


Combining multitask and transfer learning with deep Gaussian processes for autotuning-based performance engineering

The International Journal of High Performance Computing Applications
2023, Vol. 0(0) 1–16
© The Author(s) 2023
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420231166365
journals.sagepub.com/home/hpc


Piotr Luszczek¹ , Wissam M Sid-Lakhdar¹ and Jack Dongarra^{1,2,3}

Abstract

We combine deep Gaussian processes (DGPs) 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 (GPs) 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. In this article, we combine DGPs with multitask and transfer learning to allow for both an improved tuning of an application parameters on problems of interest but also the prediction of parameters on any potential problem the application might encounter.

Keywords

Gaussian process regression, performance autotuning, Efficient Global Optimization, Linear Coregionalization Model, Latin Hypercube Sampling

Introduction

Motivation

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. Another aspect of this type of autotuning that we seek to remedy is ability to transfer the trained models between application and supercomputing platforms to leverage the inherent correlations in the proposed multitask setting. Furthermore, we also aim at utilizing multiple sources of data such as application performance metrics and low-level hardware metrics to inform the multi-fidelity modeling.

Our work is motivated by *low-data regime* that precludes the use of artificial neural networks (ANNs) which need large data volumes to successfully generalize. Might be just me, but this and the following sentence/idea seem like they should be swapped in order. Gaussian process (GP) (Rasmussen and Williams, 2006) 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 processes (DGPs) (Damianou and Lawrence, 2013). This allows to produce reusable models that may be used for predictions for unknown tasks. In the future, we plan to provide reusable models for higher-level or application prediction beyond performance metrics and more along the lines of behavior and anomaly detection.

Solution

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 and (ii) *transfer learning* consists in using the

¹The University of Tennessee, Knoxville, TN, USA

²Oak Ridge National Laboratory, Oak Ridge, TN, USA

³University of Manchester, Manchester, UK

Corresponding author:

Piotr Luszczek, The University of Tennessee College of Engineering,
1122 Volunteer Blvd. #203, Knoxville, TN 37996 USA.
Email: luszczek@icl.utk.edu

Table 1. Definition of 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)
$\mathcal{D}(\mathbb{T})$	Dimension of $\mathcal{S}(\mathbb{T})$
$\mathcal{D}(\mathbb{P})$	Dimension of $\mathcal{S}(\mathbb{P})$
$\mathcal{D}(\mathbb{O})$	Dimension of $\mathcal{S}(\mathbb{O})$
$\mathcal{N}^{\circ} T$	Number of tasks
$\mathcal{N}^{\circ} P$	Number of samples per task

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 article, the Multitask Learning Autotuning (MLA) and two variants of the Transfer Learning Autotuning (TLA) (first *Transfer Learning Algorithm* (TLA1) and second *Transfer Learning Algorithm* (TLA2)). Given the knowledge gathered and model built on previously autotuned representative tasks, we would like to be able to predict an optimal, or at least good enough, parameter configuration on a new unknown task. 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.

Framework

Multitask learning and transfer learning for autotuning. Let us now define autotuning in the multitask and transfer learning setting. The notations used in this article are summarized in Table 1.

Throughout the article, 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 characterized as a finite dimensional space of dimension $\mathcal{D}(\mathbb{T})$. This means that it is possible to identify any input problem with a finite number of features. Several application problems are amenable to such a formalism, potentially after some approximations are made. If this finite dimension task space assumption does not hold, the multitask learning methods in Section “Model phase” remain valid, but some of the transfer learning methods would require additional attention that we leave as future work (e.g., space of all possible sparse matrices).

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

either real, integer, or categorical (e.g., a list of n algorithms (which can map to an interval $[1, n]$)). In our work, all parameter kinds are translated internally to the real case in the normalized interval $[0, 1]$. Moreover, most applications have constraints on their parameters as not all possible combinations of parameters (point in the parameter space) are valid.

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, and 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. The fundamental aspect of autotuning is optimization, that is, finding a configuration of the parameters of an application that makes it solve a given input problem optimally. Indeed, given a task $t \in \mathcal{S}(\mathbb{T})$, the autotuning goal is to find

$$\arg \min_{x \in \mathcal{S}(\mathbb{P})} f(t, x) \quad (1)$$

The nature of autotuning makes this optimization problem lie within the family of black-box optimization problems, which are among the hardest to solve. Indeed, an expensive run of the application is necessary in order to get the value of the objective function to optimize (e.g., runtime and energy) for a given combination of parameters. Moreover, even in the presence of approximate (coarse) analytical models of the application (e.g., flop count and memory consumption), no precise enough formulation generally exists to predict more detailed phenomena (e.g., memory hierarchy communications and network contention) influencing the application behavior. Furthermore, the human cost of deriving such models (if even possible to derive) is too expensive to be practical. Similarly, no analytical information on the gradient of the objective function is available, but only numerical approximations are, through sample evaluations of the objective function.

The autotuning stopping criteria is classically defined as one or a combination of the following: (i) a maximum number of evaluations of the objective function (runs of the application); (ii) a maximum wall-clock time; (iii) threshold on a relevant measure of the quality of the solution provided by the application (e.g., numerical accuracy and energy

consumption). In order to be able to fairly compare different autotuning algorithms, we chose to fix a maximum number of runs of the application as the stopping criteria. The total runtime spent in the application is also measured and reported.

Article 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 online tuning with pre-trained model available offline.

- *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.

Overview

This article is organized as follows. Section “Related work and proposed approach” presents the literature on autotuning. It contrasts the existing work on shallow learning versus deep learning as well as single-task versus multitask learning. Moreover, it describes the motivation and sketch of our proposed methodology. Sections “Sampling phase”, “Model phase”, and “Search phase” and describe our proposed algorithms alongside the classic EGO algorithm which is central to our work. Section “Experimental results” compares the performance of our proposed algorithms against that of existing autotuning methods. Section “Conclusion” summarizes the presented work and describes the potential future directions.

Related work and proposed approach

For a thorough survey of autotuning, multitask learning, and transfer learning, we refer the reader to [Balaprakash et al.](#)

(2018), [Zhang and Yang \(2017\)](#), and [Pan and Yang \(2010\)](#), respectively. Optimization methods that take into account application-specific knowledge are left aside to keep the focus on general-purpose approaches.

Sections “Single task optimization with deep learning”, “Single-task optimization with shallow learning” and “Multitask optimization with shallow learning” present the works on single-task shallow learning, single-task deep learning and multitask shallow learning, respectively. Specifically, black-box optimization techniques can be grouped in two categories: Section “Model-free optimization” describes *Model-free* methods, which are used in *OpenTuner* ([Ansel et al., 2014](#)) and *HbBandSter* ([Falkner et al., 2018](#)), two state-of-the-art autotuners that we compare our results against. Section “Model-based optimization” describes *Model-based* methods that comprise the backbone of the optimization method of significance to our work. The present work on deep Bayesian optimization in multitask and transfer learning settings is described in Section “Multitask optimization with shallow learning”.

Single-task optimization with shallow learning

The simplest black-box optimization methods, that are typically first attempted before resorting to more advanced methods, are known as: (i) The deterministic *exhaustive search* (resp. its variant *grid search*) which tries all (resp. a subset of all) possible combinations of parameter values and selects the best performer. The drawback is that these methods quickly become intractable when the number of parameters increases (a.k.a. curse of dimensionality ([Bellman, 1957](#))); (ii) The stochastic *random search*, which tries random combinations of parameter values to generate candidate solutions, then selects the best performing one.

Model-free optimization. Two main families of model-free optimization approaches exist. The *deterministic* approaches, such as *Nelder-Mead simplex* ([Nelder and Mead, 1965](#)) and *Orthogonal Search* ([Chan et al., 2011](#)), make improvements over the previous solutions by exploring their neighboring region until convergence to a local minimum. The *stochastic* approaches, such as Simulated Annealing (SA) ([Kirkpatrick et al., 1983](#)), Genetic Algorithms (GAs) ([Srinivas and Patnaik, 1994](#)), and Particle Swarm Optimization (PSO) ([Kennedy and Eberhart, 1995](#)), try to find a balance between exploiting the vicinity of the data gathered and exploring new promising regions of the search space.

OpenTuner. *OpenTuner* ([Ansel et al., 2014](#)) and *HbBandSter* ([Falkner et al., 2018](#)) are two state-of-the-art general-purpose autotuning frameworks. They rely on meta-heuristics to solve a multi-armed bandit problem ([Katehakis and Veinott, 1987](#)) where application runtime (our case) is the resource to be allocated. *OpenTuner*'s philosophy is that no optimization algorithm can be better

than all other algorithms all the time (a.k.a. *no free lunch* theorem). It allocates and distributes application runtime over a variety of optimization methods (mentioned in the previous paragraphs) in such a way as to adaptively select the best performing one method to solve the autotuning optimization problem. HpBandSter's algorithm mixes a Bayesian Optimization method (see the following section) with that of *Hyperband* (Li et al., 2017), an early-stopping method that allocates application runtime uniformly over randomly sampled configurations, keeping only the half best performing configurations at each iteration, and extending the corresponding runtime per remaining configurations, until a single (best) configuration remains.

Random-tree search. Several other non-Bayesian black-box optimization packages for autotuning exist in the literature. Notably, *SuRf* (Balaprakash, 2015) uses random forests to discover regions of interest for further exploration. One of its main strengths is its ability to elegantly handle categorical parameters (choices).

Model-based optimization. Any kind of statistical model can be used in model-based optimization methods. However, *Bayesian optimization* (Shahriari et al., 2016), a.k.a. *response surface methodology*, is unique in that it relies on a Bayesian (probabilistic) surrogate model of the actual objective function. A *prior*, representing the assumptions on the objective function, is chosen and a *posterior* is built from it so as to maximize the likelihood of some sample data of the objective function to be a realization of that model. Instead of directly optimizing the true objective function, the model is optimized over instead (as it is much cheaper to evaluate) and iteratively updated until convergence to an optimum.

The EGO algorithm (Jones et al., 1998) is a classical Bayesian optimization algorithm. It is composed of three phases: (i) sampling phase (Section "Sampling phase"); (ii) modeling phase (Section "Model phase"); and (iii) search/optimization phase (Section "Search phase"). Once the first phase completes, the second and third are repeated until convergence or stopping of the tuning process. Specifically, EGO starts by selecting an initial sample of data in order to build an initial model. Then, the search phase seeks to find a new candidate location to explore. In order to balance between exploration and exploitation, EGO tries to optimize a certain acquisition function (e.g., *Expected Improvement* (EI) (Qin et al., 2017)) instead of the model itself. This metric considers the value of the model at a given point in the search space together with the confidence of the model at that location. For instance, if the model predicts a good value of the objective function with a high confidence at a given location, while a worse value at another location but with a lower confidence, it might be worth exploring this second location as an optimum might be located nearby. After finding the candidate location, it is evaluated through a

call to the expensive black-box objective function. This value is used to update the surrogate model. Efficient Global Optimization iterates between the model phase and the search phase until the acquisition function reaches a certain threshold, at which point the algorithm is considered to have converged to an optimum. Several effective autotuners, such as *Spearmint* (Snoek et al., 2012) and *HyperOpt* (Bergstra et al., 2015), implement a version of EGO.

Efficient Global Optimization being the backbone of our work, it is described in detail in Sections "Sampling phase", "Model phase", and "Search phase", alongside our contributions.

Single-task optimization with deep learning

Recent studies introduced the combined use of deep Bayesian models with black-box optimization. Wang and Shan (2007) apply this approach in the field of *Design of Experiments* (DoE) (Wang and Shan, 2007) for civil engineering applications. Additionally, Hebbal et al. (2018) and Hebbal et al. (2021) combine DGP with EGO to circumvent the inherent limitations of GP in the modeling of non-stationary and discontinuous objective functions arising in the field of design optimization for aerospace applications. The authors extend their line of work to the case of constrained multi-objective design optimization of aerospace vehicles in Hebbal et al. (2019). Furthermore, Rajaram et al. (2021) assesses the benefit of using DGP, compared to GP, as a surrogate model in optimization problems relative to both canonical tests and an airfoil design problem, focusing on the accuracy of the models and their training cost.

Multitask optimization with shallow learning

The generalization of GP from real-valued functions (single-task setting) to vector-valued functions (multitask setting) through the Linear Coregionalization Model (LCM)¹ has originally found its use in geostatistics (Journel and Huijbregts, 1978 and Goovaerts, 1997), but gained renewed interest in machine learning (Seeger et al., 2005, Bonilla et al., 2008, and Alvarez and Lawrence, 2011). For instance, Swersky et al. (2013) combine LCM with a search strategy relying on an entropy-based acquisition metric (Hennig and Schuler, 2012) on the *information gain per unit cost* to achieve an efficient multitask optimization. Moreover, Liu et al. (2021) rely on LCM in the GPTune package as the core model in a dual multitask and multi-objective optimization setting.

Multitask optimization with deep learning

The present section is a sketch of our proposed autotuning methodology, while Sections "Sampling phase", "Model phase" and "Search phase" dive into further details.

Tuning an application for a specific input problem makes that application efficient on solving that particular case. However, it offers no guarantee regarding its behavior on other unprecedented cases. Our aim is to tune an application for any problem it might encounter. This goal is potentially unrealistic in general, given the infinity of input problems that might exist. Therefore, we wish to expeditiously find favorable combinations of parameters of the application for any given input problem. The fact of tuning an application on the whole space of input problems instead of a single one opens up the door for leveraging multitask and transfer learning. We believe that tuning the application on each problem independently from the others is less efficient than tuning all of them simultaneously. This allows to benefit from the gathered knowledge on all input problems to speed up the whole tuning process. This is critical especially in an exascale setting, where every run of the application is extremely expensive. The tuning process then follows a traditional machine learning approach of model training followed by model inference, where the result of the tuning in the multitask learning setting is a sufficiently powerful DGPs model that then can be exploited in the subsequent transfer learning setting. In contrast to other multitask models such as LCMs, a DGP model is naturally able to render predictions not only on the current tasks being tuned but also on future unknown tasks, which is key to the subsequent transfer learning setting.

The guiding ideas behind our work are twofold. First, our *Multitask Learning Algorithm* (MLA) is an adaptation of EGO to the multitask learning setting and relies on DGP as its core model. The sample phase (Section “Sampling phase”) encompasses not only the space of parameters but also the space of input problems. A finite set of input problems is selected, to every element of which a standard sampling is carried in the corresponding problem space. The model phase (Section “Model phase”) merges all gathered data relative to all sampled input problems into a single powerful DGPs model. While the complexity of the model training grows quadratically with data size, this additional cost is balanced by the improved prediction accuracy due to the sharing of information among input tasks. The search phase (Section “Search phase”) boils down to parallel standard EGO search phases over the set of sampled input tasks. Second, our TLA1 is a fast online tuning method that relies solely on the set of optimal parameters found on the initial sampled tasks in order to predict the optimal parameters of a new task at hand through a simple GP regression. Our TLA2 takes full advantage of the final DGPs model obtained at completion of MLA by querying it in a similar way as in the search phase of EGO in order to identify the best candidate solution for unobserved tasks. A major difference however is that, while a balance of exploration and exploitation is necessary in a standard EGO search phase, a pure exploitation strategy is needed in TLA2 as there is only a single possible attempt at finding the optimum parameters.

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 $\mathcal{S}(\mathbb{P})$).

First sampling step

The goal of this step is to select a set \mathcal{T} of \mathcal{N}^o T tasks $\mathcal{T} = [t_1; t_2; \dots; t_{\mathcal{N}^o T}] \in \mathcal{S}(\mathbb{T})^{\mathcal{N}^o T}$. 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 \mathcal{T} . Such samplings are widely used in the field of DoE. Particularly, we choose a Latin Hypercube Sampling (LHS) (McKay et al., 1979; Eglajs and Audze, 1977; and Iman et al., 1981) 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 \mathcal{T} altogether.

Second sampling step. The aim of this step is to select an initial sampling \mathcal{X} of parameter configurations for every task $\mathcal{X} = [X_1; X_2; \dots; X_{\mathcal{N}^o T}] \in \mathcal{S}(\mathbb{P})^{\mathcal{N}^o T \times \mathcal{N}^o P}$. For task t_i , its initial sampling X_i consists of \mathcal{N}^o P parameter configurations $X_i = [x_{i,j}]_{j \in [1, \mathcal{N}^o P]} \in \mathcal{S}(\mathbb{P})^{\mathcal{N}^o P}$. Two cases arise in the multitask framework: (i) The *isotropic* case, when all the tasks share the same sampling in PS ; and (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.

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. Such a situation arises frequently when the total number of combinations of parameter values is of the order of thousands of billions (curse of dimensionality) while the number of valid parameter configurations that respect the constraints is only of the order of hundreds of thousands. An example is the tuning of matrix multiplication on GPU in MAGMA (Anzt et al., 2015 and Nath et al., 2010a; 2010b). 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. In practice, this strategy may lead to a very large number of iterations before generating enough satisfiable samples. Thus, in MLA, at every iteration, we double the number of samples we generate. Consequently, the cost of the sampling algorithm adapts to the complexity of the constraints.

Samples evaluation. Once \mathcal{T} and \mathcal{X} are selected, every sample $x_{i,j}$ is evaluated through a run of the application. The set \mathcal{Y} represent the results of all these evaluations, $\mathcal{Y} = [Y_1; Y_2; \dots; Y_{N^o T}] \in \mathcal{S}(\mathbb{O})^{N^o T \times N^o P}$, where every Y_i represents the results corresponding to task t_i , $Y_i = [y_{i,j}]_{j \in [1, N^o P]} \in \mathcal{S}(\mathbb{O})^{N^o P}$.

Model phase

Once \mathcal{T} and \mathcal{X} are selected and \mathcal{Y} evaluated, the modeling phase consists in training a model of the black-box objective function relative to tasks \mathcal{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 (Snoek et al., 2012). Moreover, LCM (Journel and Huijbregts, 1978 and Goovaerts, 1997) has already been used as a shallow learning model in multitask tuning (Swersky et al., 2013; Sid-Lakhdar et al., 2020; and Liu et al., 2021). We propose to rely on DGP in MLA as the generalization of GPs to the deep learning and multitask settings.

The following Sections “Gaussian processes” and “Deep Gaussian processes” describe the GP and DGP models, respectively.

Gaussian processes

We provide here a brief presentation of *Gaussian processes*. We invite the reader to consult (Rasmussen and

Williams, 2006) for a detailed description. A GP 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

$$f(x) \sim GP(\mu(x), k(x, x')) \quad (2)$$

where

$$\mu(x) = \mathbb{E}[f(x)] \quad (3)$$

$$k(x, x') = \mathbb{E}[(f(x) - \mu(x))(f(x') - \mu(x')))] \quad (4)$$

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 run-away generic choice is the following exponential quadratic kernel:

$$k(x, x') = \sigma^2 \exp\left(-\sum_{i=1}^{D(\mathbb{P})} \frac{(x_i - x'_i)^2}{l_i}\right) \quad (5)$$

where σ^2 (variance) and l_i (lengthscales) are hyperparameters 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:

$$\begin{aligned} \log(p(y|X)) = & -\frac{1}{2}(y - \mu(X))^T (K + \sigma^2 I)^{-1} (y - \mu(X)) \\ & -\frac{1}{2} \log|K + \sigma^2 I| - \frac{n}{2} \log(2\pi) \end{aligned} \quad (6)$$

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 $M \ll N$. The location of the pseudo points (inducing inputs) is defined by $Z = \{z_1, \dots, z_M\}$, 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

$$p(y, f, u) = p(f|u; X, Z) p(u, Z) \prod_{i=1}^N p(y_i, f_i) \quad (7)$$

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

$$f(x^*) = K(x^*, X)^T K(X, X)^{-1} Y \quad (8)$$

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

$$\text{var}(x^*) = K(x^*, x^*) - K(x^*, X)^T K(X, X)^{-1} K(X, x^*) \quad (9)$$

The formulation in equations (8) and (9) holds for both the GP model and LCM.

Linear Coregionalization Model

Particularly, we chose to use the LCM (Journel and Huijbregts, 1978; Goovaerts, 1997), the choice of which is driven by its generality, flexibility, and modeling power, albeit its modeling cost, compared to the plethora of models in the literature that are derived from it as special and constrained cases (Bonilla et al., 2008). The key to LCM is the construction of an approximation of the covariance between the different outputs of the model (model of every $t \in T$).

In this method, the relations between outputs are expressed as linear combinations of independent *latent random functions*

$$f(t_i, x) = \sum_{q=1}^Q a_{i,q} u_q(x) \quad (10)$$

where $a_{i,q}$ ($i \in [1, \mathcal{N}^o T]$) are hyper-parameters to be learned, and u_q are the latent functions, whose hyper-parameters need to be learned, as well.

The independence hypothesis is important as it allows us to compute the covariance between the outputs only through the auto-covariance of the latent functions themselves, as it implies

$$\text{cov}(u_i, u_j) = \begin{cases} \text{cov}(u_i, u_i) & , \text{if } i = j \\ 0 & , \text{if } i \neq j \end{cases} \quad (11)$$

The covariance of every latent function u_q is assumed to be generated from a kernel function:

$$\text{cov}(u_q(x), u_q(x')) = k_q(x, x') \quad (12)$$

This kernel can take x and x' to be vectors as input, in which case k_q is a scalar. However, this kernel can also consider them to be vectors of vectors (i.e., matrices). In such a case, $k_q(x, x')$ is a matrix, instead of a scalar, where the (i, j) entry corresponds to the evaluation of the kernel k_q on the i^{th} vector of x and j^{th} vector of x' .

The covariance structure between two outputs can now be expressed as

$$\text{cov}(f(t_i, x), f(t_{i'}, x')) = \sum_{q=1}^Q \sum_{q'=1}^Q a_{i,q} a_{i',q'} \text{cov}(u_q(x), u_{q'}(x')) \quad (13)$$

which thanks to equation (11) simplifies to

$$\text{cov}(f(t_i, x), f(t_{i'}, x')) = \sum_{q=1}^Q a_{i,q} a_{i',q} \text{cov}(u_q(x), u_q(x')) \quad (14)$$

The covariance matrix between all the tasks on inputs x and x' can now be expressed through the kernel $K(x, x')$

$$K(x, x') = \sum_{q=1}^Q B_q \otimes k_q(x, x') + D \otimes I \quad (15)$$

where \otimes is a Kronecker product, k_q is the covariance function of the q^{th} latent function, I is the identity matrix of size $\mathcal{N}^o P \times \mathcal{N}^o P$, D is a diagonal matrix, whose elements are the variance of the noise in the measurement of the samples, and B_q is a $\mathcal{N}^o T \times \mathcal{N}^o T$ matrix of parameters of the model such that

$$B_q[i, i'] = a_{i,q} a_{i',q} = W_q W_q^T \quad (16)$$

with W_q a vector of parameters. The $D \otimes I$ term acts as a regularization term that prevents overfitting and helps make the covariance matrix non-singular. The use of a Kronecker product holds in the isotropic case, where all the tasks to be modeled are sampled at the same locations. In the heterotropic case, where every task has its own samples, the formulation is more complex. However, the idea remains the same. The parameters that need to be learned are the elements of the W_q vectors as well as the hyper-parameters of the kernels k_q . As this task can be computationally expensive, several techniques exist in the literature to reduce its cost. The one that we use is to assume that the rank of the B_q matrices is one, that is they require less effort to estimate.

In order to find the best hyper-parameters of the model, the log-likelihood of the model on the data needs to be optimized. The solution of this non-convex optimization problem is a subject of active research. Some software

libraries rely on model-free black-box optimization techniques to solve it while others rely on gradient-based optimization techniques with multi-start (to circumvent the non-convexity), as is the case in the *GP* machine learning group package we rely on.

Deep Gaussian processes

Model description. A DGP (Damianou and Lawrence, 2013) 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, \mathcal{X} is the input of the first layer and \mathcal{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 $Z = \{Z^1, \dots, Z^L\}$, and the corresponding inducing outputs are denoted by $U = \{U^1 = f^1(Z^1), \dots, U^L = f^L(Z^L)\}$. 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 (7)) as

$$p\left(Y, \{F^l, U^l\}_{l=1}^L\right) = \prod_{l=1}^L p(F^l | U^l; F^{l-1}, Z^{l-1}) p(U^l; Z^{l-1}) \prod_{i=1}^N p(y_i | f_i^l) \quad (17)$$

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) (Blei et al., 2017) and *Markov Chain Monte Carlo* (MCMC) (Andrieu et al., 2003). The difficulty with DGPs is the existence of complex correlations both within and between layers.

The original paper on DGPs (Damianou and Lawrence, 2013) relies on a mean-field variational approach that makes strong independence assumptions and Gaussianity assumptions. It uses a variational posterior which maintains the exact model conditioned on the inducing outputs but forces independence between layers. The true posterior is

likely to exhibit high correlations between layers, but mean-field variational approaches are known to severely underestimate the variance in these situations (Turner and Sahani, 2011). 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 therefore cannot express the complexity of the full model. Since the posterior loses all the correlations between, and so is likely to underestimate the variance. In practice, we found that optimizing the objective in Damianou and Lawrence (2013) results in layers being “turned off” (the signal to noise ratio tends to zero). In contrast, our posterior retains the full conditional structure of the true model.

The *Doubly Stochastic Variational Inference* (DSVI) (Salimbeni and Deisenroth, 2017) relies on a double source of stochasticity: (i) a sparse inducing point VI scheme (Matthews et al., 2016) 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).

Doubly Stochastic Variational Inference does not force independence between layers as is customarily the case with other VI techniques that assume approximate posteriors

The *Stochastic Gradient Hamiltonian Monte Carlo* (SGHMC) (Havasi et al., 2018 and Chen et al., 2014) 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

$$p(U, R|Y) \propto \exp\left(-U(u) - \frac{1}{2}r^T M^{-1}r\right) \quad (18)$$

where the negative log posterior $U(u) = -\log(p(U|Y))$ plays the role of the potential energy, and the $1/2r^T M^{-1}r$ 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) (Neal, 2010) method is intractable in DGP in the case of large training data, stochastic gradient estimates can be computed following the work of Chen et al. (2014), where

$$\Delta u = \epsilon M^{-1}r \quad (19)$$

$$\Delta r = -\epsilon \nabla U(u) - \epsilon C M^{-1} r + \mathcal{N}(0, 2\epsilon(C - B)) \quad (20)$$

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 [Havasi et al. \(2018\)](#) propose a variant of the *Monte Carlo Expectation Maximization* (MCEM) ([Wei and Tanner, 1990](#)) that they call *Moving Window Markov Chain Expectation Maximization* (MWEM). While MCEM alternates between the sampling from the posterior (E-step, equation (21)) and the maximization of the joint probability of the samples and the data (M-step, equation (22)),

$$u_{1,\dots,m} \sim p(u|Y, X, \theta) \quad (21)$$

$$\theta = \arg \max_{\theta} \frac{1}{m} \sum_{i=1}^m \log p(Y, u_i | X, \theta) \quad (22)$$

Moving Window Markov Chain Expectation Maximization 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).

Search phase

Acquisition function optimization

This section describes the mathematical formulation of inference in the GP model and LCM together, as the formulation is nearly identical, than in DGPs.

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 resulting solution found for every task is then evaluated through an expensive run of the application to be tuned and then re-injected into the model. The TLA2 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.

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 TLA2 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, TLA1, that can return a good guess within a fraction of a second.

First *Transfer Learning Algorithm* 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 GP 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 . The goal of TLA1 is to create a model that can predict the optimal configuration corresponding to an unexplored task without having to tune it.

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

$$OPT_i = \arg \min_{x \in \mathcal{S}(\mathbb{P})} f(t_i, x), \forall i \in [1, \mathcal{N}^\circ T] \quad (23)$$

An optimum parameter configuration is composed of as many parameters as $\mathcal{D}(\mathbb{P})$. Consequently, the solution that we propose is to create $\mathcal{D}(\mathbb{P})$ separate and independent GPs $G_{i \in [1, \mathcal{D}(\mathbb{P})]}$ to model every component of the optimal solutions separately. Such a GP model is described in Section “Gaussian processes”. 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 TLA1 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

$$OPT(t^*) = [G_1(t^*), G_2(t^*), \dots, G_{\mathcal{D}(\mathbb{P})}(t^*)] \in \mathcal{S}(\mathbb{P}) \quad (24)$$

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 characterizing the optima.

Experimental results

This section presents the experimental results assessing the combined advantage of a DGP model in a multitask and transfer learning setting. An introductory example is first presented in Section “Motivational autotuning example” consisting in the tuning of the Cholesky factorization routine DPOTRF in the PLASMA library. This simple example allows for the plotting of the model prediction and confidence together with a plot of the data. Section “Experimental setting” describes the applications to be tuned together with the different algorithms that are compared against. Section “Autotuning results” presents the comparative results and show the advantage that DGP-based multitask and transfer learning framework brings to application performance autotuning.

Motivational autotuning example

The Cholesky factorization routine DPOTRF in the PLASMA library has the merit to be simple enough, in that its problem space is a one dimensional space, characterized by the size of the matrices to be factorized, and its parameter space is also a one dimensional space, characterized by the block size to be used to tile matrices. In order to make the experimental results stable and reproducible, a single thread is used.

Given that, on matrix sizes below 1000, DPOTRF can take less than a second to execute, we were able to get exhaustive results, that is, for all matrices sizes in the range $[1, 1000]$ and for all block sizes in the range $[1, 1000]$. Figure 1 represents the Gflop/s rate obtained, the X-axis representing the matrix size, the y-axis representing the block size, and the shade of gray color representing the Gflop rate (black representing 0 Gflop/s and white representing the highest Gflop/s obtained on any experiment).

Given the constraint that the block size cannot be larger than the size of a matrix, the upper left triangular part of the figure corresponds to the invalid parameter configurations. Moreover, the area of low values of the block size (bottom of the figure) is in dark gray which corresponds to low Gflop/s rate. This is consistent with the fact that very small values of blocking do not allow for enough cache reuse, and thus lead the application to exhibit BLAS2 behavior instead of BLAS3 behavior. Furthermore, within the lower triangular part of the figure, representing the valid values, several diagonal gray lines are visible. These areas simply correspond to jumps in performance due to cache effects. Indeed, given the size of matrices considered, the performance of the application sharply drops whenever the matrix sizes get too large to fit in L1 or L2 caches.

Given this exhaustive dataset, we run MLA on 50 tasks and give every task a budget of 10 runs, split into 5 runs chosen in the sampling phase and 5 runs chosen by the search phase. Figure 2 shows the final results at the end of the tuning for two neighboring tasks, that is, close in the problem space. The results of the executions are represented as blue dots, the prediction of the model as a pink curve and the confidence of the model as a pink shaded area.

In both figures, it is straightforward to differentiate between the configurations proposed by the sampling phase as the blue dots spread over the whole parameter space and the configurations proposed by the search phase as the ones clustered around the best value predicted by the model. This immediate convergence to the optima is of course only possible in such a simple case as the DPOTRF routine and is not usually observed on more complicated cases. Moreover, it is interesting to see in the second figure that, although no data

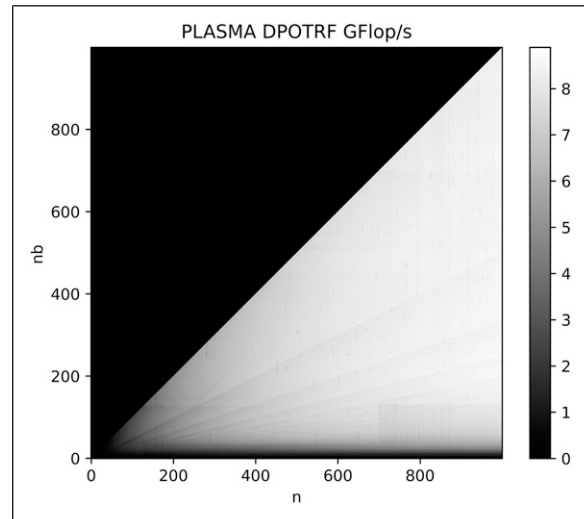


Figure 1. Exhaustive Gflop/s results of the DPOTRF routine in the PLASMA library. The X-axis represents the matrix size. The y-axis represents the block size. The shade of gray represents the Gflop/s.

have been gathered in the lower spectrum of the parameter space (values of the block size smaller than 0.2), the model is still able to predict the drop in performance of the application in that regime, and the tuner to not waste precious application runs in exploring this area. The reason is that the model has learned the behavior of the application in that range but on the similar task represented in the first figure. Indeed, in the first figure, the low end of the spectrum has indeed been sampled. This behavior is representative of the power of leveraging the multitask learning in an autotuning setting.

Experimental setting

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 2.80 GHz. BETA is a single-node computer, containing 40 cores of Intel^(R) Xeon^(R) CPU E5-2650 v3 at 2.30 GHz.

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 Gflop/s 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 $[1, 40]$. Hundred 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

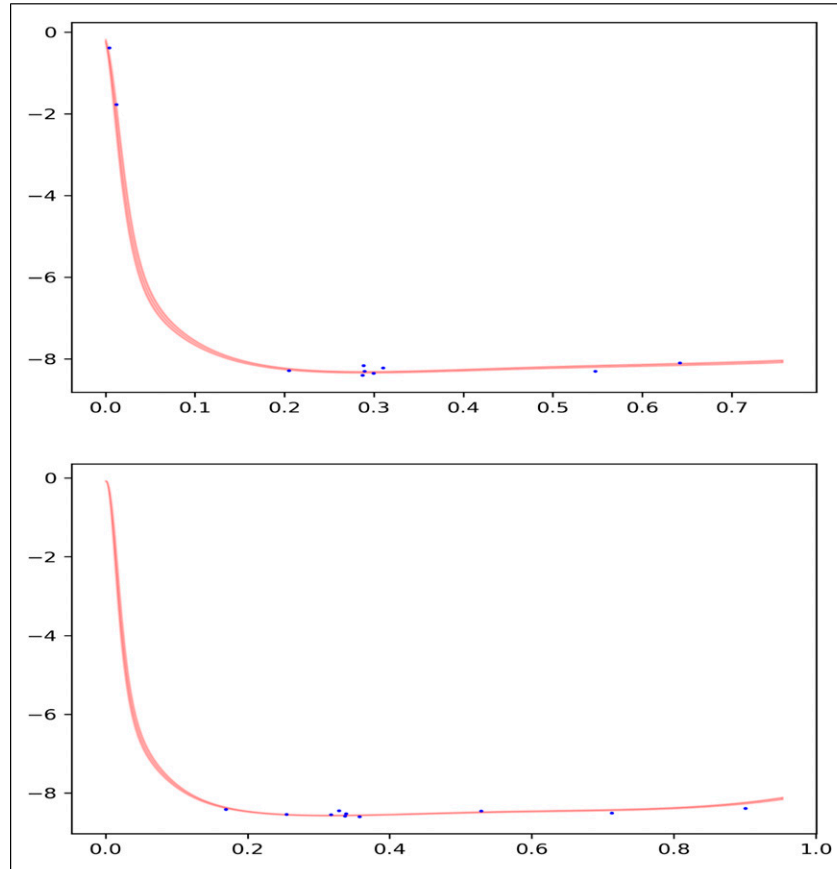


Figure 2. Result of MLA on the tuning of DPOTRF. The two subfigures correspond to two neighbor tasks. The X-axis represents the block size. The y-axis represents the Gflop/s rate. The blue dots represent runs of the application. The pink curve represents the model prediction and the pink shaded area represents the confidence of the model.

lookahead panels to be prefetched and precomputed. The range of matrix sizes is $[1, 10,000]$, 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. Moreover, the number of processes times the number of threads per process must be equal to the total number of available cores on the nodes. Hundred 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 behavior of the application on new unknown tasks. At the same time, we must keep the number of tasks small enough so that the total wall-clock 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 h for the DGEQRF problem and 48 h for the DGEMM problem.

Autotuner types. Our proposed algorithms are compared against several autotuning techniques: OpenTuner (Ansel et al., 2014) and HpBandSter (Falkner et al., 2018), two general-purpose model-free autotuners (Section “Model-free optimization”), and EGO with a GP model at its core (EGO-GP). Given that all of the Bayesian optimization methods used and compared against rely on the EGO algorithm at their core, we differentiate them by the relied upon model. Thus, the two Bayesian-based autotuners are GP and DGP. The first uses a GP model and only applies to single-task tuning. The second uses the DGP model and not only allows for multitask tuning but also for transfer learning.

Autotuning results

In order to showcase the benefits of the DGP model, a comparison of the different autotuners is made. We present results of multitask learning and transfer learning separately.

Multitask learning setting. 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 LCM and MLA attempts to tune all training tasks simultaneously, OpenTuner, HpBandSter, and EGO-GP operate on single-tasks independently. Figure 3 shows the results for the DGEQRF problem. 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

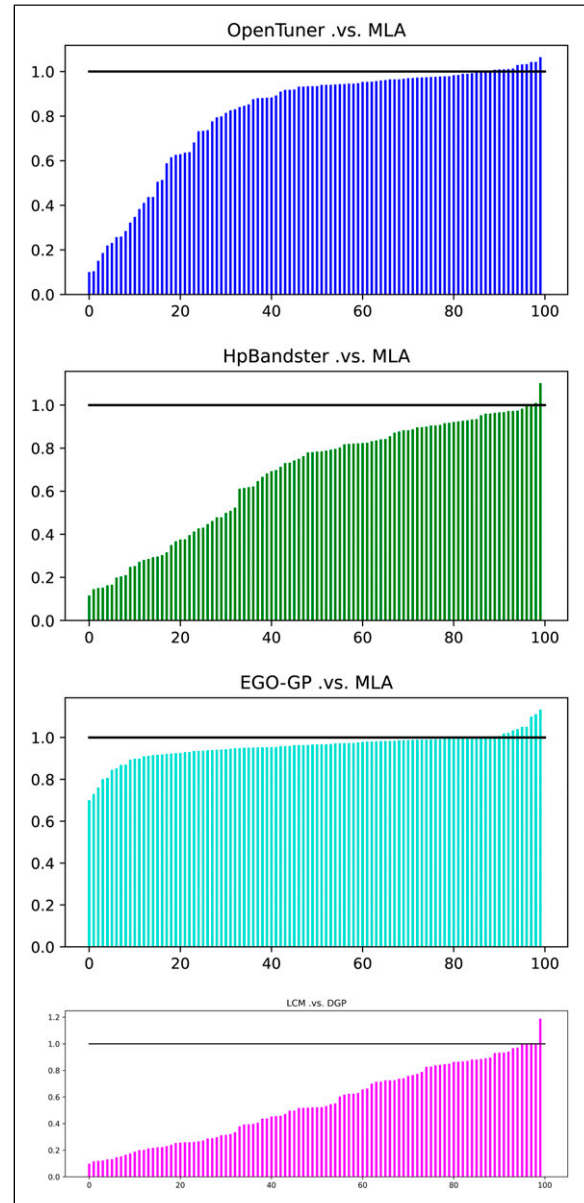


Figure 3. Relative performance of DGEQRF on train tasks given the parameters optimized by OpenTuner, HpBandSter, and EGO-GP and LCM compared to the ones found by MLA (black horizontal lines).

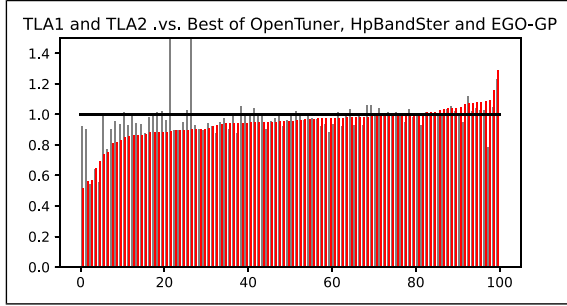


Figure 4. Relative performance of DGEQRF on test tasks given the parameters predicted by TLA1 (gray) and TLA2 (red) using the DGP model compared to the ones optimized by EGO-GP (black horizontal lines).

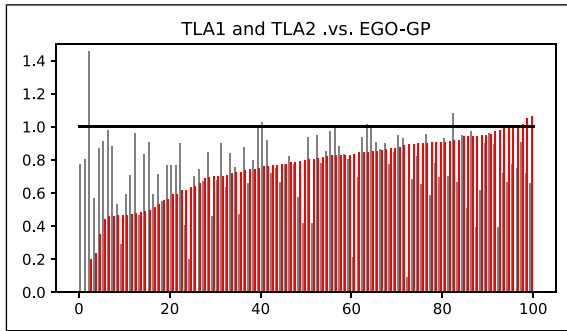


Figure 5. Relative performance of DGEMM on test tasks given the parameters predicted by TLA1 (gray) and TLA2 (red) using the DGP model compared to the ones optimized by EGO-GP (black horizontal lines).

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.

Transfer learning setting (TLA1 and TLA2). After training the DGP model on the training tasks, new test tasks are 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 (Gflop/s) obtained for each test task is then considered as the reference value. Then, the TLA1 and TLA2 methods are applied to predict the optimal parameter for each test task, but without ever running the applications on them. Figures 4 and 5 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 gray and red bars represent the results for the TLA1 and TLA2 methods, respectively. The tasks are sorted by increasing performance ratio of TLA2 compared to EGO-GP.

The surprising result is that, on the DGEQRF problem, not only TLA1 is able to outperform TLA2, but it is also competitive with the classical EGO-GP. On the other hand, as expected, TLA2 presents better results than TLA1 on the DGEMM problem.

It is important to notice that there is a clear imbalance in the results. Indeed, TLA1 and TLA2, compared to EGO-GP, yield a dreadful performance on some tasks and better performance on other tasks. However, the average performance offered by TLA2 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 TLA1 and TLA2 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 behavior 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.

Conclusion

Summary of findings

This article 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 DGP 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 (TLA2). In the case where a quick prediction is needed, an approximation is proposed (TLA1) that uses a simple GP regression of the best parameters found for all training tasks.

Perspectives

Several avenues for improvement are possible.

An ongoing work is the adaptation of the ideas in [Pearce and Branke \(2018\)](#), namely, continuous multitask Bayesian optimization with correlation. Although this latter work relies on a shallow GP model, it can be adapted (albeit with some approximations) to the deep GP model we rely on. The idea is that, given the availability of data over a whole space of tasks, not only new parameters can be tried in the loop of EGO, but the tasks to be tuned can also be chosen incrementally. This allows for a principled selection of the input problems to be tuned, thus spreading the tuning effort in the regions of the task space that need it most.

Another perspective is in the field of multi-fidelity. It is common for HPC applications to be able to execute in a somehow degraded mode. For example, FEM solvers can use coarse-grain meshes instead of fine-grain ones. Although the accuracy of the resulting numerical result is low, the computation time of the application is greatly reduced. In this setting, it is possible to leverage the availability of a large amount of low-quality data in order to improve the model prediction of the targeted scarce high-quality data. In the shallow learning context, the co-Kriging method ([Perdikaris et al., 2015](#)) has successfully been applied to this end when relying on GP models. We seek to apply a similar approach on the DGP model in a broad multitask learning and transfer learning setting.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: The surrogate performance model portion of this work was supported by the U.S. Department of Energy, Office of Science, and ASCR under Award Number DE-SC0021419. The development of some of the software libraries tested in this work was supported by the National Science Foundation Office of Advanced Cyberinfrastructure Directorate for Comp. & Info. Sci. & Eng. under Grant No. 2004541. This research and software used in this work was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of

Science and the National Nuclear Security Administration, and by the U.S. Department of Energy and Office of Science, under Contracts DE-AC05-00OR22725 and DE-AC52-07NA27344.

ORCID iD

Piotr Luszczek  <https://orcid.org/0000-0002-0089-6965>

Note

1. Linear Model of Coregionalization (LMC) and Linear Coregionalization Model (LCM) are used interchangeably in the literature.

References

- Alvarez MA and Lawrence ND (2011) Computationally efficient convolved multiple output Gaussian processes. *Journal of Machine Learning Research* 12: 1459.
- Andrieu C, de Freitas N, Doucet A, et al. (2003) An introduction to mcmc for machine learning. *Machine Learning* 50(1): 5–43. DOI: [10.1023/A:1020281327116](https://doi.org/10.1023/A:1020281327116)
- Ansel J, Kamil S, Veeramachaneni K, et al. (2014) OpenTuner: an extensible framework for program autotuning. In: *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada: Association for Computing Machinery, 303–316. <http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf>
- Anzt H, Haugen B, Kurzak J, et al. (2015) Experiences in autotuning matrix multiplication for energy minimization on GPUs. *Concurrency and Computation: Practice and Experience* 27(17): 5096–5113. DOI: [10.1002/cpe.3516](https://doi.org/10.1002/cpe.3516)
- Balaprakash P (2015) *SuRF: Search Using Random Forest*. <https://github.com/brnorris03/Orio/tree/master/orio/main/tuner/search/mlsearch>
- Balaprakash P, Dongarra JJ, Gamblin T, et al. (2018) Autotuning in high-performance computing applications. *Proceedings of the IEEE* 106(11): 2068–2083. DOI: [10.1109/JPROC.2018.2841200](https://doi.org/10.1109/JPROC.2018.2841200)
- Bellman R (1957) *Dynamic Programming*. First Edition. Princeton, NJ, USA: Princeton University Press.
- Bergstra J, Komer B, Eliasmith C, et al. (2015) Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery* 8(1): 014008. <http://stacks.iop.org/1749-4699/8/i=1/a=014008>
- Blei DM, Kucukelbir A and McAuliffe JD (2017) Variational inference: a review for statisticians. *Journal of the American Statistical Association* 112(518): 859–877. DOI: [10.1080/01621459.2017.1285773](https://doi.org/10.1080/01621459.2017.1285773)
- Bonilla EV, Chai KMA and Williams CKI (2008) *Multi-task Gaussian Process Prediction*. NIPS.
- Chan TM, Larsen KG and Pătrașcu M (2011) Orthogonal range searching on the ram, revisited. In: *Proceedings of the Twenty-Seventh Annual Symposium on Computational Geometry (Paris, France) (SoCG '11)*. New York, NY, USA: ACM, 1–10.

- Chen T, Fox EB and Guestrin C (2014) Stochastic gradient Hamiltonian Monte Carlo. In: *International Conference on Machine Learning*, pp. 1683–1691.
- Damianou A and Lawrence N (2013) Deep gaussian processes. *Artificial Intelligence and Statistics* 16: 207–215.
- Eglajs V and Audze P (1977) New approach to the design of multifactor experiments. *Problems of Dynamics and Strengths* 35(1): 104–107.
- Falkner S, Klein A and Hutter F (2018) BOHB: robust and efficient hyperparameter optimization at scale. In: Dy J and Krause A (eds), *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, PMLR, Stockholm, Sweden, 10–15 July 2018, pp. 1437–1446. <http://proceedings.mlr.press/v80/falkner18a.html>.
- Goovaerts P (1997) *Geostatistics for Natural Resources Evaluation*. Oxford, UK: Oxford University Press.
- Havasi M, Hernández-Lobato JM and Murillo-Fuentes JJ (2018) Inference in deep gaussian processes using stochastic gradient hamiltonian monte carlo. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18*. Red Hook, NY, USA: Curran Associates Inc., pp. 7517–7527.
- Hebbal A, Brevault L, Balesdent M, et al. (2018) Efficient global optimization using deep gaussian processes. In: 2018 IEEE Congress on Evolutionary Computation (CEC), Rio de Janeiro, Brazil, 08–13 July 2018, pp. 1–8. DOI: [10.1109/CEC.2018.8477946](https://doi.org/10.1109/CEC.2018.8477946)
- Hebbal A, Brevault L, Balesdent M, et al. (2019) *Multi-Objective Optimization Using Deep Gaussian Processes: Application to Aerospace Vehicle Design*. DOI: [10.2514/6.2019-1973](https://doi.org/10.2514/6.2019-1973)
- Hebbal A, Brevault L, Balesdent M, et al. (2021) Bayesian optimization using deep gaussian processes with applications to aerospace system design. *Optimization and Engineering* 22(1): 321–361. DOI: [10.1007/s11081-020-09517-8](https://doi.org/10.1007/s11081-020-09517-8)
- Hennig P and Schuler CJ (2012) Entropy search for information-efficient global optimization. *Journal of Machine Learning Research* 13: 1809.
- Iman RL, Helton JC and Campbell JE (1981) An approach to sensitivity analysis of computer models: part I—introduction, input variable selection and preliminary variable assessment. *Journal of Quality Technology* 13(3): 174–183. DOI: [10.1080/00224065.1981.11978748](https://doi.org/10.1080/00224065.1981.11978748)
- Jones DR, Schonlau M and Welch WJ (1998) Efficient global optimization of expensive black-box functions. *Journal of Global Optimization* 13(4): 455–492. DOI: [10.1023/A:1008306431147](https://doi.org/10.1023/A:1008306431147)
- Journel AG and Huijbregts CJ (1978) *Mining Geostatistics*. London: Academic Press.
- Katehakis MN and Veinott AF Jr (1987) The multi-armed bandit problem: Decomposition and computation. *Mathematics of Operations Research* 12(2): 262–268. DOI: [10.1287/moor.12.2.262](https://doi.org/10.1287/moor.12.2.262)
- Kennedy J and Eberhart R (1995) Particle swarm optimization. In: *Proceedings of ICNN'95 – International Conference on Neural Networks, Volume 4*. Perth: IEEE, 1942–1948.
- Kirkpatrick S, Gelatt CD and Vecchi MP (1983) Optimization by simulated annealing. *SCIENCE* 220(4598): 671–680.
- Li L, Jamieson K, DeSalvo G, et al. (2017) Hyperband: a novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research* 18(1): 6765–6816. <http://dl.acm.org/citation.cfm?id=3122009.3242042>
- Liu Y, Sid-Lakhdar WM, Marques OA, et al. (2021) Gptune: Multitask learning for autotuning exascale applications. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '21*. New York, NY, USA: Association for Computing Machinery, 234–246. DOI: [10.1145/3437801.3441621](https://doi.org/10.1145/3437801.3441621)
- Matthews AGG, Hensman J, Turner R, et al. (2016) On sparse variational methods and the kullback-leibler divergence between stochastic processes. In: Gretton A and Robert CC (eds), *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, Proceedings of Machine Learning Research, Volume 51*. Cadiz, Spain: PMLR, 231–239. <https://proceedings.mlr.press/v51/matthews16.html>
- McKay MD, Beckman RJ and Conover WJ (1979) A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics* 21(2): 239–245. DOI: [10.2307/1268522](https://doi.org/10.2307/1268522)
- Nath R, Tomov S and Dongarra J (2010a) Accelerating GPU kernels for dense linear algebra. In: *Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR'10*. Berkeley, CA: Springer.
- Nath R, Tomov S and Dongarra J (2010b) An Improved MAGMA GEMM For Fermi Graphics Processing Units. *The International Journal of High Performance Computing Applications* 24(4): 511–515. DOI: [10.1177/1094342010385729](https://doi.org/10.1177/1094342010385729)
- Neal RM (2010) MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo* 54: 113–162.
- Nelder JA and Mead R (1965) A simplex method for function minimization. *The Computer Journal* 7: 308–313.
- Pan SJ and Yang Q (2010) A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering* 22(10): 1345–1359.
- Pearce M and Branke J (2018) Continuous multi-task bayesian optimisation with correlation. *European Journal of Operational Research* 270(3): 1074–1085. DOI: [10.1016/j.ejor.2018.03.017](https://doi.org/10.1016/j.ejor.2018.03.017) <https://www.sciencedirect.com/science/article/pii/S0377221718302261>
- Perdikaris P, Venturi D, Royset JO, et al. (2015) Multi-fidelity modelling via recursive co-kriging and Gaussian-Markov random fields. *Proceedings. Mathematical, Physical, and Engineering Sciences* 471(2179): 20150018. DOI: [10.1098/rspa.2015.0018](https://doi.org/10.1098/rspa.2015.0018)
- Qin C, Klabjan D and Russo D (2017) Improving the expected improvement algorithm. In: Guyon I, Luxburg

UV, Bengio S, et al. (eds), *Advances in Neural Information Processing Systems, Volume 30*. Red Hook, NY: Curran Associates, Inc. URL <https://proceedings.neurips.cc/paper/2017/file/b19aa25ff58940d974234b48391b9549-Paper.pdf>

- Rajaram D, Puranik TG, Ashwin Renganathan S, et al. (2021) Empirical assessment of deep gaussian process surrogate models for engineering problems. *Journal of Aircraft* 58(1): 182–196. DOI: [10.2514/1.C036026](https://doi.org/10.2514/1.C036026)
- Rasmussen CE and Williams C (2006) *Gaussian Processes for Machine Learning*. Cambridge: MIT Press.
- Salimbeni H and Deisenroth M (2017) Doubly stochastic variational inference for deep gaussian processes. In: Guyon I, Luxburg UV, Bengio S, et al. (eds), *Advances in Neural Information Processing Systems, Volume 30*. Red Hook, NY: Curran Associates, Inc. <https://proceedings.neurips.cc/paper/2017/file/8208974663db80265e9bfe7b222dcb18-Paper.pdf>
- Seeger M, Teh YW and Jordan MI (2005) *Semiparametric Latent Factor Models*. AISTATS.
- Shahriari B, Swersky K, Wang Z, et al. (2016) Taking the human out of the loop: a review of bayesian optimization. *Proceedings of the IEEE* 104(1): 148–175.
- Sid-Lakhdar WM, Demmel JW, Li XS, et al. (2020) *Gptune Users Guide*. <https://github.com/gptune/GPTune/tree/master/Doc>
- Snoek J, Larochelle H and Adams RP (2012) Practical bayesian optimization of machine learning algorithms. In: *Proceedings of NIPS*.
- Srinivas M and Patnaik LM (1994) Genetic algorithms: a survey. *Computer* 27(6): 17–26.
- Swersky K, Snoek J and Adams RP (2013) Multi-task bayesian optimization. In: Burges CJC, Bottou L, Welling M, et al. (eds), *Advances in Neural Information Processing Systems, Volume 26*. Red Hook, NY: Curran Associates, Inc. <https://proceedings.neurips.cc/paper/2013/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf>
- Turner RE and Sahani M (2011) Two problems with variational expectation maximisation for time-series models. In: Barber D, Cemgil AT and Chiappa S (eds), *Bayesian Time Series Models*. Cambridge University Press.
- Wang GG and Shan S (2007) Review of metamodeling techniques in support of engineering design optimization. *Journal of Mechanical Design* 129(4): 370–380.
- Wei GCG and Tanner MA (1990) A Monte Carlo implementation of the em algorithm and the poor man's data augmentation algorithms. *Journal of the American Statistical Association* 85(411): 699–704.
- Zhang Y and Yang Q (2017) A survey on multi-task learning. In: CoRR abs/1707.08114, 1. arXiv:1707.08114.

Author biographies

Piotr Luszczek is a Research Assistant Professor at the Innovative Computing Laboratory in University of Tennessee, Knoxville's Department of Electrical Engineering and Computer Science. Piotr earned MSc in Computer Science from A.G.H. University of Science and Technology in Kraków, Poland, and PhD in Computer Science from the University of Tennessee Knoxville. Piotr's research interests include benchmarking, numerical linear algebra for high-performance computing, automatic performance tuning for hardware accelerators, and stochastic models for performance. Piotr has over a decade of experience developing high-performance numerical software for large scale, distributed memory systems with multicore processors and GPU accelerators. His main research interest includes benchmarking and automated performance tuning. Currently, Piotr serves a co-PI for the ECP xSDK project that has a main goal to improve access to the world-class software on the Exascale machines.

Wissam M. Sid-Lakhdar is a Research Scientist at the Innovative Computing Laboratory in University of Tennessee, Knoxville. He earned Master in Computer Science and Applied Mathematics from Institut National Polytechnique de Toulouse, France, and PhD in Computer Science, Écoles Normales Supérieures de Lyon, France. His research interests include parallel algorithms, high performance computing, sparse, or dense linear algebra with low-rank structure, Bayesian Optimization for autotuning, and scalable deep kernel methods.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in CS from the Illinois Institute of Technology in 1973. He received his PhD in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Professor Emeritus of Computer Science in the EECS Department at the University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at ORNL; Turing Fellow at Manchester University; an Adjunct Professor in the Computer Science Department at Rice University; and a Faculty Fellow of the Texas A&M University's Institute for Advanced Study. He is the director of the Innovative Computing Laboratory at the University of Tennessee. He is also the recipient of the ACM's Alan M. Turing Award for HPC research contribution.