Enhancing Autotuning Capability with a History Database

Younghyun Cho, James W. Demmel
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA, USA
{*younghyun,demmel*}*@berkeley.edu*

Xiaoye S. Li, Yang Liu, Hengrui Luo
Computational Research Division,
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
{*xsli,liuyangzhuan,hrluo*}*@lbl.gov*

Abstract—**Autotuning is gaining importance to achieve the best possible performance for exascale applications. The performance of an autotuner usually depends on the amount of performance data collected for the application, however, collecting performance data for large-scale applications is oftentimes an expensive and daunting task. This paper presents an autotuner database, which we call a *history database*, for enhancing the reusability and reproducibility of performance data. The history database is built into a publicly available autotuner called GPTune, and allows users to store performance data obtained from autotuning and download historical performance data provided by the same or other users. The database not only allows reuse of the best available tuning results for widely used codes but also enables transfer learning that can leverage knowledge of pre-trained performance models. An evaluation shows that, for ScaLAPACK's PDGEQRF routine, a transfer learning approach using the history database can attain up to 33% better tuning results compared to single task learning without using prior knowledge, on 2,048 cores of NERSC's Cori supercomputer.**

Keywords-**autotuning; transfer learning; crowd-tuning; Exascale Computing Project;**

I. Introduction

Autotuning is becoming increasingly important for optimizing a variety of applications ranging from classical linear algebra operations to machine learning algorithms. The goal of autotuning is to automatically choose tuning parameters to achieve the best possible performance (e.g. the best runtime, number of messages communicated, memory size, accuracy, etc.) of an application within a given time and/or resource budget. Choosing optimal tuning parameters using a naive autotuning approach such as grid search or random search, however, is infeasible when the number of tuning parameters is large, the application performance has complex behaviors, and/or running the application to measure the actual performance is expensive. In this context, recent autotuners such as ytopt [1], HpBandster [2], HiPerBOt [3], BLISS [4], and GPTune [5] often employ Bayesian optimization [6] (using Gaussian Processes (GP)). These Bayesian optimization-based autotuners tune applications as "black-boxes", running them for carefully chosen tuning parameter samples and building a performance model (i.e. surrogate model) based on the measured performance (i.e. function evaluation).

While these recent autotuners may achieve better tuning results than grid search or random search, tuning large-scale applications still remains a challenge due to their expensive function evaluations. Moreover, as we approach the exascale era, high performance computing (HPC) applications are becoming more complex and expensive to evaluate, and the need to optimize HPC applications on different machines and software configurations may hinder a wide adoption of autotuning techniques. As an example, climate modeling accounts for one of the largest portions of supercomputer workloads, but the common practice for climate model tuning still relies on manual input from experts due to the computational costs [7].

To overcome this challenge, we focus on database support to maximize the reusability and reproducibility of collected performance data. Several existing autotuners [1], [5], [8] support storing obtained performance data to log files, however, they do not provide some necessary information to reproduce performance data (e.g. it is unknown which software and how many compute resources are used), and reusing data is usually only possible for the identical tuning task. For enhancing reusability and reproducibility, our work aims to provide a more sophisticated database that can provide the information needed to reproduce performance data and reuse historical performance data to help tune new tuning tasks. We envision an autotuning database, where different users at different sites can share performance data, so everyone can benefit from expensive runs of widely used HPC codes and tune difficult problems by leveraging the power of crowd-tuning.

To this end, we present a *history database* for enhancing autotuning capability by improving the reusability and reproducibility of collected performance data. The history database is built into an open-source autotuner called GPTune [5] [1], and allows users to save and reuse performance data (e.g. function evaluation data and trained surrogate models) obtained from GPTune. When reusing historical data, users need to determine which performance data are relevant for learning from different machines or software versions or configurations. The history database therefore

[1]`https://github.com/gptune/GPTune`

supports storing the machine information (e.g. number of nodes/cores used to run the application) and which version of the application code and the software libraries are used as well as the function evaluation results. Furthermore, the history database enables transfer learning to leverage knowledge of pre-trained surrogate models to tune new tasks. The presented transfer learning autotuning (TLA) approach relies on GPTune's multitask learning autotuning (MLA) which permits tuning multiple tasks simultaneously. If multiple tasks exhibit highly correlated performance behaviors, MLA with a joint model across tasks may achieve optimal tuning results with fewer function evaluations compared to tuning individual tasks separately. Using the MLA feature, we can use pre-trained surrogate models to guide tuning new tasks. For crowd-tuning, we provide a public shared repository at `https://gptune.lbl.gov`, where users can upload their performance data obtained from autotuning or download performance data provided by other users.

To summarize, our work makes the following contributions:

- Design and implementation of an autotuner database providing high reproducibility of performance data. Unlike existing autotuners that store only function evaluation results, our database additionally stores the machine and software configuration along with the function evaluation results, which can be used to reproduce the performance data. The user can optionally use CK-GPTune [9] to automatically detect and store the software configuration with CK (Collective Knowledge) [10] technology.
- A TLA approach that leverages knowledge of pre-trained surrogate models to tune new tasks. With our database, users can use a pre-trained surrogate model to predict the value of certain tuning parameters, and we can treat TLA as running MLA with new function samples for the target tasks and previously-trained surrogate models for the existing tasks. An evaluation shows that, for ScaLAPACK's PDGEQRF routine, the transfer learning approach using our database feature can attain up to 33% better tuning results compared to tuning without previous knowledge, on 2,048 cores of NERSC's Cori supercomputer.
- Harnessing the power of crowd-tuning. Using our shared repository, users can post their tuning problems with a flexible hierarchical control on data access. Based on the given tuning information, multiple users at different sites can run autotuning for the same tuning problem while sharing performance data. This feature is useful when the application is expensive to run, and/or there are multiple users who want to use the same application.

The rest of this paper is organized as follows. In Section II, we discuss the related work. In Section III, we describe the background of GPTune. Section IV then presents the design and implementation of the GPTune history database. In Section V, we present several use cases of the history database and an evaluation of the TLA approach using the database. Section VI concludes this paper and presents future work.

II. Related Work

A. Existing Autotuners

There are existing autotuners for both general-purpose and HPC applications. OpenTuner [8] is a general-purpose autotuner which uses multiple heuristic techniques (e.g., simulated annealing, genetic algorithms, etc.) to find the optimal tuning parameters. HpBandSter [2] is an autotuner using Bayesian optimization and bandit-based methods to explore the tuning parameter space. With a focus on optimizing large-scale HPC applications, Menon et al. [3] presented HiPerBOt which is also a Bayesian optimization-based autotuner. BLISS [4] is also a Bayesian optimization-based autotuner using ensemble of models. Ytopt [1] explores parameter space with several machine learning techniques such as reinforcement learning within Bayesian optimization. Based on the ytopt framework, Wu et al. [11] optimize compile-time loop optimization pragmas for OpenMP applications. These autotuners have a feature to store the tuning result logs into files, so users can query performance logs from the files. Compared to these autotuners, we provide a more sophisticated database which records the machine and software configuration used to obtain the tuning data; this kind of information is useful to determine whether the data is relevant to be reused. In addition, we present a TLA approach by leveraging historical data which is one of the unique features exploiting GPTune's MLA algorithm. Furthermore, we provide a shared repository for sharing autotuning data between multiple users.

B. Crowd-tuning and reproducible autotuning

There have been many efforts to share research data between different users. A work philosophically similar to our database is the CK project [10]. CK is an interface and tool for reproducible and automated workflows, and provides a web repository to share results across different users. For reproducible workflows, users can write a meta-description of their workflow using CK's syntax and use a command line interface for automatically installing and running the workflow. CK's public repository [2] allows users to upload their performance results and download results from other users. Compared to CK, the main distinction of our database repository is that our database stores additional useful information in the context of autotuning. For example, our database stores trained surrogate models which can be used for autotuning (e.g. TLA). One of the powerful features

[2]`https://cknowledge.io/`

of CK is the reproducibilty of workflows. In CK, the user can define software dependencies to run the workflow, then CK can automatically detect the versions of the required software packages [12]. To leverage CK's software detection, in Section V-B, we provide an additional interface called CK-GPTune [9] which allows the user to run CK-enabled workflows with the GPTune history database.

Compared to CK, we particularly focus on the reusability of performance data of supercomputing applications that are expensive to evaluate. A similar idea from a different community is the Materials project [13] which provides a repository to share computed information of materials using supercomputing resources in materials science.

III. GPTune Background

A. Bayesian Optimization

Bayesian optimization [6] provides a surrogate modeling framework that allows us to model and optimize the black-box functions that describe either the runtime or resource consumption (i.e. memory usage and peak flops).

GPTune supports both single task autotuning (SLA) and multitask autotuning (MLA). For SLA, GPTune sequentially draws samples from the black-box function evaluations by acquisition maximization (for a chosen acquisition function) and fits the surrogate GP model with the samples using Bayes' theorem until the number of obtained samples reaches the prescribed sample count. For MLA, GPTune considers multiple tasks with a joint surrogate model. As a result, we can model not only the correlation between samples within one task, but also the correlation between samples across multiple tasks as well. The MLA approach, in the specific form of multiple-output Gaussian process, allows us to learn and incorporate this piece of across-task information into our model and improve the quality of surrogate models. Such a feature is in the spirit of transfer learning and utilizes the existing data more efficiently.

B. Linear Coregionalization Model

For MLA, GPTune uses the *Linear Coregionalization Model* (LCM), which is a generalization of Gaussian Process (GP) in the multi-output setting. For a set of correlated objective functions for all the δ given tasks $(y_i(x), i \in 1..\delta)$, LCM builds a joint model of the target functions $(f_i(x), i \in 1..\delta)$, through the underlying assumption of linear dependence on latent functions (u_q):

$$f_i(x) = \sum_{q=1}^{Q} a_{i,q} u_q(x)$$

where each (u_q) is an independent GP, and $a_{i,q}$ are hyperparameters. In LCM, we assume the covariance $k_q(x, x')$ of a latent function u_q is based on a Gaussian kernel, as follows:

$$k_q(x, x') = \sigma_q^2 \exp(-\sum_{j=1}^{\beta} \frac{(x_j - {x_j}')^2}{I_j^q})$$

```
1  from autotune import *
2  from gptune import *
3
4  def objective(point):
5      return run_and_measure(point)
6
7  def main():
8      # Input/Parameter/Output space definitions
9      IS = Space([Integer(0, 10, transform='
       normalize', name='t')])
10     PS = Space([Real(0, 1, transform='normalize',
       name='x')])
11     OS = Space([Real(float('-Inf'), float('Inf'),
       name='y')])
12
13     # Create GPTune instance
14     problem = TuningProblem(IS, PS, OS, objective)
15     gt = GPTune(problem)
16     giventask = [[1],[2]]
17
18     # Run GPTune for tasks [1] and [2] for NS=10
       function evaluations per task
19     gt.MLA(Igiven = giventask, NS=10)
```

Listing 1. Simplified GPTune Python interface. This example has a single integer task t (e.g. a linear algebra operation on a matrix of a certain size) and the tunable parameter is a real number x (e.g. a block size) ranging from 0.0 to 1.0. GPTune then builds a surrogate model $f(x)$ which predicts the true runtime $y(x)$. GPTune tries to find optimal x minimizing $f(x)$ with a given budget (number of function evaluations).

where σ_q and I_j^q are also hyperparameters to be learned. In addition to the aforementioned hyperparameters $a_{i,q}$, σ_q, I_j^q, we use diagonal regularization parameters for the covariance matrix during the LCM. For details about the covariance matrix and the parameter search algorithm, please refer to the GPTune paper [5].

C. User Interface

GPTune is a Python software package (including a high performance C implementation for LCM). Users should write a Python code, which we call a driver code, to use GPTune for the user's tuning problem. Autotuning of GPTune relies on the `autotune` [14] interface which requires three information spaces: *input space (IS)*, *parameter space (PS)*, *output space (OS)*. IS contains all the input problems (i.e. tasks) that the application may encounter (e.g. the size of matrices under a numerical linear algebra operation, pointers to input files for a large HPC code) and PS contains all tuning parameter configurations to be optimized (e.g. blocking parameters for improving flop performance). OS is the output space for each of the scalar objective functions (e.g. measured runtime).

Listing 1 shows an example of GPTune interface code. After importing `GPTune` and `autotune` modules (lines 1–2), the user can then define the input, parameter, and output spaces, following the autotune interface (lines 8–11). GPTune currently supports integer, real, and categorical types of parameters. The user defines the objective function for the tuning problem, which may call the target application

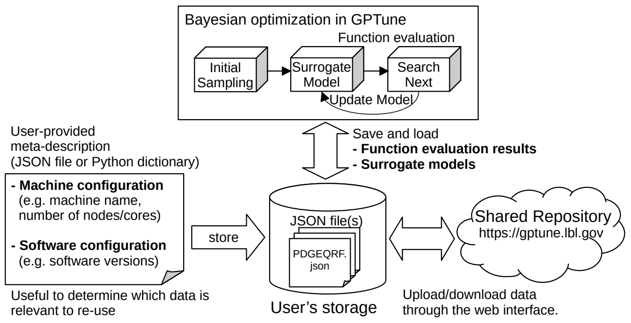

Bayesian optimization in GPTune
Function evaluation
Initial Sampling
Surrogate Model
Search Next
Update Model
User-provided meta-description (JSON file or Python dictionary)
- Machine configuration (e.g. machine name, number of nodes/cores)
- Software configuration (e.g. software versions)
Useful to determine which data is relevant to re-use
store
Save and load
- Function evaluation results
- Surrogate models
JSON file(s)
PDGEQRF. json
User's storage
Shared Repository
https://gptune.lbl.gov
Upload/download data through the web interface.

Figure 1. History database design.

and return the objective value (e.g. runtime) (lines 4–5). The remaining preparation steps are defining the tuning problem (line 14) and creating a GPTune instance (line 15). Then, the user can provide the target input tasks and run the tuner (lines 18–19).

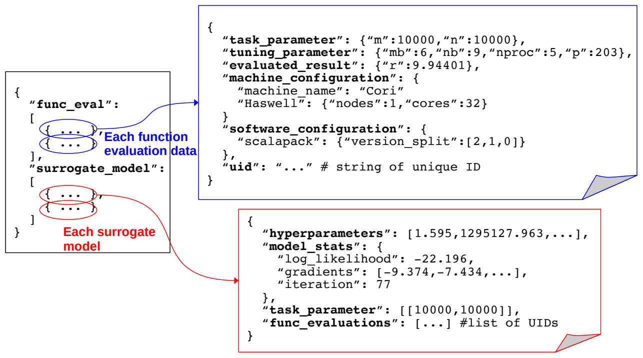
GPTune further supports several advanced options such as multi-objective tuning, multi-fidelity modeling, and incorporating an analytical performance model to guide autotuning. For more details, please refer to the GPTune User Guide [15].

IV. History Database

A. Design

Figure 1 illustrates the design of the history database. The database automatically stores and loads historical performance data to and from the performance data files with JavaScript Object Notation (JSON [16]) format in the user's local storage. Each application (tuning problem) has a separate data file that contains all the historical performance data (function evaluation results and trained surrogate models) of the application. As shown in the figure, the user can provide a meta-description of the tuning application, like machine configuration and software information (e.g. which software libraries are used for that application) using a JSON file or a Python Dict. To detect and provide the software configuration automatically, users can use our additional interface called CK-GPTune [9] which permits leveraging software detection tools in CK. This machine and software information is stored along with the function evaluation data and can be used to reproduce the performance data.

After evaluating the function with each parameter configuration, GPTune stores the task parameters, tuning parameters, and the evaluated result into the JSON file. This practice ensures that no data is lost in the cases where (a) a long run with many parameter configurations does not complete due to job allocation time limitation, or (b) a certain parameter configuration crashes and causes the tuner to stop. If GPTune is run in parallel and multiple processes attempt to update the JSON file simultaneously, the history database allows only one process to update the file at a time



```
{
  "func_eval":
  [
    { ... },
    { ... }
  ],
  "surrogate_model":
  [
    { ... },
    { ... }
  ]
}
```

```
{
  "task_parameter": {"m":10000,"n":10000},
  "tuning_parameter": {"mb":6,"nb":9,"nproc":5,"p":203},
  "evaluated_result": {"r":9.94401},
  "machine_configuration": {
    "machine_name": "Cori"
    "Haswell": {"nodes":1,"cores":32}
  }
  "software_configuration": {
    "scalapack": {"version_split":[2,1,0]}
  },
  "uid": "..." # string of unique ID
}
```

```
{
  "hyperparameters": [1.595,1295127.963,...],
  "model_stats": {
    "log_likelihood": -22.196,
    "gradients": [-9.374,-7.434,...],
    "iteration": 77
  },
  "task_parameter": [[10000,10000]],
  "func_evaluations": [...] #list of UIDs
}
```

Figure 2. Example JSON performance data file.

based on simple file access control (either using Python's filelock module or using `rsync`-based file synchronization). GPTune also supports storing and loading trained GP surrogate models along with some model statistics information such as likelihood values of the model. This unique feature of our database enables interesting use cases such as using pre-trained models for TLA (Section V-D) and sensitivity analysis on the tuning parameters (future work).

For crowd-tuning, we provide a shared repository at `https://gptune.lbl.gov` through NERSC's Science Gateways, where users can upload their performance data obtained from GPTune and download performance data provided by other users. In the shared repository, all submitted performance data is stored in NERSC storage and managed using MongoDB [17].

B. JSON Format

In this section, we explain the JSON format to store performance data from GPTune. Figure 2 shows an example of performance data for ScaLAPACK's [18] QR factorization routine PDGEQRF. Each tuning problem has a separate data file (e.g. PDGEQRF.json) that contains all performance data (obtained by the user and/or downloaded from the shared public database) of the tuning problem. Each JSON file has two labels `func_eval` and `surrogate_model`. As the name indicates, `func_eval` contains the list of all function evaluation results, and `surrogate_model` contains the list of each trained surrogate model's meta-data.

1) Function Evaluation Result: For a function evaluation, `task_parameter` contains the task parameter configuration, and `tuning_parameter` contains the tuning parameter configuration, and its evaluation result is stored in `evaluated_result`. Each function evaluation data also contains the information about the machine and software configuration to run the application. The machine information includes the machine name (e.g. Cori) and the number of cores/nodes used. The software information contains the versions of software packages used to compile/install the application. The machine and software configurations are stored in `machine_configuration` and `software_configuration`, respectively. Unlike task and tuning parameters, the machine and software information need to be given by the user. Users can use CK-GPTune to automatically detect the software dependencies and provide the detected software versions to the database (Section V-B introduces CK-GPTune).

In addition, for each function evaluation result, a timestamp and a unique ID (UID) of the function evaluation are automatically generated and appended by GPTune. If different users submit function evaluation results for the same task and parameter configurations, the database can differentiate between different function evaluation results using their UIDs.

2) Surrogate Model: This section explains what information is stored by GPTune for a GP surrogate model. Label `hyperparameters` contains the hyperparameter values which are required to reproduce the surrogate model. The database currently considers only the GPTune's default modeling scheme, the Linear Coregionalization Model (LCM) [5] discussed in Section III-B. The database can be extended to support other types of modeling algorithms; this is future work.

`model_stats` stores the model's statistics information. For the GPTune's LCM, we can store some statistics information such as *log_likelihood*, *gradients*, and *iteration* (how many iterations were required for the model to converge). Note that trained surrogate models may or may not be meaningful for different problem spaces. Therefore, the JSON data also contains task parameter information (`task_parameters`) and which function evaluation results were used (`func_eval`) to build the surrogate model, by containing the list of the UIDs of the function evaluation results. The history database can load trained models only if they match the problem space of the given optimization problem. Similar to function evaluation results, the data generation time and a unique ID of each surrogate model are also automatically appended by GPTune.

V. Use Scenarios

In this section, we present several use cases of the history database and demonstrate the benefit of the TLA approach using the database. For more detailed usage, we refer interested readers to our User Guide [15].

A. Reusing Historical Function Evaluation Data

As discussed in Section IV, the history database stores obtained function evaluation results and loads historical data based on a meta-description of the tuning application. The meta-description includes the application name, compute resources needed, and software dependence for both the current tuning experiment and loadable historical data.

Listing 2 shows an example of a tuning meta-description. For the machine configuration, users can provide the machine name and number of nodes/cores used (lines 3–6).

```
 1 {
 2   "tuning_problem_name": "PDGEQRF",
 3   "machine_configuration": {
 4     "machine_name": "Cori",
 5     "haswell": { "nodes": 8, "cores": 32 }
 6   },
 7   "software_configuration": {
 8     "openmpi": { "version_split": [4,0,1] },
 9     "scalapack": { "version_split": [2,1,0] },
10     "gcc": { "version_split": [8,3,0] }
11   },
12   "loadable_machine_configurations": {
13     "Cori": {
14       "haswell": {
15         "nodes": [1,2,3,4,5,6,7,8],
16         "cores": 32
17       }
18     }
19   },
20   "loadable_software_configurations": {
21     "openmpi": {
22       "version_from": [4,0,0],
23       "version_to": [5,0,0]
24     },
25     "scalapack": { "version_split": [2,1,0] },
26     "gcc": { "version_split": [8,3,0] }
27   }
28 }
```

Listing 2. Example meta-description to load historical data.

The software versions are passed as dictionaries which can contain a string of the software version (e.g. Git commit ID) and an array of the version split numbers (e.g. major, minor, and revision numbers) (lines 7–11). As shown in lines 12–27, users can define conditions to selectively load historical performance data. If the user wants to load performance data obtained from more than one machine configurations, the user can use an array to allow multiple configurations for loading. For example, line 15 allows loading performance data obtained from 1, 2, ..., 8 compute nodes. Loaded historical data can be used for checkpointing and restarting, which is useful for long autotuning processes, possible machine failures, limited job allocation times, etc.

B. Leveraging CK for Reproducible Tuning

As discussed in Section IV-B, for the reproducibility of tuning results, our database stores the software configuration of performance data (e.g. which software packages and versions used). This information can be automatically detected and saved by leveraging the CK's workflow automation [10] discussed in Section II. In CK, users write a meta-description of the software dependencies of their workflow and use a command line interface for auto-installation/compilation and running workflow. Then, CK automatically detects software versions of the software dependencies. To leverage CK with GPTune's history database, we provide CK-GPTune [9] which provides a command line interface to run CK-enabled autotuning workflows with the history database while taking the advantage of CK's software detection technology. Using CK-GPTune, users can use a simple command for installing or running a specific (automated) tuning application. CK-GPTune provides several examples of automated autotuning including ScaLAPACK's PDGEQRF tuning (used for TLA experiments in Section V-D). In the automated PDGEQRF tuning, users can set up the tuning task with a command, `$ ck compile CK-GPTune:program:PDGEQRF`, which detects the versions and the locations of the necessary software (GCC, OpenMPI, Intel Math Kernel Library, and ScaLAPACK). Then, users can run the automated tuning, `$ ck MLA gptune --target=PDGEQRF`, which will store the detected software in the history database.

C. Reusing a Surrogate Model

Users can read a pre-trained surrogate model and use it as a cheap black-box function. Here, a black-box function means a callable Python function (from the user side) which returns the mean value predicted by the surrogate model for the given task and parameter information. This feature is useful for many interesting scenarios: in-depth analysis using the model function (e.g. a surrogate model-based sensitivity analysis which is our future work), and using the model function to guide autotuning and/or to tune a new problem, to be used for transfer learning.

For flexible use scenarios, users can use a simple wrapper function called `ReadSurrogateModelFunction` to read a surrogate model to use as a function. The wrapper function can take meta information to selectively load surrogate models, where the task information needs to be given in the meta information. If there are multiple surrogate models, by default, the database returns the surrogate model that contains the largest number of function evaluation results. Note that users can also load LCMs trained for multiple tasks (MLA). For this, the user can simply pass all the task information used to train the model when reading the surrogate model. In the following, we show how this feature is used for transfer learning. More detailed usage and available model selection schemes are provided in the User Guide [15].

D. Transfer Learning

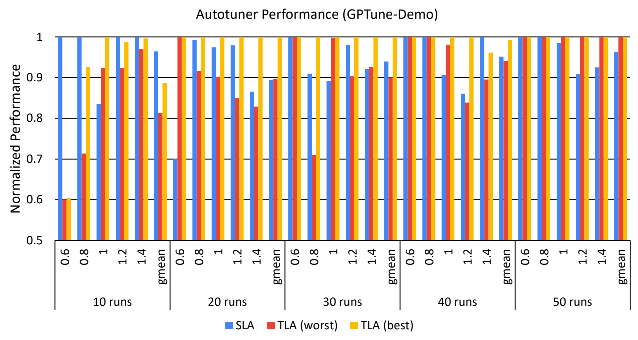
The idea of the proposed TLA approach is to treat TLA as running MLA using samples from true function evaluations on the target task, and from pre-trained surrogate models on previously tuned tasks. In our MLA design, the model update scheme assumes that every task always has the same number of sample function evaluation results. In this regard, to tune a new task, we use the previously-described model function feature to obtain additional samples of the tasks that cannot be run (e.g. results from other machines) or that we do not want to run (e.g. leverage the knowledge of already trained data without actually evaluating the function).

```
1  from autotune import *
2  from gptune import *
3
4  def objectives(point):
5    if point['t'] == 1:
6      return run_and_measure(point)
7    elif point['t'] == 2:
8      return model_function[point['m']]
9
10 def main():
11   # Input/Parameter/Output space definitions
12   IS = Space([Integer(0,10,transform='normalize',
       name='t')])
13   PS = Space([Real(0,1,transform='normalize',name=
       'x')])
14   OS = Space([Real(float('-Inf'),float('Inf'),name
       ='y')])
15
16   # Create GPTune instance
17   problem = TuningProblem(IS, PS, OS, objectives)
18   gt = GPTune(problem)
19
20   # 1: New task, 2: Use a trained surrogate model
21   giventask = [[1],[2]]
22   model_function = ReadSurrogateModelFunction(
23   meta_dict={"task_parameters": [[2,2]]})
24   gt.MLA(Igiven = giventask, NS=20)
```

Listing 3. Example code to run transfer learning with a pre-trained surrogate model.

Listing 3 shows an example of TLA to tune task 1 with a pre-trained model from task 2. As shown in the code, we load a model for task 2 in line 22, and the objective function uses the model when evaluating a sample for task 2. The TLA can also be used to leverage the knowledge obtained from different machine/software configurations. For the same task parameter with different machine/software configurations, GPTune allows the user to add categorical parameters to the task space, with label names `machine_configuration` and `software_configuration`. In other words, TLA can be used to tune a new task by leveraging prior knowledge of historical data. However, the TLA approach can only be applied when the tuning and output parameter space is the same as for the historical data.

We demonstrate the benefits of TLA compared to SLA using two applications, a synthetic demo function and ScaLAPACK's PDGEQRF routine. The demo function is given as $y(t,x) = 1+e^{(-(x+1)^{t+1})}\cos(2\pi x)\sum_{i=1}^{3}\sin\left(2\pi x(t+2)^i\right)$, having one task parameter t and one tuning parameter x. This function is highly non-convex, and finding the optimal parameter is non-trivial. The objective is to find the minimum for $x \in [0,1]$, for the given task t. As a real-world example, ScaLAPACK's PDGEQRF routine has two task parameters for the matrix size, and four tuning parameters for the blocking algorithm (row and column block sizes) and the MPI configuration (number of MPI processors and number of row processes). This application has non-trivial optimal tuning parameters [15] and needs



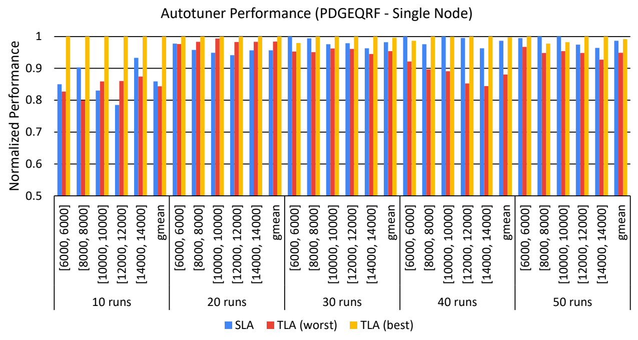


Figure 3. Benefits of TLA on a Cori Haswell node. X-axis represents the task value(s), and Y-axis is normalized performance of the obtained tuning results. In TLA, the surrogate models used for the learning task are trained from 49 samples.

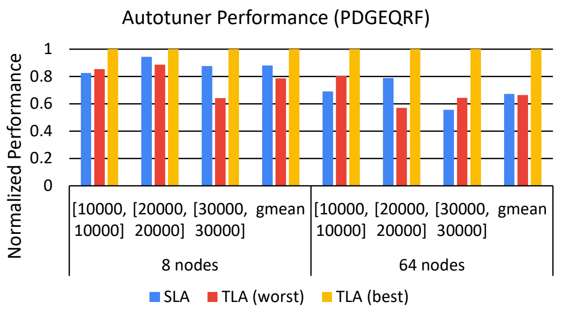


Figure 4. Benefits of TLA on multiple Cori Haswell nodes. Unlike Figure 3, here we assume only 20 runs for each tuning. The pre-trained models used for TLA are trained from 19 samples.

to be evaluated on distributed systems. Therefore, tuning this application can be expensive. We focus on finding the optimal tuning parameter configuration for minimizing the runtime to solve the given input matrix. To evaluate PDGEQRF, we mainly used Intel Haswell compute nodes on NERSC's Cori machine. Each Haswell node has two 16-core Intel Xeon E5-2698v3 processors and 128GB of 2133MHz DDR4 memory.

To compare performance of different autotuning settings, we use normalized performance comparing the best tuning results (minimizing the objective function) obtained from the autotuners; hence, the normalized performance is the best result among all the autotuners divided by the best result of a specific tuner. The TLA performance depends on which task is selected as the pre-trained surrogate model. To evaluate the potential effectiveness of TLA approaches, if there are multiple tasks to choose, we present both `TLA (worst)` and `TLA (best)`, where `TLA (worst)` means selecting the worst task among the available options, and `TLA (best)` means selecting the best task resulting in the best tuning result.

In Figure 3, we start evaluating the TLA approach using a small problem set on a single compute node. Figure 3 compares the best tuning results (the function output in the demo function and runtime in the PDGEQRF routine) obtained from SLA and TLA. Like our intuition, the benefits of TLA can be more prominent when the number of allowed runs is small. Looking at the PDGEQRF results, while TLA achieves marginal improvements (0–5%) when the number of runs is equal or greater than 20 runs, the overall benefit of `TLA (best)` (i.e. when selecting the best task among four available task options for learning) is 14% for 10 runs. In case of the demo function, although SLA achieves better results for 10 runs (this is possible given the nature of random sampling), TLA improves the tuning result for other budgets; the best improvement (10%) was achieved with 20 runs).

In Figure 4, we further evaluate the PDGEQRF tuning for larger problem sizes using 8 nodes (256 cores) and 64 nodes (2,048 cores). The tuning problem here becomes more difficult (but more important) than the single-node experiments in Figure 3 because a larger number of nodes are used, hence the MPI tuning parameters (e.g. number of MPI processes) have larger search spaces. As shown in the result, TLA improves the tuning results by a significant margin, compared to the single-node experiments. On 64 Cori nodes, for example, `TLA (best)` improves the tuning result by 31% (task 10,000), 21% (task 20,000) and 44% (task 30,000), respectively, with a geometric mean of 33% for those three tasks. Since application performance has higher variability on multi-node environments (depending on the specific node allocation), we run the experiments three times and report their average.

TLA can also be used to leverage previous knowledge obtained from a different machine. Figure 5 shows an evaluation of TLA using an exsiting model trained on a different machine, a Cori Knights Landing (KNL) node which includes an Intel Xeon Phi processor 7250. TLA achieves marginal improvements for the left three tasks but has small performance degradation for the right two tasks. The results imply that, without using good pre-trained models, the TLA approach may not lead to significant improvements, but TLA generally does not suffer noticeable performance drops.

As shown by these overall results, the performance of

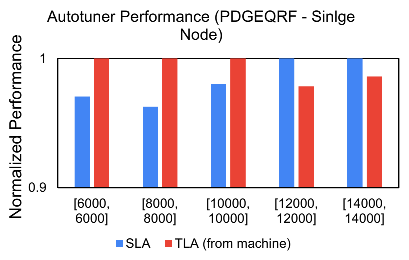


Figure 5. Benefits of TLA on a Cori Haswell node using a trained model on a Cori KNL node for the same task parameter. The number of runs for tuning is 20, and the pre-trained models used for TLA are trained from 41 (task [6000,6000]), 49 (tasks [8000,8000] and [10000,10000]), 40 (task [12000,12000]), 35 (task [14000,14000]) samples.

TLA depends on which model or task is used for TLA. Choosing appropriate models that are most relevant to the tuning task is the key to achieving the best TLA performance. We leave this as our future work.

For reproducibility, we provide the surrogate models used in the TLA experiments in our shared repository. Section V-E explains how to use the repository to access the data.

E. Crowd-tuning

Here, we present a brief introduction to our shared repository designed for crowd-tuning. For the details of the repository, please refer to our website at `https://gptune.lbl.gov/`. To assure provenance and avoid uploading bad data, the repository requires login credentials to submit any data. There are multiple accessibility options for performance data: publicly available, private, and shared with specific users/groups.

1) Downloading Performance Data: To browse and download performance data from the repository, we provide an interactive web dashboard. In the dashboard, users can select a specific tuning problem from the drop-down menu. Once the tuning problem is selected, the dashboard will display all available machine configurations, software configurations, and owner information (i.e. who submitted the data). The user can check the machine/software/user configuration(s) that match the user's interests. Then, the dashboard will display a table that contains all the filtered results that the user can access, and the user can download data in the JSON format. Interested readers can try this feature by downloading the surrogate models used in Section V-D. The models can be found by searching data for the `PDGEQRF-ATMG` application on the dashboard. The experiment scripts and their usage are available at the GPTune's Github repository [3].

[3]`https://github.com/gptune/GPTune/tree/master/examples/ScaLAPACK-PDGEQRF-ATMG`

2) Uploading Performance Data: The user can upload function evaluation results and/or surrogate models using the web interface. The user first needs to select the tuning problem and the machine used for generating the data. If the user's tuning problem or machine is not shown in the drop-down menu, the user needs to add the tuning problem or machine information to the repository. After the user uploads the obtained JSON performance data, the shared repository automatically checks if there are duplicated data in the repository, and stores only new data to the repository. The repository also checks if the submitted data match the problem space of the tuning problem and if the submitted data provide all the necessary machine configuration information.

3) Adding Tuning Problems: Before uploading any performance data, users need to define their tuning problems in the shared repository using our web interface, unless the same tuning problem already exists in the repository. With the (well-defined) tuning problem information, multiple users can run the tuner for the same tuning problem. The user needs to define the tuning name by selecting appropriate category/categories and defining the task space, the tuning parameter space, and the output space. The user also needs to provide which software and which type of information is needed for the problem. We have a list of widely used software packages/tools obtained from the CK's software database [12], which can be automatically detected by CK. The repository will assign a unique name by combining the user-provided tuning problem name by username and the date of submission; hence users can differentiate between tuning problems, tuning the same program with different settings.

4) Adding Machine Information: The user also needs to define machine information, unless the user's machine information is already available in the repository. The user first needs to provide the machine name and the site/institution information. The user can then select the system model type (e.g. HPC system manufacturer, cloud service provider, etc.). Another important data field is to provide the processor types of the machine. We have built a list of processors from popular (HPC) processor vendors such as Intel, NVIDIA, IBM, AMD, ARM, etc. The user can select one or multiple processor types (in case of heterogeneous systems) that make up the user's machine and provide information about the number of nodes/cores contained in the machine. The user can finally select the interconnect(s) of the machine. In case there are multiple records for the same machine (this is possible as the system can continue to consolidate more system resources), the user can still choose one machine record that best suits the user's tuning configurations.

VI. Conclusion and Future Work

GPTune is a performance autotuner based on multi-output Gaussian process regression. In this paper, we presented a history database for enhancing the autotuning capability of GPTune and reproducibility of performance data. The history database allows users to achieve the best possible tuning results for widely used codes, as well as a TLA approach that can leverage the knowledge of already trained models to tune a new tuning task. Our evaluation of TLA shows that, for the ScaLAPACK's PDGEQRF routine, TLA can attain up to 33% better tuning results on the NERSC's Cori supercomputer.

In future work, we plan to improve the user interface for the web-based shared repository. For example, we can provide a programmable Python interface to access performance data in the shared repository. Such an interface can provide more flexible use cases without needing to manually download data using a web browser. In addition, we plan to improve the TLA approach by using more sophisticated and intelligent (and automated) model selection for learning. Another future work is providing a surrogate model-based sensitivity analysis to determine how different values of individual tuning parameters could affect the output results.

Acknowledgement

This research was 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. We used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

[1] P. Balaprakash, R. Egele, and P. Hovland, "ytopt: Machine-learning-based Search Methods for Autotuning," 2019. [Online]. Available: https://github.com/ytopt-team/ytopt

[2] 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*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds. PMLR, 2018, pp. 1437–1446.

[3] H. Menon, A. Bhatele, and T. Gamblin, "Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 831–840.

[4] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, *Bliss: Auto-Tuning Complex Applications Using a Pool of Diverse Lightweight Learning Models*. New York, NY, USA: Association for Computing Machinery, 2021, p. 1280–1295. [Online]. Available: https://doi.org/10.1145/3453483.3454109

[5] Y. Liu, W. M. Sid-Lakhdar, O. 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

[6] P. I. Frazier, "A Tutorial on Bayesian Optimization," *arXiv preprint arXiv:1807.02811*, 2018.

[7] F. Hourdin, T. Mauritsen, A. Gettelman, J.-C. Golaz, V. Balaji, Q. Duan, D. Folini, D. Ji, D. Klocke, Y. Qian *et al.*, "The Art and Science of Climate Model Tuning," *Bulletin of the American Meteorological Society*, vol. 98, no. 3, pp. 589–602, 2017.

[8] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An Extensible Framework for Pprogram Autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[9] CK-GPTune, "CK-GPTune: CK for GPTune," 2021. [Online]. Available: https://github.com/gptune/CK-GPTune

[10] G. Fursin, "Collective knowledge: Organizing Research Projects as a Database of Reusable Components and Portable Workflows with Common APIs," *arXiv preprint arXiv:2011.01149*, 2020.

[11] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall, "Autotuning Polybench Benchmarks with LLVM Clang/Polly Loop Optimization Pragmas using Bayesian Optimization," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 61–70.

[12] G. Fursin, "CK Soft Module for Managing Software (registering in ck environment)," https://cknowledge.io/c/module/soft/.

[13] A. Jain, S. P. Ong, G. Hautier, W. Chen, W. D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder, and K. A. Persson, "Commentary: The materials Project: A Materials Genome Approach to Accelerating Materials Innovation," *APL Materials*, pp. 1–11, 2013. [Online]. Available: https://doi.org/10.1063/1.4812323

[14] Autotune, "Autotune," https://pypi.org/project/autotune/, 2018.

[15] W. M. Sid-Lakhdar, Y. Cho, J. W. Demmel, H. Luo, X. S. Li, Y. Liu, and O. Marques, "GPTune User Guide," 2021. [Online]. Available: https://github.com/gptune/GPTune

[16] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," in *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 263–273.

[17] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc., 2013.

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719642