ii) maximize the maximum bubble volume at the time when it exits the pipe, and; (iii) minimize the maximum bubble volume and therefore achieve a state without coalescence.

The shape of the pipe is defined by 10 parameters that specify the width and radial offset of the U-turn along five directions used as knots of two cubic splines. The flow parameters are the Reynolds number $Re = wV\rho_l/\mu_l = 100$ and the capillary number $Ca = \mu_l V/\sigma = 0.1$ defined from the liquid density ρ_l, liquid viscosity μ_l, surface tension σ, pipe width w, and mean inlet velocity V. The gas density and viscosity are set to $0.1\rho_l$ and $0.1\mu_l$.
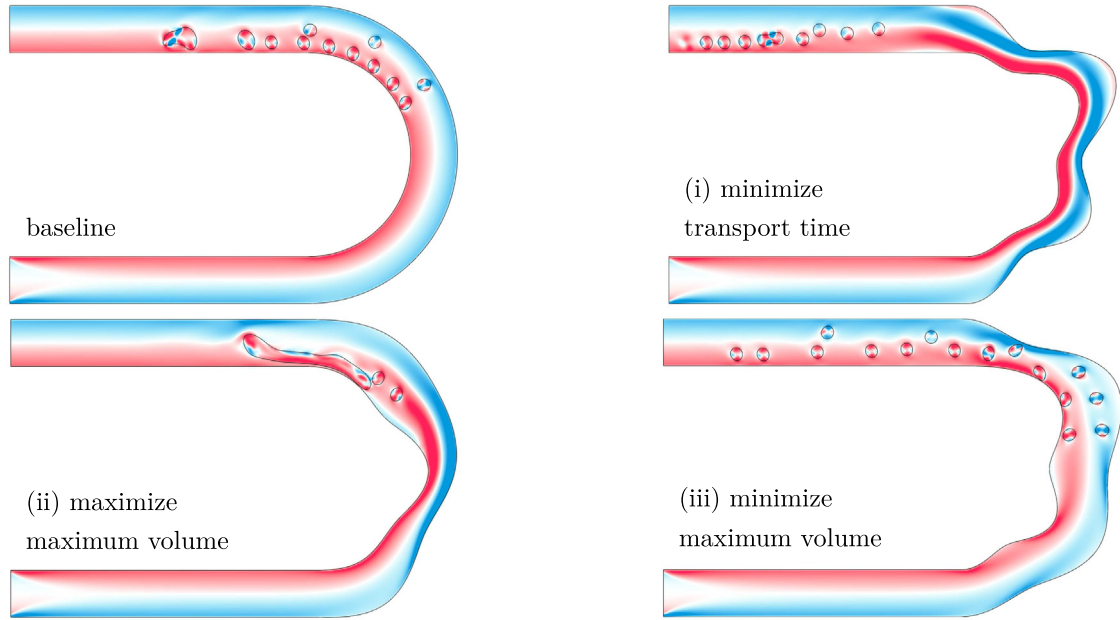
Fig. 5. Visualization of the baseline pipe shape and the results for the three optimization cases. The vorticity field is shown in blue, for clockwise, and; red, for counterclockwise. The snapshots are taken at time $tV/w = 14$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

We find the optimal shape of the pipe for the three cases using CMA-ES as optimization algorithm, with a population size of 32 samples per generation. As computational model, we used a finite-volume solver (APHROS [21,22]) with embedded boundaries [23] to treat complex geometries and a particle method [24] for surface tension forces. Each instance of APHROS ran on 4 nodes (72 cores) of the XC40 partition, a total of 128 nodes (2'304 cores) per case. Each case ran for an average of 15 h and consumed approximately 2k node hours.

Results of the optimization are shown in Fig. 5. Case (i) results in a shape where the U-turn contracts and then expands to redirect the bubbles towards the centerline, where velocity is maximized. The transport time of the optimized shape has decreased by a factor of 2.1 compared to the baseline. Case (ii) forms a cavity at the end of the U-turn where the flow stagnates and all bubbles except one coalesce into one elongated bubble. Finally, case (iii) results in a wide shape where bubbles circulate in parallel lanes, achieving a state without coalescence.

4.2. Study 2: Red blood cell membrane viscosity

RBCs are highly deformable objects that incur complex dynamical transitions in response to external disturbances. These transitions lay the foundation for understanding the rheology of blood in flows through capillaries, vascular networks, and medical devices. There is still significant uncertainty in the choice of the mechanical law to describe the RBC properties, as well as in the parameter values of each model [27].

In this study, we infer the membrane viscosity which controls the relaxation time of an RBC membrane. Here, the RBC membrane is modeled as a collection of particles placed on the nodes of a triangular network [28]. We used data from five experimental observations (four from Hochmuth,Hochmuth1979 and one from Henon [29]), on the relaxation of a stretched RBC to its equilibrium shape in order to infer the posterior distribution of the membrane viscosity (η_m, see Fig. 6), and its associated uncertainty (σ). Due to the presence of heterogeneous data, we employed a hierarchical Bayesian model. The sampling of the posterior distribution is approximated by a two stage importance sampling algorithm. A detailed description of the statistical model, the experimental data, and the results of the hierarchical model can be found in a previous work [26].

Here, we analyze the performance of the framework during the first stage, where the parameters were sampled individually, conditioned on each experimental data set. For sampling, we employed BASIS, a reduced bias
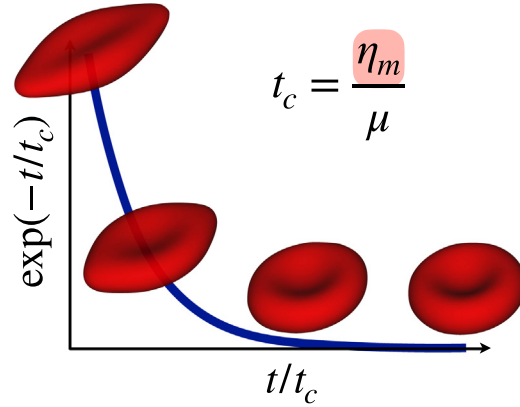
Fig. 6. Relaxation of a RBC from an initially elongated state to the biconcave resting shape. The time evolution of the length and width of the RBC are non-dimensionalized following [25], with the dimensionless RBC size following an exponential decay. Time is non-dimensionalized by the characteristic relaxation time, $t_c = \eta_m / \mu$, with μ the elastic shear modulus and η_m the RBC (2D) membrane viscosity. The latter can be inferred given a set of experimental measurements [26].
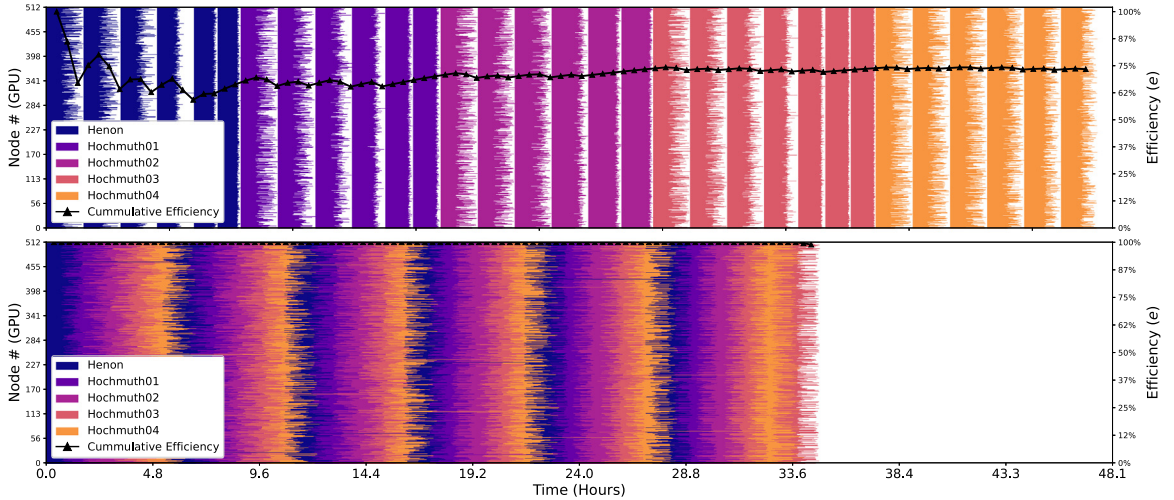

Fig. 7. Core usage timelines of Korali during sampling of the experimental datasets, starting from Henon (darkest shade), Hochmuth01, Hochmuth02, Hochmuth03, and ending on the right with Hochmuth04 (lightest shade). The figure shows two timelines: on top, with sequentially scheduled BASIS experiments, and; on bottom: multiple experiments scheduled simultaneously. The horizontal axis represents the elapsed time (minutes) from the start of the experiment. On the vertical axis, each line represents a different node. Solid lines represent the execution of the model. Blank spaces represent times where a node is idle. The black line indicates the cumulative sampling efficiency (*e*) across time. A higher efficiency reflects a better node usage. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

variant of the TMCMC algorithm for the sampling of the posterior distribution. BASIS is a sampling algorithm tailored for Bayesian uncertainty quantification and targeted to parallel architectures [30]. We configured BASIS to run a population size of 512 samples per generation. For the computational model, we used MIRHEO [31], a high-performance GPU-based library for microfluidic simulations.

We ran the five experiments on 512 nodes of the XC40 partition using 512 MPI ranks, each running an instance of MIRHEO per node. In a previous work [26], we had found that the RBC membrane relaxation model shows a high variance in running times (40 ∼ 100min per sample). This variance caused a workload imbalance among the workers, with detrimental impact on the performance of the BASIS algorithm. The effect can be appreciated when running each of the five BASIS sampling experiments individually, as shown in Fig. 7 (top). The five experiments took 48.1 h to complete on 512 nodes, requiring a total of 24.6k node hours. This approach yielded sampling

Table 1

Performance comparison between the two scheduling strategies employed in the RBC stretching experiment: Single, with individually scheduled experiments, and; Multiple, with all experiments scheduled simultaneously. Here, e represents sampling efficiency. The energy usage measurements were obtained from *Piz Daint*'s job scheduler.

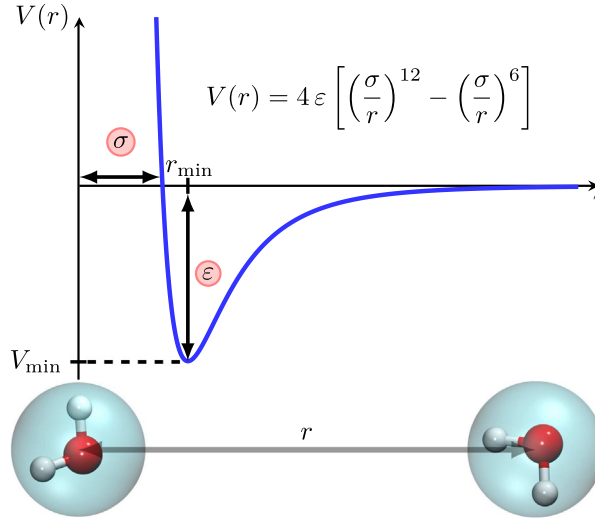| Scheduling	Time	Node Hours		e	Energy
		Total	Idle		
Single	47.32 h	24227	6604	**72.7%**	10.45 GJ
Multiple	34.78 h	17809	186	**98.9%**	7.80 GJ

Fig. 8. The Lennard-Jones potential is a function of the distance (r) between two water molecules, represented as solid spheres. It is repulsive for distances less than $r_{\min}$ and attractive for larger distances. The potential is parametrized by ε that controls the depth of the well ($V_{\min}$), and by σ that controls the change from repulsion to attraction point.

efficiency[2] (e) of 72.7%. It follows that nodes remained idle 27.3% of their running time. Table 1 (row: *Single*) shows that only this resulted in a loss of 6.6k (idle) node hours. In total, the energy usage, as reported from *Piz Daint*'s job scheduler, was of 10.45 GJ.

To alleviate the effect of load imbalance, we configured Korali to schedule all five experiments simultaneously. The timeline in Fig. 7 (bottom) shows that nodes remained busy during most of the run with the multi-experiment variant. The results, summarized in Table 1, indicate that this approach yields a superior efficiency (98.9%) compared to the former approach, wasting much fewer node hours (186), as well as requiring less energy (7.80 GJ). Furthermore, it also reduced the run-to-completion time from 47.32 to 34.78 h.

4.3. Study 3: Coarse-grained water model

In this study, we apply Bayesian inference on the assumed Lennard-Jones potential between particles with two parameters (ε and σ) for a coarse-grained water model where a single water molecule is represented by a solid sphere (see Fig. 8). We reproduced the computational setup of a previous work [32], where the parameters of the model are calibrated to experimental data (density, dielectric constant, surface tension, isothermal compressibility, and shear viscosity) at fixed temperature and pressure. To find the parameters that maximize the posterior distribution of the parameters, we use CMA-ES as optimization algorithm, with a population size of 16 samples per generation. For the computational model, we used LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [33], a well-known molecular dynamics simulation library that models atoms or ensembles of particles in solid, liquid

---

[2] Calculated as the ratio between busy and total runtime, i.e. busy and idle time combined.
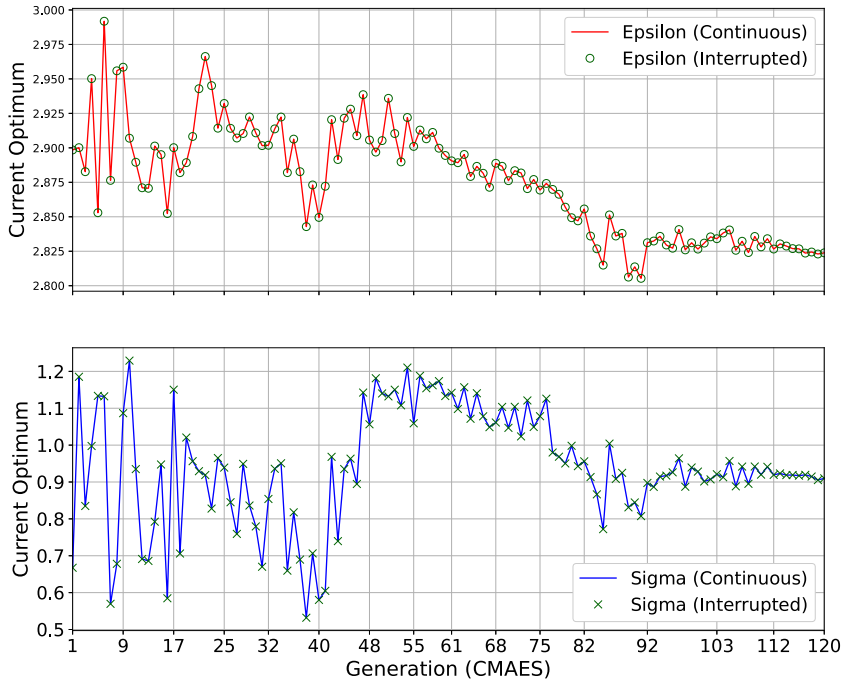
Fig. 9. Evolution of the parameter values that maximize the posterior distribution of the Bayesian problem, showing: in solid lines, the run in which CMA-ES was uninterrupted, and; in markers, the run which was interrupted every 15 min. Vertical grid lines indicate the generations at which the interrupted run was restarted.

or gaseous state. We run the experiment using 16 compute nodes of the XC50 partition and two workers per node. Each worker ran an instance of LAMMPS using 2 MPI ranks over 12 OpenMP threads.

Here, we validate the framework's fault-tolerance and reproducibility by running the same study twice. For the first run, we allow Korali to complete without interruptions. For the second run, we allow it to run for only 15 min at a time before forcing the job scheduler to terminate it. To reach the final results with the interrupted run, we re-schedule its launcher job after each interruption, reloading the internal state of CMA-ES upon restart. Since both runs use the same random seed, we expect to observe the exact same intermediate and final results. Fig. 9 shows the comparison of the per-generation evolution of parameter optima between the single-run execution (continuous line), and the interrupted execution (markers) for the two optimization parameters. In the figure, vertical grid lines indicate the generation at which the interruptions occurred for the latter, for a total of 16 restarts. Results show that the optimal parameters and their convergence path was identical for both runs.

5. Related work

Many of the problems and solver methods currently implemented in Korali can also be recognized across other statistical UQ softwares. These include *ABC-SysBio* [7], *APT-MCMC* [8,34], *BCM* [35,36], *BioBayes* [37], *Dakota* [13], *EasyVVUQ* [38], *Π4U* [12], *MUQ* [39], *PSUADE* [9], *QUESO* [10], *ScannerBit* [40] (a *GAMBIT* [41, 42] module), which are standalone applications; *Chaospy* [43], *Uncertainpy* [44,45], *UQ Toolkit* [46], *UQPy* [47], which are publicly available Python packages; *Stan* [11], a programming language for statistical inference; and *UQLab* [5], a MATLAB framework.

Here, we analyze in further detail the *Π4U* and Dakota frameworks as they offer support for distributed sampling and thus relate closer to Korali's goal. The *Π4U* [12] framework is one of the first efforts in providing support of distributed UQ experiments and, to the best of our knowledge, the only to have reported detailed performance metrics at scale. *Π4U* employs the TORC tasking library [48] to distribute the execution of samples to workers. *Dakota,dalbey2020dakota* is a well-established C++-based framework for uncertainty quantification, that interfaces with simulation software through a multi-level MPI-based parallelism interface. Dakota is used in a wide-range of applications for the US Department of Energy [49].
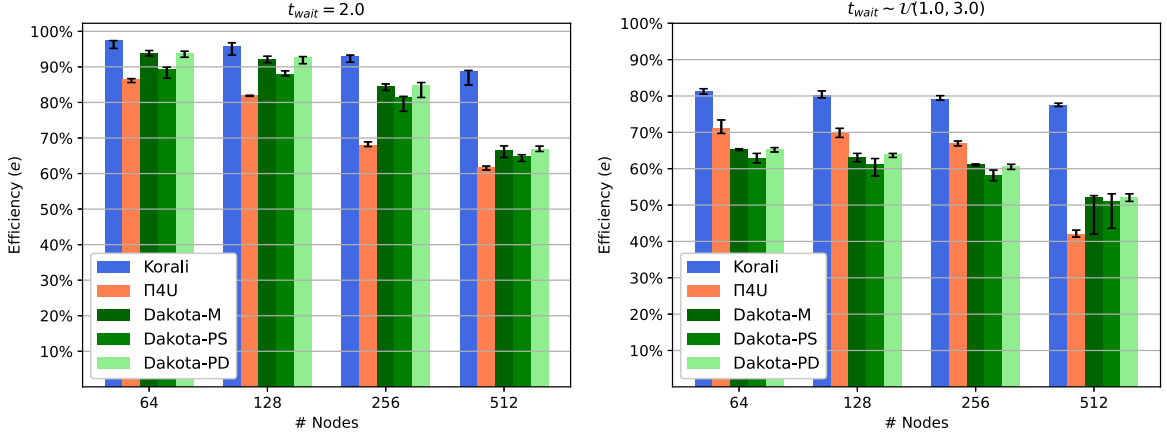
Fig. 10. Weak scaling studies comparing the sampling efficiency (e) of Korali, Π4U, and Dakota (three variants) for a synthetic benchmark (left) without ($T_{wait} = 2.0$), and (right) with load imbalance ($T_{wait} \sim \mathcal{U}(1.0, 3.0)$) on 64, 128, 256, and 512 nodes. The efficiency is calculated as the ratio of busy and total running time, i.e. busy and idle time combined. The colored bars highlight the median efficiencies, and the black intervals indicate the maximum and minimum. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

To compare the efficiency between frameworks, we created a synthetic benchmark that runs a single-variable optimization experiment on a computational model that passively waits for a given number of seconds (T_{wait}) before returning a random result. By employing a passive wait, we can fix the running time of each sample, ruling out time variances common to compute-intensive computational models. To drive sampling, we employ evolutionary optimization algorithms (CMAES [50], for Korali and Π4U, and; [51], for Dakota) configured to run 5 generations, $4N$ samples per generation,[3] where N is the number of workers, and one node per worker. This configuration allows to conduct weak scaling studies, evaluating the impact of node scaling on efficiency, while keeping the workload per worker constant. We configured average waiting times to add up to 40 s in total per experiment, which represents their ideal running time. We measure efficiency (e) for each framework by dividing the ideal time by its actual running time.

We test the following 5 variants: Korali, Π4U, Dakota-M (master/worker scheduler), Dakota-PS (static scheduler), and Dakota-PD (dynamic scheduler). We use the synthetic benchmark to run two weak scaling studies, with and without load imbalance. To account for the effect of stochastic waiting times, we ran 10 repetitions of each experiment. In Appendix B, we provide the experimental setup and data necessary to replicate the results.

The first study represents a scenario where there is no load imbalance ($T_{wait} = 2.0$ seconds, for all samples). Here, we measure the inherent efficiency of the frameworks in distributing samples to workers without the detrimental effect of imbalance. The gap between the attained and an ideal efficiency therefore illustrates the time spent on communication, I/O operations, and scheduling overhead only. Fig. 10 (left) shows the results of weak scaling by running the 5 variants from 64 to 512 nodes of the Piz Daint supercomputer. All variants provide high efficiencies ($86\% < e < 97\%$) at smaller scales (64 and 128 nodes), with Korali as the most efficient by a small margin. At the largest scale (512 nodes), the differences are evident, since both Π4U ($e = 62\%$) and Dakota ($e = 67\%$) appear to be especially susceptible to the increasing scheduling costs, compared to Korali ($e = 86\%$).

The second study simulates experiments with a high load imbalance. Here, the waiting time for each sample is drawn from a random variable $T_{wait} \sim \mathcal{U}(1.0, 3.0)$ seconds. We consider the same ideal case (in average, 40 s per experiment) as basis for the calculation of efficiency. Fig. 10 (right) shows the results for this study. We observe that load imbalance plays a detrimental effect on the efficiency of all variants. However, Korali is the least affected of them throughout all scales. We observe the larger difference between the variants when running on 512 nodes and the larger imbalance, where Π4U and Dakota show a low efficiency ($e = 41\%$ and $e = 52\%$, respectively), while Korali sustains a higher performance ($e = 78\%$).

---

[3] For fairness, we verified that the generation pre- and post-processing times for all algorithms are negligible.

6. Conclusions and future work

We introduced Korali, a unifying framework for the efficient deployment of Bayesian UQ and optimization methods on large-scale supercomputing systems. The software enables a scalable, fault-tolerant, and reproducible execution of experiments from a wide range of problem types and solver methods. The experimental results have shown that Korali is capable of attaining high sampling efficiencies on up to 512 nodes of *Piz Daint*. Furthermore, we have shown that running multiple experiments simultaneously on a common set of computational resources minimizes load imbalance and achieves almost perfect node usage, even in the presence of high sample runtime variance. We have also shown that the framework can run out-of-the-box libraries, such as LAMMPS, while providing fault tolerance mechanisms. Finally, we demonstrate that Korali attains higher efficiencies than other prominent distributed sampling frameworks.

We are currently integrating support for distributed Reinforcement Learning (RL) methods [52,53] that target the incremental optimization of a policy for multiple concurrent agents exploring and acting in a virtual environment to collect and maximize rewards. The architecture for these methods closely resembles the workflow provided by Korali, thus making them suitable candidates for integration. We believe that a platform integrating stochastic optimization, UQ and RL will be of great value for the broad scientific community.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Appendix A. Interface design

Korali employs a *descriptive* interface, in which experiments are statically defined as a set of parameters. This interface is mostly language-independent and requires only trivial knowledge of the underlying programming language (*e.g.,* Python or C++). Fig. A.11 shows an example of a Python-based[4] application that solves the problem of calibrating the parameters of a computational model on experimental data. We configured the example to describe a Bayesian inference problem where the uncertainty in the parameters is quantified by sampling the posterior distribution of the parameters conditioned on the data. To better explain the software interface, we define the statistical problem first, and then show its correspondence with the code in Fig. A.11.

The vectors $\vec{x}$ and $\boldsymbol{y}$ correspond to the variables X and Y in the code of Fig. A.11 that are initialized in Lines 6 and 7 through user defined functions. Korali works by defining and running an *experiment*. (Line 10) An experiment consists of the description of a statistical problem to be solved, a description of the involved variables and their distributions, and the configuration of the desired solver. In this application, all lines of code between Line 13 and Line 40 that are required for the description of the problem, are made entirely via dictionary-tree accesses. The rest of the code consists of importing libraries (Lines 1 and 4), initializing the experiment (Line 10), initializing the engine (Line 43), and running Korali (Line 45). The type of the problem is defined in Line 13, and the likelihood function is defined in Line 14. The observations Y are passed to Korali in Line 16 and the computational model in Line 15. In top of Fig. A.12 an example of a computational model is given, where $f(\boldsymbol{x}_i; \boldsymbol{\vartheta}) = \vartheta_1 \boldsymbol{x}_i + \vartheta_2$. Next, the variable vector $\boldsymbol{\vartheta}$ is defined by the experiment's *variables*. Each variable is defined by a unique name and represents one entry to the variable vector. The example code contains three variables, P1, P2 and Sigma, in Lines 19 to 21. The variables are passed in the user-defined model F and used to compute the likelihood function, given by the keywords ``Reference Evaluation'' and ``Standard Deviation'', respectively. To complete the description of the problem, the variables require the definition of a prior distribution $p(\boldsymbol{\vartheta})$. Here, we specify

---

[4] Although we use Python in the examples, Korali provides a similar C++-based interface that allows linking its engine against C++ and Fortran computational models.

```
 1    import korali
 2
 3    # Importing the computational model and the data
 4    from myLibrary import F
 5
 6    X = getReferenceInput()
 7    Y = getReferenceData()
 8
 9    # Creating new experiment
10    e = korali.Experiment()
11
12    # Setting up the Bayesian Inference Problem
13    e["Problem"]["Type"] = "Bayesian Inference"
14    e["Problem"]["Likelihood Model"] = "Normal"
15    e["Problem"]["Computational Model"] = lambda s:F(s,X)
16    e["Problem"]["Reference Data"] = Y
17
18    # Configuring the problem`s variables and their priors
19    e["Variables"][0]["Name"] = "P1"
20    e["Variables"][1]["Name"] = "P2"
21    e["Variables"][2]["Name"] = "Sigma"
22    e["Variables"][0]["Prior Distribution"] = "D1"
23    e["Variables"][1]["Prior Distribution"] = "D1"
24    e["Variables"][2]["Prior Distribution"] = "D2"
25
26    # Configuring the prior distributions
27    e["Distributions"][0]["Name"] = "D1"
28    e["Distributions"][0]["Type"] = "Univariate/Normal"
29    e["Distributions"][0]["Mean"]  = 0.0
30    e["Distributions"][0]["Sigma"] = +2.0
31
32    e["Distributions"][1]["Name"] = "D2"
33    e["Distributions"][1]["Type"] = "Univariate/Uniform"
34    e["Distributions"][1]["Minimum"] =  0.0
35    e["Distributions"][1]["Maximum"] = +5.0
36
37    # Configuring Solver (TMCMC)
38    e["Solver"]["Type"] = "TMCMC"
39    e["Solver"]["Population Size"] = 5000
40    e["Solver"]["Covariance Scaling Factor"] = 0.04
41
42    # Starting Korali's Engine and running experiment
43    k = korali.Engine()
44
45    k.run(e)
```

Fig. A.11. Example of a Python-based Korali Application.

that the prior distribution of the variables corresponding to the parameters P1 and P2 is a normal distribution (Lines 22 and 23), and of the variable corresponding to the variable `Sigma` a uniform distribution (Line 24). Finally, we set the solver to the TMCMC sampler [54].

A.1. Computational model support

The user specifies the computational model by passing a function as part of the problem configuration. Such a function should expect the sample's information as argument. In the example in Fig. A.11, the computational model is passed as a lambda function that calls the computational model F, imported from the `myLibrary` module.

Functions passed as computational models do not return a value. Instead, they save their results into the sample container. The expected results from the execution of the computational model depend on the selected problem type. Fig. A.12 (Top) shows the function F, as specified in the example in Fig. A.11. A Bayesian inference problem, where the likelihood is computed from reference data, requires that the model saves an evaluation of each of the reference data points into a ''`Results`'' vector. Other problem types, such as *derivative-free optimization*, require the model to store only a single numerical value corresponding to the function evaluation (''`F(x)`'') for the given parameter(s), as shown in Fig. A.12 (Middle).

The interface accommodates legacy codes through a fork/join-based `Concurrent` execution mode that allows instancing pre-compiled applications via shell commands and returns the results either through file or pipe I/O

```
1  def F(sample, X):
2      p1 = sample["Variables"]["P1"]
3      p2 = sample["Variables"]["P2"]
4      s  = sample["Variables"]["Sigma"]
5
6      s["Reference Evaluations"] = []
7      s["Standard Deviation"]    = []
8      for x in X:
9          s["Reference Evaluations"] += [a*x + b]
10         s["Standard Deviation"]    += [sig]
```

```
1  def myOptimizableModel(sample):
2      x = sample["Variables"]["X"]
3      sample["F(x)"] = -x * x
```

```
1  def myExternalModel(sample):
2      x = sample["Variables"]["X"]
3      args   = [ './myApp', '-x', str(x) ]
4      result = subprocess.check_output(args)
5      sample["F(x)"] = float(result)
```

Fig. A.12. Examples of computational models. (Top): A model that has two parameters P1 and P2 and produces as result a vector of evaluations, one for each value of the input vector X. (Middle): A model that requires a single variable X and produces a single function evaluation $f(x) = -x^2$, to be maximized using a derivative-free method. (Bottom): A model that executes an external application and returns its output as result.

operations. The Concurrent mode can also be used to launch and gather results from large-scale distributed, *e.g.,* MPI [55] applications. An example of such a model is given in Fig. A.12 (Bottom).

Appendix B. Experimental setup

We provide here the source code, configuration, and dependencies used to setup the experiments. We performed the experiments on the GPU partition of *Piz Daint* [20], a Cray XC50 system running on SUSE Linux Enterprise Server 15 SP1 (kernel v4.12). For compilation, we used a GNU-based programming environment, with gcc/8.3.0 as C/C++ compiler. We used cray-mpich/7.7.15 for MPI support, and cray-python/3.8.2.1 support Python3. The system uses SLURM 20.02 as its job scheduler. We employed open-source software for the experiments. The software versions used are as follows:

- Korali v2.0.0 (github.com/cselab/korali)
- *Π*4U v1.0 (github.com/cselab/pi4u)
- Dakota v6.12 (dakota.sandia.gov)
- LAMMPS v20.08 (lammps.sandia.gov)
- Mirheo v1.3.0 (github.com/cselab/Mirheo)
- Aphros (github.com/cselab/aphros).

The setup to reproduce the results from Section 4.1 can be found inside the examples/study.cases/bubblePipe/ folder of the Korali source code; for Section 4.2, they can be found inside the examples/study.cases/RBCStretc/ folder, and; for Section 4.3, they can be found inside the examples/study .cases/LAMMPS/ folder. The jobs/ folder provides the respective SLURM job scripts used for distributed runs.

The setup required to run the benchmarks from Section 5 can be found in the Github repository: github.com/cselab/korali_benchmark. The folders korali/, pi4u/, and dakota/ contain the configuration used for each of the frameworks tested. The code files in korali/ and pi4u/ need to be compiled with make prior to running the scripts. Inside the common/ folder is the common wait script for the three frameworks to use during sample evaluations. The scripts within the jobs/ folder contain variables that allow setting the experiment's workload imbalance.

References

[1] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, et al., The opportunities and challenges of exascale computing, Summ. Rep. Adv. Sci. Comput. Advis. Comm. (ASCAC) Subcomm. (2010) 1–77.

[2] N. Hansen, S.D. Muller, P. Koumoutsakos, Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES) nikolaus, Evol. Comput. 11 (1) (2003) 1–18.

[3] S. Kern, S.D. Müller, N. Hansen, D. Büche, J. Ocenasek, P. Koumoutsakos, Learning probability distributions in continuous evolutionary algorithms –A comparative review, Nat. Comput. 3 (1) (2004) 77–112.

[4] Y. Akimoto, A. Auger, N. Hansen, Comparison-based natural gradient optimization in high dimension, in: Genetic and Evolutionary Computation Conference GECCO'14, ACM, Vancouver, Canada, 2014, URL https://hal.inria.fr/hal-00997835.

[5] UQLab, 2019, https://www.uqlab.com, (2019-06-13).

[6] EasyVVUQ, 2019, https://easyvvuq.readthedocs.io/en/latest/, (2019-06-13).

[7] J. Liepe, C. Barnes, E. Cule, K. Erguler, P. Kirk, T. Toni, M.P. Stumpf, ABC-SysBio—approximate Bayesian computation in python with GPU support, Bioinformatics 26 (14) (2010) 1797–1799.

[8] L.A. Zhang, A. Urbano, G. Clermont, D. Swigon, I. Banerjee, R.S. Parker, APT-MCMC, A C++/python implementation of Markov chain Monte Carlo for parameter identification, Comput. Chem. Eng. 110 (2018).

[9] PSUADE, 2019, https://computation.llnl.gov/projects/psuade-uncertainty-quantification, (2019-06-13).

[10] E.E. Prudencio, K.W. Schulz, The parallel c++ statistical library 'QUESO': Quantification of uncertainty for estimation, simulation and optimization, in: Euro-Par 2011: Parallel Processing Workshops, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 398–407.

[11] Stan, 2019, https://mc-stan.org, (2019-06-13).

[12] P. Hadjidoukas, P. Angelikopoulos, C. Papadimitriou, P. Koumoutsakos, Π4U: A High performance computing framework for Bayesian uncertainty quantification of complex models, J. Comput. Phys. 284 (2015) 1–21.

[13] K. Dalbey, M.S. Eldred, G. Geraci, J.D. Jakeman, K.A. Maupin, J.A. Monschke, D.T. Seidl, L.P. Swiler, A. Tran, F. Menhorn, et al., Dakota A Multilevel Parallel Object-Oriented Framework for Design Optimization Parameter Estimation Uncertainty Quantification and Sensitivity Analysis: Version 6.12 Theory Manual, Tech. rep., Sandia National Lab, 2020.

[14] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, et al., The international exascale software project roadmap, Int. J. High Perform. Comput. Appl. 25 (1) (2011) 3–60.

[15] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: Y. Bengio, Y. LeCun (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015, URL http://arxiv.org/abs/1412.6980.

[16] M.D. Homan, A. Gelman, The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo, J. Mach. Learn. Res. 15 (1) (2014) 1593–1623.

[17] F. Feroz, M.P. Hobson, M. Bridges, Multinest: An efficient and robust Bayesian inference tool for cosmology and particle physics, Mon. Not. R. Astron. Soc. 398 (4) (2009) 1601–1614, http://dx.doi.org/10.1111/j.1365-2966.2009.14548.x, arXiv:https://academic.oup.com/mnras/article-pdf/398/4/1601/3039078/mnras0398-1601.pdf.

[18] S. Wu, P. Angelikopoulos, C. Papadimitriou, P. Koumoutsakos, BayesIan annealed sequential importance sampling (BASIS): an unbiased version of transitional Markov chain Monte Carlo, ASCE-ASME J. Risk Uncertain. Eng. Syst. B: Mech. Eng. 4 (1) (2018).

[19] Korali user manual, 2021, https://www.cse-lab.ethz.ch/korali/docs/, (2021-01-15).

[20] CSCS piz daint, 2019, https://www.cscs.ch/computers/piz-daint/, (30-10-2019).

[21] Aphros: parallel solver for incompressible multiphase flows, 2020, https://github.com/cselab/aphros.

[22] P. Karnakov, F. Wermelinger, S. Litvinov, P. Koumoutsakos, Aphros: High performance software for multiphase flows with large scale bubble and drop clusters, in: Proceedings of the Platform for Advanced Scientific Computing Conference, 2020, pp. 1–10.

[23] P. Colella, D.T. Graves, B.J. Keen, D. Modiano, A cartesian grid embedded boundary method for hyperbolic conservation laws, J. Comput. Phys. 211 (1) (2006) 347–366.

[24] P. Karnakov, S. Litvinov, P. Koumoutsakos, A hybrid particle volume-of-fluid method for curvature estimation in multiphase flows, Int. J. Multiph. Flow. 125 (2020) 103209.

[25] R.M. Hochmuth, P. Worthy, E.A. Evans, Red cell extensional recovery and the determination of membrane viscosity, Biophys. J. 26 (1) (1979) 101–114.

[26] D. Wälchli, S.M. Martin, A. Economides, L. Amoudruz, G. Arampatzis, X. Bian, P. Koumoutsakos, Load balancing in large scale Bayesian inference, in: Proceedings of the Platform for Advanced Scientific Computing Conference, in: PASC '20, ACM, 2020, pp. 1–12.

[27] J. Sigüenza, S. Mendez, F. Nicoud, How should the optical tweezers experiment be used to characterize the red blood cell membrane mechanics? Biomech. Model. Mechanobiol. 16 (5) (2017) 1645–1657.

[28] D.A. Fedosov, B. Caswell, G.E. Karniadakis, A multiscale red blood cell model with accurate mechanics, rheology, and dynamics, Biophys. J. 98 (10) (2010) 2215–2225.

[29] S. Hénon, G. Lenormand, A. Richert, F. Gallet, A new determination of the shear modulus of the human erythrocyte membrane using optical tweezers, Biophys. J. 76 (2) (1999) 1145–1151.

[30] S. Wu, P. Angelikopoulos, G. Tauriello, C. Papadimitriou, P. Koumoutsakos, Fusing heterogeneous data for the calibration of molecular dynamics force fields using hierarchical Bayesian models, J. Chem. Phys. 145 (24) (2016).

[31] D. Alexeev, L. Amoudruz, S. Litvinov, P. Koumoutsakos, Mirheo: High-performance mesoscale simulations for microfluidics, 2020,

[32] J. Zavadlav, G. Arampatzis, P. Koumoutsakos, BayesIan selection for coarse-grained models of liquid water, Sci. Rep. 9 (1) (2019) 1–10.

[33] S. Plimpton, Fast parallel algorithms for short – range molecular dynamics, J. Comput. Phys. 117 (1995) 1–19.

[34] APT-MCMC, 2020, https://apt-mcmc.readthedocs.io/en/latest/, (2020-02-27).

[35] B. Thijssen, T.M.H. Dijkstra, T. Heskes, L.F.A. Wessels, BCM: toolkit for Bayesian analysis of computational models using samplers, BMC Syst. Biol. 10 (1) (2016) 100.

[36] BCM, 2019, http://ccb.nki.nl/software/bcm/, (2019-06-14).

[37] V. Vyshemirsky, M. Girolami, BioBayes: A software package for Bayesian inference in systems biology, Bioinformatics 24 (17) (2008) 1933–1934.

[38] R.A. Richardson, D.W. Wright, W. Edeling, V. Jancauskas, J. Lakhlili, P.V. Coveney, EasyVVUQ: A library for verification, validation and uncertainty quantification in high performance computing, J. Open Res. Softw. 8 (1) (2020).

[39] MUQ - MIT uncertainty quantification library, 2020, http://muq.mit.edu, (2020-02-27).

[40] G.D. Martinez, J. McKay, B. Farmer, P. Scott, E. Roebber, A. Putze, J. Conrad, Comparison of statistical sampling methods with ScannerBit, the gambit scanning module, Eur. Phys. J. C 77 (11) (2017).

[41] The GAMBIT Collaboration, GAMBIT: The global and modular beyond-the-standard-model inference tool, Eur. Phys. J. C 77 (11) (2017) 784.

[42] GAMBIT, 2019, https://gambit.hepforge.org, (2019-06-13).

[43] Chaospy, 2020, https://chaospy.readthedocs.io/en/master/tutorial.html, (2020-02-28).

[44] S. Tennøe, G. Halnes, G.T. Einevoll, Uncertainpy: A python toolbox for uncertainty quantification and sensitivity analysis in computational neuroscience, Front. Neuroinformatics 12 (2018).

[45] Uncertainpy, 2019, https://uncertainpy.readthedocs.io, (2019-06-13).

[46] R. Ghanem, D. Higdon, H. Owhadi, Handbook of Uncertainty Quantification, Vol. 6, Springer, 2017.

[47] UQpy, 2019, https://github.com/SURGroup/UQpy, (2019-06-13).

[48] P.E. Hadjidoukas, E. Lappas, V.V. Dimakopoulos, A runtime library for platform-independent task parallelism, in: Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2012, pp. 229–236.

[49] B.M. Adams, J.A. Stephens, Dakota Optimization and UQ: Explore and Predict with Confidence, Tech. rep., Sandia National Laboratories, 2018.

[50] N. Hansen, The CMA evolution strategy: A tutorial, 2016, arXiv:1604.00772.

[51] W.E. Hart, An Introduction to the COLIN Optimization Interface, Tech. rep., Sandia National Laboratories, 2003.

[52] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al., Massively parallel methods for deep reinforcement learning, 2015, arXiv:1507.04296.

[53] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, K. Kavukcuoglu, IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures, 2018, arXiv:1802.01561.

[54] J. Ching, Y. Chen, Transitional Markov chain Monte Carlo method for Bayesian model updating, model class selection, and model averaging, J. Eng. Mech. 133 (7) (2007) 816–832.

[55] MPI forum, https://www.mpi-forum.org/.