Harnessing the Crowd for Autotuning High-Performance Computing Applications

Younghyun Cho, James W. Demmel
University of California, Berkeley
{younghyun,demmel}@berkeley.edu

Jacob King
Tech-X
jking@txcorp.com

Xiaoye S. Li, Yang Liu, Hengrui Luo
Lawrence Berkeley National Laboratory
{xsli,liuyangzhuan,hrluo}@lbl.gov

***Abstract*—This paper presents GPTuneCrowd, a crowd-based autotuning framework for tuning high-performance computing applications. GPTuneCrowd collects performance data from various users using a user-friendly tuner interface. GPTuneCrowd then presents novel autotuning techniques, based on transfer learning and parameter sensitivity analysis, to maximize tuning quality using collected data from the crowd. This paper shows several real-world case studies of GPTuneCrowd. Our evaluation shows that GPTuneCrowd's transfer learning improves the tuned performance of ScaLAPACK's PDGEQRF by 1.57x and a plasma fusion code NIMROD by 2.97x, over a non-transfer learning autotuner. We use GPTuneCrowd's sensitivity analysis to reduce the search space of SuperLU_DIST and Hypre. Tuning on the reduced search space achieves 1.17x and 1.35x better tuned performance of SuperLU_DIST and Hypre, respectively, compared to the original search space.**

***Index Terms*—autotuning, crowd-based autotuning, transfer learning, sensitivity analysis, Exascale Computing Project**

I. Introduction

High-Performance Computing (HPC) codes [1]–[4] usually have multiple tuning parameters that need to be optimized for a given system. Autotuning has therefore gaining importance to tune HPC codes with minimal human efforts. Bayesian optimization (BO), that attempts to find an optimal parameter configuration within a limited number of trials, is an attractive approach to tuning HPC applications that are expensive to evaluate and hard to model. Many recent autotuners [5]–[8] started to employ BO. The approach runs and evaluates the code with carefully chosen tuning parameter configurations, and (iteratively) builds a performance model (i.e., a surrogate model) based on the measured performance (i.e., function evaluations) and uses the performance model to search the optimal tuning parameter configuration.

Although BO promises to find a good parameter configuration for a given tuning budget, autotuning is still expensive, which hinders wide adoption of autotuning in many HPC domains. For instance, a survey [9] has shown that the majority of climate model experts is skeptical of adopting autotuning for tuning climate models. There are several fundamental challenges. First, optimal tuning parameters will vary from platform to platform. Each individual user therefore needs to perform autotuning from scratch. Second, BO requires evaluations for sufficient parameter configurations in order to have an accurate surrogate model, especially when the given search space is large. Running and measuring an HPC code on a large-scale machine can be extremely expensive.

In this paper, we present a crowd-based autotuning (crowd-tuning) approach that leverages the power of the crowd to overcome these autotuning challenges. We see that for popular applications there are multiple users who need to tune the same application (across different software/machine settings), and there will be performance data samples that can be collected over time. We aim to leverage performance data samples accumuluated from various users using Transfer Learning-based Autotuning (TLA). In our context, TLA means that we tune a given problem (target task) with pre-collected datasets from different problems (source tasks). TLA allows us to exploit existing performance data from various hardware and software platforms (from the crowd) to tune HPC codes (for the current tuner user) using a minimal tuning budget. Furthermore, we provide data analytics using collected data such as parameter sensitivity analysis. Such parameter sensitivity information can be used to reduce the tuning search space of a tuning problem.

Interfacing the tuner with the collected data is key to crowd-tuning. Although there are several performance repositories for sharing performance data among different users [10], [11], these existing works lack a user-friendly programming interface for autotuning, which makes it increasingly difficult to submit or retrieve relevant data, as the size of the database grows. For TLA, several autotuners such as GPTune [8], [11], HiPerBOt [6], and Vizier [12] use a TLA algorithm in their BO framework. However, we observe that there is no one-size-fits-all TLA algorithm given the various characteristics of tuning problems, due to their stochastic nature. In addition, to our best knowledge, none of the aforementioned tuners provides parameter sensitivity analysis. Our work focuses on the three aspects of crowd-tuning: (1) enhancing programmability of the shared database, (2) maximizing the benefit of transfer learning for a variety of tuning problems, and (3) enabling a sensitivity analysis tool to provide insights into a tuning problem.

To this end, we present GPTuneCrowd, a crowd-based autotuning framework for tuning HPC applications. Figure 1 illustrates the crowd-tuning workflow. GPTuneCrowd leverages a shared database infrastructure to collect performance data. The shared database uses database management systems to securely manage performance and user data and provides useful web-based tools that help users browse collected data. GPTuneCrowd also provides a user-friendly tuner interface. Users only need to provide a simple meta description about the given tuning problem, then our tuner can query and upload

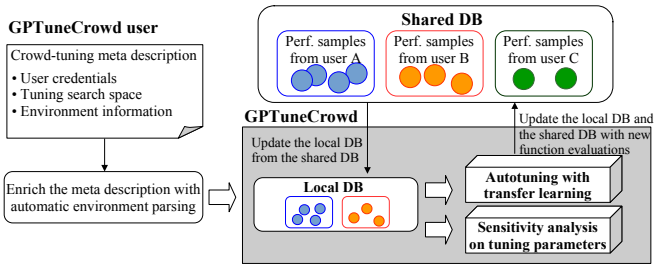

Fig. 1. Overview of GPTuneCrowd.

performance data using the shared database, along with several automatic environment parsing techniques to record and/or match the runtime environment information (e.g., software and machine configuration). GPTuneCrowd also provides a number of utilities for data analytics such as parameter sensitivity analysis and performance prediction.

GPTuneCrowd uses transfer learning to maximize tuning quality (i.e., minimizing the number of samples needed to attain (nearly) optimal performance). GPTuneCrowd has a pool of TLA algorithms which includes three existing algorithms, a multitask learning-based approach from GPTune [11], a weighted sum-based approach from HiPerBOt [6], and a model stacking approach from Vizier [12]. In addition, we provide two improved algorithms for the multitask learning-based weighted sum-based approaches that outperform the original methods in [11] and [6]. Furthermore, GPTuneCrowd provides a novel ensemble technique that can benefit from multiple TLA algorithms. Our ensemble approach dynamically selects a TLA algorithm from the algorithm pool during autotuning based on a probability distribution function. The approach can fully automatically choose the best result from different algorithms.

We present a number of real-world use cases of GPTuneCrowd. We first provide a thorough comparison of a variety of TLA algorithms using two synthetic functions. Then, we apply GPTuneCrowd's transfer learning to tune large-scale real-world applications, ScaLAPACK's PDGEQRF [1] and a fusion simulation code NIMROD [4]. These experiments show that how performance data collected from different tasks to tune the code for another task, or how data from different machine configuration can be used to tune on another machine configuration. For example, on NERSC's Cori supercomputer, with 100 pre-collected performance samples for a source task, GPTuneCrowd improves the tuned performance (runtime) of PDGEQRF (with 10 function evaluations) up to 1.57x (36.2%) compared to non-TLA autotuning. With 500 pre-collected performance data samples collected from 32 Haswell nodes, GPTuneCrowd improves the tuned performance of NIMROD (with 10 function evaluations) up to 2.97x (66.4%) on 64 Haswell nodes over non-TLA autotuning. In addition, we use GPTuneCrowd's sensitivity analysis to reduce the tuning space of SuperLU_DIST [2] and Hypre [3]. The reduced-size tuning attains 1.17x and 1.35x better results for SuperLU_DIST and Hypre, respectively, compared to tuning on the original tuning search space, based 10 function evaluations.

This paper makes the following contributions:

- We provide a user-friendly crowd-based autotuning framework that requires only a simple meta description for tuning with collected performance data (Section IV).
- We improve existing transfer learning methods and provide an ensemble technique to maximize tuning quality using a pool of transfer learning algorithms (Section V).
- We provide an evaluation of existing transfer learning algorithms along with our new algorithms (Section VI).
- We demonstrate the effectiveness of GPTuneCrowd using real-world HPC applications on large-scale machines (Section VI).

GPTuneCrowd is built on top of an open-source autotuner GPTune [8] and incorporated into the GPTune package that also contains several other useful autotuning techniques [8], [13]–[15]. The GPTune package is available at *https://github.com/gptune/GPTune*. Our user guide [16] provides more detailed user interfaces and installation options of the GPTune package (including GPTuneCrowd) for various user systems. Our shared database is located at *https://gptune.lbl.gov*.

II. Related Work

A. Autotuners for HPC

Recent research in HPC autotuners develops growing interest in Bayesian optimization (BO) to tune expensive-to-evaluate HPC codes. HiPerBOt [6] and GPTune [8] use Gaussian Process (GP) regression to build a surrogate performance model. GPTune proposes multitask learning-based autotuning using a Linear Coregionalization Model (LCM), where tuning multiple correlated tuning problems simultaneously can benefit from each other. Zhu et al. present GPTuneBand [13] that combines multitask learning with a multi-armed bandit strategy. BLISS [7] uses an ensemble of multiple surrogate models. Ytopt [5] supports multiple machine learning techniques within BO. GPTuneCrowd, on the other hand, focuses on maximizing tuning efficiency using collected data from various users using transfer learning.

B. Crowd-based autotuning

There are several existing works for sharing performance data among different users. CK [10] is an interface and tool for reproducible workflow automation, and provides a repository that permits uploading users' performance results and downloading results from other users. GPTune's history database (in 2021) [11] also provides a repository for sharing performance data samples obtained from autotuning.

GPTuneCrowd improves these existing databases as follows. First, we provide a programmable interface that enables users to write an SQL-like query to retrieve relevant performance data. Second, we improve the performance data collection scheme. GPTuneCrowd can automatically download and upload performance data for a given problem (if the user agrees on), without needing to access the web repository manually, and automatically record the HPC runtime environment information to assure the reproducibility. Third, our database provides useful data analysis tool to exploit the collected data

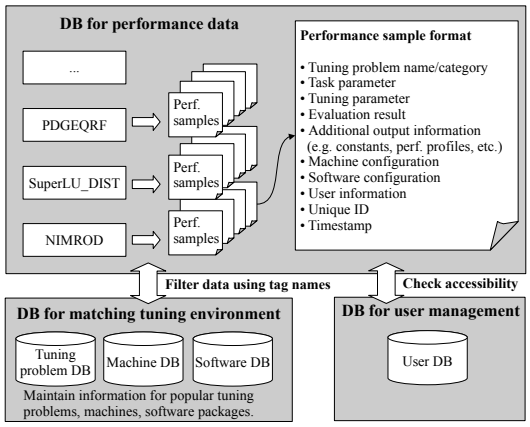

Fig. 2. Shared database structure.

such as performance prediction and sensitivity analysis. Lastly, beyond the existing works, GPTuneCrowd offers advanced TLA techniques, and this paper provides a thorough comparison of various TLA algorithms.

Outside the computer science community, the Materials project [17] provides a repository to share materials information computed using supercomputing resources. They provide a programmable interface called Materials API (MAPI) to query materials information, and their work inspired the development of our crowd-tuning interface.

C. Transfer learning

To the best of our knowledge, among the various HPC autotuners, GPTune [11] and HiPerBOt [6] offer a transfer learning approach. GPTune's transfer learning [11] relies on a multitask learning technique using LCM [8] that models the correlation of multiple different tasks. HiPerBOt [6] uses a weighted sum of the surrogate models of both source and target tasks to combine the knowledge of the source and target tasks. In their method, weight values are statically determined for both source and target tasks, and their paper does not discuss how to compute weight values for various tuning problems. Google's Vizier [12] is an autotuner for their internal services. Vizier uses a stacking approach which models the residuals among the source and target tasks. The stacking method then accumulates the modeled residuals and use the stacked model for parameter search. GPTuneCrowd not only offers all these methods but also improves the multitask learning-based and weighted sum-based approaches and provides a new ensemble technique to combine the benefits from different TLA algorithms (Section V).

III. CrowdTune Infrastructure

Figure 2 illustrates the structure of the shared database. The shared database manages collected performance samples in a JSON (JavaScript Object Notation) form using MongoDB [18]. The JSON data format is human readable, and the data can thus be used for various autotuning frameworks. All performance data sample contains *task parameters*, *tuning parameters*, and its output (*evaluation result*). Task parameters (i.e., tasks) describe the application may encounter (e.g., the sizes of matrices, pointers to input files). Tuning parameters are the tuning parameter configuration of the performance data sample (e.g., size of row/column blocks). Output is the evaluation result (e.g., measured runtime) of the task and tuning parameter configuration.

For reproducibility, the database can also record the runtime environment information of each performance sample. The information includes the machine and software configuration on which the code is evaluated. To obtain such information, users can describe the given machine and software configuration, however, human-provided information requires lots of human efforts and is prone to error. To address this challenge, GPTuneCrowd supports several automatic environment parsing techniques to parse and record the reproducibility information without manual input. Since the runtime environment can be given manually, different users might use different names to describe the same machine and software configuration. The shared database therefore internally parses the user provided information to match the tag names with the well-defined machine software information existing in the database. As shown in Figure 2, we maintain separate databases for the detailed information of popular software frameworks and user systems with possible tag names for this purpose.

For reliability, the repository allows only registered users to upload and record user information (e.g., username and email), but we also provide some options for users who do not wish to disclose their information to other users. Each performance data sample can have a different level of accessibility, e.g., the data can be publicly available, private, or shared with specific users or groups. To assure the security of the database, we use a number of techniques such as robot checking, user password protection, and regular vulnerability checking.

IV. Programming GPTuneCrowd

A. Meta description and automatic environment parsing

To leverage collected data in the shared database, GPTuneCrowd requires users to provide only a meta description about the given tuning problem. The necessary information includes the user's login credentials (API key), the tuning problem name, the tuning parameter search space, and the runtime environment information. The following code snippet shows an example of meta description.

```
{
  api_key = "your_api_key",
  tuning_problem_name = "my_example",
  problem_space = {
    "input_space": [
      {"name":"t", "type":"integer",
       "lower_bound":1, "upper_bound":10}
    ],
    "parameter_space": [
      {"name":"x", "type":"real",
       "lower_bound":0, "upper_bound":10},
    ],
    "output_space": [
      {"name":"y", "type":"real"}
    ]
  },
  configuration_space = {
    "machine_configurations": [
      {"Cori":{"haswell":{"nodes":1,"cores":32}}}
    ],
    "software_configurations": [
      {"gcc":{"version_from":[8,0,0],
              "version_to":[9,0,0]}}
    ],
    "user_configurations": ["user_A", "user_B"]
  },
```

```
    machine_configuration = {
        "machine_name": "Cori",
        "slurm": "yes"
    },
    software_configuration = {
        "spack": "ScaLAPACK"
    },
    sync_crowd_repo = "yes"
}
```

To use GPTuneCrowd, the user needs to have an API (Application Programming Interface) key and provide it in the meta description. Individual users can generate one or more API keys at our database website. Users have to manage their API keys securely, because API keys are user login credentials. By default, an API key is simply a random string of 20 characters/digits. To enhance the security of API keys, users can also use public and private key pairs for API keys. In this case, the user needs to keep the private key and we record only the public key in our user database.

The `tuning_problem_name` is used to distinguish between different tuning problems. The meta description has two types of database query parameters, `problem_space` and `configuration_space`. The `problem_space` parameter is used to describe the parameter range that the user wants to query from the shared database. For example, the `input_space` can refer to what kind of task information (e.g., the problem size) to be queried and `parameter_space` describes what tuning parameter types to be queried. The `lower/upper_bound` parameters define the range of task parameter values to be queried. In case the parameter is categorical type, the user can provide the list using `categories` field. The `configuration_space` parameter, on the other hand, describes the list of environment information, i.e., the machine, software, and the user information that the user wants to allow to download. The example means that the user wants to query performance data samples obtained from Cori system using one Haswell node, and the user gives a restriction to query data obtained from a GCC compiler with a version between 8.0.0 and 9.0.0. Lastly, users may want to trust data submitted by specific users. The user can provide the list of usernames (or email addresses) in the `user_configuration` field. Users may look up available user entries from the shared database website. If these condition information is not given, a query will download all data available to the user.

The `machine/software_configuration` parameter is used to record the runtime environment of the user. The information will be appended to each function evaluation and uploaded to the shared database (if `sync_crowd_repo`="yes"). Since GPTuneCrowd is designed for tuning HPC applications, we support several automatic environment parsing for popular HPC environments. In case the HPC code is installed with an automatic installation tool such as Spack [19] or CK [10], the user can inform the database about which software is installed with Spack or CK, so that our database can automatically parse the software configuration and record it in the database. In addition, if the workload is executed via the Slurm workload manager, our database can also record the node allocation and the machine information automatically.

Crowd-tuning API is obviously easier-to-use than working with a web interface, especially when writing a tuning script.

B. Utility functions and data analytics

GPTuneCrowd provides a set of utility functions in Python to exploit the queried data for data analysis. The functions include *QueryFunctionEvaluations*, *QuerySurrogateModel*, *QueryPredictOutput*, and *QuerySensitivityAnalysis*. For example, the user can simply call these functions using an API key and the problem description to query from the GPTuneCrowd's shared database.

```
import crowdtune
ret = crowdtune.QueryFunctionEvaluations(
        api_key = api_key,
        tuning_problem_name = "Example",
        problem_space = problem_space,
        configuration_space = configuration_space)
```

QueryFunctionEvaluations returns a list of queried function evaluation results. *QuerySurrogateModel* allows the user to get a surrogate performance model based on the queried performance data samples. Here, the user can choose a specific surrogate modeling technique among several modeling options. The queried surrogate model is a black-box function; once the input parameter set is given the function will return a predicted output value. *QueryPredictOutput* is similar to *QuerySurrogateModel*, but instead of returning a surrogate model, it returns the predicted output for the given parameter configuration.

QuerySensitivityAnalysis is one of the unique features of GPTuneCrowd among other existing tuners. It queries the relevant performance data samples and build a surrogate performance model and conduct a parameter sensitivity analysis using the model. This function provides options to specify the surrogate modeling method and the sensitivity analysis methods. GPTuneCrowd currently offers an interface for a sensitivity analysis tool based on using the Sobol analysis [20] for the learned surrogate model and using the implementation of `SALib` [21]. The Sobol method is a global sensitivity analysis providing an assessment of parameter sensitivity. The Sobol analysis requires (1) samples drawn from the trained surrogate model directly, (2) evaluating the model using the generated sample inputs and (3) conducting a variance-based mathematical analysis to compute the sensitivity indices. The Sobol analysis evaluates the part of the total variance of the response that can be attributed to the input parameter X_n. In the Sobol method, we usually consider two measures: A first-order (main-effect) index $S1_i$ is the contribution to the output variance of the main (linear) effect of an input attribute X_i, therefore it measures the effect of varying X_i alone, but averaged over variations in other input parameters. It is standardised by the total variance to provide a fractional contribution. A total effect index ST_i represents the total contribution (including interactions among parameters) of a parameter X_i to the response variance; it is obtained by summing all first-order and higher-order effects involving the parameter X_i.

V. Transfer learning algorithms

GPTuneCrowd contains a pool of multiple TLA algorithms and provides an ensemble approach that benefits from multiple

TLA algorithms. Table I summarizes the TLA algorithm pool of GPTuneCrowd. The table contains three existing techniques, `Multitask(PS)` from [11], `WeightedSum(static/equal)` from [6], and `Stacking` from [12], and two improved version of multitask and weighted sum-based approaches, `Multitask(TS)` and `WeightedSum(dynamic)`, and finally our ensemble technique `Ensemble(proposed)`. In what follows, we provide the details of these algorithms.

A. Multitask learning

This approach treats transfer learning as multitask learning to tune the new task with the available data from the source tasks. The method exploits pre-collected function evaluation data and/or pre-trained surrogate performance models of source tasks and run the LCM-based multitask surrogate modeling for both source and target tasks.

1) Using black-box surrogate models of source tasks (pseudo samples): In 2021, GPTune [11] proposed to exploit pre-trained surrogate performance models of source tasks and run multitask learning for both source and target tasks. The multitask modeling uses the Linear Coregionalization Model (LCM) [8] in order to model the correlation of the source and target tasks. The LCM model is used to predict the next sample for all the source and target tasks. As transfer learning is not intended to evaluate new samples for the source tasks, the approach uses mean values predicted by the source surrogate model as a black-box function to generate pseudo samples for the source task. It only runs and evaluates the target task for the new sample. The new samples, true samples from the target task and pseudo samples from the source tasks, are used to iteratively build an LCM model. We dub this TLA scheme as `Multitask(PS)`, where PS represents pseudo samples from the black-box surrogate models.

2) Using true performance samples of source tasks: Using black-box surrogate models for source tasks does not fully exploit the collected knowledge (all the performance samples) of source tasks. Thanks to the shared database support that allows us to access the collected datasets, here we focus on utilizing all the collected true samples of the source tasks, by supporting unequal number of samples per task within the LCM modeling. It starts with building an LCM model with the pre-trained surrogate models for the source tasks and zero sample for the target task. The proposed scheme starts with building a LCM model with many samples for the source task and zero sample for the target task. The LCM model will predict the next sample only for the target task. The new sample is then evaluated and the LCM model is updated with one more sample from the target task. This approach is labeled as `Multitask(TS)` in Table I, where TS represents true samples for the source tasks. Our evaluations in Section VI-A show the benefit of `Multitask(TS)` over `Multitask(PS)`.

B. Weighted sum using static weights

This approach builds a GP surrogate model for each of the source and target tasks. Then, the approach combines the surrogate models to explore the parameter space for the target task, by using the arithmetic sum of the surrogate models for the mean function and the geometric mean of the surrogate models for the standard deviation function. For the combined GP model $f(x) \sim GP(\mu(x), \sigma(x))$, the mean function and standard deviation function are computed as follows:

$$\mu(x) = w_{target} \cdot \mu_{target}(x) + \sum_{i=1}^{n_{src}} w_{src_i} \cdot \mu_{src_i}(x) \quad (1)$$

$$\sigma(x) = (\sigma_{target}(x)^{w_{target}} \cdot \Pi_{i=1}^{n_{src}} (\sigma_{src_i}(x)^{w_{src_i}})) \quad (2)$$

where w_{src} and w_{target} are the weights for source and target tasks, respectively. μ_{src} and σ_{src} is the mean and standard deviation function of the source task's surrogate model, and there are n_{src} source tasks. μ_{target} and σ_{target} are the mean and standard deviation functions of the target task. n_{src} represents the number of source tasks.

We label this approach as `WeightSum(static)`. This approach is used in HiPerBOt [6], however, the paper does not discuss how to compute the weight values for various problems. Therefore, if weights are not specified by the user (most cases), we simply use an equal weight 1 for all source and target tasks, which is labeled as `WeightSum(equal)`.

C. Weighted sum using dynamic weight computation

The weighted sum-based approach using static weights may work ideally only if the proper weights are given by the user. However, proper weight values are usually unknown. To address this challenge, GPTuneCrowd provides a dynamic approach to computing weights for the source and target tasks. To determine the weights, for an individual GP surrogate model (both source and target tasks), for each of the observed samples for the target task we compute the difference between the predicted output of the sample's parameter configuration and the predicted output for the best parameter configuration observed so far. Then, we compare the differences from the prediction to the differences from the actual observations (for the current target task). Considering the linear regression problem with coefficients $w_{src_1}, \ldots, w_{src_{n_{src}}}, w_{target}$ and different GP surrogates $G_1, \ldots, G_{n_{src}}, G_{target}$ and observed sequential samples $(x_1, y_1), \ldots, (x_N, y_N)$.

$$y^* - y_j = \sum_{i=1}^{n_{src}} w_{src_i} \cdot [\mu_{src_i}(x^*) - \mu_{src_i}(x_j)] +$$
$$w_{target} \cdot [\mu_{target}(x^*) - \mu_{target}(x_j)], j = 1, \ldots, n_s$$

where $y^* = f(x^*)$ is the current minimum black-box function value, and $\mu_i(\cdot)$ is the predicted mean of the GP surrogate model i. Since we are assuming a minimization problem, the left-hand side (LHS) is always non-positive. The larger the LHS, the closer this observation is from the current (observed) minima. Therefore, if we have a good combination of weights $w_1, \ldots, w_d$, the linear regression should have a good fit. Then, we compute the weight values for individual surrogate models at each iteration of the tuning for both the target and the source tasks, and apply the weight values in the weighted sum equation (Equation 1) in the search phase. Note that, the LHS and RHS of the linear regression problem of the above

TABLE I
THE TLA ALGORITHM POOL IN GPTUNECROWD.

Naming	Description	First autotuner (to our best knowledge)
Multitask (PS)	LCM-based multitask learning using pseudo samples from black-box surrogate models of the source tasks.	[11]
Multitask (TS)	LCM-based multitask learning using true samples of the source tasks.	GPTuneCrowd
WeightedSum (static/equal)	Weighted sum of surrogate models of source and target tasks using static weight values (if specified) or equal weights (if not specified).	[6]
WeightedSum (dynamic)	Weighted sum of surrogate models of source and target tasks using dynamically chosen weight values using a linear regression approach.	GPTuneCrowd
Stacking	Modeling the residuals between the surrogate models using the posterior mean of the source tasks and aggregating them.	[12]
Ensemble (proposed)	Dynamically choose a TLA algorithm for each function evaluation of the target task using Algorithm 1.	GPTuneCrowd

equation is further normalized to y^* and $G_i(x^*)$, because source and the target tasks can have different output scales.

D. Stacking surrogate models

Google's Vizier [12] proposes a transfer learning approach that uses the posterior mean of the source tasks to combine with the surrogate model of the target task. The method starts with building a surrogate model of an initial source task. Then, for the second source task, it trains another GP model for the residual between the second source task and the initial surrogate model, and combine the initial GP model with the residual model. It then repeatedly adds residual models for the remaining source tasks and finally the target task. The combined posterior mean is then computed as follows:

$$\mu(x) = \mu'_{target}(x) + \sum_{i=1}^{n_{src}} \mu'_{src_i}(x)$$

where μ'_i (when $2 \leq i \leq n_{src}$) is the GP mean function for the residuals between the observed samples for the source task i and the predicted mean of μ'_{i-1}. μ'_1 is the mean function of the surrogate model for the first source task, and μ'_{target} is the GP mean function for the residuals between the target task's observed samples and the predicted mean of $\mu'_{n_{src}}$.

The standard deviation function computation is also computed iteratively using a weighed geometric mean, based on the number of samples, as follows:

$$\sigma(x) = (\sigma'_{target}(x))^{\beta} \cdot (\sigma'_{src_{n_{src}}}(x))^{1-\beta}$$

where $\beta = \frac{n\_samples_{target}}{(n\_samples_{target}+n\_samples_{src_{n_{src}}})}$. The computation for $\sigma'_{src_i}(x)$ ($1 \leq i \leq n_{src}$) is analogous; it computes the weighted geometric mean between source task i and $i-1$.

GPTuneCrowd's TLA pool also contains Vizier's algorithm. One would expect that the sequence (ordering) of the source surrogate models can affect the quality of the combined model. We order the source tasks based on the number of available samples (the first task has the largest number of samples).

E. Ensemble of transfer learning algorithms

Finally, GPTuneCrowd provides an ensemble technique to combine the benefits from different TLA algorithms and attain a nearly optimal solution for a variety of tuning problems. This approach is labeled as `Ensemble(proposed)`.

Algorithm 1 Proposed ensemble approach to TLA.

```
T = list of TLA approaches (by default, T={Multitask (TS), WeightedSum (dynamic), Stacking})
NS = the tuning budget (number of samples to collect)
n_samples = number of obtained samples for the target task (set 0 unless there are existing samples for the target task)
while n_samples ≤ NS do
    r = Choose a value [0,1) uniformly at random
    if r < ExplorationRate from Equation 4 then
        TLA_alg. = Choose one from T uniformly at random
    else
        TLA_alg. = Choose one from T based on the probability distribution function from Equation 3.
    end if
    Build a TLA model using the chosen TLA algorithm.
    Search x the next parameter configuration to evaluate.
    Compute y(x) at the new parameter configuration x and update the database.
    n_samples = n_samples + 1.
end while
Return x_opt and y_opt
```

Algorithm 1 shows the procedure of the proposed technique, labeled as `Ensemble(proposed)`. First, we define the set of TLA algorithms; by default, we use a set of `Multitask(TS)`, `WeightedSum(dynamic)` and `Stacking`. Note that the tuner uses an iterative approach, where the tuner suggests a promising parameter configuration and evaluates the suggested parameter configuration to re-build a surrogate model. Our idea is to build a Probability Distribution Function (PDF) after each function evaluation based on the best parameter configurations found by each TLA algorithm. The probability of each TLA algorithm is computed as follows:

$$prob(t) = \frac{1/best\_output(t)}{\sum_{t_{iter} \in T}(1/best\_output(t_{iter}))} \quad (3)$$

where we assign a higher probability for TLA algorithms that suggested better parameter configurations, assuming we are minimizing the objective and the objective is a non-negative value (e.g., runtime or memory-consumption optimization). However, using only a probability distribution function is not sufficient because only the chosen TLA algorithm has a chance

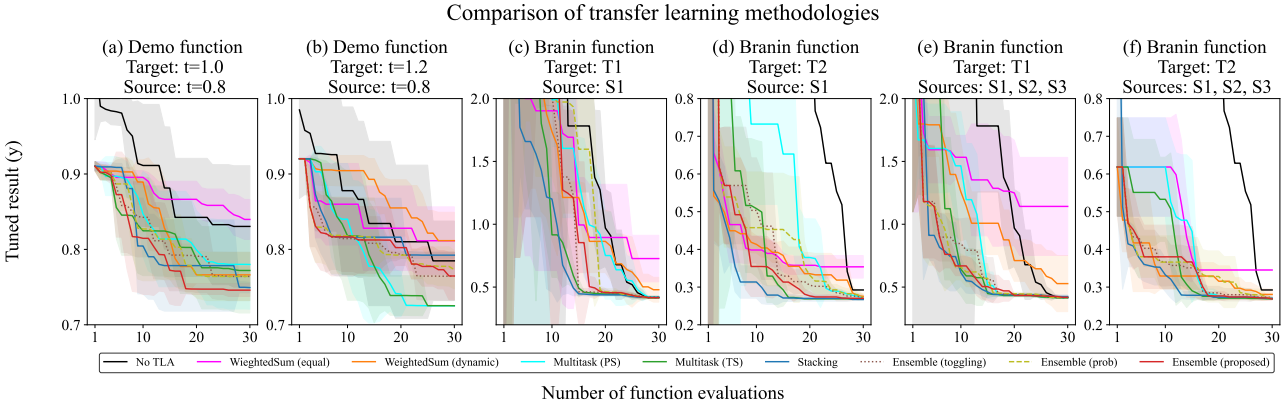

Fig. 3. Tuned results depending on the number of function evaluations. It compares different TLA algorithms using the demo and Branin synthetic functions. For TLA, (a)–(d) use one source task, and (e)–(f) use three source tasks. For each source task, we collected 200 samples for randomly chosen parameter configurations. For the demo function (a)–(b) that contains one task parameter, we specify the source (t=0.8) and target tasks (t=1.0 in (a) t=1.2 in (b)). For the Branin function (c)–(d) that contains six task parameters, we randomly choose the source and target tasks (shown by S1–S3 and T1–T2). We run each tuning experiment five times (with different random seeds). The line charts show the average of the tuned performance of the five runs and shaded areas represent their standard deviation.

to be further updated. Therefore, we also use an exploration rate that allows us to choose a TLA algorithm uniformly at runtime (lines 6–7). The exploration rate is computed dynamically using a function given by:

$$ExplorationRate = \frac{(|T| \cdot n_{parameters}/n_{samples})}{(1 + |T| \cdot n_{parameters}/n_{samples})} \quad (4)$$

where it considers the number of TLA algorithms ($|T|$) (the higher number, the higher exploration rate), the number of tuning parameters ($n_{parameters}$) (the higher number, the higher exploration rate), and the number of obtained samples for the current target task ($n_{samples}$) (the higher number, the lower exploration rate).

There can be more naive ensemble approaches such as (1) a toggling approach that chooses a TLA algorithm sequentially (`Ensemble(toggling)`) and (2) using only a probability distribution function without an exploration rate (i.e., exploration rate is always 0) (`Ensemble(prob)`). We show the benefit of our proposed scheme over these two simpler schemes in Section VI-A.

VI. Evaluation

This section demonstrates the effectiveness of GPTuneCrowd using two synthetic functions and three real-world large-scale applications, NIMROD [4], ScaLAPACK's PDGEQRF [1], SuperLU_DIST [2] and Hypre [3].

A. Comparison of transfer learning algorithms

We first compare the TLA algorithms discussed in Section V, as well as a non-TLA tuner (labeled as `NoTLA`) using two synthetic objective functions. `NoTLA` tunes a given target task using Bayesian optimization from GPTune tuning package, where it builds a GP surrogate model after every function evaluation for parameter search. We use two synthetic functions used in previous literature [8], [22]: (1) an explicit demo objective function and (2) Branin function (https://www.sfu.ca/~ssurjano/branin.html). The demo function is given by $y(t,x) = 1 + e^{(-(x+1)^{t+1})}\cos(2\pi x)\sum_{i=1}^{3}\sin\left(2\pi x(t+2)^i\right)$, where it consists of one task parameter t [0, 10) and one tuning parameter x [0, 1.0). The Branin function is given by $y(a,b,c,r,s,x_1,x_2) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)cos(x_1) + s$, where there are six task parameters (a,b,c,r,s,t) and two tuning parameters ($x1$ and $x2$).

Figure 3 compares different tuning algorithms, based on the best-so-far results depending on the number of samples, for two scenarios of the demo function and four scenarios of the Branin function. Note that, for the first function evaluation (number of evaluations=1) we used `WeightedSum(eqaul)` as there is no information about target task for dynamic weight computation or LCM. The experiments draw the following conclusions.

(1) The TLA algorithms outperform the non-TLA tuner by a significant margin. For the most scenarios in Figure 3, `NoTLA` shows the worst or nearly worst tuning quality compared to TLA algorithms. There are few cases where certain TLA algorithms perform worse than `NoTLA`. For example, `WeightedSum(equal)` becomes worse than `NoTLA` when the number of function evaluations is sufficiently large (e.g., Figure 3 (a)). It implies that the weighted sum-based approach needs an intelligent method to compute the proper weights. When the number of function evaluations is relatively small, `NoTLA` performs much worse than TLA algorithms.

(2) Our improved TLA algorithms outperform the existing algorithms significantly. As explained in Section V, GPTuneCrowd provides `Multitask(TS)` and `WeightedSum(dynamic)` to refine the existing algorithms `Multitask(PS)` and `WeightedSum(equal)`. Comparing `Multitask(TS)` and `Multitask(PS)`, we observe that `Multitask(TS)` improves the overall tuning quality, especially for the Branin function (see (c)–(f)). Comparing `WeightedSum(dynamic)` and `WeightedSum(equal)`, improvements from our dynamic method are clear for all the scenarios except (b). The results show that our dynamic approach provides a good solution to compute weight values for source and target tasks.

TABLE II
PDGEQRF TUNING PARAMETERS

Parameter	Description	Type	Range
mb	row block size $= 8 \cdot mb$	Integer	[1,16)
nb	column block size $= 8 \cdot nb$	Integer	[1,16)
lg2npernode	number of MPI processes per node $= 2^{lg2npernode}$	Integer	[0, $\log_2^{cores}$)
p	number of row processes	Integer	[1, $nodes \cdot cores$)

(3) The best TLA algorithm varies for different tuning problems. The results support our claim "there is no one-size-fits-all TLA algorithm". Comparing among `Multitask(TS)`, `WeightedSum(dynamic)` and `Stacking`, the results show that the best algorithm changes depending on the tuning scenario and the given tuning budget. For example, looking at the tuned result with 10th function evaluations, `Stacking` is the best for (a), (d) and (e), and `Multitask(TS)` is the best for (c). For 20th evaluation, `WeightedSum(dynamic)` is the best for (a) while `Multitask(TS)` is the best for (b). Note that a specific TLA algorithm can often perform poorly in certain cases. For example, in case of `Multitask(TS)`, it provides a good tuning quality for (a), (b), and (e), but it performs poorly for (f). Our experiments using real-world applications (Section VI-C and VI-C) also show that the best TLA algorithm varies depending on the problem. It is not feasible for a user to choose the best TLA algorithm for various tuning problems.

(4) The proposed ensemble technique consistently provides near optimal tuning quality. We observe that the proposed ensemble technique (Algorithm 1) can consistently achieve nearly optimal tuning quality. For example, based on 20 function evaluations, among all the 9 tuners in Figure 3, `Ensemble(proposed)` achieves the best tuning for (a) and nearly best tuning for (c), (d), (e), and (f). For (b) (worst case of `Ensemble(proposed)`), where `Multitask(TS)` is the best (based on 20th results), the ensemble technique is still effective and provides improvements over `NoTLA`, while outperforming weighted-sum and stacking approaches. The figure also shows that the proposed ensemble technique outperforms two simpler ensemble approaches `Ensemble(toggling)` and `Ensemble(prob)` (see (a), (d), (e), and (f)).

B. Case study of transfer learning: ScaLAPACK's PDGEQRF

As a real-world example, ScaLAPACK's PDGEQRF [1] is a distributed memory-parallel QR factorization routine. It contains two task parameters that describe the size of the given matrix rows (m) and columns (n), and four tuning parameters mb, nb, $lg2npernode$, and p (see Table II for the details). We used NERSC's Cori supercomputer. The experiments are performed using eight Cori Haswell compute nodes (total 256 cores), where each node has two 16-core Intel Xeon E5-2698v3 processors and 128GB of 2133MHz DDR4 memory.

Figure 4 shows an evaluation of our TLA algorithms and the non-TLA tuning (`NoTLA`) for two transfer learning scenarios. Figure 4 (a) uses one source task and (b) uses three source tasks. For the source datasets, we collected 100 samples for each source task for randomly chosen parameter configurations. In a typical tuning setting, the user can start with a collection of randomly chosen parameters to sufficiently

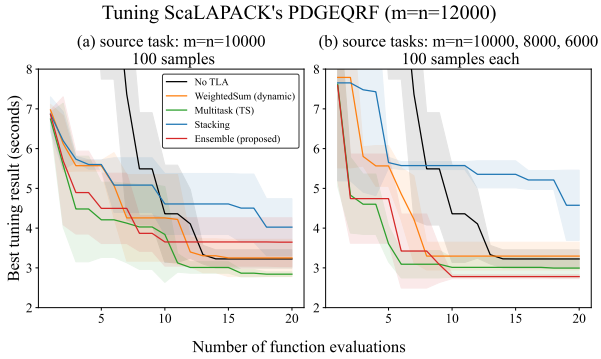

Fig. 4. Tuned performance of ScaLAPACK's PDGEQRF on 8 Haswell nodes in NERSC's Cori (256 cores). (a) uses one source task (m=n=10000), and (b) uses three source tasks (m=n=10000,8000,6000); each source task has 100 samples for randomly chosen parameter configurations. We run each tuning option three times (with different random seeds). The line charts show the average of the tuned performance of the three runs and shaded areas represent their standard deviation.

cover the search space, before the Bayesian optimization step starts [8]. Therefore, the source task can contain many random samples. Also, it is a reasonable setup to evaluate multiple TLA algorithms in a reproducible way. Note that as more and more samples are collected for the source task(s), transfer learning would generally perform better, regardless of how the samples are drawn.

In the figure, as expected, compared to `NoTLA` the TLA algorithms improves the tuned performance significantly, especially when the number of function evaluations is small. Looking at 10th evaluation on the single-source scenario, for example, on (a) `Ensemble(proposed)` achieves the best tuned performance (3.65s). Compared to the performance (4.36s) tuned by `NoTLA`, our ensemble-based TLA achieves 1.19x speedup (16.3% improvement). On the three-sources scenario, `Ensemble(proposed)` achieves 2.78s which has a speedup of 1.57x (36.2% improvement), compared to `NoTLA` (4.36s).

Among the TLA algorithms, we see that `Stacking` approach is not effective for this problem. We also observe that `Multitask(TS)` exploits the three available datasets well, leading to better tuned results for 1st–10th evaluations in (b) compared to (a). The proposed ensemble technique consistently provides a good tuning quality for both (a) and (b).

C. Case study of transfer learning: NIMROD

For a larger-scale experiment, we used an extended magnetohydrodynamics (MHD) code NIMROD [4]. NIMROD is primarily used for calculating the equilibrium, stability, and dynamics of fusion plasmas, and is a critical simulation tool for designing reactor-scale tokamaks. NIMROD uses a high-order finite-element discretization for the poloidal plane, and a pseudo-spectral method for the toroidal direction. The MHD equations are solved via time-marching where each time step solves several nonsymmetric sparse linear systems with block Jacobi preconditioned GMRES [4]. Each Jacobi block is factorized with the 3D algorithm of SuperLU_DIST [23]. In our experiment, we fix the geometry model and the number of time steps. NIMROD production simulations

TABLE III
NIMROD TUNING PARAMETERS

Parameter	Description	Type	Range
NSUP	Maximum supernode sizes in SuperLU	Integer	[30,300)
NREL	Upper bound of the minimum supernode sizes in SuperLU	Integer	[10,40)
nbx	2^{nbx} represents blocking parameter in x direction for assembling NIMROD matrices	Integer	[1,3)
nby	2^{nby} represents blocking parameter in y direction for assembling NIMROD matrices	Integer	[1,3)
npz	2^{npz} represents number of processes in z dimension of each SuperLU 3D process grid	Integer	[0,5)

require DOE leadership machines and are computationally expensive. Using an optimized tuning method is critical to achieving best performance as problem specifications and computational architectures change.

More specifically, we set the number of time steps to 30 and aim to minimize the runtime in the main time-marching loop. We consider the following task parameters mx, my, and $lphi$ that determine the size of the problem, where 2^{mx} and 2^{my} represent the number of mesh DoF in x and y directions, respectively. $floor(2^{lphi}/3)+1$ represents number of Fourier modes in the toroidal direction. The tuning parameters are = [NSUP, NREL, nbx, nby, npz], and the descriptions are detailed in Table III. NSUP, NREL and npz are tuning parameters related to running 3D sparse LU factorization algorithm of SuperLU_DIST, and nbx and nby are NIMROD's blocking parameters. For each task, we also fix the total number of compute nodes. Note that depending on the task and tuning parameters there can be idle MPI ranks. For the following experiments, as a source task's dataset, we use 500 samples collected from 32 Haswell nodes in Cori, for {mx: 5, my: 7, lphi: 1}. As clarified in Section VI-B, we randomly chose parameter configurations to collect initial 500 source samples.

(1) TLA across different node counts (Figure 5 (a)). In the first scenario, we tune NIMROD on a different number of compute nodes (i.e., 64 Haswell nodes) than the source task. We compare our TLA features with the non-TLA approach `NoTLA`. First, the TLA algorithms can provide much better tuning results compared to `NoTLA` by learning knowledge from different node resource allocations. Looking at the 10th tuning results, for example, compared to `NoTLA` the tuned performance can be improved by 16.6% (1.20x) and 13.8% (1.16x) (reduction of the runtime), using `Multitask(TS)` (performed the best) and `Ensemble(proposed)`, respectively.

(2) TLA across completely different hardware architectures and different problem sizes (Figure 5 (b)). Then, we consider a different hardware architecture with a different number of nodes. We use 32-node Knights Landing (KNL) nodes in NERSC's Cori machine. Each KNL node includes an Intel Xeon Phi processor 7250 with a memory size of 96GB (DDR4) and 16GB (MCDRAM). In Figure 5, we see that transfer learning algorithms perform similarly with non-TLA tuning, although the first few iterations from transfer

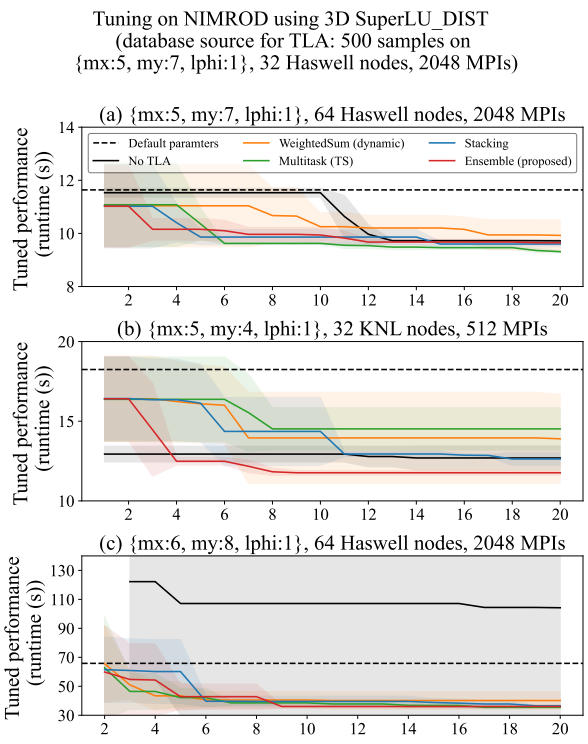

Fig. 5. Tuned performance of NIMROD. (a)–(c) use one source task data, {mx:5, my:7, lphi:1}, 500 samples obtained from 32 Haswell nodes in NERSC Cori (1024 cores) for randomly chosen parameter configurations. For the target task, (a) uses 64 Haswell nodes (2048 cores) for {mx:5, my:7, lphi:1}. (b) uses 32 KNL nodes (2176 cores) for {mx:5, my:4, lphi:1}. (c) uses 64 Haswell nodes (2048 cores) for {mx:6, my:8, lphi:1}. We run each tuning option three times (with different random seeds). The line charts show the average of the tuned performance of the three runs and shaded areas represent their standard deviation.

learning had worse performance than `NoTLA`. Looking at the 10th results, compared to non-TLA, `Ensemble(proposed)` still has 1.1x (9.0%) better tuned performance. In usual scenarios, users may choose a similar hardware architecture and problem size for the source task. However, the results show that even on a completely different hardware architecture, our transfer learning would eventually behave similar to `NoTLA`, as our transfer learning algorithms can automatically learn the correlation between source tasks (other machines) and the target task (the user's machine).

(3) TLA across different problem sizes and different node counts (Figure 5 (c)). We also consider tuning on a different problem size. In this experiment, for the target tuning problem we increase the input task {mx:6, my:8, lphi:1}. We also increased the number of Haswell nodes to 64, so that we can have more hardware resources to execute the large problem size. Our experimental results in Figure 5 (c) show a dramatic performance improvement over `NoTLA`. Moreover, `Ensemble(proposed)` performs the best among the tested TLA algorithms. Looking at the 10th tuning results, compared to `NoTLA`, `Ensemble(proposed)` (performed the best) achieves 66.4% performance gain (2.97x speedup) and `Multitask(TS)` (second best) achieves 64% improvement (2.78x speedup). Note that, in Figure 5 (c), X-axis (number of

TABLE IV
SENSITIVITY ANALYSIS OF SUPERLU_DIST (INPUT MATRIX: SI5H12)

Parameter	Type	Range	S1	S1.conf	ST	ST.conf
COLPERM	Categorical	5 choices	0.80	0.06	0.86	0.07
LOOKAHEAD	Integer	[5,20]	0.00	0.01	0.01	0.00
nprows	Integer	[1,11]	0.11	0.05	0.17	0.03
NSUP	Integer	[30,300]	0.01	0.02	0.06	0.02
NREL	Integer	[10,40]	0.00	0.01	0.01	0.00

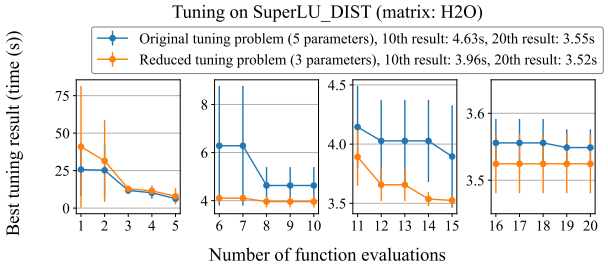

Fig. 6. Benefit of reduced tuning on SuperLU_DIST. For the reduced tuning problem, we use the default parameter values for the LOOKAHEAD and NREL, without tuning. The figure plots the mean of the best-so-far results of three runs of each tuner along with their standard deviation (in vertical lines).

function evaluations) can start from 2 or 3 depending on the tuner, because we do not draw points if the three tuning runs had any failures (e.g., bad parameter configuration(s) that result in an out-of-memory error). Such failures are disregarded when fitting a surrogate model in our tuner settings. Having such bad parameter configurations led to a poor tuning result, especially in `NoTLA`.

It is also worth mentioning that the source task in the NIMROD experiments used a previous version of CrayMPICH (7.7.10) then the target tasks (7.7.19) due to a software update on the Cori supercomputer. This shows that our transfer learning can also learn from different software versions.

D. Case study of sensitivity analysis: SuperLU_DIST

To show the effectiveness of the sensitivity analysis, we considered the 2D version of SuperLU_DIST [2], a distributed-memory sparse direct solver for non-symmetric linear systems. SuperLU_DIST uses a supernodal representation of the LU factors with nonuniform supernode sizes with a 2D block-cyclic process layout. The 2D SuperLU_DIST has the following tuning parameters: [COLPERM, LOOKAHEAD, nprows, NSUP, NREL]. Table IV shows an output of running the sensitivity analysis of SuperLU_DIST for an input matrix Si5H12 [24] using 500 samples collected on four Cori Haswell nodes. On the S1 and the ST values, the analysis shows that the COLPERM parameter has the highest influence, then the next important parameter is nprows. The other three parameters have less influence on the tuning results, however, NSUP has a moderate influence on the performance.

The sensitivity analysis quantifies the sensitivity to the output of each variable. If we need to tune SuperLU_DIST on a different hardware, resource allocation, or a different input matrix of similar sparsity patterns, we can reduce the number of tuning parameters, based on this numerical analysis and the given tuning budget. In the following experiment, we reduce the tuning problem to tune only COLPERM, nprows, and NSUP, those have high or moderate S1 and ST

TABLE V
SENSITIVITY ANALYSIS OF HYPRE (INPUT TASK: NX=NY=NZ=100).

Parameter	Type	Range	S1	S1.conf	ST	ST.conf
Px	Integer	[1,32)	0.00	0.01	0.01	0.00
Py	Integer	[1,32)	0.03	0.05	0.35	0.12
Nproc	Integer	[1,32)	0.01	0.04	0.23	0.07
strong_threshold	Real	[0,1)	0.00	0.00	0.00	0.00
trunc_factor	Real	[0,1)	0.01	0.01	0.03	0.01
P_max_elmts	Integer	[1,12)	0.00	0.00	0.00	0.00
coarsen_type	Categorical	8 choices	0.00	0.00	0.00	0.00
relax_type	Categorical	6 choices	0.00	0.00	0.00	0.00
smooth_type	Categorical	5 choices	0.11	0.07	0.71	0.21
smooth_num_levels	Integer	[0,5)	0.05	0.05	0.35	0.13
interp_type	Categorical	7 choices	0.00	0.00	0.00	0.00
agg_num_levels	Integer	[0,5)	0.11	0.11	0.56	0.14

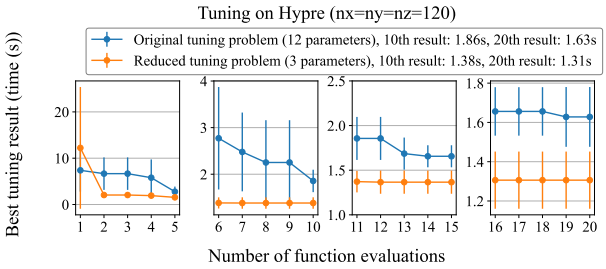

Fig. 7. Benefit of reduced tuning on Hypre (IJ). For the reduced tuning problem, we tune three most sensitive parameters, smooth_type, smooth_num_levels, and agg_num_levels, while setting default parameter values for strong_threshold, trunc_factor, P_max_elmts, coarsen_type, relax_type (we know the default parameter values) and random values for Px, Py, and Nproc (we do not know the default values). The figure plots the mean of the best-so-far results of five runs of each tuner along with their standard deviation (in vertical lines).

values. Figure 6 shows how a reduced tuning problem can be useful to achieve a good tuning result using a smaller tuning budget. Here, we tune a different matrix H2O on four Haswell nodes, and compare the tuning performance between the original tuning problem and the reduced tuning problem that deactivates tuning on the LOOKAHEAD and NREL parameters. The matrices H2O and Si5H12 (used in the sensitivity analysis) are from the PARSEC group of the SuiteSparse Matrix Collection [24] and thus have a similar sparsity pattern. The results show that, while the tuning on the original tuning problem has a better initial guess (due to the randomness), the reduced tuning problem can provide a better tuning result with a moderate number of function evaluations. For example, on the 10th tuning results, the reduced tuning problem can attain 1.17x better tuned result (14.5% improvement) compared to the original search space.

E. Case study of sensitivity analysis: Hypre

Hypre [3] is a parallel algebraic multigrid solver for sparse linear systems. Here, we focus on tuning of GMRES with the BoomerAMG preconditioner that solves the Poisson equation on structured 3-d grids of [nx, ny, nz]. The tuning problem contains a large number of tuning parameters (12 parameters), therefore, it will require a huge number of function evaluations to attain a near-optimal tuning parameter configuration. We use sensitivity analysis to reduce the search space of Hypre.

Table V describes the tuning parameters and shows their Sobol sensitivity scores. For the analysis, we use 1,000 samples pre-collected on a Cori Haswell node for

nx=ny=nz=100. The analysis shows that smooth_type and agg_num_levels have high S1 and ST scores (S1>0.1, ST>0.5), followed by smoth_num_levels (S1=0.05, ST=0.35), Py (S1=0.03, ST=0.35), and Nproc (S1=0.01, ST=0.23). Other parameters (Px, strong_threshold, trunc_factor, P_max_elmts, coarsen_type, relax_type, and interp_type) have low S1/ST values close to 0 (lower than 0.05).

In Figure 7, we tune Hypre with a tuning budget of 20 function evaluations. Due to the limited tuning budget, we chose the three most sensitive parameters, smooth_type and agg_num_levels, for tuning. The figure shows that we achieve higher tuning quality on the reduced search space, compared to tuning on the original search space. Comparing the 10th tuning results, the reduced tuning achieves 1.35x better result (25.8% improvement), compared to the original search space.

VII. Conclusion

In this paper, we presented a crowd-based autotuning framework called GPTuneCrowd. GPTuneCrowd leverages a shared autotuning database with a user-friendly tuner interface that allows users to query relevant data using a simple meta description. GPTuneCrowd improves existing transfer learning algorithms and presents an ensemble technique to benefit from multiple transfer learning algorithms to maximize tuning quality. We demonstrated the effectiveness of GPTuneCrowd's features using synthetic functions and large-scale real-world tuning problems. In future work, we plan to deploy GPTuneCrowd to tune other expensive tuning problems in HPC. Detecting/diagnosing performance variability of performance samples (caused by system noise) is also our future work.

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] 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

[2] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, p. 110–140, Jun. 2003. [Online]. Available: https://doi.org/10.1145/779359.779361

[3] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *Computational Science — ICCS 2002*. P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 632–641.

[4] C. Sovinec, A. Glasser, T. Gianakon, D. Barnes, R. Nebel, S. Kruger, S. Plimpton, A. Tarditi, M. Chu, and the NIMROD Team, "Nonlinear magnetohydrodynamics with high-order finite elements," *J. Comp. Phys.*, vol. 195, p. 355, 2004.

[5] P. Balaprakash, R. Egele, and P. Hovland, "ytopt: Machine-learning-based search methods for autotuning," 2019. [Online]. Available: https://github.com/ytopt-team/ytopt

[6] 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.

[7] 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

[8] 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*. [Online]. Available: https://doi.org/10.1145/3437801.3441621

[9] 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.

[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] Y. Cho, J. W. Demmel, X. S. Li, Y. Liu, and H. Luo, "Enhancing autotuning capability with a history database," in *The IEEE 14th International Symposium on Embedded Multicore/Manycore SoCs (MCSoC-2021)*, vol. 20, 2021.

[12] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google Vizier: A Service for Black-Box Optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. [Online]. Available: https://doi.org/10.1145/3097983.3098043

[13] X. Zhu, Y. Liu, P. Ghysels, D. Bindel, and X. S. Li, *GPTuneBand: Multi-task and Multi-fidelity Autotuning for Large-scale High Performance Computing Applications*, pp. 1–13. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611977141.1

[14] H. Luo, J. W. Demmel, Y. Cho, X. S. Li, and Y. Liu, "Non-smooth bayesian optimization in tuning problems," *arXiv preprint arXiv:2109.07563*, 2021.

[15] H. Luo, Y. Cho, J. W. Demmel, X. S. Li, and Y. Liu, "Hybrid models for mixed variables in bayesian optimization," *arXiv preprint arXiv:2206.01409*, 2022.

[16] Y. Cho, J. W. Demmel, G. Dinh, H. Luo, X. S. Li, Y. Liu, O. Marques, and W. M. Sid-Lakhdar. "GPTune user guide," 2022. [Online]. Available: https://gptune.lbl.gov/documentation/gptune-user-guide/

[17] 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

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

[19] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. [Online]. Available: https://doi.org/10.1145/2807591.2807623

[20] I. M. Sobol, "Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates," *Mathematics and computers in simulation*, pp. 271–280, 2001.

[21] J. Herman and W. Usher, "SALib: An open-source Python library for sensitivity analysis," *The Journal of Open Source Software*, vol. 2, no. 9, jan 2017. [Online]. Available: https://doi.org/10.21105/joss.00097

[22] P. Tighineanu, K. Skubch, P. Baireuther, A. Reiss, F. Berkenkamp, and J. Vinogradska, "Transfer learning with gaussian processes for bayesian optimization," in *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics* [Online]. Available: https://proceedings.mlr.press/v151/tighineanu22a.html

[23] P. Sao, X. S. Li, and R. Vuduc, "A communication-avoiding 3d lu factorization algorithm for sparse matrices," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663