Deep Gaussian process with multitask and transfer learning for performance optimization

Wissam M. Sid-Lakhdar*, Mohsen Aznaveh†, Piotr Luszczek*, Jack Dongarra*‡
*University of Tennessee at Knoxville
†Texas A&M University
‡Oak Ridge National Laboratory, University of Manchester

2022 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-9786-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPEC55821.2022.9926396

***Abstract*—We combine Deep Gaussian Processes with multitask and transfer learning for the performance modeling and optimization of HPC applications. Deep Gaussian processes merge the uncertainty quantification advantage of Gaussian Processes with the predictive power of deep learning. Multitask and transfer learning allow for improved learning efficiency when several similar tasks are to be learned simultaneously and when previous learned models are sought to help in the learning of new tasks, respectively. A comparison with state-of-the-art autotuners shows the advantage of our approach on two application problems.**

I. Introduction

Automated performance engineering, a.k.a *autotuning*, focuses on finding the best hyper-parameters of an algorithm implementation (or kernel). Recently, autotuners have been used for optimizing machine learning applications. However, these efforts lack attempts at complete performance tuning for scientific applications.

Our work is motivated by *low-data regime* that precludes the use of Artificial Neural Networks (ANN) which need large data volumes to successfully generalize. Gaussian Process (GP) [1] is a solution that needs augmentation to handle our complex scenario of multiple applications/platforms. To handle the non-stationary nature of our data sets, that are guaranteed to feature discontinuities, we build our proposed approach on top of Deep Gaussian Processs (DGPs) [2].

In the field of machine learning: (i) *multitask learning* consists of learning several tasks simultaneously while sharing common knowledge in order to improve the prediction accuracy of each task and/or speed up the training process; (ii) *transfer learning* consists in using the knowledge of one (or several) task(s) to improve the learning accuracy and/or the speed of another task. Following these two paradigms, we propose in the present paper the *Multitask Learning Algorithm* (Multitask Learning Autotuning (MLA)) and the *Transfer Learning Algorithms* (Transfer Learning Autotuning (TLA)1 and TLA2). The underlying assumption, which is at the core of these algorithms, is that the objective function to optimize is expected to be continuous or, at least similar, for similar tasks and for similar parameter values.

a) Framework: Let us now define autotuning in the multitask and transfer learning setting. The notations used in this paper are summarized in Table I.

This work was supported by the U.S. Department of Energy, Office of Science, ASCR under Award Number DE-SC0021419 the National Science Foundation under OAC grant No. 2004541.

TABLE I
Symbols used in the text.

Symbol	Interpretation
$\mathcal{S}(\mathbb{T})$	task space
$\mathcal{S}(\mathbb{P})$	parameter space
$\mathcal{S}(\mathbb{O})$	output space (e.g., runtime)
$\mathcal{D}(T)$	dimension of $\mathcal{S}(\mathbb{T})$
$\mathcal{D}(P)$	dimension of $\mathcal{S}(\mathbb{P})$
$\mathcal{D}(O)$	dimension of $\mathcal{S}(\mathbb{O})$
$№ T$	number of tasks
$№ P$	number of samples per task

Throughout the paper, we refer to *input problem* as an input of the target application to be tuned. Moreover, we refer to *task* as the problem of tuning the parameters of the target application given a specific input problem.

We define $\mathcal{S}(\mathbb{T})$, the *Task Space*, as the space of all the input problems that the application may encounter. We also make the simple (non restrictive) assumption that $\mathcal{S}(\mathbb{T})$ may be characterised as a finite dimensional space of dimension $\mathcal{D}(T)$. Several application problems are amenable to such a formalism, potentially after some approximations are made.

We then define $\mathcal{S}(\mathbb{P})$, the *Parameter Space*, or space of the parameters to be optimized, of dimension $\mathcal{D}(P)$, the number of parameters. Every point in $\mathcal{S}(\mathbb{P})$ can be referred to as a *parameter configuration*.

We define $\mathcal{S}(\mathbb{O})$, the *Output Space*, as the space of the results of the evaluation of objective function, for a given task and for a given parameter configuration. It is a single dimensional space (e.g., computation time, memory consumption, energy ...), the case of multi-dimensional spaces (corresponding to multi-objective optimization) being left for future work.

We denote by $y(t,x) \in \mathcal{S}(\mathbb{O})$ and $f(t,x) \in \mathcal{S}(\mathbb{O})$ the values of the objective function y and model prediction f, respectively, for a task $t \in \mathcal{S}(\mathbb{T})$ and for a parameter configuration $x \in \mathcal{S}(\mathbb{P})$. As is the case in Bayesian optimization, the model f is optimized instead of the measured value y, while the optimum found is hoped to be that of y.

In this setting, it is possible to describe the application performance autotuning problem under the mathematical framework of *black-box optimization*. Every evaluation of the objective function is an expensive run of the application and no gradient information is available.

b) Paper contributions: We combine DGP model (instead of GP) with multitask and transfer learning: (i) this allows us to merge $\mathcal{S}(\mathbb{T})$ and $\mathcal{S}(\mathbb{P})$ instead of considering $\mathcal{S}(\mathbb{P})$ only as in traditional approaches; (ii) we perform run-free autotuning and accelerated on-line tuning with pre-trained model available off-line.

- *Multitask learning*: the fact of tuning the application on several tasks simultaneously helps the tuning of each independent task
- *Transfer learning*: the model produced allows for predicting good guesses of the parameters of the application for completely new problems for which no data is available.
- *Independence* of the different optimizations (of the different tasks) allows for a high degree of parallelism, that can easily be exploited. This comes in addition to the parallelism available within the Efficient Global Optimization (EGO) algorithm itself.
- *Optimum* of every task should be close to the optimum of related (close) tasks. Thus, the exploration around the optimum of one task will benefit the neighboring tasks as new data is gathered in the vicinity of their own optimum.
- *Increased confidence* while predicting a given task when taking into account other tasks' contribution for the prediction (decreases variance). This leads the MLA to converge faster to the global optimum.

c) Overview: This paper is organized as follows. Sections II, III and IV describe our proposed algorithms alongside the classic EGO algorithm which is central to our work.Section V compares the performance of our proposed algorithms against that of existing autotuning methods. Section VI summarizes the presented work and describes the potential future directions.

II. Sampling phase

The sampling phase in EGO consists in choosing an initial sample of data with which an initial model can be built. While the subsequent phases aim at selecting candidates that improve upon the best solution found so far. The aim of the sampling phase is not to find optima but rather to choose locations that cover uniformly the search space, to ensure the homogeneous accuracy of the model.

While a single sampling step (over $\mathcal{S}(\mathbb{P})$) is needed in a classical single-task Bayesian optimization scheme, two sampling steps are needed in MLA (over both $\mathcal{S}(\mathbb{T})$ and $\mathcal{S}(\mathbb{P})$).

a) First sampling step: The goal of this step is to select a set T of $№T$ tasks $T = [t_1; t_2; \ldots; t_{№T}] \in \mathcal{S}(\mathbb{T})^{№T}$. This set should contain a representative sample of the variety of problems that the application may encounter, rather than focusing on a specific type of problems. Given the freedom in the selection of the tasks together with the existence of a space of tasks $\mathcal{S}(\mathbb{T})$, we choose a *space filling sampling* in $\mathcal{S}(\mathbb{T})$ to select T. Such samplings are widely used in the field of DoE. Particularly, we choose a *Latin Hypercube Sampling* (LHS) [3]–[5] in MLA. Such samplings try to cover the whole search space uniformly. Several off-the-shelf software packages exist that implement different types of sampling strategies (including LHS). Alternatively, one might opt for a specific strategy to select T or might even provide T altogether.

b) Second sampling step: The of this step is to select an initial sampling X of parameter configurations for every task $X = [X_1; X_2; \ldots; X_{№T}] \in \mathcal{S}(\mathbb{P})^{№T \times №P}$. For task t_i, its initial sampling X_i consists of $№P$ parameter configurations $X_i = [x_{i,j}]_{j \in [1, №P]} \in \mathcal{S}(\mathbb{P})^{№P}$. Two cases arise in the multitask framework: (i) The *isotropic* case, when all the tasks share the same sampling in PS; (ii) The *heterotropic* case, when different tasks do not necessarily share the same samples. In the former case, the advantage of a multi-output regression is the sharing of information for the optimization of the hyper-parameters of the model governing the tasks. In the latter case, however, more knowledge can be shared. Indeed, insights on the true cost of a task on an unknown configuration can be learned from a similar task with a sample at that location. Additionally, in real-life applications, given the existence of constraints on the parameters, not all parameter configurations are feasible for all tasks simultaneously; a configuration may be valid for a subset of tasks, but violate the constraints on another subset. Thus, we choose to generate the initial sampling X in a heterotropic way by generating the X_i as independent LHS.

c) Constraint handling: Given the application constraints, a generic sampling technique might fail, both for the selection of the task samples and for the selection of their corresponding parameter samples. In such a case, either specific knowledge of the application should be used to design a tailored space filling sampling, or a Monte Carlo strategy should be implemented.

d) Samples evaluation: Once T and X are selected, every sample $x_{i,j}$ is evaluated through a run of the application. The set Y represent the results of all these evaluations, $Y = [Y_1; Y_2; \ldots; Y_{№T}] \in \mathcal{S}(\mathbb{O})^{№T \times №P}$, where every Y_i represents the results corresponding to task t_i, $Y_i = [y_{i,j}]_{j \in [1, №P]} \in \mathcal{S}(\mathbb{O})^{№P}$.

III. Model Phase

Once T and X are selected and Y evaluated, the modeling phase consists in training a model of the black-box objective function relative to tasks T. However, instead of building a separate model for every task, as is usually the case in a regular single-task Bayesian optimization scheme, the challenge in MLA is to derive a single encompassing model that allows the sharing of knowledge between tasks in order to better predict them all.

GPs are customarily used in EGO for the modeling in single-task tuning [6]. Moreover, Linear Coregionalization Model (LCM) [7], [8] has already been used as a shallow learning model in multitask tuning [9]–[11]. We propose to rely on DGP in MLA as the generalization of GPs to the deep learning and multitask settings.

The following Sections III-A and III-B describe the GP and DGP models, respectively.

A. Gaussian Processes

We provide here a brief presentation of *Gaussian Processes*. We invite the reader to consult [1] for a detailed description. A Gaussian process is the generalization of a multivariate normal distribution to an infinite number of random variables. It is a stochastic process where every finite subset of variables follows a multivariate normal distribution. While other regression methods set a prior on the function to be predicted, attempting to learn the parameters of such a function, GPs set a prior

on some characteristics of the functions (e.g. smoothness) to learn the functions themselves. This shift in prior allows for the expressiveness of a much richer variety of functions.

A GP is completely specified by its mean function $\mu(x)$ and by its covariance function $k(x, x')$. A function $f(x)$ following such a GP is written as:

$$f(x) \sim GP(\mu(x), k(x, x')) \tag{1}$$

where:

$$\mu(x) = \mathbb{E}[f(x)] \tag{2}$$

$$k(x, x') = \mathbb{E}[(f(x) - \mu(x))(f(x') - \mu(x'))] \tag{3}$$

In most practical scenarios, μ is taken to be the null function and all the modeling is done through the kernel function k. A variety of kernels exist in the literature. The adequate choice depends on the data at hand. The runaway generic choice is the following exponential quadratic kernel:

$$k(x, x') = \sigma^2 \exp\left(- \sum_{i=1}^{\mathcal{D}(P)} \frac{(x_i - x'_i)^2}{l_i} \right) \tag{4}$$

where σ^2 (variance) and l_i (lengthscales) are hyper-parameters governing the behavior of the kernel. These are learned by optimizing the following log-likelihood of the samples X with values y on the GP:

$$\begin{aligned} log(p(y|X)) = &- \frac{1}{2}(y - \mu(X))^T (K + \sigma^2 I)^{-1}(y - \mu(X)) \\ &- \frac{1}{2} log|K + \sigma^2 I| - \frac{n}{2} log(2\pi) \end{aligned} \tag{5}$$

where $\sigma^2 I$ is a regularization term, and K is the covariance matrix whose elements are generated from the kernel k.

Given the high cost of computations as the size of the data increases ($O(N^3)$), several approximate training strategies have been derived. One of the most popular is the *inducing points* approximation. In this method, a set of pseudo data points (of size M) is used in lieu of the original data (of size N) with $M << N$. The location of the pseudo points (inducing inputs) is defined by $Z = \{z_1, \ldots, z_M\}$, whereas the corresponding values (inducing outputs) are defined by $U = f(Z)$.

After defining $f = f(X)$, the joint density of y, f and u is given by:

$$p(y, f, u) = p(f|u; X, Z)p(u, Z) \prod_{i=1}^{N} p(y_i, f_i) \tag{6}$$

Once the model is trained, i.e. its hyper-parameters optimized, it can be queried for a prediction relative to input x^* through the formulation:

$$f(x^*) = K(x^*, X)^T K(X, X)^{-1} Y \tag{7}$$

A fundamental property of GP-based models is the ability to estimate the confidence in the predictions alongside the predictions themselves. The confidence can be expressed as:

$$var(x^*) = K(x^*, x^*) - K(x^*, X)^T K(X, X)^{-1} K(x^*, X) \tag{8}$$

B. Deep Gaussian Processes

a) Model description: A *Deep Gaussian Process* (DGP) [2] is a hierarchical composition of GPs (in our case, a deep stacking). In a DGP of depth L, the GP at layer l models a vector-valued stochastic function f_l of dimension D^l whose input is the output of function f_{l-1} at the previous layer and whose output is the input of function f_{l+1} at the next layer. Then, X is the input of the first layer and Y is the output of the last layer. The noise between layers is assumed to be *i.i.d.* Gaussian, which means that the output produced by a layer is corrupted by a Gaussian noise before being fed to the next layer.

Inference in DGPs is intractable. Indeed, the tractability in GPs stems from the fact that the likelihood is assumed to be Gaussian in such models, which greatly simplifies the computations. Unfortunately, composition of Gaussian probability distributions is in general not Gaussian, which greatly complicates the integration of the probability densities. Hence, the inducing point framework is used. The inducing inputs at every layer are denoted by $Z = \{Z^1, \cdots, Z^L\}$, and the corresponding inducing outputs are denoted by $U = \{U^1 = f^1(Z^1), \cdots, U^L = f^L(Z^L)\}$. These inducing points are then parameters of the model that can be optimized and used to propagate the GPs predictions through the successive layers.

The joint probability density function of Y, F and U can be extended from that of a GP model (Equation 6) as:

$$\begin{aligned} &p(Y, \{F^l, U^l\}_{l=1}^L) = \\ &\quad \prod_{l=1}^{L} p(F^l|U^l; F^{l-1}, Z^{l-1}) p(U^l; Z^{l-1}) \prod_{i=1}^{N} p(y_i|f_i^l) \end{aligned} \tag{9}$$

b) Model training: Two main families of methods exist for training (i.e. hyper-parameter optimization) and inference (i.e. prediction) in DGPs: *Variational Inference* (VI) [12] and *Markov Chain Monte Carlo* (MCMC) [13]. The difficulty with DGPs is the existence of complex correlations both within and between layers.

The original paper on DGPs [2] relies on a mean-field variational approach that uses a variational posterior which maintains the exact model conditioned on the inducing outputs but forces independence between layers. A consequent advantage is that this approach admits a tractable lower bound on the log marginal likelihood (under some assumptions). A drawback, however, is that the probability density over the outputs is merely a single-layer GP with independent Gaussian inputs and therefor cannot express the complexity of the full model.

The *Doubly Stochastic Variational Inference* (DSVI) [14] relies on a double source of stochasticity: (i) a sparse inducing point variational inference scheme [15] is used to simplify the correlations and achieve computational tractability within each layer. However, the correlations between layers are maintained (independence is not forced as is the case in previous methods). This leads to a sacrifice of the analytic tractability of the variational lower bound. However, this is

overcome by the ability to draw efficiently unbiased samples from the variational posterior which has the same structure as the exact model conditioned on the inducing points. (ii) a minibatch subsampling of the data is used to scale to extremely large datasets (up to a billion data points).

The *Stochastic Gradient Hamiltonian Monte Carlo* (SGHMC) [16], [17] method can be used to generate samples from the intractable posterior distribution of the inducing outputs $p(U, Y)$. The underlying idea is to rely on the principles of Hamiltonian dynamics by trying to minimize a total energy of a dynamic system described as:

$$p(U, R|Y) \propto \exp\left(-U(u) - \frac{1}{2} r^T M^{-1} r\right) \tag{10}$$

where the negative log posterior $U(u) = -log(p(U|Y))$ plays the role of the potential energy, and the $\frac{1}{2} r^T M^{-1} r$ term plays the role of the kinetic energy, where r is an artificially introduced momentum variable.

Given that the computation of gradients in the *Hamiltonian Monte Carlo* (HMC) [18] method is intractable in DGP in the case of large training data, stochastic gradient estimates can be computed following the work of [17], where:

$$\Delta u = \epsilon M^{-1} r \tag{11}$$

$$\Delta r = -\epsilon \nabla U(u) - \epsilon C M^{-1} r + \mathcal{N}(0,\, 2\epsilon(C - B)) \tag{12}$$

where C is the friction term (introduced to allow for batched computations), M is the mass matrix, B is the Fisher information matrix and ϵ is the step-size.

Given this sampling framework, a Markov Chain can be built. However, due to the high correlation between successive samples, the optimization of the hyper-parameters of the model while the sampler progresses is an operation likely to fail. As a remedy, the authors in [16] propose a variant of the *Monte Carlo Expectation Maximization* (MCEM) [19] that they call *Moving Window Markov Chain Expectation Maximization* (MWEM). While MCEM alternates between the sampling from the posterior (E-step, Eq 13) and the maximization of the joint probability of the samples and the data (M-step, Equation 14),

$$u_{1,\ldots,m} \sim p(u|Y, X, \theta) \tag{13}$$

$$\theta = \arg\max_{\theta} \frac{1}{m} \Sigma_{i=1}^{m} \log p(Y, u_i|X, \theta) \tag{14}$$

MWEM maintains a window of samples, where a newly generated sample replaces the oldest one. Additionally, the number of samples in the E-step is set to $m = 1$. At every M-step, a random sample from the window is selected and the hyper-parameters θ of the model are optimized with respect to this sample only. Experimentally, MWEM is able to converge faster than alternate methods such as DSVI.

While DSVI is often used for training DGPs, we choose SGHMC instead. It makes no assumptions regarding the kernels and the likelihood being used. Moreover, the accuracy of the model can be controlled and traded off with computational training time, which is not necessarily the case with other training methods. These two characteristics are advantageous in a practical setting such as autotuning. The pros of DGPs make them a great tool for tuning exascale applications, while their usual computational drawbacks are leveraged in the exascale setting where few samples only can be collected, making the DGP model tractable (albeit with some approximations).

IV. Search phase

A. Acquisition function optimization

Once the Bayesian model is either built (at the first iteration) or updated (at subsequent iterations), the EGO algorithm relies on a model-free black-box optimization algorithm to optimize a quantity called *acquisition function*. This latter is based on both the prediction and the confidence of the model in the outcome of the black-box objective function at different locations in the search space. In our multitask setting, not one but many such optimizations need to be carried out. Given their relative independence, they can occur in parallel. The *TLA*2 strategy we propose is not an optimization of an acquisition function but rather a direct optimization of the mean prediction of the model. The underlying idea is that there are not enough runs available to balance exploration and exploitation, as the goal is only to exploit previous data. This strategy is meant for the transfer learning setting, when the user is not interested in tuning the application on a completely new input problem but is rather interested in leveraging the data and model built on a previous tuning of the application.

B. Fast online prediction of the optima

In a practical setting, after spending enough time offline in tuning the application on a variety of relevant input problems, one can rely on *TLA*2 to guess the best parameter values of the application on a new problem. However, although the optimization queries a DGP model that is much cheaper to query than the real application, the search phase can take tens of seconds to minutes. When such waiting times are unacceptable, we propose a quick online prediction strategy, *TLA*1, that can return a good guess within a fraction of a second.

*TLA*1 applies if an approximation to the optimum parameter configuration of a new task is considered enough, or, if a specific tuning for that task cannot be afforded. It consists of building a model of the optima of any new unexplored tasks, for which no data is available. Specifically, a Gaussian Process model predicting the optima is built over $\mathcal{S}(\mathbb{T})$ and is trained on the parameter configuration of the optimum found for every task t_i in T.

Let us define the set of optima OPT corresponding to the set of tasks T as:

$$OPT_i = \arg\min_{x \in \mathcal{S}(\mathbb{P})} f(t_i, x), \forall i \in [1, № T] \tag{15}$$

An optimum parameter configuration is composed of as many parameters as $\mathcal{D}(P)$. Consequently, the solution that we propose is to create $\mathcal{D}(P)$ separate and independent Gaussian Processes $G_{i \in [1, \mathcal{D}(P)]}$ to model every component of the optimal solutions separately. Such a GP model is described in Section III-A. The set OPT represents the input data of

every one of these GPs. It is important to notice that the input and output spaces of the GPs in EGO and in *TLA*1 are different. In the former case, the input space is $\mathcal{S}(\mathbb{P})$ and the output space is $\mathcal{S}(\mathbb{O})$. In the latter case, the input space is $\mathcal{S}(\mathbb{T})$ and the output space is one of the dimensions of $\mathcal{S}(\mathbb{P})$. For any unexplored task t^*, a prediction of its optimal parameter configuration is then given by:

$$OPT(t^*) = [G_1(t^*), G_2(t^*), \ldots, G_{\mathcal{D}(P)}(t^*)] \in \mathcal{S}(\mathbb{P}) \quad (16)$$

Additionally, the confidence in the prediction of the G_i models can serve as an indicator on whether an extra tuning step is needed. An alternative solution could be to define a single multi-output GP to model all the components simultaneously. However, no *a priori* hypothesis can be made in the general case on the correlation between the different components characterising the optima.

V. Results and discussion

This section presents the experimental results assessing the combined advantage of a DGP model in a multitask and transfer learning setting.

A. Experimental setting

1) Hardware Setup: Two computers are used for the experiments: *ALPHA* is a 64 nodes computer, each containing 12 cores of Intel$^{(R)}$ Xeon$^{(R)}$ X5660 at 2.80GHz. *BETA* is a single-node computer, containing 40 cores of Intel$^{(R)}$ Xeon$^{(R)}$ CPU E5-2650 v3 at 2.30GHz.

2) Optimization Problems: Two applications are considered: the *DGEQRF* routine of the *PLASMA* library and the *DGEMM* routine of the *SLATE* library.

The first problem aims at computing the QR factorization of a rectangular matrix on shared-memory computers. The task space is characterized by m and n, the number of rows and columns of a matrix, together with n_{th}, the number of OpenMP threads to be used in the computation. The parameter space is characterized by n_b and i_b, the block and internal panel sizes. The output space is simply described by the resulting GFlops of the application. The range of sizes of the matrices considered (together with the parameters to be tuned) is $[1, 5000]$ and the range of number of threads is $[1, 40]$. 100 tasks are used for training and 100 others are used for testing. The budget of number of runs per task is 30, split equally between the sampling phase and the remaining optimization phase.

The second problem aims at computing the multiplication of two matrices on distributed-memory computers. The task space is described by m, n and k, the sizes of the matrices to be multiplied, together with the number of nodes to be used for the computations. The parameter space is described by the block size n_b, the number of threads n_{th} in each MPI process, the ratio $p \times q$ of number of processes per row versus column in the 2D block cyclic process mapping, and the number la of lookahead panels to be prefetched and precomputed. The range of matrix sizes is $[1, 10000]$ and the range of compute nodes is $[1, 64]$. The range of number of threads is $[1, 12]$, the range of the $p \times q$ parameter is $[0, 1]$ and of la is $[0, 2]$. In order to get a fair comparison between different runs, the number of threads can only be $1, 2, 3, 4, 6, 12$, which guarantees that all cores of all nodes are used in all runs. 100 tasks are used for training and 100 others are used for testing. The budget of number of runs per task is 50, split equally between the sampling phase and the remaining optimization phase.

For both *DGEQRF* and *DGEMM* problems, the number of runs per task follows a simple rule of thumb of five times the number of parameters for the initial sampling phase and as many runs for the subsequent optimization iterations. This allows for enough runs for the tuners to find relevant results while keeping the budget of runs sufficiently low for the whole tuning process to be attractive in a real life setting. It is likely that, given a large enough number of runs, all tuners would likely converge to an optimum. However, the goal of autotuning is to reach such a goal at reduced cost. Moreover, the choice of number of tasks (100 for *DGEQRF* and 100 in *DGEMM*) follows a similar argument of having enough tasks to enrich the DGP model with enough data to make it able to predict the behaviour of the application on new unknown tasks. At the same time, we must keep the number of tasks small enough so that the total wallclock time of the tuning process remains attractive. The average autotuning time in our experiments for most tuners (given the chosen numbers of samples and tasks) is about 24 hours for the *DGEQRF* problem and 48 hours for the *DGEMM* problem.

3) Autotuner Types: Our proposed algorithms are compared against several autotuning techniques: OpenTuner [20] and HpBandSter [21], two general purpose model-free autotuners , and EGO with a GP model at its core (EGO-GP).

B. Experimental results

1) Multitask learning setting (MLA): The initial experiments consist in tuning a set of training tasks in a multitask learning setting while building a DGP model to be used subsequently for transfer learning. While *MLA* attempts to tune all training tasks simultaneously, OpenTuner, HpBandster and EGO-GP operate on singe-tasks independently. Figures 1 and 2 show the results for the *DGEQRF* and *DGEMM* problems, respectively. The result of a given tuner on a given task is the smallest runtime of the application obtained within the multiple runs attempted. In each figure, the black horizontal line (on y-axis at 1) is the reference result obtained by *MLA* for every test task represented on the X-axis. The tasks are sorted in each sub-figure in increasing ratio of performance of the compared tuner with *MLA*. Each sub-figure compares the best performance found with a given tuner (colored bars) with that found by *MLA* (black horizontal line). Graphically, the metric of success of *MLA* compared to the others is the number of times a colored bar is below the black horizontal line, the more often the better.

On both the *DGEQRF* and *DGEMM* cases, *MLA* is able to bring better application performance in average than the competing methods.

2) Transfer learning setting (TLA1 and TLA2): After training the DGP model on the training tasks, new test tasks are

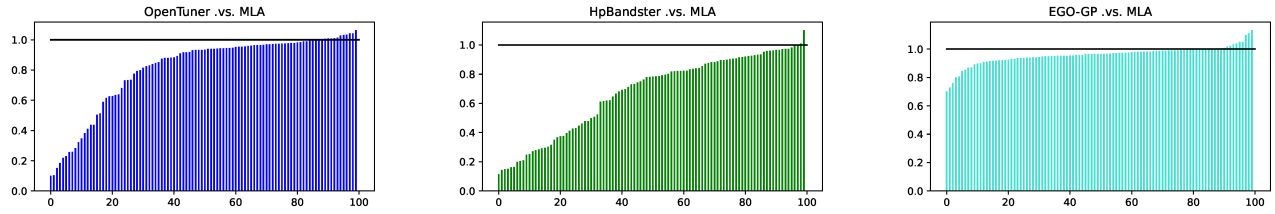

Fig. 1. Relative performance of *DGEQRF* on train tasks given the parameters optimized by OpenTuner, HpBandSter and EDO-GPcompared to the ones found by *MLA* (black horizontal lines)

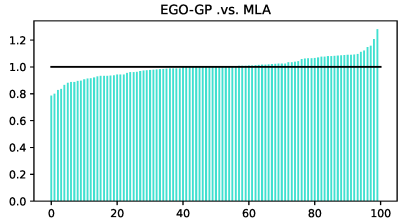

Fig. 2. Relative performance of *DGEMM* on train tasks given the parameters optimized by EGO-GP (turquoise)compared to the ones found by *MLA* (black horizontal lines)

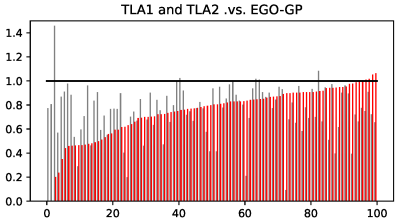

Fig. 4. Relative performance of *DGEMM* on test tasks given the parameters predicted by *TLA*1 (grey) and *TLA*2 (red) using the DGP model compared to the ones optimized by EGO-GP (black horizontal lines)

presented to the DGP-based autotuner. For each test task in each of the two problems, the standard EGO-GP is used similarly to the case of the training tasks. The best result (GFlops) obtained for each test task is then considered as the reference value. Then, the *TLA*1 and *TLA*2 methods are applied to predict the optimal parameter for each test task, but without ever running the applications on them. Figures 3 and 4 show the comparative results for the *DGEQRF* and *DGEMM* problems, respectively. In each figure, the black horizontal line (on y-axis at 1) is the reference result obtained by EGO-GP for every test task represented on the X-axis. Moreover, the grey and red bars represent the results for the *TLA*1 and *TLA*2 methods, respectively. The tasks are sorted by increasing performance ratio of *TLA*2 compared to EGO-GP.

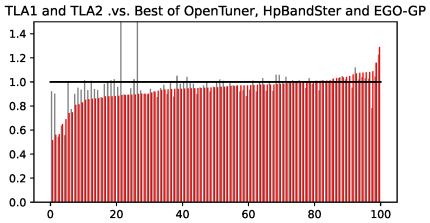

Fig. 3. Relative performance of *DGEQRF* on test tasks given the parameters predicted by *TLA*1 (grey) and *TLA*2 (red) using the DGP model compared to the ones optimized by EGO-GP (black horizontal lines)

The surprising result is that, on the *DGEQRF* problem, not only *TLA*1 is able to outperform *TLA*2, but it is also competitive with the classical EGO-GP. On the other hand, as expeted, *TLA*2 presents better results than *TLA*1 on the *DGEMM* problem.

It is important to notice that there is a clear imbalance in the results. Indeed, *TLA*1 and *TLA*2, compared to EGO-GP, yield a dreadful performance on some tasks and better performance on other tasks. However, the average performance offered by *TLA*2 compared to the reference EGO-GP is about 80% (on both *DGEQRF* and *DGEMM* problems). This result showcases the viability of transfer learning through the DGP model. Moreover, the DGP model together with the results of *TLA*1 and *TLA*2 can be used as a starting point for an additional tuning step on the test tasks, if the user of an application can afford to spare some additional runs to improve the parameters tuning.

As a final note, we must highlight the potential drawback of our work. The fundamental assumption we made by relying on multitask and transfer learning when autotuning an application is that the behaviour of the application is somehow similar on loosely similar problems. This similarity can be evaluated by the distance between two points in the input problem space, each representing a different problem. If an application fails to exhibit the assumed behavior, the use of multitask and transfer learning should not bring any benefit compared to tuning different tasks separately.

VI. Conclusion

This paper introduced multitask learning and transfer learning as effective frameworks for autotuning HPC applications. At the heart of this work is the use of the powerful deep Gaussian process Bayesian model, which is able to identify the relationships between tantamount autotuning tasks. The proposed autotuning approach is based on the classical EGO algorithm. This approach adapts autotuning in two ways: (i) multitask learning: an application is tuned not on one but multiple input problems. The sampling phase incorporates an additional sampling step to choose which problems to tune, then applies the usual sampling phase for each chosen problem. The modeling phase is the same but relies on the DGP model instead of the usual GP. The search phase is applied in parallel on each chosen input problems; (ii) transfer learning: after the tuning phase has been carried on a set of training tasks, the DGP model can be used to predict a good guess of the parameters for a new unknown test task. This is done simply by applying the search phase on the new test task (*TLA*2). In the case where a quick prediction is needed, an approximation is proposed (*TLA*1) that uses a simple GP regression of the best parameters found for all training tasks.

REFERENCES

[1] C. E. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[2] A. Damianou and N. Lawrence, "Deep gaussian processes," *Artificial Intelligence and Statistics*, pp. 207–215, 2013.

[3] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics. American Statistical Association.*, vol. 21, no. 2, pp. 239–245, May 1979.

[4] V. Egl_ajs and A. P., "New approach to the design of multifactor experiments," *Problems of Dynamics and Strengths*, vol. 35, pp. 104–107, 1977.

[5] R. L. Iman, J. C. Helton, and J. E. Campbell, "An approach to sensitivity analysis of computer models, part 1. introduction, input variable selection and preliminary variable assessment," *Journal of Quality Technology*, vol. 13, no. 3, pp. 174–183, 1981.

[6] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of NIPS*, 2012.

[7] A. G. Journel and C. J. Huijbregts, *Mining Geostatistics*. London: Academic press, 1978.

[8] P. Goovaerts, "Geostatistics for natural resources evaluation," 1997.

[9] K. Swersky, J. Snoek, and R. P. Adams, "Multi-task bayesian optimization," in *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper/2013/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf

[10] W. M. Sid-Lakhdar, J. W. Demmel, X. S. Li, Y. Liu, and O. Marques, "Gptune users guide," 2020, https://github.com/gptune/GPTune/tree/master/Doc.

[11] Y. Liu, W. M. Sid-Lakhdar, O. A. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li, "Gptune: Multitask learning for autotuning exascale applications," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 234–246. [Online]. Available: https://doi.org/10.1145/3437801.3441621

[12] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American Statistical Association*, vol. 112, no. 518, p. 859–877, Apr 2017. [Online]. Available: http://dx.doi.org/10.1080/01621459.2017.1285773

[13] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine Learning*, vol. 50, no. 1, pp. 5–43, Jan 2003. [Online]. Available: https://doi.org/10.1023/A:1020281327116

[14] H. Salimbeni and M. Deisenroth, "Doubly stochastic variational inference for deep gaussian processes," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/8208974663db80265e9bfe7b222dcb18-Paper.pdf

[15] A. G. d. G. Matthews, J. Hensman, R. Turner, and Z. Ghahramani, "On sparse variational methods and the kullback-leibler divergence between stochastic processes," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, A. Gretton and C. C. Robert, Eds., vol. 51. Cadiz, Spain: PMLR, 09–11 May 2016, pp. 231–239. [Online]. Available: https://proceedings.mlr.press/v51/matthews16.html

[16] M. Havasi, J. M. Hernández-Lobato, and J. J. Murillo-Fuentes, "Inference in deep gaussian processes using stochastic gradient hamiltonian monte carlo," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 7517–7527.

[17] T. Chen, E. B. Fox, and C. Guestrin, "Stochastic gradient Hamiltonian Monte Carlo," in *International Conference on Machine Learning*, 2014, pp. 1683–1691.

[18] R. M. Neal, "MCMC using Hamiltonian dynamics," *Handbook of Markov Chain Monte Carlo*, vol. 54, pp. 113–162, 2010.

[19] G. C. Wei and M. A. Tanner, "A Monte Carlo implementation of the em algorithm and the poor man's data augmentation algorithms," *Journal of the American Statistical Association*, vol. 85, no. 411, pp. 699–704, 1990.

[20] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: an extensible framework for program autotuning," in *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada: Association for Computing Machinery, 2014, p. 303–316, http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf.

[21] S. Falkner, A. Klein, and F. Hutter, "BOHB: robust and efficient hyperparameter optimization at scale," in *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, J. Dy and A. Krause, Eds., vol. 80, PMLR, Stockholmsmässan, Stockholm Sweden, 2018, pp. 1437–1446, http://proceedings.mlr.press/v80/ falkner18a.html.